

pyformex

pyFormex Documentation

Release 1.0.7

Benedict Verhegghe

Jun 17, 2019

CONTENTS

1	Introduction to pyFormex	1
2	Installing pyFormex	5
3	pyFormex tutorial	7
4	pyFormex user guide	31
5	pyFormex example scripts	59
6	pyFormex reference manual	73
7	pyFormex FAQ ‘n TRICKS	581
8	pyFormex file formats	587
9	BuMPix Live GNU/Linux system	591
10	GNU GENERAL PUBLIC LICENSE	597
11	About the pyFormex documentation	607
12	Glossary	609
	Python Module Index	611
	Index	613

INTRODUCTION TO PYFORMEX

Abstract

This part explains shortly what pyFormex is and what it is not. It sets the conditions under which you are allowed to use, modify and distribute the program. Next is a list of prerequisite software parts that you need to have installed in order to be able to run this program. We explain how to download and install pyFormex. Finally, you'll find out what basic knowledge you should have in order to understand the tutorial and successfully use pyFormex.

1.1 What is pyFormex?

You probably expect to find here a short definition of what pyFormex is and what it can do for you. I may have to disappoint you: describing the essence of pyFormex in a few lines is not easy to do, because the program can be (and is being) used for very different tasks. So I will give you two answers here: a short one and a long one.

The short answer is that pyFormex is a program to *generate large structured sets of coordinates by means of subsequent mathematical transformations gathered in a script*. If you find this definition too dull, incomprehensible or just not descriptive enough, read on through this section and look at some of the examples in this documentation and on the [pyFormex website](#). You will then probably have a better idea of what pyFormex is.

The initial intent of pyFormex was the rapid design of three-dimensional structures with a geometry that can easier be obtained through mathematical description than through interactive generation of its subparts and assemblage thereof. Although the initial development of the program concentrated mostly on wireframe type structures, surface and solid elements have been part of pyFormex right from the beginning. There is already an extensive plugin for working with triangulated surfaces, and pyFormex is increasingly being used to generate solid meshes of structures. Still, many of the examples included with the pyFormex distribution are of wireframe type, and so are most of the examples in the [pyFormex tutorial](#).

A good illustration of what pyFormex can do and what it was intended for is the stent¹ structure in the figure [WireStent example](#). It is one of the many examples provided with pyFormex.

The structure is composed of 22032 line segments, each defined by 2 points. Nobody in his right mind would ever even try to input all the 132192 coordinates of all the points describing that structure. With pyFormex, one could define the structure by the following sequence of operations, illustrated in the figure [First three steps in building the WireStent example](#):

1. Create a nearly planar base module of two crossing wires. The wires have a slight out-of-plane bend, to enable the crossing.
2. Extend the base module with a mirrored and translated copy.

¹ A stent is a tubular structure that is e.g. used to reopen (and keep open) obstructed blood vessels.

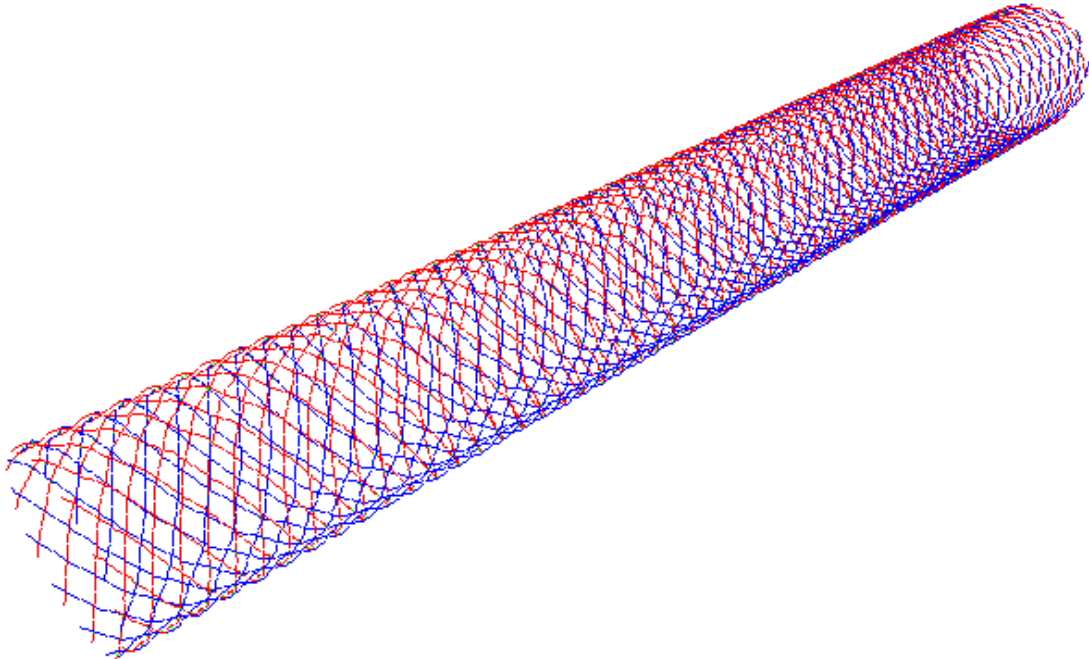


Fig. 1: WireStent example

3. Replicate the base module in both directions to create a (nearly planar) rectangular grid.
4. Roll the planar grid into a cylinder.

pyFormex provides all the operations needed to define the geometry in this way.

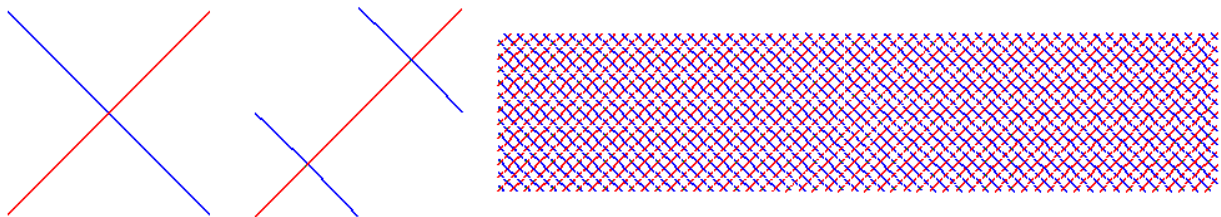


Fig. 2: First three steps in building the WireStent example

pyFormex does not fit into a single category of traditional (mostly commercial) software packages, because it is not being developed as a program with a specific purpose, but rather as a collection of tools and scripts which we needed at some point in our research projects. Many of the tasks for which we now use pyFormex could be done also with some other software package, like a CAD program or a matrix calculation package or a solid modeler/renderer or a finite element pre- and postprocessor. Each of these is probably very well suited for the task it was designed for, but none provides all the features of pyFormex in a single consistent environment, and certainly not as free software.

Perhaps the most important feature of pyFormex is that it was primarily intended to be an easy scripting language for creating geometrical models of 3D-structures. The graphical user interface (GUI) was only added as a convenient means to visualize the designed structure. pyFormex can still run without user interface, and this makes it ideal for use in a batch toolchain. Anybody involved in the simulation of the mechanical behavior of materials and structures will testify that most of the work (often 80-90%) goes into the building of the model, not into the simulations itself. Repeatedly building a model for optimization of your structure quickly becomes cumbersome, unless you use a tool

like pyFormex, allowing for automated and unattended building of model variants.

The author of pyFormex, professor in structural engineering and heavy computer user and programmer since main-frame times, deeply regrets that computing skills of nowadays engineering students are often limited to using graphical interfaces of mostly commercial packages. This greatly limits their skills, because in their way of thinking: ‘If there is no menu item to do some task, then it can not be done!’ The hope to get some of them back into coding has been a stimulus in continuing our work on pyFormex. The strength of the scripting language and the elegance of Python have already attracted many users on this path.

Finally, pyFormex is, and always will be, free software in both meanings of free: guaranteeing the freedom of the user (see *License and Disclaimer*) and without charging a fee for it.²

1.2 License and Disclaimer

pyFormex is ©2004-2019 Benedict Verhegghe

This program is free software; you can redistribute it and/or modify it under the terms of the [GNU General Public License](#) (GNU GPL), as published by the [Free Software Foundation](#); either version 3 of the License, or (at your option) any later version.

The full details of the [GNU GPL](#) are available in the [GNU GENERAL PUBLIC LICENSE](#) part of the documentation, in the file COPYING included with the distribution, under the Help->License item of the pyFormex Graphical User Interface or from <http://www.gnu.org/copyleft/gpl.html>.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

1.3 Installation

Information on how to obtain and install pyFormex can be found in the *Installing pyFormex* document.

1.4 Using pyFormex

Once you have installed and want to start using pyFormex, you will probably be looking for help on how to do it.

If you are new to pyFormex, you should start with the *pyFormex tutorial*, which will guide you step by step, using short examples, through the basic concepts of Python, NumPy and pyFormex. You have to understand there is a lot to learn at first, but afterwards the rewards will prove to be huge. You can skip the sections on Python and NumPy if you already have some experience with it.

If you have used pyFormex before or you are of the adventurous type that does not want to be told where to go and how to do it, skip the tutorial and go directly to the *pyFormex user guide*. It provides the most thorough information of all aspects of pyFormex.

1.5 Getting Help

If you get stuck somewhere with using (or installing) pyFormex and you need help, the best way is to go to the [pyFormex website](#) and ask for help via the [Support tracker](#). There’s a good chance you will get helped quickly there.

² Third parties may offer pyFormex extensions and/or professional support that are fee-based.

Remember though that pyFormex is a free and open source software project and its developers are not paid to develop or maintain pyFormex, they just do this because they find pyFormex very helpful in their normal daily activities.

If you are a professional pyFormex user and require guaranteed support, you can check with [FEops](#), a young company providing services with and support for pyFormex.²

INSTALLING PYFORMEX

Warning: This document is under construction

Abstract

This document explains the different ways for obtaining a running pyFormex installation. You will learn how to obtain pyFormex, how to install it, and how to get it running.

2.1 Installing pyFormex-1.0.x

This installation manual is for the pyFormex 1.0 series. Installation instructions for pyFormex-0.9 can be found [here](#). pyFormex is being developed on GNU/Linux systems, and currently only runs on Linux. On other systems you have the option of running Linux in a virtual machine, or you can boot your machine from a USB stick with a Linux Live system.

As there has not been an official release yet of pyFormex-1.0, this manual only deals with how to run pyFormex from source, and how to install one of the alpha releases.

Whether you are running pyFormex from source or from a packaged release, there are some required packages that you need to have on your Linux system. pyFormex will not run without them. Then, there are also some recommended packages: these are only necessary for some applications, but not for basic pyFormex.

2.2 Dependencies

In order to run pyFormex, you need at least the following installed (and working) on your computer:

Python (<http://www.python.org>) Version 2.7 is required. Lower versions are no longer supported. Development is ongoing to make pyFormex run on Python 3.x, but is not ready yet. Nearly all Linux distributions come with Python 2.7 installed by default (or as an option), so this should be no problem.

FreeType (<https://www.freetype.org/>) **Used for rendering text.** All Linux distributions come with FreeType installed, so again here's no problem.

NumPy (<http://www.numpy.org>) Version 1.0 or higher. NumPy is the package used for efficient numerical array operations in Python and is essential for pyFormex.

PIL (<https://python-pillow.org/>) The Python Imagaging Library. The original PIL project is no longer maintained, but the pillow fork fits in nicely.

Qt4 (<http://www.trolltech.com/products/qt>) The widget toolkit on which the pyFormex Graphical User Interface (GUI) was built.

PySide (<http://www.pyside.org>) The Python bindings for Qt4. There is an alternative set of Python bindings for Qt4: PyQt4 (<http://www.riverbankcomputing.co.uk/pyqt/index.php>), and you can use this instead. pyFormex will detect the installed one. If you have both, you can select either.

PyOpenGL (<http://pyopengl.sourceforge.net/>) Python bindings for OpenGL, used for drawing and manipulating the 3D-structures.

If you only want to use the Formex data model and transformation methods and do not need the GUI, then NumPy is all you need. This could e.g. suffice for a non-interactive machine that only does the numerical processing. The install procedure however does not provide this option yet, so you will have to do the install by hand. Currently we recommend to install the whole package including the GUI. Most probably you will want to visualize your structures and for that you need the GUI anyway.

In order to speed up some time-critical tasks, pyFormex has an acceleration library containing some compiled C functions. In order to install and compile these, you will need a C development environment and the required header files:

- GNU make
- gcc: GNU C compiler
- header files for Python and OpenGL (e.g. from mesa)

We highly recommend that you install these to allow the acceleration library to be compiled.

2.2.1 Installing dependencies on *Debian GNU/Linux*

Debian/Ubuntu users can install the pyFormex dependencies from their distribution's repositories. Installing the following packages should suffice: `python-numpy`, `python-pil`, `python-opengl`, `python-qt4`, `python-pyside`.

Also, for compiling the acceleration library, you should install `python-dev` and `libglul-mesa-dev`. This command will do it all:

```
apt-get install python-numpy python-opengl python-qt4 python-pyside python-qt4-gl_
↳python-dev libglul-mesa-dev libfreetype6-dev
```

2.3 Recommends

Additionally, we recommend you to also install the Python and OpenGL header files. The install procedure needs these to compile the pyFormex acceleration library. While pyFormex can run without the library (Python versions will be substituted for all functions in the library), using the library will dramatically speed up some low level operations such as drawing, especially when working with large structures .

Other optional packages that might be useful are `admesh`, `python-scipy`, `units`, `python-vtk`.

PYFORMEX TUTORIAL

Abstract

This tutorial will guide you step by step through the most important concepts of the pyFormex scripting language and the pyFormex Graphical User Interface (GUI). It is intended for first time users, giving explicit details of what to do and what to expect as result.

3.1 The philosophy

pyFormex is a Python implementation of Formex algebra. Using pyFormex, it is very easy to generate large geometrical models of 3D structures by a sequence of mathematical transformations. It is especially suited for the automated design of spatial structures. But it can also be used for other tasks, like operating on 3D geometry obtained from other sources, or for finite element pre- and postprocessing, or just for creating some nice pictures.

By writing a simple script, a large and complex geometry can be created by copying, translating, rotating, or otherwise transforming geometrical entities. pyFormex will interpret the script and draw what you have created. This is clearly very different from the traditional (mostly interactive) way of creating a geometrical model, like is done in most CAD packages. There are some huge advantages in using pyFormex:

- It is especially suited for the automated design of spatial frame structures. A dome, an arc, a hyper shell, . . . , when constructed as a space frame, can be rather difficult and tedious to draw with a general CAD program; using scripted mathematical transformations however, it may become a trivial task.
- Using a script makes it very easy to apply changes in the geometry: you simply modify the script and re-execute it. You can easily change the value of a geometrical parameter in any way you want: set it directly, interactively ask it from the user, calculate it from some formula, read it from a file, etcetera. Using CAD, you would have often have to completely redo your drawing work. The power of scripted geometry building is illustrated in figure *Same script, different domes*: all these domes were created with the same script, but with different values of some parameters.
- At times there will be operations that are easier to perform through an interactive Graphical User Interface (GUI). The GUI gives access to many such functions. Especially occasional and untrained users will benefit from it. As everything else in pyFormex, the GUI is completely open and can be modified at will by the user's application scripts, to provide an interface with either extended or restricted functionality.
- pyformex scripts are written in the **Python** programming language. This implies that the scripts are also Python-based. It is a very easy language to learn, and if you are interested in reading more about it, there are good tutorials and beginner's guides available on the Python website (<http://www.python.org/doc>). However, if you're only using Python to write pyFormex scripts, the tutorial you're reading right now should be enough.

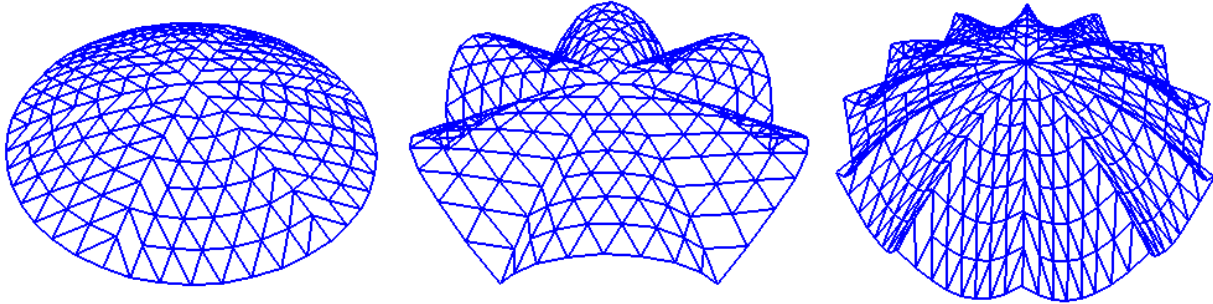


Fig. 1: Same script, different domes

3.2 Getting started

- Start the pyFormex GUI by entering the command `pyformex` in a terminal. Depending on your installation, there may also be a menu item in the application menu to start pyFormex, or even a quickstart button in the panel. Using the terminal however can still be useful, especially in the case of errors, because otherwise the GUI might suppress some of the error messages that normally are sent to the terminal.
- Create a new pyFormex script using the *File*→*Create new script* option. This will open a file dialog: enter a filename `example0.py` (be sure to be in a directory where you have write permissions). Pressing the Save button will open up your favorite editor with a pyFormex script template like the one below.

```

1  #
2  ##
3  ##  Copyright (C) 2011 John Doe (j.doe@somewhere.org)
4  ##  Distributed under the GNU General Public License version 3 or later.
5  ##
6  ##  This program is free software: you can redistribute it and/or modify
7  ##  it under the terms of the GNU General Public License as published by
8  ##  the Free Software Foundation, either version 3 of the License, or
9  ##  (at your option) any later version.
10 ##
11 ##  This program is distributed in the hope that it will be useful,
12 ##  but WITHOUT ANY WARRANTY; without even the implied warranty of
13 ##  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
14 ##  GNU General Public License for more details.
15 ##
16 ##  You should have received a copy of the GNU General Public License
17 ##  along with this program. If not, see http://www.gnu.org/licenses/.
18 ##
19
20 """pyFormex Script/App Template
21
22 This is a template file to show the general layout of a pyFormex
23 script or app.
24
25 A pyFormex script is just any simple Python source code file with
26 extension '.py' and is fully read and execution at once.
27
28 A pyFormex app can be a '.py' or '.pyc' file, and should define a function
29 'run()' to be executed by pyFormex. Also, the app should import anything that
30 it needs.
31

```

(continues on next page)

(continued from previous page)

```

32 This template is a common structure that allows the file to be used both as
33 a script or as an app, with almost identical behavior.
34
35 For more details, see the user guide under the `Scripting` section.
36
37 The script starts by preference with a docstring (like this),
38 composed of a short first line, then a blank line and
39 one or more lines explaining the intention of the script.
40
41 If you distribute your script/app, you should set the copyright holder
42 at the start of the file and make sure that you (the copyright holder) has
43 the intention/right to distribute the software under the specified
44 copyright license (GPL3 or later).
45 """
46 # This helps in getting some code working with both Python2 and Python3
47 from __future__ import absolute_import, division, print_function
48
49 # The pyFormex modeling language is defined by everything in
50 # the gui.draw module (if you use the GUI). For execution without
51 # the GUI, you should import from pyformex.script instead.
52 from pyformex.gui.draw import *
53
54 # Definitions
55 def run():
56     """Main function.
57
58     This is automatically executed on each run of an app.
59     """
60     print("This is the pyFormex template script/app")
61
62
63 # Code in the outer scope:
64 # - for an app, this is only executed on loading (module initialization).
65 # - for a script, this is executed on each run.
66
67 print("This is the initialization code of the pyFormex template script/app")
68
69 # The following is to make script and app behavior alike
70 # When executing a script in GUI mode, the global variable __name__ is set
71 # to 'draw', thus the run method defined above will be executed.
72
73 if __name__ == '__draw__':
74     print("Running as a script")
75     run()
76
77
78 # End

```

Note: If the editor does not open, you may need to configure the editor command: see *Settings* → *Commands*.

Make sure you are using an editor that understands Python code. Most modern editors will give you syntax highlighting and help with indentation.


- The template script shows the typical layout of a pyFormex script:
 - The script starts with some comment lines (all lines starting with a '#'). For the sake of this tutorial, you

can just disregard the comments. But this section typically displays a file identification, the copyright notice and the license conditions.

- Then comes a multiline documentation string, contained between two `"""` delimiters. By preference, this docstring is composed of a short first line, then a blank line and finally one or more lines explaining the intention of the script.
 - Next are the pyFormex instructions.
 - The script ends with a comment line `# End`. We recommend you to do this also. It serves as a warning for inadvertent truncation of your file.
- In the status bar at the bottom of the pyFormex GUI, you will now see the name of the script, together with a green dot. This tells you that the script has been recognized by the system as a pyFormex script, and is ready to run.
 - Read the docstring of the template script: it gives some basic information on the two application models in pyFormex. For this tutorial we will however stick to the simpler *script* model. Therefore, replace the whole code section between the `from __future__` line and `# End` with just this single line:

```
print("This is a pyFormex script")
```

Note: The `from __future__ import print_function` line makes Python import a feature from the future Python3 language, turning the `print` statement into a function. This means that you have to write `print(something)` instead of `print something`. If you are acquainted with Python and it hinders you, remove that line (but remember that you will have to learn the newer syntax sooner or later). If you are a starting Python user, leave it there and learn to use the future syntax right from the start.

- Save your changes to the script (in your editor), and execute it in pyFormex by selecting the *File* → *Play* menu option, or by just pushing the  button in the toolbar. In the message area (just above the bottom status bar), a line is printed announcing the start and end of execution. Any output created by the script during execution is displayed in between this two lines. As expected, the template script just prints the text from the `print` statement.
 - Now change the text of the string in the `print` statement, but do not save your changes yet. Execute the script again, and notice that the printed text has not changed! This is because the editor is an external program to pyFormex, and *the executed script is always the text as read from file*, not necessarily equal to what is displayed in your editor.
- Save the script, run it again, and you will see the output has changed.
- Next, change the text of the script to look like the one below, and save it as `example1.py`. Again, note that the editor and pyFormex are separate programs, and saving the script does not change the name of the current script in pyFormex.

```
1 # example1.py
2
3 """Example 1"""
4
5 F = Formex([[ [0.,0.], [1.,0.] ], [ [1.,1.], [0.,1.] ]])
6
7 # End
```

Selecting an existing script file for execution in pyFormex is done with the *File* → *Open* option. Open the `example1.py` file you just saved and check that its name is indeed displayed in the status bar. You can now execute the script if you want, but it will not produce anything visible. We'll learn you how to visualize geometry later on.

- Exit pyFormex (using the *File* → *Exit*) and then restart it. You should again see the `example1.py` displayed as the current script. On exit, pyFormex stores your last script name, and on restart it prepares to run it again. You can also easily select one of the most recent scripts you used from the *File* → *History* option. Select the oldest (bottom) one. Then close all your editor windows.
- Open the `example1.py` again, either using *File* → *Open* or *File* → *History*. The script will not be loaded into your editor. That is because often you will just want to *run* the script, not *change* it. Use the *File* → *Edit* option to load the current script into the editor.

Now that you know how to load, change and execute scripts in pyFormex, we're all set for exploring its power. But first, let's introduce you to some basic Python and NumPy concepts. If you are already familiar with them, you can just skip these sections.

3.3 Some basic Python concepts

pyFormex is written in the Python language, and Python is also the scripting language used by pyFormex. Since the whole intent of pyFormex is to generate geometrical structures from scripts, you will at least need to have some basic knowledge of Python before you can use it for your own projects.

The [Python documentation](#) website contains a variety of good documents to introduce you. If you are new to Python, but have already some programming experience, the [Python tutorial](#) may be a good starting point. Or else, you can take a look at one of the other beginners' guides. Stick with the Python 2.x documentation for now. Though pyFormex might one day use Python 3.x, we are still far off that day, because all the underlying packages need to be converted to Python 3 first.

Do not be afraid of having to learn a new programming language. Python is known as one of the easiest languages to get started with: just a few basic concepts suffice to produce quite powerful scripts. Most developers and users of pyFormex have started without any knowledge of Python.

For the really impatient who do not want to go through the [Python tutorial](#) before diving into pyFormex, we have gathered hereafter some of the most important Python concepts, hopefully enabling you to continue with this tutorial.

Here is a small example Python script.

```

1  #!/usr/bin/env python
2  """Python intro
3
4  A short introduction to some aspects of the Python programming language
5  """
6  from __future__ import print_function
7
8  for light in [ 'green', 'yellow', 'red', 'black', None ]:
9      if light == 'red':
10         print('stop')
11     elif light == 'yellow':
12         print('brake')
13     elif light == 'green':
14         print('drive')
15     else:
16         print('THE LIGHT IS BROKEN!')
17
18  appreciation = {
19     0: 'not driving',
20     30: 'slow',
21     60: 'normal',
22     90: 'dangerous',
23     120: 'suicidal'

```

(continues on next page)

(continued from previous page)

```
24     }
25
26 for i in range(5):
27     speed = 30*i
28     print("%s. Driving at speed %s is %s" % (i, speed, appreciation[speed]))
29
30 # End
```

- A '#' starts a comment: the '#', and anything following it on the same line, is disregarded. A Python script typically starts with a comment line like line 1 of the above script.
- Strings in Python can be delimited either by single quotes ('), double quotes (") or by triple double quotes ("""). The starting and ending delimiters have to be equal though. Strings in triple quotes can span several lines, like the string on lines 2-5.
- Indentation is essential. Indentation is Python's way of grouping statements. In small, sequential scripts, indentation is not needed and you should make sure that you start each new line in the first column. An `if` test or a `for` loop will however need indentation to mark the statement(s) inside the condition or loop. Thus, in the example, lines 8-15 are the block of statements that are executed repeatedly under the `for` loop construct in line 7. Notice that the condition and loop statements end with a ':'.

You should make sure that statements belonging to the same block are indented consistently. We advice you not to use tabs for indenting. A good practice commonly followed by most Python programmers is to indent with 4 spaces.

The indentation makes Python code easy to read for humans. Most modern editors will recognize Python code and help you with the indentation.

- Variables in Python do not need to be declared before using them. In fact, Python has no variables, only typed objects. An assignment is just the binding of a name to an object. That binding can be changed at each moment by a new assignment to the same name.
- Sequences of objects can be grouped in tuples or lists, and individual items of them are accessed by an index starting from 0.
- Function definitions can use both positional arguments and keyword arguments, but the keyword arguments must follow the positional arguments. The order in which keyword arguments are specified is not important.
- You can use names defined in other modules, but you need to import those first. This can be done by importing the whole module and then using a name relative to that module:

```
import mymodule
print(mymodule.some_variable)
```

or you can import specific names from a module:

```
from mymodule import some_variable
print(some_variable)
```

or you can import everything from a module (not recommended, because you can easily clutter your name space):

```
from mymodule import *
print(some_variable)
```


3.4 Some basic NumPy concepts

Warning: This section still needs to be written!

Numerical Python (or NumPy for short) is an extension to the Python language providing efficient operations on large (numerical) arrays. It relies heavily on NumPy, and most likely you will need to use some NumPy functions in your scripts. As NumPy is still quite young, the available documentation is not so extensive yet. Still, the tentative NumPy tutorial http://www.scipy.org/Tentative_NumPy_Tutorial already provides the basics.

If you have ever used some other matrix language, you will find a lot of similar concepts in NumPy.

To do: Introduce the (for users) most important NumPy concepts.

pyFormex uses the NumPy `ndarray` as implementation of fast numerical arrays in Python.

3.5 Formex data model

The most important geometrical object in pyFormex is the `Formex` class. A `Formex` (plural: `Formices`) can describe a variety of geometrical objects: points, lines, surfaces, volumes. The most simple geometrical object is the point, which in three dimensions is only determined by its coordinates (x, y, z) , which are numbered $(0, 1, 2)$ in pyFormex to be consistent with Python and NumPy indexing. Higher order geometrical objects are defined as a collection of points. The number of points of an object is called the *plexitude* of the object.

A `Formex` is a collection of geometrical objects of the same plexitude. The objects in the collection are called *elements* of the `Formex`. A `Formex` whose elements have plexitude n is also called an n -plex `Formex`. Internally, the coordinates of the points are stored in a NumPy `ndarray` with three dimensions. The coordinates of a single point are stored along the last axis (2) of the `Formex`; all the points of an element are stored along the second axis (1); different elements are stored along the first axis (0) of the `Formex`. The figure *The structure of a Formex* schematizes the structure of a `Formex`.

Warning: The beginning user should be aware not to confuse the three axes of a `Formex` with the axes of the 3D space. Both are numbered 0..2. The three coordinate axes form the components of the last axis of a `Formex`.

For simplicity of the implemented algorithms, internally pyFormex only deals with 3D geometry. This means that the third axis of a `Formex` always has length 3. You can however import 2D geometry: all points will be given a third coordinate $z = 0.0$. If you restrict your operations to transformations in the (x, y) -plane, it suffices to extract just the first two coordinates to get the transformed 2D geometry.

The `Formex` object `F` can be indexed just like a *NumPy* numerical array: `F[i]` returns the element with index i (counting from 0). For a `Formex` with plexitude n , the result will be an array with shape $(n, 3)$, containing all the points of the element. Further, `F[i][j]` will be a $(3,)$ -shaped array containing the coordinates of point j of element i . Finally, `F[i][j][k]` is a single floating point value representing one coordinate of that point.

In the following sections of this tutorial, we will first learn you how to create simple geometry using the `Formex` data model and how to use the basic pyFormex interface functions. The real power of the `Formex` class will then be established starting from the section *Transforming a Formex*.

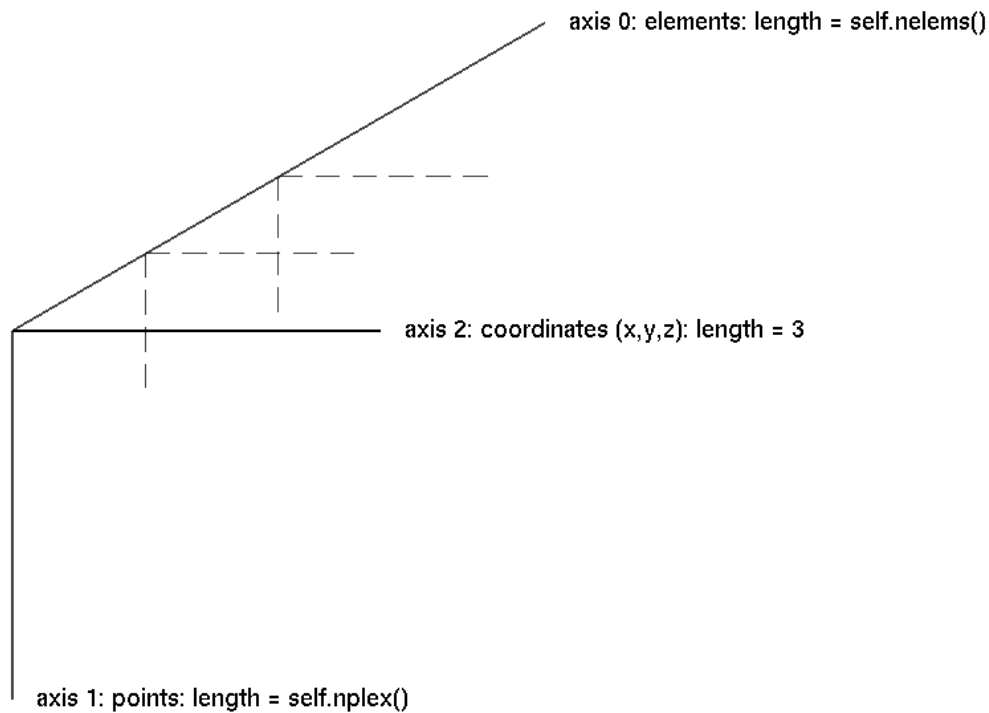


Fig. 2: The structure of a Formex

3.6 Creating a Formex

There are many, many ways to create `Formex` instances in your scripts. Most of the geometrical operations and transformations in `pyFormex` return geometry as a `Formex`. But how do you create a new geometric structure from simple coordinate data? Well, there are several ways to do that too, and we'll introduce them one by one.

3.6.1 Direct input of structured coordinate data

The most straightforward way to create a `Formex` is by directly specifying the coordinates of the points of all its elements in a way compatible to creating a 3D `ndarray`:

```
F = Formex([[ [0.,0.], [1.,0.] ], [ [1.,1.], [0.,1.] ]])
```

The data form a nested list of three levels deep. Each innermost level list holds the coordinates of a single point. There are four of them: `[0.,0.]`, `[1.,0.]`, `[1.,1.]` and `[0.,1.]`. Remark that we left out the third (`z`) coordinate and it will be set equal to zero. Also, though the values are integer, we added a dot to force floating point values.

Warning: Python by default uses integer math on integer arguments! We advice you to always write the decimal point in values that initialize variables that can have floating point values, such as lengths, angles, thicknesses. Use integer values only to initialize variables that can only have an integer value, such as the number of elements.

The second list level groups the points into elements. In this case there are two elements, each containing two points. The outermost list level then is the `Formex`: it has plexitude 2 and contains 2 elements. But what geometrical entities

does this represent? The plexitude alone does not specify what kind of geometric objects we are dealing about. A 2-plex element would presumably represent a straight line segment between two points in space, but it could just as well be used to represent a sphere (by its center and a point on the surface) or a plane (by a point in the plane and the direction of the normal).

By default, pyFormex will interpret the plexitude as follows:

Plexitude	Geometrical interpretation
1	Points
2	Straight line segments
3	Triangles
4 or higher	Polygons (possibly nonplanar)

We will see later how to override this default. For now, let's draw Formices with the default. Go back to the `example1.py` script in your editor, containing the line above, and add the `draw(F)` instruction to make it look like:

```
F = Formex([[0.,0.],[1.,0.],[1.,1.],[0.,1.]])
draw(F)
```

Save the script and execute it in pyFormex. You will see the following picture appear in the canvas.



Fig. 3: Two parallel lines

Now let's remove the two central '[' and '[' brackets in the first line:

```
F = Formex([[0.,0.],[1.,0.],[1.,1.],[0.,1.]])
draw(F,color=blue)
```

With the same data we have now created a 4-plex Formex with only one element. Execute the script again (do not forget to save it first) and you will see a square. Note that the draw command allows you to specify a color.



Fig. 4: A square.

But wait a minute! Does this represent a square surface, or just the four lines constituting the circumference of the square? Actually, it is a square surface, but since the pyFormex GUI by default displays in wireframe mode, unless

you have changed it, you will only see the border of the square. You can make surfaces and solids get fully rendered

by selecting the *Viewport* → *Render Mode* → *Flat* option or using the shortcut



button in the toolbar. You will then see



Fig. 5: The square in smooth rendering.

pyFormex by default uses wireframe rendering, because in a fully rendered mode many details are obscured. Switch

back to wireframe mode using the *Viewport* → *Render Mode* → *Wireframe* menu option or



toolbar button.

Now suppose you want to define a Formex representing the four border lines of the square, and not the surface inside that border. Obviously, you need a 4 element 2-plex Formex, using data structured like this:

```
F = Formex([[ [0., 0.], [0., 1.]],
             [ [0., 1.], [1., 1.]],
             [ [1., 1.], [1., 0.]],
             [ [1., 0.], [0., 0.]]])
draw(F, color=blue, clear=True)
```

Try it, and you will see an image identical to the earlier figure *A square*.. But now this image represents four straight lines, while the same image formerly represented a square plane surface.

Warning: When modeling geometry, always be aware that what you think you see is not necessarily what it really is!

The `clear=True` option in the `draw` statement makes sure the screen is cleared before drawing. By default the `pyFormex draw` statement does not clear the screen but just adds to what was already drawn. You can make the `clear=True` option the default from the *Viewport* → *Drawing Options* menu. Do this now before continuing.

Changing the rendering mode, the perspective and the viewpoint can often help you to find out what the image is really representing. But interrogating the Formex data itself is the definite way to make sure:

```
F = Formex([[ [0., 0.], [1., 0.], [1., 1.], [0., 1.] ]])
print(F.shape)
F = Formex([[ [0., 0.], [1., 0.]], [[1., 1.], [0., 1.]] ])
print(F.shape)
```

This will print the length of the three axes of the coordinate array. In the first case you get (1, 4, 3) (1 element of plexitude 4), while the second one gives (2, 2, 3) (2 elements of plexitude 2).

You can also print the whole Formex, using `print(F)`, giving you the coordinate data in a more readable fashion than the list input. The last example above will yield: `{[0.0, 0.0, 0.0; 1.0, 0.0, 0.0], [1.0, 1.0, 0.0; 0.0, 1.0, 0.0]}`. In the output, coordinates are separated by commas and points by semicolons. Elements are contained between brackets and the full Formex is placed inside braces.

Until now we have only dealt with plane structures, but 3D structures are as easy to create from the coordinate data. The following Formex represents a pyramid defined by four points (a tetrahedron):

```
F = Formex([[0.,0.,0.],[1.,0.,0.],[0.,1.,0.],[0.,0.,1.]],eltype='tet4')
draw(F)
```

Depending on your current rendering mode, this will produce an image like one of the following:

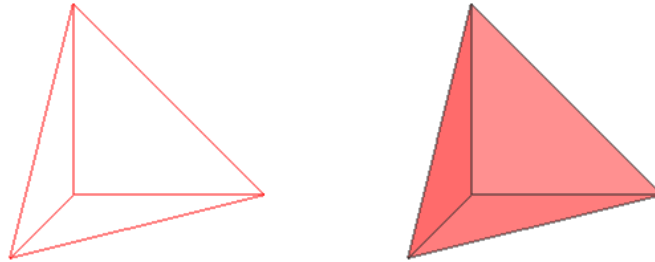




Fig. 6: The tetrahedron in wireframe and smoothwire (transparent) rendering

The smoothwire mode can be set from the *Viewport* → *Render Mode* → *Smoothwire* option or the  button.

The transparent mode can be toggled using the  button.

Hold down the left mouse button and move the mouse: the pyramid will rotate. In the same way, holding down the right button will zoom in and out. Holding down the middle button (possibly the mouse wheel, or the left and right button together) will move the pyramid over the canvas. Practice a bit with these mouse manipulations, until you get a feeling of what they do. All these mouse operations do not change the coordinates of the structure: they just change

the way you're looking at it. You can restore the default view with the *Views* → *Front* menu or the  button.

The default installation of pyFormex provides seven default views: *Front*, *Back*, *Left*, *Right*, *Top*, *Bottom* and *Iso*. They can be set from the *Views* menu items or the corresponding view buttons in the toolbar. The default *Front* corresponds to the camera looking in the $-z$ direction, with the x axis oriented to the right and the y axis upward.

We explicitly added the element type `tet4` when creating the pyramid. Without it, pyFormex would have interpreted the 4-plex Formex as a quadrilateral (though in this case a non-planar one).

3.6.2 Using the `pattern()` function

In the previous examples the Formices were created by directly specifying the coordinate data. That is fine for small structures, but quickly becomes cumbersome when the structures get larger. The `pattern()` function can reduce the amount of input needed to create a Formex from scratch.

This function creates a series of points that lie on a regular grid with unit step. These points can then be used to create some geometry. Do not worry about the regularity of the grid: pyFormex has many ways to transform it afterwards.

The points are created from a string input, interpreting each character as a code specifying how to move from the previous point to the new point. The start position on entry is the origin `[0.,0.,0.]`.

Currently the following codes are defined:

- 0: goto origin (0.,0.,0.)
- 1..8: move in the x,y plane, as specified below
- 9 or .: remain at the same place (i.e. duplicate the last point)
- A..I: same as 1..9 plus step +1. in z-direction
- a..i: same as 1..9 plus step -1. in z-direction
- /: do not insert the next point

When looking at the x,y-plane with the x-axis to the right and the y-axis up, we have the following basic moves: 1 = East, 2 = North, 3 = West, 4 = South, 5 = NE, 6 = NW, 7 = SW, 8 = SE.

Adding 16 to the ordinal of the character causes an extra move of +1. in the z-direction. Adding 48 causes an extra move of -1. This means that 'ABCDEFGHGI', resp. 'abcdefghi', correspond with '123456789' with an extra z +/- = 1. This gives the following schema:

z+=1			z unchanged			z -= 1		
F	B	E	6	2	5	f	b	e
C	-----I	-----A	3	-----9	-----1	c	-----i	-----a
G	D	H	7	4	8	g	d	h

The special character '/' can be put before any character to make the move without inserting the new point. You need to start the string with a '0' or '9' to include the origin in the output.

For example, the string '0123' will result in the following four points, on the corners of a unit square:

```
[ [ 0.  0.  0.]
  [ 1.  0.  0.]
  [ 1.  1.  0.]
  [ 0.  1.  0.] ]
```

Run the following simple script to check it:

```
P = pattern('0123')
print(P)
```

Now you can use these points to initialize a Formex

```
F = Formex(pattern('0123'))
draw(F)
```

This draws the four points. But the Formex class allows a lot more. You can directly initialize a Formex with the pattern input string, preceded by a modifier field. The modifier specifies how the list of points should be grouped into multipoint elements. It normally consists of a number specifying the plexitude of the elements, followed by a ':' character. Thus, after the following definitions:

```
F = Formex('1:0123')
G = Formex('2:0123')
H = Formex('4:0123')
```

F will be a set of 4 points (plexitude 1), G will be 2 line segments (plexitude 2) and H will a single square (plexitude 4).

Furthermore, the special modifier 'l:' can be used to create line elements between each point and the previous one. Note that this in effect doubles the number of points in the Formex and always results in a 2-plex Formex. Here's an example:

```
F = Formex('l:1234')
draw(F)
```

It creates the same circumference of a unit square as above (see figure *A square*.), but is much simpler than the explicit specification of the coordinates we used before. Notice that we have used here '1234' instead of '0123' to get the four corners of the unit square. Check what happens if you use '0123', and try to explain why.

Note: Because the creation of line segments between subsequent points is such a common task, the Formex class even allows you to drop the 'l:' modifier. If a Formex is initialized by a string without modifier field, the 'l:' is silently added.

Figure *Images generated from the patterns '127', '11722' and '22584433553388'* shows some more examples.

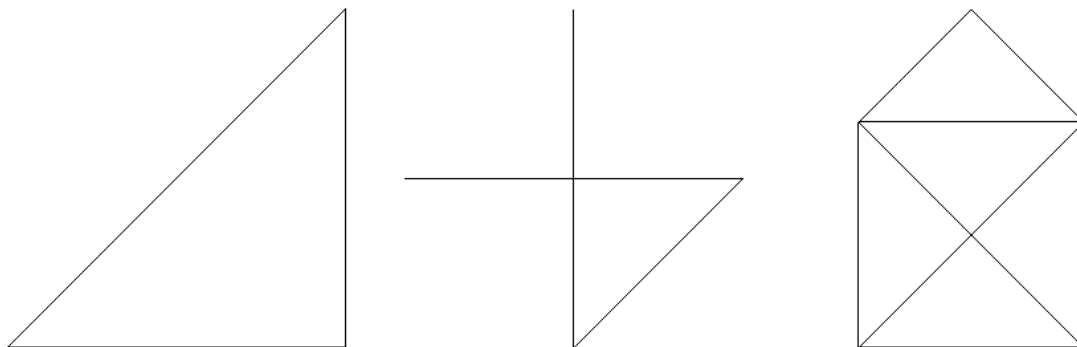



Fig. 7: Images generated from the patterns '127', '11722' and '22584433553388'

Some simple wireframe patterns are defined in `simple.py` and are ready for use. These pattern strings are stacked in a dictionary called 'Pattern'. Items of this dictionary can be accessed like `Pattern['cube']`.

```
from simple import Pattern F = Formex(Pattern['cube']) print(F.shape) draw(F,color=blue,view='iso')
```

The printed out shape of the Formex is `(12, 2, 3)`, confirming that what we have created here is not a 3D solid cube, nor the planes bounding that cube, but merely twelve straight line segments forming the edges of a cube.

The `view='iso'` option in the draw statement rotates the camera so that it looks in the `[-1,-1,-1]` direction. This is one of the predefined viewing directions and can also be set from the *Views* menu or using the  button.

While the `pattern()` function can only generate points lying on a regular cartesian grid, pyFormex provides a wealth of transformation functions to move the points to other locations after they were created. Also, the Turtle plugin module provides a more general mechanism to create planar wireframe structures.

3.6.3 Reading coordinates from a file or a string

Sometimes you want to read the coordinates from a file, rather than specifying them directly in your script. This is especially handy if you want to import geometry from some other program that can not export data in a format that is

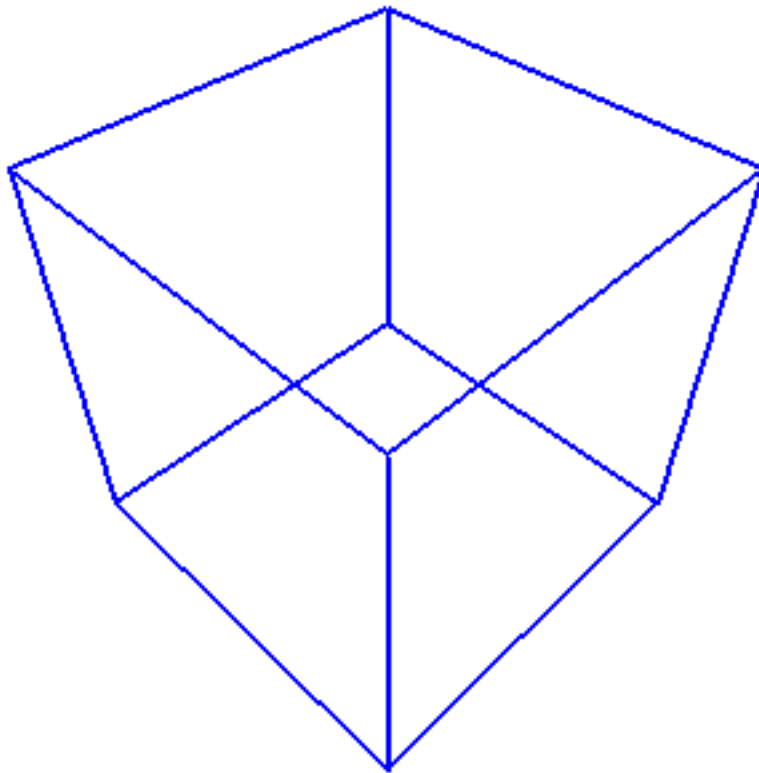


Fig. 8: A wireframe cube

understood by pyFormex. There usually is a way to write the bare coordinates to a file, and the pyFormex scripting language provides all the necessary tools to read them back.

As an example, create (in the same folder where you store your scripts) the text file `square.txt` with the following contents:

```
0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0,
1, 1, 0, 2, 1, 0, 2, 2, 0,
1, 2, 0
```

Then create and execute the following script.

```
chdir(__file__)
F = Formex.fromfile('square.txt', sep=',', nplex=4)
draw(F)
```

It will generate two squares, as shown in the figure *Two squares with coordinates read from a file*.



Fig. 9: Two squares with coordinates read from a file

The `chdir(__file__)` statement sets your working directory to the directory where the script is located, so that the filename can be specified without adding the full pathname. The `Formex.fromfile()` call reads the coordinates (as specified, separated by ‘;’) from the file and groups them into elements of the specified plexitude (4). The grouping of coordinates on a line is irrelevant: all data could just as well be given on a single line, or with just one value per line. The separator character can be accompanied by extra whitespace. Use a space character if your data are only separated by whitespace.

There is a similar `Formex.fromstring()` method, which reads coordinates directly from a string in the script. If you have a lot of coordinates to specify, this may be far more easy than using the list formatting. The following script yields the same result as the above one:

```
F = Formex.fromstring("""
0 0 0 0 1 0 1 1 0 1 0 0
1 1 0 2 1 0 2 2 0 1 2 0
""", nplex=4)
draw(F)
```

Here we used the default separator, which is a space.

Note: Make sure to use `Formex.fromfile()`, to distinguish it from `Coords.fromfile()` and `numpy.fromfile()`.

3.7 Concatenation and lists of Formices

Multiple Formices can be concatenated to form one new Formex. There are many ways to do this, but the simplest is to use the `+` or `+=` operator. Notice the difference: the `+` operator does not change any of the arguments, but the `+=` operator adds the second argument to the first, changing its definition:

```
F = Formex('1234')
G = Formex('5')
H = F + G
draw(H)
```

displays the same Formex as:

```
F += G
draw(F)
```

but in the latter case, the original definition of `F` is lost.

The `+=` operator is one of the very few operations that change an existing Formex. Nearly all other operations return a resulting Formex without changing the original ones.

Because a Formex has a single plexitude and element type, concatenation is restricted to Formices of the same plexitude and with the same `eltype`. If you want to handle structures with elements of different plexitude as a single object, you have to group them in a list:

```
F = Formex('1234')
G = Formex([0.5, 0.5, 0.])
H = [F, G]
draw(H, color=red)
```

This draws the circumference of a unit square (`F`: plexitude 2) and the center point of the square (`G`: plexitude 1), both in red.

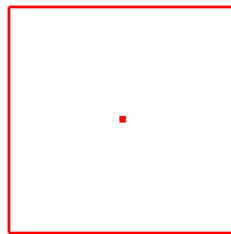


Fig. 10: A square and its center point.

3.8 Formex property numbers

Apart from the coordinates of its points, a `Formex` object can also store a set of property numbers. This is a set of integers, one for every element of the Formex. The property numbers are stored in an attribute `prop` of the Formex. They can be set, changed or deleted, and be used for any purpose the user wants, e.g. to number the elements in a different order than their appearance in the coordinate array. Or they can be used as pointers into a large database that stores all kind of properties for that element. Just remember that a Formex either has no property numbers, or a complete set of numbers: one for every element.

Property numbers can play an important role in the modeling process, because they present some means of tracking how the resulting Formex was created. Indeed, each transformation of a Formex that preserves its structure, will also preserve the property numbers. Concatenation of Formices with property numbers will also concatenate the property numbers. If any of the concatenated Formices does not have property numbers, it will receive value 0 for all its elements. If all concatenated Formices are without properties, so will be the resulting Formex.

On transformations that change the structure of the Formex, such as replication, each element of the created Formex will get the property number of the Formex element it was generated from.

To add properties to a Formex, use the `setProp()` method. It ensures that the property array is generated with the correct type and shape. If needed, the supplied values are repeated to match the number of elements in the Formex. The following script creates four triangles, the first and third get property number 1, the second and fourth get property 3.

```
F = Formex('3:.12.34.14.32')
F.setProp([1,3])
print(F.prop)    # --> [1 3 1 3]
```

As a convenience, you can also specify the property numbers as a second argument to the Formex constructor. Once the properties have been created, you can safely change individual values by directly accessing the `prop` attribute.

```
F = Formex('3:.12.34.14.32', [1, 3])
F.prop[3] = 4
print(F.prop)    # --> [1 3 1 4]
draw(F)
drawNumbers(F)
```

When you draw a Formex with property numbers using the default draw options (i.e. no color specified), pyFormex will use the property numbers as indices in a color table, so different properties are shown in different colors. The default color table has eight colors: [black, red, green, blue, cyan, magenta, yellow, white] and will wrap around if a property value larger than 7 is used. You can however specify any other and larger colorset to be used for drawing the property colors. The following figure shows different renderings of the structure created by the above script. The `drawNumbers()` function draws the element numbers (starting from 0).

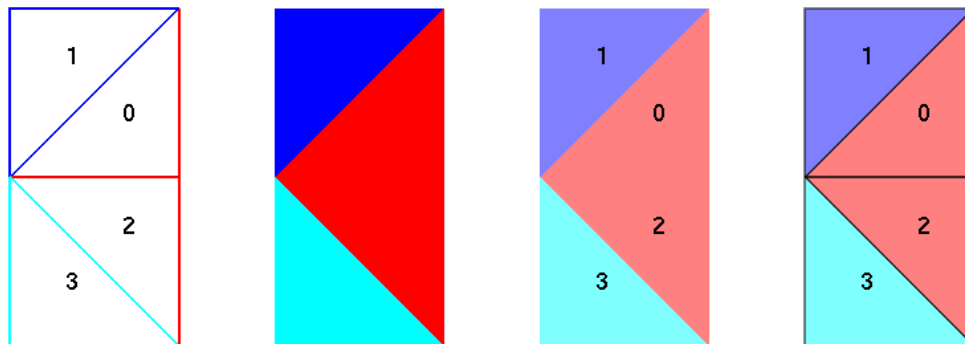



Fig. 11: A Formex with property numbers drawn as colors. From left to right: wireframe, flat, flat (transparent), flatwire (transparent).

In flat rendering mode, the element numbers may be obscured by the faces. In such case, you can make the numbers

visible by using the transparent mode, which can be toggled with the  button.

Adding properties to a Formex is often done with the sole purpose of drawing with multiple colors. But remember

you are free to use the properties for any purpose you want. You can even save, change and restore them throughout the lifetime of a Formex object, thus you can attribute multiple property sets to a Formex.

3.9 Getting information about a Formex

While the visual feedback on the canvas usually gives a good impression of the structure you created, at times the view will not provide enough information or not precise enough. Viewing a 3D geometry on a 2D screen can at times even be very misleading. The most reliable source for checking your geometry will always be the Formex data itself. We have already seen that you can print the coordinates of the Formex `F` just by printing the Formex itself: `print(F)`. Likewise you can see the property numbers from a `print(F.prop)` instruction.

But once you start using large data structures, this information may become difficult to handle. You are usually better off with some generalized information about the Formex object. The `Formex` class provides a number of methods that return such information. The following table lists the most interesting ones.

Function	Return value
<code>F.nelems()</code>	The number of elements in the Formex
<code>F.nplex()</code>	The plexitude of the Formex (the number of points in each element of the Formex)
<code>F.bbox()</code>	The bounding box of the Formex
<code>F.center()</code>	The center of the bbox of the Formex
<code>F.sizes()</code>	The size of the bbox of the Formex

3.10 Saving geometry

Sometimes you want to save the created geometry to a file, e.g. to reread it in a next session without having to create it again, or to pass it to someone else. While pyFormex can export geometry in a large number of formats, the best and easiest way is to use the `writeGeomFile()` function. This ensures a fast and problem free saving and read back of the geometry. The geometry is saved in pyFormex's own file format, in a file with extension `'pgf'`. This format is well documented (see *pyFormex file formats*) and thus accessible for other programs.

```
A = Formex('3:012/1416').setProp(1)
B = Formex('4:0123').translate([1., 1., 0.])
draw(B)
writeGeomFile('saved.pgf', [A, B])
```

When reading back such a file, the objects end up in a dictionary. Quit pyFormex, restart it and read back the just saved file.

```
D = readGeomFile('saved.pgf')
print(D)
print(D.keys())
draw(D.values())
```

In this case the keys were auto-generated. We could however specified the keys when creating the file, by specifying a dictionary instead of a list of the objects to save.

```
writeGeomFile('saved.pgf', {'two_triangles':A, 'a_square':B})
D = readGeomFile('saved.pgf')
print(D.keys())
```

3.11 Saving images

Often you will want to save an image of the created geometry to a file, e.g. to include it in some document. This can readily be done from the *File* → *Save Image* menu. You just have to fill in the file name and click the *Save* button. You can specify the file format by using the appropriate extension in the file name. The default and recommended format is `png`, but `pyFormex` can save in commonly used bitmap formats like `jpg` or `gif` as well.

But you can also create the images from inside your script. Just import the `image` module and call the `image.save()` function:

```
import gui.image
image.save("my_image.png")
```

Often you will want to change some settings, like rendering mode or background color, to get a better looking picture. Since the main goal of `pyFormex` is to automate the creation and transformation of geometrical models, all these settings can be changed from inside your script as well. The following code was used to create the four images in figure *A Formex with property numbers drawn as colors*. From left to right: *wireframe*, *flat*, *flat (transparent)*, *flatwire (transparent)*. above.

```
import gui.image
chdir(__file__)
reset()
bgcolor(white)
linewidth(2)
canvasSize(200,300)
F = Formex('3:.12.34.14.32',[1,3])
F.prop[3] = 4
clear()
draw(F)
drawNumbers(F)
wireframe()
image.save('props-000.png')
flat()
transparent(False)
image.save('props-001.png')
transparent(True)
image.save('props-002.png')
flatwire()
image.save('props-003.png')
```

The following table lists the interactive menu option and the correspondant programmable function to be used to change some of the most common rendering settings.

Purpose	Function(s)	Menu item
Background color	<code>bgcolor()</code>	<i>Viewport</i> → <i>Background Color</i>
Line width	<code>linewidth()</code>	<i>Viewport</i> → <i>LineWidth</i>
Canvas Size	<code>canvasSize()</code>	<i>Viewport</i> → <i>Canvas Size</i>
Render Mode	<code>wireframe()</code> , <code>flat()</code> , <code>flatwire()</code> , <code>smooth()</code> , <code>smoothwire()</code>	<i>Viewport</i> → <i>Render Mode</i>
Transparency	<code>transparent()</code>	

3.12 Transforming a Formex

Until now, we've only created simple Formices. The strength of `pyFormex` however is the ease to generate large geometrical models by a sequence of mathematical transformations. After creating a initial Formex, you can transform

it by creating copies, translations, rotations, projections,...

The `Formex` class has an wide range of powerful transformation methods available, and this is not the place to treat them all. The reference manual *pyFormex reference manual* describes them in detail.

We will illustrate the power of the `Formex` transformations by studying one of the examples included with pyFormex. The examples can be accessed from the *Examples* menu option.

Note: If you have installed multiple script directories, the examples may be found in a submenu *Scripts* → *Examples*.

When a script is selected from this menu, it will be executed automatically. Select the *Examples* → *Level* → *Beginner* → *Helix* example. You will see an image of a complex helical frame structure:

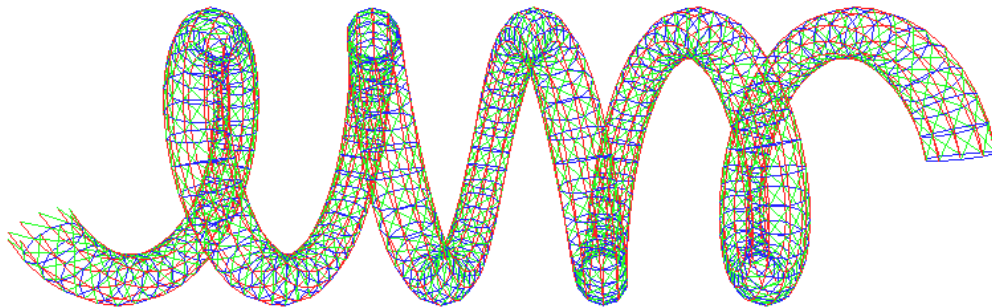


Fig. 12: A helical frame structure (Helix example)

Yet the geometry of this complex structure was built from the very simple pyFormex script shown below (Use *File* → *Edit script* to load it in your editor. Leaving out the comments and docstring, the relevant part of the script looks like this:

```

1  """Helix example from pyFormex"""
2  m = 36 # number of cells along helix
3  n = 10 # number of cells along circular cross section
4  reset()
5  setDrawOptions({'clear':True})
6  F = Formex('l:164'), [1,2,3]); draw(F)
7  F = F.replic(m,1.,0); draw(F)
8  F = F.replic(n,1.,1); draw(F)
9  F = F.translate(2,1.); draw(F,view='iso')
10 F = F.cylindrical([2,1,0],[1.,360./n,1.]); draw(F)
11 F = F.replic(5,m*1.,2); draw(F)
12 F = F.rotate(-10.,0); draw(F)
13 F = F.translate(0,5.); draw(F)
14 F = F.cylindrical([0,2,1],[1.,360./m,1.]); draw(F)
15 draw(F,view='right')
```

The script shows all steps in the building of the helical structure. We will explain and illustrate them one by one. If you want to see the intermediate results in pyFormex during execution of the script, you can set a wait time between

subsequent drawing operations with *Settings* → *Draw Wait Time*. Or alternatively, you can start the script with the



button: pyFormex will then halt before each draw function and wait until you push the



again.

The script starts with setting the two parameters *m* and *n*. It is always a good idea to put constants in a variable. That makes it easy to change the values in a single place when you want to create another structure: *your model has become a parametric model*.

Line 3 resets the drawing options to the defaults. It is not essential in this script but it is often a good idea to restore the defaults, in case they would have been changed by a script that was run previously. Setting the `clear=True` option in line 4 makes sure the subsequent drawing instructions will remove the previous step from the canvas.

In line 5 we create the basic geometrical entity for this structure: a triangle consisting of three lines, which we give the properties 1, 2 and 3, so that the three lines are shown in a different color:

```
F = Formex('l:164', [1, 2, 3])
```

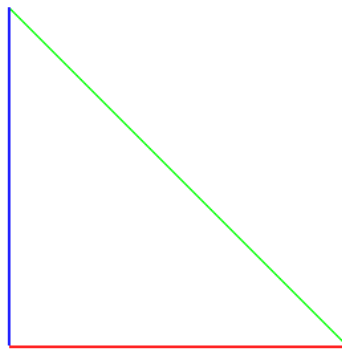


Fig. 13: The basic Formex

This basic Formex is copied *m* times with a translation step 1.0 (this is precisely the length of the horizontal edge of the triangle) in the 0 direction:

```
F = F.replic(m, 1., 0)
```



Fig. 14: Replicated in x-direction

Then, the new Formex is copied *n* times with the same step size in the direction 1.

```
F = F.replic(n, 1., 1)
```

Now a copy of this last Formex is translated in direction '2' with a translation step of '1'. This necessary for the transformation into a cylinder. The result of all previous steps is a rectangular pattern with the desired dimensions, in a plane $z=1$.

```
F = F.translate(2, 1); drawit(F, 'iso')
```

This pattern is rolled up into a cylinder around the 2-axis.

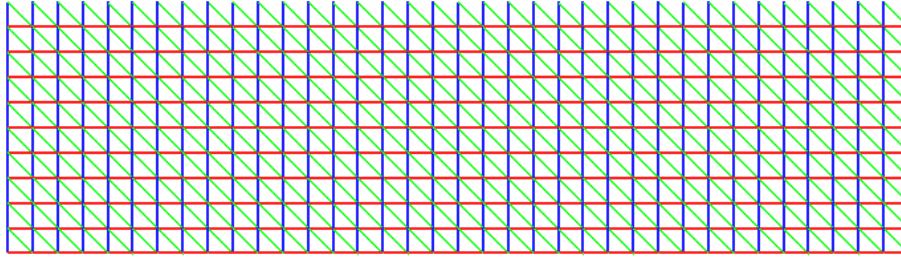


Fig. 15: Replicated in y-direction

```
F = F.cylindrical([2,1,0],[1.,360./n,1.]); drawit(F,'iso')
```

This cylinder is copied 5 times in the 2-direction with a translation step of ‘m’ (the length of the cylinder).

```
F = F.replic(5,m,2); drawit(F,'iso')
```

The next step is to rotate this cylinder -10 degrees around the 0-axis. This will determine the pitch angle of the spiral.

```
F = F.rotate(-10,0); drawit(F,'iso')
```

This last Formex is now translated in direction ‘0’ with a translation step of ‘5’.

```
F = F.translate(0,5); drawit(F,'iso')
```

Finally, the Formex is rolled up, but around a different axis than before. Due to the pitch angle, a spiral is created. If the pitch angle would be 0 (no rotation of -10 degrees around the 0-axis), the resulting Formex would be a torus.

```
F = F.cylindrical([0,2,1],[1.,360./m,1.]); drawit(F,'iso')
drawit(F,'right')
```

3.13 Converting a Formex to a Mesh model

pyFormex contains other geometry models besides the Formex. The `Mesh` model e.g. is important in exporting the geometry to finite element (FE) programs. A Formex often contains many points with (nearly) the same coordinates. In a Finite Element model, these points have to be merged into a single node, to express the continuity of the material. The `toMesh()` method of a Formex performs exactly that. It returns a Mesh instance, which has two import array attributes ‘coords’ and ‘elems’:

- `coords` is a float array with shape `(ncoords,3)`, containing the coordinates of the merged points (nodes),
- `elems` is an integer array with shape `(F.nelems(),F.nplex())`, describing each element by a list of node numbers. These can be used as indices in the `coords` array to find the coordinates of the node. The elements and their nodes are in the same order as in `F`.

```
from simple import *
F = Formex(Pattern['cube'])
draw(F)
```

(continues on next page)

(continued from previous page)

```
M = F.toMesh()
print('Coords',M.coords)
print('Elements',M.elems)
```

The output of this script are the coordinates of the unique nodes of the Mesh, and the connectivity of the elements. The connectivity is an integer array with the same shape as the first two dimensions of the Formex: (F.nelems(),F.nplex()):

```
Nodes
[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 1.  1.  0.]
 [ 0.  0.  1.]
 [ 1.  0.  1.]
 [ 0.  1.  1.]
 [ 1.  1.  1.]]
Elements
[[0 1]
 [1 3]
 [3 2]
 [2 0]
 [0 4]
 [1 5]
 [3 7]
 [2 6]
 [4 5]
 [5 7]
 [7 6]
 [6 4]]
```

The inverse operation of transforming a Mesh model back into a Formex is also quite simple: `Formex(nodes[elems])` will indeed be identical to the original `F` (within the tolerance used in merging of the nodes).

```
>>> G = Formex(nodes[elems])
>>> print(allclose(F.f,G.f))
True
```

The `allclose` function in the second line tests that all coordinates in both arrays are the same, within a small tolerance.

PYFORMEX USER GUIDE

Warning: This document and the sections below it are still very incomplete!

Abstract

The user guide explains in depth the most important components of pyFormex. It shows you how to start pyFormex, how to use the Graphical User Interface (GUI), how to use the most important data classes, functions and GUI widgets in your scripts. It also contains sections dedicated to customization and extension of pyFormex.

Sections of the user guide:

4.1 Running pyFormex

To run pyFormex, simply enter the command `pyformex` in a terminal window. This will start the Graphical User Interface (GUI), from where you can launch examples or load, edit and run your own scripts.

The installation procedure may have installed into your desktop menu or even have created a start button in the desktop panel. These provide convenient shortcuts to start the GUI by the click of a mouse button.

The program takes some optional command line arguments, that modify the behaviour of the program. Appendix *Command line options* gives a full list of all options. For normal use however you will seldom need to use any of them. Therefore, we will only explain here the more commonly used ones.

By default, sends diagnostical and informational messages to the terminal from which the program was started. Sometimes this may be inconvenient, e.g. because the user has no access to the starting terminal. You can redirect these messages to the message window of the GUI by starting `pyformex` with the command `pyformex --redirect`. The desktop starters installed by the installation procedure use this option.

In some cases the user may want to use the mathematical power of without the GUI. This is e.g. useful to run complex automated procedures from a script file. For convenience, will automatically enter this batch mode (without GUI) if the name of a script file was specified on the command line; when a script file name is absent, start in GUI mode. Even when specifying a script file, You can still force the GUI mode by adding the option `-gui` to the command line.

4.2 Command line options

The following is a complete list of the options for the `pyformex` command. This output can also be generated by the command `pyformex --help`.

```
/usr/bin/python3
```

```
usage: pyformex [-h] [--version] [--gui] [--nogui] [--nocanvas]
               [--interactive] [--uselib] [--nouselib] [--config CONFIG]
               [--nodefaultconfig] [--redirect] [--noredirect]
               [--debug DEBUG] [--debuglevel DEBUGLEVEL] [--debugitems]
               [--mesa] [--dri] [--nodri] [--opengl OPENGL] [--shader SHADER]
               [--vtk VTK] [--testcamera] [--memtrack] [--fastnurbs]
               [--pathlib] [--bindings BINDINGS] [--unicode] [--experimental]
               [--listfiles] [--listmodules [PKG [PKG ...]]] [--search]
               [--remove] [--whereami] [--detect]
               [--doctest [MODULE [MODULE ...]]]
               [--pytest [MODULE [MODULE ...]]]
               [--docmodule [MODULE [MODULE ...]]] [-c SCRIPT]
               [FILE [FILE ...]]
```

pyFormex is a tool for generating, manipulating and transforming large geometrical models of 3D structures by sequences of mathematical transformations.

positional arguments:

FILE	pyFormex script files to be executed on startup. The files should have a .py extension. Their contents will be executed as a pyFormex script. While mostly used with the --nogui option, this will also work in GUI mode.
------	---

optional arguments:

-h, --help	Show this help message and exit
--version	Show program's version number and exit
--gui	Start the GUI (this is the default when no scriptname argument is given)
--nogui	Do not start the GUI (this is the default when a scriptname argument is given)
--nocanvas	Do not add an OpenGL canvas to the GUI (this is for development purposes only!)
--interactive	Go into interactive mode after processing the command line parameters. This is implied by the --gui option.
--uselib	Use the pyFormex C lib if available. This is the default.
--nouselib	Do not use the pyFormex C-lib.
--config CONFIG	Use file CONFIG for settings. This file is loaded in addition to the normal configuration files and overwrites their settings. Any changes will be saved to this file.
--nodefaultconfig	Skip the default site and user config files. This option can only be used in conjunction with the --config option.
--redirect	Redirect standard output to the message board (ignored with --nogui)
--noredirect	Do not redirect standard output to the message board.
--debug DEBUG	Display debugging information to sys.stdout. The value is a comma-separated list of (case-insensitive) debug items. Use option --debugitems to list them. The special value 'all' can be used to switch on all debug info.
--debuglevel DEBUGLEVEL	Display debugging info to sys.stdout. The value is an

(continues on next page)

(continued from previous page)

```

int with the bits of the requested debug levels set. A
value of -1 switches on all debug info. If this option
is used, it overrides the --debug option.
--debugitems      Show all available debug items. Each of these can be
                  used with the -debug option.
--mesa            Force the use of software 3D rendering through the
                  mesa libs. The default is to use hardware accelerated
                  rendering whenever possible. This flag can be useful
                  when running pyFormex remotely on another host. The
                  hardware accelerated version will not work over remote
                  X.
--dri             Use Direct Rendering Infrastructure. By default,
                  direct rendering will be used if available.
--nodri          Do not use the Direct Rendering Infrastructure. This
                  may be used to turn off the direc rendering, e.g. to
                  allow better capturing of images and movies.
--opengl OPENGL  Force the use of a specific OpenGL version. The
                  version should be specified as a string 'a.b'. The
                  default is 2.0
--shader SHADER  Force the use of an alternate GPU shader for the
                  OpenGL rendering. If the default selected shader does
                  not work well for your hardware, you can use this
                  option to try one of the alternate shaders. See
                  'pyformex --detect' for a list of the available
                  shaders.
--vtk VTK        Specify which version of python-vtk to use in vtk_itf
                  plugin. The value can be one of: 'standard', 'light',
                  or 'default'. 'standard' is the version as distributed
                  from python-vtk. 'light' is the trimmed vtk6 version
                  as distributed with pyFormex.
--testcamera     Print camera settings whenever they change.
--memtrack       Track memory for leaks. This is only for developers.
--fastnurbs      Test C library nurbs drawing: only for developers!
--pathlib        In Python3 version, use pathlib library. This is only
                  for development and testing purposes.
--bindings BINDINGS Override the configuration setting for the Qt bindings
                  to be used. Available bindings are 'pyside',
                  'pyside2', 'pyqt4', 'pyqt5'. A value 'any' may be
                  given to let pyFormex find out which are available and
                  use one of these.
--unicode        Allow unicode filenames. Beware: this is experimental!
--experimental   Allow the pyformex/experimental modules to be loaded.
                  Beware: this should only be used if you know what you
                  are doing!
--listfiles      List the pyFormex Python source files and exit.
--listmodules [PKG [PKG ...]]
                  List the Python modules in the specified pyFormex
                  subpackage and exit. Specify 'core' to just list the
                  modules in the pyFormex top level. Specify 'all' to
                  list all modules. The default is to list the modules
                  in core, lib, plugins, gui, opengl.
--search         Search the pyformex source for a specified pattern and
                  exit. This can optionally be followed by -- followed
                  by options for the grep command and/or '-a' to search
                  all files in the extended search path. The final
                  argument is the pattern to search. '-e' before the
                  pattern will interpret this as an extended regular

```

(continues on next page)

(continued from previous page)

```

expression. '-l' option only lists the names of the
matching files.
--remove          Remove the pyFormex installation and exit. This option
                  only works when pyFormex was installed from a tarball
                  release using the supplied install procedure. If you
                  install from a distribution package (e.g. Debian), you
                  should use your distribution's package tools to remove
                  pyFormex. If you run pyFormex directly from SVN
                  sources, you should just remove the whole checked out
                  source tree.
--whereami        Show where the pyformex package is installed and exit.
--detect          Show detected helper software and exit.
--doctest [MODULE [MODULE ...]]
                  Run the docstring tests for the specified pyFormex
                  modules and exit. MODULE name is specified in Python
                  syntax, relative to pyformex package (e.g. coords,
                  plugins.curve).
--pytest [MODULE [MODULE ...]]
                  Run the pytest tests for the specified pyFormex
                  modules and exit. MODULE name is specified in Python
                  syntax, relative to pyformex package (e.g. coords,
                  plugins.curve).
--docmodule [MODULE [MODULE ...]]
                  Print the autogenerated documentation for module
                  MODULE and exit. This is mostly useful during the
                  generation of the pyFormex reference manual, as the
                  produced result still needs to be run through the
                  Sphinx documentation generator. MODULE is the name of
                  a pyFormex module (Python syntax).
-c SCRIPT, --script SCRIPT
                  A pyFormex script to be executed at startup. It is
                  executed before any specified script files. This is
                  mostly used in --nogui mode, when the script to
                  execute is very short.

```

More info on <http://pyformex.org>

4.3 Running without the GUI

If you start with the `--nogui` option, no Graphical User Interface is created. This is extremely useful to run automated scripts in batch mode. In this operating mode, will interpret all arguments remaining after interpreting the options, as filenames of scripts to be run (and possibly arguments to be interpreted by these scripts). Thus, if you want to run a script `myscript.py` in batch mode, just give the command `pyformex myscript.py`.

The running script has access to the remaining arguments in the global list variable `argv`. The script can use any arguments of it and pop them of the list. Any arguments remaining in the `argv` list when the script finishes, will be used for another execution cycle. This means that the first remaining argument should again be a script.

4.4 Importing pyFormex

Warning: This document contains very preliminary information of a new feature under development. Use at your own risk!

Abstract

This document gives some guidelines about how to import pyFormex into a Python application (instead of running the Python application from pyFormex).

4.4.1 Background

Traditionally, pyFormex is launched with the ‘pyformex’ command, and pyFormex (or Python) scripts are executed from the pyFormex environment (whether GUI or non-GUI).

Sometimes however it may be more suitable to import pyFormex into another Python script or environment. Therefore some modifications are underway to allow this.

4.4.2 How to proceed

In commit bd27925 of the git repositories the basic necessary changes were made to allow a basic pyFormex to be imported into normal Python workflow. These changes are currently only available when using pyFormex from the git sources. You may have to do a pull first.

4.4.3 Running non-GUI scripts

If you do not need the pyFormex GUI, the following is required to import pyFormex and make its scripting language usable just like in a pyFormex script/application:

- make sure Python finds the path from which to import pyFormex,
- import pyFormex (this is done implicitly when importing one of the modules or subpackages from pyFormex),
- import the pyFormex scripting language, if you want to use it. You probably want to, since you have imported pyFormex, but you would not need it if you just want to use some pyFormex classes or modules.

While there are many ways to do this, we present hereafter a few typical examples of how it can be done. Suppose we have a (partial) directory layout as follows:

```

/
|-- home
|   |-- user
|       |-- pyformex
|           |-- pyformex
|               |-- main.py
|                   |-- gui
|                       |-- opengl
|                           |-- plugins
|                               |-- apps

```

(continues on next page)

(continued from previous page)

```
| | | | | -- example1.py
| | | | | -- example2.py
```

Thus, the path of the pyFormex package (i.e. the ‘pyformex’ directory containing the pyFormex ‘main.py’ source file and having at least subdirectories ‘gui’, ‘opengl’, ‘plugins’) is ‘/home/user/pyformex/pyformex’. We have to add its parent path (‘/home/user/pyformex’) to the front of the Python paths to search for packages and modules. Furthermore, suppose we have our Python scripts in a directory ‘/home/user/apps/example1.py’.

Example1

The contents of ‘example1.py’ looks like this:

```
# Set path to import pyFormex
import sys
sys.path.insert(0, '/home/user/pyformex')

# Import the pyFormex with its full scripting language
from pyformex.script import *

# Show that we can do some pyFormex operations
a = array([[1,2,3],[4,5,6]])
print(a)
b = growAxis(a,2)
print(b)
```

It can be run with the command:

```
python example1.py
```

and produces the result:

```
[[1 2 3]
 [4 5 6]]
[[1 2 3 0 0]
 [4 5 6 0 0]]
```

Example2

Instead of hardcoding the pyFormex package path inside the script, you can set it in the PYTHONPATH environment variable. Also, in this example we do not import everything from the pyFormex scripting language, only the few things we need:

```
# Import some required modules
import numpy
from pyformex import arraytools

# Show that we can do some pyFormex operations
a = numpy.array([[1,2,3],[4,5,6]])
print(a)
b = arraytools.growAxis(a,2,axis=0)
print(b)
```

This script can now be run with a command:


```
PYTHONPATH=/home/user/pyformex python example2.py
```

and produces the results:

```
[1 2 3]
[4 5 6]]
[[1 2 3]
 [4 5 6]
 [0 0 0]
 [0 0 0]]
```

4.4.4 Caveats

- It is possible to use relative paths for the pyFormex package path. Be aware though that these may not work if you execute the python command from a directory that is actually a symlink.

4.4.5 Limitations

Currently you can not set the pyFormex command line options when not using the pyformex command.

Loading user preferences has not been tested yet. There is a function 'loadUserConfig' in pyformex.main (untested).

Using (parts of) pyFormex other than through the pyformex command has not been thoroughly tested yet. While the basic functionality should likely work, the use of some complex classes and modules may raise some problems.

Warnings can not be silenced (unless you load the user preferences and disable the warnings from the GUI first).

4.5 The Graphical User Interface

While the GUI has become much more elaborate in recent versions, its intention will never be to provide a fully interactive environment to operate on geometrical data. The main purpose of pyFormex will always remain to provide a framework for easily creating scripts to operate on geometries. Automization of otherwise tedious tasks is our primary focus.

The GUI mainly serves the following purposes:

- Display a structure in 3D. This includes changing the viewpoint, orientation and viewing distance. Thus you can interactively rotate, translate, zoom.
- Save a view in one of the supported image formats. Most of the images in this manual and on the website were created that way.
- Changing settings (though not everything can be changed through the GUI yet).
- Running scripts, possibly starting other programs and display their results.
- Interactively construct, select, change, import or export geometrical structures.

Unlike with most other geometrical modelers, in you usually design a geometrical model by writing a small script with the mathematical expressions needed to generate it. Any text editor will be suitable for this purpose. The main author of uses GNU Emacs, but this is just a personal preference. Any modern text editor will be fine, and the one you are accustomed with, will probably be the best choice. Since Python is the language used in scripts, a Python aware editor is highly preferable. It will highlight the syntax and help you with proper alignment (which is very important in Python). The default editors of KDE and Gnome and most other modern editors will certainly do well. A special

purpose editor integrated into the GUI is on our TODO list, but it certainly is not our top priority, because general purpose editors are already adequate for our purposes.

Learning how to use is best done by studying and changing some of the examples. We suggest that you first take a look at the examples included in the GUI and select those that display geometrical structures and/or use features that look interesting to you. Then you can study the source code of those examples and see how the structures got built and how the features were obtained. Depending on your installation and configuration, the examples can be found under the *Examples* or *Scripts* main menu item. The examples may appear classified according to themes or keywords, which can help you in selecting appropriate examples.

Selecting an example from the menu will normally execute the script, possibly ask for some interactive input and display the resulting geometrical structure. To see the source of the script, choose the *File* → *Edit Script* menu item.

Before starting to write your own scripts, you should probably get acquainted with the basic data structures and instructions of Python, NumPy and pyFormex. You can do this by reading the *pyFormex tutorial*.

4.5.1 Starting the GUI

You start the pyFormex GUI by entering the command `pyformex` in a terminal window. Depending on your installation, you may also have a panel or menu button on your desktop from which you can start the graphical interface by a simple mouse click. When the main window appears, it will look like the one shown in the figure *The pyFormex main window*. Your window manager will most likely have put some decorations around it, but these are very much OS and window manager dependent and are therefore not shown in the figure.

Finally, you can also start the GUI with the instruction `startGUI()` from a pyFormex script executed in non-GUI mode.

4.5.2 Basic use of the GUI

As is still in its infancy, the GUI is subject to frequent changes and it would make no sense to cover here every single aspect of it. Rather we will describe the most important functions, so that users can quickly get used to working with. Also we will present some of the more obscure features that users may not expect but yet might be very useful.

The window (figure *The pyFormex main window*) comprises 5 parts. From top to bottom these are:

1. the menu bar,
2. the tool bar,
3. the canvas (empty in the figure),
4. the message board, and
5. the status bar.

Many of these parts look and work in a rather familiar way. The menu bar gives access to most of the GUI features through a series of pull-down menus. The most important functions are described in following sections.

The toolbar contains a series of buttons that trigger actions when clicked upon. This provides an easier access to some frequently used functions, mainly for changing the viewing parameters.

The canvas is a drawing board where your scripts can show the created geometrical structures and provide them with full 3D view and manipulation functions. This is obviously the most important part of the GUI, and even the main reason for having a GUI at all. However, the contents of the canvas is often mainly created by calling drawing functions from a script. This part of the GUI is therefore treated in full detail in a separate chapter.

In the message board displays informative messages, requested results, possibly also errors and any text that your script writes out.

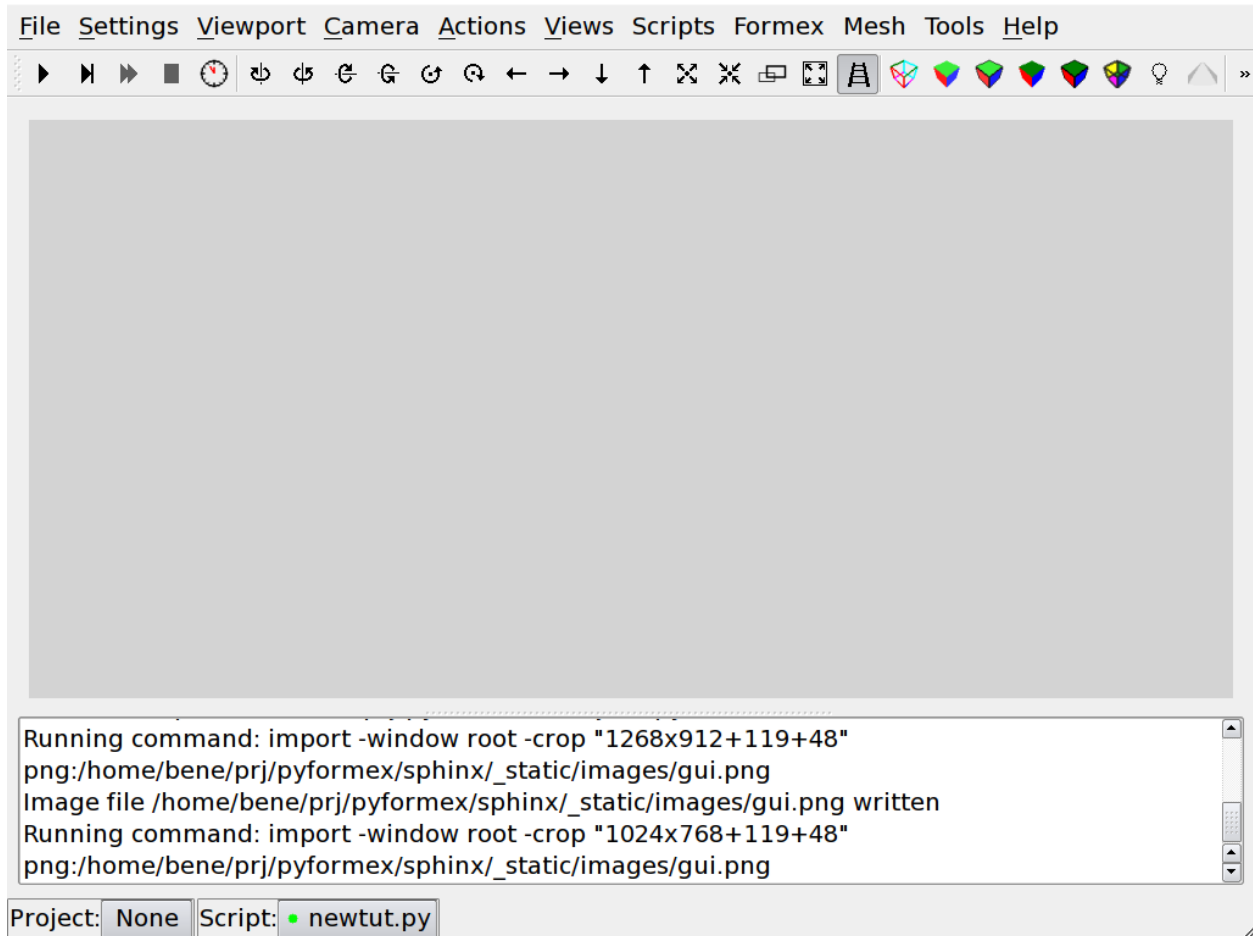


Fig. 1: The pyFormex main window

The status bar shows the current status of the GUI. For now this only contains the filename of the current script and an indicator if this file has been recognized as a script (happy face) or not (unhappy face).

Between the canvas and the message board is a splitter allowing resizing the parts of the window occupied by the canvas and message board. The mouse cursor changes to a vertical resizing symbol when you move over it. Just click on the splitter and move the mouse up or down to adjust the canvas/message board to your likings.

The main window can be resized in the usual ways.

4.5.3 The *File* menu

4.5.4 The *Settings* menu

Many aspects of the pyFormex GUI are configurable to suit better to the user's likings. This customization can be made persistent by storing it in a configuration file. This is explained in *Configuring pyFormex*.

Many of the configuration variables however can be changed interactively from the GUI itself.

- *Settings* → *Commands*: Lets you change the external command name used for the editor, the HTML/text file viewer and the HTML browser. Each of these values should be an executable command accepting a file name as parameter.

4.5.5 The viewport menu

4.5.6 Mouse interactions on the canvas

A number of actions can be performed by interacting with the mouse on the canvas. The default initial bindings of the mouse buttons are shown in the following table.

Rotate, pan and zoom

You can use the mouse to dynamically rotate, pan and zoom the scene displayed on the canvas. These actions are bound to the left, middle and right mouse buttons by default. Pressing the corresponding mouse button starts the action; moving the mouse with depressed button continuously performs the actions, until the button is released. During picking operations, the mouse bindings are changed. You can however still start the interactive rotate, pan and zoom, by holding down the ALT key modifier when pressing the mouse button.

rotate Press the left mouse button, and while holding it down, move the mouse over the canvas: the scene will rotate. Rotating in 3D by a 2D translation of the mouse is a fairly complex operation:

- Moving the mouse radially with respect to the center of the screen rotates around an axis lying in the screen and perpendicular to the direction of the movement.
- Moving tangentially rotates around an axis perpendicular to the screen (the screen z-axis), but only if the mouse was not too close to the center of the screen when the button was pressed.

Try it out on some examples to get a feeling of the working of mouse rotation.

pan Pressing the middle (or simultaneous left+right) mouse button and holding it down, will move the scene in the direction of the mouse movement. Because this is implemented as a movement of the camera in the opposite direction, the perspective of the scene may change during this operation.

zoom Interactive zooming is performed by pressing the right mouse button and move the mouse while keeping the button depressed. The type of zoom action depends on the direction of the movement:

- horizontal movement zooms by camera lens angle,

- vertical movement zooms by changing camera distance.

The first mode keeps the perspective, the second changes it. Moving right and upzooms in, left and down zooms out. Moving diagonally from upper left to lower right more or less keeps the image size, while changing the perspective.

Interactive selection

During picking operations, the mouse button functionality is changed. Click and drag the left mouse button to create a rectangular selection region on the canvas. Depending on the modifier key that was used when pressing the button, the selected items will be:

NONE set as the current selection;

SHIFT added to the currentselection;

CTRL removed from the current selection.

Clicking the right mouse button finishes the interactive selection mode.

During selection mode, using the mouse buttons in combination with the ALT modifier key will still activate the default mouse functions (rotate/pan/zoom).

4.5.7 Customizing the GUI

Some parts of the GUI can easily be customized by the user. The appearance (widget style and fonts) can be changed from the preferences menu. Custom menus can be added by executing a script. Both are very simple tasks even for beginning users. They are explained shortly hereafter.

Experienced users with a sufficient knowledge of Python and GUI building with Qt can of course use all their skills to tune every single aspect of the GUI according to their wishes. If you send us your modifications, we might even include them in the official distribution.

Changing the appearance of the GUI

Adding your scripts in a menu

By default, pyFormex adds all the example scripts that come with the distribution in a single menu accessible from the menubar. The scripts in this menu are executed by selecting them from the menu. This is easier than opening the file and then executing it.

You can customize this scripts menu and add your own scripts directories to it. Just add a line like the following to the main section of your `.pyformexc` configuration file: — `scriptdirs = [('Examples', None), ('My Scripts', '/home/me/myscripts'), ('More', '/home/me/morescripts')]`

Each tuple in this list consists of a string to be used as menu title and the absolute path of a directory with your scripts. From each such directory all the files that are recognized as scripts and do not start with a `.` or `_`, will be included in the menu. If your `scriptdirs` setting has only one item, the menu item will be created directly in the menubar. If there are multiple items, a top menu named `Scripts` will be created with submenus for each entry.

Notice the special entry for the examples supplied with the distribution. You do not specify the directory where the examples are: you would probably not even know the correct path, and it could change when a new version of is installed. As long as you keep its name to `Examples` (in any case: `examples` would work as well) and the path set to `None` (unquoted!), will itself try to detect the path to the installed examples.

Adding custom menus

When you start using for serious work, you will probably run into complex scripts built from simpler subtasks that are not necessarily always executed in the same order. While the scripting language offers enough functions to ask the user which parts of the script should be executed, in some cases it might be better to extend the GUI with custom menus to execute some parts of your script.

For this purpose, the `gui.widgets` module of provides a `Menu` widget class. Its use is illustrated in the example `Stl.py`.

4.6 pyFormex scripting

While the pyFormex GUI provides some means for creating and transforming geometry, its main purpose and major strength is the powerful scripting language. It offers you unlimited possibilities to do whatever you want and to automatize the creation of geometry up to an unmatched level.

Currently pyFormex provides two mechanisms to execute user applications: as a *script*, or as an *app*. The main menu bar of the GUI offers two menus reflecting this. While there are good reasons (of both historical and technical nature) for having these two mechanisms, the first time user will probably not be interested in studying the precise details of the differences between the two models. It suffices to know that the script model is well suited for small, quick applications, e.g. often used to test out some ideas. As your application grows larger and larger, you will gain more from the *app* model. Both require that the source file(s) be correctly formatted Python scripts. By obeying some simple code structuring rules, it is even possible to write source files that can be executed under either of the two models. The pyFormex template script as well as the many examples coming with pyFormex show how to do it.

4.6.1 Scripts

A pyFormex *script* is a simple Python source script in a file (with `.py` extension), which can be located anywhere on the filesystem. The script is executed inside pyFormex with an `exec` statement. pyFormex provides a collection of global variables to these scripts: the globals of module `gui.draw` if the script is executed with the GUI, or those from the module `script` if pyformex was started with `--nogui`. Also, the global variable `__name__` is set to either `'draw'` or `'script'`, accordingly. The automatic inclusion of globals has the advantage that the first time user has a lot of functionality without having to know what he needs to import.

Every time the script is executed (e.g. using the start or rerun button), the full source code is read, interpreted, and executed. This means that changes made to the source file will become directly available. But it also means that the source file has to be present. You can not run a script from a compiled (`.pyc`) file.

4.6.2 Apps

A pyFormex *app* is a Python module. It is usually also provided a Python source file (`.py`), but it can also be a compiled (`.pyc`) file. The app module is loaded with the `import` statement. To allow this, the file should be placed in a directory containing an `'__init__.py'` file (marking it as a Python package directory) and the directory should be on the pyFormex search path for modules (which can be configured from the GUI App menu).

Usually an app module contains a function named `'run'`. When the application is started for the first time (in a session), the module is loaded and the `'run'` function is executed. Each following execution will just apply the `'run'` function again.

When loading module from source code, it gets compiled to byte code which is saved as a `.pyc` file for faster loading next time. The module is kept in memory until explicitly removed or reloaded (another `import` does not have any effect). During the loading of a module, executable code placed in the outer scope of the module is executed. Since this will only happen on first execution of the app, the outer level should be seen as initialization code for your application.

The 'run' function defines what the application needs to perform. It can be executed over and over by pushing the 'PLAY' button. Making changes to the app source code will not have any effect, because the module loaded in memory is not changed. If you need the module to be reloaded and the initialization code to be rerun use the 'RERUN' button: this will reload the module and execute 'run'.

While a script is executed in the environment of the 'gui.draw' (or 'script') module, an app has its own environment. Any definitions needed should therefore be imported by the module.

4.6.3 Common script/app template

The template below is a common structure that allows this source to be used both as a script or as an app, and with almost identical behavior.

```

1  #
2  ##
3  ##  Copyright (C) 2011 John Doe (j.doe@somewhere.org)
4  ##  Distributed under the GNU General Public License version 3 or later.
5  ##
6  ##  This program is free software: you can redistribute it and/or modify
7  ##  it under the terms of the GNU General Public License as published by
8  ##  the Free Software Foundation, either version 3 of the License, or
9  ##  (at your option) any later version.
10 ##
11 ##  This program is distributed in the hope that it will be useful,
12 ##  but WITHOUT ANY WARRANTY; without even the implied warranty of
13 ##  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
14 ##  GNU General Public License for more details.
15 ##
16 ##  You should have received a copy of the GNU General Public License
17 ##  along with this program.  If not, see http://www.gnu.org/licenses/.
18 ##
19
20 """pyFormex Script/App Template
21
22 This is a template file to show the general layout of a pyFormex
23 script or app.
24
25 A pyFormex script is just any simple Python source code file with
26 extension '.py' and is fully read and execution at once.
27
28 A pyFormex app can be a '.py' or '.pyc' file, and should define a function
29 'run()' to be executed by pyFormex. Also, the app should import anything that
30 it needs.
31
32 This template is a common structure that allows the file to be used both as
33 a script or as an app, with almost identical behavior.
34
35 For more details, see the user guide under the `Scripting` section.
36
37 The script starts by preference with a docstring (like this),
38 composed of a short first line, then a blank line and
39 one or more lines explaining the intention of the script.
40
41 If you distribute your script/app, you should set the copyright holder
42 at the start of the file and make sure that you (the copyright holder) has
43 the intention/right to distribute the software under the specified
44 copyright license (GPL3 or later).

```

(continues on next page)

(continued from previous page)

```
45 """
46 # This helps in getting same code working with both Python2 and Python3
47 from __future__ import absolute_import, division, print_function
48
49 # The pyFormex modeling language is defined by everything in
50 # the gui.draw module (if you use the GUI). For execution without
51 # the GUI, you should import from pyformex.script instead.
52 from pyformex.gui.draw import *
53
54 # Definitions
55 def run():
56     """Main function.
57
58     This is automatically executed on each run of an app.
59     """
60     print("This is the pyFormex template script/app")
61
62
63 # Code in the outer scope:
64 # - for an app, this is only executed on loading (module initialization).
65 # - for a script, this is executed on each run.
66
67 print("This is the initialization code of the pyFormex template script/app")
68
69 # The following is to make script and app behavior alike
70 # When executing a script in GUI mode, the global variable __name__ is set
71 # to 'draw', thus the run method defined above will be executed.
72
73 if __name__ == '__draw__':
74     print("Running as a script")
75     run()
76
77
78 # End
```

The script/app source starts by preference with a docstring, consisting of a short first line, then a blank line and one or more lines explaining the intention and working of the script/app.

4.7 Modeling Geometry with pyFormex

Warning: This document is under construction!

Abstract

This chapter explains the different geometrical models in pyFormex, how and when to use them, how to convert between them, how to import and export them in various formats.

4.7.1 Introduction

Everything is geometry

In everyday life, geometry is ubiquitous. Just look around you: all the things you see, whether objects or living organisms or natural phenomena like clouds, they all have a shape or geometry. This holds for all concrete things, even if they are ungraspable, like a rainbow, or have no defined or fixed shape, like water. The latter evidently takes the shape of its container. Only abstract concepts do not have a geometry. Any material thing has though¹, hence our claim: everything is geometry.

Since geometry is such an important aspect of everybody's life, one would expect that it would take an important place in education (base as well as higher). Yet we see that in the educational system of many developed countries, attention for geometry has waned during the last decades. Important for craftsmen, technician, engineer, designer, artist

We will give some general ideas about geometry, but do not pretend to be a full geometry course. Only concepts needed for or related to modeling with pyFormex.

We could define the geometry of an object as the space it occupies. In our three-dimensional world, any object is also 3D. Some objects however have very small dimensions in one or more directions (e.g. a thin wire or a sheet of paper). It may be convenient then to model these only in one or two dimensions.²

Concrete things also have a material. Things going wrong is mostly mechanical: geometry/material

4.7.2 The Formex model

4.7.3 The Mesh model

4.7.4 The TriSurface model

4.7.5 The Curve model

4.7.6 Subclassing Geometry

The `__init__` method of the derived class should at least call `Geometry.__init__(self)` and then assign a `Coords` to `self.coords`. Furthermore, the class should override the `nelems()` method. Then a newly created instance of the subclass will at least have these attributes:

Derived classes can (and in most cases should) declare a method `_set_coords(coords)` returning an object that is identical to the original, except for its `coords` being replaced by new ones with the same array shape.

The `Geometry` class provides two possible default implementations:

- `_set_coords_inplace` sets the `coords` attribute to the provided new `coords`, thus changing the object itself, and returns itself,
- `_set_coords_copy` creates a deep copy of the object before setting the `coords` attribute. The original object is unchanged, the returned one is the changed copy.

When using the first method, a statement like `B = A.scale(0.5)` will result in both `A` and `B` pointing to the same scaled object, while with the second method, `A` would still be the untransformed object. Since the latter is in line with the design philosophy of pyFormex, it is set as the default `_set_coords` method. Many derived classes that are part of pyFormex override this default and implement a more efficient copy method.

Derived classes should implement the `_select` method

```
def _select(self,selected,**kargs): """Return a Formex only holding the selected elements.
```

¹ We obviously look here at matter in the way we observe it with our senses (visual, tactile) and not in a quantum-mechanics way.

² Mathematically we can also define geometry with higher dimensionality than 3, but this is of little practical use.

The kargs can hold optional arguments: compact = True/False

if the Coords modell can hold unused points that can be removed by compaction (the case with Mesh)

4.7.7 Analytical models

4.8 The Canvas

4.8.1 Introduction

When you have created a nice and powerful script to generate a 3D structure, you will most likely want to visually inspect that you have indeed created that what you intended. Usually you even will want or need to see intermediate results before you can continue your development. For this purpose the GUI offers a canvas where structures can be drawn by functions called from a script and interactively be manipulated by menus options and toolbar buttons.

The 3D drawing and rendering functionality is based on OpenGL. Therefore you will need to have OpenGL available on your machine, either in hardware or software. Hardware accelerated OpenGL will of course speed up and ease operations.

The drawing canvas of actually is not a single canvas, but can be split up into multiple viewports. They can be used individually for drawing different items, but can also be linked together to show different views of the same scene. The details about using multiple viewports are described in section *Multiple viewports*. The remainder of this chapter will treat the canvas as if it was a single viewport.

distinguishes three types of items that can be drawn on the canvas: actors, marks and decorations. The most important class are the actors: these are 3D geometrical structures defined in the global world coordinates. The 3D scene formed by the actors is viewed by a camera from a certain position, with a certain orientation and lens. The result as viewed by the camera is shown on the canvas. The scripting language and the GUI provide ample means to move the camera and change the lens settings, allowing translation, rotation, zooming, changing perspective. All the user needs to do to get an actor displayed with the current camera settings, is to add that actor to the scene. There are different types of actors available, but the most important is the FormexActor: a graphical representation of a Formex. It is so important that there is a special function with lots of options to create a FormexActor and add it to the OpenGL scene. This function, draw(), will be explained in detail in the next section.

The second type of canvas items, marks, differ from the actors in that only their position in world coordinates is fixed, but not their orientation. Marks are always drawn in the same way, irrespective of the camera settings. The observer will always have the same view of the item, though it can (and will) move over the canvas when the camera is changed. Marks are primarily used to attach fixed attributes to certain points of the actors, e.g. a big dot, or a text displaying some identification of the point.

Finally, offers decorations, which are items drawn in 2D viewport coordinates and unchangeably attached to the viewport. This can e.g. be used to display text or color legends on the view.

4.8.2 Drawing a Formex

The most important action performed on the canvas is the drawing of a Formex. This is accomplished with the draw() function. If you look at the reference page of the draw() function, the number of arguments looks frightening. However, most of these arguments have sensible default values, making the access to drawing functionality easy even for beginners. To display your created Formex F on the screen, a simple draw(F) will suffice in many cases.

If you draw several Formices with subsequent draw() commands, they will clutter the view. You can use the clear() instruction to wipe out the screen before drawing the next one. If you want to see them together in the same view, you can use different colors to differentiate. Color drawing is as easy as draw(F,color='red'). The color specification can take various forms. It can be a single color or an array of colors or even an array of indices in a color table. In the

latter case you use `draw(F,color=indices,colormap=table)` to draw the Formex. If multiple colors are specified, each element of the Formex will be drawn with the corresponding color, and if the color array (or the color indices array) has less entries than the number of elements, it is wrapped around.

A single color entry can be specified by a string ('red') or by a triple of RGB values in the range 0.0..1.0 (e.g. red is (1.0,0.0,0.0)) or a triplet of integer values in the range 0..255 or a hexadecimal string ('#FF0000') or generally any of the values that can be converted by the `colors.glColor()` function to a triplet of RGB values.

If no color is specified and your Formex has no properties, will draw it with the current drawing color. If the Formex has properties, will use the properties as a color index into the specified color map or a (configurable) default color map.

There should be some examples here. Draw object(s) with specified settings and direct camera to it.

4.8.3 Viewing the scene

Once the Formex is drawn, you can manipulate it interactively using the mouse: you can rotate, translate and zoom with any of the methods described in *Mouse interactions on the canvas*. You should understand though that these methods do not change your Formex, but only how it is viewed by the observer.

Our drawing board is based on OpenGL. The whole OpenGL drawing/viewing process can best be understood by making the comparison with the set of a movie, in which actors appear in a 3D scene, and a camera that creates a 2D image by looking at the scene with a certain lens from some angle and distance. Drawing a Formex then is nothing more than making an actor appear on the scene. The OpenGL machine will render it according to the current camera settings.

Viewing transformations using the mouse will only affect the camera, but not the scene. Thus, if you move the Formex by sliding your mouse with button 3 depressed to the right, the Formex will *look like it is moving to the right*, though it is actually not: we simply move the camera in the opposite direction. Therefore in perspective mode, you will notice that moving the scene will not just translate the picture: its shape will change too, because of the changing perspective.

Using a camera, there are two ways of zooming: either by changing the focal length of the lens (lens zooming) or by moving the camera towards or away from the scene (dolly zooming). The first one will change the perspective view of the scene, while the second one will not.

The easiest way to set all camera parameters for properly viewing a scene is by just telling the direction from which you want to look, and let the program determine the rest of the settings itself. `even` goes a step further and has a number of built in directions readily available: 'top', 'bottom', 'left', 'right', 'front', 'back' will set up the camera looking from that direction.

4.8.4 Other canvas items

Actors

Marks

Decorations

4.8.5 Multiple viewports

Drawing in is not limited to a single canvas. You can create any number of canvas widgets laid out in an array with given number of rows or columns. The following functions are available for manipulating the viewports.

layout (*nyps=None, ncols=None, nrows=None*)

Set the viewports layout. You can specify the number of viewports and the number of columns or rows.

If a number of viewports is given, viewports will be added or removed to match the number requested. By default they are layed out rowwise over two columns.

If ncols is an int, viewports are laid out rowwise over ncols columns and nrows is ignored. If ncols is None and nrows is an int, viewports are laid out columnwise over nrows rows.

addViewport ()

Add a new viewport.

removeViewport ()

Remove the last viewport.

linkViewport (*vp*, *tovp*)

Link viewport *vp* to viewport *tovp*.

Both *vp* and *tovp* should be numbers of viewports. The viewport *vp* will now show the same contents as the viewport *tovp*.

viewport (*n*)

Select the current viewport. All drawing related functions will henceforth operate on that viewport.

This action is also implicitly called by clicking with the mouse inside a viewport.

4.9 Creating Images

Warning: This document still needs to be written!

Abstract

This chapter explains how to create image files of the renderings you created in pyFormex.

4.9.1 Save a rendering as image

A picture tells a thousand words

4.10 Using Projects

Warning: This document still needs to be written!

Abstract

This chapter explains how to use projects to make your work persistent. We will explain how to create new projects, how to add or remove data from the project and how to save and reopen project files.

4.10.1 What is a project

A pyFormex project is a persistent copy of some data created by pyFormex. These data are saved in a project file, which you can later re-open to import the data in another pyFormex session.

4.11 Assigning properties to geometry

As of version 0.7.1, the way to define properties for elements of the geometry has changed thoroughly. As a result, the property system has become much more flexibel and powerful, and can be used for Formex data structures as well as for TriSurfaces and Finite Element models.

With properties we mean any data connected with some part of the geometry other than the coordinates of its points or the structure of points into elements. Also, values that can be calculated purely from the coordinates of the points and the structure of the elements are usually not considered properties.

Properties can e.g. define material characteristics, external loading and boundary conditions to be used in numerical simulations of the mechanics of a structure. The properties module includes some specific functions to facilitate assigning such properties. But the system is general enough to use it for any properties that you can think of.

Properties are collected in a `PropertyDB` object. Before you can store anything in this database, you need to create it. Usually, you will start with an empty database.

```
P = PropertyDB()
```

4.11.1 General properties

Now you can start entering property records into the database. A property record is a lot like a Python dict object, and thus it can contain nearly anything. It is implemented however as a `CascadingDict` object, which means that the key values are strings and can also be used as attributes to address the value. Thus, if `P` is a property record, then a field named `key` can either be addressed as `P['key']` or as `P.key`. This implementation was chosen for the convenience of the user, but has no further advantages over a normal dict object. You should not use any of the methods of Python's dict class as key in a property record: it would override this method for the object.

The property record has four more reserved (forbidden) keys: `kind`, `tag`, `set`, `setname` and `nr`. The `kind` and `nr` should never be set nor changed by the user. `kind` is used internally to distinguish among different kind of property records (see *Node properties*). It should only be used to extend the `PropertyDB` class with new kinds of properties, e.g. in subclasses. `nr` will be set automatically to a unique record number. Some application modules use this number for identification and to create automatic names for property sets.

The `tag`, `set` and `setname` keys are optional fields and can be set by the user. They should however only be used for the intended purposes explained hereafter, because they have a special meaning for the database methods and application modules.

The `tag` field can be used to attach an identification string to the property record. This string can be as complex as the user wants and its interpretation is completely left to the user. The `PropertyDB` class just provides an easy way to select the records by their tag name or by a set of tag names. The `set` and `setname` fields are treated further in *Using the set and setname fields*.

So let's create a property record in our database. The `Prop()` method does just that. It also returns the property record, so you can directly use it further in your code.

```
>>> Stick = P.Prop(color='green', name='Stick', weight=25, \
    comment='This could be anything: a gum, a frog, a usb-stick,...'))
>>> print Stick
```

(continues on next page)

(continued from previous page)

```

color = green
comment = This could be anything: a gum, a frog, a usb-stick,...
nr = 0
name = Stick
weight = 25

```

Notice the auto-generated *nr* field. Here's another example, with a tag:

```

>>> author = P.Prop(tag='author',name='Alfred E Neuman',\
                    address=CascadingDict({'street':'Krijgslaan',\
                    'city':'Gent','country':'Belgium'}))
>>> print author

nr = 1
tag = author
name = Alfred E Neuman
address =
  city = Gent
  street = Krijgslaan
  country = Belgium

```

This example shows that record values can be complex structured objects. Notice how the `CascadingDict` object is by default printed in a very readable layout, offsetting each lower level dictionary two more positions to the right.

The `CascadingDict` has yet another fine characteristic: if an attribute is not found in the toplevel, all values that are instances of `CascadingDict` or `Dict` (but not the normal Python dict) will be searched for the attribute. If needed, this searching is even repeated in the values of the next levels, and further on, thus cascading through all levels of `CascadingDict` structures until the attribute can eventually be found. The cascading does not proceed through values in a `Dict`. An attribute that is not found in any of the lower level dictionaries, will return a `None` value.

If you set an attribute of a `CascadingDict`, it is always set in the toplevel. If you want to change lower level attributes, you need to use the full path to it.

```

>>> print author.st
  Krijgslaan
>>> author.street = 'Voskenslaan'
>>> print author.street
  Voskenslaan
>>> print author.address.street
  Krijgslaan
>>> author.address.street = 'Wiemersdreef'
>>> print author.address.street
  Wiemersdreef
>>> author = P.Prop(tag='author',alias='John Doe',\
                    address={'city': 'London', 'street': 'Downing Street 10',\
                    'country': 'United Kingdom'})
>>> print author

nr = 2
tag = author
alias = John Doe
address = {'city': 'London', 'street': 'Downing Street 10',\
          'country': 'United Kingdom'}

```

In the examples above, we have given a name to the created property records, so that we could address them in the subsequent print and field assignment statements. In most cases however, it will be impractical and unnecessary to

give your records a name. They all are recorded in the `PropertyDB` database, and will exist as long as the database variable lives. There should be a way though to request selected data from that database. The `getProp()` method returns a list of records satisfying some conditions. The examples below show how it can be used.

```
>>> for p in P.getProp(rec=[0,2]):
    print p.name
Stick
John Doe
>>> for p in P.getProp(tag=['author']):
    print p.name
None
John Doe
>>> for p in P.getProp(attr=['name']):
    print p.nr
0
2
>>> for p in P.getProp(tag=['author'],attr=['name']):
    print p.name
John Doe
```

The first call selects records by number: either a single record number or a list of numbers can be specified. The second method selects records based on the value of their tag field. Again a single tag value or a list of values can be specified. Only those records having a 'tag' field matching any of the values in the list will be returned. The third selection method is based on the existence of some attribute names in the record. Here, always a list of attribute names is required. Records are returned that possess all the attributes in the list, independent from the value of those attributes. If needed, the user can add a further filtering based on the attribute values. Finally, as is shown in the last example, all methods of record selection can be combined. Each extra condition will narrow the selection further down.

4.11.2 Using the set and setname fields

In the examples above, the property records contained general data, not related to any geometrical object. When working with large geometrical objects (whether `Formex` or other type), one often needs to specify properties that only hold for some of the elements of the object.

The set can be used to specify a list of integer numbers identifying a collection of elements of the geometrical object for which the current property is valid. Absence of the set usually means that the property is assigned to all elements; however, the property module itself does not enforce this behavior: it is up to the application to implement it.

Any record that has a set field, will also have a setname field, whose value is a string. If the user did not specify one, a set name will be auto-generated by the system. The setname field can be used in other records to refer to the same set of elements without having to specify them again. The following examples will make this clear.

```
>>> P.Prop(set=[0,1,3],setname='green_elements',color='green')
    P.Prop(setname='green_elements',transparent=True)

>>> a = P.Prop(set=[0,2,4,6],thickness=3.2)
    P.Prop(setname=a.setname,material='steel')

>>> for p in P.getProp(attr=['setname']):
    print p

color = green
nr = 3
set = [0 1 3]
setname = green_elements
```

(continues on next page)

(continued from previous page)

```

nr = 4
transparent = True
setname = green_elements

nr = 5
set = [0 2 4 6]
setname = Set_5
thickness = 3.2

nr = 6
material = steel
setname = Set_5

```

In the first case, the user specifies a setname himself. In the second case, the auto-generated name is used. As a convenience, the user is allowed to write `set=name` instead of `setname=name` when referring to an already defined set.

```

>>> P.Prop(set='green_elements',transparent=False)
      for p in P.getProp(attr=['setname']):
          if p.setname == 'green_elements':
              print p.nr,p.transparent

3 None
4 True
7 False

```

Record 3 does not have the transparent attribute, so a value None is printed.

4.11.3 Specialized property records

The property system presented above allows for recording any kind of values. In many situations however we will want to work with a specialised and limited set of attributes. The main developers of e.g. often use the program to create geometrical models of structures of which they want to analyse the mechanical behavior. These numerical simulations (FEA, CFD) require specific data that support the introduction of specialised property records. Currently there are two such property record types: node properties (see *Node properties*), which are attributed to a single point in space, and element properties (*Element properties*), which are attributed to a structured collection of points.

Special purpose properties are distinguished by their kind field. General property records have `kind=""`, node properties have `kind='n'` and `kind='e'` is set for element properties. Users can create their own specialised property records by using other value for the kind parameter.

4.11.4 Node properties

Node properties are created with the `nodeProp()` method, rather than the general `Prop()`. The kind field does not need to be set: it will be done automatically. When selecting records using the `getProp()` method, add a `kind='n'` argument to select only node properties.

Node properties will recognize some special field names and check the values for consistency. Application plugins such as the Abaqus input file generator depend on these property structure, so the user should not mess with them. Currently, the following attributes are in use:

load A concentrated load at the node. This is a list of 6 items: three force components in axis directions and three force moments around the axes: `[F_0, F_1, F_2, M_0, M_1, M_2]`.

bound A boundary condition for the nodal displacement components. This can be defined in 2 ways:

- as a list of 6 items [u_0, u_1, u_2, r_0, r_1, r_2]. These items have 2 possible values:
 - 0 The degree of freedom is not restrained.
 - 1 The degree of freedom is restrained.
- as a string. This string is a standard boundary type. Abaqus will recognize the following strings:
 - PINNED
 - ENCASTRE
 - XSYMM
 - YSYMM
 - ZSYMM
 - XASYMM
 - YASYMM
 - ZASYMM

displacement Prescribed displacements. This is a list of tuples (i,v), where i is a DOF number (1..6) and v is the prescribed value for that DOF.

coords The coordinate system which is used for the definition of cload, bound and displ fields. It should be a CoordSys object.

Some simple examples:

```
P.nodeProp(cload=[5, 0, -75, 0, 0, 0])
P.nodeProp(set=[2, 3], bound='pinned')
P.nodeProp(5, displ=[1, 0.7])
```

The first line sets a concentrated load all the nodes, the second line sets a boundary condition 'pinned' on nodes 2 and 3. The third line sets a prescribed displacement on node 5 with value 0.7 along the first direction. The first positional argument indeed corresponds to the 'set' attribute.

Often the properties are computed and stored in variables rather than entered directly.

```
P1 = [ 1.0, 1.0, 1.0, 0.0, 0.0, 0.0 ]
P2 = [ 0.0 ] * 3 + [ 1.0 ] * 3
B1 = [ 1 ] + [ 0 ] * 5
CYL = CoordSystem('cylindrical', [0, 0, 0, 0, 0, 1])
P.nodeProp(bound=B1, csys=CYL)
```

The first two lines define two concentrated loads: P1 consists of three point loads in each of the coordinate directions; P2 contains three force moments around the axes. The third line specifies a boundary condition where the first DOF (usually displacement in *x*-direction) is constrained, while the remaining 5 DOF's are free. The next line defines a local coordinate system, in this case a cylindrical coordinate system with axis pointing from point [0., 0., 0.] to point [0., 0., 1.]. The last line

To facilitate property selection, a tag can be added.

```
nset1 = P.nodeProp(tag='loadcase 1', set=[2, 3, 4], cload=P1).nr
P.nodeProp(tag='loadcase 2', set=Nset(nset1), cload=P2)
```

The last two lines show how you can avoid duplication of sets in multiple records. The same set of nodes should receive different concentrated load values for different load cases. The load case is stored in a tag, but duplicating the set definition could become wasteful if the sets are large. Instead of specifying the node numbers of the set directly, we can pass a string setting a set name. Of course, the application will need to know how to interpret the set names.

Therefore the property module provides a unified way to attach a unique set name to each set defined in a property record. The name of a node property record set can be obtained with the function `Nset(nr)`, where `nr` is the record number. In the example above, that value is first recorded in `nset1` and then used in the last line to guarantee the use of the same set as in the property above.

4.11.5 Element properties

The `elemProp()` method creates element properties, which will have their `kind` attribute set to 'e'. When selecting records using the `getProp()` method, add the `kind='e'` argument to get element properties.

Like node properties, element property records have a number of specialize fields. Currently, the following ones are recognized by the Abaqus input file generator.

eltype This is the single most important element property. It sets the element type that will be used during the analysis. Notice that a Formex object also may have an `eltype` attribute; that one however is only used to describe the type of the geometric elements involved. The element type discussed here however may also define some other characteristics of the element, like the number and type of degrees of freedom to be used in the analysis or the integration rules to be used. What element types are available is dependent on the analysis package to be used. Currently, does not do any checks on the element type, so the simulation program's own element designation may be used.

section The section properties of the element. This should be an `ElemSection` instance, grouping material properties (like Young's modulus) and geometrical properties (like plate thickness or beam section).

dload A distributed load acting on the element. The value is an `ElemLoad` instance. Currently, this can include a label specifying the type of distributed loading, a value for the loading, and an optional amplitude curve for specifying the variation of a time dependent loading.

4.11.6 Property data classes

The data collected in property records can be very diverse. At times it can become quite difficult to keep these data consistent and compatible with other modules for further processing. The property module contains some data classes to help you in constructing appropriate data records for Finite Element models. The `FeAbq` module can currently interpret the following data types.

`CoordSystem` defines a local coordinate system for a node. Its constructor takes two arguments:

- a string defining the type of coordinate system, either 'Rectangular', 'Cylindrical' or 'Spherical' (the first character suffices), and
- a list of 6 coordinates, specifying two points A and B. With 'R', A is on the new *x*-axis and B is on the new 'y' axis. With 'C' and 'S', AB is the axis of the cylindrical/spherical coordinates.

Thus, `CoordSystem('C', [0., 0., 0., 0., 0., 1.])` defines a cylindrical coordinate system with the global *z* as axis.

`ElemLoad` is a distributed load on an element. Its constructor takes two arguments:

- a label defining the type of loading,
- a value for the loading,
- optionally, the name of an amplitude curve.

E.g., `ElemLoad('PZ', 2.5)` defines a distributed load of value 2.5 in the direction of the *z*-axis.

`ElemSection` can be used to set the material and section properties on the elements. It can hold:

- a section,

- a material,
- an optional orientation,
- an optional connector behavior,
- a sectiontype (deprecated). The sectiontype should preferably be set together with the other section parameters.

An example:

```
>>> steel = {
    'name': 'steel',
    'young_modulus': 207000,
    'poisson_ratio': 0.3,
    'density': 0.1,
}
>>> thin_plate = {
    'name': 'thin_plate',
    'sectiontype': 'solid',
    'thickness': 0.01,
    'material': 'steel',
}
>>> P.elemProp(eltype='CPS3', section=ElemSection(section=thin_plate, material=steel))
```

First, a material is defined. Then a thin plate section is created, referring to that material. The last line creates a property record that will attribute this element section and an element type 'CPS3' to all elements.

Exporting to finite element programs

4.12 Using Widgets

Warning: This document still needs to be written!

Abstract

This chapter gives an overview of the specialized widgets in pyFormex and how to use them to quickly create a specialized graphical interface for you application.

The pyFormex Graphical User Interface (GUI) is built on the QT4 toolkit, accessed from Python by PyQt4. Since the user has full access to all underlying libraries, he can use any part from QT4 to construct the most sophisticated user interface and change the pyFormex GUI like he wants and sees fit. However, properly programming a user interface is a difficult and tedious task, and many normal users do not have the knowledge or time to do this. pyFormex provides a simplified framework to access the QT4 tools in a way that complex and sophisticated user dialogs can be built with a minimum effort. User dialogs are create automatically from a very limited input. Specialized input widgets are included dependent on the type of input asked from the user. And when this simplified framework falls short for your needs, you can always access the QT4 functions directly to add what you want.

4.12.1 The askItems functions

The `askItems()` function reduces the effort needed to create an interactive dialog asking input data from the user.

4.12.2 The input dialog

4.12.3 The user menu

4.12.4 Other widgets

4.13 pyFormex plugins

Abstract

This chapter describes how to create plugins for and documents some of the standard plugins that come with the pyFormex distribution.

4.13.1 What are plugins?

From its inception was intended to be easily expandable. Its open architecture allows educated users to change the behavior of and to extend its functionality in any way they want. There are no fixed rules to obey and there is no registrar to accept and/or validate the provided plugins. In , any set of functions that are not an essential part of can be called a ‘plugin’, if its functionality can usefully be called from elsewhere and if the source can be placed inside the distribution.

Thus, we distinct plugins from the vital parts of which comprehend the basic data types (Formex), the scripting facilities, the (OpenGL) drawing functionality and the graphical user interface. We also distinct plugins from normal (example and user) scripts because the latter will usually be intended to execute some specific task, while the former will often only provide some functionality without themselves performing some actions.

To clarify this distinction, plugins are located in a separate subdirectory `plugins` of the tree. This directory should not be used for anything else.

The extensions provided by the plugins usually fall within one of the following categories:

Functional Extending the functionality by providing new data types and functions.

External Providing access to external programs, either by dedicated interfaces or through the command shell and file system.

GUI Extending the graphical user interface of .

The next section of this chapter gives some recommendations on how to structure the plugins so that they work well with . The remainder of the chapter discusses some of the most important plugins included with .

4.13.2 How to create a plugin.

4.14 Configuring pyFormex

Many aspects of pyFormex can be configured to better suit the user’s needs and likings. These can range from merely cosmetic changes to important extensions of the functionality. As is written in a scripting language and distributed as source, the user can change every single aspect of the program. And the GNU-GPL license under which the program is distributed guarantees that you have access to the source and are allowed to change it.

Most users however will only want to change minor aspects of the program, and would rather not have to delve into the source to do just that. Therefore we have gathered some items of that users might like to change, into separate files where they can easily be found. Some of these items can even be set interactively through the GUI menus.

Often users want to keep their settings between subsequent invocation of the program. To this end, the user preferences have to be stored on file when leaving the program and read back when starting the next time. While it might make sense to distinct between the user's current settings in the program and his default preferences, the current configuration system of (still under development) does not allow such distinction yet. Still, since the topic is so important to the user and the configuration system in is already quite complex, we thought it was necessary to provide already some information on how to configure. Be aware though that important changes to this system will likely occur.

4.14.1 Configuration files

On startup, reads its configurable data from a number of files. Often there are not less than four configuration files, read in sequence. The settings in each file being read override the value read before. The different configuration files used serve different purposes. On a typical GNU/Linux installation, the following files will be read in sequence:

- `PYFORMEX-INSTALL-PATH/pyformexcrc`: this file should never be changed, neither by the user nor the administrator. It is there to guarantee that all settings get an adequate default value to allow to correctly start up.
- `/etc/pyformexcrc`: this file can be used by the system administrator to make system-wide changes to the installation. This could e.g. be used to give all users at a site access to a common set of scripts or extensions.
- `./pyformexcrc`: this is where the user normally stores his own default settings.
- `CURRENT-DIR/.pyformexcrc`: if the current working directory from which is started contains a file named `.pyformexcrc`, it will be read too. This makes it possible to keep different configurations in different directories, depending on the purpose. Thus, one directory might aim at the use of for operating on triangulated surfaces, while another might be intended for pre- and post- processing of Finite Element models.
- Finally, the `--config=` command line option provides a way to specify another file with any name to be used as the last configuration file.

On exit, will store the changed settings on the last user configuration file that was read. The first two files mentioned above are system configuration files and will never be changed by the program. A user configuration file will be generated if none existed.

Warning: Currently, when pyFormex exits, it will just dump all the changed configuration (key,value) pairs on the last configuration file, together with the values it read from that file. pyFormex will not detect if any changes were made to that file between reading it and writing back. Therefore, the user should never edit the configuration files directly while pyFormex is still running. Always close the program first!

4.14.2 Syntax of the configuration files

All configuration files are plain text files where each non blank line is one of the following:

- a comment line, starting with a '#',
- a section header, of the form '[section-name]',
- a valid Python instruction.

The configuration file is organized in sections. All lines preceding the first section name refer to the general (unnamed) section.

Any valid Python source line can be used. This allows for quite complex configuration instructions, even importing Python modules. Any line that binds a value to a variable will cause a corresponding configuration variable to be set.

The user can edit the configuration files with any text editor, but should make sure the lines are legal Python. Any line can use the previously defined variables, even those defined in previously read files.

In the configuration files, the variable `pyformexdir` refers to the directory where was installed (and which is also reported by the `pyformex --whereami` command).

4.14.3 Configuration variables

Many configuration variables can be set interactively from the GUI, and the user may prefer to do it that way. Some variables however can not (yet) be set from th GUI. And real programmers may prefer to do it with an editor anyway. So here are some guidelines for setting some interesting variables. The user may take a look at the installed default configuration file for more examples.

General section

- `syspath = []`: Value is a list of path names that will be appended to the Python's `sys.path` variable on startup. This enables your scripts to import modules from other than default Python paths.
- `scriptdirs = [('Examples', examplesdir), ('MyScripts', myscriptsdire)]`: a list of tuples (name,path). On startup, all these paths will be scanned for scripts and these will be added in the menu under an item named name.
- `autorun = '.pyformex.startup'`: name of a script that will be executed on startup, before any other script (specified on the command line or started from the GUI).
- `editor = 'kedit'`: sets the name of the editor that will be used for editing pyformex scripts.
- `viewer = 'firefox'`: sets the name of the html viewer to be used to display the html help screens.
- `browser = 'firefox'`: sets the name of the browser to be used to access the website.
- `uselib = False`: do not use the acceleration library. The default (True) is to use it when it is available.

Section [gui]

- `splash = 'path-to-splash-image.png'`: full path name of the image to be used as splash image on startup.
- `modebar = True`: adds a toolbar with the render mode buttons. Besides True or False, the value can also be one of 'top', 'bottom', 'left' or 'right', specifying the placement of the render mode toolbar at the specified window border. Any other value that evaluates True will make the buttons get included in the top toolbar.
- `viewbar = True`: adds a toolbar with different view buttons. Possiole values as explained above for modebar.
- `timeoutbutton = True`: include the timeout button in the toolbar. The timeout button, when depressed, will cause input widgets to time out after a prespecified delay time. This feature is still experimental.
- `plugins = ['surface_menu', 'formex_menu', 'tools_menu']`: a list of plugins to load on startup. This is mainly used to load extra (non-default) menus in the GUI to provide extended functionality. The named plugins should be available in the 'plugins' subdirectory of the installation. To autoload user extensions from a different place, the autorun script can be used.

PYFORMEX EXAMPLE SCRIPTS

Warning: This document still needs some cleanup!

Sometimes you learn quicker from studying an example than from reading a tutorial or user guide. To help you we have created this collection of annotated examples. Beware that the script texts presented in this document may differ slightly from the corresponding example coming with the pyFormex distribution.

5.1 WireStent

To get acquainted with the modus operandi of pyFormex, the `WireStent.py` script is studied step by step. The lines are numbered for easy referencing, but are not part of the script itself.

```
1 #   *** pyformex ***
2 ##
3 ##   This file is part of pyFormex 1.0.7 (Mon Jun 17 12:20:39 CEST 2019)
4 ##   pyFormex is a tool for generating, manipulating and transforming 3D
5 ##   geometrical models by sequences of mathematical operations.
6 ##   Home page: http://pyformex.org
7 ##   Project page: http://savannah.nongnu.org/projects/pyformex/
8 ##   Copyright 2004-2019 (C) Benedict Verhegghe (benedict.verhegghe@ugent.be)
9 ##   Distributed under the GNU General Public License version 3 or later.
10 ##
11 ##   This program is free software: you can redistribute it and/or modify
12 ##   it under the terms of the GNU General Public License as published by
13 ##   the Free Software Foundation, either version 3 of the License, or
14 ##   (at your option) any later version.
15 ##
16 ##   This program is distributed in the hope that it will be useful,
17 ##   but WITHOUT ANY WARRANTY; without even the implied warranty of
18 ##   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
19 ##   GNU General Public License for more details.
20 ##
21 ##   You should have received a copy of the GNU General Public License
22 ##   along with this program. If not, see http://www.gnu.org/licenses/.
23 ##
24 """Wirestent.py
25
26 A pyFormex script to generate a geometrical model of a wire stent.
27
28 This version is for inclusion in the pyFormex documentation.
```

(continues on next page)

(continued from previous page)

```

29 """
30
31 from formex import *
32
33 class DoubleHelixStent:
34     """Constructs a double helix wire stent.
35
36     A stent is a tubular shape such as used for opening obstructed
37     blood vessels. This stent is made from sets of wires spiraling
38     in two directions.
39     The geometry is defined by the following parameters:
40     L : approximate length of the stent
41     De : external diameter of the stent
42     D : average stent diameter
43     d : wire diameter
44     be : pitch angle (degrees)
45     p : pitch
46     nx : number of wires in one spiral set
47     ny : number of modules in axial direction
48     ds : extra distance between the wires (default is 0.0 for
49         touching wires)
50     dz : maximal distance of wire center to average cylinder
51     nb : number of elements in a strut (a part of a wire between two
52         crossings), default 4
53     The stent is created around the z-axis.
54     By default, there will be connectors between the wires at each
55     crossing. They can be switched off in the constructor.
56     The returned formex has one set of wires with property 1, the
57     other with property 3. The connectors have property 2. The wire
58     set with property 1 is winding positively around the z-axis.
59     """
60     def __init__(self, De, L, d, nx, be, ds=0.0, nb=4, connectors=True):
61         """Create the Wire Stent."""
62         D = De - 2*d - ds
63         r = 0.5*D
64         dz = 0.5*(ds+d)
65         p = math.pi*D*tand(be)
66         nx = int(nx)
67         ny = int(round(nx*L/p)) # The actual length may differ a bit from L
68         # a single bumped strut, oriented along the x-axis
69         bump_z=lambdax: 1.-(x/nb)**2
70         base = Formex(pattern('1')).replic(nb,1.0).bump1(2,[0.,0.,dz],bump_z,0)
71         # scale back to size 1.
72         base = base.scale([1./nb,1./nb,1.])
73         # NE and SE directed struts
74         NE = base.shear(1,0,1.)
75         SE = base.reflect(2).shear(1,0,-1.)
76         NE.setProp(1)
77         SE.setProp(3)
78         # a unit cell of crossing struts
79         cell1 = (NE+SE).rosette(2,180)
80         # add a connector between first points of NE and SE
81         if connectors:
82             cell1 += Formex([[NE[0][0],SE[0][0]],2)
83             # and create its mirror
84             cell2 = cell1.reflect(2)
85             # and move both to appropriate place

```

(continues on next page)

(continued from previous page)

```

86     self.cell1 = cell1.translate([1.,1.,0.])
87     self.cell2 = cell2.translate([-1.,-1.,0.])
88     # the base pattern cell1+cell2 now has size [-2,-2]..[2,2]
89     # Create the full pattern by replication
90     dx = 4.
91     dy = 4.
92     F = (self.cell1+self.cell2).replic2(nx,ny,dx,dy)
93     # fold it into a cylinder
94     self.F = F.translate([0.,0.,r]).cylindrical(
95         dir=[2,0,1],scale=[1.,360./(nx*dx),p/nx/dy])
96     self.ny = ny
97
98     def all(self):
99         """Return the Formex with all bar elements."""
100         return self.F
101
102
103 if __name__ == "draw":
104
105     # show an example
106
107     wireframe()
108     reset()
109
110     D = 10.
111     L = 80.
112     d = 0.2
113     n = 12
114     b = 30.
115     res = askItems(['Diameter',D],
116                   ['Length',L],
117                   ['WireDiam',d],
118                   ['NWires',n],
119                   ['Pitch',b]))
120
121     if not res:
122         exit()
123
124     D = float(res['Diameter'])
125     L = float(res['Length'])
126     d = float(res['WireDiam'])
127     n = int(res['NWires'])
128     if (n % 2) != 0:
129         warning('Number of wires must be even!')
130         exit()
131     b = float(res['Pitch'])
132
133     H = DoubleHelixStent(D,L,d,n,b).all()
134     clear()
135     draw(H,view='iso')
136
137     # and save it in a lot of graphics formats
138     if ack("Do you want to save this image (in lots of formats) ?"):
139         for ext in [ 'bmp', 'jpg', 'pbm', 'png', 'ppm', 'xbm', 'xpm',
140                    'eps', 'ps', 'pdf', 'tex' ]:
141             image.save('WireStent.'+ext)
142

```

(continues on next page)

143 `# End`

As all pyFormex scripts, it starts with a comments line holding the word `pyformex` (line 1). This is followed more comments lines specifying the copyright and license notices. If you intend to distribute your scripts, you should give these certainly special consideration.

Next is a documentation string explaining the purpose of the script (lines 25-30). The script then starts by importing all definitions from other modules required to run the `WireStent.py` script (line 32).

Subsequently, the class `DoubleHelixStent` is defined which allows the simple use of the geometrical model in other scripts for e.g. parametric, optimization and finite element analyses of braided wire stents. Consequently, the latter scripts do not have to contain the wire stent geometry building and can be condensed and conveniently arranged. The definition of the class starts with a documentation string, explaining its aim and functioning (lines 34-60).

The constructor `__init__` of the `DoubleHelixStent` class requires 8 arguments (line 61):

- stent external diameter De (mm).
- stent length L (mm).
- wire diameter d (mm).
- Number of wires in one spiral set, i.e. wires with the same orientation, nx (-).
- Pitch angle β (deg).
- Extra radial distance between the crossing wires ds (mm). By default, ds is [0.0]mm for crossing wires, corresponding with a centre line distance between two crossing wires of exactly d .
- Number of elements in a strut, i.e. part of a wire between two crossings, nb (-). As every base element is a straight line, multiple elements are required to approximate the curvature of the stent wires. The default value of 4 elements in a strut is a good assumption.
- If `connectors=True`, extra elements are created at the positions where there is physical contact between the crossing wires. These elements are required to enable contact between these wires in finite element analyses.

The virtual construction of the wire stent structure is defined by the following sequence of four operations: (i) Creation of a nearly planar base module of two crossing wires; (ii) Extending the base module with a mirrored and translated copy; (iii) Replicating the extended base module in both directions of the base plane; and (iv) Rolling the nearly planar grid into the cylindrical stent structure, which is easily parametric adaptable.

5.1.1 Creating the base module

(lines 63-71)

Depending on the specified arguments in the constructor, the mean stent diameter D , the average stent radius r , the `bump` or curvature of the wires dz , the pitch p and the number of base modules in the axial direction ny are calculated with the following script. As the wire stent structure is obtained by braiding, the wires have an undulating course and the `bump dz` corresponds to the amplitude of the wave. If no extra distance ds is specified, there will be exactly one wire diameter between the centre lines of the crossing wires. The number of modules in the axial direction ny is an integer, therefore, the actual length of the stent model might differ slightly from the specified, desired length L . However, this difference has a negligible impact on the numerical results.

Of now, all parameters to describe the stent geometry are specified and available to start the construction of the wire stent. Initially a simple Formex is created using the `pattern()`-function: a straight line segment of length 1 oriented along the X-axis (East or 1-direction). The `replic()`-functionality replicates this line segment nb times with step 1 in the X-direction (0-direction). Subsequently, these nb line segments form a new Formex which is given a one-dimensional `bump` with the `bump1()`-function. The Formex undergoes a deformation in the Z-direction (2-direction), forced by the point `[0, 0, dz]`. The `bump` intensity is specified by the quadratic `bump_z` function and varies along

the X-axis (0-axis). The creation of this single bumped strut, oriented along the X-axis is summarized in the next script and depicted in figures *A straight line segment*, *The line segment with replications* and *A bumped line segment*.



Fig. 1: A straight line segment



Fig. 2: The line segment with replications



Fig. 3: A bumped line segment

The single bumped strut (`base`) is rescaled homothetically in the XY-plane to size one with the `scale()`-function. Subsequently, the `shear()`-functionality generates a new NE Formex by skewing the `base` Formex in the Y-direction (1-direction) with a `skew` factor of 1 in the YX-plane. As a result, the Y-coordinates of the `base` Formex are altered according to the following rule: $y_2 = y_1 + skewx_1$. Similarly a SE Formex is generated by a `shear()` operation on a mirrored copy of the `base` Formex. The `base` copy, mirrored in the direction of the XY-plane (perpendicular to the 2-axis), is obtained by the `reflect()` command. Both Formices are given a different property number by the `setProp()`-function, visualised by the different color codes in Figure *Unit cell of crossing wires and connectors*. This number can be used as an entry in a database, which holds some sort of property. The Formex and the database are two separate entities, only linked by the property numbers. The `rosette()`-function creates a unit cell of crossing struts by 2 rotational replications with an angular step of `[180]:math:deg` around the Z-axis (the original Formex is the

first of the 2 replicas). If specified in the constructor, an additional Formex with property 2 connects the first points of the NE and SE Formices.

(lines 72-83)



Fig. 4: Rescaled bumped strut

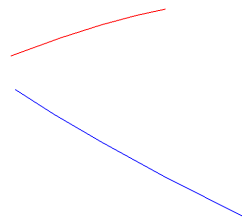


Fig. 5: Mirrored and skewed bumped strut

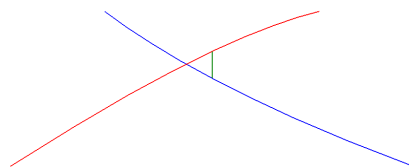


Fig. 6: Unit cell of crossing wires and connectors

5.1.2 Extending the base module

Subsequently, a mirrored copy of the base cell is generated (Figure *Mirrored unit cell*). Both Formices are translated to their appropriate side by side position with the `translate()`-option and form the complete extended base module with 4 by 4 dimensions as depicted in Figure *Completed base module*. Furthermore, both Formices are defined as an attribute of the `DoubleHelixStent` class by the `self`-statement, allowing their use after every `DoubleHelixStent` initialisation. Such further use is impossible with local variables, such as for example the NE and SE Formices.

(lines 84-89)

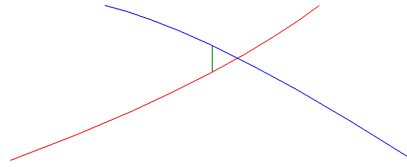


Fig. 7: Mirrored unit cell

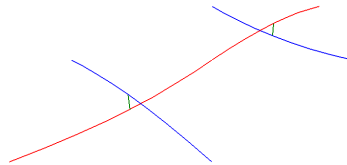


Fig. 8: Completed base module

5.1.3 Full nearly planar pattern

The fully nearly planar pattern is obtained by copying the base module in two directions and shown in Figure *Full planar topology*. `replic2()` generates this pattern with n_x and n_y replications with steps dx and dy in respectively, the default X- and Y-direction.

(lines 90-93)

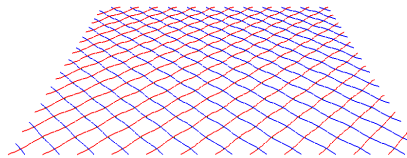


Fig. 9: Full planar topology

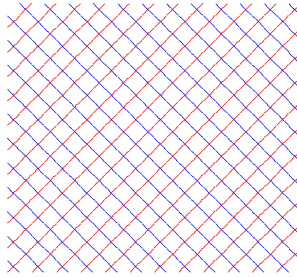


Fig. 10: Orthogonal view of the full planar topology

5.1.4 Cylindrical stent structure

Finally the full pattern is translated over the stent radius r in Z -direction and transformed to the cylindrical stent structure by a coordinate transformation with the Z -coordinates as distance r , the X -coordinates as angle θ and the Y -coordinates as height z . The `scale()`-operator rescales the stent structure to the correct circumference and length. The resulting stent geometry is depicted in Figure *Cylindrical stent*. (lines 94-96)

In addition to the stent initialization, the `DoubleHelixStent` class script contains a function `all()` representing the complete stent Formex. Consequently, the `DoubleHelixStent` class has four attributes: the Formices `cell1`, `cell2` and `all`; and the number `ny`. (lines 97-100)

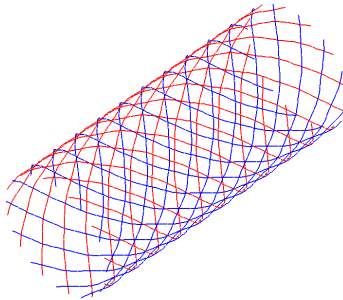


Fig. 11: Cylindrical stent

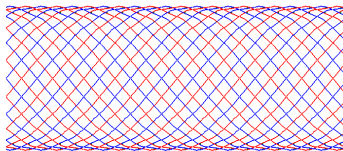


Fig. 12: Orthogonal view of the cylindrical stent

5.1.5 Parametric stent geometry

An inherent feature of script-based modeling is the possibility of easily generating lots of variations on the original geometry. This is a huge advantage for parametric analyses and illustrated in figures *Stent variant with $De=16$, $nx=6$, $\beta=25$* : these wire stents are all created with the same script, but with other values of the parameters De , nx and β . As the script for building the wire stent geometry is defined as the `DoubleHelixStent` class in the (`WireStent.py`) script, it can easily be imported for e.g. this purpose.

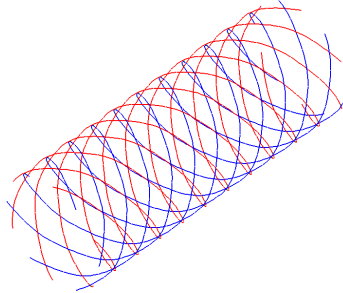


Fig. 13: Stent variant with $De = 16$, $nx = 6$, $\beta = 25$

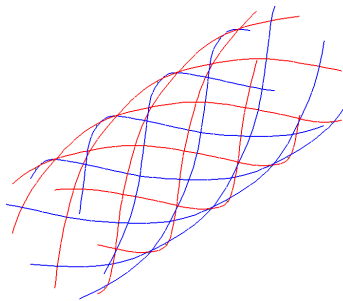


Fig. 14: Stent variant with $De = 16$, $nx = 6$, $\beta = 50$

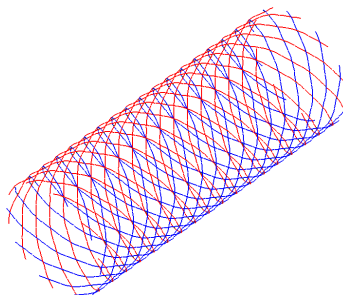


Fig. 15: Stent variant with $De = 16$, $nx = 10$, $\beta = 25$

```
#   *** pyformex ***
##
## This file is part of pyFormex 1.0.7 (Mon Jun 17 12:20:39 CEST 2019)
## pyFormex is a tool for generating, manipulating and transforming 3D
```

(continues on next page)

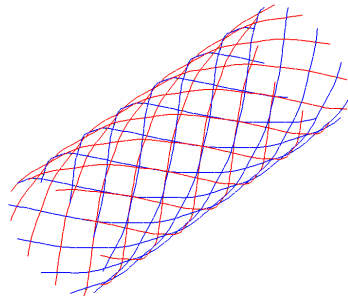


Fig. 16: Stent variant with $De = 16$, $nx = 10$, $\beta = 50$

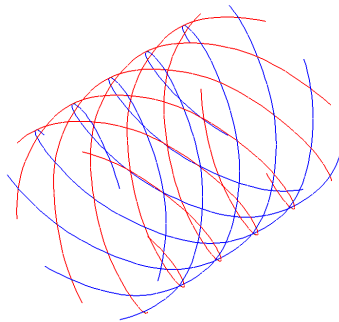


Fig. 17: Stent variant with $De = 32$, $nx = 6$, $\beta = 25$

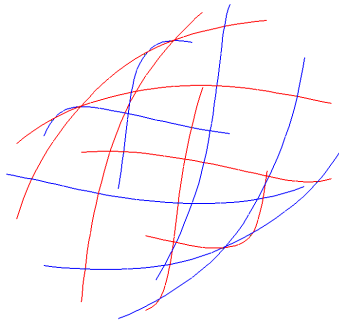


Fig. 18: Stent variant with $De = 32$, $nx = 6$, $\beta = 50$

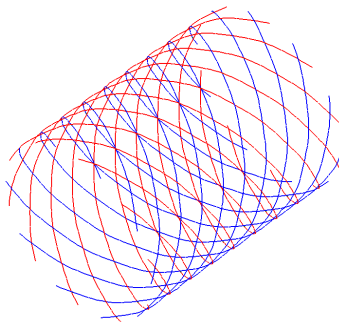


Fig. 19: Stent variant with $De = 32$, $nx = 10$, $\beta = 25$

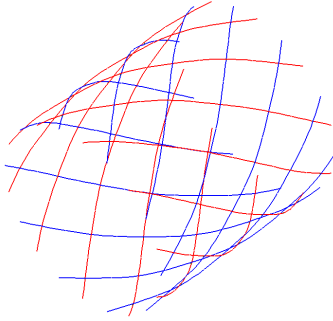


Fig. 20: Stent variant with $De = 32$, $nx = 10$, $\beta = 50$

(continued from previous page)

```
## geometrical models by sequences of mathematical operations.
## Home page: http://pyformex.org
## Project page: http://savannah.nongnu.org/projects/pyformex/
## Copyright 2004-2019 (C) Benedict Verheghe (benedict.verheghe@ugent.be)
## Distributed under the GNU General Public License version 3 or later.
##
## This program is free software: you can redistribute it and/or modify
## it under the terms of the GNU General Public License as published by
## the Free Software Foundation, either version 3 of the License, or
## (at your option) any later version.
##
## This program is distributed in the hope that it will be useful,
## but WITHOUT ANY WARRANTY; without even the implied warranty of
## MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
## GNU General Public License for more details.
##
## You should have received a copy of the GNU General Public License
## along with this program. If not, see http://www.gnu.org/licenses/.
##

from examples.WireStent import DoubleHelixStent

for De in [16.,32.]:
    for nx in [6,10]:
        for beta in [25,50]:
            stent = DoubleHelixStent(De,40.,0.22,nx,beta).all()
            draw(stent,view='iso')
            pause()
            clear()
```

Obviously, generating such parametric wire stent geometries with classical CAD methodologies is feasible, though probably (very) time consuming. However, as provides a multitude of features (such as parametric modeling, finite element pre- and postprocessing, optimization strategies, etcetera) in one single consistent environment, it appears to be the obvious way to go when studying the mechanical behavior of braided wire stents.

5.2 Operating on surface meshes

Besides being used for creating geometries, also offers interesting possibilities for executing specialized operations on surface meshes, usually STL type triangulated meshes originating from medical scan (CT) images. Some of the algorithms developed were included in .

5.2.1 Unroll stent

A stent is a medical device used to reopen narrowed arteries. The vast majority of stents are balloon-expandable, which means that the metal structure is deployed by inflating a balloon, located inside the stent. Figure *Triangulated mesh of a Cypher® stent* shows an example of such a stent prior to expansion (balloon not shown). The 3D surface is obtained by micro CT and consists of triangles.



Fig. 21: Triangulated mesh of a Cypher® stent

The structure of such a device can be quite complex and difficult to analyse. The same functions offers for creating geometries can also be employed to investigate triangulated meshes. A simple unroll operation of the stent gives a much better overview of the complete geometrical structure and allows easier analysis (see figure *Result of the unroll operation*).

```
F = F.toCylindrical().scale([1., 2*radius*pi/360, 1.])
```

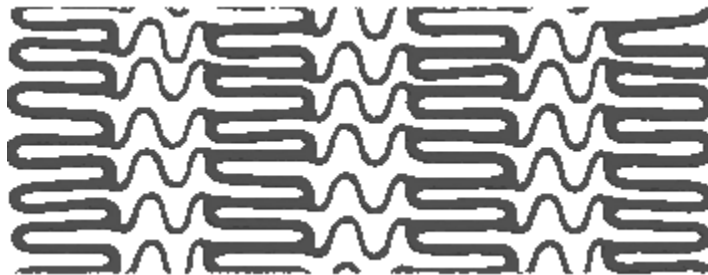


Fig. 22: Result of the unroll operation

The unrolled geometry can then be used for further investigations. An important property of such a stent is the circumference of a single stent cell. The `clip()` method can be used to isolate a single stent cell. In order to obtain a line describing the stent cell, the function `intersectionLinesWithPlane()` has been used. The result can be seen in figures *Part of the intersection with a plane*.

Finally, one connected circumference of a stent cell is selected (figure *Circumference of a stent cell*) and the `length()` function returns its length, which is 9.19 mm.



Fig. 23: Part of the intersection with a plane



Fig. 24: Circumference of a stent cell

PYFORMEX REFERENCE MANUAL

Abstract

This is the reference manual for pyFormex 1.0.7. It describes most of the classes and functions defined in the pyFormex modules. It was built automatically from the pyFormex sources and is therefore the ultimate reference document if you want to look up the precise arguments (and their meaning) of any class constructor or function in pyFormex. The `genindex` and `modindex` may be helpful in navigating through this document.

This reference manual describes the classes in functions defined in most of the pyFormex modules. It was built automatically from the docstrings in the pyFormex sources. The pyFormex modules are placed in three paths:

- `pyformex` contains the core functionality, with most of the geometrical transformations, the pyFormex scripting language and utilities,
- `pyformex/gui` contains all the modules that form the interactive graphical user interface,
- `pyformex/plugins` contains extensions that are not considered to be essential parts of pyFormex. They usually provide additional functionality for specific applications.

Some of the modules are loaded automatically when pyFormex is started. Currently this is the case with the modules `coords`, `formex`, `arraytools`, `script` and, if the GUI is used, `draw` and `colors`. All the public definitions in these modules are available to pyFormex scripts without explicitly importing them. Also available is the complete `numpy` namespace, because it is imported by `arraytools`.

The definitions in the other modules can only be accessed using the normal Python `import` statements.

6.1 Autoloaded modules

The definitions in these modules are always available to your scripts, without the need to explicitly import them.

6.1.1 `coords` — A structured collection of 3D coordinates.

The `coords` module defines the `Coords` class, which is the basic data structure in pyFormex to store the coordinates of points in a 3D space.

This module implements a data class for storing large sets of 3D coordinates and provides an extensive set of methods for transforming these coordinates. Most of pyFormex's classes which represent geometry (e.g. `Formex`, `Mesh`, `TriSurface`, `Curve`) use a `Coords` object to store their coordinates, and thus inherit all the transformation methods of this class.

While the user will mostly use the higher level classes, he might occasionally find good reason to use the `Coords` class directly as well.

Classes defined in module coords

class `coords.Coords` (*data=None, dtype=Float, copy=False*)

A structured collection of points in a 3D cartesian space.

The `Coords` class is the basic data structure used throughout pyFormex to store the coordinates of points in a 3D space. It is used by other classes, such as `Formex`, `Mesh`, `TriSurface`, `Curve`, which thus inherit the same transformation capabilities. Applications will mostly use the higher level classes, which have more elaborated consistency checking and error handling.

`Coords` is implemented as a subclass of `numpy.ndarray`, and thus inherits all its methods and attributes. The last axis of the `Coords` however always has a length equal to 3. Each set of 3 values along the last axis are the coordinates (in a global 3D cartesian coordinate system) of a single point in space. The full `Coords` array thus is a collection of points. If the array is 2-dimensional, the `Coords` is a flat list of points. But if the array has more dimensions, the collection of points itself becomes structured.

The float datatype is only checked at creation time. It is the responsibility of the user to keep this consistent throughout the lifetime of the object.

Note: Methods that transform a `Coords` object, like `scale()`, `translate()`, `rotate()`, ... do not change the original `Coords` object, but return a new object. Some methods however have an *inplace* option that allows the user to force coordinates to be changed in place. This option is seldom used however: rather we conveniently use statements like:

```
X = X.some_transform()
```

and Python can immediately free and recollect the memory used for the old object X.

Parameters

- **data** (float *array_like*, or string) – Data to initialize the `Coords`. The last axis should have a length of 1, 2 or 3, but will be expanded to 3 if it is less, filling the missing coordinates with zeros. Thus, if you only specify two coordinates, all points are lying in the $z=0$ plane. Specifying only one coordinate creates points along the x-axis.

As a convenience, data may also be entered as a string, which will be passed to the `pattern()` function to create the actual coordinates of the points.

If no data are provided, an empty `Coords` with shape (0,3) is created.

- **dtype** (*float datatype, optional*) – If not provided, the datatype of `data` is used, or the default `Float` (which is equivalent to `numpy.float32`).
- **copy** (*bool*) – If True, the data are copied. The default setting will try to use the original data if possible, e.g. if `data` is a correctly shaped and typed `numpy.ndarray`.

Returns `Coords` – An instance of the `Coords` class, which is basically an ndarray of floats, with the last axis having a length of 3.

The `Coords` instance has a number of attributes that provide views on (part of) the data. They are a notational convenience over using indexing. These attributes can be used to set all or some of the coordinates by direct assignment. The assigned data should however be broadcast compatible with the assigned shape: the shape of the `Coords` can not be changed.

xyz

The full coordinate array as an ndarray.

Type float array

- x**
The X coordinates of the points as an ndarray with shape `pshape()`.
Type float array
- y**
The Y coordinates of the points as an ndarray with shape `pshape()`.
Type float array
- z**
The Z coordinates of the points as an ndarray with shape `pshape()`.
Type float array
- xy**
The X and Y coordinates of the points as an ndarray with shape `pshape() + (2,)`.
Type float array
- xz**
The X and Z coordinates of the points as an ndarray with shape `pshape() + (2,)`.
Type float array
- yz**
The Y and Z coordinates of the points as an ndarray with shape `pshape() + (2,)`.
Type float array

Examples

```
>>> Coords([1.,2.])
Coords([ 1., 2., 0.])
>>> X = Coords(arange(6).reshape(2,3))
>>> print(X)
[[ 0.  1.  2.]
 [ 3.  4.  5.]]
>>> print(X.y)
[ 1.  4.]
>>> X.z[1] = 9.
>>> print(X)
[[ 0.  1.  2.]
 [ 3.  4.  9.]]
>>> print(X.xz)
[[ 0.  2.]
 [ 3.  9.]]
>>> X.x = 0.
>>> print(X)
[[ 0.  1.  2.]
 [ 0.  4.  9.]]
```

```
>>> Y = Coords(X)           # Y shares its data with X
>>> Z = Coords(X, copy=True) # Z is independent
>>> Y.y = 5
>>> Z.z = 6
>>> print(X)
[[ 0.  5.  2.]
 [ 0.  5.  9.]]
>>> print(Y)
```

(continues on next page)

(continued from previous page)

```

[[ 0.  5.  2.]
 [ 0.  5.  9.]]
>>> print(Z)
[[ 0.  1.  6.]
 [ 0.  4.  6.]]
>>> X.coords is X
True
>>> Z.xyz = [1,2,3]
>>> print(Z)
[[ 1.  2.  3.]
 [ 1.  2.  3.]]

```

```

>>> print(Coords('0123')) # initialize with string
[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 1.  1.  0.]
 [ 0.  1.  0.]]

```

swapaxes (*axis1*, *axis2*)

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to *numpy.swapaxes* for full documentation.

See also:

`numpy.swapaxes()` equivalent function

xyz

Returns the coordinates of the points as an ndarray.

Returns an ndarray with shape *self.shape* except last axis is reduced to 2, providing a view on all the coordinates of all the points.

x

Returns the X-coordinates of all points.

Returns an ndarray with shape *self.pshape()*, providing a view on the X-coordinates of all the points.

y

Returns the Y-coordinates of all points.

Returns an ndarray with shape *self.pshape()*, providing a view on the Y-coordinates of all the points.

z

Returns the Z-coordinates of all points.

Returns an ndarray with shape *self.pshape()*, providing a view on the Z-coordinates of all the points.

xy

Returns the X- and Y-coordinates of all points.

Returns an ndarray with shape *self.shape* except last axis is reduced to 2, providing a view on the X- and Y-coordinates of all the points.

xz

Returns the X- and Y-coordinates of all points.

Returns an ndarray with shape *self.shape* except last axis is reduced to 2, providing a view on the X- and Z-coordinates of all the points.

yz

Returns the X- and Y-coordinates of all points.

Returns an ndarray with shape *self.shape* except last axis is reduced to 2, providing a view on the Y- and Z-coordinates of all the points.

coords

Returns the *Coords* object .

This exists only for consistency with other classes.

fprint (*fmt*='%10.3e %10.3e %10.3e')

Formatted printing of the points of a *Coords* object.

Parameters *fmt* (*string*) – Format to be used to print a single point. The supplied format should contain exactly 3 formatting sequences, one for each of the three coordinates.

Examples

```
>>> x = Coords([[0.,0.], [1.,0.]], [[0.,1.], [0.,2.]])
>>> x.fprint()
0.000e+00  0.000e+00  0.000e+00
1.000e+00  0.000e+00  0.000e+00
0.000e+00  1.000e+00  0.000e+00
0.000e+00  2.000e+00  0.000e+00
>>> x.fprint("%5.2f"*3)
0.00 0.00 0.00
1.00 0.00 0.00
0.00 1.00 0.00
0.00 2.00 0.00
```

pshape ()

Return the points shape of the *Coords* object.

This is the shape of the `numpy.ndarray` with the last axis removed.

Note: The full shape of the Coords array can be obtained from the inherited (NumPy) shape attribute.

Examples

```
>>> X = Coords(arange(12).reshape(2,1,2,3))
>>> X.shape
(2, 1, 2, 3)
>>> X.pshape()
(2, 1, 2)
```

points ()

Return the *Coords* object as a flat set of points.

Returns *Coords* – The Coords reshaped to a 2-dimensional array, flattening the structure of the points.

Examples

```
>>> X = Coords(arange(12).reshape(2,1,2,3))
>>> X.shape
(2, 1, 2, 3)
>>> X.points().shape
(4, 3)
```

npoints()

Return the total number of points in the Coords.

Notes

npoints and *ncoords* are equivalent. The latter exists to provide a common interface with other geometry classes.

Examples

```
>>> Coords(arange(12).reshape(2,1,2,3)).npoints()
4
```

ncoords()

Return the total number of points in the Coords.

Notes

npoints and *ncoords* are equivalent. The latter exists to provide a common interface with other geometry classes.

Examples

```
>>> Coords(arange(12).reshape(2,1,2,3)).npoints()
4
```

bbox()

Return the bounding box of a set of points.

The bounding box is the smallest rectangular volume in the global coordinates, such that no points of the *Coords* are outside that volume.

Returns *Coords(2,3)* – Coords array with two points: the first point contains the minimal coordinates, the second has the maximal ones.

See also:

center() return the center of the bounding box

bbboxPoint() return a corner or middle point of the bounding box

bbboxPoints() return all corners of the bounding box

Examples

```
>>> X = Coords([[0.,0.,0.],[3.,0.,0.],[0.,3.,0.]])
>>> print(X.bbox())
[[ 0. 0. 0.]
 [ 3. 3. 0.]
```

center()

Return the center of the *Coords*.

The center of a *Coords* is the center of its *bbox()*. The return value is a (3,) shaped *Coords* object.

See also:

bbox() return the bounding box of the *Coords*

centroid() return the average coordinates of the points

Examples

```
>>> X = Coords([[0.,0.,0.],[3.,0.,0.],[0.,3.,0.]])
>>> print(X.center())
[ 1.5 1.5 0. ]
```

bboxPoint (*position*)

Return a bounding box point of a *Coords*.

Bounding box points are points whose coordinates are either the minimal value, the maximal value or the middle value for the *Coords*. Combining the three values in three dimensions results in $3*3 = 27$ alignment points. The corner points of the bounding box are a subset of these.

Parameters *position* (*str*) – String of three characters, one for each direction 0, 1, 2. Each character should be one of the following

- ‘-’: use the minimal value for that coordinate,
- ‘+’: use the maximal value for that coordinate,
- ‘0’: use the middle value for that coordinate.

Any other character will set the corresponding coordinate to zero.

Notes

A string ‘000’ is equivalent with *center()*. The values ‘—’ and ‘+++’ give the points of the bounding box.

See also:

Coords.align() translate *Coords* by *bboxPoint*

Examples

```
>>> X = Coords([[0.,0.,0.],[1.,1.,1.]])
>>> print(X.bboxPoint('-0+'))
[ 0. 0.5 1. ]
```

bboxPoints()

Return all the corners of the bounding box point of a *Coords*.

Returns *Coords* (8,3) – A *Coords* with the eight corners of the bounding box, in the order of a `elements.Hex8`.

See also:

bbbox() return only two points, with the minimum and maximum coordinates

Examples

```
>>> X = Coords([[0.,0.,0.],[3.,0.,0.],[0.,3.,0.]])
>>> print(X.bboxPoints())
[[ 0.  0.  0.]
 [ 3.  0.  0.]
 [ 3.  3.  0.]
 [ 0.  3.  0.]
 [ 0.  0.  0.]
 [ 3.  0.  0.]
 [ 3.  3.  0.]
 [ 0.  3.  0.]
```

average (*wts=None, axis=None*)

Returns a (weighted) average of the *Coords*.

The average of a *Coords* is a *Coords* that is obtained by averaging the points along some or all axes. Weights can be specified to get a weighted average.

Parameters

- **wts** (float *array_like*, optional) – Weight to be attributed to the points. If provided, and *axis* is an int, *wts* should be 1-dim with the same length as the specified axis. Else, it has a shape equal to `self.shape` or `self.shape[:-1]`.
- **axis** (*int or tuple of ints*, optional) – If provided, the average is computed along the specified axis/axes only. Else, the average is taken over all the points, thus over all the axes of the array except the last.

Notes

Averaging over the -1 axis does not make much sense.

Examples

```
>>> X = Coords([[0.,0.,0.],[1.,0.,0.],[2.,0.,0.]], [[4.,0.,0.
↵], [5.,0.,0.],[6.,0.,0.]])
>>> X = Coords(arange(6).reshape(3,2,1))
>>> X
Coords([[[ 0.,  0.,  0.],
          [ 1.,  0.,  0.]],
<BLANKLINE>
          [[ 2.,  0.,  0.],
          [ 3.,  0.,  0.]],
<BLANKLINE>
```

(continues on next page)

(continued from previous page)

```

        [[ 4.,  0.,  0.],
         [ 5.,  0.,  0.]])
>>> print(X.average())
[ 2.5  0.  0. ]
>>> print(X.average(axis=0))
[[ 2.  0.  0.]
 [ 3.  0.  0.]]
>>> print(X.average(axis=1))
[[ 0.5  0.  0. ]
 [ 2.5  0.  0. ]
 [ 4.5  0.  0. ]]
>>> print(X.average(wts=[0.5,0.25,0.25],axis=0))
[[ 1.5  0.  0. ]
 [ 2.5  0.  0. ]]
>>> print(X.average(wts=[3,1],axis=1))
[[ 0.25  0.  0. ]
 [ 2.25  0.  0. ]
 [ 4.25  0.  0. ]]
>>> print(X.average(wts=multiplex([3,1],3,0)))
[ 2.25  0.  0. ]

```

centroid()

Return the centroid of the *Coords*.

The centroid of *Coords* is the point whose coordinates are the mean values of all points.

Returns *Coords* (3,) – A single point that is the centroid of the *Coords*.

See also:

center() return the center of the bounding box.

Examples

```

>>> print(Coords([[0.,0.,0.],[3.,0.,0.],[0.,3.,0.]]) .centroid())
[ 1.  1.  0.]

```

centroids()

Return the *Coords* itself.

Notes

This method exists only to have a common interface with other geometry classes.

sizes()

Return the bounding box sizes of the *Coords*.

Returns *array* (3,) – The length of the bounding box along the three global axes.

See also:

dsize() The diagonal size of the bounding box.

principalSizes() the sizes of the bounding box along the principal axes

Examples

```
>>> print(Coords([[0.,0.,0.],[3.,0.,0.],[0.,3.,0.]]) .sizes())  
[ 3. 3. 0.]
```

maxsize()

Return the maximum size of a Coords in any coordinate direction.

Returns *float* – The maximum length of any edge of the bounding box.

Notes

This is a convenient shorthand for *self.sizes().max()*.

See also:

sizes() return the length of the bounding box along global axes

bbox() return the bounding box

Examples

```
>>> print(Coords([[0.,0.,0.],[3.,0.,0.],[0.,3.,0.]]) .maxsize())  
3.0
```

dsize()

Return the diagonal size of the bounding box of the *Coords*.

Returns *float* – The length of the diagonal of the bounding box.

Notes

All the points of the Coords are inside a sphere with the *center()* as center and the *dsize()* as length of the diameter (though it is not necessarily the smallest bounding sphere). *dsize()* is in general a good estimate for the maximum size of the cross section to be expected when the object can be rotated freely around its center. It is conveniently used to zoom the camera on an object, while guaranteeing that the full object remains visible during rotations.

See also:

bsphere() return radius of smallest sphere encompassing all points

sizes() return the length of the bounding box along global axes

bbox() return the bounding box

Examples

```
>>> print(Coords([[0.,0.,0.],[3.,0.,0.],[0.,3.,0.]]) .dsize())  
4.24264
```

bsphere ()

Return the radius of the bounding sphere of the *Coords*.

The bounding sphere used here is the smallest sphere with center in the center() of the *Coords*, and such that no points of the *Coords* are lying outside the sphere.

Returns *float* – The maximum distance of any point to the *Coords.center*.

Notes

This is not necessarily the absolute smallest bounding sphere, because we use the center from lookin only in the global axes directions.

Examples

```
>>> X = Coords([[0.,0.,0.],[3.,0.,0.],[0.,3.,0.]])
>>> print(X.dsize(), X.bsphere())
4.24264 2.12132
>>> X = Coords([[0.5,0.],[1.,0.5],[0.5,1.0],[0.0,0.5]])
>>> print(X.dsize(), X.bsphere())
1.41421 0.5
```

bboxes ()

Return the bboxes of all subsets of points in the *Coords*.

Subsets of points are 2-dim subarrays of the *Coords*, taken along the two last axes. If the *Coords* has *ndim*=2, there is only one subset: the full *Coords*.

Returns *float array* – Array with shape (...2,3). The elements along the penultimate axis are the minimal and maximal values of the *Coords* along that axis.

Examples

```
>>> X = Coords(arange(18).reshape(2,3,3))
>>> print(X)
[[[ 0.  1.  2.]
 [ 3.  4.  5.]
 [ 6.  7.  8.]]
<BLANKLINE>
[[ 9. 10. 11.]
 [12. 13. 14.]
 [15. 16. 17.]]]
>>> print(X.bboxes())
[[[ 0.  1.  2.]
 [ 6.  7.  8.]]
<BLANKLINE>
[[ 9. 10. 11.]
 [15. 16. 17.]]]
```

inertia (mass=None)

Return inertia related quantities of the *Coords*.

Parameters *mass* (*float array, optional*) – If provided, it is a 1-dim array with *npoints()* weight values for the points, in the order of the *points()*. The default is to attribute a weight 1.0 to each point.

Returns

Inertia – The Inertia object has the following attributes:

- `mass`: the total mass (float)
- `ctr`: the center of mass: float (3,)
- `tensor`: the inertia tensor in the central axes: shape (3,3)

See also:

principalCS() Return the principal axes of the inertia tensor

Examples

```
>>> from pyformex.elements import Tet4
>>> I = Tet4.vertices.inertia()
>>> print(I.tensor)
[[ 1.5  0.25  0.25]
 [ 0.25  1.5  0.25]
 [ 0.25  0.25  1.5 ]]
>>> print(I.ctr)
[ 0.25  0.25  0.25]
>>> print(I.mass)
4.0
```

principalCS (*mass=None*)

Return a *CoordSys* formed by the principal axes of inertia.

Parameters *mass* (1-dim float array (*points()*), optional) – The mass to be attributed to each of the points, in the order of *npoints()*. If not provided, a mass 1.0 will be attributed to each point.

Returns *CoordSys* object. – Coordinate system aligned along the principal axes of the inertia, for the specified point masses. The origin of the *CoordSys* is the center of mass of the *Coords*.

See also:

centralCS() *CoordSys* at the center of mass, but axes along global directions

Examples

```
>>> from pyformex.elements import Tet4
>>> print(Tet4.vertices.principalCS())
CoordSys: trl=[ 0.25  0.25  0.25]; rot=[[ 0.58  0.58  0.58]
                                     [ 0.34 -0.81  0.47]
                                     [ 0.82 -0.41 -0.41]]
```

principalSizes ()

Return the sizes in the principal directions of the *Coords*.

Returns *float array* (3,) – Array with the size of the bounding box along the 3 principal axes.

Notes

This is a convenient shorthand for: `self.toCS(self.principalCS()).sizes()`

Examples

```
>>> print(Coords([[0.,0.,0.],[3.,0.,0.]]) .rotate(30,2) .principalSizes())
[ 0. 0. 3.]
```

centralCS (*mass=None*)

Returns the central coordinate system of the Coords.

Parameters *mass* (1-dim float array (*points()*), optional) – The mass to be attributed to each of the points, in the order of *npoints()*. If not provided, a mass 1.0 will be attributed to each point.

Returns *CoordSys* object. – Coordinate system with origin at the center of mass of the Coords and axes parallel to the global axes.

See also:

principalCS() *CoordSys* aligned with principa axes of inertia tensor

Examples

```
>>> from pyformex.elements import Tet4
>>> print(Tet4.vertices.centralCS())
CoordSys: trl=[ 0.25  0.25  0.25]; rot=[[ 1.  0.  0.]
                                         [ 0.  1.  0.]
                                         [ 0.  0.  1.]]
```

distanceFromPoint (*p*)

Returns the distance of all points from the point *p*.

Parameters *p* (float *array_like* with shape (3,) or (1,3)) – Coordinates of a single point in space

Returns *float array* – Array with shape *pshape()* holding the distance of each point to point *p*. All values are positive or zero.

See also:

closestPoint () return the point of Coords closest to given point

Examples

```
>>> X = Coords([[0.,0.,0.],[2.,0.,0.],[1.,3.,0.],[-1.,0.,0.]])
>>> print(X.distanceFromPoint([0.,0.,0.]))
[ 0.    2.    3.16  1. ]
```

distanceFromLine (*p, n*)

Returns the distance of all points from the line (*p,n*).

Parameters

- *p* (float *array_like* with shape (3,) or (1,3)) – Coordinates of some point on the line.
- *n* (float *array_like* with shape (3,) or (1,3)) – Vector specifying the direction of the line.

Returns *float array* – Array with shape *pshape()* holding the distance of each point to the line through *p* and having direction *n*. All values are positive or zero.

Examples

```
>>> X = Coords([[0.,0.,0.],[2.,0.,0.],[1.,3.,0.],[-1.,0.,0.]])
>>> print(X.distanceFromLine([0.,0.,0.],[1.,1.,0.]))
[ 0.  1.41 1.41 0.71]
```

distanceFromPlane (*p*, *n*)

Return the distance of all points from the plane (*p*,*n*).

Parameters

- **p** (float *array_like* with shape (3,) or (1,3)) – Coordinates of some point in the plane.
- **n** (float *array_like* with shape (3,) or (1,3)) – The normal vector to the plane.

Returns *float array* – Array with shape *pshape()* holding the distance of each point to the plane through *p* and having normal *n*. The values are positive if the point is on the side of the plane indicated by the positive normal.

See also:

[*directionalSize\(\)*](#) find the most distant points at both sides of plane

Examples

```
>>> X = Coords([[0.,0.,0.],[2.,0.,0.],[1.,3.,0.],[-1.,0.,0.]])
>>> print(X.distanceFromPlane([0.,0.,0.],[1.,0.,0.]))
[ 0.  2.  1. -1.]
```

closestToPoint (*p*, *return_dist=False*)

Returns the point closest to a given point *p*.

Parameters **p** (*array_like* (3,)) – Coordinates of a single point in space

Returns *int* – Index of the point in the Coords that has the minimal Euclidean distance to the point *p*. Use this index with *self.points()* to get the coordinates of that point.

Examples

```
>>> X = Coords([[0.,0.,0.],[3.,0.,0.],[0.,3.,0.]])
>>> X.closestToPoint([2.,0.,0.])
1
>>> X.closestToPoint([2.,0.,0.], True)
(1, 1.0)
```

directionalSize (*n*, *p=None*, *return_points=False*)

Returns the extreme distances from the plane *p*,*n*.

Parameters

- **n** (a single int or a float *array_like* (3,)) – The direction of the normal to the plane. If an int, it is the number of a global axis. Else it is a vector with 3 components.
- **p** (*array_like* (3,), optional) – Coordinates of a point in the plane. If not provided, the *center()* of the Coords is used.
- **return_points** (*bool*) – If True, also return a Coords with two points along the line (*p*,*n*) and at the extreme distances from the plane(*p*,*n*).

Returns

- **dmin** (*float*) – The minimal (signed) distance of a point of the Coords to the plane (p,n). The value can be negative or positive.
- **dmax** (*float*) – The maximal (signed) distance of a point of the Coords to the plane (p,n). The value can be negative or positive.
- **points** (*Coords (2,3), optional*) – If `return_points=True` is provided, also returns a Coords holding two points on the line (p,n) with minimal and maximal distance from the plane (p,n). These two points together with the normal *n* define two parallel planes such that all points of *self* are between or on the planes.

Notes

The maximal size of *self* in the direction *n* is found from the difference `dmax - dmin`. See also `directionalWidth()`.

See also:

`directionalExtremes()` return two points in the extreme planes

`directionalWidth()` return the distance between the extreme planes

`distanceFromPlane()` return distance of all points to a plane

Examples

```
>>> X = Coords([[0.,0.,0.],[3.,0.,0.],[0.,3.,0.]])
>>> X.directionalSize([1,0,0])
(-1.5, 1.5)
>>> X.directionalSize([1,0,0],[1.,0.,0.])
(-1.0, 2.0)
>>> X.directionalSize([1,0,0],return_points=True)
(-1.5, 1.5, Coords([[ 0. ,  1.5,  0. ],
                    [ 3. ,  1.5,  0. ]]))
```

directionalExtremes (*n, p=None*)

Returns extremal planes in the direction *n*.

Parameters: see `directionalSize()`.

Returns *Coords (2,3)* – A Coords holding the two points on the line (p,n) with minimal and maximal distance from the plane (p,n). These two points together with the normal *n* define two parallel planes such that all points of *self* are between or on the planes.

See also:

`directionalSize()` return minimal and maximal distance from plane

Notes

This is like `directionalSize` with the `return_points` options, but only returns the extreme points.

Examples

```
>>> X = Coords([[0.,0.,0.],[3.,0.,0.],[0.,3.,0.]])
>>> X.directionalExtremes([1,0,0])
Coords([[ 0. ,  1.5,  0. ],
        [ 3. ,  1.5,  0. ]])
```

directionalWidth (*n*)

Returns the width of a Coords in the given direction.

Parameters: see `directionalSize()`.

Returns *float* – The size of the Coords in the direction *n*. This is the distance between the extreme planes with normal *n* touching the Coords.

See also:

`directionalSize()` return minimal and maximal distance from plane

Notes

This is like `directionalSize` but only returns the difference between *dmax* and *dmin*.

Examples

```
>>> X = Coords([[0.,0.,0.],[3.,0.,0.],[0.,3.,0.]])
>>> print(X.directionalWidth([1,0,0]))
3.0
```

test (*dir=0, min=None, max=None, atol=0.0*)

Flag points having coordinates between min and max.

Test the position of the points of the `Coords` with respect to one or two parallel planes. This method is very convenient in clipping a Coords in a specified direction. In most cases the clipping direction is one of the global coordinate axes, but a general direction may be used as well.

Testing along global axis directions is highly efficient. It tests whether the corresponding coordinate is above or equal to the *min* value and/or below or equal to the *max* value. Testing in a general direction tests whether the distance to the *min* plane is positive or zero and/or the distance to the *max* plane is negative or zero.

Parameters

- **dir** (a single int or a float *array_like* (3,)) – The direction in which to measure distances. If an int, it is one of the global axes (0,1,2). Else it is a vector with 3 components. The default direction is the global x-axis.
- **min** (*float or point-like, optional*) – Position of the minimal clipping plane. If *dir* is an int, this is a single float giving the coordinate along the specified global axis. If *dir* is a vector, this must be a point and the minimal clipping plane is defined by this point and the normal vector *dir*. If not provided, there is no clipping at the minimal side.
- **max** (*float or point-like.*) – Position of the maximal clipping plane. If *dir* is an int, this is a single float giving the coordinate along the specified global axis. If *dir* is a vector, this must be a point and the maximal clipping plane is defined by this point and the normal vector *dir*. If not provided, there is no clipping at the maximal side.

- **atol** (*float*) – Tolerance value added to the tests to account for accuracy and rounding errors. A *min* test will be ok if the point's distance from the *min* clipping plane is $> -atol$ and/or the distance from the *max* clipping plane is $< atol$. Thus a positive *atol* widens the clipping planes.

Returns bool array with shape *pshape()* – Array flagging whether the points for the *Coords* pass the test(s) or not. The return value can directly be used as an index to *self* to obtain a *Coords* with the points satisfying the test (or not).

Raises *ValueError*: *At least one of min or max have to be specified* – If neither *min* nor *max* are provided.

Examples

```
>>> x = Coords([[0.,0.], [1.,0.], [0.,1.], [0.,2.]])
>>> print(x.test(min=0.5))
[[False True]
 [False False]]
>>> t = x.test(dir=1,min=0.5,max=1.5)
>>> print(x[t])
[[ 0.  1.  0.]
 [ 1.  0.  0.]
 [ 0.  2.  0.]]
>>> print(x[~t])
[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 0.  2.  0.]]
```

set (*f*)

Set the coordinates from those in the given array.

Parameters *f* (*float array_like*, broadcastable to *self.shape*.) – The coordinates to replace the current ones. This can not be used to change the shape of the *Coords*.

Raises *ValueError*: – If the shape of *f* does not allow broadcasting to *self.shape*.

Examples

```
>>> x = Coords([[0], [1], [2]])
>>> print(x)
[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 2.  0.  0.]]
>>> x.set([0.,1.,0.])
>>> print(x)
[[ 0.  1.  0.]
 [ 0.  1.  0.]
 [ 0.  1.  0.]]
```

scale (*scale*, *dir=None*, *center=None*, *inplace=False*)

Return a scaled copy of the *Coords* object.

Parameters

- **scale** (*float or tuple of 3 floats*) – Scaling factor(s). If it is a single value, and no *dir* is provided, scaling is uniformly applied to all axes; if *dir* is provided, only to the specified directions. If it is a tuple, the three scaling factors are applied to the respective global axes.

- **dir** (*int or tuple of ints, optional*) – One or more global axis numbers (0,1,2), indicating the direction(s) that should be scaled with the (single) value *scale*.
- **center** (*point-like, optional*) – If provided, use this point as the center of the scaling. The default is the global origin.
- **inplace** (*bool, optional*) – If True, the coordinates are change in-place.

Returns *Coords* – The Coords scaled as specified.

Notes

If a *center* is provided, the operation is equivalent with `self.translate(-center).scale(scale, dir).translate(center)`

Examples

```
>>> X = Coords([1., 1., 1.])
>>> print(X.scale(2))
[ 2.  2.  2.]
>>> print(X.scale([2, 3, 4]))
[ 2.  3.  4.]
>>> print(X.scale(2, dir=(1, 2)).scale(4, dir=0))
[ 4.  2.  2.]
>>> print(X.scale(2, center=[1., 0.5, 0.]))
[ 1.   1.5  2. ]
```

translate (*dir, step=1.0, inplace=False*)

Return a translated copy of the *Coords* object.

Translate the Coords in the direction *dir* over a distance *step* * *length(dir)*.

Parameters

- **dir** (*int (0,1,2) or float array_like (... ,3)*) – The translation vector. If an *int*, it specifies a global axis and the translation is in the direction of that axis. If an *array_like*, it specifies one or more translation vectors. If more than one, the array should be broadcastable to the Coords shape: this allows to translate different parts of the Coords over different vectors, all in one operation.
- **step** (*float*) – If *dir* is an *int*, this is the length of the translation. Else, it is a multiplying factor applied to *dir* prior to applying the translation.

Returns *Coords* – The Coords translated over the specified vector(s).

Note: `trl()` is a convenient shorthand for `translate()`.

See also:

`centered()` translate to center around origin

`Coords.align()` translate to align bounding box

Examples

```

>>> x = Coords([1.,1.,1.])
>>> print(x.translate(1))
[ 1. 2. 1.]
>>> print(x.translate(1,1.))
[ 1. 2. 1.]
>>> print(x.translate([0,1,0]))
[ 1. 2. 1.]
>>> print(x.translate([0,2,0],0.5))
[ 1. 2. 1.]
>>> x = Coords(arange(4).reshape(2,2,1))
>>> x
Coords([[[ 0.,  0.,  0.],
          [ 1.,  0.,  0.]],
<BLANKLINE>
        [[ 2.,  0.,  0.],
          [ 3.,  0.,  0.]])
>>> x.translate([[10.,-5.,0.],[20.,4.,0.]]) # translate with broadcasting
Coords([[[ 10., -5.,  0.],
          [ 21.,  4.,  0.]],
<BLANKLINE>
        [[ 12., -5.,  0.],
          [ 23.,  4.,  0.]])

```

`centered()`

Return a centered copy of the Coords.

Returns *Coords* – The Coords translated over *tus* that its *center()* coincides with the origin of the global axes.

Notes

This is equivalent with `self.translate(-self.center())`

Examples

```

>>> X = Coords('0123')
>>> print(X)
[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 1.  1.  0.]
 [ 0.  1.  0.]]
>>> print(X.centered())
[[-0.5 -0.5  0. ]
 [ 0.5 -0.5  0. ]
 [ 0.5  0.5  0. ]
 [-0.5  0.5  0. ]]

```

`align(alignment='—', point=[0.0, 0.0, 0.0])`

Align a *Coords* object on a given point.

Alignment involves a translation such that the bounding box of the Coords object becomes aligned with a given point. The bounding box alignment is done by the translation of *a* to the target point.

Parameters

- **alignment** (*str*) – The requested alignment is a string of three characters, one for each of the coordinate axes. The character determines how the structure is aligned in the corresponding direction:
 - ‘-’: aligned on the minimal value of the bounding box,
 - ‘+’: aligned on the maximal value of the bounding box,
 - ‘0’: aligned on the middle value of the bounding box.Any other value will make the alignment in that direction unchanged.
- **point** (*point-like*) – The target point of the alignment.

Returns *Coords* – The *Coords* translated thus that the *alignment bboxPoint()* is at *point*.

Notes

The default parameters translate the *Coords* thus that all points are in the octant with all positive coordinate values.

`Coords.align(alignment = '000')` will center the object around the origin, just like the `centered()` (which is slightly faster). This can however be used for centering around any point.

See also:

`align()` aligning multiple objects with respect to each other.

rotate (*angle, axis=2, around=None, angle_spec=0.017453292519943295*)

Return a copy rotated over angle around axis.

Parameters

- **angle** (float or float *array_like* (3,3)) – If a float, it is the rotation angle, by default in degrees, and the parameters (*angle, axis, angle_spec*) are passed to `rotationMatrix()` to produce a (3,3) rotation matrix. Alternatively, the rotation matrix may be directly provided in the *angle* parameter. The *axis* and *angle_spec* are then ignored.
- **axis** (int (0,1,2) or float *array_like* (3,)) – Only used if *angle* is a float. If provided, it specifies the direction of the rotation axis: either one of 0,1,2 for a global axis, or a vector with 3 components for a general direction. The default (axis 2) is convenient for working with 2D-structures in the x-y plane.
- **around** (float *array_like* (3,)) – If provided, it species a point on the rotation axis. If not, the rotation axis goes through the origin of the global axes.
- **angle_spec** (*float, DEG or RAD, optional*) – Only used if *angle* is a float. The default (DEG) interpretes the angle in degrees. Use RAD to specify the angle in radians.

Returns *Coords* – The *Coords* rotated as specified by the parameters.

Note: `rot()` is a convenient shorthand for `rotate()`.

See also:

`translate()` translate a *Coords*

`affine()` rotate and translate a *Coords*

`arraytools.rotationMatrix()` create a rotation matrix for use in `rotate()`

Examples

```
>>> X = Coords('0123')
>>> print(X.rotate(30))
[[ 0.    0.    0. ]
 [ 0.87  0.5   0. ]
 [ 0.37  1.37  0. ]
 [-0.5   0.87  0. ]]
>>> print(X.rotate(30,axis=0))
[[ 0.    0.    0. ]
 [ 1.    0.    0. ]
 [ 1.    0.87  0.5 ]
 [ 0.    0.87  0.5 ]]
>>> print(X.rotate(30,axis=0,around=[0.,0.5,0.]))
[[ 0.    0.07 -0.25]
 [ 1.    0.07 -0.25]
 [ 1.    0.93  0.25]
 [ 0.    0.93  0.25]]
>>> m = rotationMatrix(30,axis=0)
>>> print(X.rotate(m))
[[ 0.    0.    0. ]
 [ 1.    0.    0. ]
 [ 1.    0.87  0.5 ]
 [ 0.    0.87  0.5 ]]
```

shear (*dir, dir1, skew, inplace=False*)

Return a copy skewed in the direction of a global axis.

This translates points in the direction of a global axis, over a distance dependent on the coordinates along another axis.

Parameters

- **dir** (*int (0, 1, 2)*) – Global axis in which direction the points are translated.
- **dir1** (*int (0, 1, 2)*) – Global axis whose coordinates determine the length of the translation.
- **skew** (*float*) – Multiplication factor to the coordinates `dir1` defining the translation distance.
- **inplace** (*bool, optional*) – If True, the coordinates are translated in-place.

Notes

This replaces the coordinate `dir` with `(dir + skew * dir1)`. If `dir` and `dir1` are different, rectangular shapes in the plane (`dir,dir1`) are thus skewed along the direction `dir` into parallelogram shapes. If `dir` and `dir1` are the same direction, the effect is that of scaling in the `dir` direction.

Examples

```
>>> X = Coords('0123')
>>> print(X.shear(0,1,0.5))
```

(continues on next page)

(continued from previous page)

```
[[ 0.  0.  0. ]
 [ 1.  0.  0. ]
 [ 1.5 1.  0. ]
 [ 0.5 1.  0. ]]
```

reflect (*dir=0, pos=0.0, inplace=False*)

Reflect the coordinates in the direction of a global axis.

Parameters

- **dir** (*int (0, 1, 2)*) – Global axis direction of the reflection (default 0 or x-axis).
- **pos** (*float*) – Offset of the mirror plane from origin (default 0.0)
- **inplace** (*bool, optional*) – If True, the coordinates are translated in-place.

Returns *Coords* – A mirror copy with respect to the plane perpendicular to axis *dir* and placed at coordinate *pos* along the *dir* axis.

Examples

```
>>> X = Coords('012')
>>> print(X)
[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 1.  1.  0.]]
>>> print(X.reflect(0))
[[ 0.  0.  0.]
 [-1.  0.  0.]
 [-1.  1.  0.]]
>>> print(X.reflect(1,0.5))
[[ 0.  1.  0.]
 [ 1.  1.  0.]
 [ 1.  0.  0.]]
>>> print(X.reflect([0,1],[0.5,0.]))
[[ 1.  0.  0.]
 [ 0.  0.  0.]
 [ 0. -1.  0.]]
```

affine (*mat, vec=None*)

Perform a general affine transformation.

Parameters

- **mat** (*float array_like (3,3)*) – Matrix used in post-multiplication on a row vector to produce a new vector. The matrix can express scaling and/or rotation or a more general (affine) transformation.
- **vec** (*float array_like (3,)*) – Translation vector to add after the transformation with *mat*.

Returns *Coords* – A *Coords* with same shape as *self*, but with coordinates given by *self * mat + vec*. If *mat* is a rotation matrix or a uniform scaling plus rotation, the full operation performs a rigid rotation plus translation of the object.

Examples

```

>>> X = Coords('0123')
>>> S = array([[2.,0.,0.],[0.,3.,0.],[0.,0.,4.]]) # non-uniform scaling
>>> R = rotationMatrix(90.,2) # rotation matrix
>>> T = [20., 0., 2.] # translation
>>> M = dot(S,R) # combined scaling and rotation
>>> print(X.affine(M,T))
[[ 20.  0.  2.]
 [ 20.  2.  2.]
 [ 17.  2.  2.]
 [ 17.  0.  2.]]

```

toCS (*cs*)

Transform the coordinates to another CoordSys.

Parameters *cs* (*CoordSys* object) – Cartesian coordinate system in which to take the coordinates of the current Coords object.

Returns *Coords* – A Coords object identical to self but having global coordinates equal to the coordinates of self in the *cs* CoordSys axes.

Note: This returns the coordinates of the original points in another CoordSys. If you use these coordinates as points in the global axes, the transformation of the original points to these new ones is the inverse transformation of the transformation of the global axes to the *cs* coordinate system.

See also:

fromCS() the inverse transformation

Examples

```

>>> X = Coords('01')
>>> print(X)
[[ 0.  0.  0.]
 [ 1.  0.  0.]]
>>> from pyformex.coordsys import CoordSys
>>> CS = CoordSys(oab=[[0.5,0.,0.],[1.,0.5,0.],[0.,1.,0.]])
>>> print(CS)
CoordSys: trl=[ 0.5  0.  0. ]; rot=[[ 0.71  0.71  0. ]
                                   [-0.71  0.71  0. ]
                                   [ 0.   -0.   1.  ]]
>>> print(X.toCS(CS))
[[-0.35  0.35  0. ]
 [ 0.35 -0.35  0.  ]]
>>> print(X.toCS(CS).fromCS(CS))
[[ 0.  0.  0.]
 [ 1. -0.  0.]]

```

fromCS (*cs*)

Transform the coordinates from another CoordSys to global axes.

Parameters *cs* (*CoordSys* object) – Cartesian coordinate system in which the current coordinate values are taken.

Returns *Coords* – A *Coords* object with the global coordinates of the same points as the input coordinates represented in the *cs* *CoordSys* axes.

See also:

`toCS()` the inverse transformation

Examples: see `toCS()`

transformCS (*cs*, *cs0=None*)

Perform a coordinate system transformation on the *Coords*.

This method transforms the *Coords* object by the transformation that turns one coordinate system into a another.

Parameters

- **cs** (*CoordSys*) – The final coordinate system.
- **cs0** (*CoordSys*, optional) – The initial coordinate system. If not provided, the global coordinate system is used.

Returns *Coords* – The input *Coords* transformed by the same affine transformation that turns the axes of the coordinate system *cs0* into those of the system *cs*.

Notes

For example, with the default *cs0* and a *cs* *CoordSys* created with the points

```

0. 1. 0.
-1. 0. 0.
0. 0. 1.
0. 0. 0.
```

the `transformCS` results in a rotation of 90 degrees around the z-axis.

See also:

`toCS()` transform coordinates to another CS

`fromCS()` transform coordinates from another CS

position (*x*, *y*)

Position a *Coords* so that 3 points *x* are aligned with *y*.

Aligning 3 points *x* with 3 points *y* is done by a rotation and translation in such way that

- point *x0* coincides with point *y0*,
- line *x0,x1* coincides with line *y0,y1*
- plane *x0,x1,x2* coincides with plane *y0,y1,y2*

Parameters

- **x** (float *array_like* (3,3)) – Original coordinates of three non-collinear points. These points can be be part of the *Coords* or not.
- **y** (float *array_like* (3,3)) – Final coordinates of the three points.

Returns *Coords* – The input *Coords* rotated and translated thus that the points *x* are aligned with *y*.

Notes

This is a convenient shorthand for `self.affine(*trfmat(x, y))`.

See also:

`arraytools.trfmat()` compute the transformation matrices from points x to y

`affine()` general transform using rotation and translation

Examples

```
>>> X = Coords([[0,0,0],[1,0,0],[1,1,0]])
>>> Y = Coords([[1,1,1],[1,10,1],[1,1,100]])
>>> print(X.position(X,Y))
[[ 1.  1.  1.]
 [ 1.  2.  1.]
 [ 1.  2.  2.]
```

cylindrical (*dir*=(0, 1, 2), *scale*=(1.0, 1.0, 1.0), *angle_spec*=0.017453292519943295)

Convert from cylindrical coordinates to cartesian.

A cylindrical coordinate system is defined by a longitudinal axis axis (z) and a radial axis (r). The cylindrical coordinates of a point are:

- r: the radial distance from the z-axis,
- theta: the circumferential angle measured positively around the z-axis starting from zero at the (r-z) halfplane,
- z: the axial distance along the z-axis,

This function interpretes the 3 coordinates of the points as (r,theta,z) values and computes the corresponding global cartesian coordinates (x,y,z).

Parameters

- **dir** (*tuple of 3 ints, optional*) – If provided, it is a permutation of (0,1,2) and specifies which of the current coordinates are interpreted as resp. distance(r), angle(theta) and height(z). Default order is (r,theta,z). Beware that if the permutation is not conserving the order of the axes, a left-handed system results, and the Coords will appear mirrored in the right-handed systems exclusively used by pyFormex
- **scale** (*tuple of 3 floats, optional*) – Scaling factors that are applied on the values prior to make the conversion from cylindrical to cartesian coordinates. These factors are always given in the order (r,theta,z), irrespective of the permutation by *dir*.
- **angle_spec** (*float, DEG or RAD, optional*) – Multiplication factor for angle coordinates. The default (DEG) interpretes the angle in degrees. Use RAD to specify the angle in radians.

Returns *Coords* – The global coordinates of the points that were specified with cylindrical coordinates as input.

Notes

The scaling can also be applied independently prior to transforming. `X.cylindrical(scale=s)` is equivalent with `X.scale(s).cylindrical()`. The scale option is provided here because in many cases you need at least to scale the theta direction to have proper angle values.

See also:

`hyperCylindrical()` similar but allowing scaling as function of angle

`toCylindrical()` inverse transformation (cartesian to cylindrical)

Examples

We want to create six points on a circle with radius 2. We start by creating the points in cylindrical coordinates with unit distances.

```
>>> X = Coords('1'+ '2'*5)
>>> print(X)
[[ 1.  0.  0.]
 [ 1.  1.  0.]
 [ 1.  2.  0.]
 [ 1.  3.  0.]
 [ 1.  4.  0.]
 [ 1.  5.  0.]
```

Remember these are (r,theta,z) coordinates of the points. So we will scale the r-direction with 2 (the target radius) and the angular direction theta with $360/6 = 60$. Then we get the cartesian coordinates of the points from

```
>>> Y = X.cylindrical(scale=(2., 60., 1.))
>>> print(Y)
[[ 2.    0.    0. ]
 [ 1.    1.73  0. ]
 [-1.    1.73  0. ]
 [-2.   -0.    0. ]
 [-1.   -1.73  0. ]
 [ 1.   -1.73  0. ]]
```

Going back to cylindrical coordinates yields

```
>>> print(Y.toCylindrical())
[[ 2.    0.    0.]
 [ 2.   60.    0.]
 [ 2.  120.    0.]
 [ 2. -180.    0.]
 [ 2. -120.    0.]
 [ 2.  -60.    0.]
```

This differs from the original input X because of the scaling factors, and the wrapping around angles are reported in the range [-180,180].

`hyperCylindrical` (*dir*=(0, 1, 2), *scale*=(1.0, 1.0, 1.0), *rfunc*=None, *zfunc*=None, *angle_spec*=0.017453292519943295)

Convert cylindrical coordinates to cartesian with advanced scaling.

This is similar to `cylindrical()` but allows the specification of two functions defining extra scaling factors for the r and z directions that are dependent on the theta value.

Parameters

- `scale, angle_spec` ((*dir*,)–

- **rfunc** (*callable, optional*) – Function $r(\theta)$ taking one, float parameter and returning a float. Like `scale[0]` it is multiplied with the provided r values before converting them to cartesian coordinates.
- **zfunc** (*callable, optional*) – Function $z(\theta)$ taking one float parameter and returning a float. Like `scale[2]` it is multiplied with the provided z values before converting them to cartesian coordinates.

See also:

`cylindrical()` similar but without the `rfunc` and `zfunc` options.

toCylindrical (*dir=(0, 1, 2), angle_spec=0.017453292519943295*)

Converts from cartesian to cylindrical coordinates.

Returns a `Coords` where the values are the coordinates of the input points in a cylindrical coordinate system. The three axes of the `Coords` then correspond to (r, θ, z) .

Parameters

- **dir** (*tuple of ints*) – A permutation of $(0,1,2)$ specifying which of the global axes are the radial, circumferential and axial direction of the cylindrical coordinate system. Make sure to keep the axes ordering in order to get a right-handed system.
- **angle_spec** (*float, DEG or RAD, optional*) – Multiplication factor for angle coordinates. The default (DEG) returns angles in degrees. Use RAD to return angles in radians.

Returns *Coords* – The cylindrical coordinates of the input points.

See also:

`cylindrical()` conversion from cylindrical to cartesian coordinates

Examples

see `cylindrical()`

spherical (*dir=(0, 1, 2), scale=(1.0, 1.0, 1.0), angle_spec=0.017453292519943295, colat=False*)

Convert spherical coordinates to cartesian coordinates.

Consider a spherical coordinate system with the global xy -plane as its equatorial plane and the z -axis as axis. The zero meridional halfplane is taken along the positive x -axis. The spherical coordinates of a point are:

- the longitude (θ): the circumferential angle, measured around the z -axis from the zero-meridional halfplane to the meridional plane containing the point: this angle normally ranges from -180 to $+180$ degrees (or from 0 to 360);
- the latitude (ϕ): the elevation angle of the point's position vector, measured from the equatorial plane, positive when the point is at the positive side of the plane: this angle is normally restricted to the range from -90 (south pole) to $+90$ (north pole);
- the distance (r): the radial distance of the point from the origin: this is normally positive.

This function interprets the 3 coordinates of the points as (θ, ϕ, r) values and computes the corresponding global cartesian coordinates (x, y, z) .

Parameters

- **dir** (*tuple of 3 ints, optional*) – If provided, it is a permutation of (0,1,2) and specifies which of the current coordinates are interpreted as resp. longitude(theta), latitude(phi) and distance(r). This allows the axis to be aligned with any of the global axes. Default order is (0,1,2), with (0,1) the equatorial plane and 2 the axis. Beware that using a permutation that is not conserving the order of the global axes (0,1,2), may lead to a confusing left-handed system.
- **scale** (*tuple of 3 floats, optional*) – Scaling factors that are applied on the coordinate values prior to making the conversion from spherical to cartesian coordinates. These factors are always given in the order (theta,phi,rz), irrespective of the permutation by *dir*.
- **angle_spec** (*float, DEG or RAD, optional*) – Multiplication factor for angle coordinates. The default (DEG) interpretes the angles in degrees. Use RAD to specify the angles in radians.
- **colat** (*bool*) – If True, the second coordinate is the colatitude instead. The colatitude is the angle measured from the north pole towards the south. In degrees, it is equal to $90 - \text{latitude}$ and ranges from 0 to 180. Applications that deal with regions around the pole may benefit from using this option.

Returns *Coords* – The global coordinates of the points that were specified with spherical coordinates as input.

See also:

toSpherical() the inverse transformation (cartesian to spherical)

cylindrical() similar function for spherical coordinates

Examples

```
>>> X = Coords('0123').scale(90).trl(2,1.)
>>> X
Coords([[ 0.,  0.,  1.],
        [ 90.,  0.,  1.],
        [ 90.,  90.,  1.],
        [ 0.,  90.,  1.]])
>>> X.spherical()
Coords([[ 1.,  0.,  0.],
        [-0.,  1.,  0.],
        [ 0., -0.,  1.],
        [-0., -0.,  1.]])
```

Note that the last two points, though having different spherical coordinates, are coinciding at the north pole.

superSpherical (*n=1.0, e=1.0, k=0.0, dir=(0, 1, 2), scale=(1.0, 1.0, 1.0), angle_spec=0.017453292519943295, colat=False*)
Performs a superspherical transformation.

superSpherical is much like *spherical()*, but adds some extra parameters to enable the quick creation of a wide range of complicated shapes. Again, the input coordinates are interpreted as the longitude, latitude and distance in a spherical coordinate system.

Parameters

- **n** (*float, >=0*) – Exponent defining the variation of the distance in north-south (latitude) direction. The default value 1 turns constant r-values into circular meridians. See notes.

- **e** (*float, >=0*) – Exponent defining the variation of the distance in north-south (latitude) direction. The default value 1 turns constant r-values into a circular latitude lines. See notes.
- **k** (*float, -1 < k < 1*) – Eggness factor. If nonzero, creates asymmetric northern and southern hemispheres. Values > 0 enlarge the southern hemisphere and shrink the northern, while negative values yield the opposite.
- **dir** (*tuple of 3 ints, optional*) – If provided, it is a permutation of (0,1,2) and specifies which of the current coordinates are interpreted as resp. longitude(theta), latitude(phi) and distance(r). This allows the axis to be aligned with any of the global axes. Default order is (0,1,2), with (0,1) the equatorial plane and 2 the axis. Beware that using a permutation that is not conserving the order of the global axes (0,1,2), may lead to a confusing left-handed system.
- **scale** (*tuple of 3 floats, optional*) – Scaling factors that are applied on the coordinate values prior to making the conversion from spherical to cartesian coordinates. These factors are always given in the order (theta,phi,rz), irrespective of the permutation by *dir*.
- **angle_spec** (*float, DEG or RAD, optional*) – Multiplication factor for angle coordinates. The default (DEG) interpretes the angles in degrees. Use RAD to specify the angles in radians.
- **colat** (*bool*) – If True, the second coordinate is the colatitude instead. The colatitude is the angle measured from the north pole towards the south. In degrees, it is equal to $90 - \text{latitude}$ and ranges from 0 to 180. Applications that deal with regions around the pole may benefit from using this option.

Raises `ValueError` – If one of *n*, *e* or *k* is out of the acceptable range.

Notes

Values of *n* and *e* should not be negative. Values equal to 1 create a circular shape. Other values keep the radius at angles corresponding to multiples of 90 degrees, while the radius at the intermediate 45 degree angles will be maximally changed. Values larger than 1 shrink at 45 degrees directions, while lower values increase it. A value 2 creates a straight line between the 90 degrees points (the radius at 45 degrees being reduced to $1/\sqrt{2}$).

See also example SuperShape.

Examples

```
>>> X = Coords('02222').scale(22.5).trl(2,1.)
>>> X
Coords([[ 0. ,  0. ,  1. ],
        [ 0. , 22.5,  1. ],
        [ 0. , 45. ,  1. ],
        [ 0. , 67.5,  1. ],
        [ 0. , 90. ,  1. ]])
>>> X.superSpherical(n=3).toSpherical()
Coords([[ 90. ,  0. ,  1. ],
        [ 85.93,  0. ,  0.79],
        [ 45. ,  0. ,  0.5 ],
        [ 4.07,  0. ,  0.79],
        [-0. , -0. ,  1. ]])
```

The result is smaller radius at angle 45.

toSpherical (*dir*=[0, 1, 2], *angle_spec*=0.017453292519943295)

Converts from cartesian to spherical coordinates.

Returns a Coords where the values are the coordinates of the input points in a spherical coordinate system. The three axes of the Coords then correspond to (theta, phi, r).

Parameters

- **dir** (*tuple of ints*) – A permutation of (0,1,2) specifying how the spherical coordinate system is oriented in the global axes. The last value is the axis of the system; the first two values are the equatorial plane; the first and last value define the meridional zero plane. Make sure to preserve the axes ordering in order to get a right-handed system.
- **angle_spec** (*float, DEG or RAD, optional*) – Multiplication factor for angle coordinates. The default (DEG) returns angles in degrees. Use RAD to return angles in radians.

Returns *Coords* – The spherical coordinates of the input points.

See also:

spherical() conversion from spherical to cartesian coordinates

Examples

See *superSpherical()*

circulize (*n*)

Transform sectors of a regular polygon into circular sectors.

Parameters *n* (*int*) – Number of edges of the regular polygon.

Returns *Coords* – A Coords where the points inside each sector of a n-sided regular polygon around the origin are reposition to fill a circular sector. The polygon is in the x-y-plane and has a vertex on the x-axis.

Notes

Points on the x-axis and on radii at $i * 360 / n$ degrees are not moved. Points on the bisector lines between these radii are move maximally outward. Points on a regular polygon will become points on a circle if circulized with parameter n equal to the number of sides of the polygon.

Examples

```
>>> Coords([[1.,0.],[0.5,0.5],[0.,1.]]) .circulize(4)
Coords([[ 1. ,  0. ,  0. ],
        [ 0.71,  0.71,  0. ],
        [-0. ,  1. ,  0. ]])
```

bump (*dir, a, func=None, dist=None, xb=1.0*)

Create a 1-, 2-, or 3-dimensional bump in a Coords.

A bump is a local modification of the coordinates of a collection of points. The bump can be 1-, 2- or 3-dimensional, meaning that the intensity of the coordinate modification varies in 1, 2 or 3 axis directions. In all cases, the bump only changes one coordinate of the points. This method can produce various effects, but one of the most common uses is to force a surface to be indented at some point.

Parameters

- **dir** (*int*, one of (0, 1, 2)) – The axis of the coordinates to be modified.
- **a** (*point* (3,)) – The point that sets the bump location and intensity.
- **func** (*callable*, *optional*) – A function that returns the bump intensity in function of the distance from the bump point *a*. The distance is the Euclidean distance over all directions except *dir*. The function takes a single (positive) float value and returns a float (the bump intensity). Its value should not be zero at the origin. The function may include constants, which can be specified as *xb*. If no function is specified, the default function will be used: `lambda x: where(x<xb, 1.-(x/3)**2, 0)` This makes the bump quadratically die out over a distance *xb*.
- **dist** (*int or tuple of ints*, *optional*) – Specifies how the distance from points to the bump point *a* is measured. It can be a single axis number (0,1,2) or a tuple of two or three axis numbers. If a single axis, the bump will vary only in one direction and distance is measured along that direction and is signed. The bump will only vary in that direction. If two or three axes, distance is the (always positive) euclidean distance over these directions and the bump will vary in these directions. Default value is the set of 3 axes minus the direction of modification *dir*.
- **xb** (*float or list of floats*) – Constant(s) to be used in *func*. Often, this includes the distance over which the bump will extend. The default bump function will reach zero at this distance.

Returns *Coords* – A *Coords* with same shape as input, but having a localized change of coordinates as specified by the parameters.

Notes

This function replaces the *bump1* and *bump2* functions in older pyFormex versions. The default value of *dist* makes it work like *bump2*. Specifyin a single axis for *dist* makes it work like *bump1*.

See also examples BaumKuchen, Circle, Clip, Novation

Examples

One-dimensional bump in a linear set of points.

```
>>> X = Coords(arange(6).reshape(-1,1))
>>> X.bump1(1, [3., 5., 0.], dist=0)
Coords([[ 0.,  0.,  0.],
        [ 1.,  0.,  0.],
        [ 2.,  0.,  0.],
        [ 3.,  5.,  0.],
        [ 4.,  0.,  0.],
        [ 5.,  0.,  0.]])
>>> X.bump(1, [3., 5., 0.], dist=0, xb=3.)
Coords([[ 0. ,  0. ,  0. ],
        [ 1. ,  2.78,  0. ],
        [ 2. ,  4.44,  0. ],
        [ 3. ,  5. ,  0. ],
        [ 4. ,  4.44,  0. ],
        [ 5. ,  2.78,  0. ]])
```

Create a grid a points in xz-plane, with a bump in direction y with a maximum 5 at x=1.5, z=0., extending over a distance 2.5.

```

>>> X = Coords(arange(4).reshape(-1,1)).replicate(4,dir=2)
>>> X.bump(1,[1.5,5.,0.],xb=2.5)
Coords([[ 0. , 3.75, 0. ],
        [ 1. , 4.86, 0. ],
        [ 2. , 4.86, 0. ],
        [ 3. , 3.75, 0. ]],
<BLANKLINE>
        [[ 0. , 3.19, 1. ],
        [ 1. , 4.31, 1. ],
        [ 2. , 4.31, 1. ],
        [ 3. , 3.19, 1. ]],
<BLANKLINE>
        [[ 0. , 0. , 2. ],
        [ 1. , 2.64, 2. ],
        [ 2. , 2.64, 2. ],
        [ 3. , 0. , 2. ]],
<BLANKLINE>
        [[ 0. , 0. , 3. ],
        [ 1. , 0. , 3. ],
        [ 2. , 0. , 3. ],
        [ 3. , 0. , 3. ]])

```

flare (*xf, f, dir=(0, 2), end=0, exp=1.0*)

Create a flare at the end of a *Coords* block.

A flare is a local change of geometry (widening, narrowing) at the end of a structure.

Parameters

- **xf** (*float*) – Length over which the local change occurs, measured along `dir[0]`.
- **f** (*float*) – Maximum amplitude of the flare, in the direction `dir[1]`.
- **dir** (*tuple of two ints (0,1,2)*) – Two axis designations. The first axis defines the direction along which the flare decays. The second is the direction of the coordinate modification.
- **end** (*0 or 1*) – With `end=0`, the flare exists at the end with the smallest coordinates in `dir[0]` direction; with `end=1`, at the end with the highest coordinates.
- **exp** (*float*) – Exponent setting the speed of decay of the flare. The default makes the flare change linearly over the length *f*.

Returns *Coords* – A *Coords* with same shape as the input, but having a localized change of coordinates at one end of the point set.

Examples

```

>>> Coords(arange(6).reshape(-1,1)).flare(3.,1.6,(0,1),0)
Coords([[ 0. , 1.6 , 0. ],
        [ 1. , 1.07, 0. ],
        [ 2. , 0.53, 0. ],
        [ 3. , 0. , 0. ],
        [ 4. , 0. , 0. ],
        [ 5. , 0. , 0. ]])

```

map (*func*)

Map a *Coords* by a 3-D function.

This allows any mathematical transformation being applied to the coordinates of the Coords.

Parameters `func` (*callable*) – A function taking three float arguments (x,y,z coordinates of a point) and returning a tuple of three float values: the new coordinate values to replace (x,y,z) .

The function must be applicable to NumPy arrays, so it should only include numerical operations and functions understood by the numpy module.

Often an inline lambda function is used, but a normally defined function will work as well.

Returns *Coords object* – The input Coords mapped through the specified function

See also:

`map1()` apply a 1-dimensional mapping to one coordinate direction

`mapd()` map one coordinate by a function of the distance to a point

Notes

See also examples Cones, Connect, HorseTorse, Manantiales, Mobius, ScallopDome

Examples

```
>>> print(Coords([[1., 1., 1.]]) .map(lambda x, y, z: [2*x, 3*y, 4*z]))
[[ 2.  3.  4.]]
```

`map1` (*dir, func, x=None*)

Map one coordinate by a 1-D function of one coordinate.

Parameters

- `dir` (*int (0, 1 or 2)*) – The coordinate axis to be modified.
- `func` (*callable*) – Function taking a single float argument (the coordinate x) and returning a float value: the new coordinate to replace the *dir* coordinate.

The function must be applicable to NumPy arrays, so it should only include numerical operations and functions understood by the numpy module.

Often an inline lambda function is used, but a normally defined function will work as well.

- `x` (*int (0, 1, 2), optional*) – If provided, specifies the coordinate that is used as argument in *func*. Default is to use the same as *dir*.

Returns *Coords object* – The input Coords where the *dir* coordinate has been mapped through the specified function.

See also:

`map()` apply a general 3-dimensional mapping function

`mapd()` map one coordinate by a function of the distance to a point

Notes

See also example SplineSurface

Examples

```
>>> Coords(arange(4).reshape(-1,1)).map1(1, lambda x:0.1*x, 0)
Coords([[ 0. ,  0. ,  0. ],
        [ 1. ,  0.1,  0. ],
        [ 2. ,  0.2,  0. ],
        [ 3. ,  0.3,  0. ]])
```

mapd (*dir, func, point=(0.0, 0.0, 0.0), dist=None*)

Map one coordinate by a function of the distance to a point.

Parameters

- **dir** (*int (0, 1 or 2)*) – The coordinate that will be replaced with `func(d)`, where d is calculated as the distance to *point*.
- **func** (*callable*) – Function taking one float argument (distance to *point*) and returning a float: the new value for the *dist* coordinate. *dir* coordinate.

The function must be applicable to NumPy arrays, so it should only include numerical operations and functions understood by the `numpy` module.

Often an inline lambda function is used, but a normally defined function will work as well.

- **point** (float *array_like (3,)*) – The point to where the distance is computed.
- **dist** (*int or tuple of ints (0, 1, 2)*) – The coordinate directions that are used to compute the distance to *point*. The default is to use 3-D distances.

Examples

Map a regular 4x4 point grid in the xy-plane onto a sphere with radius 1.5 and center at the corner of the grid.

```
>>> from .simple import regularGrid
>>> X = Coords(regularGrid([0.,0.],[1.,1.],[3,3]))
>>> X.mapd(2, lambda d:sqrt(1.5**2-d**2), X[0,0],[0,1])
Coords([[ [ 0. ,  0. ,  1.5 ],
          [ 0.33,  0. ,  1.46],
          [ 0.67,  0. ,  1.34],
          [ 1. ,  0. ,  1.12]],
        <BLANKLINE>
          [ [ 0. ,  0.33,  1.46],
          [ 0.33,  0.33,  1.42],
          [ 0.67,  0.33,  1.3 ],
          [ 1. ,  0.33,  1.07]],
        <BLANKLINE>
          [ [ 0. ,  0.67,  1.34],
          [ 0.33,  0.67,  1.3 ],
          [ 0.67,  0.67,  1.17],
          [ 1. ,  0.67,  0.9 ]],
        <BLANKLINE>
          [ [ 0. ,  1. ,  1.12],
          [ 0.33,  1. ,  1.07],
          [ 0.67,  1. ,  0.9 ],
          [ 1. ,  1. ,  0.5 ]]])
```

copyAxes (*i, j, other=None*)

Copy the coordinates along the axes *j* to the axes *i*.

Parameters

- **i** (*int* (0,1,2) or *tuple of ints* (0,1,2)) – One or more coordinate axes that should have replaced their coordinates by those along the axes *j*.
- **j** (*int* (0,1,2) or *tuple of ints* (0,1,2)) – One or more axes whose coordinates should be copied along the axes *i*. *j* should have the same type and length as *i*.
- **other** (*Coords object, optional*) – If provided, this is the source Coords for the coordinates. It should have the same shape as self. The default is to take the coords from self.

Returns *Coords object* – A Coords where the coordinates along axes *i* have been replaced by those along axes *j*.

Examples

```
>>> X = Coords([[1],[2]]).trl(2,5)
>>> X
Coords([[ 1.,  0.,  5.],
        [ 2.,  0.,  5.]])
>>> X.copyAxes(1,0)
Coords([[ 1.,  1.,  5.],
        [ 2.,  2.,  5.]])
>>> X.copyAxes((0,1),(1,0))
Coords([[ 0.,  1.,  5.],
        [ 0.,  2.,  5.]])
>>> X.copyAxes((0,1,2),(1,2,0))
Coords([[ 0.,  5.,  1.],
        [ 0.,  5.,  2.]])
```

swapAxes(*i,j*)

Swap two coordinate axes.

Parameters

- **i** (*int* (0,1,2)) – First coordinate axis
- **j** (*int* (0,1,2)) – Second coordinate axis

Returns *Coords* – A Coords with interchanged *i* and *j* coordinates.

Warning: `Coords.swapAxes` merely changes the order of the elements along the last axis of the ndarray. This is quite different from `numpy.ndarray.swapaxes()`, which is inherited by the Coords class. The latter method interchanges the array axes of the ndarray, and will not yield a valid Coords object if the interchange involves the last axis.

Notes

This is equivalent with `self.copyAxes((i,j),(j,i))`

Swapping two coordinate axes has the same effect as mirroring against the bisector plane between the two axes.

Examples

```
>>> X = Coords(arange(6).reshape(-1,3))
>>> X
Coords([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.]])
>>> X.swapAxes(2,0)
Coords([[ 2.,  1.,  0.],
        [ 5.,  4.,  3.]])
>>> X.swapaxes(1,0)
array([[ 0.,  3.],
       [ 1.,  4.],
       [ 2.,  5.]])
```

rollAxes ($n=1$)

Roll the coordinate axes over the given amount.

Parameters n (*int*) – Number of positions to roll the axes. With the default (1), the old axes (0,1,2) become the new axes (2,0,1).

Returns *Coords* – A *Coords* where the coordinate axes of the points have been rolled over n positions.

Notes

`X.rollAxes(1)` can also be obtained by `X.copyAxes((0,1,2),(2,0,1))`. It is also equivalent with a rotation over -120 degrees around the trisectrice of the first quadrant.

Examples

```
>>> X = Coords('0123')
>>> X
Coords([[ 0.,  0.,  0.],
        [ 1.,  0.,  0.],
        [ 1.,  1.,  0.],
        [ 0.,  1.,  0.]])
>>> X.rollAxes(1)
Coords([[ 0.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  1.,  1.],
        [ 0.,  0.,  1.]])
>>> X.rotate(120,axis=[1.,1.,1.])
Coords([[ 0.,  0.,  0.],
        [-0.,  1., -0.],
        [-0.,  1.,  1.],
        [-0., -0.,  1.]])
```

projectOnPlane ($n=2, P=(0.0, 0.0, 0.0)$)

Project a *Coords* on a plane.

Creates a parallel projection of the *Coords* on a plane.

Parameters

- n (*int* (0,1,2) or *float array_like* (3,)) – The normal direction to the plane on which to project the *Coords*. If an *int*, it is a global axis.

- **P** (float *array_like* (3,)) – A point in the projection plane, by default the global origin.

Returns *Coords* – The points of the *Coords* projected on the specified plane.

Notes

For projection on a plane parallel to a coordinate plane, it is far more efficient to specify the normal by an axis number rather than by a three component vector.

This method will also work if any or both of P and n have the same shape as self, or can be reshaped to the same shape. This will project each point on its individual plane.

See also example `BorderExtension`

Examples

```
>>> X = Coords(arange(6).reshape(2,3))
>>> X.projectOnPlane(0,P=[2.5,0.,0.])
Coords([[ 2.5,  1. ,  2. ],
        [ 2.5,  4. ,  5. ]])
>>> X.projectOnPlane([1.,1.,0.])
Coords([[ -0.5,  0.5,  2. ],
        [ -0.5,  0.5,  5. ]])
```

projectOnSphere (*radius=1.0, center=(0.0, 0.0, 0.0)*)

Project a *Coords* on a sphere.

Creates a central projection of a *Coords* on a sphere.

Parameters

- **radius** (*float, optional*) – The radius of the sphere, default 1.
- **center** (float *array_like* (3,), optional) – The center of the sphere. This point should not be part the the *Coords*. The default is the origin of the global axes.

Returns *Coords* – A *Coords* with the input points projected on the specified sphere.

Notes

This is a central projection from the center of the sphere. If you want a parallel projection on a spherical surface, you can use `map()`. See the Examples there.

Examples

```
>>> X = Coords([[x,x,1.] for x in range(1,4)])
>>> X
Coords([[ 1.,  1.,  1.],
        [ 2.,  2.,  1.],
        [ 3.,  3.,  1.]])
>>> X.projectOnSphere()
Coords([[ 0.58,  0.58,  0.58],
        [ 0.67,  0.67,  0.33],
        [ 0.69,  0.69,  0.23]])
```

projectOnCylinder (*radius=1.0, dir=0, center=[0.0, 0.0, 0.0]*)

Project the Coords on a cylinder with axis parallel to a global axis.

Given a cylinder with axis parallel to a global axis, the points of the Coords are projected from the axis onto the surface of the cylinder. The default cylinder has its axis along the x-axis and a unit radius. No points of the *Coords* should belong to the axis.

Parameters

- **radius** (*float, optional*) – The radius of the sphere, default 1.
- **dir** (*int (0,1,2), optional*) – The global axis parallel to the cylinder’s axis.
- **center** (*float array_like (3,)*, optional) – A point on the axis of the cylinder. Default is the origin of the global axes.

Returns *Coords* – A Coords with the input points projected on the specified cylinder.

Notes

This is a projection from the axis of the cylinder. If you want a parallel projection on a cylindrical surface, you can use *map()*.

Examples

```
>>> X = Coords([[x,x,1.] for x in range(1,4)])
>>> X
Coords([[ 1.,  1.,  1.],
        [ 2.,  2.,  1.],
        [ 3.,  3.,  1.]])
>>> X.projectOnCylinder()
Coords([[ 1. ,  0.71,  0.71],
        [ 2. ,  0.89,  0.45],
        [ 3. ,  0.95,  0.32]])
```

projectOnSurface (*S, dir=0, missing='e', return_indices=False*)

Project a *Coords* on a triangulated surface.

The points of the Coords are projected in the specified direction *dir* onto the surface *S*. If a point has multiple projections in the direction, the one nearest to the original is returned.

Parameters

- **S** (*TriSurface*) – A triangulated surface
- **dir** (*int (0,1,2) or float array_like (3,)*) – The direction of the projection, either a global axis direction or specified as a vector with three components.
- **missing** (*'o', 'r' or 'e'*) – Specifies how to treat cases where the projective line does not intersect the surface:
 - 'o': return the original point,
 - 'r': remove the point from the result. Use *return_indices = True* to find out which original points correspond with the projections.
 - 'e': raise an exception (default).
- **return_indices** (*bool, optional*) – If True, also returns the indices of the points that have a projection on the surface.

Returns

- **x** (*Coords*) – A *Coords* with the projections of the input points on the surface. With *missing='o'*, this will have the same shape as the input, but some points might not actually lie on the surface. With *missing='r'*, the shape will be (npoints,3) and the number of points may be less than the input.
- **ind** (*int array, optional*) – Only returned if *return_indices* is True: an index in the input *Coords* of the points that have a projection on the surface. With *missing='r'*, this gives the indices of the original points corresponding with the projections. With *missing='o'*, this can be used to check which points are located on the surface. The index is sequential, no matter what the shape of the input *Coords* is.

Examples

```
>>> from pyformex import simple
>>> S = simple.sphere().scale(2).tr1([0.,0.,0.2])
>>> x = pattern('0123')
>>> print(x)
[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 1.  1.  0.]
 [ 0.  1.  0.]]
>>> xp = x.projectOnSurface(S, [0.,0.,1.])
>>> print(xp)
[[ 0.    0.   -1.8 ]
 [ 1.    0.   -1.52]
 [ 1.    1.   -1.2 ]
 [ 0.    1.   -1.53]]
```

isopar (*eltype, coords, oldcoords*)

Perform an isoparametric transformation on a *Coords*.

This creates an isoparametric transformation *Isopar* object and uses it to transform the input *Coords*. It is equivalent to:

```
Isopar(eltype, coords, oldcoords).transform(self)
```

See *Isopar* for parameters.

addNoise (*rsize=0.05, asize=0.0*)

Add random noise to a *Coords*.

A random amount is added to each individual coordinate of the *Coords*. The maximum difference of the coordinates from their original value is controlled by two parameters *rsize* and *asize* and will not exceed $asize+rsize*self.maxsize()$.

Parameters

- **rsize** (*float*) – Relative size of the noise compared with the maximum size of the input *Coords*.
- **asize** (*float*) – Absolute size of the noise

Examples

```
>>> X = Coords(arange(6).reshape(2,3))
>>> print((abs(X.addNoise(0.1) - X) < 0.1 * X.sizes()).all())
True
```

replicate (*n*, *dir*=0, *step*=1.0)

Replicate a Coords *n* times with a fixed translation step.

Parameters

- **n** (*int*) – Number of times to replicate the Coords.
- **dir** (*int* (0,1,2) or float *array_like* (3,)) – The translation vector. If an *int*, it specifies a global axis and the translation is in the direction of that axis.
- **step** (*float*) – If *dir* is an *int*, this is the length of the translation. Else, it is a multiplying factor applied to the translation vector.

Returns *Coords* – A Coords with an extra first axis with length *n*. The new shape thus becomes $(n,) + \text{self.shape}$. The first component along the axis 0 is identical to the original Coords. Each following component is equal to the previous translated over $(\text{dir}, \text{step})$, where *dir* and *step* are interpreted just like in the `translate()` method.

Notes

`rep()` is a convenient shorthand for `replicate()`.

Examples

```
>>> Coords([0.,0.,0.]).replicate(4,1,1.2)
Coords([[ 0. ,  0. ,  0. ],
        [ 0. ,  1.2,  0. ],
        [ 0. ,  2.4,  0. ],
        [ 0. ,  3.6,  0. ]])
>>> Coords([0.]).replicate(3,0).replicate(2,1)
Coords([[[ 0.,  0.,  0.],
         [ 1.,  0.,  0.],
         [ 2.,  0.,  0.]],
        <BLANKLINE>
        [[ 0.,  1.,  0.],
         [ 1.,  1.,  0.],
         [ 2.,  1.,  0.]])
```

split ()

Split the Coords in blocks along first axis.

Returns *list of Coords objects* – A list of Coords objects being the subarrays taken along the axis 0. The number of objects in the list is `self.shape[0]` and each Coords has the shape `self.shape[1:]`.

Raises `ValueError` – If `self.ndim < 2`.

Examples

```
>>> Coords(.arange(6).reshape(2,3)).split()
[Coords([ 0.,  1.,  2.]), Coords([ 3.,  4.,  5.])]
```

sort (*order=(0, 1, 2)*)

Sort points in the specified order of their coordinates.

Parameters **order** (*int (0,1,2) or tuple of ints (0,1,2)*) – The order in which the coordinates have to be taken into account during the sorting operation.

Returns *int array* – An index into the sequential point list `self.points()` thus that the points are sorted in order of the specified coordinates.

Examples

```
>>> X = Coords([[5,3,0],[2,4,3],[2,3,3],[5,6,2]])
>>> X.sort()
array([2, 1, 0, 3])
>>> X.sort((2,1,0))
array([0, 3, 2, 1])
>>> X.sort(1)
array([0, 2, 1, 3])
```

boxes (*ppb=1, shift=0.5, minsize=1e-05*)

Create a grid of equally sized boxes spanning the *Coords*.

A regular 3D grid of equally sized boxes is created enclosing all the points of the *Coords*. The size, position and number of boxes are determined from the specified parameters.

Parameters

- **ppb** (*int*) – Average number of points per box. The box sizes and number of boxes will be determined to approximate this number.
- **shift** (*float (0.0 .. 1.0)*) – Relative shift value for the grid. Applying a shift of 0.5 will make the lowest coordinate values fall at the center of the outer boxes.
- **minsize** (*float*) – Absolute minimal size of the boxes, in each coordinate direction.

Returns

- **ox** (*float array (3,)*) – The minimal coordinates of the box grid.
- **dx** (*float array (3,)*) – The box size in the three global axis directions.
- **nx** (*int array (3,)*) – Number of boxes in each of the coordinate directions.

Notes

The primary purpose of this method is its use in the *fuse()* method. The boxes allow to quickly label the points inside each box with an integer value (the box number), so that it becomes easy to find close points by their same label.

Because of the possibility that two very close points fall in different boxes (if they happen to be close to a box border), procedures based on these boxes are often repeated twice, with a different shift value.

Examples

```
>>> X = Coords([[5, 3, 0], [2, 4, 3], [2, 3, 3], [5, 6, 2]])
>>> print(*X.bboxes())
[ 0.5  1.5 -1.5] [ 3.  3.  3.] [2 2 2]
>>> print(* X.bboxes(shift=0.1))
[ 1.7  2.7 -0.3] [ 3.  3.  3.] [2 2 2]
>>> X = Coords([[1., 1., 0.], [1.001, 1., 0.], [1.1, 1., 0.]])
>>> print(*X.bboxes())
[ 0.98  0.98 -0.02] [ 0.03  0.03  0.03] [4 1 1]
```

fuse (*ppb=1, shift=0.5, rtol=1e-05, atol=1e-08, repeat=True*)

Find (almost) coinciding points and return a compressed set.

This method finds the points that are very close to each other and replaces them with a single point. See Notes below for explanation about the method being used and the parameters being used. In most cases, *atol* and *rtol* are probably the only ones you want to change from the defaults. Two points are considered the same if all their coordinates differ less than the maximum of *atol* and *rtol* * *self.maxsize*().

Parameters

- **ppb** (*int, optional*) – Average number of points per box. The box sizes and number of boxes will be determined to approximate this number.
- **shift** (*float (0.0 .. 1.0), optional*) – Relative shift value for the box grid. Applying a shift of 0.5 will make the lowest coordinate values fall at the center of the outer boxes.
- **rtol** (*float, optional*) – Relative tolerance used when considering two points for fusing.
- **atol** (*float, optional*) – Absolute tolerance used when considering two points for fusing.
- **repeat** (*bool, optional*) – If True, repeat the procedure with a second shift value.

Returns

- **coords** (*Coords object (npts,3)*) – The unique points obtained from merging the very close points of a Coords.
- **index** (*int array*) – An index in the unique coordinates array *coords* for each of the original points. The shape of the index array is equal to the point shape of the input Coords (*self.pshape*()). All the values are in the range 0..npts.

Note: From the return values *coords[index]* will restore the original Coords (with accuracy equal to the tolerance used in the fuse operation)

Notes

The procedure works by first dividing the 3D space in a number of equally sized boxes, with a average population of *ppb* points. The arguments *ppb* and *shift* are passed to *bboxes*() for this purpose. The boxes are identified by 3 integer coordinates, from which a unique integer scalar is computed, which is then used to sort the points. Finally only the points inside the same box need to be compared. Two points are considered equal if all their coordinates differ less than the maximum of *atol* and *rtol* * *self.maxsize*(). Points considered very close are replaced by a single one, and an index is kept from the original points to the new list of points.

Running the procedure once does not guarantee finding all close nodes: two close nodes might be in adjacent boxes. The performance hit for testing adjacent boxes is rather high, and the probability of separating two close nodes with the computed box limits is very small. Therefore, the most sensible way is to run the procedure twice, with a different shift value (they should differ more than the tolerance). Specifying `repeat=True` will automatically do this with a second shift value equal to `shift+0.25`.

Because fusing points is a very important and frequent step in many geometrical modeling and conversion procedures, the core part of this function is available in a C as well as a Python version, in the module `pyformex.lib.misc`. The much faster C version will be used if available.

Examples

```
>>> X = Coords([[1.,1.,0.],[1.001,1.,0.],[1.1,1.,0.]])
>>> x,e = X.fuse(atol=0.01)
>>> print(x)
[[ 1.  1.  0. ]
 [ 1.1 1.  0. ]]
>>> print(e)
[0 0 1]
>>> allclose(X,x[e],atol=0.01)
True
```

`unique (**kargs)`

Returns the unique points after fusing.

This is just like `fuse()` and takes the same arguments, but only returns the first argument: the unique points in the `Coords`.

`adjust (**kargs)`

Find (almost) identical nodes and adjust them to be identical.

This is like the `fuse()` operation, but it does not fuse the close neighbours to a single point. Instead it adjust the coordinates of the points to be identical.

The parameters are the same as for the `fuse()` method.

Returns `Coords` – A `Coords` with the same shape as the input, but where close points now have identical coordinates.

Examples

```
>>> X = Coords([[1.,1.,0.],[1.001,1.,0.],[1.1,1.,0.]])
>>> print(X.adjust(atol=0.01))
[[ 1.  1.  0. ]
 [ 1.  1.  0. ]
 [ 1.1 1.  0. ]]
```

`match (coords, **kargs)`

Match points in another `Coords` object.

Find the points from another `Coords` object that coincide with (or are very close to) points of `self`. This method works by concatenating the serialized point sets of both `Coords` and then fusing them.

Parameters

- **coords** (`Coords`) – The `Coords` object to compare the points with.

- ****kargs** (*keyword arguments*) – Keyword arguments passed to the `fuse()` method.

Returns *1-dim int array* – The array has a length of `coords.npoints()`. For each point in `coords` it holds the index of a point in `self` coinciding with it, or a value -1 if there is no matching point. If there are multiple matching points in `self`, it is undefined which one will be returned. To avoid this ambiguity, you can first fuse the points of `self`.

See also:

`hasMatch()`, `fuse()`

Examples

```
>>> X = Coords([[1.], [2.], [3.], [1.]])
>>> Y = Coords([[1.], [4.], [2.00001]])
>>> print(X.match(Y))
[ 0 -1  1]
```

hasMatch (*coords*, ***kargs*)

Find out which points are also in another Coords object.

Find the points from `self` that coincide with (or are very close to) some point of `coords`. This method is very similar to `match()`, but does not give information about which point of `self` matches which point of `coords`.

Parameters

- **coords** (*Coords*) – The Coords object to compare the points with.
- ****kargs** (*keyword arguments*) – Keyword arguments passed to the `fuse()` method.

Returns *int array* – A 1-dim int array with the unique sorted indices of the points in `self` that have a (nearly) matching point in `coords`.

Warning: If multiple points in `self` coincide with the same point in `coords`, only one index will be returned for this case. To avoid this, you can fuse `self` before using this method.

See also:

`match()`

Examples

```
>>> X = Coords([[1.], [2.], [3.], [1.]])
>>> Y = Coords([[1.], [4.], [2.00001]])
>>> print(X.hasMatch(Y))
[0 1]
```

append (*coords*)

Append more coords to a Coords object.

The appended coords should have matching dimensions in all but the first axis.

Parameters **coords** (*Coords object*) – A Coords having a shape with `shape[1:]` equal to `self.shape[1:]`.

Returns *Coords* – The concatenated *Coords* object (self,coords).

Notes

This is comparable to `numpy.append()`, but the result is a *Coords* object, the default axis is the first one instead of the last, and it is a method rather than a function.

See also:

concatenate() concatenate a list of *Coords*

Examples

```
>>> X = Coords([[1],[2]])
>>> Y = Coords([[3],[4]])
>>> X.append(Y)
Coords([[ 1.,  0.,  0.],
        [ 2.,  0.,  0.],
        [ 3.,  0.,  0.],
        [ 4.,  0.,  0.]])
```

classmethod *concatenate* (*L*, *axis=0*)

Concatenate a list of *Coords* objects.

Class method to concatenate a list of *Coords* along the given axis.

Parameters **L** (*list of Coords objects*) – The *Coords* objects to be concatenated. All should have the same shape except for the length of the specified axis.

Returns *Coords* – A *Coords* with at least two dimensions, even when the list contains only a single *Coords* with a single point, or is empty.

Raises *ValueError* – If the shape of the *Coords* in the list do not match or if concatenation along the last axis is attempted.

Notes

This is a class method. It is commonly invoked as `Coords.concatenate`, and if used as a method on a *Coords* object, that object will not be included in the list.

It is like `numpy.concatenate()` (which it uses internally), but makes sure to return *Coords* object, and sets the first axis as default instead of the last (which would not make sense).

See also:

append() append a *Coords* to self

Examples

```
>>> X = Coords([1.,1.,0.])
>>> Y = Coords([[2.,2.,0.],[3.,3.,0.]])
>>> print(Coords.concatenate([X,Y]))
[[ 1.  1.  0.]
 [ 2.  2.  0.]
```

(continues on next page)

(continued from previous page)

```

[ 3. 3. 0.]
>>> print(Coords.concatenate([X,X]))
[[ 1. 1. 0.]
 [ 1. 1. 0.]]
>>> print(Coords.concatenate([X]))
[[ 1. 1. 0.]]
>>> print(Coords.concatenate([Y]))
[[ 2. 2. 0.]
 [ 3. 3. 0.]]
>>> print(X.concatenate([Y]))
[[ 2. 2. 0.]
 [ 3. 3. 0.]]
>>> Coords.concatenate([])
Coords([], shape=(0, 3))
>>> Coords.concatenate([[Y],[Y]],axis=1)
Coords([[ [ 2., 2., 0.],
          [ 3., 3., 0.],
          [ 2., 2., 0.],
          [ 3., 3., 0.] ]])

```

classmethod fromstring (*s*, *sep*=' ', *ndim*=3, *count*=-1)

Create a *Coords* object with data from a string.

This uses `numpy.fromstring()` to read coordinates from a string and creates a *Coords* object from them.

Parameters

- **s** (*str*) – A string containing a single sequence of float numbers separated by whitespace and a possible separator string.
- **sep** (*str*) – The separator used between the coordinates. If not a space, all extra whitespace is ignored.
- **ndim** (*int*,) – Number of coordinates per point. Should be 1, 2 or 3 (default). If 1, resp. 2, the coordinate string only holds x, resp. x,y values.
- **count** (*int*, *optional*) – Total number of coordinates to read. This should be a multiple of *ndim*. The default is to read all the coordinates in the string.

Returns *Coords* – A *Coords* object with the coordinates read from the string.

Raises *ValueError* – If *count* was provided and the string does not contain that exact number of coordinates.

Notes

For writing the coordinates to a string, `numpy.tostring()` can be used.

Examples

```

>>> Coords.fromstring('4 0 0 3 1 2 6 5 7')
Coords([[ 4., 0., 0.],
        [ 3., 1., 2.],
        [ 6., 5., 7.]])
>>> Coords.fromstring('1 2 3 4 5 6',ndim=2)

```

(continues on next page)

(continued from previous page)

```
Coords([[ 1.,  2.,  0.],
        [ 3.,  4.,  0.],
        [ 5.,  6.,  0.]])
```

classmethod `fromfile` (*fil*, ***kargs*)

Read a *Coords* from file.

This uses `numpy.fromfile()` to read coordinates from a file and create a *Coords*. Coordinates X, Y and Z for subsequent points are read from the file. The total number of coordinates on the file should be a multiple of 3.

Parameters

- **fil** (*str* or *file*) – If *str*, it is a file name. An open file object can also be passed
- ****kargs** – Arguments to be passed to `numpy.fromfile()`.

Returns *Coords* – A *Coords* formed by reading all coordinates from the specified file.

Raises `ValueError` – If the number of coordinates read is not a multiple of 3.

See also:

`numpy.fromfile()` read an array to file

`numpy.tofile()` write an array to file

interpolate (*X*, *div*)

Create linear interpolations between two *Coords*.

A linear interpolation of two equally shaped *Coords* X and Y at parameter value t is a *Coords* with the same shape as X and Y and with coordinates given by $X * (1.0-t) + Y * t$.

Parameters

- **X** (*Coords* object) – A *Coords* object with same shape as *self*.
- **div** (*seed*) – This parameter is sent through the `arraytools.smartSeed()` to generate a list of parameter values for which to compute the interpolation. Usually, they are in the range 0.0 (*self*) to 1.0 (X). Values outside the range can be used however and result in linear extrapolations.

Returns *Coords* – A *Coords* object with an extra (first) axis, containing the concatenation of the interpolations of *self* and X at all parameter values in *div*. Its shape is (n,) + *self*.shape, where n is the number of values in *div*.

Examples

```
>>> X = Coords([0])
>>> Y = Coords([1])
>>> X.interpolate(Y,4)
Coords([[ 0. ,  0. ,  0. ],
        [ 0.25,  0. ,  0. ],
        [ 0.5 ,  0. ,  0. ],
        [ 0.75,  0. ,  0. ],
        [ 1. ,  0. ,  0. ]])
>>> X.interpolate(Y,[-0.1, 0.5, 1.25])
Coords([[ -0.1 ,  0. ,  0. ],
```

(continues on next page)

(continued from previous page)

```

    [ 0.5 , 0. , 0. ],
    [ 1.25, 0. , 0. ]])
>>> X.interpolate(Y, (4,0.3,0.2))
Coords([[ 0. , 0. , 0. ],
        [ 0.21, 0. , 0. ],
        [ 0.47, 0. , 0. ],
        [ 0.75, 0. , 0. ],
        [ 1. , 0. , 0. ]])

```

convexHull (*dir=None, return_mesh=False*)

Return the 2D or 3D convex hull of a *Coords*.

Parameters

- **dir** (*int (0,1,2), optional*) – If provided, it is one if the global axes and the 2D convex hull in the specified viewing direction will be computed. The default is to compute the 3D convex hull.
- **return_mesh** (*bool, optional*) – If True, returns the convex hull as a *Mesh* object instead of a *Connectivity*.

Returns

Connectivity or *Mesh* – The default is to return a *Connectivity* table containing the indices of the points that constitute the convex hull of the *Coords*. For a 3D hull, the *Connectivity* has plexitude 3, and eltype ‘tri3’; for a 2D hull these are respectively 2 and ‘line2’. The values in the *Connectivity* refer to the flat points list as obtained from *points()*.

If *return_mesh* is True, a compacted *Mesh* is returned instead of the *Connectivity*. For a 3D hull, the *Mesh* will be a *TriSurface*, otherwise it is a *Mesh* of ‘line2’ elements.

The returned *Connectivity* or *Mesh* will be empty if all the points are in a plane for the 3D version, or an a line in the viewing direction for the 2D version.

Notes

This uses SciPy to compute the convex hull. You need to have SciPy version 0.12.0 or higher.

See also example *ConvexHull*.

rot (*angle, axis=2, around=None, angle_spec=0.017453292519943295*)

Return a copy rotated over angle around axis.

Parameters

- **angle** (float or float *array_like (3,3)*) – If a float, it is the rotation angle, by default in degrees, and the parameters (*angle, axis, angle_spec*) are passed to *rotationMatrix()* to produce a (3,3) rotation matrix. Alternatively, the rotation matrix may be directly provided in the *angle* parameter. The *axis* and *angle_spec* are then ignored.
- **axis** (int (0,1,2) or float *array_like (3,)*) – Only used if *angle* is a float. If provided, it specifies the direction of the rotation axis: either one of 0,1,2 for a global axis, or a vector with 3 components for a general direction. The default (axis 2) is convenient for working with 2D-structures in the x-y plane.
- **around** (float *array_like (3,)*) – If provided, it species a point on the rotation axis. If not, the rotation axis goes through the origin of the global axes.

- **angle_spec** (*float*, *DEG* or *RAD*, *optional*) – Only used if *angle* is a float. The default (DEG) interpretes the angle in degrees. Use RAD to specify the angle in radians.

Returns *Coords* – The Coords rotated as specified by the parameters.

Note: `rot()` is a convenient shorthand for `rotate()`.

See also:

`translate()` translate a Coords

`affine()` rotate and translate a Coords

`arraytools.rotationMatrix()` create a rotation matrix for use in `rotate()`

Examples

```
>>> X = Coords('0123')
>>> print(X.rotate(30))
[[ 0.    0.    0. ]
 [ 0.87  0.5   0. ]
 [ 0.37  1.37  0. ]
 [-0.5   0.87  0. ]]
>>> print(X.rotate(30,axis=0))
[[ 0.    0.    0. ]
 [ 1.    0.    0. ]
 [ 1.    0.87  0.5 ]
 [ 0.    0.87  0.5 ]]
>>> print(X.rotate(30,axis=0,around=[0.,0.5,0.]))
[[ 0.    0.07 -0.25]
 [ 1.    0.07 -0.25]
 [ 1.    0.93  0.25]
 [ 0.    0.93  0.25]]
>>> m = rotationMatrix(30,axis=0)
>>> print(X.rotate(m))
[[ 0.    0.    0. ]
 [ 1.    0.    0. ]
 [ 1.    0.87  0.5 ]
 [ 0.    0.87  0.5 ]]
```

trl (*dir*, *step=1.0*, *inplace=False*)

Return a translated copy of the *Coords* object.

Translate the Coords in the direction *dir* over a distance *step* * *length(dir)*.

Parameters

- **dir** (int (0,1,2) or float *array_like* (... ,3)) – The translation vector. If an int, it specifies a global axis and the translation is in the direction of that axis. If an *array_like*, it specifies one or more translation vectors. If more than one, the array should be broadcastable to the Coords shape: this allows to translate different parts of the Coords over different vectors, all in one operation.
- **step** (*float*) – If *dir* is an int, this is the length of the translation. Else, it is a multiplying factor applied to *dir* prior to applying the translation.

Returns *Coords* – The Coords translated over the specified vector(s).

Note: `trl()` is a convenient shorthand for `translate()`.

See also:

`centered()` translate to center around origin

`Coords.align()` translate to align bounding box

Examples

```
>>> x = Coords([1.,1.,1.])
>>> print(x.translate(1))
[ 1. 2. 1.]
>>> print(x.translate(1,1.))
[ 1. 2. 1.]
>>> print(x.translate([0,1,0]))
[ 1. 2. 1.]
>>> print(x.translate([0,2,0],0.5))
[ 1. 2. 1.]
>>> x = Coords(arange(4).reshape(2,2,1))
>>> x
Coords([[[ 0.,  0.,  0.],
         [ 1.,  0.,  0.]],
<BLANKLINE>
        [[ 2.,  0.,  0.],
         [ 3.,  0.,  0.]])
>>> x.translate([[10.,-5.,0.],[20.,4.,0.]]) # translate with broadcasting
Coords([[[ 10., -5.,  0.],
         [ 21.,  4.,  0.]],
<BLANKLINE>
        [[ 12., -5.,  0.],
         [ 23.,  4.,  0.]])
```

rep (*n*, *dir*=0, *step*=1.0)

Replicate a Coords *n* times with a fixed translation step.

Parameters

- **n** (*int*) – Number of times to replicate the Coords.
- **dir** (*int* (0,1,2) or float *array_like* (3,)) – The translation vector. If an *int*, it specifies a global axis and the translation is in the direction of that axis.
- **step** (*float*) – If *dir* is an *int*, this is the length of the translation. Else, it is a multiplying factor applied to the translation vector.

Returns *Coords* – A Coords with an extra first axis with length *n*. The new shape thus becomes (*n*,) + *self.shape*. The first component along the axis 0 is identical to the original Coords. Each following component is equal to the previous translated over (*dir*,*step*), where *dir* and *step* are interpreted just like in the `translate()` method.

Notes

`rep()` is a convenient shorthand for `replicate()`.

Examples

```
>>> Coords([0.,0.,0.]).replicate(4,1,1.2)
Coords([[ 0. ,  0. ,  0. ],
        [ 0. ,  1.2,  0. ],
        [ 0. ,  2.4,  0. ],
        [ 0. ,  3.6,  0. ]])
>>> Coords([0.]).replicate(3,0).replicate(2,1)
Coords([[ [ 0.,  0.,  0.],
          [ 1.,  0.,  0.],
          [ 2.,  0.,  0.]],
        <BLANKLINE>
        [[ 0.,  1.,  0.],
          [ 1.,  1.,  0.],
          [ 2.,  1.,  0.]])
```

Functions defined in module coords

`coords.otherAxes` (*i*)

Return all global axes except the specified one

Parameters *i* (*int* (0, 1, 2)) – One of the global axes.

Returns *tuple of ints* – Two ints (j,k) identifying the other global axes in such order that (i,j,k) is a right-handed coordinate system.

`coords.bbox` (*objects*)

Compute the bounding box of a list of objects.

The bounding box of an object is the smallest rectangular cuboid in the global Cartesian coordinates, such that no points of the objects lie outside that cuboid. The resulting bounding box of the list of objects is the smallest bounding box that encloses all the objects in the list.

Parameters *objects* (*object or list of objects*) – One or more (list or tuple) objects that have a method `bbox()` returning the object's bounding box as a `Coords` with two points.

Returns *Coords* – A `Coords` object with two points: the first contains the minimal coordinate values, the second has the maximal ones of the overall bounding box encompassing all objects.

Notes

Objects that do not have a `bbox()` method or whose `bbox()` method returns invalid values, are silently ignored.

See also:

`Coords.bbox()` compute the bounding box of a `Coords` object.

Examples

```
>>> bbox((Coords([-1., 1., 0.]), Coords([2, -3])))
Coords([[ -1., -3.,  0.],
        [ 2.,  1.,  0.]])
```

`coords.bboxIntersection` (*A*, *B*)

Compute the intersection of the bounding box of two objects.

Parameters

- **A** (*first object*) – An object having a `bbox` method returning its boundary box.
- **B** (*second object*) – Another object having a `bbox` method returning its boundary box.

Returns *Coords* (2,3) – A *Coords* specifying the intersection of the bounding boxes of the two objects. This again has the format of a bounding box: a *coords* with two points: one with the minimal and one with the maximal coordinates. If the two bounding boxes do not intersect, an empty *Coords* is returned.

Notes

Since bounding boxes are *Coords* objects, it is possible to pass computed bounding boxes as arguments. The bounding boxes are indeed their own bounding box.

Examples

```
>>> A = Coords([[ -1., 1.], [2, -3]])
>>> B = Coords([[ 0., 1.], [4, 2]])
>>> C = Coords([[ 0., 2.], [4, 2]])
>>> bbox(A, B)
Coords([[ -1., -3.,  0.],
        [ 4.,  2.,  0.]])
```

The intersection of the bounding boxes of A and B degenerates into a line segment parallel to the x-axis:

```
>>> bboxIntersection(A, B)
Coords([[ 0.,  1.,  0.],
        [ 2.,  1.,  0.]])
```

The bounding boxes of A and C do not intersect:

```
>>> bboxIntersection(A, C)
Coords([], shape=(0, 3))
```

`coords.origin()`

Return Create a *Coords* holding the origin of the global coordinate system.

Returns

- *Coords* (3,) – A *Coords* holding a single point with coordinates (0.,0.,0.).
- *Exmaples*
- ———
- `>>> origin()`
- *Coords* ([0., 0., 0.]

`coords.pattern(s, aslist=False)`

Generate a sequence of points on a regular grid.

This function creates a sequence of points that are on a regular grid with unit step. These points are created from a simple string input, interpreting each character as a code specifying how to move from the last to the next point. The start position is always the origin (0.,0.,0.).

Currently the following codes are defined:

- 0 or +: goto origin (0.,0.,0.)
- 1..8: move in the x,y plane
- 9 or .: remain at the same place (i.e. duplicate the last point)
- A..I: same as 1..9 plus step +1. in z-direction
- a..i: same as 1..9 plus step -1. in z-direction
- /: do not insert the next point

Any other character raises an error.

When looking at the x,y-plane with the x-axis to the right and the y-axis up, we have the following basic moves: 1 = East, 2 = North, 3 = West, 4 = South, 5 = NE, 6 = NW, 7 = SW, 8 = SE.

Adding 16 to the ordinal of the character causes an extra move of +1. in the z-direction. Adding 48 causes an extra move of -1. This means that 'ABCDEFGHGI', resp. 'abcdefghi', correspond with '123456789' with an extra z +/- 1. This gives the following schema:

	z += 1		z unchanged		z -= 1			
F	B	E	6	2	5	f	b	e
C-----I-----A			3-----9-----1			c-----i-----a		
G	D	H	7	4	8	g	d	h

The special character '/' can be put before any character to make the move without inserting the new point. The string should start with a '0' or '9' to include the starting point (the origin) in the output.

Parameters

- **s** (*str*) – A string with characters generating subsequent points.
- **aslist** (*bool*, *optional*) – If True, the points are returned as lists of **integer** coordinates instead of a *Coords* object.

Returns *Coords* or *list of ints* – The default is to return the generated points as a *Coords*. With `aslist=True` however, the points are returned as a list of tuples holding 3 integer grid coordinates.

See also:

`xpattern()`

Examples

```
>>> pattern('0123')
Coords([[ 0.,  0.,  0.],
        [ 1.,  0.,  0.],
        [ 1.,  1.,  0.],
        [ 0.,  1.,  0.]])
>>> pattern('2'*4)
Coords([[ 0.,  1.,  0.],
        [ 0.,  2.,  0.],
        [ 0.,  3.,  0.],
        [ 0.,  4.,  0.]])
```

`coords.xpattern` (*s*, *nplex*=1)

Create a Coords object from a string pattern.

Create a sequence of points using `pattern()`, and groups the points by `nplex` to create a Coords with shape `(-1, nplex, 3)`.

Parameters

- **s** (*str*) – The string to pass to `pattern()` to produce the sequence of points.
- **nplex** (*int*) – The number of subsequent points to group together to create the structured Coords.

Returns *Coords* – A Coords with shape `(-1,nplex,3)`.

Raises `ValueError` – If the number of points produced by the input string *s* is not a multiple of *nplex*.

Examples

```
>>> print(xpattern('.12.34',3))
[[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 1.  1.  0.]]
<BLANKLINE>
[[ 1.  1.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  0.]]]
```

`coords.align` (*L*, *align*, *offset*=(0.0, 0.0, 0.0))

Align a list of geometrical objects.

Parameters

- **L** (*list of Coords or Geometry objects*) – A list of objects that have an appropriate `align` method, like the *Coords* and *Geometry* (and its subclasses).
- **align** (*str*) – A string of three characters, one for each coordinate direction, that define how the subsequent objects have to be aligned in each of the global axis directions:
 - ‘-’ : align on the minimal coordinate value
 - ‘+’ : align on the maximal coordinate value
 - ‘0’ : align on the middle coordinate value
 - ‘|’ [align the minimum value on the maximal value of the] previous itemThus the string ‘| - -’ will juxtapose the objects in the x-direction, while aligning them on their minimal coordinates in the y- and z- direction.
- **offset** (float *array_like* (3,), optional) – An extra translation to be given to each subsequent object. This can be used to create a space between the objects, instead of juxtaposing them.

Returns *list of objects* – A list with the aligned objects.

Notes

See also example `Align`.

See also:

`Coords.align()` align a single object with respect to a point.

6.1.2 formex — Formex algebra in Python

This module defines the `Formex` class, which is one of the two major classes for representing geometry in pyFormex (the other one being `Mesh`). The `Formex` class represents geometry as a simple 3-dim `Coords` array. This allows an implementation of most functionality of Formex algebra with a consistent and easy to use syntax.

Classes defined in module formex

class `formex.Formex` (*data=[]*, *prop=None*, *eltype=None*)

A structured collection of points in 3D space.

A `Formex` is a collection of points in a 3D cartesian space. The collection is structured into a set of elements all having the same number of points (e.g. a collection triangles all having three points).

As the `Formex` class is derived from `Geometry`, a `Formex` object has a `coords` attribute which is a `Coords` object. In a `Formex` this is always an array with 3 axes (numbered 0,1,2). Each scalar element of this array represents a coordinate. A row along the last axis (2) is a set of 3 coordinates and represents a point (aka. node, vertex).

For simplicity's sake, the current implementation only deals with points in a 3-dimensional space. This means that the length of axis 2 always equals 3. The user can create `Formices` (plural of `Formex`) in a 2-D space, but internally these will be stored with 3 coordinates, by adding a third value 0. All operations work with 3-D coordinate sets. However, it is easy to extract only a limited set of coordinates from the results, permitting to return to a 2-D environment

A plane of the array along the axes 2 and 1 is a set of points: we call this an element. This can be thought of as a geometrical shape (2 points form a line segment, 3 points make a triangle, ...) or as an element in Finite Element terms. But it really is up to the user as to how this set of points is to be interpreted. He can set an element type on the `Formex` to make this clear (see below).

The whole `Formex` then represents a collection of such elements. The `Formex` concept and layout is made more clear in *Formex data model* in the *pyFormex tutorial*.

Additionally, a `Formex` may have a property set, which is an 1-D array of integers. The length of the array is equal to the length of axis 0 of the `Formex` data (i.e. the number of elements in the `Formex`). Thus, a single integer value may be attributed to each element. It is up to the user to define the use of this integer (e.g. it could be an index in a table of element property records). If a property set is defined, it will be copied together with the `Formex` data whenever copies of the `Formex` (or parts thereof) are made. Properties can be specified at creation time, and they can be set, modified or deleted at any time. Of course, the properties that are copied in an operation are those that exist at the time of performing the operation.

Finally, a `Formex` object can have an element type, because plexitude alone does not uniquely define what the geometric entities are, and how they should be rendered. By default, pyFormex will render plex-1 as points, plex-2 as line segments, plex-3 as triangles and any higher plexitude as polygons. But the user could e.g. set `eltype = 'tet4'` on a plex-4 `Formex`, and then that would be rendered as tetraeders.

Parameters

- **data** (`Formex`, `Coords`, *array_like* or string) – Data to initialize the coordinates attribute `coords` in the `Formex`. See more details below.
- **prop** (int *array_like*, optional) – 1-dim int array with non-negative element property numbers. If provided, `setProp()` will be called to assign the specified properties.

- **eltype** (str | *ElementType*, optional) – The element type of the geometric entities (elements). If provided, it should be an *ElementType* instance or the name of such an instance. If not provided, the pyFormex default is used when needed and is based on the plexitude: 1 = point, 2 = line segment, 3 = triangle, 4 or more is a polygon.

The Formex coordinate data can be initialized by another *Formex*, by a *Coords*, by a 1D, 2D or 3D *array_like*, or by a string to be used in one of the pattern functions to create a coordinate list. If 2D coordinates are given, a 3-rd coordinate 0.0 is added. Internally, Formices always work with 3D coordinates. Thus:

```
F = Formex([[ [1,0], [0,1] ], [ [0,1], [1,2] ]])
```

creates a Formex with two elements, each having 2 points in the global z-plane. The innermost level of brackets group the coordinates of a point, the next level groups the points in an element, and the outermost brackets group all the elements of the Formex. Because the coordinates are stored in an array with 3 axes, all the elements in a Formex must contain the same number of points. This number is called the plexitude of the Formex.

A Formex may be initialized with a string instead of the numerical coordinate data. The string has the format *#:data* where *#* is a leader specifying the plexitude of the elements to be created. The *data* part of the string is passed to the *pattern()* function to generate a list of points on a regular grid of unit distances. Then the generated points are grouped in elements. If *#* is a number it just specifies the plexitude:

```
F = Formex('3:012034')
```

This creates six points, grouped by 3, thus leading to two elements (triangles). The leader can also be the character *l*. In that case each generated point is turned into a 2-point (line) element, by connecting it to the previous point. The following are two equivalent definitions of (the circumference of) a triangle:

```
F = Formex('2:010207')
G = Formex('l:127')
```

Note: The legacy variant of initializing a Formex with a string without the leading '#' is no longer accepted.

Because the *Formex* class is derived from *Geometry*, it has the following attributes:

- *coords*,
- *prop*,
- *attrib*,
- *fields*.

Furthermore it has the following properties and methods that are applied on the *coords* attribute.

- *xyz*,
- *x*,
- *y*,
- *z*,
- *xy*,
- *yz*,
- *xz*,
- *points()*,
- *bbox()*,

- `center()`,
- `bboxPoint()`,
- `centroid()`,
- `sizes()`,
- `dsize()`,
- `bsphere()`,
- `bboxes()`,
- `inertia()`,
- `principalCS()`,
- `principalSizes()`,
- `distanceFromPlane()`,
- `distanceFromLine()`,
- `distanceFromPoint()`,
- `directionalSize()`,
- `directionalWidth()`,
- `directionalExtremes()`.

Also, the following Coords transformation methods can be directly applied to a *Formex* object. The return value is a new Formex identical to the original, except for the coordinates, which are transformed by the specified method. Refer to the corresponding *Coords* method for the usage of these methods:

- `scale()`,
- `adjust()`,
- `translate()`,
- `centered()`,
- `align()`,
- `rotate()`,
- `shear()`,
- `reflect()`,
- `affine()`,
- `toCS()`,
- `fromCS()`,
- `transformCS()`,
- `position()`,
- `cylindrical()`,
- `hyperCylindrical()`,
- `toCylindrical()`,
- `spherical()`,
- `superSpherical()`,

- `toSpherical()`,
- `bump()`,
- `flare()`,
- `map()`,
- `map1()`,
- `mapd()`,
- `copyAxes()`,
- `swapAxes()`,
- `rollAxes()`,
- `projectOnPlane()`,
- `projectOnSphere()`,
- `projectOnCylinder()`,
- `isopar()`,
- `addNoise()`,
- `rot()`,
- `trl()`.

Examples

```
>>> print(Formex([[0,1],[2,3]]))
{[0.0,1.0,0.0], [2.0,3.0,0.0]}
>>> print(Formex('1:0123'))
{[0.0,0.0,0.0], [1.0,0.0,0.0], [1.0,1.0,0.0], [0.0,1.0,0.0]}
>>> print(Formex('4:0123'))
{[0.0,0.0,0.0; 1.0,0.0,0.0; 1.0,1.0,0.0; 0.0,1.0,0.0]}
>>> print(Formex('2:0123'))
{[0.0,0.0,0.0; 1.0,0.0,0.0], [1.0,1.0,0.0; 0.0,1.0,0.0]}
>>> F = Formex('1:1234')
>>> print(F)
{[0.0,0.0,0.0; 1.0,0.0,0.0], [1.0,0.0,0.0; 1.0,1.0,0.0], [1.0,1.0,0.0; 0.0,1.0,0.
↪0], [0.0,1.0,0.0; 0.0,0.0,0.0]}
>>> print(F.info())
shape      = (4, 2, 3)
bbox[lo]   = [ 0.  0.  0.]
bbox[hi]   = [ 1.  1.  0.]
center     = [ 0.5  0.5  0. ]
maxprop    = -1
<BLANKLINE>
>>> F.nelems()
4
>>> F.level()
1
>>> F.x
array([[ 0.,  1.],
       [ 1.,  1.],
       [ 1.,  0.],
       [ 0.,  0.]])
```

(continues on next page)

(continued from previous page)

```
>>> F.center()
Coords([ 0.5,  0.5,  0. ])
>>> F.bboxPoint('+++')
Coords([ 1.,  1.,  0.]
```

The Formex class defines the following attributes above the ones inherited from Geometry:

eltype

Type None or *ElementType*

shape

Return the shape of the Formex.

The shape of a Formex is the shape of its coords array.

Returns *tuple of ints* – A tuple (nelems, nplex, ndim).

Examples

```
>>> Formex('1:1234').shape
(4, 2, 3)
>>> Formex('1:1234').shape
(4, 1, 3)
```

nelems()

Return the number of elements of the *Formex*.

The number of elements is the length of the first axis of the `coords` array.

Returns *int* – The number of elements in the Formex

Examples

```
>>> Formex('1:1234').nelems()
4
```

nplex()

Return the plexitude of the *Formex*.

The plexitude is the number of points per element. This is the length of the second axis of the `coords` array.

Examples:

1. unconnected points,
2. straight line elements,
3. triangles or quadratic line elements,
4. tetraeders or quadrilaterals or cubic line elements.

Returns *int* – The plexitude of the elements in the Formex

Examples

```
>>> Formex('1:1234').nplex()
2
```

`ndim()`

Return the number of dimensions.

This is the number of coordinates for each point. In the current implementation this is always 3, though you can define 2D Formices by given only two coordinates: the third will automatically be set to zero.

Returns *int* – The number of coordinates per point: currently, this is always 3.

Examples

```
>>> Formex('1:1234').ndim()
3
```

`npoints()`

Return the number of points in the Formex.

This is the product of the number of elements in the Formex with the plexitude of the elements.

Returns *int* – The total number of points in the Formex

Notes

`ncoords` is an alias for `npoints`

Examples

```
>>> Formex('1:1234').npoints()
8
```

`ncoords()`

Return the number of points in the Formex.

This is the product of the number of elements in the Formex with the plexitude of the elements.

Returns *int* – The total number of points in the Formex

Notes

`ncoords` is an alias for `npoints`

Examples

```
>>> Formex('1:1234').npoints()
8
```

`elType()`

Return the element type of the Formex.

Returns *ElementType* or None – If an element type was defined for the Formex, returns the corresponding *ElementType*; else returns None.

See also:

e1Name() Return the name of the *ElementType*

Examples

```
>>> Formex('l:1234').elType()
>>> Formex('l:1234',eltype='line2').elType()
Line2
```

e1Name()

Return the element name of the Formex.

Returns *str* or *None* – If an element type was defined for the Formex, returns the name of the *ElementType* ; else returns None.

See also:

e1Type() Return the *ElementType*

Examples

```
>>> Formex('l:1234').e1Name()
>>> Formex('l:1234',eltype='line2').e1Name()
'line2'
```

level()

Return the level (dimensionality) of the Formex.

The level or dimensionality of a geometrical object is the minimum number of parametric directions required to describe the object. Thus we have the following values:

0: points 1: lines 2: surfaces 3: volumes

Because the geometrical meaning of a Formex is not always defined, the level may be unknown. In that case, -1 is returned.

If the Formex has an ‘eltype’ set, the value is determined from the Element database. Else, the value is equal to the plexitude minus one for plexitudes up to 3, and equal to 2 for any higher plexitude (since the default is to interpret a higher plexitude as a polygon).

Returns *int* – An int 0..3 giving the number of parametric dimensions of the geometric entities in the Formex.

Examples

```
>>> Formex('1:123').level()
0
>>> Formex('l:123').level()
1
>>> Formex('3:123').level()
2
```

(continues on next page)

(continued from previous page)

```
>>> Formex('3:123', eltype='line3').level()
1
```

view()

Return the Formex coordinates as a numpy array (ndarray).

Returns a view to the Coords array as an ndarray. The use of this method is deprecated: use the `xyz` property instead.

element(i)

Return element *i* of the Formex.

Parameters *i* (*int*) – The index of the element to return.

Returns *Coords object* – A Coords with shape (self.nplex(), 3)

Examples

```
>>> Formex('1:12').element(0)
Coords([[ 0.,  0.,  0.],
        [ 1.,  0.,  0.]])
>>> Formex('1:12').select(0)
Formex([[[ 0.,  0.,  0.],
         [ 1.,  0.,  0.]])
```

point(i,j)

Return point *j* of element *i*.

Parameters

- *i* (*int*) – The index of the element from which to return a point.
- *j* (*int*) – The index in element *i* of the point to be returned.

Returns *Coords object* – A Coords with shape (3,), being point *j* of element *i*.

Examples

```
>>> Formex('1:12').point(0,1)
Coords([ 1.,  0.,  0.]
```

coord(i,j,k)

Return coordinate *k* of point *j* of element *i*.

Parameters

- *i* (*int*) – The index of the element from which to return a point.
- *j* (*int*) – The index in element *i* of the point for which to return a coordinate.
- *k* (*int*) – The index in point (*i,j*) of the coordinate to be returned.

Returns *float* – The value of coordinate *k* of point *j* of element *i*.

Examples

```
>>> Formex('1:12').coord(0,1,0)
1.0
```

centroids()

Return the centroids of all elements of the Formex.

The centroid of an element is the point whose coordinates are the average values of all points of the element.

Returns *Coords* – A Coords object with shape (*nelems()*, 3), holding the centroids of all the elements in the Formex.

Examples

```
>>> Formex('1:123').centroids()
Coords([[ 0.5,  0. ,  0. ],
        [ 1. ,  0.5,  0. ],
        [ 0.5,  1. ,  0. ]])
```

toMesh (**kargs)

Convert a Formex to a Mesh.

Converts a geometry in Formex model to the equivalent Mesh model. In the Mesh model, all points with nearly identical coordinates are fused into a single point (using *fuse()*), and elements are defined by a connectivity table with integers pointing to the corresponding vertex.

Parameters *kargs* – Keyword parameters to be passed to *fuse()*.

Returns *Mesh* – A Mesh representing the same geometrical model as the input Formex. Property numbers *prop* and element type *eltype* are also set to the same values as in the Formex.

Examples

```
>>> F = Formex('1:12')
>>> F
Formex([[[[ 0.,  0.,  0.],
          [ 1.,  0.,  0.]],
<BLANKLINE>
        [[ 1.,  0.,  0.],
          [ 1.,  1.,  0.]]])
>>> M = F.toMesh()
>>> print(M)
Mesh: nnodes: 3, nelems: 2, nplex: 2, level: 1, eltype: line2
BBox: [ 0.  0.  0.], [ 1.  1.  0.]
Size: [ 1.  1.  0.]
Length: 2.0
```

toSurface()

Convert a Formex to a Surface.

Tries to convert the Formex to a TriSurface. First the Formex is converted to a Mesh, and then the resulting Mesh is converted to a TriSurface.

Returns *TriSurface* – A TriSurface if the conversion is succesful, else an error is raised.

Notes

The conversion will only work if the Formex represents a surface and its elements are triangles or quadrilaterals. If the plexitude of the Formex is 3, the element type is 'tri3' or None, the returned TriSurface is equivalent with the Formex. If the Formex contains higher order triangles or quadrilaterals, The new geometry will be an approximation of the input. Any other input geometry will fail to convert.

Examples

```
>>> F = Formex('3:.12.34')
>>> F
Formex([[ [ 0.,  0.,  0.],
          [ 1.,  0.,  0.],
          [ 1.,  1.,  0.] ],
        <BLANKLINE>
        [[ [ 1.,  1.,  0.],
          [ 0.,  1.,  0.],
          [ 0.,  0.,  0.] ]])
>>> print(F.toSurface())
Mesh: nnodes: 4, nelems: 2, nplex: 3, level: 2, eltype: tri3
BBBox: [ 0.  0.  0.], [ 1.  1.  0.]
Size: [ 1.  1.  0.]
Area: 1.0
```

info()

Return information about a Formex.

Returns

- A multiline string with some basic information about the Formex
- its shape, bounding box, center and maxprop.

Examples

```
>>> print(Formex('3:.12.34').info())
shape      = (2, 3, 3)
bbox[lo]   = [ 0.  0.  0.]
bbox[hi]   = [ 1.  1.  0.]
center     = [ 0.5  0.5  0. ]
maxprop    = -1
<BLANKLINE>
```

classmethod point2str (point)

Return a string representation of a point

Parameters **elem** (float *array_like* (3,)) – The coordinates of the point to return as a string.

Returns *str* – A string with the representation of a single point.

Examples

```
>>> Formex.point2str([1.,2.,3.])
'1.0,2.0,3.0'
```

classmethod `element2str (elem)`

Return a string representation of an element

Parameters `elem` (float *array_like* (nplex,3)) – The element to return as a string.

Returns `str` – A string with the representation of a single element.

Examples

```
>>> Formex.element2str([[1.,2.,3.],[4.,5.,6.]])
'[1.0,2.0,3.0; 4.0,5.0,6.0]'
```

asFormex ()

Return string representation of all the coordinates in a Formex.

Returns `str` – A single string with all the coordinates of the Formex. Coordinates are separated by commas, points are separated by semicolons and grouped between brackets, elements are separated by commas and grouped between braces.

Examples

```
>>> F = Formex([[1,0],[0,1]],[[0,1],[1,2]])
>>> F.asFormex()
'{{1.0,0.0,0.0; 0.0,1.0,0.0}, [0.0,1.0,0.0; 1.0,2.0,0.0]}'
```

asFormexWithProp ()

Return string representation as Formex with properties.

Returns `str` – The string representation as done by `asFormex()`, followed by the words “with prop” and a list of the properties.

Examples

```
>>> F = Formex([[1,0],[0,1]],[[0,1],[1,2]]).setProp([1,2])
>>> F.asFormexWithProp()
'{{1.0,0.0,0.0; 0.0,1.0,0.0}, [0.0,1.0,0.0; 1.0,2.0,0.0]} with prop [1 2]'
```

asCoords ()

Return string representation as a Coords.

Returns `str` – A multiline string with the coordinates of the Formex as formatted by the meth:`coords.Coords.__repr__` method.

Examples

```
>>> F = Formex([[1,0],[0,1]],[[0,1],[1,2]])
>>> print(F.asCoords())
Formex([[ [ 1.,  0.,  0.],
         [ 0.,  1.,  0.]],
<BLANKLINE>
        [[ 0.,  1.,  0.],
         [ 1.,  2.,  0.]])
```

asArray()

Return string representation as a numpy array.

Returns *str* – A multiline string with the coordinates of the Formex as formatted by the meth:*coords.Coords.__str__* method.

Examples

```
>>> F = Formex([[ [1,0], [0,1] ], [ [0,1], [1,2] ]])
>>> print(F.asArray())
[[ [ 1.  0.  0.]
  [ 0.  1.  0.]]
<BLANKLINE>
[[ [ 0.  1.  0.]
  [ 1.  2.  0.]]]
```

classmethod setPrintFunction(*func*)

Choose the default formatting for printing formices.

This sets how formices will be formatted by a print statement. Currently there are two available functions: *asFormex*, *asArray*. The user may create his own formatting method. This is a class method. It should be used as follows: *Formex.setPrintFunction(Formex.asArray)*.

classmethod concatenate(*Flist*)

Concatenate a list of Formices.

All the Formices in the list should have the same plexitude, If any of the Formices has property numbers, the resulting Formex will inherit the properties. In that case, any Formices without properties will be assigned property 0. If all Formices are without properties, so will be the result. The eltype of the resulting Formex will be that of the first Formex in the list.

Parameters *Flist* (*list of Formex objects*) – A list of Formices all having the same plexitude.

Returns *Formex* – The concatenation of all the Formices in the list. The number of elements in the Formex is the sum of the number of elements in all the Formices.

Note: This is a class method, not an instance method. It is commonly invoked as *Formex.concatenate*.

See also:

[__add__\(\)](#) implements concatenation as simple addition (F+G)

Examples

```
>>> F = Formex([1.,1.,1.]).setProp(1)
>>> G = Formex([2.,2.,2.])
>>> H = Formex([3.,3.,3.]).setProp(3)
>>> K = Formex.concatenate([F,G,H])
>>> print(K.asFormexWithProp())
{[1.0,1.0,1.0], [2.0,2.0,2.0], [3.0,3.0,3.0]} with prop [1 0 3]
```

__add__(*F*)

Concatenate two formices.

Parameters **F** (*Formex*) – A Formex with the same plexitude as self.

Returns *Formex* – The concatenation of the Formices self and F.

Note: This method implements the addition operation and allows to write simple expressions as $F+G$ to concatenate the Formices F and G. When concatenating many Formices, `concatenate()` is more efficient however, because all the Formices in the list are concatenated in one operation.

See also:

`concatenate()` concatenate a list of Formices

Examples

```
>>> F = Formex([1.,1.,1.]).setProp(1)
>>> G = Formex([2.,2.,2.])
>>> H = Formex([3.,3.,3.]).setProp(3)
>>> K = F+G+H
>>> print(K.asFormexWithProp())
{[1.0,1.0,1.0], [2.0,2.0,2.0], [3.0,3.0,3.0]} with prop [1 0 3]
```

split (*n=1*)

Split a Formex in Formices containing n elements.

Parameters **n** (*int*) – Number of elements per Formex

Returns *list of Formices* – A list of Formices all containing n elements, except for the last, which may contain less.

Examples

```
>>> Formex('1:111').split(2)
[Formex([[ [ 0., 0., 0.],
           [ 1., 0., 0.]],
<BLANKLINE>
           [[ 1., 0., 0.],
            [ 2., 0., 0.] ]]), Formex([[ [ 2., 0., 0.],
           [ 3., 0., 0.] ]])]
```

selectNodes (*idx*)

Extract a Formex holding only some points of the parent.

This creates subentities of all elements in the Formex. The returned Formex inherits the properties of the parent.

Parameters **idx** (*list of ints*) – Indices of the points to retain in the new Formex.

Notes

For example, if F is a plex-3 Formex representing triangles, the sides of the triangles are given by $F.selectNodes([0,1]) + F.selectNodes([1,2]) + F.selectNodes([2,0])$

See also:

`select()` Select elements from a Formex

Examples

```
>>> F = Formex('3:.12.34')
>>> print(F.selectNodes((0,1)))
{[0.0,0.0,0.0; 1.0,0.0,0.0], [1.0,1.0,0.0; 0.0,1.0,0.0]}
```

asPoints()

Reduce the Formex to a simple set of points.

This removes the element structure of the Formex.

Returns *Formex* – A Formex with plexitude 1 and number of elements (points) equal to `self.nelems() * self.nplex()`. The Formex shares the coordinate data with the parent. If the parent has properties, they are multiplexed so that each point has the property of its parent element. The eltype of the returned Formex is None.

See also:

`points()` returns the list of points as a Coords object

Examples

```
>>> F = Formex('3:.12.34',prop=[1,2]).asPoints()
>>> print(F.asFormexWithProp())
{[0.0,0.0,0.0], [1.0,0.0,0.0], [1.0,1.0,0.0], [1.0,1.0,0.0], [0.0,1.0,
↪0.0], [0.0,0.0,0.0]} with prop [1 1 1 2 2 2]
```

remove(*F*, *permutations*='roll', *rtol*=1e-05, *atol*=1e-05)

Remove elements that also occur in another Formex.

Parameters

- **F** (*Formex*) – Another Formex with the same plexitude as self.
- **permutations** (*bool*, *optional*) – If True, elements consisting of the
- **is also the subtraction of the current Formex with F. (This) –**
- **are only removed if they have the same nodes in the same (Elements) –**
- **order.** –

Examples

```
>>> F = Formex('1:111')
>>> G = Formex('1:1')
>>> print(F.remove(G))
{[1.0,0.0,0.0; 2.0,0.0,0.0], [2.0,0.0,0.0; 3.0,0.0,0.0]}
```

removeDuplicate(*permutations*='all', *rtol*=1e-05, *atol*=1e-08)

Return a Formex which holds only the unique elements.

Parameters

- **permutations** (*str*) – Defines which permutations of the element points are allowed while still considering the elements equal. Possible values are:

- ‘none’: no permutations are allowed: elements must have matching points at all locations. This is the default;
- ‘roll’: rolling is allowed. Elements that can be transformed into each other by rolling their points are considered equal;
- ‘all’: any permutation of the same points will be considered an equal element.
- **rtol** (*float*, *optional*) – Relative tolerance used when considering two points being equal.
- **atol** (*float*, *optional*) – Absolute tolerance used when considering two points being equal.

Notes

`rtol` and `atol` are passed to `coords.Coords.fuse()` to find equal points. `permutation` is passed to `arraytools.unique()` to remove the duplicates.

Examples

```
>>> F = Formex('1:111') + Formex('1:1')
>>> print(F.removeDuplicate())
{[0.0,0.0,0.0; 1.0,0.0,0.0], [1.0,0.0,0.0; 2.0,0.0,0.0]}, [2.0,0.0,0.0; 3.0,0.0,0.0]}
```

test (*nodes='all'*, *dir=0*, *min=None*, *max=None*, *atol=0.0*)

Flag elements having coordinates between `min` and `max`.

This is comparable with `coords.Coords.test()` but operates at the Formex element level. It tests the position of one or more points of the elements of the *Formex* with respect to one or two parallel planes. This is very useful in clipping a Formex in a specified direction. In most cases the clipping direction is one of the global coordinate axes, but a general direction may be used as well.

Testing along global axis directions is highly efficient. It tests whether the corresponding coordinate is above or equal to the `min` value and/or below or equal to the `max` value. Testing in a general direction tests whether the distance to the `min` plane is positive or zero and/or the distance to the `max` plane is negative or zero.

Parameters

- **nodes** (*int*, *list of ints or string*) – Specifies which points of the elements are taken into account in the tests. It should be one of the following:
 - a single point index (smaller than `self.nplex()`),
 - a list of point numbers (all smaller than `< self.nplex()`),
 - one of the special strings: ‘all’, ‘any’, ‘none’.

The default (‘all’) will flag the elements that have all their nodes between the planes `x=min` and `x=max`, i.e. the elements that fall completely between these planes.

- **dir** (a single *int* or a float *array_like* (3,)) – The direction in which to measure distances. If an *int*, it is one of the global axes (0,1,2). Else it is a vector with 3 components. The default direction is the global x-axis.
- **min** (*float or point-like*, *optional*) – Position of the minimal clipping plane. If *dir* is an *int*, this is a single float giving the coordinate along the specified global axis. If *dir* is a vector, this must be a point and the minimal clipping plane is defined by

this point and the normal vector *dir*. If not provided, there is no clipping at the minimal side.

- **max** (*float* or *point-like.*) – Position of the maximal clipping plane. If *dir* is an int, this is a single float giving the coordinate along the specified global axis. If *dir* is a vector, this must be a point and the maximal clipping plane is defined by this point and the normal vector *dir*. If not provided, there is no clipping at the maximal side.
- **atol** (*float*) – Tolerance value added to the tests to account for accuracy and rounding errors. A *min* test will be ok if the point's distance from the *min* clipping plane is $> -atol$ and/or the distance from the *max* clipping plane is $< atol$. Thus a positive *atol* widens the clipping planes.

Returns *1-dim bool array* – Array with length `self.nelems()` flagging the elements that pass the test(s). The return value can directly be used as an index in `select()` or `cselect` to obtain a *Formex* with the elements satisfying the test or not. Or you can use `where(result)[0]` to get the indices of the elements passing the test.

Raises *ValueError: At least one of min or max have to be specified* – If neither *min* nor *max* are provided.

See also:

select() return only the selected elements

cselect() return all but the selected elements

Examples

```
>>> F = Formex('1:1122')
>>> print(F)
[[0.0,0.0,0.0; 1.0,0.0,0.0], [1.0,0.0,0.0; 2.0,0.0,0.0], [2.0,0.0,0.0;
↪ 2.0,1.0,0.0], [2.0,1.0,0.0; 2.0,2.0,0.0]]
>>> F.test(min=0.0,max=1.0)
array([ True, False, False, False])
>>> F.test(nodes=[0],min=0.0,max=1.0)
array([ True,  True, False, False])
>>> F.test(dir=[1.,-1.,0.],min=[1.,1.,0.])
array([False,  True,  True, False])
>>> F.test(nodes='any',dir=[1.,-1.,0.],min=[1.,1.,0.])
array([ True,  True,  True,  True])
```

shrink (*factor*)

Scale all elements with respect to their own center.

Parameters **factor** (*float*) – Scaling factor for the elements. A value < 1.0 will shrink the elements, while a factor > 1.0 will enlarge them.

Returns *Formex* – A *Formex* where each element has been scaled with the specified factor in local axes with origin at the element's center.

Notes

This operation is called 'shrink' because it is commonly used with a factor smaller than 1 (often around 0.9) to draw an exploded view where touching elements are disconnected.

Examples

```
>>> Formex('1:12').shrink(0.8)
Formex([[ [ 0.1, 0. , 0. ],
          [ 0.9, 0. , 0. ]],
<BLANKLINE>
        [[ 1. , 0.1, 0. ],
         [ 1. , 0.9, 0. ]]])
```

circulize1()

Transforms the first octant of the 0-1 plane into 1/6 of a circle.

Points on the 0-axis keep their position. Lines parallel to the 1-axis are transformed into circular arcs. The bisector of the first quadrant is transformed in a straight line at an angle $\text{Pi}/6$. This function is especially suited to create circular domains where all bars have nearly same length. See the Diamatic example.

reverse()

Return a Formex where all elements have been reversed.

Reversing an element means reversing the order of its points.

Returns *Formex* – A Formex with same shape, where the points of all elements are in reverse order.

Notes

This is equivalent to `self.selectNodes(arange(self.nplex()-1,-1,-1))`.

Examples

```
>>> F = Formex('1:11')
>>> F.reverse()
Formex([[ [ 1., 0., 0.],
          [ 0., 0., 0.]],
<BLANKLINE>
        [[ 2., 0., 0.],
         [ 1., 0., 0.]])
```

mirror (*dir=0, pos=0.0, keep_orig=True*)

Add a reflection in one of the coordinate directions.

This method is like `reflect()`, but by default adds the reflected part to the original.

Parameters

- **dir** (*int* (0, 1, 2)) – Global axis direction of the reflection (default 0 or x-axis).
- **pos** (*float*) – Offset of the mirror plane from origin (default 0.0)
- **keep_orig** (*bool, optional*) – If True (default) the original plus the mirrored geometry is returned. Setting it to False will only return the mirror, and thus behaves just like `reflect()`.

Returns *Formex* – A Formex with the original and the mirrored elements, or only the mirrored elements if `keep_orig` is False.

Examples

```

>>> F = Formex('1:11')
>>> F.mirror()
Formex([[ [ 0., 0., 0.],
          [ 1., 0., 0.]],
<BLANKLINE>
        [[ 1., 0., 0.],
          [ 2., 0., 0.]],
<BLANKLINE>
        [[ 0., 0., 0.],
          [-1., 0., 0.]],
<BLANKLINE>
        [[-1., 0., 0.],
          [-2., 0., 0.]])
>>> F.mirror(keep_orig=False)
Formex([[ [ 0., 0., 0.],
          [-1., 0., 0.]],
<BLANKLINE>
        [[-1., 0., 0.],
          [-2., 0., 0.]])

```

translate(**args*)

Multiple subsequent translations in axis directions.

Parameters **args* (one or more tuples (*axis*, *step*)) – Each argument is a tuple (*axis*, *step*) which will do a translation over a length *step* in the direction of the global axis *axis*.

Returns *Formex* – The input *Formex* translated over the combined translation vector of the arguments.

Notes

This function is especially convenient to translate over computed steps.

See also:

translate() translate a *Formex*

Examples

```

>>> F = Formex('1:11')
>>> d = random.random(3)
>>> allclose(F.translate((0,d[0]), (2,d[2]), (1,d[1])).coords,
             ↪translate(d).coords)
True

```

replicate(*n*, *dir=0*, *step=1.0*)

Create copies at regular distances along a straight line.

Parameters

- **n** (*int*) – Number of copies to create
- **dir** (*int* (0,1,2) or float *array_like* (3,)) – The translation vector. If an *int*, it specifies a global axis and the translation is in the direction of that axis.

- **step** (*float*) – If `dir` is an int, this is the length of the translation. Else, it is a multiplying factor applied to the translation vector.

Returns *Formex* – A *Formex* with the concatenation of `n` copies of the original. Each copy is equal to the previous one translated over a distance `step * length(dir)` in the direction `dir`. The first of the copies is equal to the original.

See also:

[*repm\(\)*](#) replicate in multiple directions

[*replic2\(\)*](#) replicate in two directions with bias and taper

Examples

```
>>> Formex('1:1').replicate(4,1)
Formex([[ [ 0., 0., 0.],
          [ 1., 0., 0.]],
<BLANKLINE>
        [ [ 0., 1., 0.],
          [ 1., 1., 0.]],
<BLANKLINE>
        [ [ 0., 2., 0.],
          [ 1., 2., 0.]],
<BLANKLINE>
        [ [ 0., 3., 0.],
          [ 1., 3., 0.]])
```

rep (*n*, *dir=0*, *step=1.0*)

Create copies at regular distances along a straight line.

Parameters

- **n** (*int*) – Number of copies to create
- **dir** (int (0,1,2) or float *array_like* (3,)) – The translation vector. If an int, it specifies a global axis and the translation is in the direction of that axis.
- **step** (*float*) – If `dir` is an int, this is the length of the translation. Else, it is a multiplying factor applied to the translation vector.

Returns *Formex* – A *Formex* with the concatenation of `n` copies of the original. Each copy is equal to the previous one translated over a distance `step * length(dir)` in the direction `dir`. The first of the copies is equal to the original.

See also:

[*repm\(\)*](#) replicate in multiple directions

[*replic2\(\)*](#) replicate in two directions with bias and taper

Examples

```
>>> Formex('1:1').replicate(4,1)
Formex([[ [ 0., 0., 0.],
          [ 1., 0., 0.]],
<BLANKLINE>
```

(continues on next page)

(continued from previous page)

```

    [[ 0., 1., 0.],
     [ 1., 1., 0.]],
<BLANKLINE>
    [[ 0., 2., 0.],
     [ 1., 2., 0.]],
<BLANKLINE>
    [[ 0., 3., 0.],
     [ 1., 3., 0.]])

```

repm (*n*, *dir*=(0, 1, 2), *step*=(1.0, 1.0, 1.0))

Repeatedly replication in different directions

This repeatedly applies `replicate()` a number of times. The parameters are lists of values like those for `replicate`.

Parameters

- **n** (*list of int*) – Number of copies to create in the subsequent replications.
- **dir** (*list of int (0,1,2) or list of float array_like (3,)*) – Subsequent translation vectors. See `replicate()`.
- **step** (*list of floats*) – The step for the subsequent replications.

Returns *Formex* – A Formex with the concatenation of `prod(n)` copies of the original, translated as specified by the `dir` and `step` parameters. The first of the copies is equal to the original.

Note: If the parameter lists `n`, `dir`, `step` have different lengths, the operation is executed only for the shortest of the three.

See also:

`replicate()` replicate in a single direction

`replic2()` replicate in two directions with bias and taper

Examples

```

>>> Formex('1:1').repm((2,2),(1,2))
Formex([[[ 0., 0., 0.],
          [ 1., 0., 0.]],
<BLANKLINE>
        [[ 0., 1., 0.],
          [ 1., 1., 0.]],
<BLANKLINE>
        [[ 0., 0., 1.],
          [ 1., 0., 1.]],
<BLANKLINE>
        [[ 0., 1., 1.],
          [ 1., 1., 1.]])

```

```

>>> print(Formex([origin()]).repm((2,2)))
{[0.0,0.0,0.0], [1.0,0.0,0.0], [0.0,1.0,0.0], [1.0,1.0,0.0]}

```

replic (*n*, *step*=1.0, *dir*=0)

Return a Formex with `n` replications in direction `dir` with `step`.

Note: This works exactly like `replicate()` but has another order of the parameters. It is kept for historical reasons, but should not be used in new code.

replic2 (*n1*, *n2*, *t1=1.0*, *t2=1.0*, *d1=0*, *d2=1*, *bias=0*, *taper=0*)

Replicate in two directions with bias and taper.

Parameters

- **n1** (*int*) – Number of replications in first direction
- **n2** (*int*) – Number of replications in second direction
- **t1** (*float*) – Step length in the first direction
- **t2** (*float*) – Step length in the second direction
- **d1** (*int*) – Global axis of the first direction
- **d2** (*int*) – Global axis of the second direction
- **bias** (*float*) – Extra translation in direction d1 for each step in direction d2
- **taper** (*int*) – Extra number of copies generated in direction d1 for each step in direction d2

Note: If no bias nor taper is needed, the use of `repm()` is recommended.

See also:

`replicate()` replicate in a single direction

`repm()` replicate in multiple directions

Examples

```
>>> print(Formex([origin()]).replic2(2,2))
{[0.0,0.0,0.0], [1.0,0.0,0.0], [0.0,1.0,0.0], [1.0,1.0,0.0]}
>>> print(Formex([origin()]).replic2(2,2,bias=0.2))
{[0.0,0.0,0.0], [1.0,0.0,0.0], [0.2,1.0,0.0], [1.2,1.0,0.0]}
>>> print(Formex([origin()]).replic2(2,2,taper=-1))
{[0.0,0.0,0.0], [1.0,0.0,0.0], [0.0,1.0,0.0]}
```

replicm (*n*, *step=(1.0, 1.0, 1.0)*, *dir=(0, 1, 2)*)

Replicate in multiple global axis directions.

Note: This works exactly like `repm()` but has another order of the parameters. It is kept for historical reasons, but should not be used in new code.

rosette (*n*, *angle*, *axis=2*, *around=(0.0, 0.0, 0.0)*, *angle_spec=0.017453292519943295*, ***kargs*)

Create rotational replications of a Formex.

Parameters

- **n** (*int*) – Number of copies to create
- **angle** (*float*) – Angle between successive copies.

- **axis** (*int or (3,) float :term:*) – The rotation axis. If an int one of 0,1,2, specifying a global axis, or a vector with 3 components specifying an axis through the origin. The returned matrix is 3D.
- **around** (*float array_like (3,)*) – If provided, it species a point on the rotation axis. If not, the rotation axis goes through the origin of the global axes.
- **angle_spec** (*float, DEG or RAD, optional*) – The default (DEG) interpretes the angle in degrees. Use RAD to specify the angle in radians.

Returns *Formex* – A Formex with n rotational replications with given angular step. The original Formex is the first of the n replicas.

Examples

```
>>> Formex('l:l').rosette(4,90.)
Formex([[[ 0.,  0.,  0.],
          [ 1.,  0.,  0.]],
        <BLANKLINE>
        [[ 0.,  0.,  0.],
          [ 0.,  1.,  0.]],
        <BLANKLINE>
        [[ 0.,  0.,  0.],
          [-1.,  0.,  0.]],
        <BLANKLINE>
        [[ 0.,  0.,  0.],
          [-0., -1.,  0.]])
>>> Formex('l:l').rosette(3,90.,around=(0.,1.,0.))
Formex([[[ 0.,  0.,  0.],
          [ 1.,  0.,  0.]],
        <BLANKLINE>
        [[ 1.,  1.,  0.],
          [ 1.,  2.,  0.]],
        <BLANKLINE>
        [[ 0.,  2.,  0.],
          [-1.,  2.,  0.]])
```

extrude (*args, **kargs)

Extrude a Formex along a straight line.

The Formex is extruded over a given length in the given direction. This operates by converting the Formex to a *Mesh*, extruding the Mesh with the given parameters, and converting the result back to a Formex.

Parameters: see *extrude()*.

Returns

- *Formex* – The Formex obtained by extruding the input Formex over the given *length* in direction *dir*, subdividing this length according
- to the seeds specified by *dir*. The plexitude of the result will be
- *double that of the input*.
- This method works by converting the Formex to a *Mesh*,
- using the *Mesh.extrude()* and then converting the result
- *back to a Formex*.

See also:

`connect ()` create a higher plexitude Formex by connecting Formices

Examples

```
>>> Formex(origin()).extrude(4,dir=0,length=3)
Formex([[ [ 0. , 0. , 0. ],
          [ 0.75, 0. , 0. ]],
<BLANKLINE>
        [ [ 0.75, 0. , 0. ],
          [ 1.5 , 0. , 0. ]],
<BLANKLINE>
        [ [ 1.5 , 0. , 0. ],
          [ 2.25, 0. , 0. ]],
<BLANKLINE>
        [ [ 2.25, 0. , 0. ],
          [ 3. , 0. , 0. ]]])
```

interpolate (*G*, *div*, *swap=False*)

Create linear interpolations between two Formices.

A linear interpolation of two equally shaped Formices *F* and *G* at parameter value *t* is an equally shaped Formex *H* where each coordinate is obtained from: $H_{ijk} = F_{ijk} + t * (G_{ijk} - F_{ijk})$. Thus, a `F.interpolate(G, [0., 0.5, 1.0])` will contain all elements of *F* and *G* and all elements with mean coordinates between those of *F* and *G*.

Parameters

- **G** (*Formex*) – A Formex with same shape as *self*.
- **div** (*int* or *list of floats*) – The list of parameter values for which to compute the interpolation. Usually, they are in the range 0.0 (*self*) to 1.0 (*X*). Values outside the range can be used however and result in linear extrapolations.

If an *int* is provided, a list with (*div*+1) parameter values is used, obtained by dividing the interval [0..1] into *div* equal segments. Then, specifying *div*=*n* is equivalent to specifying `div=arange(n+1)/float(n)`.

- **swap** (*bool*, *optional*) – If *swap*=*True*, the returned Formex will have the elements of the interpolation Formices interleaved. The default is to return a simple concatenation.

Returns *Formex* – A Formex with the concatenation of all generated interpolations, if *swap* is *False* (default). With *swap*=*True*, the elements of the interpolations are interleaved: first all the first elements from all the interpolations, then all the second elements, etc. The elements inherit the property numbers from *self*, if any. The Formex has the same *eltpe* as *self*, if it is set.

See also:

`coords.Coords.interpolate()`

Notes

See also example `Interpolate`.

Examples

```

>>> F = Formex([[0.0,0.0,0.0],[1.0,0.0,0.0]])
>>> G = Formex([[1.5,1.5,0.0],[4.0,3.0,0.0]])
>>> F.interpolate(G,div=3)
Formex([[ 0. ,  0. ,  0. ],
        [ 1. ,  0. ,  0. ]],
<BLANKLINE>
        [[ 0.5,  0.5,  0. ],
        [ 2. ,  1. ,  0. ]],
<BLANKLINE>
        [[ 1. ,  1. ,  0. ],
        [ 3. ,  2. ,  0. ]],
<BLANKLINE>
        [[ 1.5,  1.5,  0. ],
        [ 4. ,  3. ,  0. ]])
>>> F = Formex([[0.0,0.0,0.0],[1.0,0.0,0.0]])
>>> G = Formex([[1.5,1.5,0.0],[4.0,3.0,0.0]])
>>> F.interpolate(G,div=3)
Formex([[ 0. ,  0. ,  0. ],
<BLANKLINE>
        [[ 1. ,  0. ,  0. ]],
<BLANKLINE>
        [[ 0.5,  0.5,  0. ]],
<BLANKLINE>
        [[ 2. ,  1. ,  0. ]],
<BLANKLINE>
        [[ 1. ,  1. ,  0. ]],
<BLANKLINE>
        [[ 3. ,  2. ,  0. ]],
<BLANKLINE>
        [[ 1.5,  1.5,  0. ]],
<BLANKLINE>
        [[ 4. ,  3. ,  0. ]])
>>> F.interpolate(G,div=3,swap=True)
Formex([[ 0. ,  0. ,  0. ],
<BLANKLINE>
        [[ 0.5,  0.5,  0. ]],
<BLANKLINE>
        [[ 1. ,  1. ,  0. ]],
<BLANKLINE>
        [[ 1.5,  1.5,  0. ]],
<BLANKLINE>
        [[ 1. ,  0. ,  0. ]],
<BLANKLINE>
        [[ 2. ,  1. ,  0. ]],
<BLANKLINE>
        [[ 3. ,  2. ,  0. ]],
<BLANKLINE>
        [[ 4. ,  3. ,  0. ]])

```

subdivide (*div*)

Subdivide a plex-2 Formex at the parameter values in *div*.

Replaces each element of the plex-2 Formex (line segments) by a sequence of elements obtained by subdividing the Formex at the specified parameter values.

Parameters *div* (*int* or *list of floats*) – The list of parameter values at which to subdivide the elements. Usually, they are in the range 0.0 to 1.0.

If an int is provided, a list with $(div+1)$ parameter values is used, obtained by dividing the interval $[0..1]$ into div equal segments. Thus, specifying $div=n$ is equivalent to specifying $div=arange(n+1)/float(n)$.

Examples

```
>>> Formex('1:1').subdivide(4)
Formex([[ [ 0. , 0. , 0. ],
          [ 0.25, 0. , 0. ]],
<BLANKLINE>
          [ [ 0.25, 0. , 0. ],
            [ 0.5 , 0. , 0. ]],
<BLANKLINE>
          [ [ 0.5 , 0. , 0. ],
            [ 0.75, 0. , 0. ]],
<BLANKLINE>
          [ [ 0.75, 0. , 0. ],
            [ 1. , 0. , 0. ]]])
>>> Formex('1:1').subdivide([-0.1,0.3,0.7,1.1])
Formex([[ [-0.1, 0. , 0. ],
          [ 0.3, 0. , 0. ]],
<BLANKLINE>
          [ [ 0.3, 0. , 0. ],
            [ 0.7, 0. , 0. ]],
<BLANKLINE>
          [ [ 0.7, 0. , 0. ],
            [ 1.1, 0. , 0. ]]])
```

intersectionWithPlane ($p, n, atol=0.0$)

Compute the intersection of a Formex with a plane.

Note: This is currently only available for plexitude 2 (lines) and 3 (triangles).

Parameters

- **p** (*array_like* (3,)) – A point in the plane
- **n** (*array_like* (3,)) – The normal vector on the plane.
- **atol** (*float*) – A tolerance value: points whose distance from the plane is less than `atol` are considered to be lying in the plane.

Returns *Formex* – A Formex of plexitude `self.nplex()-1` holding the intersection with the plane (p,n). For a plex-2 Formex (lines), the returned Formex has plexitude 1 (points). For a plex-3 Formex (triangles) the returned Formex has plexitude 2 (lines).

See also:

[*cutWithPlane\(\)*](#) return parts of Formex after cutting with a plane

Examples

```
>>> Formex('1:1212').intersectionWithPlane([0.5,0.,0.],[-1.,1.,0.])
Formex([[ [ 0.5, 0. , 0. ]],
<BLANKLINE>
        [ [ 1. , 0.5, 0. ]],
<BLANKLINE>
        [ [ 1.5, 1. , 0. ]],
<BLANKLINE>
        [ [ 2. , 1.5, 0. ]]])
>>> Formex('3:.12.34').intersectionWithPlane([0.5,0.,0.],[1.,0.,0.])
Formex([[ [ 0.5, 0. , 0. ],
        [ 0.5, 0.5, 0. ]],
<BLANKLINE>
        [ [ 0.5, 0.5, 0. ],
        [ 0.5, 1. , 0. ]]])
```

cutWithPlane (*p*, *n*, *side=""*, *atol=None*, *newprops=None*)
 Cut a Formex with the plane (p,n).

Note: This is currently only available for plexitude 2 (lines) and 3 (triangles).

Parameters

- **p** (*array_like* (3,)) – A point in the cutting plane.
- **n** (*array_like* (3,)) – The normal vector to the cutting plane.
- **side** (*str*, one of '', '+' or '-') – Specifies which side of the plane should be returned. If an empty string (default), both sides are returned. If '+' or '-', only the part at the positive, resp. negative side of the plane (as defined by its normal) is returned.

Returns

- **Fpos** (*Formex*) – Formex with the part of the Formex at the positive side of the plane. This part is not returned is `side=='-'`.
- **Fneg** (*Formex*) – Formex with the part of the Formex at the negative side of the plane. This part is not returned is `side=='+'`.

Notes

Elements of the input Formex that are lying completely on one side of the plane will return unaltered. Elements that are cut by the plane are split up into multiple parts.

See also:

[`intersectionWithPlane\(\)`](#) return intersection of Formex and plane

lengths ()

Compute the length of all elements of a 2-plex Formex.

The length of an element is the distance between its two points.

Returns *float array (self.nelem(),)* – An array with the length of each element.

Raises `ValueError` – If the Formex is not of plexitude 2.

Examples

```
>>> Formex('1:127').lengths()
array([ 1. ,  1. ,  1.41])
```

areas()

Compute the areas of all elements of a 3-plex Formex.

The area of an element is the area of the triangle formed by its three points.

Returns *float array (self.nelem(),)* – An array with the area of each element.

Raises `ValueError` – If the Formex is not of plexitude 3.

Examples

```
>>> Formex('3:.12.34').areas()
array([ 0.5,  0.5])
```

volumes()

Compute the volume of all elements of a 4-plex Formex.

The volume of an element is the volume of the tetraeder formed by its 4 points.

Returns *float array (self.nelem(),)* – An array with the volume of each element.

Raises `ValueError` – If the Formex is not of plexitude 4.

Examples

```
>>> Formex('4:164I').volumes()
array([ 0.17])
```

classmethod fromstring(*s, sep=' ', nplex=1, ndim=3, count=-1*)

Create a *Formex* reading coordinates from a string.

This uses the `Coords.fromstring()` method to read coordinates from a string and restructures them into a Formex of the specified plexitude.

Parameters

- **s** (*str*) – A string containing a single sequence of float numbers separated by whitespace and a possible separator string.
- **sep** (*str, optional*) – The separator used between the coordinates. If not a space, all extra whitespace is ignored.
- **nplex** (*int, optional*) – Plexitude of the elements to be read.
- **ndim** (*int, optional*) – Number of coordinates per point. Should be 1, 2 or 3 (default). If 1, resp. 2, the coordinate string only holds x, resp. x,y values.
- **count** (*int, optional*) – Total number of coordinates to read. This should be a multiple of *ndim*. The default is to read all the coordinates in the string.

Returns *Formex* – A Formex object of the given plexitude, with the coordinates read from the string.

Raises `ValueError` – If count was provided and the string does not contain that exact number of coordinates. If the number of points read is not a multiple of nplex.

Examples

```
>>> Formex.fromstring('4 0 0 3 1 2 6 5 7', nplex=3)
Formex([[[ 4.,  0.,  0.],
          [ 3.,  1.,  2.],
          [ 6.,  5.,  7.]])
```

classmethod fromfile (*fil*, *nplex=1*, ***kargs*)

Read the coordinates of a Formex from a file

This uses `Coords.fromfile()` to read coordinates from a file and create a Formex of the specified plexitude. Coordinates X, Y and Z for subsequent points are read from the file. The total number of coordinates on the file should be a multiple of 3.

Parameters

- **fil** (*str* or *file*) – If *str*, it is a file name. An open file object can also be passed
- **nplex** (*int*, *optional*) – Plexitude of the elements to be read.
- ****kargs** – Arguments to be passed to `numpy.fromfile()`.

Returns *Formex* – A Formex object of the given plexitude, with the coordinates read from the specified file.

Raises `ValueError` – If the number of coordinates read is not a multiple of $3 * nplex$.

See also:

`Coords.fromfile()` read a Coords object from file

`numpy.fromfile()` read an array to file

Functions defined in module formex

`formex.connect` (*Flist*, *nodid=None*, *bias=None*, *loop=False*, *eltype=None*)

Return a Formex which connects the Formices in list.

Creates a Formex of any plexitude by combining corresponding points from a number of Formices.

Parameters

- **Flist** (*list of Formices*) – The Formices to connect. The number of Formices in the list will be the plexitude of the newly created Formex. One point of an element in each Formex is taken to create a new element in the output Formex.
- **nodid** (*list of int*, *optional*) – List of point indices to be used from each of the input Formices. If provided, the list should have the same length as *Flist*. The default is to use the first point of each element.
- **bias** (*list of int*, *optional*) – List of element bias values for each of the input Formices. Element iteration in the Formices will start at this number. If provided, the list should have the same length as *Flist*. The default is to start at element 0.
- **loop** (*bool*) – If `False` (default), element generation will stop when the first input Formex runs out of elements. If `True`, element iteration in the shorted Formices will wrap around until all elements in all Formices have been used.

Returns

Formex – A Formex with plexitude equal to `len(Flist)`. Each element of the Formex consists of a point from the corresponding element of each of the Formices in list. By default this is the

first point of that element, but a `nodid` list may specify another point index. Corresponding elements in the Formices are by default those with the same element index; the `bias` argument may specify another value to start the element indexing for each of the input Formices.

If `loop` is `False` (default), the number of elements is the minimum over all Formices of the number of elements minus the corresponding bias. If `loop` is `True`, the number of elements is the maximum of the number of elements of all input Formices.

Notes

See also example `Connect`.

Examples

```
>>> F = Formex('1:1111')
>>> G = Formex('1:222')
>>> connect([F,G])
Formex([[ [ 1., 0., 0.],
          [ 0., 0., 0.]],
<BLANKLINE>
        [[ 2., 0., 0.],
          [ 0., 1., 0.]],
<BLANKLINE>
        [[ 3., 0., 0.],
          [ 0., 2., 0.]])
>>> connect([F,G],nodid=[0,1])
Formex([[ [ 1., 0., 0.],
          [ 0., 1., 0.]],
<BLANKLINE>
        [[ 2., 0., 0.],
          [ 0., 2., 0.]],
<BLANKLINE>
        [[ 3., 0., 0.],
          [ 0., 3., 0.]])
>>> connect([F,F],bias=[0,1])
Formex([[ [ 1., 0., 0.],
          [ 2., 0., 0.]],
<BLANKLINE>
        [[ 2., 0., 0.],
          [ 3., 0., 0.]],
<BLANKLINE>
        [[ 3., 0., 0.],
          [ 4., 0., 0.]])
>>> connect([F,F],bias=[0,1],loop=True)
Formex([[ [ 1., 0., 0.],
          [ 2., 0., 0.]],
<BLANKLINE>
        [[ 2., 0., 0.],
          [ 3., 0., 0.]],
<BLANKLINE>
        [[ 3., 0., 0.],
          [ 4., 0., 0.]],
<BLANKLINE>
        [[ 4., 0., 0.],
          [ 1., 0., 0.]])
```

`formex.interpolate` (*self*, *G*, *div*, *swap=False*)

Create linear interpolations between two Formices.

A linear interpolation of two equally shaped Formices *F* and *G* at parameter value *t* is an equally shaped Formex *H* where each coordinate is obtained from: $H_{ijk} = F_{ijk} + t * (G_{ijk} - F_{ijk})$. Thus, a `F.interpolate(G, [0., 0.5, 1.0])` will contain all elements of *F* and *G* and all elements with mean coordinates between those of *F* and *G*.

Parameters

- **G** (*Formex*) – A Formex with same shape as *self*.
- **div** (*int or list of floats*) – The list of parameter values for which to compute the interpolation. Usually, they are in the range 0.0 (*self*) to 1.0 (*X*). Values outside the range can be used however and result in linear extrapolations.

If an *int* is provided, a list with (*div*+1) parameter values is used, obtained by dividing the interval [0..1] into *div* equal segments. Then, specifying *div*=*n* is equivalent to specifying `div=arange(n+1)/float(n)`.

- **swap** (*bool, optional*) – If *swap*=True, the returned Formex will have the elements of the interpolation Formices interleaved. The default is to return a simple concatenation.

Returns *Formex* – A Formex with the concatenation of all generated interpolations, if *swap* is False (default). With *swap*=True, the elements of the interpolations are interleaved: first all the first elements from all the interpolations, then all the second elements, etc. The elements inherit the property numbers from *self*, if any. The Formex has the same *eltype* as *self*, if it is set.

See also:

`coords.Coords.interpolate()`

Notes

See also example Interpolate.

Examples

```
>>> F = Formex([[0.0,0.0,0.0],[1.0,0.0,0.0]])
>>> G = Formex([[1.5,1.5,0.0],[4.0,3.0,0.0]])
>>> F.interpolate(G,div=3)
Formex([[ [ 0. ,  0. ,  0. ],
          [ 1. ,  0. ,  0. ]],
<BLANKLINE>
          [ [ 0.5,  0.5,  0. ],
            [ 2. ,  1. ,  0. ]],
<BLANKLINE>
          [ [ 1. ,  1. ,  0. ],
            [ 3. ,  2. ,  0. ]],
<BLANKLINE>
          [ [ 1.5,  1.5,  0. ],
            [ 4. ,  3. ,  0. ]]])
>>> F = Formex([[0.0,0.0,0.0]],[[1.0,0.0,0.0]])
>>> G = Formex([[1.5,1.5,0.0]],[[4.0,3.0,0.0]])
>>> F.interpolate(G,div=3)
Formex([[ [ 0. ,  0. ,  0. ]],
<BLANKLINE>
          [ [ 1. ,  0. ,  0. ]],
```

(continues on next page)

(continued from previous page)

```

<BLANKLINE>
    [[ 0.5,  0.5,  0. ]],
<BLANKLINE>
    [[ 2. ,  1. ,  0. ]],
<BLANKLINE>
    [[ 1. ,  1. ,  0. ]],
<BLANKLINE>
    [[ 3. ,  2. ,  0. ]],
<BLANKLINE>
    [[ 1.5,  1.5,  0. ]],
<BLANKLINE>
    [[ 4. ,  3. ,  0. ]]])
>>> F.interpolate(G,div=3,swap=True)
Formex([[[ 0. ,  0. ,  0. ]],
<BLANKLINE>
    [[ 0.5,  0.5,  0. ]],
<BLANKLINE>
    [[ 1. ,  1. ,  0. ]],
<BLANKLINE>
    [[ 1.5,  1.5,  0. ]],
<BLANKLINE>
    [[ 1. ,  0. ,  0. ]],
<BLANKLINE>
    [[ 2. ,  1. ,  0. ]],
<BLANKLINE>
    [[ 3. ,  2. ,  0. ]],
<BLANKLINE>
    [[ 4. ,  3. ,  0. ]]])

```

6.1.3 arraytools — A collection of numerical utilities.

This module contains a large collection of numerical utilities. Many of them are related to processing arrays. Some are similar to existing NumPy functions but offer some extended functionality.

Note: While these functions were historically developed for pyFormex, this module only depends on numpy and can be used outside of pyFormex without changes.

Definitions imported from numpy

These all have the obvious meaning. Seed numpy documentation for details: **pi**, **sin**, **cos**, **tan**, **arcsin**, **arccos**, **arctan**, **arctan2**, **sqrt**, **abs** **linalg.norm**

Variables defined in module arraytools

```
arraytools.Float = <class 'numpy.float32'>
```

Single-precision floating-point number type, compatible with C `float`. Character code: `'f'`. Canonical name: `np.single`. Alias *on this platform*: `np.float32`: 32-bit-precision floating-point number type: sign bit, 8 bits exponent, 23 bits mantissa.

`arraytools.Int = <class 'numpy.int32'>`

Signed integer type, compatible with C `int`. Character code: 'i'. Canonical name: `np.intc`. Alias *on this platform*: `np.int32`: 32-bit signed integer (-2147483648 to 2147483647).

`arraytools.DEG = 0.017453292519943295`

multiplier to transform degrees to radians = $\pi/180$.

Type float

`arraytools.RAD = 1.0`

multiplier to transform radians to radians

Type float

`arraytools.golden_ratio = 1.618033988749895`

golden ratio is defined as $0.5 * (1.0 + \text{sqrt}(5.))$

Functions defined in module arraytools

`arraytools.isInt (obj)`

Test if an object is an integer number.

Returns *bool* – True if the object is a single integer number (a Python `int` or a `numpy.integer` type), else False.

Examples

```
>>> isInt(1)
True
>>> isInt(np.arange(3)[1])
True
```

`arraytools.isFloat (obj)`

Test if an object is a floating point number.

Returns *bool* – True if the object is a single floating point number (a Python `float` or a `numpy.floating` type), else False.

Examples

```
>>> isFloat(1.)
True
>>> isFloat(array([1,2], dtype=np.float32)[1])
True
```

`arraytools.isNum (obj)`

Test if an object is an integer or a floating point number.

Returns *bool* – True if the object is a single integer or floating point number, else False. The type of the object can be either a Python `int` or `float` or a `numpy.integer` or `floating`.

Examples

```

>>> isNum(1)
True
>>> isNum(1.0)
True
>>> isNum(array([1,2], dtype=np.int32)[1])
True
>>> isNum(array([1,2], dtype=np.float32)[1])
True

```

`arraytools.checkInt` (*value*, *min=None*, *max=None*)

Check that a value is an int in the range min..max.

Parameters

- **value** (*int-like*) – The value to check.
- **min** (*int*, *optional*) – If provided, minimal value to be accepted.
- **max** (*int*, *optional*) – If provided, maximal value to be accepted.

Returns `checked_int` (*int*) – An integer not exceeding the provided boundaries.

Raises `ValueError`: – If the value is not convertible to an integer type or exceeds one of the specified boundaries.

Examples

```

>>> checkInt(1)
1
>>> checkInt(1, min=0, max=1)
1
>>> checkInt('2', min=0)
2

```

`arraytools.checkFloat` (*value*, *min=None*, *max=None*)

Check that a value is a float in the range min..max.

Parameters

- **value** (*float-like*) – The value to check
- **min** (*float-like*, *optional*) – If provided, minimal value to be accepted.
- **max** (*float-like*, *optional*) – If provided, maximal value to be accepted.

Returns `checked_float` (*float*) – A float not exceeding the provided boundaries.

Raises `ValueError`: – If the value is not convertible to a float type or exceeds one of the specified boundaries.

Examples

```

>>> checkFloat(1)
1.0
>>> checkFloat(1, min=0, max=1)
1.0
>>> checkFloat('2', min=0)
2.0

```

`arraytools.checkBroadcast` (*shape1*, *shape2*)

Check that two array shapes are broadcast compatible.

In many numerical operations, NumPy will automatically broadcast arrays of different shapes to a single shape, if they have broadcast compatible shapes. Two array shapes are broadcast compatible if, in all the last dimensions that exist in both arrays, either the shape of both arrays has the same length, or one of the shapes has a length 1.

Parameters

- **shape1** (*tuple of ints*) – Shape of first array
- **shape2** (*tuple of ints*) – Shape of second array

Returns *tuple of ints* – The broadcasted shape of the arrays.

Raises *ValueError: Shapes are not broadcast compatible* – If the two shapes can not be broadcast to a single one.

Examples

```
>>> checkBroadcast((8,1,6,1),(7,1,5))
(8, 7, 6, 5)
>>> checkBroadcast((5,4),(1,))
(5, 4)
>>> checkBroadcast((5,4),(4,))
(5, 4)
>>> checkBroadcast((15,3,5),(15,1,5))
(15, 3, 5)
>>> checkBroadcast((15,3,5),(3,5))
(15, 3, 5)
>>> checkBroadcast((15,3,5),(3,1))
(15, 3, 5)
>>> checkBroadcast((7,1,5),(8,1,6,1))
(8, 7, 6, 5)
```

`arraytools.checkArray` (*a*, *shape=None*, *kind=None*, *allow=None*, *size=None*, *ndim=None*, *bcast=None*)

Check that an array *a* has the correct shape, type and/or size.

Parameters

- **a** – Anything that can be converted into a numpy array.
- **shape** (*tuple of ints, optional*) – If provided, the shape of the array should match this value along each axis for which a nonzero value is specified. The length of the shape tuple should also match.
- **kind** (*dtype.kind character code, optional*) – If provided, the array's `dtype.kind` should match this value, or one of the values in `allow`, if provided.
- **allow** (*string of dtype.kind character codes, optional*) – If provided, and `kind` is also specified, any of the specified array types will also be accepted if it is convertible to the specified `kind`. See also Notes below.
- **size** (*int, optional*) – If provided, the total array size should match this value.
- **ndim** (*int, optional*) – If provided, the array should have precisely `ndim` dimensions.
- **bcast** (*tuple of ints, optional*) – If provided, the array's shape should be broadcast compatible with the specified shape.

Returns *array* – The `checked_array` is equivalent to the input data. It has the same contents and shape. It also has the same type, unless `kind` is provided, in which case the result is converted to this type.

Raises *ValueError: invalid array* – The input data failed for one of more of the tests requested by the provided parameters.

Notes

Currently, the only allowed conversion from an `allow` type to `kind` type, is to 'f'. Thus specifying `kind='f', allow='i'` will accept integer input but return float32 output.

See also:

`checkArray1D()`

Examples

```
>>> checkArray([1,2])
array([1, 2])
>>> checkArray([1,2], shape=(2,))
array([1, 2])
>>> checkArray([[1,2],[3,4]], shape=(2,-1))
array([[1, 2], [3, 4]])
>>> checkArray([1,2], kind='i')
array([1, 2])
>>> checkArray([1,2], kind='f', allow='i')
array([ 1.,  2.])
>>> checkArray([1,2], size=2)
array([1, 2])
>>> checkArray([1,2], ndim=1)
array([1, 2])
>>> checkArray([[1,2],[3,4]], bcast=(3,1,2))
array([[1, 2], [3, 4]])
```

`arraytools.checkArray1D(a, kind=None, allow=None, size=None)`

Check and force an array to be 1D.

This is equivalent to calling `checkArray()` without the `shape` and `ndim` parameters, and then turning the result into a 1D array.

:param See `checkArray()` .:

Returns *1D array* – The `checked_array` holds the same data as the input, but the shape is rveled to 1D. It also has the same type, unless `kind` is provided, in which case the result is converted to this type.

Examples

```
>>> checkArray1D([[1,2],[3,4]], size=4)
array([1, 2, 3, 4])
```

`arraytools.checkUniqueNumbers(nrs, nmin=0, nmax=None)`

Check that an array contains a set of unique integers in a given range.

This functions tests that all integer numbers in the array are within the range $\text{math:nmin} \leq i < \text{nmax}$. Default range is [0,unlimited].

Parameters

- **nrs** (*array_like*, int) – Input array with integers to check against provided limits.
- **nmin** (*int* or *None*, *optional*) – If not None, no value in a should be lower than this.
- **nmax** (–) –
- **nmax** – If provided, no value in a should be higher than this.

Returns *1D int array* – Containing the sorted unique numbers from the input.

Raises *ValueError* – If the numbers are not unique or some input value surpasses one of the specified limmits.

Examples

```
>>> checkUniqueNumbers([0,5,1,7,2])
array([0, 1, 2, 5, 7])
>>> checkUniqueNumbers([0,5,1,7,-2],nmin=None)
array([-2, 0, 1, 5, 7])
```

`arraytools.addAxis` (*a*, *axis*, *warn=True*)

Add an additional axis with length 1 to an array.

Parameters

- **a** – The array in wich to add an extra axis.
- **axis** (*int*) – Position in the expanded array where the new axis is created. Should be in the range [-a.ndim,a.ndim].

Returns *term:* – Same type and data as *a*, but shape has one extra axis with length 1 in the specified position.

Note: This function is now equivalent to `numpy.expand_dims`. If a negative value is specified, a warning will be issued because of broken compatibility with older pyFormex versions ($\leq 1.0.4$).

Examples

```
>>> A = array([[1,2,3],[4,5,6]])
>>> addAxis(A,0)
array([[[1, 2, 3],
        [4, 5, 6]]])
>>> addAxis(A,1)
array([[[1, 2, 3],
<BLANKLINE>
        [4, 5, 6]]])
>>> addAxis(A,-1)
array([[1],
        [2],
        [3]],
```

(continues on next page)

(continued from previous page)

```
<BLANKLINE>
    [[4],
     [5],
     [6]])
```

`arraytools.growAxis(a, add, axis=-1, fill=0)`

Increase the length of an array axis.

Parameters

- **a** – The array in which to extend n axis.
- **add** (*int*) – The length over which the specified axis should grow. If `add<=0`, the array is returned unchanged.
- **axis** (*int*) – Position of the target axis in the array. Default is last (-1).
- **fill** (*int or float*) – Value to set the new elements along the grown axis to.

Returns *term*: – Same type and data as *a*, but length of specified axis has been increased with a value *add* and the new elements are filled with the value *fill*.

Raises `ValueError`: – If the specified axis exceeds the array dimensions.

Examples

```
>>> growAxis([[1,2,3],[4,5,6]],2)
array([[1, 2, 3, 0, 0],
       [4, 5, 6, 0, 0]])
>>> growAxis([[1,2,3],[4,5,6]],1,axis=0,fill=-3)
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [-3, -3, -3]])
>>> growAxis([[1,2,3],[4,5,6]],-1)
array([[1, 2, 3],
       [4, 5, 6]])
```

`arraytools.reorderAxis(a, order, axis=-1)`

Reorder the planes of an array along the specified axis.

Parameters

- **a** – The array in which to reorder the elements.
- **order** (*int array_like | str*) – Specifies how to reorder the elements. It can be an integer index which should be a permutation of `arange(a.shape[axis])`. Each value in the index specified the old index of the elements that should be placed at its position. This is equivalent to `a.take(order,axis)`.

order can also be one of the following predefined sting values, resulting in the corresponding renumbering scheme being generated:

- 'reverse': the elements along axis are placed in reverse order
- 'random': the elements along axis are placed in random order

- **axis** (*int*) – The axis of the array along which the elements are to be reordered. Default is last (-1).

Returns *term*: – Same type and data as *a*, but the element planes are along *axis* have been reordered.

Examples

```
>>> reorderAxis([[1,2,3],[4,5,6]], [2,0,1])
array([[3, 1, 2],
       [6, 4, 5]])
```

`arraytools.reverseAxis(a, axis=-1)`

Reverse the order of the elements along an axis.

Parameters

- **a** – The array in which to reorder the elements.
- **axis** (*int*) – The axis of the array along which the elements are to be reordered. Default is last (-1).

Returns *term*: – Same type and data as *a*, but the elements along *axis* are now in reversed order.

Note: This function is especially useful if *axis* has a computed value. If the axis is known in advance, it is more efficient to use an indexing operation. Thus `reverseAxis(A,-1)` is equivalent to `A[...::-1]`.

Examples

```
>>> A = array([[1,2,3],[4,5,6]])
>>> reverseAxis(A)
array([[3, 2, 1],
       [6, 5, 4]])
>>> A[...::-1]
array([[3, 2, 1],
       [6, 5, 4]])
```

`arraytools.interleave(a, b)`

Interleave two arrays along their first axis.

Parameters

- **a** – First array
- **b** – Second array, with same type and shape as *a*, except that the first dimension may be one less than that of *a*, and if data type of *b* can be one that is convertible to that of *a*.

Returns *term*: – An array with interleaved rows from *a* and *b*. The array has the datatype of *a* and its first axis has the combined length of that of *a* and *b*.

Examples

```
>>> interleave(arange(4), 10*arange(3))
array([ 0,  0,  1, 10,  2, 20,  3])
>>> a = arange(8).reshape(2,4)
>>> print(interleave(a,10*a))
[[ 0  1  2  3]
 [ 0 10 20 30]
 [ 4  5  6  7]
 [40 50 60 70]]
```


`arraytools.multiplex(a, n, axis, warn=True)`

Multiplex an array over a length `n` in direction of a new axis.

Inserts a new axis in the array at the specified position and repeats the data of the array `n` times in the direction of the new axis.

Parameters

- **a** – The input array.
- **n** (*int*) – Number of times to repeat the data in direction of *axis*.
- **axis** (*int, optional*) – Position of the new axis in the expanded array. Should be in the range `-a.ndim..a.ndim`.

Returns *array* – An array with `n` times the original data repeated in the direction of the specified axis.

See also:

[`repeatValues\(\)`](#) Repeat values in a 1-dim array a number of times

Examples

```
>>> a = arange(6).reshape(2,3)
>>> print(a)
[[0 1 2]
 [3 4 5]]
>>> print(multiplex(a,4,-1))
[[[0 0 0 0]
  [1 1 1 1]
  [2 2 2 2]]
 <BLANKLINE>
 [[3 3 3 3]
  [4 4 4 4]
  [5 5 5 5]]]
>>> print(multiplex(a,4,-2))
[[[0 1 2]
  [0 1 2]
  [0 1 2]
  [0 1 2]]
 <BLANKLINE>
 [[3 4 5]
  [3 4 5]
  [3 4 5]
  [3 4 5]]]
```

`arraytools.repeatValues(a, n)`

Repeat values in a 1-dim array a number of times.

Parameters

- **a** (*array_like*, 1-dim) – The input array. Can be a list or a single element.
- **n** (*int array_like*, 1-dim) – Number of times to repeat the corresponding value of `a`. If `n` has less elements than `a`, it is reused until the end of `a` is reached.

Returns *array* – An 1-dim array of the same dtype as `a` with the value `a[i]` repeated `n[i]` times.

See also:

`multiplex()` Multiplex an array over a length `n` in direction of a new axis

Examples

```
>>> repeatValues(2,3)
array([2, 2, 2])
>>> repeatValues([2,3],2)
array([2, 2, 3, 3])
>>> repeatValues([2,3,4],[2,3])
array([2, 2, 3, 3, 3, 4, 4])
>>> repeatValues(1.6,[3,5])
array([ 1.6,  1.6,  1.6])
```

`arraytools.concat` (*a*, *axis=0*)

Smart array concatenation ignoring empty arrays.

Parameters

- **a** (*list of arrays*) – All arrays should have same shape except for the length of the concatenation axis, or be empty arrays.
- **axis** (*int*) – The axis along which the arrays are concatenated.

Returns *array* – The concatenation of all non-empty arrays in the list, or an empty array if all arrays in the list are empty.

Note: This is just like `numpy.concatenate`, but allows empty arrays in the list and silently ignores them.

Examples

```
>>> concat([array([0,1]),array([]),array([2,3])])
array([0, 1, 2, 3])
```

`arraytools.splitrange` (*n*, *nblk*)

Split a range of integers 0..*n* in almost equal sized slices.

Parameters

- **n** (*int*) – Highest integer value in the range.
- **nblk** (*int*) – Number of blocks to split into. Should be $\leq n$ to allow splitting.

Returns *1-dim int array* – If $nblk \leq n$, returns the boundaries that divide the integers in the range 0..*n* in *nblk* almost equal slices. The outer boundaries 0 and *n* are included, so the length of the array is *nblk*+1. If $nblk > n$, returns `range(n+1)`, thus all slices have length 1.

Examples

```
>>> splitrange(7,3)
array([0, 2, 5, 7])
```

`arraytools.splitar` (*a*, *nblk*, *axis=0*, *close=False*)

Split an array in *nblk* subarrays along a given axis.

Parameters

- **a** – Array to be divided in subarrays.
- **nblk** (*int*) – Number of subarrays to obtain. The subarrays will be of almost the same size.
- **axis** (*int*;) – Axis along which to split the array (default 0)
- **close** (*bool*) – If True, the last item of each block will be repeated as the first item of the next block.

Returns *list of arrays* – A list of subarrays obtained by splitting *a* along the specified axis. All arrays have almost the same shape. The number of arrays is equal to *nblk*, unless *nblk* is larger than *a.shape[axis]*, in which case a list with only the original array is returned.

Examples

```
>>> splitar(arange(7),3)
[array([0, 1]), array([2, 3, 4]), array([5, 6])]
>>> splitar(arange(7),3,close=True)
[array([0, 1, 2]), array([2, 3, 4]), array([4, 5, 6])]
>>> X = array([[0.,1.,2.],[3.,4.,5.]])
>>> splitar(X,2)
[array([[ 0.,  1.,  2.]], array([[ 3.,  4.,  5.]])]
>>> splitar(np.matrix(X),2,axis=-1)
[matrix([[ 0.,  1.],
         [ 3.,  4.]])], matrix([[ 2.],
                                [ 5.]])]
>>> splitar(X,3)
[array([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.]])]
```

`arraytools.minmax(a, axis=-1)`

Compute the minimum and maximum along an axis.

Parameters

- **a** – The data array for which to compute the minimum and maximum.
- **axis** (*int*) – The array axis along which to compute the minimum and maximum.

Returns *array* – The array has the same dtype as *a*. It also has the same shape, except for the specified axis, which will have a length of 2. The first value along this axis holds the minimum value of the input, the second holds the maximum value.

Examples

```
>>> a = array([[ [1.,0.,0.], [0.,1.,0.] ],
...           [ [2.,0.,0.], [0.,2.,0.] ] ])
>>> print(minmax(a,axis=1))
[[[ 0. 0. 0.]
  [ 1. 1. 0.]]
<BLANKLINE>
[[ 0. 0. 0.]
 [ 2. 2. 0.]])
```

`arraytools.stretch(a, min=None, max=None, axis=None)`

Scale the values of an array to fill a given range.

Parameters

- **a** (*array_like*, int or float) – Input data.
- **min** (*int or float, optional*) – The targeted minimum value in the array. Same type as *a*. If not provided, the minimum of *a* is used.
- **max** (*int or float, optional*) – The targeted maximum value in the array. Same type as *a*. If not provided, the maximum of *a* is used.
- **axis** (*int, optional*) – If provided, each slice along the specified axis is independently scaled.

Returns *array* – Array of the same type and size as the input array, but in which the values have been linearly scaled to fill the specified range.

Examples

```
>>> stretch([1., 2., 3.], min=0, max=1)
array([ 0. , 0.5, 1. ])
>>> A = arange(6).reshape(2, 3)
>>> stretch(A, min=20, max=30)
array([[20, 22, 24],
       [26, 28, 30]])
>>> stretch(A, min=20, max=30, axis=1)
array([[20, 25, 30],
       [20, 25, 30]])
>>> stretch(A, max=30)
array([[ 0,  6, 12],
       [18, 24, 30]])
>>> stretch(A, min=2, axis=1)
array([[2, 4, 5],
       [2, 4, 5]])
>>> stretch(A.astype(Float), min=2, axis=1)
array([[ 2. ,  3.5,  5. ],
       [ 2. ,  3.5,  5. ]])
```

`arraytools.stringer(s, a)`

Nicely format a string followed by an array.

Parameters

- **s** (*str*) – String to appear before the formatted array
- **a** (*array*) – Array to be formatted after the string, with proper vertical alignment

Returns *str* – A multiline string where the first line consists of the string *s* and the first line of the formatted array, and the next lines hold the remainder of the array lines, properly indented to align with the first line of the array.

Examples

```
>>> print(stringer("Indented array: ", np.arange(4).reshape(2, 2)))
Indented array: [[0 1]
                 [2 3]]
```

`arraytools.array2str(a)`

String representation of an array.

This creates a string representation of an array. It is visually equivalent with `numpy.ndarray.__repr__` without the dtype, except for 'uint.' types.

Note: This function can be used to set the default string representation of numpy arrays, using the following:

```
import numpy as np
np.set_string_function(array2str)
```

To reset it to the default, do:

```
np.set_string_function(None)
```

Because this reference manual was created with the default numpy routine replaced with ours, you will never see the dtype, except for uint types. See also the examples below.

Parameters `a (array)` – Any `numpy.ndarray` object.

Returns

- *The string representation of the array as created by its*
- `__repr__` method, except that the dtype is left away.

Examples

```
>>> a = arange(5).astype(np.int8)
>>> print(array2str(a))
array([0, 1, 2, 3, 4])
>>> a
array([0, 1, 2, 3, 4])
```

Reset the numpy string function to its default. `>>> np.set_string_function(None) >>> a array([0, 1, 2, 3, 4], dtype=int8)`

Change back to ours. `>>> np.set_string_function(array2str) >>> a array([0, 1, 2, 3, 4])`

`arraytools.printar(s, a)`

Print a string followed by a vertically aligned array.

Parameters

- `s (str)` – String to appear before the formatted array
- `a (array)` – Array to be formatted after the string, with proper vertical alignment

Note: This is a shorthand for `print(stringar(s, a))`.

Examples

```
>>> printar("Indented array: ", np.arange(4).reshape(2,2))
Indented array: [[0 1]
                 [2 3]]
```

`arraytools.writeArray(fil, array, sep='')`

Write an array to an open file.

This uses `numpy.tofile()` to write an array to an open file.

Parameters

- **fil** (*file or str*) – Open file object or filename.
- **array** – The array to write to the file.
- **sep** (*str*) – If empty, the array is written in binary mode. If not empty, the array is written in text mode, with this string as separator between the elements.

See also:

`readArray()`

`arraytools.readArray(fil, dtype, shape, sep='')`

Read data for an array with known size and type from an open file.

This uses `numpy.fromfile()` to read an array with known shape and data type from an open file.

Parameters

- **fil** (*file or str*) – Open file object or filename.
- **dtype** (*data-type*) – Data type of the array to be read.
- **shape** (*tuple of ints*) – The shape of the array to be read.
- **sep** (*str*) – If not empty, the array is read in text mode, with this string as separator between the elements. If empty, the array is read in binary mode and an extra ‘n’ after the data will be stripped off

See also:

`writeArray()`

`arraytools.powers(x, n)`

Compute all the powers of x from zero up to n.

Parameters

- **x** (*int, float or array (int, float)*) – The number or numbers to be raised to the specified powers.
- **n** (*int*) – Maximal power to raise the numbers to.

Returns `powers` (*list*) – A list of numbers or arrays of the same shape and type as the input. The list contains N+1 items, being the input raised to the powers in `range(n+1)`.

Examples

```
>>> powers(2, 5)
[1, 2, 4, 8, 16, 32]
>>> powers(array([1.0, 2.0]), 5)
[array([ 1.,  1.]), array([ 1.,  2.]), array([ 1.,  4.]), array([ 1.,  8.]),
array([ 1., 16.]), array([ 1., 32.])]
```

`arraytools.sind(arg, angle_spec=0.017453292519943295)`

Return the sine of an angle in degrees.

Parameters

- **arg** (*float number or array*) – Angle(s) for which the sine is to be returned. By default, angles are specified in degrees (see `angle_spec`).
- **angle_spec** (*DEG, RAD or float*) – Multiplier to apply to `arg` before taking the sine. The default multiplier DEG makes the argument being interpreted as an angle in degrees. Use RAD when angles are specified in radians.

Returns *float number or array* – The sine of the input angle(s)

See also:

`cosd()`, `tand()`, `arcsind()`, `arccosd()`, `arctand()`, `arctand2()`

Examples

```
>>> print(sind(30), sind(pi/6,RAD))
0.5 0.5
>>> sind(array([0.,30.,45.,60.,90.]))
array([ 0. ,  0.5 ,  0.71,  0.87,  1.  ])
```

`arraytools.cosd(arg, angle_spec=0.017453292519943295)`

Return the cosine of an angle in degrees.

Parameters

- **arg** (*float number or array*) – Angle(s) for which the cosine is to be returned. By default, angles are specified in degrees (see `angle_spec`).
- **angle_spec** (*DEG, RAD or float*) – Multiplier to apply to `arg` before taking the sine. The default multiplier DEG makes the argument being interpreted as an angle in degrees. Use RAD when angles are specified in radians.

Returns *float number or array* – The cosine of the input angle(s)

See also:

`sind()`, `tand()`, `arcsind()`, `arccosd()`, `arctand()`, `arctand2()`

Examples

```
>>> print(cosd(60), cosd(pi/3,RAD))
0.5 0.5
```

`arraytools.tand(arg, angle_spec=0.017453292519943295)`

Return the tangens of an angle in degrees.

Parameters

- **arg** (*float number or array*) – Angle(s) for which the tangens is to be returned. By default, angles are specified in degrees (see `angle_spec`).
- **angle_spec** (*DEG, RAD or float*) – Multiplier to apply to `arg` before taking the sine. The default multiplier DEG makes the argument being interpreted as an angle in degrees. Use RAD when angles are specified in radians.

Returns *float number or array* – The tangens of the input angle(s)

See also:

`sind()`, `cosd()`, `arcsind()`, `arccosd()`, `arctand()`, `arctand2()`

Examples

```
>>> print(tand(45), tand(pi/4,RAD))
1.0 1.0
```

`arraytools.arcsind` (*arg*, *angle_spec*=0.017453292519943295)

Return the angle whose sine is equal to the argument.

Parameters

- **arg** (*float number or array, in the range -1.0 to 1.0.*) – Value(s) for which to return the arcsine.
- **angle_spec** (*DEG, RAD or float, nonzero.*) – Divisor applied to the resulting angles before returning. The default divisor DEG makes the angles be returned in degrees. Use RAD to get angles in radians.

Returns *float number or array* – The angle(s) for which the input value(s) is/are the cosine. The default *angle_spec*=DEG returns values in the range -90 to +90.

See also:

`sind()`, `cosd()`, `tand()`, `arccosd()`, `arctand()`, `arctand2()`

Examples

```
>>> print("{:.1f} {:.4f}".format(arcsind(0.5), arcsind(1.0,RAD)))
30.0 1.5708
>>> arcsind(-1)
-90.0
>>> arcsind(1)
90.0
```

`arraytools.arccosd` (*arg*, *angle_spec*=0.017453292519943295)

Return the angle whose cosine is equal to the argument.

Parameters

- **arg** (*float number or array, in the range -1.0 to 1.0.*) – Value(s) for which to return the arccos.
- **angle_spec** (*DEG, RAD or float, nonzero.*) – Divisor applied to the resulting angles before returning. The default divisor DEG makes the angles be returned in degrees. Use RAD to get angles in radians.

Returns *float number or array* – The angle(s) for which the input value(s) is/are the cosine. The default *angle_spec*=DEG returns values in the range 0 to 180.

See also:

`sind()`, `cosd()`, `tand()`, `arcsind()`, `arctand()`, `arctand2()`

Examples

```
>>> print("{:.1f} {:.4f}".format(arccosd(0.5), arccosd(-1.0,RAD)))
60.0 3.1416
>>> arccosd(array([-1,0,1]))
array([ 180.,   90.,   0.])
```


`arraytools.arctand` (*arg*, *angle_spec*=0.017453292519943295)

Return the angle whose tangens is equal to the argument.

Parameters

- **arg** (*float number or array.*) – Value(s) for which to return the arctan.
- **angle_spec** (*DEG, RAD or float, nonzero.*) – Divisor applied to the resulting angles before returning. The default divisor DEG makes the angles be returned in degrees. Use RAD to get angles in radians.

Returns *float number or array* – The angle(s) for which the input value(s) is/are the tangens. The default *angle_spec*=DEG returns values in the range -90 to +90.

See also:

`sind()`, `cosd()`, `tand()`, `arcsind()`, `arccosd()`, `arctand2()`

Examples

```
>>> print("{:.1f} {:.4f}".format(arctand(1.0), arctand(-1.0,RAD)))
45.0 -0.7854
>>> arctand(array([-inf,-1,0,1,inf]))
array([-90., -45.,  0.,  45., 90.])
```

`arraytools.arctand2` (*sin, cos, angle_spec*=0.017453292519943295)

Return the angle whose sine and cosine values are given.

Parameters

- **sin** (*float number or array with same shape as cos.*) – Sine value(s) for which to return the corresponding angle.
- **cos** (*float number or array with same shape as sin*) – Cosine value(s) for which to return the corresponding angle.
- **angle_spec** (*DEG, RAD or float, nonzero.*) – Divisor applied to the resulting angles before returning. The default divisor DEG makes the angles be returned in degrees. Use RAD to get angles in radians.

Returns *float number or array with same shape as sin and cos.* – The angle(s) for which the input value(s) are the sine and cosine. The default *angle_spec*=DEG returns values in the range [-180, 180].

Note: The input values *sin* and *cos* are not restricted to the [-1.,1.] range. The returned angle is that for which the tangens is given by *sin/cos*, but with a sine and cosine that have the same sign as the *sin* and *cos* values.

See also:

`sind()`, `cosd()`, `tand()`, `arcsind()`, `arccosd()`, `arctand()`

Examples

```
>>> print("{:.1f} {:.4f}".format(arctand2(0.0,-1.0), arctand2(-sqrt(0.5),-sqrt(0.
↳5),RAD)))
180.0 -2.3562
>>> arctand2(array([0., 1., 0., -1.]), np.array([1., 0., -1., 0.]))
array([ 0.,  90., 180., -90.])
>>> arctand2(2.,2.)
45.0
```

`arraytools.niceLogSize(f)`

Return an integer estimate of the magnitude of a float number.

Parameters *f* (*float*) – Value for which the integer magnitude has to be computed. The sign of the value is disregarded.

Returns *int* – An integer magnitude estimator for the input.

Note: The returned value is the smallest integer e such that $10^{**e} > \text{abs}(f)$. If positive, it is equal to the number of digits before the decimal point; if negative, it is equal to the number of leading zeros after the decimal point.

See also:

`nicenumber()`

Examples

```
>>> print([ niceLogSize(a) for a in [1.3, 35679.23, 0.4, 0.0004567, -1.3] ])
[1, 5, 0, -3, 1]
```

`arraytools.niceNumber(f, round=<ufunc 'ceil'>)`

Return a nice number close to $\text{abs}(f)$.

A nice number is a number which only has only one significant digit (in the decimal system).

Parameters

- *f* (*float*) – A float number to approximate with a nice number. The sign of *f* is disregarded.
- *round* (*callable*) – A function that rounds a float to the nearest integer. Useful functions are `ceil`, `floor` and `round` from either NumPy or Python's `math` module. Default is `numpy.ceil`.

Returns *float* – A float value close to the input value, but having only a single decimal digit.

Examples

```
>>> numbers = [ 0.0837, 0.867, 8.5, 83.7, 93.7]
>>> [ str(niceNumber(f)) for f in numbers ]
['0.09', '0.9', '9.0', '90.0', '100.0']
>>> [ str(niceNumber(f,round=np.floor)) for f in numbers ]
['0.08', '0.8', '8.0', '80.0', '90.0']
>>> [ str(niceNumber(f,round=np.round)) for f in numbers ]
['0.08', '0.9', '8.0', '80.0', '90.0']
```

`arraytools.isqrt(n)`

Compute the square root of an integer number.

Parameters `n` (*int*) – An integer number that is a perfect square.

Returns *int* – The square root from the input number

Raises `ValueError`: – If the input integer is not a perfect square.

Examples

```
>>> isqrt(36)
6
```

`arraytools.dotpr(A, B, axis=-1)`

Return the dot product of vectors of A and B in the direction of axis.

Parameters

- **A** (*float :term:*) – Array containing vectors in the direction of axis.
- **B** (*float :term:*) – Array containing vectors in the direction of axis. Same shape as A, or broadcast-compatible.
- **axis** (*int*) – Axis of A and B in which direction the vectors are layed out. Default is the last axis. A and B should have the same length along this axis.

Returns `float array_like`, shape as A and B with axis direction removed. – The elements contain the dot product of the vectors of A and B at that position.

Note: This multiplies the elements of the A and B and then sums them in the direction of the specified axis.

Examples

```
>>> A = array( [[1.0, 1.0], [1.0, -1.0], [0.0, 5.0]] )
>>> B = array( [[5.0, 3.0], [2.0, 3.0], [1.33, 2.0]] )
>>> print(dotpr(A,B))
[ 8. -1. 10.]
>>> print(dotpr(A,B,0))
[ 7. 10.]
```

`arraytools.length(A, axis=-1)`

Returns the length of the vectors of A in the direction of axis.

Parameters

- **A** (*float :term:*) – Array containing vectors in the direction of axis.
- **axis** (*int*) – Axis of A in which direction the vectors are layed out. Default is the last axis. A and B should have the same length along this axis.

Returns `float array_like`, shape of A with axis direction removed. – The elements contain the length of the vector in A at that position.

Note: This is equivalent with `sqrt(dotpr(A,A))`.

Examples

```
>>> A = array( [[1.0, 1.0], [1.0,-1.0], [0.0, 5.0]] )
>>> print(length(A))
[ 1.41 1.41 5. ]
>>> print(length(A,0))
[ 1.41 5.2 ]
```

`arraytools.normalize(A, axis=-1, on_zeros='n', return_length=False, ignore_zeros=False)`
 Normalize the vectors of A in the direction of axis.

Parameters

- **A** (*float :term:*) – Array containing vectors in the direction of axis.
- **axis** (*int*) – Axis of A in which direction the vectors are layed out.
- **on_zeros** (*'n', 'e' or 'i'*) – Specifies how to treat occurrences of zero length vectors (having all components equal to zero):
 - 'n': return a vector of nan values
 - 'e': raise a ValueError
 - 'i': ignore zero vectors and return them as such.
- **return_length** (*bool*) – If True, also returns also the length of the original vectors.
- **ignore_zeros** (*bool*) – (Deprecated) Equivalent to specifying `on_zeros='i'`.

Returns

- **norm** (*float array*) – Array with same shape as A but where each vector along axis has been rescaled so that its length is 1.
- **len** (*float array, optional*) – Array with shape like A but with axis removed. The length of the original vectors in the direction of axis. Only returned if `return_length=True` provided.

Raises *ValueError: Can not normalize zero-length vector* – If any of the vectors of B is a zero vector.

Examples

```
>>> A = array( [[3.0, 3.0], [4.0,-3.0], [0.0, 0.0]] )
>>> print(normalize(A))
[[ 0.71  0.71]
 [ 0.8  -0.6 ]
 [ nan  nan]]
>>> print(normalize(A,on_zeros='i'))
[[ 0.71  0.71]
 [ 0.8  -0.6 ]
 [ 0.   0.  ]]
>>> print(normalize(A,0))
[[ 0.6  0.71]
 [ 0.8  -0.71]
 [ 0.   0.  ]]
>>> n,l = normalize(A,return_length=True)
>>> print(n)
[[ 0.71  0.71]
 [ 0.8  -0.6 ]
```

(continues on next page)

(continued from previous page)

```
[ nan  nan]
>>> print(1)
[ 4.24  5.   0. ]
```

`arraytools.projection(A, B, axis=-1)`

Return the (signed) length of the projection of vectors of A on B.

Parameters

- **A** (*float :term:*) – Array containing vectors in the direction of axis.
- **B** (*float :term:*) – Array containing vectors in the direction of axis. Same shape as A, or broadcast-compatible.
- **axis** (*int*) – Axis of A and B in which direction the vectors are layed out. Default is the last axis. A and B should have the same length along this axis.

Returns float *array_like*, shape as A and B with axis direction removed. – The elements contain the length of the projections of vectors of A on the directions of the corresponding vectors of B.

Raises *ValueError: Can not normalize zero-length vector* – If any of the vectors of B is a zero vector.

Note: This returns `dotpr(A, normalize(B))`.

Examples

```
>>> A = [[2.,0.],[1.,1.],[0.,1.]]
>>> projection(A,[1.,0.])
array([ 2., 1., 0.])
>>> projection(A,[1.,1.])
array([ 1.41, 1.41, 0.71])
>>> projection(A,[[1.],[1.],[0.]],axis=0)
array([ 2.12, 0.71])
```

`arraytools.parallel(A, B, axis=-1)`

Return the component of vector of A that is parallel to B.

Parameters

- **B** (*A*) – Broadcast compatible arrays containing vectors in the direction of axis.
- **axis** (*int*) – Axis of A and B in which direction the vectors are layed out. Default is the last axis. A and B should have the same length along this axis.

Returns float *array_like*, same shape as A and B. – The vectors in the axis direction are the vectors of A projected on the direction of the corresponding vectors of B.

See also:

`orthog()`

Examples

```

>>> A = [[2.,0.],[1.,1.],[0.,1.]]
>>> parallel(A,[1.,0.])
array([[ 2.,  0.],
       [ 1.,  0.],
       [ 0.,  0.]])
>>> parallel(A,A)
array([[ 2.,  0.],
       [ 1.,  1.],
       [ 0.,  1.]])
>>> parallel(A,[1.],[1.],[0.],axis=0)
array([[ 1.5,  0.5],
       [ 1.5,  0.5],
       [ 0. ,  0. ]])

```

`arraytools.orthog(A, B, axis=-1)`

Return the component of vector of A that is orthogonal to B.

Parameters

- **A** (*float :term:*) – Array containing vectors in the direction of axis.
- **B** (*float :term:*) – Array containing vectors in the direction of axis. Same shape as A, or broadcast-compatible.
- **axis** (*int*) – Axis of A and B in which direction the vectors are layed out. Default is the last axis. A and B should have the same length along this axis.

Returns float *array_like*, same shape as A and B. – The vectors in the axis direction are the components of the vectors of A orthogonal to the direction of the corresponding vectors of B.

See also:

`parallel()`

Examples

```

>>> A = [[2.,0.],[1.,1.],[0.,1.]]
>>> orthog(A,[1.,0.])
array([[ 0.,  0.],
       [ 0.,  1.],
       [ 0.,  1.]])
>>> orthog(A,[1.],[1.],[0.],axis=0)
array([[ 0.5, -0.5],
       [-0.5,  0.5],
       [ 0. ,  1. ]])

```

`arraytools.inside(p, mi, ma)`

Return true if point p is inside bbox defined by points mi and ma.

Parameters

- **p** (float *array_like* with shape (ndim,)) – Point to test against the boundaries.
- **mi** (float *array_like* with shape (ndim,)) – Minimum values for the components of p
- **ma** (float *array_like* with shape (ndim,)) – Maximum values for the components of p

Returns *bool* – True is all components are inside the specified limits, limits included. This means that the n-dimensional point p lies within the n-dimensional rectangular bounding box defined by the two n-dimensional points (mi,ma).

Examples

```
>>> inside([0.5,0.5],[0.,0.],[1.,1.])
True
>>> inside([0.,1.],[0.,0.],[1.,1.])
True
>>> inside([0.,1.1],[0.,0.],[1.,1.])
False
```

`arraytools.unitVector(v)`

Return a unit vector in the direction of *v*.

Parameters *v* (a single integer or a (3,) shaped float :term:.) – If an int, it specifies one of the global axes (0,1,2). Else, it is a vector in 3D space.

Returns (3,) shaped float array – A unit vector along the specified direction.

Examples

```
>>> unitVector(1)
array([ 0.,  1.,  0.])
>>> unitVector([0.,3.,4.])
array([ 0. ,  0.6,  0.8])
```

`arraytools.rotationMatrix(angle, axis=None, angle_spec=0.017453292519943295)`

Create a 2D or 3D rotation matrix over angle, optionally around axis.

Parameters

- **angle** (*float*) – Rotation angle, by default in degrees.
- **axis** (int or (3,) float *array_like*, optional) – If not provided, a 2D rotation matrix is returned. If provided, it specifies the rotation axis in a 3D world. It is either one of 0,1,2, specifying a global axis, or a vector with 3 components specifying an axis through the origin. The returned matrix is 3D.
- **angle_spec** (*float, DEG or RAD, optional*) – The default (DEG) interpretes the angle in degrees. Use RAD to specify the angle in radians.

Returns *float array* – Rotation matrix which will rotate a vector over the specified angle. Shape is (3,3) if axis is specified, or (2,2) if not.

See also:

`rotationMatrix3()` subsequent rotation around 3 axes

`rotmat()` rotation matrix specified by three points in space

`trfmat()` transformation matrix to transform 3 points

`rotMatrix()` rotation matrix transforming global axis 0 into a given vector

`rotMatrix2()` rotation matrix that transforms one vector into another

Examples

```
>>> rotationMatrix(30.,1)
array([[ 0.87,  0.  , -0.5 ],
       [ 0.  ,  1.  ,  0.  ],
       [ 0.5 ,  0.  ,  0.87]])
>>> rotationMatrix(45.,[1.,1.,0.])
array([[ 0.85,  0.15, -0.5 ],
       [ 0.15,  0.85,  0.5 ],
       [ 0.5 , -0.5 ,  0.71]])
```

`arraytools.rotationMatrix3(rx, ry, rz, angle_spec=0.017453292519943295)`

Create a rotation matrix defined by three angles.

This applies successive rotations about the 0, 1 and 2 axes, over the angles rx, ry and rz, respectively. These angles are also known as the cardan angles.

Parameters

- **rx** (*float*) – Rotation angle around the 0 axis.
- **ry** (*float*) – Rotation angle around the 1 axis.
- **rz** (*float*) – Rotation angle around the 2 axis.
- **angle_spec** (*float, DEG or RAD, optional*) – The default (DEG) interpretes the angles in degrees. Use RAD to specify the angle in radians.

Returns *float array (3,3)* – Rotation matrix that performs the combined rotation equivalent to subsequent rotations around the three global axes.

See also:

`rotationMatrix()` rotation matrix specified by an axis and angle

`cardanAngles()` find cardan angles that produce a given rotation matrix

Examples

```
>>> rotationMatrix3(60,45,30)
array([[ 0.61,  0.35, -0.71],
       [ 0.28,  0.74,  0.61],
       [ 0.74, -0.57,  0.35]])
```

`arraytools.cardanAngles(R, angle_spec=0.017453292519943295)`

Compute cardan angles from rotation matrix

Computes the angles over which to rotate subsequently around the 0-axis, the 1-axis and the 2-axis to obtain the rotation corresponding to the given rotation matrix.

Parameters

- **R** ((3,3) float *array_like*) – Rotation matrix for post multiplication (see Notes)
- **angle_spec** (*DEG, RAD or float, nonzero.*) – Divisor applied to the resulting angles before returning. The default divisor DEG makes the angles be returned in degrees. Use RAD to get angles in radians.

Returns (**rx,ry,rz**) (*tuple of floats*) – The three rotation angles around that when applied subsequently around the global 0, 1 and 2 axes, yield the same rotation as the input. The default `angle_spec=DEG` returns the angles in degrees.

Notes

The returned angles are but one of many ways to obtain a given rotation by three subsequent rotations around frame axes. Look on the web for ‘Euler angles’ to get comprehensive information. Different sets of angles can be obtained depending on the sequence of rotation axes used, and whether fixed axes (extrinsic) or rotated axes (intrinsic) are used in subsequent rotations. The here obtained ‘cardan’ angles are commonly denoted as a zy’x” system with intrinsic angles or xyz with extrinsic angles. It is the latter angles that are returned.

Because pyFormex stores rotation matrices as post-multiplication matrices (to be applied on row-vectors), the combined rotation around first the 0-axis, then the 1-axis and finally the 2-axis, is found as the matrix product $R_x.R_y.R_z$. (Traditionally, vectors were often written as column matrices, and rotation matrices were pre-multiplication matrices, so the subsequent rotation matrices would have to be multiplied in reverse order.)

Even if one chooses a single frame system for the subsequent rotations, the resulting angles are not unique. There are infinitely many sets of angles that will result in the same rotation matrix. The implementation here results in angles r_x and r_z in the range $[-\pi, \pi]$, while the angle r_y will be in $[-\pi/2, \pi/2]$. Even then, there remain infinite solutions in the case where the elements $R[0,2] == R[2,0]$ equal +1 or -1 ($r_y = +\pi/2$ or $-\pi/2$). The result will then be the solution with $r_x == 0$.

Examples

```
>>> print ("%8.2f " * 3 % cardanAngles(rotationMatrix3(60,45,30)))
60.00  45.00  30.00
>>> print ("%8.2f " * 3 % cardanAngles(rotationMatrix3(0,90,77)))
0.00   90.00  77.00
>>> print ("%8.2f " * 3 % cardanAngles(rotationMatrix3(0,-90,30)))
0.00  -90.00  30.00
```

But:

```
>>> print ("%8.2f " * 3 % cardanAngles(rotationMatrix3(20,-90,30)))
0.00  -90.00  50.00
```

`arraytools.rotmat(x)`

Create a rotation matrix defined by 3 points in space.

Parameters **x** (*array_like* (3,3)) – The rows contain the coordinates in 3D space of three non-colinear points x_0, x_1, x_2 .

Returns

rotmat (*matrix*(3,3)) – Rotation matrix which transforms the global axes into a new (orthonormal) coordinate system with the following properties:

- the origin is at point x_0 ,
- the 0 axis is along the direction $x_1 - x_0$
- the 1 axis is in the plane (x_0, x_1, x_2) with x_2 lying at the positive side.

Notes

The rows of the rotation matrix represent the unit vectors of the resulting coordinate system. The coordinates in the rotated axes of any point are obtained by the reverse transformation, i.e. multiplying the point with the transpose of the rotation matrix.

See also:

`rotationMatrix()` rotation matrix specified by angle and axis
`trfmat()` transformation matrices defined by 2 sets of 3 points
`rotMatrix()` rotation matrix transforming global axis 0 into a given vector
`rotMatrix2()` rotation matrix that transforms one vector into another

Examples

```
>>> rotmat([[0,0,0],[1,0,0],[0,1,0]])
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> rotmat(eye(3,3))
array([[ -0.71,  0.71,  0. ],
       [-0.41, -0.41,  0.82],
       [ 0.58,  0.58,  0.58]])
>>> s,c = sind(30),cosd(30)
>>> R = rotmat([[0,0,0],[c,s,0],[0,1,0]])
>>> print(R)
[[ 0.87  0.5  0. ]
 [-0.5  0.87  0. ]
 [ 0.   -0.   1. ]]
>>> B = array([[2.,0.,0.],[3*s,3*c,3]])
>>> D = dot(B,R)      # Rotate some vectors with the matrix R
>>> print(D)
[[ 1.73  1.   0. ]
 [-0.   3.   3. ]]
```

`arraytools.trfmat(x,y)`

Find the transformation matrices from 3 points x into y .

Constructs the rotation matrix and translation vector that will transform the points x thus that:

- point x_0 coincides with point y_0 ,
- line x_0,x_1 coincides with line y_0,y_1
- plane x_0,x_1,x_2 coincides with plane y_0,y_1,y_2

Parameters

- \mathbf{x} (float *array_like* (3,3)) – Original coordinates of three non-collinear points.
- \mathbf{y} (float *array_like* (3,3)) – Final coordinates of the three points.

Returns

- **rot** (float *array* (3,3)) – The rotation matrix for the transformation x to y .
- **trf** – float *array*(3,) The translation vector for the transformation x to y , Obviously, this is equal to y_0-x_0 .

The rotation is to be applied first and should be around the first point x_0 . The full transformation of a `Coords` object is thus obtained by $(\text{coords}-x_0)*\text{rot}+\text{trf}+x_0 = \text{coords}*\text{rot}+(\text{trf}+x_0-x_0*\text{rot})$.

Examples

```
>>> R,T = trfmat(eye(3,3), [[0,0,0],[1,0,0],[0,1,0]])
>>> print(R)
[[-0.71 -0.41  0.58]
 [ 0.71 -0.41  0.58]
 [ 0.     0.82  0.58]]
>>> print(T)
[ 0.71  0.41 -0.58]
```

`arraytools.rotMatrix(u, w=[0.0, 0.0, 1.0])`

Create a rotation matrix that rotates global axis 0 to a given vector.

Parameters

- **u** (*(3,)* :term:) – Vector specifying the direction to which the global axis 0 should be rotated by the returned rotation matrix.
- **w** (*(3,)* :term:) – Vector that is not parallel to u. This vector is used to uniquely define the resulting rotation. It will be equivalent to rotating first around w, until the target u lies in the plane of the rotated axis 0 and w, then rotated in that plane until the rotated axis 0 coincides with u. See also Note. If a parallel w is provided, it will be replaced with a non-parallel one.

Returns *float array (3,3)* – Rotation matrix that transforms a vector [1.,0.,0.] into u. The returned matrix should be used in postmultiplication to the coordinates.

See also:

`rotMatrix2()` rotation matrix that transforms one vector into another

`rotationMatrix()` rotation matrix specified by an axis and angle

`rotmat()` rotation matrix specified by three points in space

`trfmat()` rotation and translation matrix that transform three points

Examples

```
>>> rotMatrix([1,0,0])
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> rotMatrix([0,1,0])
array([[ 0.,  1.,  0.],
       [-1.,  0.,  0.],
       [ 0., -0.,  1.]])
>>> rotMatrix([0,0,1])
array([[ 0.,  0.,  1.],
       [ 1.,  0.,  0.],
       [ 0.,  1.,  0.]])
>>> rotMatrix([0,1,1])
array([[ 0. ,  0.71,  0.71],
       [-1. ,  0. ,  0. ],
       [ 0. , -0.71,  0.71]])
>>> rotMatrix([1,0,1])
array([[ 0.71,  0. ,  0.71],
       [ 0. ,  1. ,  0. ],
```

(continues on next page)

(continued from previous page)

```

    [-0.71, 0. , 0.71]])
>>> rotMatrix([1,1,0])
array([[ 0.71,  0.71,  0. ],
       [-0.71,  0.71,  0. ],
       [ 0. , -0. ,  1. ]])
>>> rotMatrix([1,1,1])
array([[ 0.58,  0.58,  0.58],
       [-0.71,  0.71,  0. ],
       [-0.41, -0.41,  0.82]])

```

```

>>> dot([1,0,0], rotMatrix([1,1,1]))
array([ 0.58,  0.58,  0.58])

```

`arraytools.rotMatrix2` (*vec1*, *vec2*, *upvec=None*)

Create a rotation matrix that rotates a vector *vec1* to *vec2*.

Parameters

- **vec1** ((3,) :term:) – Original vector.
- **vec2** ((3,) :term:) – Direction of *vec1* after rotation.
- **upvec** ((3,) *array_like*, optional) – If provided, the rotation matrix will be such that the plane of *vec2* and the rotated *upvec* will be parallel to the original *upvec*. If not provided, the rotation matrix will perform a rotation around the normal to the plane of the two vectors.

Returns *float array* (3,3) – Rotation matrix that transforms a vector [1.,0.,0.] into *u*. The returned matrix should be used in postmultiplication to the coordinates.

See also:

rotMatrix() rotation matrix transforming global axis 0 into a given vector

rotationMatrix() rotation matrix specified by an axis and angle

rotmat() rotation matrix specified by three points in space

trfmat() rotation and translation matrix that transform three points

Examples

```

>>> rotMatrix2([1,0,0],[1,0,0])
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> rotMatrix2([1,0,0],[0,1,0])
array([[ 0.,  1.,  0.],
       [-1.,  0.,  0.],
       [ 0.,  0.,  1.]])
>>> rotMatrix2([1,0,0],[0,0,1])
array([[ 0.,  0.,  1.],
       [ 0.,  1.,  0.],
       [-1.,  0.,  0.]])
>>> rotMatrix2([1,0,0],[0,1,1])
array([[ 0. ,  0.71,  0.71],
       [-0.71,  0.5 , -0.5 ],
       [-0.71, -0.5 ,  0.5 ]])

```

(continues on next page)

(continued from previous page)

```

>>> rotMatrix2([1,0,0],[1,0,1])
array([[ 0.71,  0.  ,  0.71],
       [ 0.  ,  1.  ,  0.  ],
       [-0.71,  0.  ,  0.71]])
>>> rotMatrix2([1,0,0],[1,1,0])
array([[ 0.71,  0.71,  0.  ],
       [-0.71,  0.71,  0.  ],
       [ 0.  ,  0.  ,  1.  ]])
>>> rotMatrix2([1,0,0],[1,1,1])
array([[ 0.58,  0.58,  0.58],
       [-0.58,  0.79, -0.21],
       [-0.58, -0.21,  0.79]])

```

```

>>> rotMatrix2([1,0,0],[1,0,0],[0,0,1])
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> rotMatrix2([1,0,0],[0,1,0],[0,0,1])
array([[ 0.,  1.,  0.],
       [-1.,  0.,  0.],
       [ 0.,  0.,  1.]])
>>> rotMatrix2([1,0,0],[0,0,1],[0,0,1])
array([[ 0.,  0.,  1.],
       [ 1.,  0.,  0.],
       [ 0.,  1.,  0.]])
>>> rotMatrix2([1,0,0],[0,1,1],[0,0,1])
array([[ 0.  ,  0.71,  0.71],
       [-1.  ,  0.  ,  0.  ],
       [ 0.  , -0.71,  0.71]])
>>> rotMatrix2([1,0,0],[1,0,1],[0,0,1])
array([[ 0.71,  0.  ,  0.71],
       [ 0.  ,  1.  ,  0.  ],
       [-0.71,  0.  ,  0.71]])
>>> rotMatrix2([1,0,0],[1,1,0],[0,0,1])
array([[ 0.71,  0.71,  0.  ],
       [-0.71,  0.71,  0.  ],
       [ 0.  ,  0.  ,  1.  ]])
>>> rotMatrix2([1,0,0],[1,1,1],[0,0,1])
array([[ 0.58,  0.58,  0.58],
       [-0.71,  0.71,  0.  ],
       [-0.41, -0.41,  0.82]])

```

`arraytools.abat(a, b)`

Compute the matrix product $a * b * at$.

Parameters

- **a** (*array_like*, 2-dim) – Array with shape (m,n).
- **b** (*array_like*, 2-dim) – Array with square shape (n,n).

Returns *array* – Array with shape (m,m) holding the matrix product $a * b * at$.

See also:

`atba()`

Examples

```
>>> abat([[1],[2]],[[3]])
array([[ 3,  6],
       [ 6, 12]])
>>> abat([[1,2]],[[0,1],[2,3]])
array([[18]])
```

`arraytools.atba(a, b)`

Compute the matrix product $a * b * a$

Parameters

- **a** (*array_like*, 2-dim) – Array with shape (n,m).
- **b** (*array_like*, 2-dim) – Array with square shape (n,n).

Returns *array* – Array with shape (m,m) holding the matrix product $a * b * a$.

Note: This multiplication typically occurs when rotating a symmetric tensor *b* to axes defined by the rotation matrix *a*.

See also:

`abat()`

Examples

```
>>> atba([[1,2]],[[3]])
array([[ 3,  6],
       [ 6, 12]])
>>> atba([[1],[2]],[[0,1],[2,3]])
array([[18]])
```

`arraytools.horner(a, u)`

Compute the value of a polynomial using Horner's rule.

Parameters

- **a** (float *array_like* (n+1,nd)) – *nd*-dimensional coefficients of a polynomial of degree *n* in a scalar variable *u*. The coefficients are in order of increasing degree.
- **u** (float *array_like* (nu)) – Parametric values where the polynomial is to be evaluated.

Returns *float array*(*nu,nd*) – The *nd*-dimensional values of the polynomial at the specified *nu* parameter values.

Examples

```
>>> print(horner([[1.,1.,1.],[1.,2.,3.]], [0.5,1.0]))
[[ 1.5 2.  2.5]
 [ 2.  3.  4. ]]
```

`arraytools.solveMany(A, b, direct=True)`

Solve many systems of linear equations.

Parameters

- **A** (*(ndof, ndof, nsys) shaped float array*) – Coefficient matrices for nsys systems of ndof linear equations in ndof unknowns.
- **b** (*(ndof, nrhs, nsys) shaped float array*) – Right hand sides for each of the nsys systems of linear equations. Each of the nsys systems is solved simultaneously for nrhs right hand sides.
- **direct** (*bool*) – If True (default), systems with ndof=1, 2 or 3 are solved with a (faster) direct method instead of using the general linear equation solver.

Returns **x** (float array with same shape as *b*) – The set of values $x[:, i, j]$ that solve the systems of linear equations $A[:, :, j].x[:, i, j] = b[:, i, j]$.

`arraytools.cubicEquation(a, b, c, d)`

Solve a cubiq equation using a direct method.

Given a polynomial equation of the third degree with real coefficients:

$$a*x**3 + b*x**2 + c*x + d = 0$$

Such an equation (with a non-zero) always has exactly three roots, with some possibly being complex, or identical. This function computes all three roots of the equation and returns full information about their nature, multiplicity and sorting order. It uses scaling of the variables to enhance the accuracy.

Parameters

- **a** (*float*) – Coefficient of the third degree term.
- **b** (*float*) – Coefficient of the second degree term.
- **c** (*float*) – Coefficient of the first degree term.
- **d** (*float*) – Constant in the third degree polynomial.

Returns

- **r1** (*float*) – First real root of the cubiq equation
- **r2** (*float*) – Second real root of the cubiq equation or real part of the complex conjugate second and third root.
- **r3** (*float*) – Third real root of the cubiq equation or imaginary part of the complex conjugate second and third root.
- **kind** (*int*) – A value specifying the nature and ordering of the roots:

kind	roots
0	three real roots $r1 < r2 < r3$
1	three real roots $r1 < r2 = r3$
2	three real roots $r1 = r2 < r3$
3	three real roots $r1 = r2 = r3$
4	one real root $r1$ and two complex conjugate roots with real part $r2$ and imaginary part $r3$; the complex roots are thus: $r2+i*r3$ en $r2-i*r3$, where $i=\text{sqrt}(-1)$.

Raises `ValueError`: – If the coefficient $a=0$ and the equation reduces to a second degree.

Examples

```
>>> cubicEquation(1.,-6.,11.,-6.)
(array([ 1.,  2.,  3.]), 0)
>>> cubicEquation(1.,-2.,1.,0.)
(array([-0.,  1.,  1.]), 1)
>>> cubicEquation(1.,-5.,8.,-4.)
(array([ 1.,  2.,  2.]), 1)
>>> cubicEquation(1.,-4.,5.,-2.)
(array([ 1.,  1.,  2.]), 2)
>>> cubicEquation(1.,-3.,3.,-1.)
(array([ 1.,  1.,  1.]), 3)
>>> cubicEquation(1.,-1.,1.,-1.)
(array([ 1.,  0.,  1.]), 4)
>>> cubicEquation(1.,-3.,4.,-2.)
(array([ 1.,  1.,  1.]), 4)
```

`arraytools.renumberIndex` (*index*)

Renumber an index sequentially.

Given a one-dimensional integer array with only non-negative values, and *nval* being the number of different values in it, and you want to replace its elements with values in the range *0..nval*, such that identical numbers are always replaced with the same number and the new values at their first occurrence form an increasing sequence *0..nval*. This function will give you the old numbers corresponding with each position *0..nval*.

Parameters *index* (1-dim int *array_like*) – Array with non-negative integer values.

Returns *int array* – A 1-dim int array with length equal to *nval*, where *nval* is the number of different values in *index*. The elements are the original values corresponding to the new values *0..nval*.

See also:

[*inverseUniqueIndex* \(\)](#) get the inverse mapping.

Examples

```
>>> renumberIndex([0,5,2,2,6,0])
array([0, 5, 2, 6])
>>> inverseUniqueIndex(renumberIndex([0,5,2,2,6,0]))[0,5,2,2,6,0]
array([0, 1, 2, 2, 3, 0])
```

`arraytools.inverseUniqueIndex` (*index*)

Inverse an index.

Given a 1-D integer array with *unique* non-negative values, and *max* being the highest value in it, this function returns the position in the array of the values *0..max*. Values not occurring in input index get a value -1 in the inverse index.

Parameters *index* (1-dim int *array_like*) – Array with non-negative values, which all have to be unique. It's highest value is *max = index.max()*.

Returns *1-dim int array* – Array with length *max+1*, with the position in *index* of each of the values *0..max*, or -1 if the value does not occur in *index*.

Note: This is a low level function that does not check whether the input has indeed all unique values.

The inverse index translates the unique index numbers in a sequential index, so that `inverseUniqueIndex(index)[index] == arange(1+index.max())`.

Examples

```
>>> inverseUniqueIndex([0,5,2,6])
array([ 0, -1,  2, -1, -1,  1,  3])
>>> inverseUniqueIndex([0,5,2,6])[0,5,2,6]
array([0, 1, 2, 3])
```

`arraytools.cumsum0(a)`

Cumulative sum of a list of numbers prepended with a 0.

Parameters *a* (*array_like*, int) – List of integers to compute the cumulative sum for.

Returns *array*, *int* – Array with `len(a)+1` integers holding the cumulative sum of the integers from *a* with a 0 prepended.

Examples

```
>>> cumsum0([2,4,3])
array([0, 2, 6, 9])
```

A common use case is when concatenating some blocks of different length. If the list *a* holds the length of each block, the `cumsum0(a)` holds the start and end of each block in the concatenation.

```
>>> L = [ [0,1], [2,3,4,5], [6], [7,8,9] ]
>>> n = cumsum0([len(i) for i in L])
>>> print(n)
[ 0  2  6  7 10]
>>> C = concatenate(L)
>>> print(C)
[0 1 2 3 4 5 6 7 8 9]
>>> for i,j in zip(n[:-1],n[1:]):
...     print("%s:%s = %s" % (i,j,C[i:j]))
...
0:2 = [0 1]
2:6 = [2 3 4 5]
6:7 = [6]
7:10 = [7 8 9]
```

`arraytools.multiplicity(a)`

Return the multiplicity of the numbers in an array.

Parameters *a* (*array_like*, 1-dim) – The data array, will be flattened if it is not 1-dim.

Returns

- **mult** (*1-dim int array*) – The multiplicity of the unique values in *a*
- **uniq** (*1-dim array*) – Array of same type as *a*, with the sorted list of unique values in *a*.

Examples

```
>>> multiplicity([0,1,4,3,1,4,3,4,3,3])
(array([1, 2, 4, 3]), array([0, 1, 3, 4]))
>>> multiplicity([[1.0, 0.0, 0.5],[0.2,0.5,1.0]])
(array([1, 1, 2, 2]), array([ 0. ,  0.2,  0.5,  1. ]))
```

`arraytools.subsets(a)`

Split an array of integers into subsets.

The subsets of an integer array are sets of elements with the same value.

Parameters **a** (int *array_like*, 1-dim) – Array with integer values to be split in subsets

Returns

- **val** (*array of ints*) – The unique values in **a**, sorted in increasing order.
- **ind** (*varray.Varray*) – The *Varray* has the same number of rows as the number of values in **ind**. Each row contains the indices in **a** of the elements with the corresponding value in **val**.

Examples

```
>>> A = [0,1,4,3,1,4,3,4,3,3]
>>> val,ind = subsets(A)
>>> print(val)
[0 1 3 4]
>>> print(ind)
Varray (4,4)
 [0]
 [1 4]
 [3 6 8 9]
 [2 5 7]
<BLANKLINE>
```

The inverse of **ind** can be used to restore **A** from **val**.

```
>>> inv = inverseIndex(ind).reshape(-1)
>>> print(inv)
[0 1 3 2 1 3 2 3 2 2]
>>> (val[inv] == A).all()
True
```

`arraytools.sortSubsets(a, w=None)`

Sort subsets of an integer array **a**.

Subsets of an array are the sets of elements with equal values. By default the subsets are sorted according to decreasing number of elements in the set, or if a weight for each element is provided, according to decreasing sum of weights in the set.

Parameters

- **a** (*1-dim int :term:*) – Input array containing non-negative integer sets to be sorted.
- **w** (*1-dim int or float array_like*, optional) – If provided, it should have the same length as **a**. Each element of **a** will be attributed the corresponding weight. Specifying no weight is equivalent to giving all elements the same weight.

Returns *int array* – Array with same size as **a**, specifying for each element of **a** the index of its subset in the sorted list of subsets.

Examples

```
>>> sortSubsets([0,1,3,2,1,3,2,3,2,2])
array([3, 2, 1, 0, 2, 1, 0, 1, 0, 0])
>>> sortSubsets([0,1,4,3,1,4,3,4,3,3])
array([3, 2, 1, 0, 2, 1, 0, 1, 0, 0])
>>> sortSubsets([0,1,4,3,1,4,3,4,3,3],w=[9,8,7,6,5,4,3,2,1,0])
array([3, 1, 0, 2, 1, 0, 2, 0, 2, 2])
```

`arraytools.collectOnLength` (*items*, *return_index=False*)

Separate items in a list according to their length.

The length of all items in the list are determined and the items are put in separate lists according to their length.

Parameters

- **items** (*list*) – A list of any items that can be accepted as parameter of the `len()` function.
- **return_index** (*bool*) – If `True`, also return an index with the positions of the equal length items in the original iterable.

Returns

- **col** (*dict*) – A dict whose keys are the item lengths and values are lists of items with this length.
- **ind** (*dict, optional*) – A dict with the same keys as `col`, and the values being a list of indices in the list where the corresponding item of `col` appeared.

Examples

```
>>> collectOnLength(['a','bc','defg','hi','j','kl'])
{1: ['a', 'j'], 2: ['bc', 'hi', 'kl'], 4: ['defg']}
>>> collectOnLength(['a','bc','defg','hi','j','kl'],return_index=True) [1]
{1: [0, 4], 2: [1, 3, 5], 4: [2]}
```

`arraytools.complement` (*index*, *n=-1*)

Return the complement of an index in a range(0,n).

The complement is the list of numbers from the range(0,n) that are not included in the index.

Parameters

- **index** (1-dim *array_like*, of type `int` or `bool`.) – If integer, the array contains non-negative numbers in the range(0,n) and the return value will be the numbers in range(0,n) not included in index. If boolean, `False` value flag elements to be included (having a value `True`) in the output.
- **n** (*int*) – Upper limit for the range of numbers. If *index* is of type integer and *n* is not specified or is negative, it will be set equal to the largest number in *index* plus 1. If *index* is of type boolean and *n* is larger than the length of *index*, *index* will be padded with `False` values until length *n*.

Returns *1-dim array, type int or bool.* – The output array has the same dtype as the input. If *index* is integer: it is an array with the numbers from range(0,n) that are not included in *index*. If *index* is boolean, it is the negated input, padded to or cut at length *n*.

Examples

```
>>> print (complement ([0, 5, 2, 6]))
[1 3 4]
>>> print (complement ([0, 5, 2, 6], 10))
[1 3 4 7 8 9]
>>> print (complement ([False, True, True, True], 6))
[ True False False False  True  True]
```

`arraytools.sortByColumns(a)`

Sort an array on all its columns, from left to right.

The rows of a 2-dimensional array are sorted, first on the first column, then on the second to resolve ties, etc..

Parameters *a* (*array_like*, 2-dim) – The array to be sorted

Returns *int array*, 1-dim – Index specifying the order in which the rows have to be taken to obtain an array sorted by columns.

Examples

```
>>> sortByColumns ([[1, 2], [2, 3], [3, 2], [1, 3], [2, 3]])
array([0, 3, 1, 4, 2])
```

`arraytools.minroll(a)`

Roll a 1-D array to get the lowest values in front

If the lowest value occurs more than once, the one with the lowest next value is chosen, etcetera.

Parameters *a* (*array*, 1-dim) – The array to roll

Returns *m* (*int*) – The index of the element that should be put in front. This means that the `roll(a, -m)` gives the rolled array with the lowest elements in front.

Examples

```
>>> minroll([1, 3, 5, 1, 2, 6])
3
>>> minroll([0, 0, 2, 0, 0, 1])
3
>>> minroll([0, 0, 0, 0, 0, 0])
0
```

`arraytools.isroll(a, b)`

Check that two 1-dim arrays can be rolled into each other

Parameters

- *a* (*array*, 1-dim) – The first array
- *b* (*array*, 1-dim) – The second array, same length and dtype as *a* to be non-trivial.

Returns *m* (*int*) – The number of positions (non-negative) that *b* has to be rolled to be equal to *a*, or -2 if the two arrays have a different length, or -1 if their elements are not the same or not in the same order.

Examples

```
>>> isroll(array([1,2,3,4]),array([2,3,4,1]))
1
>>> isroll(array([1,2,3,4]),array([2,3,1,4]))
-1
>>> isroll(array([1,2,3,4]),array([3,2,1,4]))
-1
>>> isroll(array([1,2,3,4]),array([1,2,3]))
-2
```

`arraytools.findEqualRows` (*a*, *permutations*='none')

Find equal rows in a 2-dim array.

Parameters

- **a** (*array_like*, 2-dim) – The array in which to find the equal rows.
- **permutations** (*str*) – Defines which permutations of the row data are allowed while still considering the rows equal. Possible values are:
 - 'none': no permutations are allowed: rows must match the same data at the same positions. This is the default;
 - 'roll': rolling is allowed. Rows that can be transformed into each other by rolling are considered equal;
 - 'all': any permutation of the same data will be considered an equal row.

Returns

- **ind** (*1-dim int array*) – A row index sorting the rows in such order that equal rows are grouped together.
- **ok** (*1-dim bool array*) – An array flagging the rows in the order of `index` with True if it is the first row of a group of equal rows, or with False if the row is equal to the previous.

Notes

This function provides the functionality for detecting equal rows, but is seldomly used directly. There are wrapper functions providing more practical return values. See below.

See also:

`equalRows()` return the indices of the grouped equal rows

`uniqueRows()` return the indices of the unique rows

`uniqueRowsIndex()` like `uniqueRows`, but also returns index for all rows

Examples

```
>>> print(*findEqualRows([[1,2],[2,3],[3,2],[1,3],[2,3]]))
[0 3 1 4 2] [ True True True False True]
>>> print(*findEqualRows([[1,2],[2,3],[3,2],[1,3],[2,3]],permutations='all'))
[0 3 1 2 4] [ True True True False False]
>>> print(*findEqualRows([[1,2,3],[3,2,1],[2,3,1],[1,2,3]]))
[0 3 2 1] [ True False True True]
```

(continues on next page)

(continued from previous page)

```
>>> print(*findEqualRows([[1,2,3],[3,2,1],[2,3,1],[1,2,3]],permutations='all'))
[0 1 2 3] [ True False False False]
>>> print(*findEqualRows([[1,2,3],[3,2,1],[2,3,1],[1,2,3]],permutations='roll'))
[0 2 3 1] [ True False False  True]
```

`arraytools.equalRows(a, permutations='none')`

Return equal rows in a 2-dim array.

Parameters: see `findEqualRows()`

Returns `V(varray.Varray)` – A `Varray` where each row contains a list of the row numbers from `a` that are considered equal. The entries in each row are sorted and the rows are sorted according to their first element.

Notes

The return `Varray` holds a lot of information:

- `V.col(0)` gives the indices of the unique rows.
- `complement(V.col(0), len(a))` gives the indices of duplicate rows.
- `V.col(0)[V.lengths==1]` gives the indices of rows without duplicate.
- `Va.inverse(expand=True).reshape(-1)` returns an index into the unique rows for each of the rows of `a`.

See also:

`findEqualRows()` sorts and detects equal rows

`uniqueRows()` return the indices of the unique rows

`uniqueRowsIndex()` like `uniqueRows`, but also returns index for all rows

Examples

```
>>> equalRows([[1,2],[2,3],[3,2],[1,3],[2,3]])
Varray([[0], [1, 4], [2], [3]])
>>> equalRows([[1,2],[2,3],[3,2],[1,3],[2,3]],permutations='all')
Varray([[0], [1, 2, 4], [3]])
>>> equalRows([[1,2,3],[3,2,1],[2,3,1],[1,2,3]])
Varray([[0, 3], [1], [2]])
>>> equalRows([[1,2,3],[3,2,1],[2,3,1],[1,2,3]],permutations='all')
Varray([[0, 1, 2, 3]])
>>> equalRows([[1,2,3],[3,2,1],[2,3,1],[1,2,3]],permutations='roll')
Varray([[0, 2, 3], [1]])
```

`arraytools.uniqueRows(a, permutations='none')`

Find the unique rows of a 2-D array.

Parameters: see `findEqualRows()`

Returns `uniq (1-dim int array)` – Contains the indices of the unique rows in `a`.

See also:

`equalRows()` return the indices of the grouped equal rows

`uniqueRowsIndex()` like `uniqueRows`, but also returns index for all rows

Examples

```
>>> uniqueRows ([[1,2], [2,3], [3,2], [1,3], [2,3]])
array([0, 1, 2, 3])
>>> uniqueRows ([[1,2], [2,3], [3,2], [1,3], [2,3]], permutations='all')
array([0, 1, 3])
>>> uniqueRows ([[1,2,3], [3,2,1], [2,3,1], [1,2,3]])
array([0, 1, 2])
>>> uniqueRows ([[1,2,3], [3,2,1], [2,3,1], [1,2,3]], permutations='all')
array([0])
>>> uniqueRows ([[1,2,3], [3,2,1], [2,3,1], [1,2,3]], permutations='roll')
array([0, 1])
>>> uniqueRows ([[1,2,3], [3,2,1], [2,3,1], [1,2,3]])
array([0, 1, 2])
```

`arraytools.uniqueRowsIndex(a, permutations='none')`

Return the unique rows of a 2-D array and an index for all rows.

Parameters

- **a** (*array_like*, 2-dim) – Array from which to find the unique rows.
- **permutations** (*bool*) – If True, rows which are permutations of the same data are considered equal. The default is to consider permutations as different.
- **roll** (*bool*) – If True, rows which can be rolled into the same contents are considered equal.

Returns

- **uniq** (*1-dim int array*) – Contains the indices of the unique rows in *a*. The order of the elements in *uniq* is determined by the sorting procedure: in the current implementation this is `sortByColumns()`. If `permutations==True`, *a* is sorted along its last axis -1 before calling this sorting function. If `roll=True`, the rows of *a* are rolled to put the lowest values at the front.
- **ind** (*1-dim int array*) – For each row of *a*, holds the index in *uniq* where the row with the same data is found.

See also:

`equalRows()` return the indices of the grouped equal rows

`uniqueRows()` return the indices of the unique rows

Examples

```
>>> print(*uniqueRowsIndex([[1,2], [2,3], [3,2], [1,3], [2,3]]))
[0 1 2 3] [0 1 2 3 1]
>>> print(*uniqueRowsIndex([[1,2], [2,3], [3,2], [1,3], [2,3]], permutations='all'))
[0 1 3] [0 1 1 2 1]
>>> print(*uniqueRowsIndex([[1,2,3], [3,2,1], [2,3,1], [1,2,3]]))
[0 1 2] [0 1 2 0]
>>> print(*uniqueRowsIndex([[1,2,3], [3,2,1], [2,3,1], [1,2,3]], permutations='all'))
[0] [0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
>>> print(*uniqueRowsIndex([[1,2,3],[3,2,1],[2,3,1],[1,2,3]],permutations='roll'))
[0 1] [0 1 0 0]
>>> print(*uniqueRowsIndex([[1,2,3],[3,2,1],[2,3,1],[1,2,3]]))
[0 1 2] [0 1 2 0]
```

`arraytools.argNearestValue` (*values*, *target*)

Return the index of the item nearest to target.

Find in a list of floats the position of the value nearest to the target value.

Parameters

- **values** (*list*) – List of float values.
- **target** (*float*) – Target value to look up in list.

Returns *int* – The index in *values* of the float value that is closest to *target*.

See also:

`nearestValue()`

Examples

```
>>> argNearestValue([0.1,0.5,0.9],0.7)
1
```

`arraytools.nearestValue` (*values*, *target*)

Return the float nearest to target.

Find in a list of floats the value that is closest to the target value.

Parameters

- **values** (*list*) – List of float values.
- **target** (*float*) – Target value to look up in list.

Returns *float* – The value from the list that is closest to *target*.

See also:

`argNearestValue()`

Examples

```
>>> nearestValue([0.1,0.5,0.9],0.7)
0.5
```

`arraytools.inverseIndex` (*a*, *sort=True*, *expand=True*)

Create the inverse of a 2D index array.

A 2D index array is a 2D integer array where only the nonnegative values are relevant. Negative values are flagging a non-existent element. This allows for rows with different number of entries. While in most practical cases all (non-negative) values in a row are unique, this is not a requirement.

Note: This function is a wrapper around `varray.inverseIndex()` with other default values for the optional arguments. See there for parameters and return values.

Examples

```
>>> A = array([[0,1],[0,2],[1,2],[0,3]])
>>> print(A)
[[0 1]
 [0 2]
 [1 2]
 [0 3]]
>>> inv = inverseIndex(A)
>>> print(inv)
[[ 0  1  3]
 [-1  0  2]
 [-1  1  2]
 [-1 -1  3]]
```

We can use `varray.Varray` to get rid of the -1 entries:

```
>>> inv1 = inverseIndex([[0,1],[0,2],[1,2],[0,3]],expand=False)
>>> print(inv1)
Varray (4,3)
 [0 1 3]
 [0 2]
 [1 2]
 [3]
<BLANKLINE>
```

In both cases, the inverse of the inverse returns the original:

```
>>> (inverseIndex(inv) == A).all()
True
>>> (inverseIndex(inv1) == A).all()
True
```

`arraytools.findFirst(target, values)`

Find first position of values in target.

Find the first position in the array *target* of all the elements in the array *values*.

Parameters

- **target** (*1-dim int array*) – Integer array with all non-negative values. If not 1-dim, it will be flattened.
- **values** (*1-dim int array*) – Array with values to look up in target. If not 1-dim, it will be flattened.

Returns *int array* – Array with same size as *values*. For each element in *values*, the return array contains the position of that value in the flattened *target*, or -1 if that number does not occur in *target*. If an element from *values* occurs more than once in *target*, it is currently undefined which of those positions is returned.

Note: After `m = findIndex(target, values)` the equality `target[m] == values` holds for all the non-negative positions of *m*.

Examples

```
>>> A = array([1,3,4,5,7,3,8,9])
>>> B = array([0,7,1,3])
>>> ind = findFirst(A,B)
>>> print(ind)
[-1  4  0  1]
>>> (A[ind[ind>=0]] == B[ind>=0]).all()
True
```

`arraytools.findAll` (*target*, *values*)

Find all locations of values in target.

Find the position in the array *target* of all occurrences of the elements in the array *values*.

Parameters

- **target** (*1-dim int array*) – Integer array with all non-negative values. If not 1-dim, it will be flattened.
- **values** (*1-dim int array*) – Array with values to look up in target. If not 1-dim, it will be flattened.

Returns *list of int arrays*. – For each element in values, an array is returned with the indices in target of the elements with the same value.

See also:

`findFirst()`

Examples

```
>>> gid = array([ 2, 1, 1, 6, 6, 1 ])
>>> values = array([ 1, 2, 6 ])
>>> print(findAll(gid,values))
[array([1, 2, 5]), array([0]), array([3, 4])]
```

`arraytools.groupArgmin` (*val*, *gid*)

Compute the group minimum.

Computes the minimum value per group of a set of values tagged with a group number.

Parameters

- **val** (*1-dim array*) – Data values
- **gid** (*1-dim int :term:*) – Array with same length as val, containing the group identifiers.

Returns

- **ugid** (*1-dim int array*) – (ngrp,) shaped array with unique group identifiers.
- **minpos** (*1-dim int array*) – (ngrp,) shaped array giving the position in *val* of the minimum of all values with the corresponding group identifier in *ugid*. The minimum values corresponding to the groups in *ugid* can be obtained with `val[minpos]`.

Examples

```
>>> val = array([ 0.0, 1.0, 2.0, 3.0, 4.0, -5.0 ])
>>> gid = array([ 2, 1, 1, 6, 6, 1 ])
>>> print(groupArgmin(val,gid))
(array([1, 2, 6]), array([5, 0, 3]))
```

`arraytools.vectorPairAreaNormals` (*vec1*, *vec2*)

Compute area of and normals on parallelograms formed by *vec1* and *vec2*.

Parameters

- **vec1** ((3,) or (n,3) shaped float :term:) – Array with 1 or n vectors in 3D space.
- **vec2** ((3,) or (n,3) shaped float :term:.) – Array with 1 or n vectors in 3D space.

Returns

- **area** ((n,) shaped float array) – The area of the parallelograms formed by the vectors *vec1* and *vec2*.
- **normal** ((n,3) shaped float array) – The unit length vectors normal to each vector pair (*vec1*,*vec2*).

Note: This first computes the cross product of *vec1* and *vec2*, which is a normal vector with length equal to the area. Then `normalize()` produces the required results.

Note that where two vectors are parallel, an area zero results and an axis with components NaN.

See also:

`vectorPairNormals()` only returns the normal vectors

`vectorPairArea()` only returns the area

Examples

```
>>> a = array([[3.,4,0],[1,0,0],[1,-2,1]])
>>> b = array([[1.,3.,0],[1,0,1],[-2,4,-2]])
>>> l,v = vectorPairAreaNormals(a,b)
>>> print(l)
[ 5.  1.  0.]
>>> print(v)
[[ 0.  0.  1.]
 [ 0. -1.  0.]
 [ nan nan nan]]
```

`arraytools.vectorPairNormals` (*vec1*, *vec2*)

Create unit vectors normal to *vec1* and *vec2*.

Parameters

- **vec1** ((3,) or (n,3) shaped float :term:) – Array with 1 or n vectors in 3D space.

- **vec2** ((3,) or (n,3) shaped float :term:.) – Array with 1 or n vectors in 3D space.

Returns normal ((n,3) shaped float array) – The unit length vectors normal to each vector pair (vec1,vec2).

See also:

vectorPairAreaNormals() returns the normals and the area between vectors

vectorPairArea() only returns the area between vectors

Examples

```
>>> a = array([[3., 4, 0], [1, 0, 0], [1, -2, 1]])
>>> b = array([[1., 3., 0], [1, 0, 1], [-2, 4, -2]])
>>> v = vectorPairNormals(a,b)
>>> print(v)
[[ 0.  0.  1.]
 [ 0. -1.  0.]
 [ nan nan nan]]
```

arraytools.**vectorPairArea** (vec1, vec2)

Compute area of the parallelogram formed by a vector pair vec1,vec2.

Parameters

- **vec1** ((3,) or (n,3) shaped float :term:.) – Array with 1 or n vectors in 3D space.
- **vec2** ((3,) or (n,3) shaped float :term:.) – Array with 1 or n vectors in 3D space.

Returns area ((n,) shaped float array) – The area of the parallelograms formed by the vectors vec1 and vec2.

See also:

vectorPairAreaNormals() returns the normals and the area between vectors

vectorPairNormals() only returns the normal vectors

Examples

```
>>> a = array([[3., 4, 0], [1, 0, 0], [1, -2, 1]])
>>> b = array([[1., 3., 0], [1, 0, 1], [-2, 4, -2]])
>>> l = vectorPairArea(a,b)
>>> print(l)
[ 5.  1.  0.]
```

arraytools.**vectorTripleProduct** (vec1, vec2, vec3)

Compute triple product vec1 . (vec2 x vec3).

Parameters vec2, vec3 ((vec1,) – Three arrays with same shape holding 1 or n vectors in 3D space.

Returns (n,) shaped float array – The triple product of each set of corresponding vectors from vec1, vec2, vec3.

Note: The triple product is the dot product of the first vector(s) and the normal(s) on the second and third vector(s). This is also twice the volume of the parallelepiped formed by the 3 vectors.

If `vec1` has a unit length, the result is also the area of the parallelogram (`vec2,vec3`) projected in the direction `vec1`.

This is functionally equivalent with `dotpr(vec1, cross(vec2, vec3))` but is implemented in a more efficient way, using the determinant formula.

Examples

```
>>> vectorTripleProduct([[1.,0.,0.],[2.,0.,0.]],
...                      [[1.,1.,0.],[2.,2.,0.]],
...                      [[1.,1.,1.],[2.,2.,2.]])
array([ 1., 8.])
```

`arraytools.vectorPairCosAngle` (`v1`, `v2`)

Return the cosinus of the angle between the vectors `v1` and `v2`.

`vec1` and `vec2` are (n,3) shaped arrays holding collections of vectors. The result is an (n) shaped array with the cosinus of the angle between each pair of vectors (`vec1,vec2`).

`arraytools.vectorPairAngle` (`v1`, `v2`)

Return the angle (in radians) between the vectors `v1` and `v2`.

`vec1` and `vec2` are (n,3) shaped arrays holding collections of vectors. The result is an (n) shaped array with the angle between each pair of vectors (`vec1,vec2`).

```
>>> vectorPairAngle([1,0,0],[0,1,0]) / DEG
90.0
>>> vectorPairAngle([[1,0,0],[0,1,0]],[[1,1,0],[1,1,1]]) / DEG
array([ 45. , 54.74])
```

`arraytools.det2` (`a`)

Compute the determinant of 2x2 matrices.

Parameters `a` (*int or float :term:(..,2,2)*) – Array containing one or more (2,2) square matrices.

Returns *int or float number or array(...)* – The determinant(s) of the matrices. The result has the same type as the input array.

Note: This method is faster than the generic `numpy.linalg.det`.

See also:

`det3()` determinant of (3,3) matrices

`det4()` determinant of (4,4) matrices

`numpy.linalg.det()` determinant of any size matrix

Examples

```
>>> det2([[1, 2], [2, 1]])
-3
>>> det2([[1, 2], [2, 1]], [[4, 2], [1, 3]])
array([-3, 10])
```

`arraytools.det3(a)`

Compute the determinant of 3x3 matrices.

Parameters *a* (*int* or *float* :*term*:(..., 3, 3)) – Array containing one or more (3,3) square matrices.

Returns *int* or *float* number or *array(...)* – The determinant(s) of the matrices. The result has the same type as the input array.

Note: This method is faster than the generic `numpy.linalg.det`.

See also:

`det2()` determinant of (2,2) matrices

`det4()` determinant of (4,4) matrices

`numpy.linalg.det()` determinant of any size matrix

Examples

```
>>> det3([[1, 2, 3], [2, 2, 2], [3, 2, 1]])
0
>>> det3([[1., 0., 0.], [1., 1., 0.], [1., 1., 1.]],
...       [[2., 0., 0.], [2., 2., 0.], [2., 2., 2.]])
array([ 1.,  8.])
```

`arraytools.det4(a)`

Compute the determinant of 4x4 matrices.

Parameters *a* (*int* or *float* :*term*:(..., 4, 4)) – Array containing one or more (4,4) square matrices.

Returns *int* or *float* number or *array(...)* – The determinant(s) of the matrices. The result has the same type as the input array.

Note: This method is faster than the generic `numpy.linalg.det`.

See also:

`det2()` determinant of (2,2) matrices

`det3()` determinant of (3,3) matrices

`numpy.linalg.det()` determinant of any size matrix

Examples

```
>>> det4([[ [1., 0., 0., 0.], [1., 1., 0., 0.], [1., 1., 1., 0.], [1., 1., 1., 1.] ],
...       [[2., 0., 0., 0.], [2., 2., 0., 0.], [2., 2., 2., 0.], [2., 2., 2., 2.] ]])
array([ 1., 16.] )
```

`arraytools.percentile` (*values*, *perc*=[25.0, 50.0, 75.0], *wts*=None)

Return percentiles of a set of values.

A percentiles is the value such that at least a given percent of the values is lower or equal than the value.

Parameters

- **values** (*1-dim int or float :term:*) – The set of values for which to compute the percentiles.
- **perc** (*1-dim int or float :term:*) – One or multiple percentile values to compute. All values should be in the range [0,100]. By default, the quartiles are computed.
- **wts** (*1-dim array*) – Array with same shape as values and all positive values. These are weights to be assigned to the values.

Returns *1-dim float array* – Array with the percentile value(s) that is/are greater or equal than *perc* percent of *values*. If the result lies between two items of *values*, it is obtained by interpolation.

Examples

```
>>> percentile(arange(100), [10, 50, 90])
array([ 9., 49., 89.])
>>> percentile([1, 1, 1, 1, 1, 2, 2, 2, 3, 5])
array([ 1., 1., 2.])
```

`arraytools.histogram2` (*a*, *bins*, *range*=None)

Compute the histogram of a set of data.

This is similar to the numpy histogram function, but also returns the bin index for each individual entry in the data set.

Parameters

- **a** (*array_like.*) – Input data. The histogram is computed over the flattened array.
- **bins** (*int or sequence of scalars.*) – If *bins* is an int, it defines the number of equal-width bins in the given range (nbins). If *bins* is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths. The number of bins (nbins) is then equal to $\text{len}(\text{bins}) - 1$. A value *v* will be sorted in bin *i* if $\text{bins}[i] \leq v < \text{bins}[i+1]$, except for the last bin, which will also contain the values equal to the right bin edge.
- **range`** (*(float, float), optional.*) – The lower and upper range of the bins. If not provided, range is simply (a.min(), a.max()). Values outside the range are ignored. This parameter is ignored if bins is a sequence.

Returns

- **hist** (*int array*) – The number of elements from *a* sorted in each of the bins.
- **ind** (list of nbins int arrays) – Each array holds the indices the elements sorted in the corresponding bin.
- **bin_edges** (*float array*) – The array contains the $\text{len}(\text{hist}) + 1$ bin edges.

Example

```
>>> hist,ind,bins = histogram2([1,2,3,4,2,3,1],[1,2,3,4,5])
>>> print(hist)
[2 2 2 1]
>>> for i in ind: print(i)
[0 6]
[1 4]
[2 5]
[3]
>>> print(bins)
[1 2 3 4 5]
>>> hist,bins = histogram([1,2,3,4,2,3,1],5)
>>> print(hist)
[2 2 0 2 1]
>>> hist,ind,bins = histogram2([1,2,3,4,2,3,1],5)
>>> print(hist)
[2 2 0 2 1]
>>> for i in ind: print(i)
[0 6]
[1 4]
[]
[2 5]
[3]
```

`arraytools.movingView(a, size)`

Create a moving view along the first axis of an array.

A moving view of an array is a view stacking a sequence of subarrays with fixed size along the 0 axis of the array, where each next subarray shifts one position down along the 0 axis.

Parameters

- **a** (*array_like*) – Array for which to create a moving view.
- **size** (*int*) – Size of the moving view: this is the number of rows to include in the subarray.

Returns view of the array *a* – The view of the original array has an extra first axis with length $1 + a.shape[0] - size$, a second axis with length *size*, and the remaining axes have the same length as those in *a*.

Note: While this function limits the moving view to the direction of the 0 axis, using `swapaxes(0,axis)` allows to create moving views over any axis.

See also:

[`movingAverage\(\)`](#) compute moving average values along axis 0

Examples

```
>>> x=arange(10).reshape((5,2))
>>> print(x)
[[0 1]
 [2 3]
 [4 5]
 [6 7]
```

(continues on next page)

(continued from previous page)

```
[8 9]]
>>> print(movingView(x, 3))
[[[0 1]
   [2 3]
   [4 5]]
<BLANKLINE>
 [[2 3]
  [4 5]
  [6 7]]
<BLANKLINE>
 [[4 5]
  [6 7]
  [8 9]]]
```

Calculate rolling sum of first axis:

```
>>> print(movingView(x, 3).sum(axis=0))
[[ 6  9]
 [12 15]
 [18 21]]
```

`arraytools.movingAverage` (*a*, *n*, *m0=None*, *m1=None*)

Compute the moving average along the first axis of an array.

Parameters

- **a** (*array_like*) – The array to be averaged.
- **n** (*int*) – Sample length along axis 0 over which to compute the average.
- **m0** (*int, optional*) – If provided, the first data row of *a* will be prepended to *a* this number of times.
- **m1** (*int, optional*) – If provided, the last data row of *a* will be appended to *a* this number of times.

Returns *float array* – Array containing the moving average over data sets of length *n* along the first axis of *a*. The array has a shape like *a* except for its first axis, which may have a different length. If neither *m0* nor *m1* are set, the first axis will have a length of $1 + a.shape[0] - n$. If both *m0* and *m1* are given, the first axis will have a length of $1 + a.shape[0] - n + m0 + m1$. If either *m0* or *m1* are set and the other not, the missing value *m0* or *m1* will be computed thus that the return array has a first axis with length *a.shape[0]*.

Examples

```
>>> x=arange(10).reshape((5,2))
>>> print(x)
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
>>> print(movingAverage(x,3))
[[ 2.  3.]
 [ 4.  5.]
 [ 6.  7.]]
```

(continues on next page)

(continued from previous page)

```
>>> print (movingAverage(x,3,2))
[[ 0.  1.  ]
 [ 0.67 1.67]
 [ 2.  3.  ]
 [ 4.  5.  ]
 [ 6.  7.  ]]
```

`arraytools.randomNoise` (*shape*, *min*=0.0, *max*=1.0)

Create an array with random float values between min and max

Parameters

- **shape** (*tuple of ints*) – Shape of the array to create.
- **min** (*float*) – Minimum value of the random numbers.
- **max** (*float*) – Maximum value of the random numbers.

Returns *float array* – An array of the requested shape filled with random numbers in the specified range.

Examples

```
>>> x = randomNoise((3,4))
>>> x.shape == (3,4)
True
>>> (x >= 0.0).all()
True
>>> (x <= 1.0).all()
True
```

`arraytools.stuur` (*x*, *xval*, *yval*, *exp*=2.5)

Returns a (non)linear response on the input *x*.

xval and *yval* should be lists of 3 values: [*xmin*, *x0*, *xmax*], [*ymin*, *y0*, *ymax*]. Together with the exponent *exp*, they define the response curve as function of *x*. With an exponent > 0, the variation will be slow in the neighbourhood of (*x0*,*y0*). For values *x* < *xmin* or *x* > *xmax*, the limit value *ymin* or *ymax* is returned.

Examples

```
>>> x = unitDivisor(4)
>>> x
array([ 0.  ,  0.25,  0.5  ,  0.75,  1.  ])
>>> array([stuur(xi, (0.,0.5,1.0), (0.,0.5,1.0) ) for xi in x])
array([ 0.  ,  0.41,  0.5  ,  0.59,  1.  ])
```

`arraytools.unitDivisor` (*div*)

Divide a unit interval in equal parts.

This function is intended to be used by interpolation functions that accept an input as either an int or a list of floats.

Parameters **div** (*int*, or *list of floats in the range [0.0, 1.0]*) – If it is an integer, it specifies the number of equal sized parts in which the interval [0.0, 1.0] is to be divided. If a list of floats, its values should be monotonously increasing from 0.0 to 1.0. The values are returned unchanged.

Returns

- *1-dim float array* – The float values that border the parts of the interval. If *div* is a an integer, returns the floating point values
- dividing the unit interval in *div* equal parts. If *div* is a list,
- just returns *div* as a 1D array.

Examples

```
>>> unitDivisor(4)
array([ 0. , 0.25, 0.5 , 0.75, 1.  ])
>>> unitDivisor([0., 0.3, 0.7, 1.0])
array([ 0. , 0.3, 0.7, 1.  ])
```

`arraytools.uniformParamValues` (*n*, *umin*=0.0, *umax*=1.0)

Create a set of uniformly distributed parameter values in a range.

Parameters

- **n** (*int*) – Number of intervals in which the range should be divided. The number of values returned is *n*+1.
- **umin** (*float*) – Starting value of the interval.
- **umax** (*float*) – Ending value of the interval.

Returns *1-dim float array* – The array contains *n*+1 equidistant values in the range [*umin*, *umax*]. For *n* > 0, both of the endpoints are included. For *n*=0, a single value at the center of the interval will be returned. For *n*<0, an empty array is returned.

Examples

```
>>> uniformParamValues(4).tolist()
[0.0, 0.25, 0.5, 0.75, 1.0]
>>> uniformParamValues(0).tolist()
[0.5]
>>> uniformParamValues(-1).tolist()
[]
>>> uniformParamValues(2,1.5,2.5).tolist()
[1.5, 2.0, 2.5]
```

`arraytools.unitAttractor` (*x*, *e0*=0.0, *e1*=0.0)

Moves values in the range 0..1 closer to or away from the limits.

Parameters

- **x** (*float :term:*) – Values in the range 0.0 to 1.0, to be pulled to/pushed from ends.
- **e0** (*float*) – Attractor force to the start of the interval (0.0). A negative value will push the values away from this point.
- **e1** (*float*) – Attractor force to the end of the interval (1.0). A negative value will push the values away from this point.

Note: This function is usually called from the `seed()` function, passing an initially uniformly distributed set of points.

Examples

```
>>> set_printoptions(precision=4)
>>> print(unitAttractor([0.,0.25,0.5,0.75,1.0], 2.))
[ 0.      0.0039 0.0625 0.3164 1.      ]
>>> set_printoptions(precision=2)
>>> unitAttractor([0.,0.25,0.5,0.75,1.0])
array([ 0. ,  0.25,  0.5 ,  0.75,  1.  ])
```

`arraytools.seed(n, e0=0.0, e1=0.0)`

Create a list of seed values.

A seed list is a list of float values in the range 0.0 to 1.0. It can be used to subdivide a line segment or to seed nodes along lines for meshing purposes.

This function divides the unit interval in n parts, resulting in $n+1$ seed values. While the intervals are by default of equal length, the $e0$ and $e1$ can be used to create unevenly spaced seed values.

Parameters

- **n** (*int*) – Positive integer: the number of elements (yielding $n+1$ parameter values).
- **e0** (*float*) – Attractor force at the start of the interval. A value larger than zero will attract the points closer to 0.0, while a negative value will repulse them.
- **e1** (*float*) – Attractor force at the end of the interval. A value larger than zero will attract the points closer to 1.0, while a negative value will repulse them.

Returns

- float arraya list of $n+1$ float values in the range 0.0 to 1.0.
- *The values are in ascending order, starting with 0.0 and ending with 1.0.*

See also:

`seed1()` attractor at one end and equidistant points at the other.

`smartSeed()` similar function accepting a variety of input.

Examples

```
>>> set_printoptions(precision=4)
>>> print(seed(5,2.,2.))
[ 0.      0.0639 0.3362 0.6638 0.9361 1.      ]
>>> set_printoptions(precision=2)
>>> for e0 in [ 0., 0.1, 0.2, 0.5, 1.0]: print(seed(5,e0))
[ 0.    0.2  0.4  0.6  0.8  1.  ]
[ 0.    0.18 0.37 0.58 0.79 1.  ]
[ 0.    0.16 0.35 0.56 0.77 1.  ]
[ 0.    0.1  0.27 0.49 0.73 1.  ]
[ 0.    0.04 0.16 0.36 0.64 1.  ]
```

`arraytools.seed1(n, nuni=0, e0=0.0)`

Create a list of seed values.

A seed list is a list of float values in the range 0.0 to 1.0. It can be used to subdivide a line segment or to seed nodes along lines for meshing purposes.

This function divides the unit interval in n parts, resulting in $n+1$ seed values. While the intervals are by default of equal length, the *nuni* and *e0* can be used to create unevenly spaced seed values.

Parameters:

- *n*: positive integer: the number of elements (yielding $n+1$ parameter values).
- *nuni*: 0..n-1: number of intervals at the end of the range that will have equal length. If $n < 2$, this function is equivalent with `seed(n,e0,0.0)`.
- *e0*: float: attractor for the start of the range. A value larger than zero will attract the points closer to the startpoint, while a negative value will repulse them.

Returns a list of $n+1$ float values in the range 0.0 to 1.0. The values are in ascending order, starting with 0.0 and ending with 1.0.

See also :func: `seed` for an analogue function with attractors at both ends of the range.

Example:

```
>>> set_printoptions(precision=4)
>>> S = seed1(5,0,1.)
>>> print(S)
[ 0.  0.04 0.16 0.36 0.64 1.  ]
>>> print(S[1:]-S[:-1])
[ 0.04 0.12 0.2  0.28 0.36]
>>> S = seed1(5,2,1.)
>>> print(S)
[ 0.  0.0435 0.1739 0.3913 0.6957 1.  ]
>>> print(S[1:]-S[:-1])
[ 0.0435 0.1304 0.2174 0.3043 0.3043]
>>> set_printoptions(precision=2)
```

`arraytools.smartSeed(n)`

Create a list of seed values.

Like the `seed()` function, this function creates a list of float values in the range 0.0 to 1.0. It accepts however a variety of inputs, making it the preferred choice when it is not known in advance how the user wants to control the seeds: automatically created or self specified.

Parameters *n* (int, tuple or float *seed*) – Action depends on the argument:

- if an int, returns `seed(n)`,
- if a tuple (n,), (n,e0) or (n,e0,e1): returns `seed(*n)`,
- if a float array-like, it is normally a sorted list of float values in the range 0.0 to 1.0: the values are returned unchanged in an array.

Returns *float array* – The values created depending on the input argument.

Examples

```
>>> set_printoptions(precision=4)
>>> print(smartSeed(5))
[ 0.  0.2 0.4 0.6 0.8 1.  ]
>>> print(smartSeed((5,2.,1.)))
[ 0.  0.01  0.1092 0.3701 0.7504 1.  ]
>>> print(smartSeed([0.0,0.2,0.3,0.4,0.8,1.0]))
[ 0.  0.2 0.3 0.4 0.8 1.  ]
>>> set_printoptions(precision=2)
```

`arraytools.gridpoints` (*seed0*, *seed1=None*, *seed2=None*)

Create weights for 1D, 2D or 3D element coordinates.

Parameters

- **seed0** (*int* or *list of floats*) – Subdivisions along the first parametric direction
- **seed1** (*int* or *list of floats*) – Subdivisions along the second parametric direction
- **seed2** (*int* or *list of floats*) – Subdivisions along the third parametric direction
- **these parameters are integer values the divisions will be equally** (*If*) –
- **between 0 and 1.** (*spaced*) –

Examples

```
>>> gridpoints(4)
array([ 0. ,  0.25,  0.5 ,  0.75,  1.  ])
>>> gridpoints(4,2)
array([[ 1. ,  0. ,  0. ,  0. ],
       [ 0.75,  0.25,  0. ,  0. ],
       [ 0.5 ,  0.5 ,  0. ,  0. ],
       [ 0.25,  0.75,  0. ,  0. ],
       [ 0. ,  1. ,  0. ,  0. ],
       [ 0.5 ,  0. ,  0. ,  0.5 ],
       [ 0.38,  0.12,  0.12,  0.38],
       [ 0.25,  0.25,  0.25,  0.25],
       [ 0.12,  0.38,  0.38,  0.12],
       [ 0. ,  0.5 ,  0.5 ,  0. ],
       [ 0. ,  0. ,  0. ,  1. ],
       [ 0. ,  0. ,  0.25,  0.75],
       [ 0. ,  0. ,  0.5 ,  0.5 ],
       [ 0. ,  0. ,  0.75,  0.25],
       [ 0. ,  0. ,  1. ,  0. ]])
```

`arraytools.nodalSum` (*val*, *elems*, *nnod=-1*)

Compute the nodal sum of values defined on element nodes.

Parameters

- **val** (*float array (nelems, nplex, nval)*) – Array with *nval* values at *nplex* nodes of *nelems* elements.
- **elems** (*int array (nelems, nplex)*) – The node indices of the elements.
- **nnod** (*int, optional*) – If provided, the length of the output arrays will be set to this value. It should be higher than the highest node number appearing in *elems*. The default will set it automatically to `elems.max() + 1`.
- **Returns** –
- **sum** (*float array (nnod, nval)*) – The sum of all the values at the same node.
- **cnt** (*int array (nnod)*) – The number of values summed at each node.

See also:

`nodalAvg()` compute the nodal average of values defined on element nodes

`arraytools.nodalAvg(val, elems, nnod=-1)`

Compute the nodal average of values defined on element nodes.

Parameters

- **val** (*float array (nelems, nplex, nval)*) – Array with *nval* values at *nplex* nodes of *nelems* elements.
- **elems** (*int array (nelems, nplex)*) – The node indices of the elements.
- **nnod** (*int, optional*) – If provided, the length of the output arrays will be set to this value. It should be higher than the highest node number appearing in *elems*. The default will set it automatically to `elems.max() + 1`.
- **Returns** –
- **avg** (*float array (nnod, nval)*) – The average of all the values at the same node.

See also:

`nodalSum()` compute the nodal sum of values defined on element nodes

`arraytools.fmtData1d(data, npl=8, sep=', ', linesep='\n', fmt=<class 'str'>)`

Format data in lines with maximum *npl* items.

Formats a list or array of data items in groups containing a maximum number of items. The data items are converted to strings using the *fmt* function, concatenated in groups of *npl* items using *sep* as a separator between them. Finally, the groups are concatenated with a *linesep* separator.

Parameters

- **data** (*list or array.*) – List or array with data. If an array, it will be flattened.
- **npl** (*int*) – Maximum number of items per group. Items will be concatenated groups of this number of items. The last group may contain less items.
- **sep** (*str*) – Separator to add between individual items in a group.
- **linesep** (*str*) – Separator to add between groups. The default (newline) will put each group of *npl* items on a separate line.
- **fmt** (*callable*) – Used to convert a single item to a string. Default is the Python built-in string converter.

Returns *str* – Multiline string with the formatted data.

Examples

```
>>> print (fmtData1d(arange(10)))
0, 1, 2, 3, 4, 5, 6, 7
8, 9
>>> print (fmtData1d([1.25, 3, 'no', 2.50, 4, 'yes'], npl=3))
1.25, 3, no
2.5, 4, yes
>>> myformat = lambda x: "%10s" % str(x)
>>> print (fmtData1d([1.25, 3, 'no', 2.50, 4, 'yes'], npl=3, fmt=myformat))
1.25,          3,          no
2.5,           4,          yes
```

6.1.4 `script` — Basic pyFormex script functions

The `script` module provides the basic functions available in all pyFormex scripts. These functions are available in GUI and NONGUI applications, without the need to explicitly importing the `script` module.

Functions defined in module `script`

`script.Globals()`

Return the globals that are passed to the scripts on execution.

When running pyformex with the `-nogui` option, this contains all the globals defined in the module `formex` (which include those from `coords`, `arraytools` and `numpy`).

When running with the GUI, this also includes the globals from `gui.draw` (including those from `gui.color`).

Furthermore, the global variable `__name__` will be set to either `'draw'` or `'script'` depending on whether the script was executed with the GUI or not.

`script.export(dic)`

Export the variables in the given dictionary.

`script.export2(names, values)`

Export a list of names and values.

`script.forget(names)`

Remove the global variables specified in list.

`script.forgetAll()`

Delete all the global variables.

`script.rename(oldnames, newnames)`

Rename the global variables in `oldnames` to `newnames`.

`script.listAll(clas=None, like=None, filtr=None, dic=None, sort=False)`

Return a list of all objects in dictionary that match criteria.

- *clas*: a class or list of classes: if specified, only instances of this/these class(es) will be returned
- *like*: a string: if given, only object names starting with this string will be returned
- *filtr*: a function taking an object name as parameter and returning True or False. If specified, only objects passing the test will be returned.
- *dic*: a dictionary object with strings as keys, defaults to `pyformex.PF`.
- *sort*: bool: if True, the returned list will be sorted.

The return value is a list of keys from *dic*.

`script.named(name)`

Returns the global object named *name*.

`script.getcfg(name, default=None)`

Return a value from the configuration or None if nonexistent.

`script.ask(question, choices=None, default="")`

Ask a question and present possible answers.

If no choices are presented, anything will be accepted. Else, the question is repeated until one of the choices is selected. If a default is given and the value entered is empty, the default is substituted. Case is not significant, but choices are presented unchanged. If no choices are presented, the string typed by the user is returned. Else the return value is the lowest matching index of the users answer in the choices list. Thus, `ask('Do you agree',['Y','n'])` will return 0 on either `'y'` or `'Y'` and 1 on either `'n'` or `'N'`.

`script.ack` (*question*)

Show a Yes/No question and return True/False depending on answer.

`script.error` (*message*)

Show an error message and wait for user acknowledgement.

`script.autoExport` (*g*)

Autoexport globals from script/app globals.

g: dict holding the globals dict from a script/app run environment.

This exports some objects from the script/app runtime globals to the pf.PF session globals directory. The default is to export all instances of class `Geometry`.

This can be customized in the script/app by setting the global variable `autoglobals`. If set to a value that evaluates to False, no autoexport will be done. If set to True, the default autoexport will be done: all instances of `geometry`. If set to a list of names, only the specified names will be exported. Furthermore, a global variable `autoclasses` may be set to a list of class names. All global instances of the specified classes will be exported.

Remember that the variables need to be globals in your script/app in order to be autoexported, and that auto-globals feature needs to be enabled in your configuration.

`script.playScript` (*scr, name=None, filename=None, argv=[], encoding=None*)

Play a pyformex script *scr*. *scr* should be a valid Python text.

There is a lock to prevent multiple scripts from being executed at the same time. This implies that pyFormex scripts can currently not be recurrent. If *name* is specified, set the global variable `pyformex.scriptName` to it when the script is started. If *filename* is specified, set the global variable `__file__` to it.

`script.breakpt` (*msg=None*)

Set a breakpoint where the script can be halted on a signal.

If an argument is specified, it will be written to the message board.

The exitrequested signal is usually emitted by pressing a button in the GUI.

`script.stopatbreakpt` ()

Set the exitrequested flag.

`script.convertPrintSyntax` (*filename*)

Convert a script to using the print function

`script.checkPrintSyntax` (*filename*)

Check whether the script in the given files uses print function syntax.

Returns the compiled object if no error was found during compiling. Returns the filename if an error was found and correction has been attempted. Raises an exception if an error is found and no correction attempted.

`script.runScript` (*fn, argv=[]*)

Play a formex script from file *fn*.

fn is the name of a file holding a pyFormex script. A list of arguments can be passed. They will be available under the name *argv*. This variable can be changed by the script and the resulting *argv* is returned to the caller.

`script.runApp` (*appname, argv=[], refresh=False, lock=True, check=True*)

Run a pyFormex application.

A pyFormex application is a Python module that can be loaded in pyFormex and that contains a function 'run()'. Running the application is equivalent to executing this function.

Parameters:

- *appname*: name of the module in Python dot notation. The module should live in a path included the the a file holding a pyFormex script.

- *argv*: list of arguments. This variable can be changed by the app and the resulting argv will be returned to the caller.

Returns the exit value of the run function. A zero value is supposed to mean a normal exit.

`script.runAny` (*appname=None, argv=[], step=False, refresh=False, remember=True*)
Run the current pyFormex application or script file.

Parameters:

- *appname*: either the name of a pyFormex application (app) or a file containing a pyFormex script. An app name is specified in Python module syntax (package.subpackage.module) and the path to the package should be in the configured app paths.

This function does nothing if no *appname*/filename is passed or no current script/app was set. If arguments are given, they are passed to the script. If *step* is True, the script is executed in step mode. The 'refresh' parameter will reload the app.

`script.exit` (*all=False*)
Exit from the current script or from pyformex if no script running.

`script.quit` ()
Quit the pyFormex program

This is a hard exit from pyFormex. It is normally not called directly, but results from an `exit(True)` call.

`script.processArgs` (*args*)
Run the application without gui.

Arguments are interpreted as names of script files, possibly interspersed with arguments for the scripts. Each running script should pop the required arguments from the list.

`script.setPrefs` (*res, save=False*)
Update the current settings (store) with the values in *res*.

res is a dictionary with configuration values. The current settings will be update with the values in *res*.

If *save* is True, the changes will be stored to the user's configuration file.

`script.chdir` (*path, create=False*)
Change the current working directory.

If *path* exists and it is a directory name, make it the current directory. If *path* exists and it is a file name, make the containing directory the current directory. If *path* does not exist and *create* is True, create the path and make it the current directory. If *create* is False, raise an Error.

Parameters:

- *path*: pathname of the directory or file. If it is a file, the name of the directory holding the file is used. The path can be an absolute or a relative pathname. A '~' character at the start of the pathname will be expanded to the user's home directory.
- *create*: bool. If True and the specified path does not exist, it will be created. The default is to do nothing if the specified path does not exist.

The changed to current directory is stored in the user's preferences for persistence between pyFormex invocations.

`script.pwdir` ()
Print the current working directory.

`script.mkdir` (*path, clear=False, new=False*)
Create a directory.

Create a directory, including any needed parent directories. Any part of the path may already exist.

- *path*: pathname of the directory to create, either an absolute or relative path. A '~' character at the start of the pathname will be expanded to the user's home directory.
- *clear*: bool. If True, and the directory already exists, its contents will be deleted.
- *new*: bool. If True, requires the directory to be a new one. An error will be raised if the path already exists.

The following table gives an overview of the actions for different combinations of the parameters:

clear	new	path does not exist	path exists
F	F	kept as is	newly created
T	F	emptied	newly created
T/F	T	raise	newly created

If succesful, returns the tilde-expanded path of the directory. Raises an exception in the following cases:

- the directory could not be created,
- *clear* is True, and the existing directory could not be cleared,
- *new* is False, *clear* is False, and the existing path is not a directory,
- *new* is True, and the path exists.

`script.mkpdir (path)`

Make sure the parent directory of path exists.

`script.runtime ()`

Return the time elapsed since start of execution of the script.

`script.startGui (args=[])`

Start the gui

`script.writeGeomFile (filename, objects, sep=' ', mode='w', shortlines=False, **kargs)`

Save geometric objects to a pyFormex Geometry File.

A pyFormex Geometry File can store multiple geometrical objects in a native format that can be efficiently read back into pyformex. The format is portable over different pyFormex versions and even to other software.

- *filename*: the name of the file to be written. If it ends with '.gz' or '.bz2', the file will be compressed with gzip or bzip2, respectively.
- *objects*: a list or a dictionary. If it is a dictionary, the objects will be saved with the key values as there names. Objects that can not be exported to a Geometry File will be silently ignored.
- *mode*: can be set to 'a' to append to an existing file.
- *sep*: the string used to separate data. If set to an empty string, the data will be written in binary format and the resulting file will be smaller but less portable.
- *kargs*: more arguments are passed to `geomfile.GeometryFile.write ()`.

Returns the number of objects written to the file.

`script.readGeomFile (filename, count=-1)`

Read a pyFormex Geometry File.

A pyFormex Geometry File can store multiple geometrical objects in a native format that can be efficiently read back into pyformex. The format is portable over different pyFormex versions and even to other software.

- *filename*: the name of an existing pyFormex Geometry File. If the filename ends on '.gz', it is considered to be a gzipped file and will be uncompressed transparently during the reading.

Returns a dictionary with the geometric objects read from the file. If object names were stored in the file, they will be used as the keys. Else, default names will be provided.

6.1.5 `gui.draw` — Create 3D graphical representations.

The `draw` module provides the basic user interface to the OpenGL rendering capabilities of pyFormex. The full contents of this module is available to scripts running in the pyFormex GUI without the need to import it.

Functions defined in module `gui.draw`

`gui.draw.exitGui (res=0)`

Terminate the GUI with a given status.

`gui.draw.closeGui ()`

Close the GUI.

Calling this function from a script closes the GUI and terminates pyFormex.

`gui.draw.closeDialog (name)`

Close the named dialog.

Closes the `InputDialog` with the given name. If multiple dialogs are open with the same name, all these dialogs are closed.

This only works for dialogs owned by the pyFormex GUI.

`gui.draw.showMessage (text, actions=['OK'], level='info', modal=True, align='00', **kargs)`

Show a short message widget and wait for user acknowledgement.

There are three levels of messages: 'info', 'warning' and 'error'. They differ only in the icon that is shown next to the text. By default, the message widget has a single button with the text 'OK'. The dialog is closed if the user clicks a button. The return value is the button text.

`gui.draw.showInfo (text, actions=['OK'], modal=True)`

Show an informational message and wait for user acknowledgement.

`gui.draw.warning (text, actions=['OK'])`

Show a warning message and wait for user acknowledgement.

`gui.draw.error (text, actions=['OK'])`

Show an error message and wait for user acknowledgement.

`gui.draw.ask (question, choices=None, **kargs)`

Ask a question and present possible answers.

Return answer if accepted or default if rejected. The remaining arguments are passed to the `InputDialog` `getResult` method.

`gui.draw.ack (question, **kargs)`

Show a Yes/No question and return True/False depending on answer.

`gui.draw.showText (text, itemtype='text', actions=[('OK', None)], modal=True, mono=False)`

Display a text in a dialog window.

Creates a dialog window displaying some text. The dialog can be modal (blocking user input to the main window) or modeless. Scrollbars are added if the text is too large to display at once. By default, the dialog has a single button to close the dialog.

Parameters:

- `text`: a multiline text to be displayed. It can be plain text or html or `reStructuredText` (starts with `..`).

- *itemtype*: an InputItem type that can be used for text display. This should be either 'text' or 'info'.
- *actions*: a list of action button definitions.
- *modal*: bool: if True, a modal dialog is constructed. Else, the dialog is modeless.
- *mono*: if True, a monospace font will be used. This is only useful for plain text, e.g. to show the output of an external command.

Returns:

Modal dialog the result of the dialog after closing. The result is a dictionary with a single key: 'text' having the displayed text as a value. If an itemtype 'text' was used, this may be a changed text.

Modeless dialog the open dialog window itself.

`gui.draw.showFile(filename, mono=True, **kargs)`

Display a text file.

This will use the `showText()` function to display a text read from a file. By default this uses a monospaced font. Other arguments may also be passed to ShowText.

`gui.draw.showDoc(obj=None, rst=True, modal=False)`

Show the docstring of an object.

Parameters:

- *obj*: any object (module, class, method, function) that has a `__doc__` attribute. If None is specified, the docstring of the current application is shown.
- *rst*: bool. If True (default) the docstring is treated as being reStructuredText and will be nicely formatted accordingly. If False, the docstring is shown as plain text.

`gui.draw.editFile(fn, exist=False)`

Load a file into the editor.

Parameters:

- *fn*: filename. The corresponding file is loaded into the editor.
- *exist*: bool. If True, only existing filenames will be accepted.

Loading a file in the editor is done by executing an external command with the filename as argument. The command to be used can be set in the configuration. If none is set, pyFormex will try to look at the `EDITOR` and `VISUAL` environment settings.

The main author of pyFormex uses 'emacsclient' as editor command, to load the files in a running copy of Emacs.

`gui.draw.askItems(items, timeout=None, **kargs)`

Ask the value of some items to the user.

Create an interactive widget to let the user set the value of some items. The items are specified as a list of dictionaries. Each dictionary contains the input arguments for a `widgets.InputItem`. It is often convenient to use one of the `_I`, `_G`, or `_T` functions to create these dictionaries. These will respectively create the input for a `simpleInputItem`, a `groupInputItem` or a `tabInputItem`.

For convenience, simple items can also be specified as a tuple. A tuple (key,value) will be transformed to a dict {'key':key, 'value':value}.

See the `widgets.InputDialog` class for complete description of the available input items.

A timeout (in seconds) can be specified to have the input dialog interrupted automatically and return the default values.

The remaining arguments are keyword arguments that are passed to the `widgets.InputDialog.getResult` method.

Returns a dictionary with the results: for each input item there is a (key,value) pair. Returns an empty dictionary if the dialog was canceled. Sets the dialog timeout and accepted status in global variables.

`gui.draw.currentDialog()`
Returns the current dialog widget.

This returns the dialog widget created by the `askItems()` function, while the dialog is still active. If no `askItems()` has been called or if the user already closed the dialog, `None` is returned.

`gui.draw.dialogAccepted()`
Returns `True` if the last `askItems()` dialog was accepted.

`gui.draw.dialogRejected()`
Returns `True` if the last `askItems()` dialog was rejected.

`gui.draw.dialogTimedOut()`
Returns `True` if the last `askItems()` dialog timed out.

`gui.draw.askFile` (*cur=None, filter='all', exist=True, multi=False, compr=False, change=True, timeout=None, caption=None, sidebar=None, **kargs*)
Ask for one or more files using a customized file dialog.

This is like `askFileName()` but returns a `Dict` with the full dialog results instead of the filename(s) themselves. This is especially intended for file types that add custom fields to the `FileDialog`.

Returns *Dict | None* – A `Dict` with the results of the file dialog. If the user accepted the selection, the `Dict` has at least a key 'fn' holding the selected filename(s): a single file name is if *multi* is `False`, or a list of file names if *multi* is `True`. If the user canceled the selection process, the `Dict` is empty.

`gui.draw.askFilename` (**args, **kargs*)
Ask for a file name or multiple file names using a file dialog.

Parameters

- **cur** (*path_like*) – Path of the starting point of the selection dialog. It can be a directory or a file. All the files in the provided directory (or the file's parent) that match the *filter* will be initially presented to the user. If *cur* is a file, it will be set as the initial selection.
- **filter** (*str or list of str*) – Specifies a (set of) filter(s) to be applied on the files in the selected directory. This allows to narrow down the selection possibilities. The *filter* argument is passed through the `utils.fileDescription()` function to create the actual filter set. If multiple filters are included, the user can switch between them in the dialog.
- **exist** (*bool*) – If `True`, the filename must exist. The default (`False`) will allow a new file to be created or an existing to be used.
- **multi** (*bool*) – If `True`, allows the user to pick multiple file names in a single operation.
- **compr** (*bool*) – If `True`, the specified filter pattern will be extended with the corresponding compressed file types. For example, a filter for '.pgf' files will also allow to pick '.pgf.gz' or '.pgf.bz2' files.
- **change** (*bool*) – If `True` (default), the current working directory will be changed to the parent directory of the selection.
- **caption** (*str*) – A string to be displayed as the dialog title instead of the default one.
- **timeout** (*float*) – If provided, the dialog will timeout after the specified number of seconds.
- **sidebar** (*list of path_like.*) – If provided, these will be added to the sidebar (in addition to the configured paths).

- **kargs** (*keyword arguments*) – More arguments to be passed to the `FileDialog`.

Returns *Path* | *list of Paths* | *None* – The selected file *Path*(s) if the user accepted the dialog, or *None* if the user canceled the dialog.

```
gui.draw.askNewFilename (cur=None, filter='All files (*.*)', compr=False, timeout=None, caption=None, sidebar=None, **kargs)
```

Ask a single new filename.

This is a convenience function for calling `askFilename` with the argument `exist=False`.

```
gui.draw.askDirname (path=None, change=True, byfile=False, caption=None)
```

Interactively select a directory and change the current workdir.

The user is asked to select a directory through the standard file dialog. Initially, the dialog shows all the subdirectories in the specified path, or by default in the current working directory.

The selected directory becomes the new working directory, unless the user canceled the operation, or the `change` parameter was set to `False`.

```
gui.draw.checkWorkdir ()
```

Ask the user to change the current workdir if it is not writable.

Returns `True` if the current workdir is writable.

```
gui.draw.printMessage (s, **kargs)
```

Print a message on the message board.

Parameters:

- *s*: string to print
- *kargs*: more keyword arguments are passed to `meth:MessageBoard.write`.

This function forces an update of the GUI, so that the output message is guaranteed to be visible. If a logfile was opened, the message is also written to the log file.

```
gui.draw.delay (s=None)
```

Get/Set the draw delay time.

Returns the current setting of the draw wait time (in seconds). This drawing delay is obeyed by drawing and viewing operations.

A parameter may be given to set the delay time to a new value. It should be convertible to a float. The function still returns the old setting. This may be practical to save that value to restore it later.

```
gui.draw.wait (relock=True)
```

Wait until the drawing lock is released.

This uses the drawing lock mechanism to pause. The drawing lock ensures that subsequent draws are retarded to give the user the time to view. The use of this function is preferred over that of `pause()` or `sleep()`, because it allows your script to continue the numerical computations while waiting to draw the next screen.

This function can be used to retard other functions than `draw` and `view`.

```
gui.draw.play (refresh=False)
```

Start the current script or if already running, continue it.

```
gui.draw.replay ()
```

Replay the current app.

This works pretty much like the `play()` function, but will reload the current application prior to running it. This function is especially interesting during development of an application. If the current application is a script, then it is equivalent with `play()`.

`gui.draw.fforward()`
Releases the drawing lock mechanism indefinitely.

Releasing the drawing lock indefinitely means that the lock will not be set again and your script will execute till the end.

`gui.draw.pause(timeout=None, msg=None)`
Pause the execution until an external event occurs or timeout.

When the pause statement is executed, execution of the pyformex script is suspended until some external event forces it to proceed again. Clicking the PLAY, STEP or CONTINUE button will produce such an event.

- *timeout*: float: if specified, the pause will only last for this many seconds. It can still be interrupted by the STEP buttons.
- *msg*: string: a message to write to the board to explain the user about the pause

`gui.draw.sleep(duration, granularity=0.01)`
Hold execution for some duration

This holds the execution of the thread where the function is called for the specified time (in seconds).

`gui.draw.do_after(sec, func)`
Call a function in another thread after a specified elapsed time.

Parameters

- **sec** (*float*) – Time in seconds to wait before starting the execution. As the function will be executed in a separate thread, the calling thread will immediately continue.
- **func** (*callable*) – The function (or bound method) to be called.

`gui.draw.zoomRectangle()`
Zoom a rectangle selected by the user.

`gui.draw.getRectangle()`
Zoom a rectangle selected by the user.

`gui.draw.zoomBbox(bb)`
Zoom thus that the specified bbox becomes visible.

`gui.draw.zoomObj(object)`
Zoom thus that the specified object becomes visible.

object can be anything having a `bbox()` method or a list thereof.

`gui.draw.zoomAll()`
Zoom thus that all actors become visible.

`gui.draw.zoom(f)`
Zoom with a factor *f*
A factor > 1.0 zooms out, a factor < 1.0 zooms in.

`gui.draw.focus(point)`
Move the camera focus to the specified point.

Parameters:

- *point*: float(3,) or alike

The camera focus is set to the specified point, while keeping a parallel camera direction and same zoom factor. The specified point becomes the center of the screen and the center of camera rotations.

`gui.draw.flyAlong(path, upvector=[0.0, 1.0, 0.0], sleeptime=None)`
Fly through the current scene along the specified path.

- *path*: a plex-2 or plex-3 Formex (or convertible to such Formex) specifying the paths of camera eye and center (and upvector).
- *upvector*: the direction of the vertical axis of the camera, in case of a 2-plex camera path.
- *sleeptime*: a delay between subsequent images, to slow down the camera movement.

This function moves the camera through the subsequent elements of the Formex. For each element the first point is used as the center of the camera and the second point as the eye (the center of the scene looked at). For a 3-plex Formex, the third point is used to define the upvector (i.e. the vertical axis of the image) of the camera. For a 2-plex Formex, the upvector is constant as specified in the arguments.

`gui.draw.viewport (n=None)`
Select the current viewport.

n is an integer number in the range of the number of viewports, or is one of the viewport objects in `pyformex.GUI.viewports`

if *n* is None, selects the current GUI viewport for drawing

`gui.draw.nViewports ()`
Return the number of viewports.

`gui.draw.layout (nrows=None, ncols=None, ncols=None, pos=None, rstretch=None, cstretch=None)`
Set the viewports layout.

`gui.draw.addViewport ()`
Add a new viewport.

`gui.draw.removeViewport ()`
Remove the last viewport.

`gui.draw.linkViewport (vp, tovp)`
Link viewport *vp* to viewport *tovp*.

Both *vp* and *tovp* should be numbers of viewports.

`gui.draw.updateGUI ()`
Update the GUI.

`gui.draw.highlightActor (actor)`
Highlight an actor in the scene.

`gui.draw.removeHighlight ()`
Remove the highlights from the current viewport

`gui.draw.pick (mode='actor', filter=None, oneshot=False, func=None, pickable=None, prompt=None, _rect=None)`
Enter interactive picking mode and return selection.

See `canvas.Canvas.pick ()` for more details. This function differs in that it provides default highlighting during the picking operation and OK/Cancel buttons to stop the picking operation.

Parameters

- **mode** (*str*) – Defines what to pick : one of ‘actor’, ‘element’, ‘face’, ‘edge’, ‘point’ or ‘prop’. ‘actor’ picks complete actors. ‘element’ picks elements from one or more actor(s). ‘face’ and ‘edge’ pick faces, resp. edges of elements (only available for Mesh objects). ‘point’ picks points of Formices or nodes of Meshes. ‘prop’ is like ‘element’, but returns the property numbers of the picked elements instead of the element numbers.
- **filter** (*str*) – The picking filter that is activated on entering the pick mode. It should be one of the `Canvas.selection_filters`: ‘none’, ‘single’, ‘closest’, ‘connected’, ‘closest-connected’ The active filter can be changed from a combobox in the statusbar.

- **oneshot** (*bool.*) – If True, the function returns as soon as the user ends a picking operation. The default is to let the user modify his selection and to return only after an explicit cancel (ESC or right mouse button).
- **func** (*callable, optional*) – If specified, this function will be called after each atomic pick operation. The Collection with the currently selected objects is passed as an argument. This can e.g. be used to highlight the selected objects during picking.
- **pickable** (*list of Actors, optional*) – List of Actors from which can be picked. The default is to use a list with all Actors having the pickable=True attribute (which is the default for newly constructed Actors).
- **prompt** (*str*) – The text printed to prompt the user to start picking. If None, a default prompt is printed. Specify an empty string to avoid printing a prompt.

Returns *Collection* – A (possibly empty) Collection with the picked items. After return, the value of the `pf.canvas.selection_accepted` variable can be tested to find how the picking operation was exited: True means accepted (right mouse click, ENTER key, or OK button), False means canceled (ESC key, or Cancel button). In the latter case, the returned Collection is always empty.

`gui.draw.pickProps` (*filter=None, oneshot=False, func=None, pickable=None, prompt=None*)
Pick property numbers

This is like `pick('element')`, but returns the (unique) property numbers of the picked elements of the actors instead.

`gui.draw.pickNumbers` (*marks=None*)
Pick drawn numbers

`gui.draw.pickFocus` ()
Enter interactive focus setting.

This enters interactive point picking mode and sets the focus to the center of the picked points.

`gui.draw.set_edit_mode` (*s*)
Set the drawing edit mode.

`gui.draw.drawLinesInter` (*mode='line', single=False, func=None*)
Enter interactive drawing mode and return the line drawing.

See `viewport.py` for more details. This function differs in that it provides default displaying during the drawing operation and a button to stop the drawing operation.

The drawing can be edited using the methods 'undo', 'clear' and 'close', which are presented in a combobox.

`gui.draw.showLineDrawing` (*L*)
Show a line drawing.

L is usually the return value of an interactive draw operation, but might also be set by the user.

`gui.draw.exportWebGL` (*fn, createdby=50, **kargs*)
Export the current scene to WebGL.

Parameters:

- *fn* : string: the (relative or absolute) filename of the .html, .js and .pgf files comprising the WebGL model. It can contain a directory path and an any extension. The latter is dropped and not used.
- *createdby*: int: width in pixels of the 'Created by pyFormex' logo appearing on the page. If < 0, the logo is displayed at its natural width. If 0, the logo is suppressed.
- ***kargs*: any other keyword parameteris passed to the WebGL initialization. The *name* can not be specified: it is derived from the *fn* parameter.

Returns the absolute pathname of the generated .html file.

`gui.draw.multiWebGL` (*name=None, fn=None, title=None, description=None, keywords=None, author=None, createdby=50*)

Export the current scene to WebGL.

fn is the (relative or absolute) pathname of the .html and .js files to be created.

When the export is finished, returns the absolute pathname of the generated .html file. Else, returns None.

`gui.draw.showURL` (*url*)

Show an URL in the browser

- *url*: url to load

`gui.draw.showHTML` (*fn=None*)

Show a local .html file in the browser

- *fn*: name of a local .html file. If unspecified, a FileDialog dialog is popped up to select a file.

`gui.draw.resetGUI` ()

Reset the GUI to its default operating mode.

When an exception is raised during the execution of a script, the GUI may be left in a non-consistent state. This function may be called to reset most of the GUI components to their default operating mode.

`gui.draw.flatten` (*objects, recurse=True*)

Flatten a list of objects.

Each item in the list should be either:

- a drawable object,
- a string with the name of such an object,
- a list of any of these three.

This function will flatten the lists and replace the string items with the object they point to. The result is a single list of drawable objects. This function does not enforce the objects to be drawable. That should be done by the caller.

`gui.draw.drawn_as` (*object*)

Check how an object can be drawn.

An object can be drawn (using `draw()`) if it has a method 'actor', 'toFormex' or 'toMesh'. In the first case, it has a native Actor, else, it is first transformed to Formex or Mesh.

Parameters *object* (any object, though usually a Geometry instance) – An object to check for a drawing method.

Returns *object* (*drawable object or None*) – If the object is drawable (directly or after conversion), returns a directly drawable object, else None.

`gui.draw.drawable` (*objects*)

Filters the drawable objects from a list of objects.

Parameters *objects* (*list or sequence of objects.*) – The list of objects to filter for drawable objects.

Returns *list of objects* – The list of objects that can be drawn.

`gui.draw.draw` (*F, clear=None, **kargs*)

Draw geometrical object(s) with specified drawing options and settings.

This is the generic drawing function in pyFormex. The function requires a single positional parameter specifying the geometry to be drawn. There are also a whole lot of optional keyword parameters, divided in two groups.

The first are the drawing options, which modify the way the draw function operates. If not specified, or a value None is specified, they are filled in from the current viewport drawing options, which can be changed using the `setDrawOptions()` function. The initial defaults are: `clear=False`, `view='last'`, `bbox='auto'`, `shrink=False`, `shrinkfactor=0.8`, `wait=True`, `silent=True`, `single=False`.

The second group are rendering attributes that define the way the geometrical objects should be rendered. These have default values in `canvas.Canvas.settings`, and can be overridden per object by the object's `attrib()` settings. These options are listed below under Notes.

Parameters

- **F** (*object or list of objects*) – The object(s) to be drawn. It can be a single item or a (possibly nested) list of items. The list will be flattened. Strings are looked up in the pyFormex global project dictionary and replaced with their value. Nondrawable objects are filtered out from the list (see also option `silent`). The resulting list of drawable objects is processed with the same drawing options and default rendering attributes.
- **clear** (*bool, optional*) – If True, the scene is cleared before drawing. The default is to add to the existing scene.
- **view** (*str*) – Either the name of a defined view or 'last'. This defines the orientation of the camera looking at the drawn objects. Predefined views are 'front', 'back', 'top', 'bottom', 'left', 'right', 'iso' and a whole list of other ones. * TODO: we should expand this * On creation of a viewport, the initial default view is 'front' (looking in the -z direction). With `view='last'`, the camera angles will be set to the same camera angles as in the last draw operation, undoing any interactive changes. With `view=None` the camera settings remain unchanged (but still may be changed interactively through the user interface). This may make the drawn object out of view. See also `bbox`.
- **bbox** (*array_like or str*) – Specifies the 3D volume at which the camera will be aimed (using the angles set by `view`). The camera position will be set thus that the volume comes in view using the current lens (default 45 degrees). `bbox` is a list of two points or compatible (array with shape (2,3)). Setting the `bbox` to a volume not enclosing the object may make the object invisible on the canvas. The special value `bbox='auto'` will use the bounding box of the objects getting drawn (`object.bbox()`), thus ensuring that the camera will focus on these objects. This is the default when creating a new viewport. A value `bbox=None` will use the bounding box of the previous drawing operation, thus ensuring that the camera's target volume is unchanged.
- **shrink** (*bool*) – If specified, each object will be transformed by the `Coords.shrink()` transformation (with the default or specified `shrink_factor` as a parameter), thus showing all the elements of the object separately (sometimes called an 'exploded' view).
- **shrink_factor** (*float*) – Overrides the default `shrink_factor` for the current draw operation. If provided, it forces `shrink=True`.
- **wait** (*bool*) – If True (initial default), the draw action activates a locking mechanism for the next draw action, which will only be allowed after `drawdelay` seconds have elapsed. This makes it easier to see subsequent renderings and is far more efficient than adding an explicit `sleep()` operation, because the script processing can continue up to the next drawing instruction. The value of `drawdelay` can be changed in the user settings or using the `delay()` function. Setting this value to 0 will disable the waiting mechanism for all subsequent draw statements (until set > 0 again). But often the user wants to specifically disable the waiting lock for some draw operation(s). This can be done without changing the `drawdelay` setting, by specifying `wait=False`. This means that the *next* draw operation does not have to wait.
- **silent** (*bool*) – If True (initial default), non-drawable objects are silently ignored. If set False, an error is raised if `F` contains an object that is not drawable.

- **single** (*bool, optional*) – If True, the return value will be a single Actor, corresponding with the first drawable object in the flattened list of F. The remainder of the drawable objects in F are then set as children of the main return value. The default is to return a single Actor if F is a single drawable object, or a list of Actors if F is a list.
- **kargs** (*keyword parameters*) – The remaining keyword parameters are the default rendering attributes to be used for all the objects in F. They will apply unless overridden by attributes set in the object itself (see `geometry.Geometry.attrib()`). There is a long list of possible settings. The important ones are listed below (see Notes).

Returns Actor or list of Actors – If F is a single object or `single==True` was provided, returns a single Actor instance. If F is a list and `single==True` was not set, a list of Actors is returned.

Notes

- This section is incomplete and needs an update *

Here is an (incomplete) list of rendering attributes that can be provided to the draw function and will be used as defaults for drawing the objects that do not have the needed values set as attributes on the object itself. While the list is long, in most cases only a few are used, and the remainder are taken from the canvas rendering defaults.

These arguments will be passed to the corresponding Actor for the object. The Actor is the graphical representation of the geometry. Not all Actors use all of the settings that can be specified here. But they all accept specifying any setting even if unused. The settings hereafter are thus a superset of the settings used by the different Actors. Settings have a default value per viewport, and if unspecified, most Actors will use the viewport default for that value.

- *color, colormap*: specify the color of the object (see below)
- *alpha*: float (0.0..1.0): alpha value to use in transparent mode. 0.0 means fully transparent (invisible), while 1.0 means opaque.
- *bkcolor, bkcolormap*: color for the backside of surface type geometry, if it is to be different from the front side. Specifications are as for front color and colormap.
- *bkalpha*: float (0.0..1.0): transparency alpha value for the back side.
- *linewidth*: float, thickness of line drawing
- *linestipple*: stipple pattern for line drawing
- *marksize*: float: point size for dot drawing
- *nolight*: bool: render object as unlighted in modes with lights on
- *ontop*: bool: render object as if it is on top. This will make the object fully visible, even when it is hidden by other objects. If more than one objects is drawn with *ontop=True* the visibility of the object will depend on the order of drawing.

Specifying color: Color specification can take many different forms. Some Actors recognize up to six different color modes and the draw function adds even another mode (property color)

- no color: *color=None*. The object will be drawn in the current viewport foreground color.
- single color: the whole object is drawn with the specified color.
- element color: each element of the object has its own color. The specified color will normally contain precisely *nelems* colors, but will be resized to the required size if not.
- vertex color: each vertex of each element of the object has its color. In smooth shading modes intermediate points will get an interpolated color.

- element index color: like element color, but the color values are not specified directly, but as indices in a color table (the *colormap* argument).
- vertex index color: like vertex color, but the colors are indices in a color table (the *colormap* argument).
- property color: as an extra mode in the draw function, if *color='prop'* is specified, and the object has an attribute 'prop', that attribute will be used as a color index and the object will be drawn in element index color mode. If the object has no such attribute, the object is drawn in no color mode.

Element and vertex color modes are usually only used with a single object in the *F* parameter, because they require a matching set of colors. Though the color set will be automatically resized if not matching, the result will seldomly be what the user expects. If single colors are specified as a tuple of three float values (see below), the correct size of a color array for an object with *nelems* elements of plexitude *nplex* would be: (*nelems*,3) in element color mode, and (*nelems*,*nplex*,3) in vertex color mode. In the index modes, color would then be an integer array with shape respectively (*nelems*,) and (*nelems*,*nplex*). Their values are indices in the colormap array, which could then have shape (*ncolors*,3), where *ncolors* would be larger than the highest used value in the index. If the colormap is insufficiently large, it will again be wrapped around. If no colormap is specified, the current viewport colormap is used. The default contains eight colors: black=0, red=1, green=2, blue=3, cyan=4, magenta=5, yellow=6, white=7.

A color value can be specified in multiple ways, but should be convertible to a normalized OpenGL color using the `colors.GLcolor()` function. The normalized color value is a tuple of three values in the range 0.0..1.0. The values are the contributions of the red, green and blue components.

`gui.draw.setDrawOptions` (*kargs0*={}, ***kargs*)
 Set default values for the draw options.

Draw options are a set of options that hold default values for the draw() function arguments and for some canvas settings. The draw options can be specified either as a dictionary, or as keyword arguments.

`gui.draw.reset` ()
 reset the canvas

`gui.draw.setShrink` (*shrink*=None, *factor*=None)
 Set shrink mode on or off, and optionally the shrink factor.

In shrink mode, all elements are drawn shrunked around their centroid. This results in an exploded view showing individual elements and permitting look through the inter-element gaps to what is behind.

Parameters

- **shrink** (*float* | *bool* | *None*) – If a float, switches shrink mode on and sets the shrink factor to the provided value. If True, switches on shrink mode with the current shrink factor (`pf.canvas.drawoptions['shrink_factor']`). If False, switches off shrink mode.
- **factor** (float, optional, *deprecated*) – If provided, sets the default shrink factor to this value.

Notes

Useful values for the shrink factor are in the range 0.0 to 1.0. The initial value is 0.8. The current shrink status and factor are stored in `pf.canvas.drawoptions`.

`gui.draw.drawVectors` (*P*, *v*, *size*=None, *nolight*=True, ***drawOptions*)
 Draw a set of vectors.

If *size* is None, draws the vectors *v* at the points *P*. If *size* is specified, draws the vectors `size*normalize(v)` *P*, *v* and *size* are single points or sets of points. If sets, they should be of the same size.

Other drawoptions can be specified and will be passed to the draw function.

`gui.draw.drawMarks` (*X*, *M*, *color='black'*, *prefix=""*, *ontop=True*, ***kargs*)
 Draw a list of marks at points *X*.

Parameters:

- *X*: Coords.
- *M*: list of length `X.ncoords()`. The string representation of the items in the list are drawn at the corresponding 3D coordinate of *X*.
- *prefix*: string. If specified, it is prepended to all drawn strings.
- *ontop*: bool. If True, the marks are drawn on top, meaning they will all be visible, even those drawn at points hidden by the geometry. If False, hidden marks can be hidden by the drawn geometry.

Other parameters can be passed to the `actors.TextArray` class.

`gui.draw.drawFreeEdges` (*M*, *color='black'*)
 Draw the feature edges of a Mesh

`gui.draw.drawNumbers` (*G*, *numbers=None*, *color='black'*, *trl=None*, *offset=0*, *prefix=""*, *ontop=None*, ***kargs*)
 Draw numbers on all elements of a Geometry *G*.

Parameters:

- *G*: Geometry like (Coords, Formex, Mesh)
- *numbers*: int array of length `F.nelems()`. If not specified, the range from 0 to `F.nelems()-1` is used.
- *color*: color to be used in drawing the numbers.
- *trl*: If unspecified, the numbers are drawn at the centroids of the elements. A translation (*x,y,z*) may be given to put the numbers out of the centroids, e.g. to put them in front of the objects to make them visible, or to allow to view a mark at the centroids.
- *offset*: int. If specified, this value is added to the numbers. This is an easy ways to compare the drawing with systems using base 1 numbering.
- *prefix*: string. If specified, it is added before every drawn number.

Other parameters are passed to the `drawMarks()` function.

`gui.draw.drawPropNumbers` (*F*, ***kargs*)
 Draw property numbers on all elements of *F*.

This calls `drawNumbers` to draw the property numbers on the elements. All arguments of `drawNumbers` except *numbers* may be passed. If the object *F* thus not have property numbers, -1 values are drawn.

`gui.draw.drawVertexNumbers` (*F*, *color='black'*, *trl=None*, *ontop=False*)
 Draw (local) numbers on all vertices of *F*.

Normally, the numbers are drawn at the location of the vertices. A translation may be given to put the numbers out of the location, e.g. to put them in front of the objects to make them visible, or to allow to view a mark at the vertices.

`gui.draw.drawBbox` (*F*, *color='black'*, ***kargs*)
 Draw the bounding box of the geometric object *F*.

F is any object that has a `bbox` method. Returns the drawn Annotation.

`gui.draw.drawText` (*text*, *pos*, ***kargs*)
 Show a text at position *pos*.

Draws a text at a given position. The position can be either a 2D canvas position, specified in pixel coordinates (int), or a 3D position, specified in global world coordinates (float). In the latter case the text will be displayed

on the canvas at the projected world point, and will move with that projection, while keeping the text unscaled and oriented to the viewer. The 3D mode is especially useful to annotate parts of the geometry with a label.

Parameters:

- *text*: string to be displayed.
- *pos*: (2,) int or (3,) float: canvas or world position.
- any other parameters are passed to `opengl.texttext.Text`.

`gui.draw.drawText3D(text, pos, **kargs)`
 Show a text at position pos.

Draws a text at a given position. The position can be either a 2D canvas position, specified in pixel coordinates (int), or a 3D position, specified in global world coordinates (float). In the latter case the text will be displayed on the canvas at the projected world point, and will move with that projection, while keeping the text unscaled and oriented to the viewer. The 3D mode is especially useful to annotate parts of the geometry with a label.

Parameters:

- *text*: string to be displayed.
- *pos*: (2,) int or (3,) float: canvas or world position.
- any other parameters are passed to `opengl.texttext.Text`.

`gui.draw.drawViewportAxes3D(pos, color=None)`
 Draw two viewport axes at a 3D position.

`gui.draw.drawAxes(cs=None, **kargs)`
 Draw the axes of a coordinate system.

Parameters:

- *cs*: a `coordsys.CoordSys` If not specified, the global coordinate system is used.

Other arguments can be added just like in the `candy.Axes` class.

By default this draws the positive parts of the axes in the colors R,G,B and the negative parts in C,M,Y.

`gui.draw.drawPrincipal(F, weight=None, **kargs)`
 Draw the principal axes of the geometric object F.

F is Coords or Geometry. If weight is specified, it is an array of weights attributed to the points of F. It should have the same length as `F.coords`. Other parameter are drawing attributes passed to `drawAxes()`.

`gui.draw.drawImage3D(image, nx=0, ny=0, pixel='dot')`
 Draw an image as a colored Formex

Draws a raster image as a colored Formex. While there are other and better ways to display an image in pyFormex (such as using the `imageView` widget), this function allows for interactive handling the image using the OpenGL infrastructure.

Parameters:

- *image*: a `QImage` or any data that can be converted to a `QImage`, e.g. the name of a raster image file.
- *nx*, *ny*: width and height (in cells) of the Formex grid. If the supplied image has a different size, it will be rescaled. Values ≤ 0 will be replaced with the corresponding actual size of the image.
- *pixel*: the Formex representing a single pixel. It should be either a single element Formex, or one of the strings 'dot' or 'quad'. If 'dot' a single point will be used, if 'quad' a unit square. The difference will be important when zooming in. The default is 'dot'.

Returns the drawn Actor.

See also `drawImage()`.

```
gui.draw.drawImage (image, w=0, h=0, x=-1, y=-1, color=(1.0, 1.0, 1.0), ontop=False)
```

Draws an image as a viewport decoration.

Parameters:

- *image*: a QImage or any data that can be converted to a QImage, e.g. the name of a raster image file. See also the `loadImage()` function.
- *w*, *h*: width and height (in pixels) of the displayed image. If the supplied image has a different size, it will be rescaled. A value ≤ 0 will be replaced with the corresponding actual size of the image.
- *x*, *y*: position of the lower left corner of the image. If negative, the image will be centered on the current viewport.
- *color*: the color to mix in (AND) with the image. The default (white) will make all pixels appear as in the image.
- *ontop*: determines whether the image will appear as a background (default) or at the front of the 3D scene (as on the camera glass).

Returns the Decoration drawn.

Note that the Decoration has a fixed size (and position) on the canvas and will not scale when the viewport size is changed. The `bgcolor()` function can be used to draw an image that completely fills the background.

```
gui.draw.drawField (fld, comp=0, scale='RAINBOW', symmetric_scale=False, cvalues=None,
                  **kargs)
```

Draw intensity of a scalar field over a Mesh.

Parameters:

- *fld*: a Field, specifying some value over a Geometry.
- *comp*: int: if *fld* is a vectorial Field, specifies the component that is to be drawn.
- *scale*: one of the color palettes defined in `colorscale`. If an empty string is specified, the scale is not drawn.
- *symmetric_scale*: bool : if *True* the mid value of the color scale will be set to the value corresponding to the middle value of the *fld* data range. If *False* the mid value of the color scale will be set to 0.0 if the range extends over negative and positive values.
- *cvalues*: None or list : specify the values between which to span the color palette. If *None* the min, max and mid values are taken from the field data. If *list* the values can be defined by the user as a list of 2 values (min, max) or 3 values (min, mid, max).
- ***kargs*: any not-recognized keyword parameters are passed to the draw function to draw the Geometry.

Draws the Field's Geometry with the Field data converted to colors. A color legend is added to convert colors to values. NAN data are converted to numerical values using `numpy.nan_to_num`.

```
gui.draw.drawActor (A)
```

Draw an actor and update the screen.

```
gui.draw.drawAny (A)
```

Draw an Actor/Annotation/Decoration and update the screen.

```
gui.draw.undraw (items)
```

Remove an item or a number of items from the canvas.

Use the return value from one of the draw... functions to remove the item that was drawn from the canvas. A single item or a list of items may be specified.

`gui.draw.view(v, wait=True)`

Show a named view, either a builtin or a user defined.

This shows the current scene from another viewing angle. Switching views of a scene is much faster than redrawing a scene. Therefore this function is preferred over `draw()` when the actors in the scene remain unchanged and only the camera viewpoint changes.

Just like `draw()`, this function obeys the drawing lock mechanism, and by default it will restart the lock to retard the next drawing operation.

`gui.draw.createView(name, angles, addtogui=False)`

Create a new named view (or redefine an old).

The angles are (longitude, latitude, twist). The named view is global to all viewports. If `addtogui` is True, a view button to set this view is added to the GUI.

`gui.draw.setView(name, angles=None)`

Set the default view for future drawing operations.

If no angles are specified, the name should be an existing view, or the predefined value 'last'. If angles are specified, this is equivalent to `createView(name,angles)` followed by `setView(name)`.

`gui.draw.setTriade(on=None, pos='lb', siz=50, triade=None)`

Toggle the display of the global axes on or off.

This is a convenient feature to display the global axes directions with rotating actor at fixed viewport size and position.

Parameters:

- *on*: boolean. If True, the global axes triade is displayed. If False, it is removed. The default (None) toggles between on and off. The remaining parameters are only used on enabling the triade.
- *pos*: string of two characters. The characters define the horizontal (one of 'l', 'c', or 'r') and vertical (one of 't', 'c', 'b') position on the camera's viewport. Default is left-bottom.
- *siz*: size (in pixels) of the triade.
- *triade*: None, Geometry or str: defines the Geometry to be used for representing the global axes.

If None: use the previously set triade, or set a default if no previous.

If Geometry: use this to represent the axes. To be useful and properly displayed, the Geometry's bbox should be around $[(-1,-1,-1),(1,1,1)]$. Drawing attributes may be set on the Geometry to influence the appearance. This allows to fully customize the Triade.

If str: use one of the predefined Triade Geometries. Currently, the following are available:

- 'axes': axes and coordinate planes as in `candy.Axes`
- 'man': a model of a man as in data file 'man.pgf'

`gui.draw.setGrid(on=None, d=None, s=None, **kargs)`

Toggle the display of the canvas grid on or off.

Parameters

- *on* (*bool.*) – If True, the grid is displayed. If False, it is removed. The default (None) toggles between on and off.
- *d* (*None, int or (int,int), optional*) – Only used when `on==True`. Distance in pixels between the grid lines. A tuple of two values specifies the distance in x,y direction. If not specified, the previous grid is used, or a default grid with `d=100` is created.

- **s** (*None, int or (int,int), optional*) – Only used when `on==True`. The grid size in pixels. A tuple of two values specifies size in x,y direction. If not specified the size is set equal to the desktop screen size. This allows resizing the window while still seeing the grid on the full canvas.
- **kargs** (*optional*) – Extra drawing parameters that influence the appearance of the grid.
Example:

```
setGrid(d=200,linewidth=3,color=red,ontop=True)
```

Notes

This is a convenient function to display a grid on the canvas. The grid may someday become an integral part of the Canvas.

`gui.draw.annotate` (*annot*)
Draw an annotation.

`gui.draw.decorate` (*decor*)
Draw a decoration.

`gui.draw.bgcolor` (*color=None, image=None*)
Change the background color and image.

Parameters:

- *color*: a single color or a list of 4 colors. A single color sets a solid background color. A list of four colors specifies a gradient. These 4 colors are those of the Bottom Left, Bottom Right, Top Right and Top Left corners respectively.
- *image*: the name of an image file. If specified, the image will be overlaid on the background colors. Specify a solid white background color to see the image unaltered.

`gui.draw.fgcolor` (*color*)
Set the default foreground color.

`gui.draw.hicolor` (*color*)
Set the highlight color.

`gui.draw.colormap` (*color=None*)
Gets/Sets the current canvas color map

`gui.draw.colorindex` (*color*)
Return the index of a color in the current colormap

`gui.draw.renderModes` ()
Return a list of predefined render profiles.

`gui.draw.renderMode` (*mode, light=None*)
Change the rendering profile to a predefined mode.

Currently the following modes are defined:

- wireframe
- smooth
- smoothwire
- flat
- flatwire

- `smooth_avg`

`gui.draw.wireMode(mode)`

Change the wire rendering mode.

Currently the following modes are defined: 'none', 'border', 'feature', 'all'

`gui.draw.lights(state=True)`

Set the lights on or off

`gui.draw.transparent(state=True)`

Set the transparency mode on or off.

`gui.draw.set_material_value(typ, val)`

Set the value of one of the material lighting parameters

`typ` is one of 'ambient', 'specular', 'emission', 'shininess' `val` is a value between 0.0 and 1.0

`gui.draw.linewidth(wid)`

Set the linewidth to be used in line drawings.

`gui.draw.linestipple(factor, pattern)`

Set the linewidth to be used in line drawings.

`gui.draw.pointsize(siz)`

Set the size to be used in point drawings.

`gui.draw.canvasSize(width, height)`

Resize the canvas to (width x height).

If a negative value is given for either width or height, the corresponding size is set equal to the maximum visible size (the size of the central widget of the main window).

Note that changing the canvas size when multiple viewports are active is not approved.

`gui.draw.clear(sticky=False)`

Clear the canvas.

Removes everything from the current scene and displays an empty background.

This function waits for the drawing lock to be released, but will not reset it.

6.1.6 `opengl.colors` — Playing with colors.

This module defines some colors and color conversion functions. It also defines a default palette of colors.

The following table shows the colors of the default palette, with their name, RGB values in 0..1 range and luminance.

```
>>> for k,v in palette.items():
...     print("%12s = %s -> %0.3f" % (k,v,luminance(v)))
darkgrey = (0.4, 0.4, 0.4) -> 0.133
red = (1.0, 0.0, 0.0) -> 0.213
green = (0.0, 1.0, 0.0) -> 0.715
blue = (0.0, 0.0, 1.0) -> 0.072
cyan = (0.0, 1.0, 1.0) -> 0.787
magenta = (1.0, 0.0, 1.0) -> 0.285
yellow = (1.0, 1.0, 0.0) -> 0.928
white = (1.0, 1.0, 1.0) -> 1.000
black = (0.0, 0.0, 0.0) -> 0.000
darkred = (0.5, 0.0, 0.0) -> 0.046
darkgreen = (0.0, 0.5, 0.0) -> 0.153
darkblue = (0.0, 0.0, 0.5) -> 0.015
```

(continues on next page)

(continued from previous page)

```

darkcyan = (0.0, 0.5, 0.5) -> 0.169
darkmagenta = (0.5, 0.0, 0.5) -> 0.061
darkyellow = (0.5, 0.5, 0.0) -> 0.199
lightgrey = (0.8, 0.8, 0.8) -> 0.604

```

Functions defined in module `opengl.colors`

`opengl.colors.GLcolor` (*color*)

Convert a color to an OpenGL RGB color.

The output is a tuple of three RGB float values ranging from 0.0 to 1.0. The input can be any of the following:

- a QColor
- a string specifying the X11 name of the color
- a hex string '#RGB' with 1 to 4 hexadecimal digits per color
- a tuple or list of 3 integer values in the range 0..255
- a tuple or list of 3 float values in the range 0.0..1.0

Any other input may give unpredictable results.

```

Examples: >>> GLcolor('red') (1.0, 0.0, 0.0) >>> GLcolor('indianred') # doctest: +ELLIPSIS (0.8039...,
0.3607..., 0.3607...) >>> GLcolor('grey90') # doctest: +ELLIPSIS (0.8980..., 0.8980..., 0.8980...)
>>> print(GLcolor('#ff0000')) (1.0, 0.0, 0.0) >>> GLcolor(red) (1.0, 0.0, 0.0) >>> GLcolor([200,200,255])
(0.7843137254901961, 0.7843137254901961, 1.0) >>> GLcolor([1.,1.,1.]) (1.0, 1.0, 1.0) >>> GLcolor(0.6)
(0.6, 0.6, 0.6)

```

`opengl.colors.GLcolorA` (*color*)

Convert a color to an OpenGL RGB color.

The output is a tuple of three RGB float values ranging from 0.0 to 1.0. The input can be any of the following:

- a QColor
- a string specifying the Xwindow name of the color
- a hex string '#RGB' with 1 to 4 hexadecimal digits per color
- a tuple or list of 3 integer values in the range 0..255
- a tuple or list of 3 float values in the range 0.0..1.0

Any other input may give unpredictable results.

Example

```

>>> GLcolorA('indianred')
array([ 0.8 ,  0.36,  0.36])
>>> print(GLcolorA('#ff0000'))
[ 1.  0.  0.]
>>> GLcolorA(red)
array([ 1.,  0.,  0.])
>>> GLcolorA([200,200,255])
array([ 0.78,  0.78,  1.  ])
>>> GLcolorA([1.,1.,1.])
array([ 1.,  1.,  1.])

```

(continues on next page)

(continued from previous page)

```
>>> GLcolorA(0.6)
array([ 0.6,  0.6,  0.6])
>>> print(GLcolorA(['black', 'red', 'green', 'blue']))
[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

`opengl.colors.RGBcolor` (*color*)

Return an RGB (0-255) tuple for a color

color can be anything that is accepted by GLcolor.

Returns the corresponding RGB colors as a numpy array of type uint8 and shape (...3).

Example

```
>>> RGBcolor(red)
array([255,  0,  0], dtype=uint8)
```

`opengl.colors.RGBAcolor` (*color*, *alpha*)

Return an RGBA (0-255) tuple for a color and alpha value.

color can be anything that is accepted by GLcolor.

Returns the corresponding RGBA colors as a numpy array of type uint8 and shape (...4).

`opengl.colors.WEBcolor` (*color*)

Return an RGB hex string for a color

color can be anything that is accepted by GLcolor. Returns the corresponding WEB color, which is a hexadecimal string representation of the RGB components.

Example

```
>>> WEBcolor(red)
'ff0000'
```

`opengl.colors.colorName` (*color*)

Return a string designation for the color.

color can be anything that is accepted by GLcolor. In the current implementation, the returned color name is the WEBcolor (hexadecimal string).

Example

```
>>> colorName('red')
'ff0000'
>>> colorName('#ffddff')
'ffddff'
>>> colorName([1., 0., 0.5])
'ff0080'
```

`opengl.colors.luminance` (*color*, *gamma=True*)

Compute the luminance of a color.

Returns a floating point value in the range 0..1 representing the luminance of the color. The higher the value, the brighter the color appears to the human eye.

This can be for example be used to derive a good contrasting foreground color to display text on a colored background. Values lower than 0.5 contrast well with white, larger value contrast better with black.

Example

```
>>> print([ "%0.2f" % luminance(c) for c in ['black','red','green','blue']])
['0.00', '0.21', '0.72', '0.07']
>>> print(luminance(['black','red','green','blue']))
[ 0.    0.21  0.72  0.07]
```

`opengl.colors.closestColorName` (*color*)

Return the closest color name.

`opengl.colors.RGBA` (*rgb*, *alpha=1.0*)

Adds an alpha channel to an RGB color

`opengl.colors.GREY` (*val*, *alpha=1.0*)

Returns a grey OpenGL color of given intensity (0..1)

6.2 Other pyFormex core modules

Together with the autoloaded modules, the following modules located under the main pyformex path are considered to belong to the pyformex core functionality.

6.2.1 mesh —

This module defines the Mesh class, which can be used to describe discrete geometrical models like those used in Finite Element models. It also contains some useful functions to create such models.

Classes defined in module mesh

class `mesh.Mesh` (*coords=None*, *elems=None*, *prop=None*, *eltype=None*)

A Mesh is a discrete geometrical model defined by nodes and elements.

The Mesh class is one of the two basic geometrical models in pyFormex, the other one being the `Formex`. Both classes have a lot in common: they represent a collection of geometrical entities of the same type (e.g., lines, or triangles, ...). The geometrical entities are also called 'elements', and the number of elements in the Mesh is `nelems()`. The *plexitude* (the number of points in an element) of a Mesh is found from `nplex()`. Each point has `ndim=3` coordinates. While in a `Formex` all these points are stored in an array with shape `(nelems, nplex, 3)`, the `Mesh` stores the information in two arrays: the coordinates of all the points are gathered in a single twodimensional array with shape `(ncoords,3)`. The individual geometrical elements are then described by indices into that array: we call that the connectivity, with shape `(nelems, nplex)`.

This model has some advantages over the `Formex` data model:

- a more compact storage, because coordinates of coinciding points require only be stored once (and we usually call the points *nodes*);

- the single storage of coinciding points represents the notion of connections between elements (a `Formex` to the contrary is always a loose collection of elements);
- connectivity related algorithms are generally faster;
- the connectivity info also allows easy identification of geometric subentities (entities of a lower *level*, like the border lines of a surface).

The downside is that geometry generating and replicating algorithms are often far more complex and possibly slower.

In pyFormex we therefore mostly use the Formex data model when creating, copying and replicating geometry, but when we come to the point of needing connectivity related algorithms or exporting the geometry to file (and to other programs), a Mesh data model usually becomes more appropriate. A `Formex` can be converted into a `Mesh` with the `:meth:`Formex.toMesh` method, while the `Mesh.toFormex()` method performs the inverse conversion.

Parameters

- **coords** (*Coords* or other object.) – Usually, a 2-dim *Coords* object holding the coordinates of all the nodes used in the Mesh geometry. See details below for different initialization methods.
- **elems** (*Connectivity* (nelems,nplex)) – A *Connectivity* object, defining the elements of the geometry by indices into the *coords* *Coords* array. All values in *elems* should be in the range $0 \leq \text{value} < \text{ncoords}$.
- **prop** (int *array_like*, optional) – 1-dim int array with non-negative element property numbers. If provided, `setProp()` will be called to assign the specified properties.
- **eltype** (str or *ElementType*, optional) – The element type of the geometric entities (elements). This is only needed if the element type has not yet been set in the *elems* *Connectivity*. See below.

A `Mesh` object can be initialized in many different ways, depending on the values passed for the *coords* and *elems* arguments.

- **Coords, Connectivity:** This is the most obvious case: *coords* is a 2-dim *Coords* object holding the coordinates of all the nodes in the Mesh, and *elems* is a *Connectivity* object describing the geometric elements by indices into the *coords*.
- **Coords, :** If A *Coords* is passed as first argument, but no *elems*, the result is a Mesh of points, with plexitude 1. The *Connectivity* will be constructed automatically.
- **object with toMesh, :** As a convenience, if another object is provided that has a `toMesh` method and *elems* is not provided, the result of the `toMesh` method will be used to initialize both *coords* and *elems*.
- **None:** If neither *coords* nor *elems* are specified, but *eltype* is, a unit sized single element Mesh of the specified *ElementType* is created.
- **Specifying no parameters at all** creates an empty Mesh, without any data.

Setting the element type can also be done in different ways. If *elems* is a *Connectivity*, it will normally already have a element type. If not, it can be done by passing it in the *eltype* parameter. In case you pass a simple array or list in the *elems* parameter, an element type is required. Finally, the user can specify an *eltype* to override the one in the *Connectivity*. It should however match the plexitude of the connectivity data.

eltype should be one of the *ElementType* instances or the name of such an instance. If required but not provided, the pyFormex default is used, which is based on the plexitude: 1 = point, 2 = line segment, 3 = triangle, 4 or more is a polygon.

A properly initialized Mesh has the following attributes:

coords

A 2-dim Coords object holding the coordinates of all the nodes used to describe the Mesh geometry.

Type *Coords* (ncoords,3)

elems

A Connectivity object, defining the elements of the geometry by indices into the *coords* Coords array. All values in elems should be in the range $0 \leq \text{value} < \text{ncoords}$.

The Connectivity also stores the element type of the Mesh.

Type *Connectivity* (nelems,nplex)

prop

Element property numbers. See *geometry.Geometry.prop*.

Type int array, optional

attrib

An Attributes object. See *geometry.Geometry.attrib*.

Type *Attributes*

fields

The Fields defined on the Mesh. See *geometry.Geometry.fields*.

Type OrderedDict

Note: The *coords*' attribute of a Mesh can hold points that are not used or needed to describe the Geometry. They do not influence the result of Mesh operations, but only use up some memory. If their number becomes large, you may want to free up that memory by calling the *compact()* method. Also, before exporting a Mesh (e.g. to a numerical simulation program), you may want to compact the Mesh first.

Examples

Create a Mesh with four points and two triangle elements of type 'tri3'.

```
>>> coords = Coords('0123')
>>> elems = [[0,1,2], [0,2,3]]
>>> M = Mesh(coords,elems,eltype='tri3')
>>> print(M.report())
Mesh: nnodes: 4, nelems: 2, nplex: 3, level: 2, eltype: tri3
  BBox: [ 0.  0.  0.], [ 1.  1.  0.]
  Size: [ 1.  1.  0.]
  Area: 1.0
  Coords: [[ 0.  0.  0.]
           [ 1.  0.  0.]
           [ 1.  1.  0.]
           [ 0.  1.  0.]]
  Elems: [[0 1 2]
          [0 2 3]]
>>> M.nelems(), M.ncoords(), M.nplex(), M.level(), M.elName()
(2, 4, 3, 2, 'tri3')
```

And here is a line Mesh converted from of a Formex:

```

>>> M1 = Formex('1:11').toMesh()
>>> print(M1.report())
Mesh: nnodes: 3, nelems: 2, nplex: 2, level: 1, eltype: line2
  BBox: [ 0.  0.  0.], [ 2.  0.  0.]
  Size: [ 2.  0.  0.]
  Length: 2.0
  Coords: [[ 0.  0.  0.]
           [ 1.  0.  0.]
           [ 2.  0.  0.]]
  Elems: [[0 1]
          [1 2]]

```

Indexing returns the full coordinate set of the element(s):

```

>>> M1[0]
Coords([[ 0.,  0.,  0.],
        [ 1.,  0.,  0.]])

```

The Mesh class inherits from `Geometry` and therefore has all the coordinate transform methods defined there readily available:

```

>>> M2 = M1.rotate(90)
>>> print(M2.coords)
[[ 0.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  2.  0.]]

```

eltype

Return the element type of the Mesh.

Returns `elements.ElementType` – The eltype attribute of the `elems` attribute.

Examples

```

>>> M = Mesh(eltype='tri3')
>>> M.eltype
Tri3
>>> M.eltype = 'line3'
>>> M.eltype
Line3
>>> print(M)
Mesh: nnodes: 3, nelems: 1, nplex: 3, level: 1, eltype: line3
  BBox: [ 0.  0.  0.], [ 1.  1.  0.]
  Size: [ 1.  1.  0.]
  Length: 1.0

```

One cannot set an element type with nonmatching plexitude:

```

>>> M.eltype = 'quad4'
>>> M.eltype
'plex3'

```

setEltype (eltype=None)

Set the eltype from a character string.

Parameters `eltype` (str or `ElementType`, optional) – The element type to be set in the `elems` Connectivity. It is either one of the `ElementType` instances defined in `elements.py`, or

the name of such an instance. The plexitude of the ElementType should match the plexitude of the Mesh.

Returns *Mesh* – The Mesh itself with possibly changed eltype.

Examples

```
>>> Mesh(eltype='tri3').setEltype('line3').eltype
Line3
```

`elType()`

Return the element type of the Mesh.

Returns *ElementType* – The ElementType of the Mesh.

See also:

`elName()` returns the name of the ElementType.

Examples

```
>>> Formex('4:0123').toMesh().elType()
Quad4
```

`elName()`

Return the element name of the Mesh.

Returns *str* – The name of the ElementType of the Mesh.

See also:

`elType()` returns the ElementType instance

Examples

```
>>> Formex('4:0123').toMesh().elName()
'quad4'
```

`setNormals (normals=None)`

Set/Remove the normals of the mesh.

Parameters **normals** (float *array_like*) – A float array of shape (ncoords,3) or (nelems,nplex,3). If provided, this will set these normals for use in rendering, overriding the automatically computed ones. If None, this will clear any previously set user normals.

`__getitem__(i)`

Return element *i* of the Mesh.

This allows addressing element *i* of Mesh *M* as *M[i]*.

Parameters ***i*** (*index*) – The index of the element(s) to return. This can be a single element number, a slice, or an array with a list of numbers.

Returns *Coords* – A Coords with a shape (nplex, 3), or if multiple elements are requested, a shape (nelements, nplex, 3), holding the coordinates of all points of the requested elements.

Notes

This is normally used in an expression as `M[i]`, which will return the element `i`. Then `M[i][j]` will return the coordinates of node `j` of element `i`.

`ndim()`

Returns the dimensionality of the global coordinate space. Currently, this always returns 3.

`level()`

Return the level of the elements in the Mesh.

Returns *int* – The dimensionality of the elements: 0 (point), 1(line), 2 (surface), 3 (volume).

`nelems()`

Return the number of elements in the Mesh. This is the first dimension of the `elems` array.

`nplex()`

Return the plexitude of the elements in the Mesh. This is the second dimension of the `elems` array.

`ncoords()`

Return the number of nodes in the Mesh. This is the first dimension of the `coords` array.

`nnodes()`

Return the number of nodes in the Mesh. This is the first dimension of the `coords` array.

`npoints()`

Return the number of nodes in the Mesh. This is the first dimension of the `coords` array.

`shape()`

Return the shape of the `elems` array.

`nedges()`

Return the number of edges.

Returns *int* – The number of rows that would be returned by `getEdges()`, without actually constructing the edges.

Notes

This is the total number of edges for all elements. Edges shared by multiple elements are counted multiple times.

`info()`

Return short info about the Mesh.

Returns *str* – A string with info about the shape of the `coords` and `elems` attributes.

`report (full=True)`

Create a report on the Mesh shape and size.

The report always contains the number of nodes, number of elements, plexitude, dimensionality, element type, bbox and size. If `full==True`(default), it also contains the nodal coordinate list and element connectivity table. Because the latter can be rather bulky, they can be switched off.

Note: NumPy normally limits the printed output. You will have to change numpy settings to actually print the full arrays.

`shallowCopy (prop=None)`

Return a shallow copy.

Parameters `prop` (int *array_like*, optional) – 1-dim int array with non-negative element property numbers.

Returns *Mesh* – A shallow copy of the Mesh, using the same data arrays for `coords` and `elems`. If `prop` was provided, the new Mesh can have other property numbers. This is a convenient method to use the same Mesh with different property attributes.

toFormex()

Convert a Mesh to a Formex.

Returns *Formex* – A Formex equivalent with the calling Mesh. The Formex inherits the element property numbers and `eltype` from the Mesh. Drawing attributes and Fields are not transferred though.

Examples

```
>>> M = Mesh([[0,0,0],[1,0,0]],[[0,1],[1,0]],eltype='line2')
>>> M.toFormex()
Formex([[ [ 0., 0., 0.],
          [ 1., 0., 0.]],
<BLANKLINE>
        [ [ 1., 0., 0.],
          [ 0., 0., 0.]])
```

toMesh()

Convert to a Mesh.

Returns *Mesh* – The Mesh itself. This is provided as a convenience for use in functions that need to work on different Geometry types.

toSurface()

Convert a Mesh to a TriSurface.

Only Meshes of level 2 (surface) and 3 (volume) can be converted to a TriSurface. For a level 3 Mesh, the border Mesh is taken first. A level 2 Mesh is converted to element type ‘tri3’ and then to a TriSurface.

Returns *TriSurface* – A TriSurface corresponding with the input Mesh. If that has `eltype` ‘tri3’, the resulting TriSurface is fully equivalent. Otherwise, a triangular approximation is returned.

Raises `ValueError` – If the Mesh can not be converted to a TriSurface.

toCurve (*connect=False*)

Convert a Mesh to a Curve.

If the element type is one of ‘line*’ types, the Mesh is converted to a Curve. The type of the returned Curve is dependent on the element type of the Mesh:

- ‘line2’: `PolyLine`,
- ‘line3’: `BezierSpline` (degree 2),
- ‘line4’: `BezierSpline` (degree 3)

If `connect` is `False`, this is equivalent to

```
self.toFormex().toCurve()
```

Any other type will raise an exception.

centroids ()

Return the centroids of all elements of the Mesh.

The centroid of an element is the point with coordinates equal to the average of those of all nodes of the element.

Returns *Coords* – A Coords object with shape (*nelems ()*, 3), holding the centroids of all the elements in the Mesh.

Examples

```
>>> rectangle(L=3,W=2,nl=3,nw=2).centroids()
Coords([[ 0.5,  0.5,  0. ],
        [ 1.5,  0.5,  0. ],
        [ 2.5,  0.5,  0. ],
        [ 0.5,  1.5,  0. ],
        [ 1.5,  1.5,  0. ],
        [ 2.5,  1.5,  0. ]])
```

bboxes ()

Returns the bboxes of all elements in the Mesh.

Returns *float array (nelems,2,3)*. – An array with the minimal and maximal values of the coordinates of the nodes of each element, stored along the 1-axis.

getLowerEntities (level=-1, unique=False)

Get the entities of a lower dimensionality.

Parameters

- **level** (*int*) – The *level* of the entities to return. If negative, it is a value relative to the level of the caller. If non-negative, it specifies the absolute level. Thus, for a Mesh with a 3D element type, `getLowerEntities(-1)` returns the faces, while for a 2D element type, it returns the edges. For both meshes however, `getLowerEntities(+1)` returns the edges.
- **unique** (*bool, optional*) – If True, return only the unique entities.

Returns

Connectivity – A Connectivity defining the lower entities of the specified level in terms of the nodes of the Mesh. By default, all entities for all elements are returned and entities shared by multiple elements will appear multiple times. With `unique=True` only the unique ones are returned.

The return value may be an empty table, if the element type does not have the requested entities (e.g. ‘quad4’ Mesh does not have entities of level 3).

If the targeted entity level is outside the range 0..3, the return value is None.

See also:

level () return the dimensionality of the Mesh

connectivity.Connectivity.insertLevel () returns two tables: elems vs. lower entities, lower entities vs. nodes.

Examples

Mesh with one ‘quad4’ element and 4 nodes.

```
>>> M = Mesh(etype='quad4')
```

The element defined in function of the nodes.

```
>>> print(M.elems)
[[0 1 2 3]]
```

The edges of the element defined in function of the nodes.

```
>>> print(M.getLowerEntities(-1))
[[0 1]
 [1 2]
 [2 3]
 [3 0]]
```

And finally, the nodes themselves: not very useful, but works.

```
>>> print(M.getLowerEntities(-2))
[[0]
 [1]
 [2]
 [3]]
```

getElems()

Get the elems table.

Returns *Elems* – The element connectivity table (the *elems* attribute).

Notes

This is deprecated. Use the *elems* attribute instead.

getNodes()

Return the set of unique node numbers in the Mesh.

Returns *int array* – The sorted node numbers that are actually used in the connectivity table. For a compacted Mesh, it is equal to `arange(self.nelems)`.

getPoints()

Return the nodal coordinates of the Mesh.

Returns *Coords* – The coordinates of the nodes that are actually used in the connectivity table. For a compacted Mesh, it is equal to the *coords* attribute.

getEdges()

Return the unique edges of all the elements in the Mesh.

Returns *Elems* – A connectivity table defining the unique element edges in function of the nodes. This is like `self.getLowerEntities(1, unique=True)`, but the result is stored internally in the Mesh object so that it does not need recomputation on a next call.

getFaces()

Return the unique faces of all the elements in the Mesh.

Returns *Elems* – A connectivity table defining all the element faces in function of the nodes. This is like `self.getLowerEntities(2, unique=True)`, but the result is stored internally in the Mesh object so that it does not need recomputation on a next call.

getCells()

Return the cells of the elements.

This is a convenient function to create a table with the element cells. It is equivalent to `self.getLowerEntities(3, unique=True)`, but this also stores the result internally so that future requests can return it without the need for computing it again.

edgeMesh()

Return a Mesh with the unique edges of the elements.

This can only be used with a Mesh of level ≥ 1 .

faceMesh()

Return a Mesh with the unique faces of the elements.

This can only be used with a Mesh of level ≥ 2 .

getElemEdges()

Defines the elements in function of its edges.

Returns *Elms* – A connectivity table with the elements defined in function of the edges.

Notes

As a side effect, this also stores the definition of the edges and the returned element to edge connectivity in the attributes *edges*, resp. *elem_edges*.

getFreeEntities (*level=-1, return_indices=False*)

Return the free entities of the specified level.

Parameters

- **level** (*int*) – The *level* of the entities to return. If negative, it is a value relative to the level of the caller. If non-negative, it specifies the absolute level.
- **return_indices** (*bool*) – If True, also returns an index array (nentities,2) for inverse lookup of the higher entity (column 0) and its local lower entity number (column 1).

Returns *Elms* – A connectivity table with the free entities of the specified level of the Mesh. Free entities are entities that are only connected to a single element.

See also:

getFreeEntitiesMesh() return the free entities as a Mesh

getBorder() return the free entities of the first lower level

Examples

```
>>> M = Formex('3:.12.34').toMesh()
>>> print(M.report())
Mesh: nnodes: 4, nelems: 2, nplex: 3, level: 2, eltype: tri3
  BBox: [ 0.  0.  0.], [ 1.  1.  0.]
  Size: [ 1.  1.  0.]
  Area: 1.0
  Coords: [[ 0.  0.  0.]
            [ 1.  0.  0.]
            [ 0.  1.  0.]
            [ 1.  1.  0.]
```

(continues on next page)

(continued from previous page)

```

Elems: [[0 1 3]
        [3 2 0]]
>>> M.getFreeEntities(1)
Elems([[0, 1],
       [1, 3],
       [3, 2],
       [2, 0]], eltype=Line2)
>>> M.getFreeEntities(1, True) [1]
array([[0, 0],
       [0, 1],
       [1, 0],
       [1, 1]])

```

getFreeEntitiesMesh (*level=-1, compact=True*)

Return a Mesh with lower entities.

Parameters

- **level** (*int*) – The *level* of the entities to return. If negative, it is a value relative to the level of the caller. If non-negative, it specifies the absolute level.
- **compact** (*bool*) – If True (default), the returned Mesh will be compacted. If False, the returned Mesh will contain all the nodes present in the input Mesh.

Returns *Mesh* – A Mesh containing the lower entities of the specified level. If the Mesh has property numbers, the lower entities inherit the property of the element to which they belong.

See also:

[*getFreeEdgesMesh* \(\)](#) return a Mesh with the free entities of the level 1

[*getBorderMesh* \(\)](#) return the free entities Mesh of the first lower level

getFreeEdgesMesh (*compact=True*)

Return a Mesh with the free edges.

Parameters **compact** (*bool*) – If True (default), the returned Mesh will be compacted. If False, the returned Mesh will contain all the nodes present in the input Mesh.

Returns *Mesh* – A Mesh containing the free edges of the input Mesh. If the input Mesh has property numbers, the edge elements inherit the property of the element to which they belong.

See also:

[*getFreeEntitiesMesh* \(\)](#) return the free entities Mesh of any lower level

[*getBorderMesh* \(\)](#) return the free entities Mesh of level -1

getBorder (*return_indices=False*)

Return the border of the Mesh.

Border entities are the free entities of the first lower level.

Parameters **return_indices** (*bool*) – If True, also returns an index array (*nentities,2*) for inverse lookup of the higher entity (column 0) and its local lower entity number (column 1).

Returns *Elems* – A connectivity table with the border entities of the specified level of the Mesh. Free entities are entities that are only connected to a single element.

See also:

`getFreeEntities()` return the free entities of any lower level

`getBorderMesh()` return the border as a Mesh

Notes

This is a convenient shorthand for

```
self.getFreeEntities(level=-1,return_indices=return_indices)
```

getBorderMesh (*compact=True*)

Return a Mesh representing the border.

Parameters **compact** (*bool*) – If True (default), the returned Mesh will be compacted. If False, the returned Mesh will contain all the nodes present in the input Mesh.

Returns *Mesh* – A Mesh containing the border of the input Mesh. The level of the Mesh is one less than that of the input Mesh. If the input Mesh has property numbers, the border elements inherit the property of the element to which they belong.

Notes

This is a convenient shorthand for

```
self.getFreeEntitiesMesh(level=-1,compact=compact)
```

borderMesh (*compact=True*)

Return a Mesh representing the border.

Parameters **compact** (*bool*) – If True (default), the returned Mesh will be compacted. If False, the returned Mesh will contain all the nodes present in the input Mesh.

Returns *Mesh* – A Mesh containing the border of the input Mesh. The level of the Mesh is one less than that of the input Mesh. If the input Mesh has property numbers, the border elements inherit the property of the element to which they belong.

Notes

This is a convenient shorthand for

```
self.getFreeEntitiesMesh(level=-1,compact=compact)
```

getBorderElems ()

Find the elements that are touching the border of the Mesh.

Returns *int array* – A list of the numbers of the elements that fully contain at least one of the elements of the border Mesh. Thus, in a volume Mesh, elements only touching the border by a vertex or an edge are not considered border elements.

getBorderNodes ()

Find the nodes that are on the border of the Mesh.

Returns *int array* – A list of the numbers of the nodes that are on the border of the Mesh.

peel (*nodal=False*)

Remove the border elements from a Mesh.

Parameters `nodal` (*bool*) – If True, all elements connected to a border node are removed. The default will only remove the elements returned by `getBorderElements()`.

Returns *Mesh* – A Mesh with the border elements removed.

connectedTo (*entities, level=0*)

Find the elements connected to specific lower entities.

Parameters

- **entities** (int or int *array_like*) – The indices of the lower entities to which connection should exist.
- **level** (*int*) – The *level* of the entities to which connection should exist. If negative, it is a value relative to the level of the caller. If non-negative, it specifies the absolute level. Default is 0 (nodes).

Returns *int array* – A list of the numbers of the elements that contain at least one of the specified lower entities.

adjacentTo (*elements, level=0*)

Find the elements adjacent to the specified elements.

Adjacent elements are elements that share some lower entity.

Parameters

- **elements** (int or int *array_like*) – Element numbers to find the adjacent elements for.
- **level** (*int*) – The *level* of the entities used to define adjacency. If negative, it is a value relative to the level of the caller. If non-negative, it specifies the absolute level. Default is 0 (nodes).

Returns *int array* – A list of the numbers of all the elements in the Mesh that are adjacent to any of the specified elements.

reachableFrom (*elements, level=0*)

Select the elements reachable from the specified elements.

Elements are reachable if one can travel from one of the origin elements to the target, by only following the specified level of connections.

Parameters

- **elements** (int or int *array_like*) – Element number(s) from where to start the walk.
- **level** (*int*) – The *level* of the entities used to define connections. If negative, it is a value relative to the level of the caller. If non-negative, it specifies the absolute level. Default is 0 (nodes).

Returns *int array* – A list of the numbers of all the elements in the Mesh reachable from any of the specified elements by walking over entities of the specified level. The list will include the original set of elements.

adjacency (*level=0, diflevel=-1*)

Create an element adjacency table.

Two elements are said to be adjacent if they share a lower entity of the specified level.

Parameters

- **level** (*int*) – Hierarchic level of the geometric items connecting two elements: 0 = node, 1 = edge, 2 = face. Only values of a lower hierarchy than the level of the Mesh itself make sense. Default is to consider nodes as the connection between elements.

- **diflevel** (*int, optional*) – If \geq level, and smaller than the level of the Mesh itself, elements that have a connection of this level are removed. Thus, in a Mesh with volume elements, `self.adjacency(0,1)` gives the adjacency of elements by a node but not by an edge.

Returns **adj** (*Adjacency*) – An Adjacency table specifying for each element its neighbours connected by the specified geometrical subitems.

frontWalk (*level=0, startat=0, frontinc=1, partinc=1, maxval=-1*)

Visit all elements using a frontal walk.

In a frontal walk a forward step is executed simultaneously from all the elements in the current front. The elements thus reached become the new front. An element can be reached from the current element if both are connected by a lower entity of the specified level. Default level is ‘point’.

Parameters

- **level** (*int*) – Hierarchy of the geometric items connecting two elements: 0 = node, 1 = edge, 2 = face. Only values of a lower hierarchy than the elements of the Mesh itself make sense. There are no connections on the upper level.
- **startat** (*int or list of ints*) – Initial element number(s) in the front.
- **frontinc** (*int*) – Increment for the front number on each frontal step.
- **partinc** (*int*) – Increment for the front number when the front gets empty and a new part is started.
- **maxval** (*int*) – Maximum frontal value. If negative (default) the walk will continue until all elements have been reached. If non-negative, walking will stop as soon as the frontal value reaches this maximum.

Returns *int array* – An array of ints specifying for each element in which step the element was reached by the walker.

See also:

adjacency.Adjacency.frontWalk()

Examples

```
>>> M = Mesh(etype='quad4').subdivide(5,2)
>>> print(M.frontWalk())
[0 1 2 3 4 1 1 2 3 4]
```

maskedEdgeFrontWalk (*mask=None, startat=0, frontinc=1, partinc=1, maxval=-1*)

Perform a front walk over masked edge connections.

This is like `frontWalk(level=1)`, but allows to specify a mask to select the edges that are used as connectors between elements.

Parameters:

- **mask**: Either None or a boolean array or index flagging the nodes which are to be considered connectors between elements. If None, all nodes are considered connections.

The remainder of the parameters are like in *adjacency.Adjacency.frontWalk()*.

partitionByConnection (*level=0, startat=0, sort='number', nparts=-1*)

Detect the connected parts of a Mesh.

The Mesh is partitioned in parts in which all elements are connected. Two elements are connected if it is possible to draw a continuous line from a point in one element to a point in the other element without leaving the Mesh.

Parameters:

- *sort*: str. Weighted sorting method. It can assume values ‘number’ (default), ‘length’, ‘area’, ‘volume’.
- *nparts*: is the equivalent of parameter *maxval* in *frontWalk()*. Maximum frontal value. If negative (default) the walk will continue until all elements have been reached. If non-negative, walking will stop as soon as the frontal value reaches this maximum.

The remainder of the parameters are like in *frontWalk()*.

The partitioning is returned as a integer array having a value for each element corresponding to the part number it belongs to.

By default the parts are sorted in decreasing order of the number of elements. If you specify *nparts*, you may wish to switch off the sorting by specifying *sort=''*.

splitByConnection (*level=0, startat=0, sort='number'*)

Split the Mesh into connected parts.

The parameters *level* and *startat* are like in *frontWalk()*. The parameter *sort* is like in *partitionByConnection()*.

Returns a list of Meshes that each form a connected part. By default the parts are sorted in decreasing order of the number of elements.

largestByConnection (*level=0*)

Return the largest connected part of the Mesh.

This is equivalent with, but more efficient than

```
self.splitByConnection(level)[0]
```

growSelection (*sel, mode='node', nsteps=1*)

Grow a selection of a surface.

p is a single element number or a list of numbers. The return value is a list of element numbers obtained by growing the front *nsteps* times. The *mode* argument specifies how a single frontal step is done:

- ‘node’ : include all elements that have a node in common,
- ‘edge’ : include all elements that have an edge in common.

partitionByAngle (***kargs*)

Partition a level-2 Mesh by the angle between adjacent elements.

The Mesh is partitioned in parts bounded by the sharp edges in the surface. The arguments and return value are the same as in *trisurface.TriSurface.partitionByAngle()*.

For eltypes other than ‘tri3’, a conversion to ‘tri3’ is done before computing the partitions.

nodeConnections ()

Find and store the elems connected to nodes.

nNodeConnected ()

Find the number of elems connected to nodes.

edgeConnections ()

Find and store the elems connected to edges.

nEdgeConnected ()

Find the number of elems connected to edges.

nodeAdjacency ()

Find the elems adjacent to each elem via one or more nodes.

nNodeAdjacent ()

Find the number of elems which are adjacent by node to each elem.

edgeAdjacency ()

Find the elems adjacent to elems via an edge.

nEdgeAdjacent ()

Find the number of adjacent elems.

nonManifoldNodes ()

Return the non-manifold nodes of a Mesh.

Non-manifold nodes are nodes where subparts of a mesh of level ≥ 2 are connected by a node but not by an edge.

Returns an integer array with a sorted list of non-manifold node numbers. Possibly empty (always if the dimensionality of the Mesh is lower than 2).

nonManifoldEdges ()

Return the non-manifold edges of a Mesh.

Non-manifold edges are edges where subparts of a mesh of level 3 are connected by an edge but not by an face.

Returns an integer array with a sorted list of non-manifold edge numbers. Possibly empty (always if the dimensionality of the Mesh is lower than 3).

As a side effect, this constructs the list of edges in the object. The definition of the nonManifold edges in terms of the nodes can thus be got from

```
self.edges[self.nonManifoldEdges ()]
```

nonManifoldEdgeNodes ()

Return the non-manifold edge nodes of a Mesh.

Non-manifold edges are edges where subparts of a mesh of level 3 are connected by an edge but not by an face.

Returns an integer array with a sorted list of numbers of nodes on the non-manifold edges. Possibly empty (always if the dimensionality of the Mesh is lower than 3).

fuse (parts=None, nodes=None, **kargs)

Fuse the nodes of a Meshes.

Nodes that are within the tolerance limits of each other are merged into a single node.

Parameters:

- *parts*: int *array_like* with length equal to number of elements. If specified, it will be used to split the Mesh into parts (see `splitProp()`) and do the fuse operation per part. Elements for which the value of *nparts* is negative will not be involved in the fuse operations.
- *nodes*: int :term:: a list of node numbers. If specified, only these nodes will be involved in the fuse operation. This option can not be used together with the *parts* option.
- Extra arguments for tuning the fuse operation are passed to the `coords.Coords:fuse()` method.

matchCoords (*coords*, ***kargs*)

Match nodes of coords with nodes of self.

coords can be a Coords or a Mesh object This is a convenience function equivalent to

```
self.coords.match(mesh.coords, **kargs)
```

or

```
self.coords.match(coords, **kargs)
```

See also `coords.Coords.match()`

matchCentroids (*mesh*, ***kargs*)

Match elems of Mesh with elems of self.

self and Mesh are same eltype meshes and are both without duplicates.

Elems are matched by their centroids.

compact (*return_index=False*)

Remove unconnected nodes and renumber the mesh.

Returns a mesh where all nodes that are not used in any element have been removed, and the nodes are renumbered to a compacter scheme.

If return_index is True, also returns an index specifying the index of the new nodes in the old node scheme.

Examples

```
>>> x = Coords([[i] for i in arange(5)])
>>> M = Mesh(x, [[0,2], [1,4], [4,2]])
>>> M, ind = M.compact(True)
>>> print(M.coords)
[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 2.  0.  0.]
 [ 4.  0.  0.]]
>>> print(M.elems)
[[0 2]
 [1 3]
 [3 2]]
>>> M = Mesh(x, [[0,2], [1,3], [3,2]])
>>> M = M.compact()
>>> print(M.coords)
[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 2.  0.  0.]
 [ 3.  0.  0.]]
>>> print(M.elems)
[[0 2]
 [1 3]
 [3 2]]
>>> print(ind)
[0 1 2 4]
>>> M = M.cselect([0,1,2])
>>> M.coords.shape, M.elems.shape
((4, 3), (0, 2))
>>> M = M.compact()
```

(continues on next page)

(continued from previous page)

```
>>> M.coords.shape, M.elems.shape
((0, 3), (0, 2))
```

avgNodes (*nodsel*, *wts=None*)

Create average nodes from the existing nodes of a mesh.

nodsel is a local node selector as in `selectNodes()`. Returns the (weighted) average coordinates of the points in the selector as $(nelems * nnod, 3)$ array of coordinates, where *nnod* is the length of the node selector. *wts* is a 1-D array of weights to be attributed to the points. Its length should be equal to that of *nodsel*.

meanNodes (*nodsel*)

Create nodes from the existing nodes of a mesh.

nodsel is a local node selector as in `selectNodes()`. Returns the mean coordinates of the points in the selector as $(nelems * nnod, 3)$ array of coordinates, where *nnod* is the length of the node selector.

addNodes (*newcoords*, *eltype=None*)

Add new nodes to elements.

newcoords is an $(nelems, nnod, 3)$ or $(nelems * nnod, 3)$ array of coordinates. Each element gets exactly *nnod* extra nodes from this array. The result is a Mesh with plexitude `self.nplex() + nnod`.

addMeanNodes (*nodsel*, *eltype=None*)

Add new nodes to elements by averaging existing ones.

nodsel is a local node selector as in `selectNodes()`. Returns a Mesh where the mean coordinates of the points in the selector are added to each element, thus increasing the plexitude by the length of the items in the selector. The new element type should be set to correct value.

selectNodes (*nodsel*, *eltype=None*)

Return a mesh with subsets of the original nodes.

nodsel is an object that can be converted to a 1-dim or 2-dim array. Examples are a tuple of local node numbers, or a list of such tuples all having the same length. Each row of *nodsel* holds a list of local node numbers that should be retained in the new connectivity table.

hits (*entities*, *level*)

Count the lower entities from a list connected to the elements.

entities: a single number or a list/array of entities *level*: 0 or 1 or 2 if entities are nodes or edges or faces, respectively.

The numbering of the entities corresponds to `self.insertLevel(level)`. Returns an $(nelems,)$ shaped int array with the number of the entities from the list that are contained in each of the elements. This method can be used in selector expressions like:

```
self.select(self.hits(entities, level) > 0)
```

splitRandom (*n*, *compact=True*)

Split a Mesh in *n* parts, distributing the elements randomly.

Returns a list of *n* Mesh objects, constituting together the same Mesh as the original. The elements are randomly distributed over the subMeshes.

By default, the Meshes are compacted. Compaction may be switched off for efficiency reasons.

reverse (*sel=None*)

Return a Mesh where the elements have been reversed.

Reversing an element has the following meaning:

- for 1D elements: reverse the traversal direction,
- for 2D elements: reverse the direction of the positive normal,
- for 3D elements: reverse inside and outside directions of the element's border surface. This also changes the sign of the element's volume.

The `reflect()` method by default calls this method to undo the element reversal caused by the reflection operation.

Parameters:

`-sel`: a selector (index or True/False array)

reflect (`dir=0, pos=0.0, reverse=True, **kargs`)

Reflect the coordinates in one of the coordinate directions.

Parameters:

- `dir`: int: direction of the reflection (default 0)
- `pos`: float: offset of the mirror plane from origin (default 0.0)
- `reverse`: boolean: if True, the `reverse()` method is called after the reflection to undo the element reversal caused by the reflection of its coordinates. This will in most cases have the desired effect. If not however, the user can set this to False to skip the element reversal.

convert (`totype, fuse=False, verbose=False`)

Convert a Mesh to another element type.

Converting a Mesh from one element type to another can only be done if both element types are of the same dimensionality. Thus, 3D elements can only be converted to 3D elements.

The conversion is done by splitting the elements in smaller parts and/or by adding new nodes to the elements.

Not all conversions between elements of the same dimensionality are possible. The possible conversion strategies are implemented in a table. New strategies may be added however.

The return value is a Mesh of the requested element type, representing the same geometry (possibly approximatively) as the original mesh.

If the requested conversion is not implemented, an error is raised.

Warning: Conversion strategies that add new nodes may produce double nodes at the common border of elements. The `fuse()` method can be used to merge such coincident nodes. Specifying `fuse=True` will also enforce the fusing. This option become the default in future.

convertRandom (`choices`)

Convert choosing randomly between choices

Returns a Mesh obtained by converting the current Mesh by a randomly selected method from the available conversion type for the current element type.

subdivide (`*ndiv, **kargs`)

Subdivide the elements of a Mesh.

Note: This only works for some element types: 'line2', 'tri3', 'quad4', 'hex8'.

Parameters:

- *ndiv*: specifies the number (and place) of divisions (seeds) along the edges of the elements. Accepted type and value depend on the element type of the Mesh. Currently implemented:
 - ‘tri3’: *ndiv* is a single int value specifying the number of divisions (of equal size) for each edge.
 - ‘quad4’: *ndiv* is a sequence of two int values *nx,ny*, specifying the number of divisions along the first, resp. second parametric direction of the element
 - ‘hex8’: *ndiv* is a sequence of three int values *nx,ny,nz* specifying the number of divisions along the first, resp. second and the third parametric direction of the element
- *fuse*: bool, if True (default), the resulting Mesh is completely fused. If False, the Mesh is only fused over each individual element of the original Mesh.

Returns a Mesh where each element is replaced by a number of smaller elements of the same type.

Note: This is currently only implemented for Meshes of type ‘tri3’ and ‘quad4’ and ‘hex8’ and for the derived class ‘TriSurface’.

splitDegenerate (*reduce=True, return_indices=False*)

Split a Mesh in non-degenerate and degenerate elements.

Splits a Mesh in non-degenerate elements and degenerate elements, and tries to reduce degenerate elements to lower plexitude elements.

Parameters

- **reduce** (bool or *ElementType* name) – If True, the degenerate elements will be tested against known degeneration patterns, and the matching elements will be transformed to non-degenerate elements of a lower plexitude. If a string, it is an element name and only transforms to this element type will be considered. If False, no reduction of the degenerate elements will be attempted.
- **return_indices** (*bool, optional*) – If True, also returns the element indices in the original Mesh for all of the elements in the derived Meshes.

Returns *ML* (*list of Mesh objects*) – The list of Meshes resulting from the split operation. The first holds the non-degenerate elements of the original Mesh. The last holds the remaining degenerate elements. The intermediate Meshes, if any, hold elements of a lower plexitude than the original.

Warning: The Meshes that hold reduced elements may still contain degenerate elements for the new element type

Examples

```
>>> M = Mesh(np.zeros((4,3)),
...         [[0,0,0],
...          [0,0,1],
...          [0,0,1,2],
...          [0,1,2,3],
...          [1,2,3,3],
...          [2,3,3,3],
...         ], eltype='quad4')
```

(continues on next page)

(continued from previous page)

```

>>> M.elems.listDegenerate()
array([0, 1, 2, 4, 5])
>>> for Mi in M.splitDegenerate(): print(Mi)
Mesh: nnodes: 4, nelems: 1, nplex: 4, level: 2, eltype: quad4
  BBox: [ 0.  0.  0.], [ 0.  0.  0.]
  Size: [ 0.  0.  0.]
  Area: 0.0
Mesh: nnodes: 4, nelems: 5, nplex: 3, level: 2, eltype: tri3
  BBox: [ 0.  0.  0.], [ 0.  0.  0.]
  Size: [ 0.  0.  0.]
  Area: 0.0
>>> conn, ind = M.splitDegenerate(return_indices=True)
>>> print(ind[0], ind[1])
[3] [0 1 2 5 4]
>>> print(conn[1].elems)
[[0 0 0]
 [0 0 1]
 [0 1 2]
 [2 3 3]
 [1 2 3]]

```

removeDegenerate()

Remove the degenerate elements from a Mesh.

Returns *Mesh* – A Mesh with all degenerate elements removed.

removeDuplicate (*permutations='all'*)

Remove the duplicate elements from a Mesh.

Duplicate elements are elements that consist of the same nodes.

Parameters

- **permutations** (*str*) – Defines which permutations of the nodes are allowed while still considering the elements duplicates. Possible values are:
 - **'none'** (-) – must have the same value at every position in order to be considered duplicates;
 - **'roll'** (-) – each other by rolling are considered equal;
 - **'all'** (-) – a duplicate element. This is the default.

Returns *Mesh* – A Mesh with all duplicate elements removed.

renumber (*order='elems'*)

Renumber the nodes of a Mesh in the specified order.

Parameters **order** (int *array_like* or *str*) – If an array, it is an index with length equal to the number of nodes. It should be a permutation of `arange(self.nnodes())`. The index specifies the node number that should come at this position. Thus, the order values are the old node numbers on the new node number positions.

`order` can also be a predefined string that will generate the node index automatically:

- **'elems'**: the nodes are number in order of their appearance in the Mesh connectivity.
- **'random'**: the nodes are numbered randomly.
- **'front'**: the nodes are numbered in order of their frontwalk.

Returns *Mesh* – A Mesh equivalent with the input, but with the nodes numbered differently.

reorder (*order='nodes'*)

Reorder the elements of a Mesh.

Parameters **order** (*array_like* or str) – If an array, it is a permutation of the numbers in `range(self.nelems())`, specifying the requested order of the elements.

`order` can also be one of the following predefined strings:

- 'nodes': order the elements in increasing node number order.
- 'random': number the elements in a random order.
- 'reverse': number the elements in reverse order.

Returns *Mesh* – A Mesh equivalent with `self` but with the elements ordered as specified.

connectedElements (*startat, mask, level=0*)

Return the elements reachable from `startat`.

Finds the elements which can be reached from `startat` by walking along a mask (a subset of elements). Walking is possible over nodes, edges or faces, as specified in `level`.

Parameters

- **startat** (int or *array_like*, int.) – The starting element number(s).
- **level** (*int*) – Specifies how elements can be reached: via node (0), edge (1) or face (2).
- **mask** (*array_like*, bool or int.) – Flags the elements that are considered walkable. It is an int array with the walkable element numbers, or a bool array flagging these elements with a value True.

connect (*coordslist, div=1, degree=1, loop=False, eltype=None*)

Connect a sequence of topologically congruent Meshes into a hypermesh.

Parameters:

- *coordslist*: either a list of Coords objects, or a list of Mesh objects or a single Mesh object.

If Mesh objects are given, they should (all) have the same element type as *self*. Their connectivity tables will not be used though. They will only serve to construct a list of Coords objects by taking the *coords* attribute of each of the Meshes. If only a single Mesh was specified, *self.coords* will be added as the first Coords object in the list.

All Coords objects in the *coordslist* (either specified or constructed from the Mesh objects), should have the exact same shape as *self.coords*. The number of Coords items in the list should be a multiple of *degree*, plus 1.

Each of the Coords in the final *coordslist* is combined with the connectivity table, element type and property numbers of *self* to produce a list of topologically congruent meshes. The return value is the hypermesh obtained by connecting each consecutive slice of (*degree*+1) of these meshes. The hypermesh has a dimensionality that is one higher than the original Mesh (i.e. points become lines, lines become surfaces, surfaces become volumes). The resulting elements will be of the given *degree* in the direction of the connection.

Notice that unless a single Mesh was specified as *coordslist*, the coords of *self* are not used. In many cases however *self* or *self.coords* will be one of the items in the specified *coordslist*.

- *degree*: degree of the connection. Currently only degree 1 and 2 are supported.
 - If degree is 1, every Coords from the *coordslist* is connected with hyperelements of a linear degree in the connection direction.

- If degree is 2, quadratic hyperelements are created from one Coords item and the next two in the list. Note that all Coords items should contain the same number of nodes, even for higher order elements where the intermediate planes contain less nodes.

Currently, degree=2 is not allowed when *coordslist* is specified as a single Mesh.

- *loop*: if True, the connections with loop around the list and connect back to the first. This is accomplished by adding the first Coords item back at the end of the list.
- *div*: This should only be used for degree==1.

With this parameter the generated connections can be further subdivided along the connection direction. *div* is either a single input for *smartSeed()*, or a list thereof. In the latter case, the length of the list should be one less than the length of the *coordslist*. Each pair of consecutive items from the coordinate list will be connected using the seeds generated by the corresponding value from *div*, passed to *smartSeed()*. Notice that if seed values are specified directly as a list of floats, the list should start with a value 0.0 and end with 1.0.

- *eltype*: the element type of the constructed hypermesh. Normally, this is set automatically from the base element type and the connection degree. If a different element type is specified, a final conversion to the requested element type is attempted.

extrude (*div*, *dir*=0, *length*=1.0, *degree*=1, *eltype*=None)

Extrude a Mesh along a straight line.

The Mesh is extruded over a given length in the given direction.

Parameters

- **div** (*smartseed*) – Specifies how the extruded direction will be subdivided in elements. It can be anything that is acceptable as input for *smartSeed()*.
- **dir** (int (0,1,2) or float *array_like* (3,)) – The direction of the extrusion: either a global axis number or a direction vector.
- **length** (*float*) – The length of the extrusion, measured along the direction *dir*.

Returns *Mesh* – A Mesh obtained by extruding the input Mesh over the given *length* in direction *dir*, subdividing this length according to the seeds generated by *smartSeed(div)*.

Examples

```
>>> M = Mesh(Formex(origin())).extrude(3,0,3)
>>> print(M)
Mesh: nnodes: 4, nelems: 3, nplex: 2, level: 1, eltype: line2
  BBox: [ 0.  0.  0.], [ 3.  0.  0.]
  Size: [ 3.  0.  0.]
  Length: 3.0
```

revolve (*n*, *axis*=0, *angle*=360.0, *around*=None, *loop*=False, *eltype*=None)

Revolve a Mesh around an axis.

Returns a new Mesh obtained by revolving the given Mesh over an angle around an axis in *n* steps, while extruding the mesh from one step to the next. This extrudes points into lines, lines into surfaces and surfaces into volumes.

sweep (*path*, *eltype*=None, ***kargs*)

Sweep a mesh along a path, creating an extrusion

Parameters:

- *path*: Curve object. The path over which to sweep the Mesh.
- *eltype*: string. Name of the element type on the returned Meshes.
- ***kargs*: keyword arguments that are passed to `curve.Curve.sweep2()`, with the same meaning. Usually, you will need to at least set the *normal* parameter.

Returns a Mesh obtained by sweeping the given Mesh over a path. The returned Mesh has double plexitude of the original. If *path* is a closed Curve connect back to the first.

This operation is similar to the `extrude()` method, but the path can be any 3D curve.

smooth (*iterations=1, lamb=0.5, k=0.1, edg=True, exclnod=[], exclelem=[], weight=None*)

Return a smoothed mesh.

Smoothing algorithm based on lowpass filters.

If *edg* is True, the algorithm tries to smooth the outer border of the mesh separately to reduce mesh shrinkage.

Higher values of *k* can reduce shrinkage even more (up to a point where the mesh expands), but will result in less smoothing per iteration.

- *exclnod*: It contains a list of node indices to exclude from the smoothing. If *exclnod* is 'border', all nodes on the border of the mesh will be unchanged, and the smoothing will only act inside. If *exclnod* is 'inner', only the nodes on the border of the mesh will take part to the smoothing.
- *exclelem*: It contains a list of elements to exclude from the smoothing. The nodes of these elements will not take part to the smoothing. If *exclnod* and *exclelem* are used at the same time the union of them will be excluded from smoothing.

-weight [it is a string that can assume 2 values *inversedistance* and] *distance*. It allows to specify the weight of the adjacent points according to their distance to the point

classmethod concatenate (*meshes, fuse=True, **kargs*)

Concatenate a list of meshes of the same plexitude and eltype

All Meshes in the list should have the same plexitude. Meshes with plexitude are ignored though, to allow empty Meshes to be added in.

Merging of the nodes can be tuned by specifying extra arguments that will be passed to `coords.Coords.fuse()`.

If any of the meshes has property numbers, the resulting mesh will inherit the properties. In that case, any meshes without properties will be assigned property 0. If all meshes are without properties, so will be the result.

This is a class method, and should be invoked as follows:

```
Mesh.concatenate([mesh0, mesh1, mesh2])
```

test (*nodes='all', dir=0, min=None, max=None, atol=0.0*)

Flag elements having nodal coordinates between min and max.

This function is very convenient in clipping a Mesh in a specified direction. It returns a 1D integer array flagging (with a value 1 or True) the elements having nodal coordinates in the required range. Use `where(result)` to get a list of element numbers passing the test. Or directly use `clip()` or `cclip()` to create the clipped Mesh

The test plane can be defined in two ways, depending on the value of *dir*. If *dir* == 0, 1 or 2, it specifies a global axis and min and max are the minimum and maximum values for the coordinates along that axis. Default is the 0 (or x) direction.

Else, `dir` should be compatible with a (3,) shaped array and specifies the direction of the normal on the planes. In this case, `min` and `max` are points and should also evaluate to (3,) shaped arrays.

`nodes` specifies which nodes are taken into account in the comparisons. It should be one of the following:

- a single (integer) point number (< the number of points in the Formex)
- a list of point numbers
- one of the special strings: 'all', 'any', 'none'

The default ('all') will flag all the elements that have all their nodes between the planes $x=\min$ and $x=\max$, i.e. the elements that fall completely between these planes. One of the two clipping planes may be left unspecified.

clipAtPlane (*p, n, nodes='any', side='+'*)

Return the Mesh clipped at plane (p,n).

This is a convenience function returning the part of the Mesh at one side of the plane (p,n)

intersectionWithLines (*approximated=True, **kargs*)

Return the intersections of a level-2 Mesh with lines.

The Mesh is intersected with lines. The arguments and return values are the same as in `trisurface.TriSurface.intersectionWithLines()`, except for the *approximated*.

For a Mesh with eltype 'tri3', the intersections are exact. For other eltypes, if *approximated* is True a conversion to 'tri3' is done before computing the intersections. This may produce an exact result, an approximate result or no result (if the conversion fails). Of course the user can create his own approximation to a 'tri3' surface first, before calling this method.

levelVolumes ()

Return the level volumes of all elements in a Mesh.

The level volume of an element is defined as:

- the length of the element if the Mesh is of level 1,
- the area of the element if the Mesh is of level 2,
- the (signed) volume of the element if the Mesh is of level 3.

The level volumes can be computed directly for Meshes of eltypes 'line2', 'tri3' and 'tet4' and will produce accurate results. All other Mesh types are converted to one of these before computing the level volumes. Conversion may result in approximation of the results. If conversion can not be performed, None is returned.

If successful, returns an (nelems,) float array with the level volumes of the elements. Returns None if the Mesh level is 0, or the conversion to the level's base element was unsuccessful.

Note that for level-3 Meshes, negative volumes will be returned for elements having a reversed node ordering.

lengths ()

Return the length of all elements in a level-1 Mesh.

For a Mesh with eltype 'line2', the lengths are exact. For other eltypes, a conversion to 'line2' is done before computing the lengths. This may produce an exact result, an approximated result or no result (if the conversion fails).

If successful, returns an (nelems,) float array with the lengths. Returns None if the Mesh level is not 1, or the conversion to 'line2' does not succeed.

areas ()

Return the area of all elements in a level-2 Mesh.

For a Mesh with eltype 'tri3', the areas are exact. For other eltypes, a conversion to 'tri3' is done before computing the areas. This may produce an exact result, an approximate result or no result (if the conversion fails).

If succesful, returns an (nelems,) float array with the areas. Returns None if the Mesh level is not 2, or the conversion to 'tri3' does not succeed.

volumes ()

Return the signed volume of all the mesh elements

For a 'tet4' tetraeder Mesh, the volume of the elements is calculated as $1/3 * \text{surface of base} * \text{height}$.

For other Mesh types the volumes are calculated by first splitting the elements into tetraeder elements.

The return value is an array of float values with length equal to the number of elements. If the Mesh conversion to tetraeder does not succeed, the return value is None.

length ()

Return the total length of a Mesh.

Returns the sum of `self.lengths()`, or 0.0 if the `self.lengths()` returned None.

area ()

Return the total area of a Mesh.

Returns the sum of `self.areas()`, or 0.0 if the `self.areas()` returned None.

volume ()

Return the total volume of a Mesh.

For a Mesh of level < 3, a value 0.0 is returned. For a Mesh of level 3, the volume is computed by converting its border to a surface and taking the volume inside that surface. It is equivalent with

```
self.toSurface().volume()
```

This is far more efficient than `self.volumes().sum()`.

fixVolumes ()

Reverse the elements with negative volume.

Elements with negative volume may result from incorrect local node numbering. This method will reverse all elements in a Mesh of dimensionality 3, provide the volumes of these elements can be computed.

Functions defined in module mesh

`mesh.mergeNodes (nodes, fuse=True, **kargs)`

Merge all the nodes of a list of node sets.

Merging the nodes creates a single Coords object containing all nodes, and the indices to find the points of the original node sets in the merged set.

Parameters:

- *nodes*: a list of Coords objects, all having the same shape, except possibly for their first dimension
- *fuse*: if True (default), coincident (or very close) points will be fused to a single point
- ***kargs*: keyword arguments that are passed to the fuse operation

Returns:

- a Coords with the coordinates of all (unique) nodes,
- a list of indices translating the old node numbers to the new. These numbers refer to the serialized Coords.

The merging operation can be tuned by specifying extra arguments that will be passed to `coords.Coords.fuse()`.

`mesh.mergeMeshes(meshes, fuse=True, **kargs)`

Merge all the nodes of a list of Meshes.

Each item in meshes is a Mesh instance. The return value is a tuple with:

- the coordinates of all unique nodes,
- a list of elems corresponding to the input list, but with numbers referring to the new coordinates.

The merging operation can be tuned by specifying extra arguments that will be passed to `coords.Coords.fuse()`. Setting `fuse=False` will merely concatenate all the `mesh.coords`, but not fuse them.

`mesh.line2_wts(seed0)`

Create weights for line2 subdivision.

Parameters `seed0` (*int or list of floats*) – The subdivisions along the first parametric direction of the element. If an int, the subdivisions will be equally spaced between 0 and 1

Examples

```
>>> line2_wts(4)
array([[ 0.  ,  1.  ],
       [ 0.25,  0.75],
       [ 0.5  ,  0.5  ],
       [ 0.75,  0.25],
       [ 1.  ,  0.  ]])
```

`mesh.quad4_wts(seed0, seed1)`

Create weights for quad4 subdivision.

Parameters:

- ‘seed0’: int or list of floats . It specifies divisions along the first parametric direction of the element
- ‘seed1’: int or list of floats . It specifies divisions along the second parametric direction of the element

If these parameters are integer values the divisions will be equally spaced between 0 and 1

This is equivalent with `~arraytools.gridpoints(seed0, seed1)`.

`mesh.quad4_els(nx, ny)`

Quad4 element connectivity for a regular stack of nx,ny elements.

The node numbers vary first in the x, then in the y direction.

`mesh.quad9_wts(seed0, seed1)`

Create weights for quad4 subdivision.

Parameters:

- ‘seed0’: int or list of floats . It specifies divisions along the first parametric direction of the element
- ‘seed1’: int or list of floats . It specifies divisions along the second parametric direction of the element

If these parameters are integer values the divisions will be equally spaced between 0 and 1

This is equivalent with `~arraytools.gridpoints(seed0, seed1)`.

`mesh.quad9_els(nx, ny)`

Quad4 element connectivity for a regular stack of nx,ny elements.

The node numbers vary first in the x, then in the y direction.

`mesh.quadgrid` (*seed0, seed1, roll=0*)

Create a quadrilateral mesh of unit size with the specified seeds.

Parameters:

- *seed0*, 'seed1': seeds for the elements along the parametric directions 0 and 1. Each can be one of the following:
 - an integer number, specifying the number of equally sized elements along that direction,
 - a tuple (n,) or (n,e0) or (n,e0,e1), to be used as parameters in the `mesh.seed()` function,
 - a list of monotonously increasing float values in the range 0.0 to 1.0, specifying the relative positions of the nodes. Normally, the first and last values of the seeds are 0. and 1., leading to a unit square grid.

The node and element numbers vary first in the x, then in the y direction.

`mesh.hex8_wts` (*seed0, seed1, seed2*)

Create weights for hex8 subdivision.

Parameters:

- 'seed0': int or list of floats . It specifies divisions along the first parametric direction of the element
- 'seed1': int or list of floats . It specifies divisions along the second parametric direction of the element
- 'seed2': int or list of floats . It specifies divisions along the t parametric direction of the element

If these parameters are integer values the divisions will be equally spaced between 0 and 1

`mesh.hex8_els` (*nx, ny, nz*)

Create connectivity table for hex8 subdivision.

`mesh.rectangle` (*L=1.0, W=1.0, nl=1, nw=1*)

Create a plane rectangular mesh of quad4 elements.

Parameters: - *L*, 'W': length,width of the rectangle.

`mesh.rectangleWithHole` (*L, W, r, nr, nl, nw=None, e0=0.0, eltype='quad4'*)

Create a quarter of rectangle with a central circular hole.

Parameters:

- *L*: float. Length of the (quarter) rectangle
- *W*: float. Width of the (quarter) rectangle
- *r*: float. Radius of the hole
- *nr*: integer. Number of elements over radial direction
- *nl*: integer. Number of elements over tangential direction along *L*
- *nw*: integer. Number of elements over tangential direction along *W*. If None (default), it will be set equal to *nl*.
- *e0*: float. Concentration factor for elements in the radial direction

Returns a Mesh

`mesh.quadrilateral` (*x, n1, n2*)

Create a quadrilateral mesh

Parameters:

- *x*: Coords(4,3): Four corners of the mesh, in anti-clockwise order.
- *n1*: number of elements along sides x_0-x_1 and x_2-x_3
- *n2*: number of elements along sides x_1-x_2 and x_3-x_4

Returns a Mesh of quads filling the quadrilateral defined by the four points *x*.

`mesh.continuousCurves (c0, c1)`

Make sure the two curves are continuous.

Ensures that the end point of curve *c0* and the start point of curve *c1* are coincident. This is done by replacing these two points with their mean value.

`mesh.triangleQuadMesh (P0, C0, n, P0wgt=1.0)`

Create a quad Mesh over a triangular region

The triangle can have a single non-straight edge. The domain is described by a curve and a point. The straight lines between the curve ends and the point are the other two sides.

Parameters:

- *P0*: a point
- *C*: a curve
- *ndiv*: a tuple of 3 int's. The quad kernel near the point will have n_0*n_1 elements (n_0 to the start of the curve, n_1 to the end. The zone near the curve has n_0+n_1 elements along the curve, and n_2 elements perpendicular to the curve.

`mesh.quarterCircle (n1, n2)`

Create a mesh of quadrilaterals filling a quarter circle.

Parameters

- **n1** (*int*) – Number of elements along the edges 01 and 23
- **n2** (*int*) – Number of elements along the edges 12 and 30

Returns *Mesh* – A ‘quad4’ Mesh filling a quarter circle with radius 1 and center at the origin, in the first quadrant of the axes.

Notes

The quarter circle mesh has a kernel of n_1*n_1 cells, and two border meshes of n_1*n_2 cells. The resulting mesh has n_1+n_2 cells in radial direction and $2*n_1$ cells in tangential direction (in the border mesh).

`mesh.wedge6_roll (elems)`

Roll wedge6 elems to make the lowest node of bottom plane the first

This is a helper function for the `wedge6_tet4()` conversion.

`mesh.wedge6_tet4 (M)`

Convert a Mesh from wedge6 to tet4

This converts a ‘wedge6’ Mesh to ‘tet4’, by replacing each wedge element with three tets. The conversion ensures that the subdivision of the wedge elements are compatible in the common quad faces of any two wedge elements.

Parameters *M* (*Mesh*) – A Mesh of eltype ‘wedge6’.

Returns *Mesh* – A Mesh of eltype ‘tet4’ representing the same domain as the input Mesh. The nodes are the same as those of the input Mesh. The number of elements is three times that of the input Mesh. The order of numbering of the elements is dependent on the conversion algorithm.

6.2.2 trisurface — Operations on triangulated surfaces.

A triangulated surface is a surface consisting solely of triangles. Any surface in space, no matter how complex, can be approximated with a triangulated surface.

Classes defined in module trisurface

class trisurface.**TriSurface** (**args, **kargs*)

A class representing a triangulated 3D surface.

A triangulated surface is a surface consisting of a collection of triangles. The TriSurface is subclassed from Mesh with a fixed plexitude 3. The surface contains *ntri* triangles and *nedg* edges. Each triangle has 3 vertices with 3 coordinates. The total number of vertices is *ncoords*. The TriSurface can be initialized from one of the following sets of data:

- an (ntri,3,3) shaped array of floats
- a Formex with plexitude 3
- a Mesh with plexitude 3
- an (ncoords,3) float array of vertex coordinates and an (ntri,3) integer array of vertex numbers
- an (ncoords,3) float array of vertex coordinates, an (nedg,2) integer array of vertex numbers, an (ntri,3) integer array of edges numbers.

Additionally, a keyword argument `prop` may be provided to specify property values, as in Mesh.

nedges ()

Return the number of edges of the TriSurface.

nfaces ()

Return the number of faces of the TriSurface.

vertices ()

Return the coordinates of the nodes of the TriSurface.

shape ()

Return the number of points, edges, faces of the TriSurface.

getElemEdges ()

Get the faces' edge numbers.

setCoords (*coords*)

Change the coords.

setElems (*elems*)

Change the elems.

setEdgesAndFaces (*edges, faces*)

Change the edges and faces.

append (*S*)

Merge another surface with self.

This just merges the data sets, and does not check whether the surfaces intersect or are connected! This is intended mostly for use inside higher level functions.

classmethod read (*fn, ftype=None*)

Read a surface from file.

If no file type is specified, it is derived from the filename extension. Currently supported file types:

- .off
- .gts
- .stl (ASCII or BINARY)
- .neu (Gambit Neutral)
- .smesh (Tetgen)

Compressed (gzip or bzip2) files are also supported. Their names should be the normal filename with '.gz' or '.bz2' appended. These files are uncompressed on the fly during the reading and the uncompressed versions are deleted after reading.

The file type can be specified explicitly to handle file names where the extension does not directly specify the file type.

write (*fname, ftype=None, color=None*)

Write the surface to file.

If no filetype is given, it is deduced from the filename extension. If the filename has no extension, the 'off' file type is used. For a file with extension 'stl', the ftype may be 'stla' or 'stlb' to force ascii or binary STL format. The color is only useful for 'stlb' format.

avgVertexNormals ()

Compute the average normals at the vertices.

areaNormals ()

Compute the area and normal vectors of the surface triangles.

The normal vectors are normalized. The area is always positive.

The values are returned and saved in the object.

areas ()

Return the areas of all facets

volume ()

Return the enclosed volume of the surface.

This will only be correct if the surface is a closed manifold.

volumeInertia (*density=1.0*)

Return the inertia properties of the enclosed volume of the surface.

The surface should be a closed manifold and is supposed to be the border of a volume of constant density 1.

Returns an *inertia.Inertia* instance with attributes

- *mass*: the total mass (float)
- *ctr*: the center of mass: float (3,)
- *tensor*: the inertia tensor in the central axes: shape (3,3)

This will only be correct if the surface is a closed manifold.

See *inertia()* for the inertia of the surface.

Example:

```
>>> from pyformex.simple import sphere
>>> S = sphere(8)
>>> I = S.volumeInertia()
>>> print(I.mass) # doctest: +ELLIPSIS
```

(continues on next page)

(continued from previous page)

```

4.1526...
>>> print(I.ctr)
[ 0.  0.  0.]
>>> print(I.tensor)
[[ 1.65  0.  -0. ]
 [ 0.   1.65 -0. ]
 [-0.  -0.   1.65]]

```

inertia (*volume=False, density=1.0*)

Return inertia related quantities of the surface.

This computes the inertia properties of the centroids of the triangles, using the triangle area as a weight. The result is therefore different from `self.coords.inertia()` and usually better suited for the surface, especially if the triangle areas differ a lot.

Returns a tuple with the center of gravity, the principal axes of inertia, the principal moments of inertia and the inertia tensor.

See also `volumeInertia()`.

curvature (*neighbours=1*)

Return the curvature parameters at the nodes.

This uses the nodes that are connected to the node via a shortest path of ‘neighbours’ edges. Eight values are returned: the Gaussian and mean curvature, the shape index, the curvedness, the principal curvatures and the principal directions.

surfaceType ()

Check whether the TriSurface is a manifold and if it’s closed.

borderEdges ()

Detect the border elements of TriSurface.

The border elements are the edges having less than 2 connected elements. Returns True where edge is on the border.

borderEdgeNrs ()

Returns the numbers of the border edges.

borderNodeNrs ()

Detect the border nodes of TriSurface.

The border nodes are the vertices belonging to the border edges. Returns a list of vertex numbers.

isManifold ()

Check whether the TriSurface is a manifold.

A surface is a manifold if a small sphere exists that cuts the surface to a surface that can continuously be deformed to an open disk.

nonManifoldEdges ()

Return the non-manifold edges.

Non-manifold edges are edges having more than two triangles connected to them.

Returns the indices of the non-manifold edges in a TriSurface.

nonManifoldEdgesFaces ()

Return the non-manifold edges and faces.

Non-manifold edges are edges that are connected to more than two faces.

Returns

- **edges** (*list of int*) – The list of non-manifold edges.
- **faces** (*list of int*) – The list of faces connected to any of the non-manifold edges.

isClosedManifold()

Check whether the TriSurface is a closed manifold.

isConvexManifold()

Check whether the TriSurface is a convex manifold.

removeNonManifold()

Remove the non-manifold edges.

Removes the non-manifold edges by iteratively applying two `removeDuplicate()` and `collapseEdge()` until no edge has more than two connected triangles.

Returns the reduced surface.

checkBorder()

Return the border of TriSurface.

Returns a list of connectivity tables. Each table holds the subsequent line segments of one continuous contour of the border of the surface.

border (*compact=True*)

Return the border(s) of TriSurface.

The complete border of the surface is returned as a list of plex-2 Meshes. Each Mesh constitutes a continuous part of the border. By default, the Meshes are compacted. Setting `compact=False` will return all Meshes with the full surface coordinate sets. This is useful for filling the border and adding to the surface.

fillBorder (*method='radial', dir=None, compact=True*)

Fill the border areas of a surface to make it closed.

Returns a list of surfaces, each of which fills a singly connected part of the border of the input surface. Adding these surfaces to the original will create a closed surface. The surfaces will have property values set above those used in the parent surface. If the surface is already closed, an empty list is returned.

There are three methods: 'radial', 'planar' and 'border', corresponding to the methods of the `fillBorder()` function.

close (*method='radial', dir=None*)

This method needs documentation!!!!

edgeCosAngles (*return_mask=False*)

Return the cos of the angles over all edges.

The surface should be a manifold (max. 2 elements per edge). Edges adjacent to only one element get `cosangles = 1.0`. If `return_mask == True`, a second return value is a boolean array with the edges that connect two faces.

As a side effect, this method also sets the `area`, `normals`, `elem_edges` and `edges` attributes.

edgeAngles ()

Return the angles over all edges (in degrees). It is the angle (0 to 180) between 2 face normals.

edgeSignedAngles (*return_mask=False*)

Return the signed angles over all edges (in degrees). It is the angle (-180 to 180) between 2 face normals.

Positive/negative angles are associated to convexity/concavity at that edge. The border edges attached to one triangle have angle 0. NB: The sign of the angle is relevant if the surface has fixed normals. Should this check be done?

edgeLengths ()

Returns the lengths of all edges

Returns an array with the length of all the edges in the surface. As a side effect, this stores the connectivities of the edges to nodes and the elements to edges in the attributes `edges`, resp. `elem_edges`.

perimeters ()

Compute the perimeters of all triangles.

quality ()

Compute a quality measure for the triangle shapes.

The quality of a triangle is defined as the ratio of the square root of its surface area to its perimeter relative to this same ratio for an equilateral triangle with the same area. The quality is then one for an equilateral triangle and tends to zero for a very stretched triangle.

aspectRatio ()

Return the aspect ratio of the triangles of the surface.

The aspect ratio of a triangle is the ratio of the longest edge over the smallest altitude of the triangle.

Equilateral triangles have the smallest edge ratio (2 over square root 3).

smallestAltitude ()

Return the smallest altitude of the triangles of the surface.

longestEdge ()

Return the longest edge of the triangles of the surface.

shortestEdge ()

Return the shortest edge of the triangles of the surface.

stats ()

Return a text with full statistics.

distanceOfPoints (*X*, *return_points=False*)

Find the distances of points *X* to the TriSurface.

The distance of a point is either: - the closest perpendicular distance to the facets; - the closest perpendicular distance to the edges; - the closest distance to the vertices.

X is a (*nX*,3) shaped array of points. If `return_points = True`, a second value is returned: an array with the closest (foot)points matching *X*.

degenerate ()

Return a list of the degenerate faces according to area and normals.

A face is degenerate if its surface is less or equal to zero or the normal has a nan.

Returns the list of degenerate element numbers in a sorted array.

removeDegenerate (*compact=False*)

Remove the degenerate elements from a TriSurface.

Returns a TriSurface with all degenerate elements removed. By default, the coords set is unaltered and will still contain all points, even ones that are no longer connected to any element. To reduce the coordinate set, set the `compact` argument to `True` or use the `compact ()` method afterwards.

collapseEdge (*edg*)

Collapse an edge in a TriSurface.

Collapsing an edge removes all the triangles connected to the edge and replaces the two vertices by a single one, placed at the center of the edge. Triangles only having one of the edge vertices, will all be connected to the new vertex.

Parameters:

- *edg*: the index of the edge to be removed. This is an index in the array of edges as returned by `getElemEdges()`.

Returns a TriSurface with the specified edge number removed.

offset (*distance=1.0*)

Offset a surface with a certain distance.

All the nodes of the surface are translated over a specified distance along their normal vector.

dualMesh (*method='median'*)

Return the dual mesh of a compacted triangulated surface.

It creates a new triangular mesh where all triangles with prop *p* represent the dual mesh region around the original surface node *p*. For more info, see <http://users.led-inc.eu/~phk/mesh-dualmesh.html>.

- *method*: 'median' or 'voronoi'.

Returns:

- *method* = 'median': the Median dual mesh and the area of the region around each node. The sum of the node-based areas is equal to the original surface area.
- *method* = 'voronoi': the Voronoi polyeders and a None.

featureEdges (*angle=60.0*)

Return the feature edges of the surface.

Feature edges are edges that are prominent features of the geometry. They are either border edges or edges where the normals on the two adjacent triangles differ more than a given angle. The non feature edges then represent edges on a rather smooth surface.

Parameters:

- *angle*: The angle by which the normals on adjacent triangles should differ in order for the edge to be marked as a feature.

Returns a boolean array with shape (nedg,) where the feature angles are marked with True.

Note: As a side effect, this also sets the *elem_edges* and *edges* attributes, which can be used to get the edge data with the same numbering as used in the returned mask. Thus, the following constructs a Mesh with the feature edges of a surface *S*:

```
p = S.featureEdges()
Mesh(S.coords, S.edges[p])
```

partitionByAngle (*angle=60.0, sort='number'*)

Partition the surface by splitting it at sharp edges.

The surface is partitioned in parts in which all elements can be reach without ever crossing a sharp edge angle. More precisely, any two elements that can be connected by a line not crossing an edge between two elements having their normals differ more than angle (in degrees), will belong to the same part.

The partitioning is returned as an integer array specifying the part number for each triangle.

By default the parts are assigned property numbers in decreasing order of the number of triangles in the part. Setting the sort argument to 'area' will sort the parts according to decreasing area. Any other value will return the parts unsorted.

Beware that the existence of degenerate elements may cause unexpected results. If unsure, use the `removeDegenerate()` method first to remove those elements.

cutWithPlane1 (*p*, *n*, *side=""*, *return_intersection=False*, *atol=0.0*)

Cut a surface with a plane.

Cuts the surface with a plane defined by a point *p* and normal *n*.

Parameters:

- *p*: float, shape (3,): a point in the cutting plane
- *n*: float, shape (3,): the normal vector to the plane
- *side*: ‘’, ‘+’ or ‘-’: selector of the returned parts. Default is to return a tuple of two surfaces, with the parts at the positive, resp. negative side of the plane as defined by the normal vector. If a ‘+’ or ‘-’ is specified, only the corresponding part is returned.

Returns:

A tuple of two TriSurfaces, or a single TriSurface, depending on the value of *side*. The returned surfaces will have their normals fixed wherever possible. Property values will be set containing the triangle number of the original surface from which the elements resulted.

cutWithPlane (**args*, ***kwargs*)

Cut a surface with a plane or a set of planes.

Cuts the surface with one or more plane and returns either one side or both.

Parameters:

- *p*, ‘*n*’: a point and normal vector defining the cutting plane. *p* and *n* can be sequences of points and vector, allowing to cut with multiple planes. Both *p* and *n* have shape (3) or (npoints,3).

The parameters are the same as in `Formex.CutWithPlane()`. The returned surface will have its normals fixed wherever possible.

intersectionWithPlane (*p*, *n*, *atol=0.0*, *sort='number'*)

Return the intersection lines with plane (*p*,*n*).

Returns a plex-2 mesh with the line segments obtained by cutting all triangles of the surface with the plane (*p*,*n*) *p* is a point specified by 3 coordinates. *n* is the normal vector to a plane, specified by 3 components.

The return value is a plex-2 Mesh where the line segments defining the intersection are sorted to form continuous lines. The Mesh has property numbers such that all segments forming a single continuous part have the same property value.

By default the parts are assigned property numbers in decreasing order of the number of line segments in the part. Setting the sort argument to ‘distance’ will sort the parts according to increasing distance from the point *p*.

The `splitProp()` method can be used to get a list of Meshes.

slice (*dir=0*, *nplanes=20*)

Intersect a surface with a sequence of planes.

A sequence of *nplanes* planes with normal *dir* is constructed at equal distances spread over the bbox of the surface.

The return value is a list of `intersectionWithPlane()` return values, i.e. a list of Meshes, one for every cutting plane. In each Mesh the simply connected parts are identified by property number.

smooth (*method='lowpass'*, *iterations=1*, *lambda_value=0.5*, *neighbourhood=1*, *alpha=0.0*, *beta=0.2*)

Smooth the surface.

Returns a TriSurface which is a smoothed version of the original. Two smoothing methods are available: 'lowpass' and 'laplace'.

Parameters:

- *method*: 'lowpass' or 'laplace'
- *iterations*: int: number of iterations
- *lambda_value*: float: lambda value used in the filters

Extra parameters for 'lowpass' and 'laplace':

- *neighbourhood*: int: maximum number of edges followed in defining the node neighbourhood

Extra parameters for 'laplace':

- *alpha, beta*: float: parameters for the laplace method.

Returns the smoothed TriSurface

smoothLowPass (*iterations=2, lambda_value=0.5, neighbours=1*)

Apply a low pass smoothing to the surface.

smoothLaplaceHC (*iterations=2, lambda_value=0.5, alpha=0.0, beta=0.2*)

Apply Laplace smoothing with shrinkage compensation to the surface.

refine (*max_edges=None, min_cost=None, method='gts'*)

Refine the TriSurface.

Refining a TriSurface means increasing the number of triangles and reducing their size, while keeping the changes to the modeled surface minimal. Construct a refined version of the surface. This uses the external program *gtsrefine*. The surface should be a closed orientable non-intersecting manifold. Use the *check()* method to find out.

Parameters:

- *max_edges*: int: stop the refining process if the number of edges exceeds this value
- *min_cost*: float: stop the refining process if the cost of refining an edge is smaller
- *log*: boolean: log the evolution of the cost
- *verbose*: boolean: print statistics about the surface

similarity (*S*)

Compute the similarity with another TriSurface.

Compute a quantitative measure of the similarity of the volumes enclosed by two TriSurfaces. Both the calling and the passed TriSurface should be closed manifolds (see *isClosedManifold()*).

Returns a tuple a tuple (jaccard, dice, overlap). If A and B are two closed manifolds, VA and VB are their respective volumes, VC is the volume of the intersection of A and B, and VD is the volume of the union of A and B, then the following similarity measures are defined:

- jaccard coefficient: VC / VD
- dice: $2 * VC / (VA + VB)$
- overlap: $VC / \min(VA, VB)$

Both jaccard and dice range from 0 when the surfaces are completely disjoint to 1 when the surfaces are identical. The overlap coefficient becomes 1 when one of the surfaces is completely inside the other.

This method uses gts library to compute the intersection or union. If that fails, nan values are returned.

fixNormals (*outwards=True*)

Fix the orientation of the normals.

Some surface operations may result in improperly oriented normals, switching directions from one triangle to the adjacent one. This method tries to reverse improperly oriented normals so that a singly oriented surface is achieved.

If the surface is a (possibly non-orientable) manifold, the result will be an orientable manifold.

If the surface is a closed manifold, the normals will be oriented to the outside. This is done by computing the volume inside the surface and reversing the normals if that turns out to be negative.

Parameters:

- *outwards*: boolean: if True (default), a test is done whether the surface is a closed manifold, and if so, the normals are oriented outwards. Setting this value to False will skip this test and the (possible) reversal of the normals.

check (*matched=True, verbose=False*)

Check the surface using gtscheck.

Uses the external program *gtscheck* to check whether the surface is an orientable, non self-intersecting manifold. This is a necessary condition for using the *gts* methods: *split*, *coarsen*, *refine*, *boolean*. Additionally, the surface should be closed: this can be checked with *isClosedManifold()*.

Parameters

- **matched** (*bool*) – If True, self intersecting triangles are returned as element indices of self. This is the default. If False, the self intersecting triangles are returned as a separate *TriSurface*.
- **verbose** (*bool*) – If True, prints the statistics reported by the *gtscheck* command.

Returns

- **status** (*int*) – Return code from the checking program. One of the following:
 - 0: the surface is an orientable, non self-intersecting manifold.
 - 1: the created GTS file is invalid: this should normally not occur.
 - 2: the surface is not an orientable manifold. This may be due to misoriented normals. The *fixNormals()* and *reverse()* methods may be used to help fixing the problem in such case.
 - 3: the surface is an orientable manifold but is self-intersecting. The self intersecting triangles are returned as the second return value.
- **intersect** (*None | list of ints | TriSurface*) – None in case of a status 0, 1 or 2. For status value 3, returns the self intersecting triangles as a list of element numbers (if *matched* is True) or as a *TriSurface* (if *matched* is False).

split (*base, verbose=False*)

Split the surface using *gtssplit*.

Splits the surface into connected and manifold components. This uses the external program *gtssplit*. The surface should be a closed orientable non-intersecting manifold. Use the *check()* method to find out.

This method creates a series of files with given base name, each file contains a single connected manifold.

coarsen (*min_edges=None, max_cost=None, mid_vertex=False, length_cost=False, max_fold=1.0, volume_weight=0.5, boundary_weight=0.5, shape_weight=0.0, progressive=False, log=False, verbose=False*)

Coarsen a surface using *gtscoarsen*.

Construct a coarsened version of the surface. This uses the external program *gtscoarsen*. The surface should be a closed orientable non-intersecting manifold. Use the *check()* method to find out.

Parameters

- **min_edges** (*int*) – Stop the coarsening process if the number of edges was to fall below it.
- **max_cost** (*float*) – Stop the coarsening process if the cost of collapsing an edge is larger than the specified value.
- **mid_vertex** (*bool*) – Use midvertex as replacement vertex instead of the default, which is a volume optimized point.
- **length_cost** (*bool*) – Use length² as cost function instead of the default optimized point cost.
- **max_fold** (*float*) – Maximum fold angle in degrees.
- **volume_weight** (*float*) – Weight used for volume optimization.
- **boundary_weight** (*float*) – Weight used for boundary optimization.
- **shape_weight** (*float*) – Weight used for shape optimization.
- **progressive** (*bool*) – If True, write progressive surface file.
- **log** (*bool*) – If True, log the evolution of the cost.
- **verbose** (*bool*) – If True, print statistics about the surface.

gts_refine (*max_edges=None, min_cost=None, log=False, verbose=False*)

Refine the TriSurface.

Refining a TriSurface means increasing the number of triangles and reducing their size, while keeping the changes to the modeled surface minimal. This uses the external program *gtsrefine*. The surface should be a closed orientable non-intersecting manifold. Use the *check()* method to find out.

Parameters

- **max_edges** (*int*) – Stop the refining process if the number of edges exceeds this value.
- **min_cost** (*float*) – Stop the refining process if the cost of refining an edge is smaller.
- **log** (*bool*) – If True, log the evolution of the cost.
- **verbose** (*bool*) – If True, print statistics about the surface.

gts_smooth (*iterations=1, lambda_value=0.5, verbose=False*)

Smooth the surface using *gtssmooth*.

Smooth a surface by applying iterations of a Laplacian filter. This uses the external program *gtssmooth*. The surface should be a closed orientable non-intersecting manifold. Use the *check()* method to find out.

Parameters

- **lambda_value** (*float*) – Laplacian filter parameter.
- **iterations** (*int*) – Number of iterations.
- **verbose** (*bool*) – If True, print statistics about the surface.

See also:

smoothLowPass(), *smoothLaplaceHC()*

inside (*pts*, *method*='gts', *tol*='auto', *multi*=False)

Test which of the points *pts* are inside the surface.

Parameters

- **pts** (*:term_`coords_like`*) – The points to check against the surface.
- **method** (*str*) – Method to be used for the detection. Depending on the software you have installed the following are possible:
 - 'gts': provided by pyformex-extra (default)
 - 'vtk': provided by python-vtk (slower)
- **tol** (*float*) – Tolerance on equality of floating point values.

Returns *int array* – The indices of the points that are inside the surface. The indices refer to the onedimensional list of points as obtained from `Coords(pts).points()`.

outside (*pts*, ***kargs*)

Returns the points outside the surface.

This is the complement of `inside()`. See there for parameters and return value.

voxelize (*n*, *bbox*=0.01, *return_formex*=False)

Voxelize the volume inside a closed surface.

Parameters

- **n** (*int or (int, int, int)*) – Resolution, i.e. number of voxel cells to use along the three axes. If a single *int* is specified, the number of cells will be adapted according to the surface's `sizes()` (as the voxel cells are always cubes). The specified number of voxels will be used along the largest direction.
- **bbox** (*float or (point, point)*) – Defines the bounding box of the volume that needs to be voxelized. A float specifies a relative amount to add to the surface's bounding box. Note that this defines the bounding box of the centers of the voxels.
- **return_formex** (*bool*) – If True, also returns a Formex with the centers of the voxels.

Returns

- **voxels** (*int array (nz,ny,nx)*) – The array has a value 1 for the voxels whose center is inside the surface, else 0.
- **centers** (*Formex*) – A plex-1 Formex with the centers of the voxels, and property values 0 or 1 if the point is respectively outside or inside the surface. The voxel cell ordering in the Formex is z-direction first, then y, then x.

Notes

See also example `Voxelize`, for saving the voxel values in a stack of binary images.

tetgen (*quality*=2.0, *volume*=None, *filename*=None)

Create a tetrahedral mesh inside the surface.

This uses `tetMesh()` to generate a quality tetrahedral mesh inside the surface. The surface should be a closed manifold.

Parameters

- **quality** (*float*) – The quality of the output tetrahedral mesh. The value is a constraint on the circumradius-to-shortest-edge ratio. The default (2.0) already provides a

high quality mesh. Providing a larger value will reduce quality but increase speed. With `quality=None`, no quality constraint will be imposed.

- **volume** (*float, optional*) – If provided, applies a maximum tetrahedron volume constraint.
- **filename** (*path_like*) – Specifies where the intermediate files will be stored. The default will use a temporary directory which will be destroyed after return. If the path of an existing directory is provided, the files will be stored in that directory with a name ‘surface.off’ for the original surface model and files ‘surface.1.*’ for the generated tetrahedral model (in tetgen format). If the path does not exist or is an existing file, the parent directory should exist and files are stored with the given file name as base. Existing files will be silently overwritten.

Returns *Mesh* – A tetrahedral Mesh (eltype=‘tet4’) filling the input surface, provided the `tetMesh()` function finished successfully.

boolean (*surf, op, check=False, verbose=False*)

Perform a boolean operation with another surface.

Boolean operations between surfaces are a basic operation in free surface modeling. Both surfaces should be closed orientable non-intersecting manifolds. Use the `check()` method to find out.

The boolean operations are set operations on the enclosed volumes: union(‘+’), difference(‘-’) or intersection(‘*’).

Parameters

- **surf** (*TriSurface*) – Another TriSurface that is a closed manifold surface.
- **op** (‘+’, ‘-’ or ‘*’) – The boolean operation to perform: union(‘+’), difference(‘-’) or intersection(‘*’).
- **check** (*bool*) – If True, a check is done that the surfaces are not self-intersecting; if one of them is, the set of self-intersecting faces is written (as a GtsSurface) on standard output
- **verbose** (*bool*) – If True, print statistics about the surface.

Returns *TriSurface* – A closed manifold TriSurface that is the volume union, difference or intersection of self with surf.

Note: This method uses the external command ‘gtsset’ and will not run if it is not installed (available from pyformex/extras).

gtsset (*surf, op, filt="", ext="", curve=False, check=False, verbose=False*)

_Perform the boolean/intersection methods.

See the boolean/intersection methods for more info. Parameters not explained there:

- *filt*: a filter command to be executed on the gtsset output
- *ext*: extension of the result file
- *curve*: if True, an intersection curve is computed, else the surface.

Returns the resulting TriSurface (*curve=False*), or a plex-2 Formex (*curve=True*), or None if the input surfaces do not intersect.

intersection (*surf, check=False, verbose=False*)

Return the intersection curve of two surfaces.

Boolean operations between surfaces are a basic operation in free surface modeling. Both surfaces should be closed orientable non-intersecting manifolds. Use the `check()` method to find out.

Parameters:

- `surf`: a closed manifold surface
- `check`: boolean: check that the surfaces are not self-intersecting; if one of them is, the set of self-intersecting faces is written (as a `GtsSurface`) on standard output
- `verbose`: boolean: print statistics about the surface

Returns: a list of intersection curves.

webgl (*name*, *caption=None*)

Create a WebGL model of a surface

- `S`: `TriSurface`
- `name`: basename of the output files
- `caption`: text to use as caption

Functions defined in module trisurface

`trisurface.adjacencyArrays` (*elems*, *nsteps=1*)

Create adjacency arrays for 2-node elements.

`elems` is a (nr,2) shaped integer array. The result is a list of adjacency arrays, where row `i` of adjacency array `j` holds a sorted list of the nodes that are connected to node `i` via a shortest path of `j` elements, padded with -1 values to create an equal list length for all nodes. This is: [adj0, adj1, ..., adjj, ..., adjn] with `n=nsteps`.

Examples

```
>>> for a in adjacencyArrays([[0,1],[1,2],[2,3],[3,4],[4,0]],3):
...     print(a)
[[0]
 [1]
 [2]
 [3]
 [4]
 [1 4]
 [0 2]
 [1 3]
 [2 4]
 [0 3]]
[[2 3]
 [3 4]
 [0 4]
 [0 1]
 [1 2]]
[]
```

`trisurface.stlConvert` (*stlname*, *outname=None*, *binary=False*, *options='-d'*)

Convert an .stl file to .off or .gts or binary .stl format.

Parameters

- `stlname` (*path_like*) – Name of an existing .stl file (either ascii or binary).
- `outname` (*str or Path*) – Name or suffix of the output file. The suffix defines the format and should be one of '.off', '.gts', '.stl', '.stla', or '.stlb'. If a suffix only is given

(other than `.stl`), then the `outname` will be constructed by changing the suffix of the input `stlname`. If not specified, the suffix of the configuration variable `'surface/stlread'` is used.

- **binary** (*bool*) – Only used if the extension of `outname` is `.stl`. Defines whether the output format is a binary or ascii STL format.
- **options** (*str*) –

Returns

- **outname** (*Path*) – The name of the output file.
- **status** (*int*) – The exit status (0 if successful) of the conversion program.
- **stdout** (*str*) – The output of running the conversion program or a `'file is already up to date'` message.

Notes

If the `outname` file exists and its `mtime` is more recent than the `stlname`, the `outname` file is considered up to date and the conversion program will not be run.

The conversion program will be chosen depending on the extension. This uses the external commands `'admesh'` or `'stl2gts'`.

`trisurface.read_stl` (*fn, intermediate=None*)

Read a surface from `.stl` file.

This is done by first converting the `.stl` to `.gts` or `.off` format. The name of the intermediate file may be specified. If not, it will be generated by changing the extension of `fn` to `.gts` or `.off` depending on the setting of the `'surface/stlread'` config setting.

Return a `(coords,edges,faces)` or a `(coords,elems)` tuple, depending on the intermediate format.

`trisurface.curvature` (*coords, elems, edges, neighbours=1*)

Calculate curvature parameters at the nodes.

Algorithms based on Dong and Wang 2005; Koenderink and Van Doorn 1992. This uses the nodes that are connected to the node via a shortest path of `'neighbours'` edges. Eight values are returned: the Gaussian and mean curvature, the shape index, the curvedness, the principal curvatures and the principal directions.

`trisurface.fillBorder` (*border, method='radial', dir=None*)

Create a triangulated surface inside a given closed polygonal line.

Parameters

- **border** (*PolyLine, Mesh or Coords*) – A closed polygonal line that forms the border of the triangulated surface to be created. The polygon does not have to be planar. The line can be provided as one of the following:
 - a closed `PolyLine`,
 - a 2-plex `Mesh`, with a `Connectivity` table such that the elements in the specified order form a closed polyline,
 - a simple `Coords` holding the subsequent vertices of the polygonal border line.
- **method** (*str*) – Specifies the algorithm to be used to fill the polygon. Currently available are:
 - `'radial'`: this method adds a central point and connects all border segments with the center to create triangles.

- ‘border’: this method creates subsequent triangles by connecting the endpoints of two consecutive border segments and thus works its way inwards until the hole is closed. Triangles are created at the line segments that form the smallest angle.

See also Notes below.

Returns *TriSurface* – A *TriSurface* filling the hole inside the border.

Notes

The ‘radial’ method produces nice results if the border is relative smooth, nearly convex and nearly planar. It adds an extra point though, which may be unwanted. On irregular 3D borders there is a high chance that the resulting *TriSurface* contains intersecting triangles.

The ‘border’ method is slower on large borders, does not introduce any new point and has a better chance of avoiding intersecting triangles on irregular 3D borders.

The resulting surface can be checked for intersecting triangles with the `check()` method.

Because the ‘border’ does not create any new points, the returned surface can use the same point coordinate array as the input object.

6.2.3 coordsys — Coordinate Systems.

class `coordsys.CoordSys` (*oab=None, points=None, rot=None, trl=None*)

A Cartesian coordinate system in 3D space.

The coordinate system is stored as a rotation matrix and a translation vector, which transform the global coordinate axes into the axes of the *CoordSys*. Clearly, the translation vector is the origin of the *CoordSys*, and the rows of the rotation matrix are the unit vectors along the three axes.

The *CoordSys* can be initialized in different ways and has only optional parameters to achieve this. If none are specified at all, the global coordinate axis results.

Parameters

- **oab** (float *array_like* (3,3), optional) – Three non-collinear points O, A and B, that define the *CoordSys* in the following way: O is the origin of the coordinate system, A is a point along the positive first axis and B is a point B in the plane of the first two axes.
- **points** (float *array_like* (4,3), optional) – The *CoordSys* is specified by four points: the first three are points on the three coordinate axes, at distance +1 from the origin; the fourth point is the origin. The use of this parameter is deprecated. It can be replaced with `oab=points[3,0,1]`.
- **rot** (float *array_like* (3,3), optional) – Rotation matrix that transforms the global global axes to be parallel to the constructed *CoordSys*. The user is responsible to make sure that *rot* is a proper orthonormal rotation matrix.
- **trl** (float *array_like* (3,)) – Translation vector that moves the global origin to the origin of the *CoordSys*, in other words, this is the origin of the new *CoordSys* in global coordinates.

Notes

If *oab* is provided, it takes precedence and the other parameters are ignored. If not, and *points* is provided, this takes precedence and the remaining are ignored. If neither *oab* nor *points* are provided, *rot* and *trl* are used and have default values equal to the rotation matrix and translation vector of the global coordinate axes.

A `Coords` object has a number of attributes that provide quick access to its internal data. Of these, `trl` and `rot` can be used to set the data of the `CoordSys` and thus change the `CoordSys` in-place.

trl

The origin of the `CoordSys`

Type float array (3,)

rot

The rotation matrix of the `CoordSys`

Type float array (3,3)

u

The unit vector along the first axis (0).

Type float array (3,)

v

The unit vector along the second axis (1).

Type float array (3,)

w

The unit vector along the third axis (2).

Type float array (3,)

o

The origin of the `CoordSys`.

Type float array (3,)

Examples

Three points O,A,B in the xy-plane, first two parallel to x-axis, third at higher y-value. The resulting `CoordSys` is global axes translated to point O.

```
>>> OAB = Coords([[ 2., 1., 0.], [ 5., 1., 0.], [ 0., 3., 0.]])
>>> print(CoordSys(oab=OAB))
CoordSys: trl=[ 2.  1.  0.]  rot=[[ 1.  0.  0.]
                                [ 0.  1.  0.]
                                [ 0.  0.  1.]
```

Now translate the points so that O is on the z-axis, and rotate the points 30 degrees around z-axis.

```
>>> OAB = OAB.trl([-2., -1., 5.]).rot(30)
>>> print(OAB)
[[ 0.    0.    5. ]
 [ 2.6   1.5   5. ]
 [-2.73  0.73  5. ]]
>>> C = CoordSys(oab=OAB)
>>> print(C)
CoordSys: trl=[ 0.  0.  5.]  rot=[[ 0.87  0.5  0. ]
                                [-0.5  0.87  0. ]
                                [ 0.   -0.   1. ]]
```

Reverse axes x and y. The resulting `CoordSys` is still righthanded. This is equivalent to a rotation over 180 degrees around z-axis.

```
>>> print(C.reverse(0,1))
CoordSys: trl=[ 0.  0.  5.]; rot=[[-0.87 -0.5  -0.  ]
                                [ 0.5  -0.87 -0.  ]
                                [ 0.   -0.   1.  ]]
```

Now rotate C over 150 degrees around z-axis to become parallel with the global axes.

```
>>> print(C.rotate(150,2))
CoordSys: trl=[ 0.  0.  5.]; rot=[[ 1.  0.  0.]
                                [-0.  1.  0.]
                                [ 0.  0.  1.]]
```

trl

Return the origin as a (3,) vector

rot

Return the (3,3) rotation matrix

u

Return unit vector along axis 0 (x)

v

Return unit vector along axis 1 (y)

w

Return unit vector along axis 2 (z)

o

Return the origin

origin

Return the origin

axes

Return the (3,3) rotation matrix

axis (*i*)

Return the unit vector along an axis.

Parameters *i* (*int* (0, 1, 2)) – The axis number.

Returns *float array* (3,) – A unit vector along axis *i*.

Notes

If the axis is known in advance, it is more appropriate to use one of the u, v or w attributes

Examples

```
>>> CoordSys().rotate(30).axis(1)
array([-0.5 ,  0.87,  0.  ])
```

points ()

Return origin and endpoints of unit vectors along axes.

Returns *Coords* (4,3) – A Coords object with four points: the endpoints of the unit vectors along the three axes of the CoordSys, and the origin of the CoordSys.

Examples

```
>>> CoordSys().rotate(30).points()
Coords([[ 0.87,  0.5 ,  0. ],
        [-0.5 ,  0.87,  0. ],
        [ 0.  ,  0.  ,  1. ],
        [ 0.  ,  0.  ,  0. ]])
```

translate (*args, **kargs)

Translate the CoordSys like a Coords object.

Parameters are like `Coords.translate()`.

Returns A new CoordSys obtained by giving *self* a translation.

Examples

```
>>> print(CoordSys().translate([-2., -1., 5.]))
CoordSys: trl=[-2. -1.  5.]; rot=[[ 1.  0.  0.]
                                [ 0.  1.  0.]
                                [ 0.  0.  1.]]
```

rotate (*args, **kargs)

Rotate the CoordSys like a Coords object.

Parameters are like `Coords.rotate()`.

Returns A new CoordSys obtained by giving *self* a rotation.

Examples

```
>>> print(CoordSys().rotate(30))
CoordSys: trl=[ 0.  0.  0.]; rot=[[ 0.87  0.5  0. ]
                                [-0.5  0.87  0. ]
                                [ 0.  0.  1. ]]
```

reverse (*axes)

Reverse some axes of the CoordSys.

Parameters **axes** (*int* (0,1,2) or *tuple of ints*) – The axes to be reversed.

Note: The reversing a single axis (or all three axes) will change a right-hand-sided CoordSys into a left-hand-sided one. Therefore, this method is normally used only with two axes.

Examples

```
>>> print(CoordSys().reverse(0,1))
CoordSys: trl=[ 0.  0.  0.]; rot=[[ -1. -0. -0.]
                                [-0. -1. -0.]
                                [ 0.  0.  1.]]
```

6.2.4 geometry — A generic interface to the Coords transformation methods

This module defines a generic Geometry superclass which adds all the possibilities of coordinate transformations offered by the Coords class to the derived classes.

class geometry.Geometry

A generic geometry class allowing manipulation of large coordinate sets.

The Geometry class is a generic parent class for all geometry classes. It is not intended to be used directly, but only through derived classes. Examples of derived classes are *Formex*, *Mesh* and its subclass *TriSurface*, *Curve*.

The basic entity of geometry is the point, defined by its coordinates. The Geometry class expects these to be stored in a *Coords* object assigned to the *coords* attribute (it is the responsibility of the derived class object initialisation to do this).

The Geometry class exposes the following attributes of the *coords* attribute, so that they can be directly used on the Geometry object: *xyz*, *x*, *y*, *z*, *xy*, *yz*, *xz*.

The Geometry class exposes a large set of Coords methods for direct use on the derived class objects. These methods are automatically executed on the *coords* attribute of the object. One set of such methods are those returning some information about the Coords: *points()*, *bbox()*, *center()*, *bboxPoint()*, *centroid()*, *sizes()*, *dsize()*, *bsphere()*, *bboxes()*, *inertia()*, *principalCS()*, *principalSizes()*, *distanceFromPlane()*, *distanceFromLine()*, *distanceFromPoint()*, *directionalSize()*, *directionalWidth()*, *directionalExtremes()*. Thus, if F is an instance of class *Formex*, then one can use *F.center()* as a convenient shorthand for *F.coords.center()*.

Likewise, most of the transformation methods of the `:class:~coords.Coords` class are exported through the Geometry class to the derived classes. When called, they will return a new object identical to the original, except for the coordinates, which are transformed by the specified method. Refer to the corresponding *Coords* method for the precise arguments of these methods: *scale()*, *adjust()*, *translate()*, *centered()*, *align()*, *rotate()*, *shear()*, *reflect()*, *affine()*, *toCS()*, *fromCS()*, *transformCS()*, *position()*, *cylindrical()*, *hyperCylindrical()*, *toCylindrical()*, *spherical()*, *superSpherical()*, *toSpherical()*, *circulize()*, *bump()*, *flare()*, *map()*, *map1()*, *mapd()*, *copyAxes()*, *swapAxes()*, *rollAxes()*, *projectOnPlane()*, *projectOnSphere()*, *projectOnCylinder()*, *isopar()*, *addNoise()*, *rot()*, *trl()*.

Geometry is a lot more than points however. Therefore the Geometry and its derived classes can represent higher level entities, such as lines, planes, circles, triangles, cubes,... These entities are often represented by multiple points: a line segment would e.g. need two points, a triangle three. Geometry subclasses can implement collections of many such entities, just like the *Coords* can hold many points. We call these geometric entities 'elements'. The subclass must at least define a method *nelems()* returning the number of elements in the object, even if there is only one.

The Geometry class allows the attribution of a property number per element. This is an integer number that can be used as the subclass or user wants. It could be just the element number, or a key into a database with lots of more element attributes. The Geometry class provides methods to handle these property numbers.

The Geometry class provides a separate mechanism to store attributes related to how the geometry should be rendered. For this purpose the class defines an *attrib* attribute which is an object of class *Attributes*. This attribute is callable to set any key/value pairs in its dict. For example, *F.attrib(color=yellow)* will make the object F always be drawn in yellow.

The Geometry class provides for adding fields to instances of the derived classes. Fields are numerical data (scalar or vectorial) that are defined over the geometry. For example, if the geometry represents a surface, the gaussian curvature of that surface is a field defined over the surface. Field data are stored in *Field* objects and

the Geometry object stores them internally in a dict object with the field name as key. The dict is kept in an attribute `fields` that is only created when the first Field is added to the object.

Finally, the Geometry class also provides the interface for storing the Geometry object on a file in pyFormex's own 'pgf' format.

Note: When subclassing the Geometry class, care should be taken to obey some rules in order for all the above to work properly. See *UserGuide*.

coords

A Coords object that holds the coordinates of the points required to describe the type of geometry.

Type *Coords*

prop

Element property numbers. This is a 1-dim int array with length `nelems()`. Each element of the Geometry can thus be assigned an integer value. It is up to the subclass to define if and how its instances are divided into elements, and how to use this element property number.

Type int array

attrib

An Attributes object that is primarily used to define persisting drawing attributes (like color) for the Geometry object.

Type *Attributes*

fields

A dict with the Fields defined on the object. This attribute only exists when at least one Field has been defined. See `addField()`.

Type *OrderedDict*

xyz

Return the `xyz` property of the `coords` attribute of the Geometry object.

See `coords.Coords.xyz` for details.

x

Return the `x` property of the `coords` attribute of the Geometry object.

See `coords.Coords.x` for details.

y

Return the `y` property of the `coords` attribute of the Geometry object.

See `coords.Coords.y` for details.

z

Return the `z` property of the `coords` attribute of the Geometry object.

See `coords.Coords.z` for details.

xy

Return the `xy` property of the `coords` attribute of the Geometry object.

See `coords.Coords.xy` for details.

yz

Return the `yz` property of the `coords` attribute of the Geometry object.

See `coords.Coords.yz` for details.

xz

Return the `xz` property of the `coords` attribute of the Geometry object.

See `coords.Coords.xz` for details.

nelems ()

Return the number of elements in the Geometry.

Returns *int* – The number of elements in the Geometry. This is an abstract method that should be reimplemented by the derived class.

level ()

Return the dimensionality of the Geometry

The level of a *Geometry* is the dimensionality of the geometric object(s) represented:

- 0: points
- 1: line objects
- 2: surface objects
- 3: volume objects

Returns *int* – The level of the Geometry, or -1 if it is unknown. This should be implemented by the derived class. The Geometry base class always returns -1.

info ()

Return a short string representation about the object

points (*args, **kargs)

Call `points ()` method on the `coords` attribute of the Geometry object.

See `coords.Coords.points ()` for details.

bbox (*args, **kargs)

Call `bbox ()` method on the `coords` attribute of the Geometry object.

See `coords.Coords.bbox ()` for details.

center (*args, **kargs)

Call `center ()` method on the `coords` attribute of the Geometry object.

See `coords.Coords.center ()` for details.

bboxPoint (*args, **kargs)

Call `bboxPoint ()` method on the `coords` attribute of the Geometry object.

See `coords.Coords.bboxPoint ()` for details.

centroid (*args, **kargs)

Call `centroid ()` method on the `coords` attribute of the Geometry object.

See `coords.Coords.centroid ()` for details.

sizes (*args, **kargs)

Call `sizes ()` method on the `coords` attribute of the Geometry object.

See `coords.Coords.sizes ()` for details.

dsize (*args, **kargs)

Call `dsize ()` method on the `coords` attribute of the Geometry object.

See `coords.Coords.dsize ()` for details.

bsphere (*args, **kargs)

Call *bsphere()* method on the `coords` attribute of the Geometry object.

See *coords.Coords.bsphere()* for details.

bboxes (*args, **kargs)

Call *bboxes()* method on the `coords` attribute of the Geometry object.

See *coords.Coords.bboxes()* for details.

inertia (*args, **kargs)

Call *inertia()* method on the `coords` attribute of the Geometry object.

See *coords.Coords.inertia()* for details.

principalCS (*args, **kargs)

Call *principalCS()* method on the `coords` attribute of the Geometry object.

See *coords.Coords.principalCS()* for details.

principalSizes (*args, **kargs)

Call *principalSizes()* method on the `coords` attribute of the Geometry object.

See *coords.Coords.principalSizes()* for details.

distanceFromPlane (*args, **kargs)

Call *distanceFromPlane()* method on the `coords` attribute of the Geometry object.

See *coords.Coords.distanceFromPlane()* for details.

distanceFromLine (*args, **kargs)

Call *distanceFromLine()* method on the `coords` attribute of the Geometry object.

See *coords.Coords.distanceFromLine()* for details.

distanceFromPoint (*args, **kargs)

Call *distanceFromPoint()* method on the `coords` attribute of the Geometry object.

See *coords.Coords.distanceFromPoint()* for details.

directionalSize (*args, **kargs)

Call *directionalSize()* method on the `coords` attribute of the Geometry object.

See *coords.Coords.directionalSize()* for details.

directionalWidth (*args, **kargs)

Call *directionalWidth()* method on the `coords` attribute of the Geometry object.

See *coords.Coords.directionalWidth()* for details.

directionalExtremes (*args, **kargs)

Call *directionalExtremes()* method on the `coords` attribute of the Geometry object.

See *coords.Coords.directionalExtremes()* for details.

convexHull (*dir=None, return_mesh=True*)

Return the convex hull of a Geometry.

This is the *convexHull()* applied to the `coords` attribute, but it has `return_mesh=True` as default.

Returns *Mesh* – The convex hull of the geometry.

testBbox (*bb, dirs=(0, 1, 2), nodes='any', atol=0.0*)

Test which part of a Formex or Mesh is inside a given bbox.

The Geometry object needs to have a *test()* method, This is the case for the *Formex* and *Mesh* classes. The test can be applied in 1, 2 or 3 viewing directions.

Parameters

- **bb** (*Coords (2, 3) or alike*) – The bounding box to test for.
- **dirs** (*tuple of ints (0, 1, 2)*) – The viewing directions in which to check the bbox bounds.
- **nodes** – Same as in *formex.Formex.test()* or *mesh.Mesh.test()*.

Returns *bool array* – The array flags the elements that are inside the given bounding box.

scale (**args, **kargs*)

Apply the *scale* transformation to the Geometry object.

See *coords.Coords.scale()* for details.

resized (*size=1.0, tol=1e-05*)

Return a copy of the Geometry scaled to the given size.

size can be a single value or a list of three values for the three coordinate directions. If it is a single value, all directions are scaled to the same size. Directions for which the geometry has a size smaller than *tol* times the maximum size are not rescaled.

adjust (**args, **kargs*)

Apply the *adjust* transformation to the Geometry object.

See *coords.Coords.adjust()* for details.

translate (**args, **kargs*)

Apply the *translate* transformation to the Geometry object.

See *coords.Coords.translate()* for details.

centered (**args, **kargs*)

Apply the *centered* transformation to the Geometry object.

See *coords.Coords.centered()* for details.

align (**args, **kargs*)

Apply the *align* transformation to the Geometry object.

See *coords.Coords.align()* for details.

rotate (**args, **kargs*)

Apply the *rotate* transformation to the Geometry object.

See *coords.Coords.rotate()* for details.

shear (**args, **kargs*)

Apply the *shear* transformation to the Geometry object.

See *coords.Coords.shear()* for details.

reflect (**args, **kargs*)

Apply the *reflect* transformation to the Geometry object.

See *coords.Coords.reflect()* for details.

affine (**args, **kargs*)

Apply the *affine* transformation to the Geometry object.

See *coords.Coords.affine()* for details.

toCS (**args, **kargs*)

Apply the *toCS* transformation to the Geometry object.

See *coords.Coords.toCS()* for details.

- fromCS** (*args, **kargs)
Apply the *fromCS* transformation to the Geometry object.
See `coords.Coords.fromCS()` for details.
- transformCS** (*args, **kargs)
Apply the *transformCS* transformation to the Geometry object.
See `coords.Coords.transformCS()` for details.
- position** (*args, **kargs)
Apply the *position* transformation to the Geometry object.
See `coords.Coords.position()` for details.
- cylindrical** (*args, **kargs)
Apply the *cylindrical* transformation to the Geometry object.
See `coords.Coords.cylindrical()` for details.
- hyperCylindrical** (*args, **kargs)
Apply the *hyperCylindrical* transformation to the Geometry object.
See `coords.Coords.hyperCylindrical()` for details.
- toCylindrical** (*args, **kargs)
Apply the *toCylindrical* transformation to the Geometry object.
See `coords.Coords.toCylindrical()` for details.
- spherical** (*args, **kargs)
Apply the *spherical* transformation to the Geometry object.
See `coords.Coords.spherical()` for details.
- superSpherical** (*args, **kargs)
Apply the *superSpherical* transformation to the Geometry object.
See `coords.Coords.superSpherical()` for details.
- toSpherical** (*args, **kargs)
Apply the *toSpherical* transformation to the Geometry object.
See `coords.Coords.toSpherical()` for details.
- circulize** (*args, **kargs)
Apply the *circulize* transformation to the Geometry object.
See `coords.Coords.circulize()` for details.
- bump** (*args, **kargs)
Apply the *bump* transformation to the Geometry object.
See `coords.Coords.bump()` for details.
- flare** (*args, **kargs)
Apply the *flare* transformation to the Geometry object.
See `coords.Coords.flare()` for details.
- map** (*args, **kargs)
Apply the *map* transformation to the Geometry object.
See `coords.Coords.map()` for details.

map1 (*args, **kargs)

Apply the *map1* transformation to the Geometry object.

See *coords.Coords.map1()* for details.

mapd (*args, **kargs)

Apply the *mapd* transformation to the Geometry object.

See *coords.Coords.mapd()* for details.

copyAxes (*args, **kargs)

Apply the *copyAxes* transformation to the Geometry object.

See *coords.Coords.copyAxes()* for details.

swapAxes (*args, **kargs)

Apply the *swapAxes* transformation to the Geometry object.

See *coords.Coords.swapAxes()* for details.

rollAxes (*args, **kargs)

Apply the *rollAxes* transformation to the Geometry object.

See *coords.Coords.rollAxes()* for details.

projectOnPlane (*args, **kargs)

Apply the *projectOnPlane* transformation to the Geometry object.

See *coords.Coords.projectOnPlane()* for details.

projectOnSphere (*args, **kargs)

Apply the *projectOnSphere* transformation to the Geometry object.

See *coords.Coords.projectOnSphere()* for details.

projectOnCylinder (*args, **kargs)

Apply the *projectOnCylinder* transformation to the Geometry object.

See *coords.Coords.projectOnCylinder()* for details.

isopar (*args, **kargs)

Apply the *isopar* transformation to the Geometry object.

See *coords.Coords.isopar()* for details.

addNoise (*args, **kargs)

Apply the *addNoise* transformation to the Geometry object.

See *coords.Coords.addNoise()* for details.

rot (*args, **kargs)

Apply the *rotate* transformation to the Geometry object.

See *coords.Coords.rotate()* for details.

trl (*args, **kargs)

Apply the *translate* transformation to the Geometry object.

See *coords.Coords.translate()* for details.

setProp (*prop=None*)

Create or destroy the property array for the Geometry.

A property array is a 1-dim integer array with length equal to the number of elements in the Geometry. Each element thus has its own property number. These numbers can be used for any purpose. In derived classes like *Formex* and `:class~mesh.Mesh` they play an import role when creating new geometry: new elements

inherit the property number of their parent element. Properties are also preserved on pure coordinate transformations.

Because elements with different property numbers can be drawn in different colors, the property numbers are also often used to impose color.

Parameters `prop` (int, int *array_like* or 'range') – The property number(s) to assign to the elements. If a single int, all elements get the same property value. If the number of passed values is less than the number of elements, the list will be repeated. If more values are passed than the number of elements, the excess ones are ignored.

A special value 'range' may be given to set the property numbers equal to the element number. This is equivalent to passing `arange(self.nelems())`.

A value None (default) removes the properties from the Geometry.

Returns

- The calling object `self`, with the new properties inserted
- *or with the properties deleted if argument is None.*

Note: This is one of the few operations that change the object in-place. It still returns the object itself, so that this operation can be used in a chain with other operations.

See also:

`toProp()` Create a valid set of properties for the object

`whereProp()` Find the elements having some property value

`toProp(prop)`

Create a valid set of properties for the object.

Parameters `prop` (*int* or *int :term:*) – The property values to turn into a valid set for the object. If a single int, all elements get the same property value. If the number of passed values is less than the number of elements, the list will be repeated. If more values are passed than the number of elements, the excess ones are ignored.

Returns *int array* – A 1-dim int array that is valid as a property array for the Geometry object. The length of the array will is `self.nelems()` and the dtype is *Int*.

See also:

`setProp()` Set the properties for the object

`whereProp()` Find the elements having some property value

Note: When you set the properties (using `setProp()`) you do not need to call this method to validate the properties. It is implicitly called from `setProp()`.

`maxProp()`

Return the highest property value used.

Returns *int* – The highest value used in the properties array, or -1 if there are no properties.

`propSet()`

Return a list with unique property values in use.

Returns *int array* – The unique values in the properties array. If no properties are defined, an empty array is returned.

whereProp (*prop*)

Find the elements having some property value.

Parameters **prop** (*int or int :term:*) – The property value(s) to be found.

Returns

int array – A 1-dim int array with the indices of all the elements that have the property value `prop`, or one of the values in `prop`.

If the object has no properties, an empty array is returned.

See also:

setProp () Set the properties for the object

toProp () Create a valid set of properties for the object

selectProp () Return a Geometry object with only the matching elements

copy ()

Return a deep copy of the Geometry object.

Returns *Geometry (or subclass) object* – An object of the same class as the caller, having all the same data (for *coords*, *prop*, *attrib*, *fields*, and any other attributes possibly set by the subclass), but not sharing any data with the original object.

Note: This is seldomly used, because it may cause wildly superfluous copying of data. Only used it if you absolutely require data that are independent of those of the caller.

select (*sel, compact=False*)

Select some element(s) from a Geometry.

Parameters

- **sel** (*index-like*) – The index of element(s) to be selected. This can be anything that can be used as an index in an array:
 - a single element number
 - a list, or array, of element numbers
 - a bool list, or array, of length `self.nelems()`, where True values flag the elements to be selected
- **compact** (*bool, optional*) – This option is only useful for subclasses that have a `compact` method, such as `mesh.Mesh` and its subclasses. If True, the returned object will be compacted, i.e. unused nodes are removed and the nodes are renumbered from zero. If False (default), the node set and numbers are returned unchanged.

Returns *Geometry (or subclass) object* – An object of the same class as the caller, but only containing the selected elements.

See also:

cselect () Return all but the selected elements.

clip () Like `select`, but with `compact=True` as default.

cselect (*sel*, *compact=False*)

Return the Geometry with the selected elements removed.

Parameters

- **sel** (*index-like*) – The index of element(s) to be selected. This can be anything that can be used as an index in an array:
 - a single element number
 - a list, or array, of element numbers
 - a bool list, or array, of length `self.nelems()`, where True values flag the elements to be selected
- **compact** (*bool, optional*) – This option is only useful for subclasses that have a `compact` method, such as `mesh.Mesh` and its subclasses. If True, the returned object will be compacted, i.e. unused nodes are removed and the nodes are renumbered from zero. If False (default), the node set and numbers are returned unchanged.

Returns *Geometry (or subclass) object* – An object of the same class as the caller, but containing all but the selected elements.

See also:

`select ()` Return a Geometry with only the selected elements.

`cclip ()` Like `cselect`, but with `compact=True` as default.

clip (*sel*)

Return a Geometry only containing the selected elements.

This is equivalent with `select ()` but having `compact=True` as default.

See also:

`select ()` Return a Geometry with only the selected elements.

`cclip ()` The complement of `clip`, returning all but the selected elements.

cclip (*sel*)

Return a Geometry with the selected elements removed.

This is equivalent with `select ()` but having `compact=True` as default.

See also:

`cselect ()` Return a Geometry with only the selected elements.

`clip ()` The complement of `cclip`, returning only the selected elements.

selectProp (*prop*, *compact=False*)

Select elements by their property value.

Parameters **prop** (*int or int :term:*) – The property value(s) for which the elements should be selected.

Returns *Geometry (or subclass) object* – An object of the same class as the caller, but only containing the elements that have a property value equal to `prop`, or one of the values in `prop`. If the input object has no properties, a copy containing all elements is returned.

See also:

cselectProp() Return all but the elements with property `prop`.

whereProp() Get the numbers of the elements having a specified property.

select() Select the elements with the specified indices.

cselectProp (*prop*, *compact=False*)

Return an object without the elements with property *val*.

Parameters **prop** (*int* or *int :term:*) – The property value(s) of the elements that should be left out.

Returns *Geometry (or subclass) object* – An object of the same class as the caller, with all but the elements that have a property value equal to `prop`, or one of the values in `prop`. If the input object has no properties, a copy containing all elements is returned.

See also:

selectProp() Return only the elements with property `prop`.

whereProp() Get the numbers of the elements having a specified property.

cselect() Remove elements by their index.

splitProp (*prop=None*, *compact=True*)

Partition a Geometry according to the values in `prop`.

Parameters **prop** (*int array_like*, optional) – A 1-dim int array with length `self.nelems()` to be used in place of the objects own `prop` attribute. If `None` (default), the latter will be used.

Returns

List of Geometry objects – A list of objects of the same class as the caller. Each object in the list contains all the elements having the same value of `prop`. The number of objects in the list is equal to the number of unique values in `prop`. The list is sorted in ascending order of the `prop` value.

If `prop` is `None` and the the object has no `prop` attribute, an empty list is returned.

fields

Return the Fields dict of the Geometry.

If the Geometry has no Fields, an empty dict is returned.

addField (*fldtype*, *data*, *fldname*)

Add *Field* data to the *Geometry*.

Field data are scalar or vectorial data defined over the Geometry. This convenience function creates a *Field* object with the specified data and adds it to the Geometry object's *fields* dict.

Parameters

- **fldtype** (*str*) – The field type. See *Field* for details.
- **data** – The field data. See *Field* for details.
- **fldname** (*str*) – The field name. See *Field* for details. This name is also used as key to store the Field in the *fields* dict.

Returns *Field* – The constructed and stored Field object.

Note: Whenever a Geometry object is exported to a PGF file, all Fields stored inside the Geometry object are included in the PGF file.

See also:

`getField()` Retrieve a Field by its name.

`delField()` Deleted a Field.

`convertField()` Convert the Field to another Field type.

`fieldReport()` Return a short report of the stored Fields

getField (*fldname*)

Get the data field with the specified name.

Parameters **fldname** (*str*) – The name of the Field to retrieve.

Returns *Field* – The data field with the specified name, if it exists in the Geometry object's *fields*. Returns None if no such key exists.

delField (*fldname*)

Delete the Field with the specified name.

Parameters **fldname** (*str*) – The name of the Field to delete from the Geometry object. A nonexisting name is silently ignored.

Returns *None*

convertField (*fldname, totype, toname*)

Convert the data field with the specified name to another type.

Parameters

- **fldname** (*str*) – The name of the data Field to convert to another type. A nonexisting name is silently ignored.
- **totype** (*str*) – The field type to convert to. See *Field* for details.
- **toname** (*str*) – The name of the new (converted) Field (and key to store it). If the same name is specified as the old Field, that one will be overwritten by the new. Otherwise, both will be kept in the Geometry object's *fields* dict.

Returns *Field* – The converted and stored data field. Returns None if the original data field does not exist.

fieldReport ()

Return a short report of the stored fields

Returns *str* – A multiline string with the stored Fields' attributes: name, type, dtype and shape.

write (*filename, sep=' ', mode='w', **kargs*)

Write a Geometry to a PGF file.

This writes the Geometry to a pyFormex Geometry File (PGF) with the specified name.

It is a convenient shorthand for:

```
writeGeomFile(filename, self, sep=sep, mode=mode, **kargs)
```

See `writeGeomFile()` for details.

classmethod `read(filename)`

Read a Geometry from a PGF file.

This reads a single object (the object) from a PGF file and returns it.

It is a convenient shorthand for:

```
next(readGeomFile(filename, 1).values())
```

See `readGeomFile()` for details.

6.2.5 connectivity — A class and functions for handling nodal connectivity.

This module defines a specialized array class for representing nodal connectivity. This is e.g. used in mesh models, where geometry is represented by a set of numbered points (nodes) and the geometric elements are described by referring to the node numbers. In a mesh model, points common to adjacent elements are unique, and adjacency of elements can easily be detected from common node numbers.

class `connectivity.Connectivity(data=[], dtyp=None, copy=False, nplex=0, eltype=None)`

A class for handling element to node connectivity.

A connectivity object is a 2-dimensional integer array with all non-negative values. Each row of the array defines an element by listing the numbers of its lower entity types. A typical use is a `Mesh` object, where each element is defined in function of its nodes. While in a `Mesh` the word ‘node’ will normally refer to a geometrical point, here we will use ‘node’ for the lower entity whatever its nature is. It doesn’t even have to be a geometrical entity.

Note: The current implementation limits a Connectivity object to numbers that are smaller than $2^{*}31$. That is however largely sufficient for all practical cases.

In a row (element), the same node number may occur more than once, though usually all numbers in a row are different. Rows containing duplicate numbers are called *degenerate* elements. Rows containing the same node sets, albeit different permutations thereof, are called duplicates.

Parameters

- **data** (*int :term:*) – Data to initialize the Connectivity. The data should be 2-dim with shape (`nelems`, `nplex`), where `nelems` is the number of elements and `nplex` is the plexitude of the elements.
- **dtyp** (*float datatype, optional*) – If not provided, the datatype of `data` is used.
- **copy** (*bool, optional*) – If True, the data are copied. The default setting will try to use the original data if possible, e.g. if `data` is a correctly shaped and typed `numpy.ndarray`.
- **nplex** (*int, optional*) – The plexitude of the data. This can be specified to force a check on the plexitude of the data, or to set the plexitude for an empty Connectivity. If an `eltype` is specified, the plexitude of the element type will override this value.
- **eltype** (*str or `elements.ElementType` subclass, optional*) – The element type associated with the Connectivity. It can be either a subclass of `class:elements.ElementType` or the name of such a subclass. If not provided, a non-typed Connectivity will result. If that is used to create a `Mesh`, the proper element type will have to be specified at `Mesh` creation time. If the Connectivity will be used for other purposes, the element type may not be needed or not be important.

Raises `ValueError` – If `nplex` is provided and the specified data do not match the specified `plexitude`.

Notes

Empty Connectivities with `nelems==0` and `nplex > 0` can be useful, but a `Connectivity` with `nplex==0` generally is not.

Examples

```
>>> Connectivity([[0,1,2],[0,1,3],[0,3,2],[0,5,3]])
Connectivity([[0, 1, 2],
              [0, 1, 3],
              [0, 3, 2],
              [0, 5, 3]])
```

```
>>> Connectivity(np.array([], dtype=at.Int).reshape(0,3))
Connectivity([], shape=(0, 3))
```

`nelems()`

Return the number of elements in the `Connectivity` table.

Returns `int` – The number of rows in the table.

Examples

```
>>> Connectivity([[0,1,2],[0,1,3],[0,3,2],[0,5,3]]).nelems()
4
```

`maxnodes()`

Return an upper limit for number of nodes in the `Connectivity`.

Returns `int` – The highest node number plus one.

See also:

`nnodes()` the actual number of nodes in the table

Examples

```
>>> Connectivity([[0,1,2],[0,1,3],[0,3,2],[0,5,3]]).maxnodes()
6
```

`nnodes()`

Return the actual number of nodes in the `Connectivity`.

This returns the count of the unique node numbers.

See also:

`maxnodes()` the highest node number + 1

Examples

```
>>> Connectivity([[0,1,2],[0,1,3],[0,3,2],[0,5,3]]).nnodes()
5
```

nplex()

Return the plexitude of the elements in the Connectivity table.

Examples

```
>>> Connectivity([[0,1,2],[0,1,3],[0,3,2],[0,5,3]]).nplex()
3
```

report()

Format a Connectivity table

testDegenerate()

Flag the degenerate elements (rows).

A degenerate element is a row which contains at least two equal values.

Returns *bool array* – A 1-dim bool array with length `self.nelems()`, holding True values for the degenerate rows.

Examples

```
>>> Connectivity([[0,1,2],[0,1,1],[0,3,2]]).testDegenerate()
array([False,  True,  False])
```

listDegenerate()

Return a list with the numbers of the degenerate elements.

Returns *int array* – A 1-dim int array holding the row indices of the degenerate elements.

Examples

```
>>> Connectivity([[0,1,2],[0,1,1],[0,3,2]]).listDegenerate()
array([1])
```

listNonDegenerate()

Return a list with the numbers of the non-degenerate elements.

Returns *int array* – A 1-dim int array holding the row indices of the non-degenerate elements.

Examples

```
>>> Connectivity([[0,1,2],[0,1,1],[0,3,2]]).listNonDegenerate()
array([0, 2])
```

removeDegenerate()

Remove the degenerate elements from a Connectivity table.

Returns *Connectivity* – A Connectivity object with the degenerate elements removed.

Examples

```
>>> Connectivity([[0,1,2],[0,1,1],[0,3,2]]).removeDegenerate()
Connectivity([[0, 1, 2],
              [0, 3, 2]])
```

findDuplicate (*permutations='all'*)

Find duplicate rows in the Connectivity.

Parameters **permutations** (*str*) – Defines which permutations of the row data are allowed while still considering the rows equal. Possible values are:

- 'none': no permutations are allowed: rows must match the same data at the same positions.
- 'roll': rolling is allowed. Rows that can be transformed into each other by rolling are considered equal;
- 'all': any permutation of the same data will be considered an equal row. This is the default.

Returns **V** (*Varray*) – A Varray where each row contains a list of the row numbers from a that are considered equal. The entries in each row are sorted and the rows are sorted according to their first element.

Notes

This is like `arraytools.equalRows()` but has a different default value for `permutations`.

Examples

```
>>> C = Connectivity([[0,1,2],[0,1,3],[0,1,2],[2,0,1],[2,1,0]])
>>> C.findDuplicate()
Varray([[0, 2, 3, 4], [1]])
>>> C.findDuplicate(permutations='roll')
Varray([[0, 2, 3], [1], [4]])
>>> C.findDuplicate(permutations='none')
Varray([[0, 2], [1], [3], [4]])
```

listDuplicate (*permutations='all'*)

Return a list with the numbers of the duplicate elements.

Returns *1-dim int array* – The indices of the unique rows in the Connectivity array.

Examples

```
>>> C = Connectivity([[0,1,2],[0,1,3],[0,1,2],[2,0,1],[2,1,0]])
>>> C.listDuplicate()
array([2, 3, 4])
>>> C.listDuplicate(permutations='roll')
array([2, 3])
>>> C.listDuplicate(permutations='none')
array([2])
```

listUnique (*permutations='all'*)

Return a list with the numbers of the unique elements.

Returns *1-dim int array* – The indices of the unique rows in the Connectivity array.

See also:

findDuplicate() find duplicate rows

listDuplicate() list duplicate rows

removeDuplicate() remove duplicate rows

Examples

```
>>> C = Connectivity([[0,1,2],[0,1,3],[0,1,2],[2,0,1],[2,1,0]])
>>> C.listUnique()
array([0, 1])
>>> C.listUnique(permutations='roll')
array([0, 1, 4])
>>> C.listUnique(permutations='none')
array([0, 1, 3, 4])
```

removeDuplicate (*permutations='all'*)

Remove duplicate elements from a Connectivity list.

By default, duplicates are elements that consist of the same set of nodes, in any particular order. Setting permutations to 'none' will only remove the duplicate rows that have matching values at matching positions.

Returns *Connectivity* – A new Connectivity with the duplicate elements removed.

Examples

```
>>> C = Connectivity([[0,1,2],[0,1,3],[0,1,2],[2,0,1],[2,1,0]])
>>> C.removeDuplicate()
Connectivity([[0, 1, 2],
              [0, 1, 3]])
>>> C.removeDuplicate(permutations='roll')
Connectivity([[0, 1, 2],
              [0, 1, 3],
              [2, 1, 0]])
>>> C.removeDuplicate(permutations='none')
Connectivity([[0, 1, 2],
              [0, 1, 3],
              [2, 0, 1],
              [2, 1, 0]])
```

reorder (*order='nodes'*)

Reorder the elements of a Connectivity in a specified order.

This does not actually reorder the elements itself, but returns an index with the order of the rows (elements) in the Connectivity table that meets the specified ordering requirements.

Parameters *order* (*str or list of ints*) – Specifies how to reorder the elements. It is either one of the special string values defined below, or else it is an index with length equal to the number of elements. The index should be a permutation of the numbers in `range(self.nelems())`. Each value gives the number of the old element that should be placed at this position. Thus, the order values are the old element numbers on the position of the new element number.

order can also take one of the following predefined values, resulting in the corresponding renumbering scheme being generated:

- 'nodes': the elements are renumbered in order of their appearance in the inverse index, i.e. first are the elements connected to node 0, then the as yet unlisted elements connected to node 1, etc.
- 'random': the elements are randomly renumbered.
- 'reverse': the elements are renumbered in reverse order.

Returns *1-dim int array* – Int array with a permutation of `arange(self.nelems())`, such that taking the elements in this order will produce a Connectivity reordered as requested. In case an explicit order was specified as input, this order is returned after checking that it is indeed a permutation of `range(self.nelems())`.

Examples

```
>>> A = Connectivity([[1,2],[2,3],[3,0],[0,1]])
>>> A[A.reorder('reverse')]
Connectivity([[0, 1],
              [3, 0],
              [2, 3],
              [1, 2]])
>>> A[A.reorder('nodes')]
Connectivity([[0, 1],
              [3, 0],
              [1, 2],
              [2, 3]])
>>> A[A.reorder([2,3,0,1])]
Connectivity([[3, 0],
              [0, 1],
              [1, 2],
              [2, 3]])
```

renumber (*start=0*)

Renumber the nodes to a consecutive integer range.

The node numbers in the table are changed thus that they form a consecutive integer range starting from the specified value.

Parameters **start** (*int*) – Lowest node number to be used in the renumbered Connectivity.

Returns

- **elems** (*Connectivity*) – The renumbered Connectivity
- **oldnrs** (*1-dim int array*) – The sorted list of unique (old) node numbers. The new node numbers are assigned in order of increasing old node numbers, thus the old node number for new node number *i* can be found at position *i - start*.

Examples

```
>>> e,n = Connectivity([[0,2],[1,4],[4,2]]).renumber(7)
>>> print(e)
[[ 7  9]
 [ 8 10]
```

(continues on next page)

(continued from previous page)

```
[10  9]]
>>> print(n)
[0 1 2 4]
```

Find the old node number of new node 10 >>> n[10-7] 4

inverse (*expand=True*)

Return the inverse index of a Connectivity table.

Returns *int array* – The inverse index of the Connectivity, as computed by *arraytools.inverseIndex()*.

Examples

```
>>> Connectivity([[0,1,2],[0,1,4],[0,4,2]]).inverse()
array([[ 0,  1,  2],
       [-1,  0,  1],
       [-1,  0,  2],
       [-1, -1, -1],
       [-1,  1,  2]])
>>> Connectivity([[0,1,2],[0,1,4],[0,4,2]]).inverse(expand=False)
Varray([[0, 1, 2], [0, 1], [0, 2], [], [1, 2]])
```

nParents ()

Return the number of elements connected to each node.

Returns *1-dim int array* – The number of elements connected to each node. The length of the array is equal to the highest node number + 1. Unused node numbers will have a count of zero.

Examples

```
>>> Connectivity([[0,1,2],[0,1,4],[0,4,2]]).nParents()
array([3, 2, 2, 0, 2])
```

connectedTo (*nodes, return_ncon=False*)

Check if the elements are connected to the specified nodes.

Parameters

- **nodes** (*int or int :term:*) – One or more node numbers to check for connections in the table.
- **return_ncon** (*bool, optional*) – If True, also return the number of connections for each element.

Returns

- **connections** (*int array*) – The numbers of the elements that contain at least one of the specified nodes.
- **ncon** (*int array, optional*) – The number of connections for each connected element. This is only provided if *return_ncon* is True.

Examples

```
>>> A = Connectivity([[0,1,2],[0,1,3],[0,3,2],[1,2,3]])
>>> print(A.connectedTo(2))
[0 2 3]
>>> A.connectedTo([0,1,3],True)
(array([0, 1, 2, 3]), array([2, 3, 2, 2]))
```

hits (nodes)

Count the nodes from a list connected to the elements.

Parameters **nodes** (*int* or *list of ints*) – One or more node numbers.

Returns *int array (nelems)* – An int array holding the number of nodes from the specified input that are contained in each of the elements.

Notes

This information can also be got from meth:*connectedTo*. This method however expands the results to the full element set, making it apt for use in selector expressions like `self[self.hits(nodes) >= 2]`.

Examples

```
>>> A = Connectivity([[0,1,2],[0,1,3],[0,3,2],[1,2,3]])
>>> A.hits(2)
array([1, 0, 1, 1])
>>> A.hits([0,1,3])
array([2, 3, 2, 2])
```

adjacency (kind='e', mask=None)

Create a table of adjacent items.

This creates an element adjacency table or node adjacency table. An element *i* is said to be adjacent to element *j*, if the two elements have at least one common node. A node *i* is said to be adjacent to node *j*, if there is at least one element containing both nodes.

Parameters

- **kind** ('e' or 'n') – Select element ('e') or node ('n') adjacency table. Default is element adjacency.
- **mask** (*bool array or int index, optional*) – Node selector. If provided (with `kind=='e'`) this defines by a bool flag array or int index numbers the list of nodes that are to be considered connectors between elements. The default is to consider all nodes as connectors.

This option is only useful in the case `kind == 'e'`. If you want to use an element mask for the 'n' case, just apply the (element) mask beforehand by using `self[mask].adjacency('n')`.

Returns *Adjacency* object – An Adjacency array with shape (nr,nc), where row *i* holds a sorted list of all the items that are adjacent to item *i*, padded with -1 values to create an equal list length for all items.

Examples

```

>>> Connectivity([[0,1],[0,2],[1,3],[0,5]]).adjacency('e')
Adjacency([[ 1,  2,  3],
           [-1,  0,  3],
           [-1, -1,  0],
           [-1,  0,  1]])
>>> Connectivity([[0,1],[0,2],[1,3],[0,5]]).adjacency('e',mask=[1,2,3,5])
Adjacency([[ 2],
           [-1],
           [ 0],
           [-1]])
>>> Connectivity([[0,1],[0,2],[1,3],[0,5]]).adjacency('n')
Adjacency([[ 1,  2,  5],
           [-1,  0,  3],
           [-1, -1,  0],
           [-1, -1,  1],
           [-1, -1, -1],
           [-1, -1,  0]])
>>> Connectivity([[0,1,2],[0,1,3],[2,4,5]]).adjacency('n')
Adjacency([[[-1,  1,  2,  3],
           [-1,  0,  2,  3],
           [ 0,  1,  4,  5],
           [-1, -1,  0,  1],
           [-1, -1,  2,  5],
           [-1, -1,  2,  4]])
>>> Connectivity([[0,1,2],[0,1,3],[2,4,5]])[[0,2]].adjacency('n')
Adjacency([[[-1, -1,  1,  2],
           [-1, -1,  0,  2],
           [ 0,  1,  4,  5],
           [-1, -1, -1, -1],
           [-1, -1,  2,  5],
           [-1, -1,  2,  4]])
>>> Connectivity([[0,1],[2,3]]).adjacency('e')
Adjacency([], shape=(2, 0))

```

adjacentElements (*els, mask=None*)

Compute adjacent elements.

This creates an element adjacency table or node adjacency table. An element i is said to be adjacent to element j , if the two elements have at least one common node. A node i is said to be adjacent to node j , if there is at least one element containing both nodes.

Parameters

- **else** (*int or list of ints*) – The element number(s) for which to compute the adjacent elements
- **mask** (*bool array or int index, optional*) – Node selector. If provided (with `kind=='e'`) this defines by a bool flag array or int index numbers the list of nodes that are to be considered connectors between elements. The default is to consider all nodes as connectors.

This option is only useful in the case `kind == 'e'`. If you want to use an element mask for the 'n' case, just apply the (element) mask beforehand by using `self[mask].adjacency('n')`.

Returns *Adjacency* object – An Adjacency array with shape (nr,nc), where row i holds a sorted list of all the items that are adjacent to item i , padded with -1 values to create an equal

list length for all items.

Examples

```
>>> Connectivity([[0,1],[0,2],[1,3],[0,5]]).adjacentElements([0,1,2,3])
array([[ 1,  2,  3],
       [-1,  0,  3],
       [-1, -1,  0],
       [-1,  0,  1]])
>>> Connectivity([[0,1],[0,2],[1,3],[0,5]]).adjacentElements([0,1,2])
array([[ 1,  2,  3],
       [-1,  0,  3],
       [-1, -1,  0]])
>>> Connectivity([[0,1],[0,2],[1,3],[0,5]]).adjacentElements([1,2,3])
array([[ 0,  3],
       [-1,  0],
       [ 0,  1]])
>>> Connectivity([[0,1],[0,2],[1,3],[0,5]]).adjacentElements([0,2])
array([[ 1,  2,  3],
       [-1, -1,  0]])
>>> Connectivity([[0,1],[0,2],[1,3],[0,5]]).adjacentElements([2])
array([[0]])
>>> Connectivity([[0,1],[0,2],[1,3],[0,5]]).adjacentElements(1)
array([[0, 3]])
```

frontGenerator (*startat=0, frontinc=1, partinc=1*)

Generator function returning the frontal elements.

This is a generator function and is normally not used directly, but via the `frontWalk()` method.

Parameters: see `frontWalk()`.

Returns *int array* – Int array with a value for each element. On the initial call, all values are -1, except for the elements in the initial front, which get a value 0. At each call a new front is created with all the elements that are connected to any of the current front and which have not yet been visited. The new front elements get a value equal to the last front's value plus the `frontinc`. If the front becomes empty and a new starting front is created, the front value is extra incremented with `partinc`.

Examples

```
>>> C = Connectivity([[2,8,7],[2,3,8],[3,9,8],[4,10,9],[5,6,11],
...                 [6,12,11]])
>>> C.adjacentElements([0])
array([[1, 2]])
>>> for p in C.frontGenerator(): print(p)
[ 0 -1 -1 -1 -1 -1]
[ 0  1  1 -1 -1 -1]
[ 0  1  1  2 -1 -1]
[ 0  1  1  2  4 -1]
[0 1 1 2 4 5]
>>> A = C.adjacency()
>>> for p in A.frontGenerator(): print(p)
[ 0 -1 -1 -1 -1 -1]
[ 0  1  1 -1 -1 -1]
[ 0  1  1  2 -1 -1]
```

(continues on next page)

(continued from previous page)

```
[ 0  1  1  2  4 -1]
[0 1 1 2 4 5]
```

frontWalk (*startat=0, frontinc=1, partinc=1, maxval=-1*)

Walks through the elements by their node front.

A frontal walk is executed starting from the given element(s). A number of steps is executed, each step advancing the front over a given number of single pass increments. The step number at which an element is reached is recorded and returned.

Parameters

- **startat** (*int or list of ints*) – Initial element number(s) in the front.
- **frontinc** (*int*) – Increment for the front number on each frontal step.
- **partinc** (*int*) – Increment for the front number when the front gets empty and a new part is started.
- **maxval** (*int*) – Maximum frontal value. If negative (default) the walk will continue until all elements have been reached. If non-negative, walking will stop as soon as the frontal value reaches this maximum.

Returns *int array* – An array of ints specifying for each element in which step the element was reached by the walker.

Examples

```
>>> C = Connectivity([[2,8,7], [2,3,8], [3,9,8], [4,10,9], [5,6,11],
...                 [6,12,11]])
>>> print(C.frontWalk())
[0 1 1 2 4 5]
```

front (*startat=0, add=False*)

Returns the elements of the first node front.

Parameters

- **startat** (*int or list of ints*) – Element number(s) or a list of element numbers. The list of elements to find the next front for.
- **add** (*bool, optional*) – If True, the *startat* elements will be included in the return value. The default (False) will only return the elements in the next front line.

Returns *int array* – A list of the elements that are connected to any of the nodes that are part of the *startat* elements.

Notes

This is equivalent to the first step of a *frontWalk()* with the same *startat* elements, and could thus also be obtained from `where(self.frontWalk(startat,maxval=1) == 1)[0]`.

Here however another implementation is used, which is more efficient for very large models: it avoids the creation of the large array as returned by *frontWalk*.

Examples

```
>>> C = Connectivity([[2,8,7],[2,3,8],[3,9,8],[4,10,9],[5,6,11],
...                 [6,12,11]])
>>> print(C.front([2]))
[0 1 3]
```

`selectNodes` (*selector*)

Return a *Connectivity* containing subsets of the nodes.

Parameters *selector* (*int :term:*) – An object that can be converted to a 1-dim or 2-dim int array. Examples are a tuple of local node numbers, or a list of such tuples all having the same length. Each row of *selector* holds a list of the local node numbers that should be retained in the new Connectivity table. As an example, if the Connectivity is plex-3 representing triangles, a selector `[[0,1],[1,2],[2,0]]` would extract the edges of the triangle.

Returns *Connectivity* – A new Connectivity object with shape `(self.nelems*selector.nelems,selector.nplex)`. Duplicate elements created by the selector are retained. If the selector has an eltype (for example if it is a Connectivity itself), the returned Connectivity will have the same eltype.

Examples

```
>>> Connectivity([[0,1,2],[0,2,1],[0,3,2]]).selectNodes([[0,1],[0,2]])
Connectivity([[0, 1],
              [0, 2],
              [0, 2],
              [0, 1],
              [0, 3],
              [0, 2]])
```

`insertLevel` (*selector*, *permutations='all'*)

Insert an extra hierarchical level in a Connectivity table.

A Connectivity table identifies higher hierarchical entities in function of lower ones. This method inserts an extra level in the hierarchy. For example, if you have volumes defined in function of points, you can insert an intermediate level of edges, or faces. Each element may generate multiple instances of the intermediate level.

Parameters

- **selector** (*int :term:*) – An object that can be converted to a 1-dim or 2-dim int array. Examples are a tuple of local node numbers, or a list of such tuples all having the same length. Each row of *selector* holds a list of the local node numbers that should be retained in the new Connectivity table.
- **permutations** (*str*) – Defines which permutations of the row data are allowed while still considering the rows equal. Equal rows in the intermediate level are collapsed into single items. Possible values are:
 - 'none': no permutations are allowed: rows must match the same data at the same positions.
 - 'roll': rolling is allowed. Rows that can be transformed into each other by rolling are considered equal;
 - 'all': any permutation of the same data will be considered an equal row. This is the default.

Returns

- **hi** (*Connectivity*) – A Connectivity defining the original elements in function of the intermediate level ones.
- **lo** (*Connectivity*) – A Connectivity defining the intermediate level items in function of the lowest level ones (the original nodes). If the `selector` has an `eltype` attribute, then `lo` will inherit the same `eltype` value.
- *The resulting node numbering of the created intermediate entities*
- (the `lo` return value) respects the numbering order of the original
- *elements and the applied the selector, but in case of collapsing*
- *duplicate rows, it is undefined which of the collapsed sequences is*
- *returned.*
- *Because the precise order of the data in the collapsed rows is lost,*
- *it is in general not possible to restore the exact original table*
- *from the two result tables.*
- See however `mesh.Mesh.getBorder()` for an application where an
- *inverse operation is possible, because the border only contains*
- *unique rows.*
- See also `mesh.Mesh.combine()`, which is an almost inverse operation
- *for the general case, if the selector is complete.*
- *The resulting rows may however be permutations of the original.*

Examples

```
>>> Connectivity([[0,1,2],[0,2,1],[0,3,2]]).      insertLevel([[0,1],[1,2],
↪[2,0]])
(Connectivity([[0, 1, 2],
               [2, 1, 0],
               [3, 4, 2]]), Connectivity([[0, 1],
               [1, 2],
               [2, 0],
               [0, 3],
               [3, 2]]))
>>> Connectivity([[0,1,2,3]]).      insertLevel([[0,1,2],[1,2,3],[0,1,1],[0,
↪0,1],[1,0,0]])
(Connectivity([[0, 1, 2, 2, 2]], Connectivity([[0, 1, 2],
               [1, 2, 3],
               [0, 1, 1]]))
```

combine (*lo*)

Combine two hierarchical Connectivity levels to a single one.

`self` and `lo` are two hierarchical Connectivity tables, representing higher and lower level respectively. This means that the elements of `self` hold numbers which point into `lo` to obtain the lowest level items.

In the current implementation, the plexitude of `lo` should be 2!

As an example, in a structure of triangles, `hi` could represent triangles defined by 3 edges and `lo` could represent edges defined by 2 vertices. This method will then result in a table with plexitude 3 defining the triangles in function of the vertices.

This is the inverse operation of `insertLevel()` with a selector which is complete. The algorithm only works if all node numbers of an element are unique.

Examples

```
>>> hi, lo = Connectivity([[0, 1, 2], [0, 2, 1], [0, 3, 2]]) . insertLevel([[0,
↵1], [1, 2], [2, 0]])
>>> hi.combine(lo)
Connectivity([[0, 1, 2],
              [0, 2, 1],
              [0, 3, 2]])
```

`resolve()`

Resolve the connectivity into plex-2 connections.

Creates a Connectivity table with a plex-2 (edge) connection between any two nodes that are connected to a common element.

There is no point in resolving a plexitude 2 structure. Plexitudes lower than 2 can not be resolved.

Returns a plex-2 Connectivity with all connections between node pairs. In each element the nodes are sorted.

Examples

```
>>> print([ i for i in combinations(range(3), 2) ])
[(0, 1), (0, 2), (1, 2)]
>>> Connectivity([[0, 1, 2], [0, 2, 1], [0, 3, 2]]) . resolve()
Connectivity([[0, 1],
              [0, 2],
              [0, 3],
              [1, 2],
              [2, 3]])
```

`sharedNodes (elist=None)`

Return the list of nodes shared by all elements in `elist`

Parameters `elist` (*int :term:*) – List of element numbers. If not specified, all elements are considered.

Returns *int array* – A 1-dim int array with the list of nodes that are common to all elements in the specified list. This array may be empty.

Examples

```
>>> a = Connectivity([[0, 1, 2], [0, 2, 1], [0, 3, 2]])
>>> a.sharedNodes()
array([0, 2])
>>> a.sharedNodes([0, 1])
array([0, 1, 2])
```

replic (*n, inc*)

Repeat a Connectivity with increasing node numbers.

Parameters

- **n** (*int*) – Number of copies to make.
- **inc** (*int*) – Increment in node numbers for each copy.

Returns *Connectivity* – A Connectivity with the concatenation of *n* replicas of *self*, where the first replica is identical to *self* and each next one has its node numbers increased by *inc*.

Examples

```
>>> Connectivity([[0,1,2],[0,2,3]]).replic(2,4)
Connectivity([[0, 1, 2],
              [0, 2, 3],
              [4, 5, 6],
              [4, 6, 7]])
```

chain (*disconnect=None, return_conn=False*)

Reorder the elements into simply connected chains.

Chaining the elements involves reordering them such that the first node of the next element is equal to the last node of the previous. This is especially useful in converting line elements to continuous curves or polylines. It will work with any plexitude though, and only look at the first and last node of the elements in the chaining process.

Parameters

- **disconnect** (*int array_like* | *str*, optional) – List of node numbers where the resulting chains should be split. None of the resulting chains will have any of the listed node numbers as an interior node. A chain may start and end at such a node. A special value ‘branch’ will set the disconnect array to all the nodes owned by more than two elements. This will split all chains at branching points.
- **return_conn** (*bool*) – If True, also return the list of Connectivities corresponding with the chains.

Returns

- **chains** (*list of int arrays*) – A list of tables with the same column length as those in *conn*, and having two columns. The first column contains the original element numbers of a chain, and the second column a value +1 or -1 depending on whether the element traversal in the connected segment is in the original direction (+1) or the reverse (-1). The list of chains is sorted in order of decreasing length.
- **conn** (*list of Connectivity instances*, optional) – Only returned if *return_conn* is True: a list a Connectivity tables of plexitude *nplex* corresponding to each chain. The elements in each Connectivity are ordered to form a continuous connected segment, i.e. the last node of each element in the table is equal to the first node of the following element (if any).

See also:

chained() return only the chained Connectivities

Examples

```

>>> Connectivity([[0,1],[1,2],[0,4],[4,2]]).chain()
[array([[ 0,  1],
        [ 1,  1],
        [ 3, -1],
        [ 2, -1]])]
>>> Connectivity([[0,1],[1,2],[0,4]]).chain()
[array([[ 1, -1],
        [ 0, -1],
        [ 2,  1]])]
>>> Connectivity([[0,1],[0,2],[0,3],[5,4]]).chain()
[array([[ 0, -1],
        [ 1,  1]])],
array([[3,  1]]),
array([[2,  1]])]
>>> Connectivity([[0,1],[0,2],[0,3],[5,4]]).chain(disconnect='branch')
[array([[3,  1]]), array([[2,  1]]), array([[1,  1]]), array([[0,  1]])]
>>> Connectivity([[0,1],[0,2],[0,3],[5,4]]).chain(return_conn=True)
([array([[ 0, -1],
        [ 1,  1]])],
 array([[3,  1]]),
 array([[2,  1]]),
 [Connectivity([[1,  0],
                [0,  2]])],
 Connectivity([[5,  4]]),
 Connectivity([[0,  3]])])
>>> Connectivity([[0,1,2],[2,0,3],[0,3,1],[4,5,2]]).chain()
[array([[ 1, -1],
        [ 0, -1],
        [ 2,  1]])],
array([[3,  1]])]
>>> Connectivity([[0,1,2],[2,0,3],[0,3,1],[4,5,2]]).chain(
...     disconnect=[0])
[array([[0,  1],
        [1,  1]]), array([[3,  1]]), array([[2,  1]])]

```

chained (*disconnect=None*)

Return the Connectivities of the chained elements.

This is a convenience method calling `chain()` with the `return_conn=True` parameter and only returning the second return value. It is equivalent with:

```
self.chain(disconnect, return_conn=True)[1]
```

Examples

```

>>> Connectivity([[0,1],[1,2],[0,4],[4,2]]).chained()
[Connectivity([[0,  1],
               [1,  2],
               [2,  4],
               [4,  0]])]

```

```

>>> Connectivity([[0,1],[1,2],[0,4]]).chained()
[Connectivity([[4,  0],

```

(continues on next page)

(continued from previous page)

```
[0, 1],
[1, 2]])]
```

```
>>> Connectivity([[0,1],[0,2],[0,3],[4,5]]).chained()
[Connectivity([[1, 0],
               [0, 2]]), Connectivity([[4, 5]]), Connectivity([[0, 3]])]
```

```
>>> Connectivity([[0,1],[0,2],[0,3],[5,4]]).chained(disconnect='branch')
[Connectivity([[5, 4]]), Connectivity([[0, 3]]),
Connectivity([[0, 2]]), Connectivity([[0, 1]])]
>>> Connectivity([[0,1,2],[2,0,3],[0,3,1],[4,5,2]]).chained()
[Connectivity([[1, 3, 0],
               [0, 1, 2],
               [2, 0, 3]]),
Connectivity([[4, 5, 2]])]
>>> Connectivity([[0,1,2],[2,0,3],[0,3,1],[4,5,2]],).chained(
...     disconnect=[0])
[Connectivity([[0, 1, 2],
               [2, 0, 3]]),          Connectivity([[4, 5, 2]]),
←Connectivity([[0, 3, 1]])]
```

static connect (*clist*, *nodid*=None, *bias*=None, *loop*=False)

Connect nodes from multiple Connectivity objects.

Parameters

- **clist** (*list of Connectivity objects*) – The Connectivities to connect.
- **nodid** (int *array_like*, optional) – List of node indices, same length as *clist*. This specifies which node of the elements will be used in the connect operation.
- **bias** (int *array_like*, optional) – List of element bias values, same length as *clist*. If provided, then element looping will start at this number instead of at zero.
- **loop** (*bool*) – If False (default), new element generation will stop as soon as the shortest Connectivity runs out of elements. If set to True, the shorter lists will wrap around until all elements of all Connectivities have been used.

Returns *Connectivity* – A Connectivity with plexitude equal to the number of Connectivities in *clist*. Each element of the new Connectivity consist of a node from the corresponding element of each of the Connectivities in *clist*. By default this will be the first node of that element, but a *nodid* list may be given to specify the node index to be used for each of the Connectivities. Finally, a list of bias values may be given to specify an offset in element number for the subsequent Connectivities. If *loop*=False, the length of the Connectivity will be the minimum length of the Connectivities in *clist*, each minus its respective bias. If *loop*=True, the length will be the maximum length in of the Connectivities in *clist*.

Examples

```
>>> a = Connectivity([[0,1],[2,3],[4,5]])
>>> b = Connectivity([[10,11,12],[13,14,15]])
>>> c = Connectivity([[20,21],[22,23]])
>>> print(Connectivity.connect([a,b,c]))
[[ 0 10 20]
 [ 2 13 22]]
```

(continues on next page)

(continued from previous page)

```

>>> print(Connectivity.connect([a,b,c], nodid=[1,0,1]))
[[ 1 10 21]
 [ 3 13 23]]
>>> print(Connectivity.connect([a,b,c], bias=[1,0,1]))
[[ 2 10 22]]
>>> print(Connectivity.connect([a,b,c], bias=[1,0,1], loop=True))
[[ 2 10 22]
 [ 4 13 20]
 [ 0 10 22]]

```

6.2.6 elements — Definition of elements for the Mesh model

This module provides local numbering schemes of element connectivities for *Mesh* models. It allows a consistent numbering throughout pyFormex. When interfacing with other programs, one should be aware that conversions may be necessary. Conversions to/from external programs are done by the interface modules.

The module defines the *ElementType* class and a whole slew of its instances, which are the element types used in pyFormex. Here is also the definition of the *Elms* class, which is a specialisation of the *Connectivity* using an *ElementType* instance as the *eltype*. The *Elms* class is one of the basic data holders in the *Mesh* model.

Classes defined in module elements

class `elements.ElementType` (*name, doc, ndim, vertices, edges=None, faces=None, **kargs*)
 Element types for Mesh models.

ElementType instances store all data that define a particular geometrical entity type and that can not be derived from the plexitude. The class is mostly a storage of provided (and sanitized) data. All successfully created elements are stored in the class-owned *register*, from where they can be looked up by their name.

Parameters

- **name** (*str*) – The name of the element. Case is ignored. It is stored in the *ElementType* instance in capitalized form. The key in the register is the name in all lower case. Usually the name has a numeric last part equal to the plexitude of the element, but this is not a requirement.
- **doc** (*str*) – A short description of the element.
- **ndim** (*int*) – The dimensionality (*level*) of the element (0..3):
 - 0: point
 - 1: line
 - 2: surface
 - 3: volume
- **vertices** (float *array_like*) – The natural coordinates of the nodes (nplex,3). This also defines the plexitude (the number of nodes) of the element and the local node numbering: `range(nplex)`. The vertices of the elements are usually defined in a unit space [0,1] in each axis direction.
- **edges** (*Elms*) – A connectivity table listing the edges of the element with local node numbers. The *eltype* of the *Elms* should be an *ElementType* instance of level 1. *This parameter should be provided if and only if the element is of level 2 or 3.* The edges should be the conceptually correct edges of the element. If the element contains edges of different

plexitudes, they should be specified with the highest plexitude and degenerate elements should be used for the lower plexitude edges.

- **faces** (*Elms*) – A connectivity table listing the faces of the element with local node numbers. The eltype of the Elms should be an `ElementType` instance of level 2. *This parameter should be provided if and only iff the element is of level 3.* The faces should be the conceptually correct faces of the element. If the element contains faces of different plexitudes, they should be specified with the highest plexitude and degenerate elements should be used for the lower plexitude faces.
- ***kargs** (*keyword arguments*) – Other keyword arguments are stored in the `Element` instance as is. These can also be set after initialization by direct assignment to the instance's attribute. Below are some predefined values used by pyFormex.
- **reversed** (int *array_like*) – The order of the nodes for the reversed element. Reversing a line element reverses its parametric direction. Reversing a surface element reverses the direction of the positive normal. Reversing a volume element turns the element inside out (which could possibly be used to represent holes in space).
- **drawgl2faces** (*Elms*) – A connectivity defining the entities to be used in rendering the element. This should only be provided if the rendered geometry should be different from the element itself or from the element `faces`. This is used to draw approximate renderings of elements for which there is no correct functionality available: linear approximations for higher order elements, triangular subdivisions for quadrilaterals. Note that although the name suggests *faces*, this connectivity can be of any level.
- **drawgl2edges** (*Elms*) – A connectivity defining the entities to be used in *wire* rendering of the element. This is like `drawgl2faces` but defines the edges to be rendered in the wireframe and smoothwire and flatwire rendering modes.
- **conversions** (*dict*) – A dict holding the possible strategies for conversion of the element to other element types. The key is the target element name, possibly extended with an extra string to discriminate between multiple strategies leading to the same element. The value is a list of actions to undertake to get from the initial element type to the target element type. See more details in the section *Element type conversion* below.
- **extruded** (*dict*) – A dict with data for how to extrude an element to a higher level element. Extrusion increases the level of the element with one, by creating 1 or 2 new planes of nodes in a new direction. The key in the dict is the degree of the extrusion: 1 or 2. The value is a tuple (eltype, nodes), where eltype is the target element type (an existing `ElementType` instance) and nodes is a list of nodes numbers as they will appear in the new node planes. If nodes is left out of the tuple, they will be ordered exactly like in the original element.
- **degenerate** (*dict*) – A dict with data for how to replace a degenerate element with a reduced element. A degenerate element has one or more coinciding nodes. The reduced elements replaces coinciding nodes with a single node yielding an element with lower plexitude. The keys in the dict are reduced element types. The values are lists of coinciding node conditions and matching reduction schemes: see the section **Degenerate element reduction** below for more details.

Notes

The ordering of the `vertices` defines a fixed local numbering scheme of the nodes in the element. The ordering of the items in `edges` and `faces` attributes specify the local numbering of the edges and faces. For solid elements, it is guaranteed that the vertices of all faces are numbered in a consecutive order spinning positively around the outward normal on the face.

Some of the parameters of an `ElementType` are instances of `Elems`, but `Elems` instances contain themselves an `ElementType` instance. Therefore care should be taken not to define circular dependencies. If the `ElementType` instances are created in order of increasing level, there is no problem with the `edges` and `faces` parameters, as these are of a lower level than the element itself, and will have been defined before. For other attributes however this might not always be the case. These attributes can then be defined by direct assignment, after the needed `ElementTypes` have been initialized.

List of elements

The list of available element types can be found from:

```
>>> ElementType.listall()
Available Element Types:
  0-dimensional elements: ['point']
  1-dimensional elements: ['line2', 'line3', 'line4']
  2-dimensional elements: ['tri3', 'tri6', 'quad4', 'quad6', 'quad8',
↪ 'quad9', 'quad12']
  3-dimensional elements: ['tet4', 'tet10', 'tet14', 'tet15', 'wedge6',
↪ 'hex8', 'hex16', 'hex20', 'hex27', 'octa', 'icosa']
```

Element type conversion

Element type conversion in pyFormex is a powerful feature to transform Mesh type objects. While mostly used to change the element type, there are also conversion types that refine the Mesh.

Available conversion methods are defined in an attribute `conversion` of the input element type. This attribute should be a dictionary, where the keys are the name of the conversion method and the values describe what steps need be taken to achieve this conversion. The method name should be the name of the target element, optionally followed by a suffix to discriminate between different methods yielding the same target element type. The suffix should always start with a '-'. The part starting at the '-' will be stripped of to set the final target element name.

E.g., a 'line3' element type is a quadratic line element through three points. There are two available methods to convert it to 'line2' (straight line segments between two points), named named 'line2', resp. 'line2-2'. The first will transform a 'line3' element in a single 'line2' between the two endpoints (i.e. the chord of the quadratic element); the second will replace each 'line3' with two straight segments: from first to central node, and from central node to end node.

The values in the dictionary are a list of execution steps to be performed in the conversion. Each step is a tuple of a single character defining the type of the step, and the data needed by this type of step. The steps are executed one by one to go from the source element type to the target.

Currently, the following step types are defined:

Type	Data
's' (select)	connectivity list of selected nodes
'a' (average)	list of tuples of nodes to be averaged
'v' (via)	string with name of intermediate element type
'x' (execute)	the name of a proper conversion function
'r' (random)	list of conversion method names

The operation of these methods is as follows:

- 's' (select): This is the most common conversion type. It selects a set of nodes of the input element, and creates one or more new elements with these nodes. The data field is a list of tuples defining for each created element which node numbers from the source element should be included. This method is typically used to reduce the plexitude of the element.
- 'a' (average): Creates new nodes, the position of which is computed as an average of existing nodes. The data field is a list of tuples with the numbers of the nodes that should be averaged for each new node. The

resulting new nodes are added in order at the end of the existing nodes. If this order is not the proper local node numbering, an ‘s’ step should follow to put the (old and new) nodes in the proper order. This method will usually increase the plexitude of the elements.

- ‘v’ (via): The conversion is made via an intermediate element type. The initial element type is first converted to this intermediate type and the result is then transformed to the target type.
- ‘x’ (execute): Calls a function to do the conversion. The function takes the input Mesh as argument and returns the converted Mesh. Currently this function should be a global function in the mesh module. Its name is specified as data in the conversion rule.
- ‘r’ (random): Chooses a random method between a list of alternatives. The data field is a list of conversion method names defined for the same element (and thus inside the same dictionary). While this could be considered an amusement (e.g. used in the Carpetry example), there are serious applications for this, e.g. when transforming a Mesh of squares or rectangles into a Mesh of triangles, by adding one diagonal in each element. Results with such a Mesh may however be different dependent on the choice of diagonal. The converted Mesh has directional preferences, not present in the original. The Quad4 to Tri3 conversion therefore has the choice to use either ‘up’ or ‘down’ diagonals. But a better choice is often the ‘random’ method, which will put the diagonals in a random direction, thus reducing the effect.

Degenerate element reduction

Each element can have an attribute `degenerate`, defining some rules of how the element can be reduced to a lower plexitude element in case it becomes degenerate. An element is said to be degenerate if the same node number is used more than once in the connectivity of a single element. Reduction of degenerate elements is usually only done if the element can be reduced to another element of the same level. For example, if two nodes of a quadrilateral element are coinciding, it could be replaced with a triangle. Both are level 2 elements. When the triangle has two coinciding nodes however, the element is normally not reduced to a line (level 1), but rather completely removed from the (level 2) model. However, nothing prohibits cross-level reductions.

The `degenerate` attribute of an element is a dict where the key is a target element and the corresponding value is a list of reduction rules. Each rule is a tuple of two items: a set of conditions and a node selector to reduce the element. The conditions item is itself a tuple of any number of conditions, where each condition is a tuple of two node indices. If these nodes are equal, the condition is met. If all conditions in a rule are met, the reduction rule is applied. The second item in a rule, the node selector, is an index specifying the indices of the nodes that should be retained in the new (reduced) elements.

As an example, take the Line3 element, which has 3 nodes defining a curved line element. Node 1 is the middle node, and nodes 0 and 2 are the end nodes. The element has four ways of being degenerate: nodes 0 and 1 are the same, nodes 1 and 2 are the same, nodes 0 and 2 are the same, and all three nodes are the same. For the first two of them, a reduction scheme is defined, reducing the element to a straight line segment (Line2) between the two end points:

```
Line3.degenerate = {
    Line2: [((0, 1), ), (0, 2)),
           ((1, 2), ), (0, 2)), ],
}
```

In this case each of the reduction rules contains only a single condition, but there exist cases where multiple conditions have to be met at the same time, which is why the condition (0, 1) is itself enclosed in a tuple. But what about the other degenerate cases. If both end points coincide, it is not clear what to do: reduce to a line segment between the coincident end points, or between an end point and the middle. Here pyFormex made the choice to not reduce such case and leave the degenerate Line3 element. But the user could add a rule to e.g. reduce the case to a line segment between end point and middle point:

```
Line3.degenerate[Line2].append(((0, 2), ), (0, 1))
```

Also the case of three coinciding points is left unhandled. But the user could reduce such cases to a Point:

```
Line3.degenerate[Point] = [(((0, 1), (1, 2)), (0, ))]
```

Here we need two conditions to check that nodes 0, 1 and 2 are equal. However, in this case the user probably also wants the third degenerate case (nodes 0 and 2 are equal) to be reduced to a Point. So he could just use:

```
Line3.degenerate[Point] = [(((0, 2), ), (0, ))]
```

register

This is a class attribute collecting all the created `ElementType` instances with their name in lower case as key.

Type dict

default

A class attribute providing the default `ElementType` for a given plexitude.

Type dict

Examples

```
>>> print(list(ElementType.register.keys()))
['point', 'line2', 'line3', 'line4', 'tri3', 'tri6', 'quad4', 'quad6',
'quad8', 'quad9', 'quad12', 'tet4', 'tet10', 'tet14', 'tet15', 'wedge6',
'hex8', 'hex16', 'hex20', 'hex27', 'octa', 'icosa']
>>> print(ElementType.default)
{1: Point, 2: Line2, 3: Tri3, 4: Quad4, 6: Wedge6, 8: Hex8}
```

nplex()

Return the plexitude of the element

nvertices()

Return the plexitude of the element

nnodes()

Return the plexitude of the element

nedges()

Return the number of edges of the element

nfaces()

Return the number of faces of the element

getEntities(level)

Return the type and connectivity table of some element entities.

Parameters `level` (*int*) – The *level* of the entities to return. If negative, it is a value relative to the level of the caller. If non-negative, it specifies the absolute level. Thus, for an `ElementType` of level 3, `getEntities(-1)` returns the faces, while for a level 2 `ElementType`, it returns the edges. In both cases however, `getEntities(1)` returns the edges.

Returns `Ellems | Connectivity` – The connectivity table and element type of the entities of the specified level. The type is normally `Ellems`. If the requested entity level is outside the range `0..ndim`, an empty `Connectivity` is returned.

Examples

```
>>> Tri3.getEntities(0)
Elems([[0],
       [1],
       [2]], eltype=Point)
>>> Tri3.getEntities(1)
Elems([[0, 1],
       [1, 2],
       [2, 0]], eltype=Line2)
>>> Tri3.getEntities(2)
Elems([[0, 1, 2]], eltype=Tri3)
>>> Tri3.getEntities(3)
Connectivity([], shape=(0, 1))
```

getPoints()

Return the level 0 entities

getEdges()

Return the level 1 entities

getFaces()

Return the level 2 entities

getCells()

Return the level 3 entities

getElement()

Return the element connectivity: the entity of level self.ndim

getDrawFaces(*quadratic=False*)

Return the local connectivity for drawing the element's faces

getDrawEdges(*quadratic=False*)

Return the local connectivity for drawing the element's edges

toMesh()

Convert the element type to a Mesh.

Returns a Mesh with a single element of natural size.

toFormex()

Convert the element type to a Formex.

Returns a Formex with a single element of natural size.

name()

Return the lowercase name of the element.

For compatibility, name() returns the lower case version of the ElementType's name. To get the real name, use the attribute *_name* or format the ElementType as a string.

family()

Return the element family name.

The element family name is the first part of the name that consists only of lower case letter.

classmethod list(*ndim=None, types=False*)

Return a list of available ElementTypes.

Parameters

- **ndim** (*int, optional*) – If provided, only return the elements of the specified level.

- **types** (*bool*, *optional*) – If True, return `ElementType` instances. The default is to return element names.

Returns *list* – A list of `ElementType` names (default) or instances.

Examples

```
>>> ElementType.list()
['point', 'line2', 'line3', 'line4', 'tri3', 'tri6', 'quad4',      'quad6',
 ↪ 'quad8', 'quad9', 'quad12', 'tet4', 'tet10', 'tet14',      'tet15',
 ↪ 'wedge6', 'hex8', 'hex16', 'hex20', 'hex27', 'octa', 'icosa']
>>> ElementType.list(ndim=1)
['line2', 'line3', 'line4']
>>> ElementType.list(ndim=1, types=True)
[Line2, Line3, Line4]
```

classmethod `listall` (*lower=False*)

Print all available element types.

Prints a list of the names of all available element types, grouped by their dimensionality.

static get (*eltype=None*, *nplex=None*)

Find the `ElementType` matching an element name and/or plexitude.

Parameters

- **eltype** (*ElementType* | *str* | *None*) – The element type or name. If not provided and `nplex` is provided, the default element type for that plexitude will be used, if it exists. If a name, it should be the name of one of the existing `ElementType` instances (case insensitive).
- **nplex** (*int*) – The *plexitude* of the element. If provided and an `eltype` was provided, it should match the `eltype` plexitude. If no `eltype` was provided, the default element type for this plexitude is returned.

Returns *ElementType* – The `ElementType` matching the provided `eltype` and/or `nplex`

Raises `ValueError` – If neither name nor `nplex` can resolve into an element type.

Examples

```
>>> ElementType.get('tri3')
Tri3
>>> ElementType.get(nplex=2).name()
'line2'
>>> ElementType.get('QUAD4')
Quad4
```

class `elements.Elems`

A `Connectivity` where the `eltype` is an `ElementType` subclass.

This is used to store the connectivity of a `Mesh` instance. It is also used to store some subitems in the `ElementType`.

```
>>> C = Elems([[0,1,2],[0,1,3],[0,5,3]], 'tri3')
>>> C
Elems([[0, 1, 2],
```

(continues on next page)

(continued from previous page)

```

        [0, 1, 3],
        [0, 5, 3]], eltype=Tri3)
>>> sel = C.levelSelector(1)
>>> sel
Elems([[0, 1],
       [1, 2],
       [2, 0]], eltype=Line2)
>>> C.selectNodes(sel)
Elems([[0, 1],
       [1, 2],
       [2, 0],
       [0, 1],
       [1, 3],
       [3, 0],
       [0, 5],
       [5, 3],
       [3, 0]], eltype=Line2)
>>> C.selectNodes(Elems([[0,1],[0,2]],eltype=Line2))
Elems([[0, 1],
       [0, 2],
       [0, 1],
       [0, 3],
       [0, 5],
       [0, 3]], eltype=Line2)
>>> C.selectNodes([[0,1],[0,2]])
Connectivity([[0, 1],
              [0, 2],
              [0, 1],
              [0, 3],
              [0, 5],
              [0, 3]])
>>> hi, lo = C.insertLevel(1)
>>> hi
Connectivity([[0, 1, 2],
              [0, 3, 4],
              [5, 6, 4]])
>>> lo
Elems([[0, 1],
       [1, 2],
       [2, 0],
       [1, 3],
       [3, 0],
       [0, 5],
       [5, 3]], eltype=Line2)
>>> hi, lo = C.insertLevel([[0,1],[1,2],[2,0]])
>>> hi
Connectivity([[0, 1, 2],
              [0, 3, 4],
              [5, 6, 4]])
>>> lo
Connectivity([[0, 1],
              [1, 2],
              [2, 0],
              [1, 3],
              [3, 0],
              [0, 5],
              [5, 3]])

```

(continues on next page)

(continued from previous page)

```

>>> C = Elems([[0,1,2,3]], 'quad4')
>>> hi, lo = C.insertLevel([[0,1,2], [1,2,3], [0,1,1], [0,0,1], [1,0,0]])
>>> hi
Connectivity([[0, 1, 2, 2, 2]])
>>> lo
Connectivity([[0, 1, 2],
              [1, 2, 3],
              [0, 1, 1]])

```

levelSelector (*level*)

Return a selector for lower level entities.

Parameters *level* (*int*) – An specifying one of the hierarchical levels of element entities. See `Element.getEntities()`.

Returns *Elems* – A new Elems object with shape `(self.nelems*selector.nelems, selector.nplex)`.

selectNodes (*selector*)

Return a Connectivity with subsets of the nodes.

Parameters *selector* (*int* | *int array_like*) – A single int specifies a relative or absolute hierarchical level of element entities (See the Element class). A 2-dim int array selector is then constructed automatically from `self.eltype.getEntities(selector)`.

Else, it is a 2-dim int array like (often a *Connectivity* or another *Elems*. Each row of *selector* holds a list of the local node numbers that should be retained in the new Connectivity table.

As an example, if the Elems is plex-3 representing triangles, a selector `[[0,1],[1,2],[2,0]]` would extract the edges of the triangle. The same would be obtained with `selector=-1` or `selector=1`.

Returns *Elems* | *Connectivity* – An Elems or Connectivity object with `eltype` equal to that of the selector. This means that if the selector has an `eltype` that is one of the elements defined in *elements*, the return type will be Elems, else Connectivity. The shape is `(self.nelems*selector.nelems, selector.nplex)`. Duplicate elements created by the selector are retained.

insertLevel (*selector*, *permutations='all'*)

Insert an extra hierarchical level in a Connectivity table.

A Connectivity table identifies higher hierarchical entities in function of lower ones. This method inserts an extra level in the hierarchy. For example, if you have volumes defined in function of points, you can insert an intermediate level of edges, or faces. Each element may generate multiple instances of the intermediate level.

Parameters

- **selector** (*int* | *int array_like*) – A single int specifies a relative or absolute hierarchical level of element entities (See the Element class). A 2-dim int array selector is then constructed automatically from `self.eltype.getEntities(selector)`.

Else, it is a 2-dim int array like (often a *Connectivity* or another *Elems*. Each row of *selector* holds a list of the local node numbers that should be retained in the new Connectivity table.

- **permutations** (*str*) – Defines which permutations of the row data are allowed while still considering the rows equal. Equal rows in the intermediate level are collapsed into single items. Possible values are:

- 'none': no permutations are allowed: rows must match the same date at the same positions.
- 'roll': rolling is allowed. Rows that can be transformed into each other by rolling are considered equal;
- 'all': any permutation of the same data will be considered an equal row. This is the default.

Returns

- **hi** (*Connectivity*) – A Connectivity defining the original elements in function of the intermediate level ones.
- **lo** (*Elms | Connectivity*) – An Elms or Connectivity object with eltype equal to that of the selector. This means that if the selector has an eltype that is one of the elements defined in *elements*, the return type will be Elms, else Connectivity.
- *The resulting node numbering of the created intermediate entities*
- (the *lo* return value) respects the numbering order of the original
- *elements and the applied the selector, but in case of collapsing*
- *duplicate rows, it is undefined which of the collapsed sequences is*
- *returned.*
- *Because the precise order of the data in the collapsed rows is lost,*
- *it is in general not possible to restore the exact original table*
- *from the two result tables.*
- See however *mesh.Mesh.getBorder()* for an application where an
- *inverse operation is possible, because the border only contains*
- *unique rows.*
- See also *mesh.Mesh.combine()*, which is an almost inverse operation
- *for the general case, if the selector is complete.*
- *The resulting rows may however be permutations of the original.*

reduceDegenerate (*target=None, return_indices=False*)

Reduce degenerate elements to lower plexitude elements.

This will try to reduce the degenerate elements of the Elms to lower plexitude elements. This is only possible if the ElementType has an attribute `degenerate` containing the proper reduction rules.

Parameters

- **target** (*str, optional*) – Target element name. If provided, only reductions to that element type are performed. Else, all the target element types for which a reduction scheme is available, will be tried.
- **return_indices** (*bool, optional*) – If True, also returns the indices of the elements in the input Elms that are contained in each of the parts returned.

Returns

- **conn** (*list of Elms instances*) – A list of Elms of which the first one contains the originally non-degenerate elements, the next one(s) contain the reduced elements (per reduced element type) and the last one contains elements that could not be reduced (this may be absent). In the following cases a list with only the original is returned:

- there are no degenerate elements in the Elems;
- the ElementType does not have any reduction scheme defined;
- the ElementType does not have a reduction scheme for the target.
- **ind** (*list of indices, optional*) – Only returned if `return_indices` is `True`: the indices of the elements of `self` contained in each item in `conn`.

Note: If the Elems is part of a *Mesh*, you should use the `mesh.Mesh.splitDegenerate()` method instead, as that will preserve the property numbers in the resulting Meshes.

The returned reduced Elems may still be degenerate for their own element type.

See also:

`mesh.Mesh.splitDegenerate()` split mesh in non-degenerate and reduced

Examples

```
>>> Elems([[0,1,2],[0,1,3]],eltype=Line3).reduceDegenerate()
[Elems([[0, 1, 2],
        [0, 1, 3]], eltype=Line3)]
>>> C = Elems([[0,1,2],[0,1,1],[0,3,2]],eltype='line3')
>>> C.reduceDegenerate()
[Elems([[0, 1, 2],
        [0, 3, 2]], eltype=Line3),      Elems([[0, 1]], eltype=Line2)]
>>> C = Elems([[0,1,2],[0,1,1],[0,3,2],[1,1,1],[0,0,2]],eltype=Line3)
>>> C.reduceDegenerate()
[Elems([[0, 1, 2],
        [0, 3, 2]], eltype=Line3),      Elems([[1, 1],
        [0, 2],
        [0, 1]], eltype=Line2)]
>>> conn, ind = C.reduceDegenerate(return_indices=True)
>>> conn
[Elems([[0, 1, 2],
        [0, 3, 2]], eltype=Line3),
Elems([[1, 1],
        [0, 2],
        [0, 1]], eltype=Line2)]
>>> ind
[array([0, 2]), array([3, 4, 1])]
```

extrude (*nmod, degree*)

Extrude an Elems to a higher level Elems.

Parameters

- **nmod** (*int*) – Node increment for each new node plane. It should be higher than the highest node number in self.
- **degree** (*int*) – Number of node planes to add. This is also the degree of the extrusion. Currently it is limited to 1 or 2.
- **extrusion adds degree planes of nodes, each with a node** (*The*) –
- **nmod, to the original Elems and then selects** (*increment*) –

- **target nodes from it as defined by the** (*the*) –
- **value.** (*self.etype.extruded[degree]*) –

Returns *Elms* – An *Elms* for the extruded element.

Examples

```
>>> a = Elms([[0,1],[1,2]],etype=Line2).extrude(3,1)
>>> print(a)
[[0 1 4 3]
 [1 2 5 4]]
>>> print(a.etype.name())
quad4
>>> a = Elms([[0,1,2],[0,2,3]],etype=Line3).extrude(4,2)
>>> print(a)
[[ 0  2 10  8  1  6  9  4  5]
 [ 0  3 11  8  2  7 10  4  6]]
>>> print(a.etype.name())
quad9
```

6.2.7 field — Field data in pyFormex.

This module defines the `Field` class, which can be used to describe scalar and vectorial field data over a geometrical domain.

class `field.Field`(*geometry, fldtype, data, fldname=None*)

Scalar or vectorial field data defined over a geometric domain.

A scalar data field is a quantity having exactly one value in each point of some geometric domain. A vectorial field is a quantity having exactly *nval* values at each point, where *nval* ≥ 1 . A vectorial field can also be considered as a collection of *nval* scalar fields: as far as the `Field` class is concerned, there is no relation between the *nval* components of a vectorial field.

The definition of a field is always tied to some geometric domain. Currently `pyFormex` allows fields to be defined on `Formex` and `Mesh` type geometries.

Fields should only be defined on geometries whose topology does not change anymore. This means that for `Formex` type, the shape of the *coords* attribute should not be changed, and for `Mesh` type, the shape of *coords* and the full contents of *elems* should not be changed. It is therefore best to only add field data to geometry objects that will not be changed in place. Nearly all methods in `pyFormex` return a copy of the object, and the copy currently loses all the fields defined on the parent. In future however, selected transformations may inherit fields from the parent.

While `Field` class instances are usually created automatically by the `Geometry.addField()` method of some `Geometry`, it is possible to create `Field` instances yourself and manage the objects like you want. The `Fields` stored inside `Geometry` objects have some special features though, like being exported to a `PGF` file together with the geometry.

A `Field` is defined by the following parameters:

- *geometry*: `Geometry` instance: describes the geometrical domain over which the field is defined. Currently this has to be a `Formex` or a `Mesh`.
- *fldtype*: string: one of the following predefined field types:
 - ‘node’: the field data are specified at the nodes of the geometry;
 - ‘elemc’: the field data are constant per element;

- ‘elemn’: the field data vary over the element and are specified at the nodes of the elements;
- ‘elemg’: the field data are specified at a number of points of the elements, from which they can be inter- or extrapolated;

The actually available field types depend on the type of the *geometry* object. Formex type has only ‘elemc’ and ‘elemn’. *Mesh* currently has ‘node’, ‘elemc’ and ‘elemn’.

- *data*: an array with the field values defined at the specified points. The required shape of the array depends on *fldtype*:
 - ‘node’: (nnodes,) or (nnodes, nval)
 - ‘elemc’: (nelems,) or (nelems, nval)
 - ‘elemn’: (nelems, nplex) or (nelems, nplex, nval)
 - ‘elemg’: (nelems, ngp) or (nelems, ngp, nval)
- *fldname*: string: the name used to identify the field. Fields stored in a Geometry object can be retrieved using this name. See `Geometry.getField()`. If no name is specified, one is generated.

Example:

```
>>> from pyformex.formex import Formex
>>> M = Formex('4:0123').replic(2).toMesh()
>>> print(M.coords)
[[ 0.  0.  0.]
 [ 0.  1.  0.]
 [ 1.  0.  0.]
 [ 1.  1.  0.]
 [ 2.  0.  0.]
 [ 2.  1.  0.]]
>>> print(M.elems)
[[0 2 3 1]
 [2 4 5 3]]
>>> d = M.coords.distanceFromPlane([0.,0.,0.],[1.,0.,0.])
>>> f1 = Field(M,'node',d)
>>> print(f1)
Field 'field-0', type 'node', shape (6,), nnodes=6, nelems=2, nplex=4
[ 0.  0.  1.  1.  2.  2.]
>>> f2 = f1.convert('elemn')
>>> print(f2)
Field 'field-1', type 'elemn', shape (2, 4), nnodes=6, nelems=2, nplex=4
[[ 0.  1.  1.  0.]
 [ 1.  2.  2.  1.]]
>>> f3 = f2.convert('elemc')
>>> print(f3)
Field 'field-2', type 'elemc', shape (2,), nnodes=6, nelems=2, nplex=4
[ 0.5  1.5]
>>> d1 = M.coords.distanceFromPlane([0.,0.,0.],[0.,1.,0.])
>>> f4 = Field(M,'node',at.column_stack([d,d1]))
>>> print(f4)
Field 'field-3', type 'node', shape (6, 2), nnodes=6, nelems=2, nplex=4
[[ 0.  0.]
 [ 0.  1.]
 [ 1.  0.]
 [ 1.  1.]
 [ 2.  0.]
 [ 2.  1.]]
>>> f5 = f4.convert('elemn')
```

(continues on next page)

(continued from previous page)

```

>>> print(f5)
Field 'field-4', type 'elemn', shape (2, 4, 2), nnodes=6, nelems=2, nplex=4
[[[ 0. 0.]
   [ 1. 0.]
   [ 1. 1.]
   [ 0. 1.]]
<BLANKLINE>
  [[ 1. 0.]
   [ 2. 0.]
   [ 2. 1.]
   [ 1. 1.]]]
>>> f6 = f5.convert('elemc')
>>> print(f6)
Field 'field-5', type 'elemc', shape (2, 2), nnodes=6, nelems=2, nplex=4
[[ 0.5 0.5]
 [ 1.5 0.5]]
>>> print(f3.convert('elemn'))
Field 'field-6', type 'elemn', shape (2, 4), nnodes=6, nelems=2, nplex=4
[[ 0.5 0.5 0.5 0.5]
 [ 1.5 1.5 1.5 1.5]]
>>> print(f3.convert('node'))
Field 'field-8', type 'node', shape (6, 1), nnodes=6, nelems=2, nplex=4
[[ 0.5]
 [ 0.5]
 [ 1. ]
 [ 1. ]
 [ 1.5]
 [ 1.5]]
>>> print(f6.convert('elemn'))
Field 'field-9', type 'elemn', shape (2, 4, 2), nnodes=6, nelems=2, nplex=4
[[[ 0.5 0.5]
   [ 0.5 0.5]
   [ 0.5 0.5]
   [ 0.5 0.5]]
<BLANKLINE>
  [[ 1.5 0.5]
   [ 1.5 0.5]
   [ 1.5 0.5]
   [ 1.5 0.5]]]
>>> print(f6.convert('node'))
Field 'field-11', type 'node', shape (6, 2), nnodes=6, nelems=2, nplex=4
[[ 0.5 0.5]
 [ 0.5 0.5]
 [ 1.  0.5]
 [ 1.  0.5]
 [ 1.5 0.5]
 [ 1.5 0.5]]

```

comp (*i*)

Return the data component *i* of a vectorial Field.

Parameters:

- *i*: int: component number of a vectorial Field. If the Field is a scalar one, any value will return the full scalar data.

Returns an array representing the scalar data over the Geometry.

convert (*toype*, *toname=None*)

Convert a Field to another type.

Parameters:

- *totype*: string: target field type. Can be any of the available field types. See *Field* class. If the target type is equal to the source type, a copy of the original Field will result. This may or may not be a shallow copy.
- *toname*: string: the name of the target field. If not specified, a autoname is generated.

Returns a Field of type *totype*.

6.2.8 `utils` — A collection of miscellaneous utility functions.

The `pyformex.utils` module contains a wide variety of utility functions. Because there are so many and they are so widely used, the `utils` module is imported in the environment where scripts and apps are executed, so that users can always call the `utils` functions without explicitly importing the module.

Classes defined in module `utils`

class `utils.TempDir` (*suffix=None, prefix=None, dir=None*)

A temporary directory that can be used as a context manager.

This is a wrapper around Python's `tempfile.TemporaryDirectory`. The difference is that it has an extra `.path` attribute returning the directory name as a `Path`, and the context manager also returns a `Path` instead of a `str`.

class `utils.File` (*filename, mode, compr=None, level=5, delete_temp=True*)

Transparent file compression.

This class is a context manager providing transparent file compression and decompression. It is commonly used in a *with* statement, as follows:

```
with File('filename.ext', 'w') as f:
    f.write('something')
    f.write('something more')
```

This will create an uncompressed file with the specified name, write some things to the file, and close it. The file can be read back similarly:

```
with File('filename.ext', 'r') as f:
    for line in f:
        print(f)
```

Because `File` is a context manager, the file is closed automatically when leaving the *with* block.

By specifying a filename ending with `.gz` or `.bz2`, the file will be compressed (on writing) or decompressed (on reading) automatically. The code can just stay the same as above.

Parameters:

- *filename*: string: name of the file to open. If the filename ends with `.gz` or `.bz2`, transparent (de)compression will be used, with `gzip` or `bzip2` compression algorithms respectively.
- *mode*: string: file open mode: `r` for read, `w` for write or `a` for append mode. See also the documentation for Python's `open` function. For compressed files, append mode is not yet available.
- *compr*: string, one of `gz` or `bz2`: identifies the compression algorithm to be used: `gzip` or `bzip2`. If the file name is ending with `.gz` or `.bz2`, *compr* is set automatically from the extension.

- *level*: an integer from 1..9: gzip/bzip2 compression level. Higher values result in smaller files, but require longer compression times. The default of 5 gives already a fairly good compression ratio.
- *delete_temp*: bool: if True (default), the temporary files needed to do the (de)compression are deleted when the File instance is closed.

The File class can also be used outside a *with* statement. In that case the user has to open and close the File himself. The following are more or less equivalent with the above examples (the *with* statement is better at handling exceptions):

```
fil = File('filename.ext', 'w')
f = fil.open()
f.write('something')
f.write('something more')
fil.close()
```

This will create an uncompressed file with the specified name, write some things to the file, and close it. The file can be read back similarly:

```
fil = File('filename.ext', 'r')
f = fil.open()
for line in f:
    print(f)
fil.close()
```

open()

Open the File in the requested mode.

This can be used to open a File object outside a *with* statement. It returns a Python file object that can be used to read from or write to the File. It performs the following:

- If no compression is used, open the file in the requested mode.
- For reading a compressed file, decompress the file to a temporary file and open the temporary file for reading.
- For writing a compressed file, open a temporary file for writing.

See the documentation for the *File* class for an example of its use.

close()

Close the File.

This can be used to close the File if it was not opened using a *with* statement. It performs the following:

- The underlying file object is closed.
- If the file was opened in write or append mode and compression is requested, the file is compressed.
- If a temporary file was in use and *delete_temp* is True, the temporary file is deleted.

See the documentation for the *File* class for an example of its use.

reopen (mode='r')

Reopen the file, possibly in another mode.

This allows e.g. to read back data from a just saved file without having to destroy the File instance.

Returns the open file object.

class `utils.NameSequence (name, ext=)`

A class for autogenerating sequences of names.

Sequences of names are autogenerated by combining a fixed part with a numeric part. The latter is incremented at each creation of a new name (by the `next()` function).

Parameters

- **name** (*str*) – Base of the names to be generated. The name is split in three parts (prefix, numeric, suffix), where numeric only contains digits and suffix does not contain any digits. Thus, numeric is the last numeric part in the string. The prefix and suffix are invariable parts, while the numeric part will be incremented starting from the value in the provided name. Use `ext` if the variable part is not the last numeric part of name. If `name` does not contain any numeric part, it is split as a file name in stem and suffix, and `'-0'` is appended to the stem. If `name` is empty, it will be replaced with `'0'`.
- **ext** (*str, optional*) – If provided, this is an invariable string added to the suffix from `name` to construct the full name template. This may contain numeric parts, allowing the variable numeric part at any place in the full template.

Examples

```
>>> N = NameSequence('obj')
>>> [ next(N) for i in range(3) ]
['obj-0', 'obj-1', 'obj-2']
>>> N.peek()
'obj-3'
>>> next(N), next(N)
('obj-3', 'obj-4')
>>> N.template
'obj-%d'
>>> N = NameSequence('obj-005')
>>> [ next(N) for i in range(3) ]
['obj-005', 'obj-006', 'obj-007']
>>> N = NameSequence('abc.98')
>>> [ next(N) for i in range(3) ]
['abc.98', 'abc.99', 'abc.100']
>>> N = NameSequence('abc-8x.png')
>>> [ next(N) for i in range(3) ]
['abc-8x.png', 'abc-9x.png', 'abc-10x.png']
>>> N.template
'abc-%01dx.png'
>>> N.glob()
'abc-*x.png'
>>> next(NameSequence('abc', '.png'))
'abc-0.png'
>>> next(NameSequence('abc.png'))
'abc-0.png'
>>> N = NameSequence('/home/user/abc23', '5.png')
>>> [ next(N) for i in range(2) ]
['/home/user/abc235.png', '/home/user/abc245.png']
>>> N = NameSequence('')
>>> next(N), next(N)
('0', '1')
>>> N = NameSequence('12')
>>> next(N), next(N)
('12', '13')
```

`next()`

Return the next name in the sequence

peek()

Return the next name in the sequence without incrementing.

glob()

Return a UNIX glob pattern for the generated names.

A NameSequence is often used as a generator for file names. The glob() method returns a pattern that can be used in a UNIX-like shell command to select all the generated file names.

class `utils.DictDiff` (*current_dict*, *past_dict*)

A class to compute the difference between two dictionaries

Parameters:

- *current_dict*: dict
- *past_dict*: dict

The differences are reported as sets of keys: - items added - items removed - keys same in both but changed values - keys same in both and unchanged values

added()

Return the keys in *current_dict* but not in *past_dict*

removed()

Return the keys in *past_dict* but not in *current_dict*

changed()

Return the keys for which the value has changed

unchanged()

Return the keys with same value in both dicts

equal()

Return True if both dicts are equivalent

Functions defined in module `utils`

`utils.filterWarning` (*message*, *module=""*, *category='U'*, *action='ignore'*)

Add a warning message to the warnings filter.

category can be a Warning subclass or a key in the `_warn_category` dict

`utils.warning` (*message*, *level=<class 'UserWarning'>*, *stacklevel=3*)

Decorator to add a warning to a function.

Adding this decorator to a function will warn the user with the supplied message when the decorated function gets executed for the first time in a session. An option is provided to switch off this warning in future sessions.

Decorating a function is done as follows:

```
@utils.warning('This is the message shown to the user')
def function(args):
    ...
```

`utils.deprecated` (*message*, *stacklevel=4*)

Decorator to deprecate a function

This is like `warning()`, but the level is set to `FutureWarning`.

`utils.deprecated_by` (*old*, *new*, *stacklevel=4*)

Decorator to deprecate a function by another one.

Adding this decorator to a function will warn the user with a message that the *old* function is deprecated in favor of *new*, at the first execution of *old*.

See also: `deprecated()`.

`utils.deprecated_future()`

Decorator to warn that a function may be deprecated in future.

See also: `deprecated()`.

`utils.system(cmd, timeout=None, wait=True, verbose=False, raise_error=False, **kargs)`

Execute an external command.

This and the more user oriented `command()` are the preferred ways to execute external commands from inside pyFormex.

Parameters

- **cmd** (*str*) – The command to be executed
- **timeout** (*float, optional*) – If specified and > 0.0, the command will time out and be terminated or killed after the specified number of seconds.
- **wait** (*bool*) – If True (default), the caller waits for the process to terminate. Setting this value to False will allow the caller to continue immediately, but it will not be able to retrieve the standard output and standard error of the process.
- **verbose** (*bool*) – If True, some extra informative messages are printed:
 - the command that will be run,
 - an occurring timeout condition,
 - in case of a nonzero exit, the full stdout, exit status and stderr.
- **raise_error** (*bool.*) – If True, and verbose is True, and the command fails to execute or returns with a nonzero return code other than one cause by a timeout, an error is raised.
- **kargs** (*optional keyword parameters*) – Additional parameters that are passed to the Popen constructor. See the Python documentation for full info. Some often used parameters are below.
- **shell** (*bool*) – The default (False) is to run the command as a subprocess. This has limitations however, as it will only accept parameters that are processed by that command. Typically, you can not use composite commands, I/O redirection, glob expansions.

With `shell=True`, the command is run in a new shell. The `cmd` should then be specified exactly as it would be entered in a shell, and can contain anything the shell will accept.
- **stdout** (*open file object*) – If specified, the standard output of the command will be written to that file.
- **stdin** (*open file object*) – If specified, the standard input of the command will be read from that file.

Returns `Process` object. – The `Process` used to run the command. This gives access to all its info, like the exit code, stdout and stderr, and whether the command timed out or not. See `Process` for more info.

Notes

The returned `Process` is also saved as the global variable `last_command` to allow checking the outcome of the command from other places but the caller.

`utils.command(cmd, verbose=True, raise_error=True, **kargs)`

Run an external command in a user friendly way.

This is equivalent with the `system()` function but has `verbose=True` and `raise_error=True` options on by default.

`utils.lastCommandReport()`

Produce a report about the last run external command.

Returns *str* – An extensive report about the last run command, including output and error messages.

Notes

This is mostly useful in interactive work, to find out why a command failed.

`utils.killProcesses(pids, signal=15)`

Send the specified signal to the processes in list

Parameters

- **pids** (*list of int*) – List of process ids to be killed.
- **signal** (*int*) – Signal to be send to the processes. The default (15) will try to terminate the process in a friendly way. See `man kill` for more values.

`utils.execSource(script, glob={})`

Execute Python code in another thread.

Parameters

- **script** (*str*) – A string containing some executable Python/pyFormex code.
- **glob** (*dict, optional*) – A dict with globals specifying the environment in which the source code is executed.

`utils.matchMany(regexps, target)`

Return multiple regular expression matches of the same target string.

`utils.matchCount(regexps, target)`

Return the number of matches of target to regexps.

`utils.matchAny(regexps, target)`

Check whether target matches any of the regular expressions.

`utils.matchNone(regexps, target)`

Check whether target matches none of the regular expressions.

`utils.matchAll(regexps, target)`

Check whether targets matches all of the regular expressions.

`utils.fileDescription(ftype, compr=False)`

Return a description of the specified file type(s).

Parameters **ftype** (*str or list of str*) – The file type (or types) for which a description is requested. The case of the string(s) is ignored: it is converted to lower case.

Returns

str of list of str – The file description(s) corresponding with the specified file type(s). The return value(s) depend(s) on the value of the input string(s) in the the following way (see Examples below):

- if it is a key in the `file_description` dict, the corresponding value is returned;

- if it is a string of only alphanumerical characters: it is interpreted as a file extension and the corresponding return value is FTYPE files (*.ftype);
- any other string is returned as as: this allows the user to compose his filters himself.

See also:

`splitFileDescription()`

Examples

```
>>> fileDescription('img')
'Images (*.png *.jpg *.jpeg *.eps *.gif *.bmp)'
>>> fileDescription(['stl','all'])
['STL files (*.stl)', 'All files (*)']
>>> fileDescription('inp')
'Abaqus or CalCuliX input files (*.inp)'
>>> fileDescription('doc')
'DOC files (*.doc)'
>>> fileDescription('*.inp')
'*.inp'
>>> fileDescription('pgf', compr=True)
'pyFormex geometry files (*.pgf *.pgf.gz *.pgf.bz2)'
```

This lists all known types in pyFormex:

```
>>> print(formatDict(file_description))
all = 'All files (*)'
ccx = 'CalCuliX files (*.dat *.inp)'
dcm = 'DICOM images (*.dcm)'
dxf = 'AutoCAD .dxf files (*.dxf)'
dxfall = 'AutoCAD .dxf or converted (*.dxf *.dxfext)'
dxfext = 'Converted AutoCAD files (*.dxfext)'
flavia = 'flavia results (*.flavia.msh *.flavia.res)'
gts = 'GTS files (*.gts)'
gz = 'Compressed files (*.gz *.bz2)'
html = 'Web pages (*.html)'
icon = 'Icons (*.xpm)'
img = 'Images (*.png *.jpg *.jpeg *.eps *.gif *.bmp)'
inp = 'Abaqus or CalCuliX input files (*.inp)'
neu = 'Gambit Neutral files (*.neu)'
obj = 'Wavefront OBJ files (*.obj)'
off = 'Geomview object files (*.off)'
pgf = 'pyFormex geometry files (*.pgf)'
ply = 'Stanford Polygon File Format files (*.ply)'
png = 'PNG images (*.png)'
postproc = 'Postproc scripts (*.post.py *.post)'
pyformex = 'pyFormex scripts (*.py *.pye)'
pyf = 'pyFormex projects (*.pyf)'
pzf = 'pyFormex zip files (*.pzf)'
smesh = 'Tetgen surface mesh files (*.smesh)'
stl = 'STL files (*.stl)'
stlb = 'Binary STL files (*.stl)'
surface = 'Surface models (*.off *.gts *.stl *.smesh *.vtp *.vtk)'
tetgen = 'Tetgen files (*.poly *.smesh *.ele *.face *.edge *.node *.neigh)'
vtk = 'All VTK types (*.vtk *.vtp)'
vtp = 'vtkPolyData file (*.vtp)'
<BLANKLINE>
```

`utils.splitFileDescription` (*fdesc*, *compr=False*)

Split a file descriptor.

A file descriptor is a string consisting of an initial part followed by a second part enclosed in parentheses. The second part is a space separated list of glob patterns. An example file descriptor is 'file type text (*.ext1 *.ext2)'. The values of `file_description` all have this format.

This function splits the file descriptor in two parts: the leading text and a list of global patterns.

Parameters

- **fdesc** (*str*) – A file descriptor string.
- **compr** (*bool, optional*) – If True, the compressed file types are automatically added.

Returns

- **desc** (*str*) – The file type description text.
- **ext** (*list of str*) – A list of the matching extensions for this type. Each string starts with a '.'.

See also:

`fileDescription()` return the file descriptor from file type

`fileExtensionsFromFilter()` return only the list of extensions

Examples

```
>>> splitFileDescription(file_description['img'])
('Images ', ['.png', '.jpg', '.jpeg', '.eps', '.gif', '.bmp'])
>>> splitFileDescription(file_description['pgf'], compr=True)
('pyFormex geometry files ', ['.pgf', '.pgf.gz', '.pgf.bz2'])
```

`utils.fileExtensionsFromFilter` (*fdesc*, *compr=False*)

Return the list of file extensions for a given file type.

Parameters

- **fdesc** (*str*) – A file descriptor string.
- **compr** (*bool, optional*) – If True, the compressed file types are automatically added.

Returns *list of str* – A list of the matching extensions for this type. Each string starts with a '.'.

Examples

```
>>> fileExtensionsFromFilter(file_description['ccx'])
['.dat', '.inp']
>>> fileExtensionsFromFilter(file_description['ccx'], compr=True)
['.dat', '.dat.gz', '.dat.bz2', '.inp', '.inp.gz', '.inp.bz2']
```

`utils.fileExtensions` (*ftype*, *compr=False*)

Return the list of file extensions from a given type.

Parameters

- **ftype** (*str*) – The file type (see `fileDescription()`).
- **compr** (*bool, optional*) – If True, the compressed file types are automatically added.

Returns *list of str* – A list of the matching extensions for this type. Each string starts with a '.'.

Examples

```
>>> fileExtensions('pgf')
['.pgf']
>>> fileExtensions('pgf', compr=True)
['.pgf', '.pgf.gz', '.pgf.bz2']
```

`utils.fileTypes` (*ftype*, *compr=False*)

Return the list of file extension types for a given type.

Parameters

- **ftype** (*str*) – The file type (see `fileDescription()`).
- **compr** (*bool, optional*) – If True, the compressed file types are automatically added.

Returns *list of str* – A list of the normalized matching extensions for this type. Normalized extension do not have the leading dot and are lower case only.

Examples

```
>>> fileTypes('pgf')
['pgf']
>>> fileTypes('pgf', compr=True)
['pgf', 'pgf.gz', 'pgf.bz2']
```

`utils.addCompressedTypes` (*ext*)

Add the defined compress types to a list of extensions

Parameters *ext* (*list of str*) – A list a filename extensions

Returns *list of str* – The list of filename extensions echo extended with the corresponding compressed types.

Examples

```
>>> addCompressedTypes(['.ccx', '.inp'])
['.ccx', '.ccx.gz', '.ccx.bz2', '.inp', '.inp.gz', '.inp.bz2']
```

`utils.projectName` (*fn*)

Derive a project name from a file name.

The project name is the basename of the file without the extension. It is equivalent with `Path(fn).stem`

Examples

```
>>> projectName('aa/bb/cc.dd')
'cc'
>>> projectName('cc.dd')
'cc'
>>> projectName('cc')
'cc'
```

`utils.findIcon` (*name*)

Return the file name for an icon with given name.

Parameters `name` (*str*) – Name of the icon: this is the stem of the filename.

Returns *str* – The full path name of an icon file with the specified name, found in the pyFormex icon folder, or the question mark icon file, if no match was found.

Examples

```
>>> print(findIcon('view-xr-yu').relative_to(pf.cfg['pyformexdir']))
icons/view-xr-yu.xpm
>>> print(findIcon('right').relative_to(pf.cfg['pyformexdir']))
icons/64x64/right.png
>>> print(findIcon('xyz').relative_to(pf.cfg['pyformexdir']))
icons/question.xpm
```

`utils.listIconNames` (*dirs=None, types=None*)

Return the list of available icons by their name.

Parameters

- **dirs** (*list of paths, optional*) – If specified, only return icons names from these directories.
- **types** (*list of strings, optional*) – List of file suffixes, each starting with a dot. If specified, Only names of icons having one of these suffixes are returned.

Returns *list of str* – A sorted list of the icon names available in the pyFormex icons folder.

Examples

```
>>> listIconNames()[:4]
['clock', 'dist-angle', 'down', 'down']
>>> listIconNames([pf.cfg['icondir'] / '64x64'])[:4]
['down', 'ff', 'info', 'lamp']
>>> listIconNames(types=['.xpm'])[:4]
['clock', 'dist-angle', 'down', 'far']
```

`utils.sourceFiles` (*relative=False, symlinks=True, extended=False*)

Return the list of pyFormex .py source files.

Parameters

- **relative** (*bool*) – If True, returned filenames are relative to the current directory.
- **symlinks** (*bool*) – If False, files that are symbolic links are retained in the list. The default is to remove them.
- **extended** (*bool*) – If True, also return the .py files in all the paths in the configured appdirs and scriptdirs.

Returns *list of str* – A list of filenames of .py files in the pyFormex source tree, and, if `extended` is True, .py files in the configured app and script dirs as well.

`utils.grepSource` (*pattern, options=""*, *relative=True*)

Finds pattern in the pyFormex source files.

Uses the `grep` program to find all occurrences of some specified pattern text in the pyFormex source .py files (including the examples). Extra options can be passed to the `grep` command. See *man grep* for more info.

Returns the output of the `grep` command.

`utils.moduleList` (*package='all'*)

Return a list of all pyFormex modules in a subpackage.

This is like `sourceFiles()`, but returns the files in a Python module syntax.

`utils.diskSpace` (*path, units=None, ndigits=2*)

Returns the amount of disk space of a file system.

Parameters

- **path** (*path_like*) – A path name inside the file system to be probed.
- **units** (*str*) – If provided, results are reported in this units. See `humanSize()` for possible values. The default is to return the number of bytes.
- **ndigits** (*int*) – If provided, and also `units` is provided, specifies the number of decimal digits to report. See `humanSize()` for details.

Returns

- **total** (*int | float*) – The total disk space of the file system containing `path`.
- **used** (*int | float*) – The used disk space on the file system containing `path`.
- **available** (*int | float*) – The available disk space on the file system containing `path`.

Notes

The sum `used + available` does not necessarily equal `total`, because a file system may (and usually does) have reserved blocks.

`utils.humanSize` (*size, units, ndigits=-1*)

Convert a number to a human size.

Large numbers are often represented in a more human readable form using k, M, G prefixes. This function returns the input size as a number with the specified prefix.

Parameters

- **size** (*int or float*) – A number to be converted to human readable form.
- **units** (*str*) – A string specifying the target units. The first character should be one of k,K,M,G,T,P,E,Z,Y. 'k' and 'K' are equivalent. A second character 'i' can be added to use binary (K=1024) prefixes instead of decimal (k=1000).
- **ndigits** (*int, optional*) – If provided and ≥ 0 , the result will be rounded to this number of decimal digits.

Returns *float* – The input value in the specified units and possibly rounded to `ndigits`.

Examples

```
>>> humanSize(1234567890, 'k')
1234567.89
>>> humanSize(1234567890, 'M', 0)
1235.0
>>> humanSize(1234567890, 'G', 3)
1.235
>>> humanSize(1234567890, 'Gi', 3)
1.15
```

`utils.TempFile (*args, **kwargs)`

Return a temporary file that can be used as a context manager.

This is a wrapper around Python's `tempfile.NamedTemporaryFile`. The difference is that the returned object has an extra `.path` attribute holding the file name as a `Path`.

`utils.zipList (filename)`

List the files in a zip archive

Returns a list of file names

`utils.zipExtract (filename, members=None)`

Extract the specified member(s) from the zip file.

The default extracts all.

`utils.setSaneLocale (localestring=)`

Set a sane local configuration for `LC_NUMERIC`.

localestring is the locale string to be set, e.g. 'en_US.UTF-8' or 'C' for no locale.

Sets the `LC_ALL` locale to the specified string if that is not empty, and (always) sets `LC_NUMERIC` and `LC_COLLATE` to 'C'.

Changing the `LC_NUMERIC` setting is a very bad idea! It makes floating point values to be read or written with a comma instead of a the decimal point. Of course this makes input and output files completely incompatible. You will often not be able to process these files any further and create a lot of troubles for yourself and other people if you use an `LC_NUMERIC` setting different from the standard.

Because we do not want to help you shoot yourself in the foot, this function always sets `LC_NUMERIC` back to a sane 'C' value and we call this function when pyFormex is starting up.

`utils.strNorm (s)`

Normalize a string.

Text normalization removes all '&' characters and converts it to lower case.

```
>>> strNorm("&MenuItem")
'menuitem'
```

`utils.slugify (text, delim='-')`

Convert a string into a URL-ready readable ascii text.

Example: `>>> slugify("http://example.com/blog/[]Some] _ Article's Title-")` 'http-example-com-blog-some-article-s-title' `>>> slugify("&MenuItem")` 'menuitem'

`utils.forceReST (text, underline=False)`

Convert a text string to have it recognized as `reStructuredText`.

Returns the text with two lines prepended: a line with `..'` and a blank line. The text display functions will then recognize the string as being `reStructuredText`. Since the `..'` starts a comment in `reStructuredText`, it will not be displayed.

Furthermore, if *underline* is set `True`, the first line of the text will be underlined to make it appear as a header.

`utils.underlineHeader (s, char='-')`

Underline the first line of a text.

Adds a new line of text below the first line of *s*. The new line has the same length as the first, but all characters are equal to the specified char.

```
>>> print(underlineHeader("Hello World"))
Hello World
-----
```

`utils.framedText` (*text*, *padding*=[0, 2, 0, 2], *border*=[1, 2, 1, 2], *margin*=[0, 0, 0, 0], *borderchar*='####', *cornerchar*=None, *width*=None, *adjust*='l')

Create a text with a frame around it.

- *adjust*: 'l', 'c' or 'r': makes the text lines be adjusted to the left, center or right.

```
>>> print(framedText("Hello World,\nThis is me calling", adjust='c'))
#####
##      Hello World,      ##
##  This is me calling  ##
#####
```

`utils.prefixText` (*text*, *prefix*)

Add a prefix to all lines of a text.

- *text*: multiline string
- *prefix*: string: prefix to insert at the start of all lines of text.

```
>>> print(prefixText("line1\nline2", "** "))
** line1
** line2
```

`utils.versaText` (*obj*)

Versatile text creator

This functions converts any input into a (multiline) string. Currently, different inputs are treated as follows:

- string: return as is,
- function: use the output of the function as input,
- anything else: use `str()` or `repr()` on the object.

```
>>> versaText("Me")
'Me'
>>> versaText(1)
'1'
>>> versaText(len("Me"))
'2'
>>> versaText({1:"Me"})
"{1: 'Me'}"
```

`utils.dos2unix` (*infile*)

Convert a text file to unix line endings.

`utils.unix2dos` (*infile*, *outfile*=None)

Convert a text file to dos line endings.

`utils.gzip` (*filename*, *gzipped*=None, *remove*=True, *level*=5, *compr*='gz')

Compress a file in gzip/bzip2 format.

Parameters:

- *filename*: input file name
- *gzipped*: output file name. If not specified, it will be set to the input file name + '.' + *compr*. An existing output file will be overwritten.

- *remove*: if True (default), the input file is removed after succesful compression
- *level*: an integer from 1..9: gzip/bzip2 compression level. Higher values result in smaller files, but require longer compression times. The default of 5 gives already a fairly good compression ratio.
- *compr*: 'gz' or 'bz2': the compression algorithm to be used. The default is 'gz' for gzip compression. Setting to 'bz2' will use bzip2 compression.

Returns the name of the compressed file.

`utils.gunzip(filename, unzipped=None, remove=True, compr='gz')`
Uncompress a file in gzip/bzip2 format.

Parameters:

- *filename*: compressed input file name (usually ending in '.gz' or '.bz2')
- *unzipped*: output file name. If not specified and *filename* ends with '.gz' or '.bz2', it will be set to the *filename* with the '.gz' or '.bz2' removed. If an empty string is specified or it is not specified and *filename* does not end in '.gz' or '.bz2', the name of a temporary file is generated. Since you will normally want to read something from the decompressed file, this temporary file is not deleted after closing. It is up to the user to delete it (using the returned file name) when he is ready with it.
- *remove*: if True (default), the input file is removed after succesful decompression. You probably want to set this to False when decompressing to a temporary file.
- *compr*: 'gz' or 'bz2': the compression algorithm used in the input file. If *filename* ends with either '.gz' or '.bz2', it is automatically set from the extension. Else, the default 'gz' is used.

Returns the name of the decompressed file.

`utils.timeEval(s, glob=None)`
Return the time needed for evaluating a string.

s is a string with a valid Python instructions. The string is evaluated using Python's `eval()` and the difference in seconds between the current time before and after the evaluation is printed. The result of the evaluation is returned.

This is a simple method to measure the time spent in some operation. It should not be used for microlevel instructions though, because the overhead of the time calls. Use Python's `timeit` module to measure microlevel execution time.

`utils.countLines(fn)`
Return the number of lines in a text file.

`utils.userName()`
Find the name of the user.

`utils.is_script(appname)`
Checks whether an application name is rather a script name

`utils.is_pyFormex(appname)`
Checks whether an application name is rather a script name

`utils.getDocString(scriptfile)`
Return the docstring from a script file.

This actually returns the first multiline string (delimited by triple double quote characters) from the file. It does relies on the script file being structured properly and indeed including a doctring at the beginning of the file.

`utils.hsorted(l)`
Sort a list of strings in human order.

When human sort a list of strings, they tend to interpret the numerical fields like numbers and sort these parts numerically, instead of the lexicographic sorting by the computer.

Returns the list of strings sorted in human order.

Example: `>>> hsorted(['a1b','a11b','a1.1b','a2b','a1'])` `['a1', 'a1.1b', 'a1b', 'a2b', 'a11b']`

`utils.numsplit(s)`

Split a string in numerical and non-numerical parts.

Returns a series of substrings of `s`. The odd items do not contain any digits. The even items only contain digits. Joined together, the substrings restore the original.

The number of items is always odd: if the string ends or starts with a digit, the first or last item is an empty string.

Example:

```
>>> print(numsplit("aa11.22bb"))
['aa', '11', '.', '22', 'bb']
>>> print(numsplit("11.22bb"))
['', '11', '.', '22', 'bb']
>>> print(numsplit("aa11.22"))
['aa', '11', '.', '22', '']
```

`utils.splitDigits(s, pos=-1)`

Split a string at a sequence of digits.

The input string is split in three parts, where the second part is a contiguous series of digits. The second argument specifies at which numerical substring the splitting is done. By default (`pos=-1`) this is the last one.

Returns a tuple of three strings, any of which can be empty. The second string, if non-empty is a series of digits. The first and last items are the parts of the string before and after that series. Any of the three return values can be an empty string. If the string does not contain any digits, or if the specified splitting position exceeds the number of numerical substrings, the second and third items are empty strings.

Example:

```
>>> splitDigits('abc123')
('abc', '123', '')
>>> splitDigits('123')
('', '123', '')
>>> splitDigits('abc')
('abc', '', '')
>>> splitDigits('abc123def456fghi')
('abc123def', '456', 'fghi')
>>> splitDigits('abc123def456fghi',0)
('abc', '123', 'def456fghi')
>>> splitDigits('123-456')
('123-', '456', '')
>>> splitDigits('123-456',2)
('123-456', '', '')
>>> splitDigits('')
('', '', '')
```

`utils.globFiles(pattern, sort=<function hsorted>)`

Return a (sorted) list of files matching a filename pattern.

A function may be specified to sort/filter the list of file names. The function should take a list of filenames as input. The output of the function is returned. The default sort function will sort the filenames in a human order. This will sort numeric fields in order of increasing numbers instead of alphanumerically.

Examples

```
>>> globFiles('pyformex/o*.py')
['pyformex/olist.py', 'pyformex/options.py']
```

`utils.autoName` (*clas*)

Return the autoname class instance for objects of type *clas*.

This allows for objects of a certain class to be automatically named throughout pyFormex.

Parameters *clas* (*str or class or object*) – The object class name. If a *str*, it is the class name. If a class, the name is found from it. If an object, the name is taken from the object's class. In all cases the name is converted to lower case

Returns *NameSequence instance* – A *NameSequence* that will generate subsequent names corresponding with the specified class.

Examples

```
>>> from pyformex.formex import Formex
>>> F = Formex()
>>> print(next(autoName(Formex)))
formex-0
>>> print(next(autoName(F)))
formex-1
>>> print(next(autoName('Formex')))
formex-2
```

`utils.prefixDict` (*d, prefix=""*)

Prefix all the keys of a dict with the given prefix.

Parameters

- **d** (*dict*) – A dict where all keys are strings.
- **prefix** (*str*) – A string to prepend to all keys in the dict.

Returns *dict* – A dict with the same contents as the input, but where all keys have been prefixed with the given prefix string.

Examples

```
>>> prefixDict({'a':0,'b':1}, 'p_')
{'p_a': 0, 'p_b': 1}
```

`utils.subDict` (*d, prefix="", strip=True*)

Return a dict with the items whose key starts with *prefix*.

Parameters

- **d** (*dict*) – A dict where all the keys are strings.
- **prefix** (*str*) – The string that is to be found at the start of the keys.
- **strip** (*bool*) – If *True* (default), the prefix is stripped from the keys.

Returns *dict* – A dict with all the items from *d* whose key starts with *prefix*. The keys in the returned dict will have the prefix stripped off, unless *strip=False* is specified.

Examples

```
>>> subDict({'p_a':0,'q_a':1,'p_b':2}, 'p_')
{'a': 0, 'b': 2}
>>> subDict({'p_a':0,'q_a':1,'p_b':2}, 'p_', strip=False)
{'p_a': 0, 'p_b': 2}
```

`utils.selectDict(d, keys, remove=False)`

Return a dict with the items whose key is in keys.

Parameters

- **d** (*dict*) – The dict to select items from.
- **keys** (*set of str*) – The keys to select from d. This can be a set or list of key values, or another dict, or any object having the `key in object` interface.
- **remove** (*bool*) – If True, the selected keys are removed from the input dict.

Returns *dict* – A dict with all the items from d whose key is in keys.

See also:

`removeDict()` the complementary operation, returns items not in keys.

Examples

```
>>> d = dict([(c,c*c) for c in range(4)])
>>> selectDict(d, [2,0])
{0: 0, 2: 4}
>>> print(d)
{0: 0, 1: 1, 2: 4, 3: 9}
>>> selectDict(d, [2,0,6], remove=True)
{0: 0, 2: 4}
>>> print(d)
{1: 1, 3: 9}
```

`utils.removeDict(d, keys)`

Return a dict with the specified keys removed.

Parameters

- **d** (*dict*) – The dict to select items from.
- **keys** (*set of str*) – The keys to select from d. This can be a set or list of key values, or another dict, or any object having the `key in object` interface.

Returns *dict* – A dict with all the items from d whose key is not in keys.

See also:

`selectDict()` the complementary operation returning the items in keys

Examples

```
>>> d = dict([(c,c*c) for c in range(6)])
>>> removeDict(d, [4,0])
{1: 1, 2: 4, 3: 9, 5: 25}
```

`utils.refreshDict` (*d*, *src*)

Refresh a dict with values from another dict.

The values in the dict *d* are update with those in *src*. Unlike the `dict.update` method, this will only update existing keys but not add new keys.

`utils.inverseDict` (*d*)

Return the inverse of a dictionary.

Returns a dict with keys and values interchanged.

Example:

```
>>> inverseDict({'a':0, 'b':1})
{0: 'a', 1: 'b'}
```

`utils.selectDictValues` (*d*, *values*)

Return the keys in a dict which have a specified value

- *d*: a dict where all the keys are strings.
- *values*: a list/set of values.

The return value is a list with all the keys from *d* whose value is in *keys*.

Example:

```
>>> d = dict([(c,c*c) for c in range(6)])
>>> selectDictValues(d, range(10))
[0, 1, 2, 3]
```

`utils.dictStr` (*d*, *skipkeys=[]*)

Convert a dict to a string.

This is much like `dict.__str__`, but formats all keys as strings and prints the items with the keys sorted.

This function is can be used as replacement for the `__str__` method od dict-like classes.

A list of strings *skipkeys* can be specified, to suppress the printing of some special key values.

Example:

```
>>> dictStr({'a':0, 'b':1, 'c':2}, ['b'])
"{'a': 0, 'c': 2}"
```

`utils.listAllFonts` ()

List all fonts known to the system.

Returns a sorted list of path names to all the font files found on the system.

This uses `fontconfig` and will produce a warning if `fontconfig` is not installed.

`utils.is_valid_mono_font` (*fontfile*)

Filter valid monotype fonts

Parameters *fontfile* (`Path`) – Path of a font file.

Returns *bool* – True if the provided font file has a `.ttf` suffix, is a fixed width font and the font basename is not listed in the ‘`fonts/ignore`’ configuration variable.

`utils.listMonoFonts` ()

List all monospace font files found on the system.

`utils.defaultMonoFont` ()

Return a default monospace font for the system.

`utils.interrogate` (*item*)

Print useful information about item.

`utils.memory_report` (*keys=None*)

Return info about memory usage

`utils.totalMemSize` (*o, handlers={}, verbose=False*)

Return the approximate total memory footprint of an object.

This function returns the approximate total memory footprint of an object and all of its contents.

Automatically finds the contents of the following builtin containers and their subclasses: tuple, list, deque, dict, set and frozenset. To search other containers, add handlers to iterate over their contents:

```
handlers = {SomeContainerClass: iter, OtherContainerClass:
            Class.get_elements}
            OtherContainer-
```

Adapted from <http://code.activestate.com/recipes/577504/>

6.2.9 varray — Working with variable width tables.

Mesh type geometries use tables of integer data to store the connectivity between different geometric entities. The basic connectivity table in a Mesh with elements of the same type is a table of constant width: the number of nodes connected to each element is constant. However, the inverse table (the elements connected to each node) does not have a constant width.

Tables of constant width can conveniently be stored as a 2D array, allowing fast indexing by row and/or column number. A variable width table can be stored (using arrays) in two ways:

- as a 2D array, with a width equal to the maximal row length. Unused positions in the row are then filled with an invalid value (-1).
- as a 1D array, storing a simple concatenation of the rows. An additional array then stores the position in that array of the first element of each row.

In pyFormex, variable width tables were initially stored as 2D arrays: a remnant of the author's past FORTRAN experience. With a growing professional use of pyFormex involving ever larger models, it became clear that there was a large memory and speed penalty related to the use of 2D arrays with lots of unused entries. This is illustrated in the following table, obtained on the inversion of a connectivity table of 10000 rows and 25 columns. The table shows the memory size of the inverse table, the time needed to compute it, and the time to compute both tables. The latter involves an extra conversion of the stored array to the other data type.

Stored as:	2D (ndarray)	1D (Varray)	1D (Varray)
Rows are sorted:	yes	yes	no
Memory size	450000	250000	250000
Time to create table	128 ms	49 ms	25ms
Time to create both	169 ms	82 ms	57ms

The memory and speed gains of using the Varray are important. The 2D array can even be faster generated by first creating the 1D array, and then converting that to 2D. Not sorting the entries in the Varray provides a further gain. The Varray class defined below therefore does not sort the rows by default, but provides methods to sort them when needed.

Classes defined in module varray

```
class varray.Varray (data=[], ind=None)
```

A variable width 2D integer array

This class provides an efficient way to store tables of nonnegative integers when the rows of the table may have different length.

For large tables this may allow an important memory saving compared to a rectangular array where the non-existent entries are filled by some special value. Data in the Varray are stored as a single 1D array, containing the concatenation of all rows. An index is kept with the start position of each row in the 1D array.

Parameters

- **data** – Data to initialize to a new Varray object. This can either of:
 - another Varray instance: a shallow copy of the Varray is created.
 - a list of lists of integers. Each item in the list contains one row of the table.
 - a 2D ndarray of integer type. The nonnegative numbers on each row constitute the data for that row.
 - a 1D array or list of integers, containing the concatenation of the rows. The second argument *ind* specifies the indices of the first element of each row.
 - a 1D array or list of integers, containing the concatenation of the rows obtained by prepending each row with the row length. The caller should make sure these 1D data are consistent.
- **ind** (1-dim int *array_like*, optional) – This is only used when *data* is a pure concatenation of all rows. It holds the position in *data* of the first element of each row. Its length is equal to the number of rows (*nrows*) or *nrows+1*. It is a non-decreasing series of integer values, starting with 0. If it has *nrows+1* entries, the last value is equal to the total number of elements in *data*. This last value may be omitted, and will then be added automatically. Note that two subsequent elements may be equal, corresponding with an empty row.

Attributes

nrows

The number of rows in the table

Type int

width

The length of the longest row in the table

Type int

size

The total number of entries in the table

Type int

shape

The combined (*nrows*, ‘width’) values.

Type tuple of two ints

Examples

Create a Varray is by default printed in user-friendly format:

```
>>> Va = Varray([[0], [1, 2], [0, 2, 4], [0, 2]])
>>> Va
Varray([[0], [1, 2], [0, 2, 4], [0, 2]])
```

The Varray prints in a user-friendly format:

```
>>> print (Va)
Varray (4,3)
 [0]
 [1 2]
 [0 2 4]
 [0 2]
<BLANKLINE>
```

Other initialization methods resulting in the same Varray:

```
>>> Vb = Varray(Va)
>>> print (str(Vb) == str(Va))
True
>>> Vb = Varray(np.array([[ -1, -1, 0], [-1, 1, 2], [0, 2, 4], [-1, 0, 2]]))
>>> print (str(Vb) == str(Va))
True
>>> Vc = Varray([0,1,2,0,2,4,0,2],at.cumsum([0,1,2,3,2]))
>>> print (str(Vc) == str(Va))
True
>>> Vd = Varray([1,0, 2,1,2, 3,0,2,4, 2,0,2])
>>> print (str(Vd) == str(Va))
True
```

Show info about the Varray

```
>>> print (Va.nrows, Va.width, Va.shape)
4 3 (4, 3)
>>> print (Va.size, Va.lengths)
8 [1 2 3 2]
```

Indexing: The data for any row can be obtained by simple indexing:

```
>>> print (Va[1])
[1 2]
```

This is equivalent with

```
>>> print (Va.row(1))
[1 2]
```

```
>>> print (Va.row(-1))
[0 2]
```

Change elements:

```
>>> Va[1][0] = 3
>>> print (Va[1])
[3 2]
```

Full row can be changed with matching length:

```
>>> Va[1] = [1, 2]
>>> print (Va[1])
[1 2]
```

Negative indices are allowed:

Extracted columns are filled with -1 values where needed

```
>>> print (Va.col(1))
[-1  2  2  2]
```

Select takes multiple rows using indices or bool:

```
>>> print (Va.select([1,3]))
Varray (2,2)
 [1 2]
 [0 2]
<BLANKLINE>
>>> print (Va.select(Va.lengths==2))
Varray (2,2)
 [1 2]
 [0 2]
<BLANKLINE>
```

Iterator: A Varray provides its own iterator:

```
>>> for row in Va:
...     print(row)
[0]
[1 2]
[0 2 4]
[0 2]
```

```
>>> print (Varray())
Varray (0,0)
<BLANKLINE>
```

```
>>> L,R = Va.sameLength()
>>> print (L)
[1 2 3]
>>> print (R)
[array([0]), array([1, 3]), array([2])]
>>> for a in Va.split():
...     print(a)
[[0]]
[[1 2]
 [0 2]]
[[0 2 4]]
```

lengths

Return the length of all rows of the Varray

nrows

Return the number of rows in the Varray

size

Return the total number of elements in the Varray

shape

Return a tuple with the number of rows and maximum row length

length (i)

Return the length of row i

row (i)

Return the data for row i

This returns `self[i]`.

setRow (*i*, *data*)

Replace the data of row *i*

This is equivalent to `self[i] = data`.

col (*i*)

Return the data for column *i*

This always returns a list of length `nrows`. For rows where the column index *i* is missing, a value -1 is returned.

select (*sel*)

Select some rows from the Varray.

Parameters *sel* (*iterable of ints or bools*) – Specifies the row(s) to be selected.

If type is `int`, the values are the row numbers. If type is `bool`, the length of the iterable should be exactly `self.nrows`; the positions where the value is `True` are the rows to be returned.

Returns *Varray object* – A Varray with only the selected rows.

Examples

```
>>> Va = Varray([[0], [1, 2], [0, 2, 4], [0, 2]])
>>> Va.select((1, 3))
Varray([[1, 2], [0, 2]])
>>> Va.select((False, True, False, True))
Varray([[1, 2], [0, 2]])
```

index (*sel*)

Convert a selector to an index.

Parameters *sel* (*iterable of ints or bools*) – Specifies the elements of the Varray to be selected. If type is `int`, the values are the index numbers in the flat array. If type is `bool`, the length of the iterable should be exactly `self.size`; the positions where the value is `True` will be returned.

Returns *int array* – The selected element numbers.

Examples

```
>>> Va = Varray([[0], [1, 2], [0, 2, 4], [0, 2]])
>>> Va.index((1, 3, 5, 7))
array([1, 3, 5, 7])
>>> Va.index((False, True, False, True, False, True, False, True))
array([1, 3, 5, 7])
```

rowindex (*sel*)

Return the rowindex for the elements flagged by selector *sel*.

sel is either a list of element numbers or a `bool` array with length `self.size`

colindex (*sel*)

Return the column index for the elements flagged by selector *sel*.

sel is either a list of element numbers or a `bool` array with length `self.size`

where (*sel*)

Return row and column index of the selected elements

sel is either a list of element numbers or a bool array with length *self.size*

Returns a 2D array where the first column is the row index and the second column the corresponding column index of an element selected by *sel*

index1d (*i,j*)

Return the sequential index for the element with 2D index *i,j*

sorted ()

Returns a sorted Varray.

Returns a Varray with the same entries but where each row is sorted.

This returns a copy of the data, and leaves the original unchanged.

See also *sort* () for sorting the rows inplace.

removeFlat (*ind*)

Remove the nelement with flat index *i*

Parameters *ind* (*int* or *int :term:*) – Index in the flat data of the element(s) to remove.

Returns *Varray* – A Varray with the element(s) *ind* removed.

Examples

```
>>> Va = Varray([[0], [1, 2], [0, 2, 4], [0, 2]])
>>> Va.removeFlat(3)
Varray([[0], [1, 2], [2, 4], [0, 2]])
>>> Va.removeFlat([0, 2, 7])
Varray([[], [1], [0, 2, 4], [0]])
```

sort ()

Sort the Varray inplace.

Sorting a Varray sorts the elements in each row. The sorting is done inplace.

See also *sorted* () for sorting the rows without changing the original.

toArray ()

Convert the Varray to a 2D array.

Returns a 2D array with shape (*self.nrows*,*self.width*), containing the row data of the Varray. Rows which are shorter than width are padded at the start with values -1.

sameLength ()

Groups the rows according to their length.

Returns a tuple of two lists (*lengths*,*rows*):

- *lengths*: the sorted unique row lengths,
- *rows*: the indices of the rows having the corresponding length.

split ()

Split the Varray into 2D arrays.

Returns a list of 2D arrays with the same number of columns and the indices in the original Varray.

toList ()

Convert the Varray to a nested list.

Returns a list of lists of integers.

inverse (expand=False)

Return the inverse of a Varray.

The inverse of a Varray is again a Varray. Values k on a row i will become values i on row k . The number of data in both Varrays is thus the same.

The inverse of the inverse is equal to the original. Two Varrays are equal if they have the same number of rows and all rows contain the same numbers, independent of their order.

Example:

```
>>> a = Varray([[0,1], [2,0], [1,2], [4]])
>>> b = a.inverse()
>>> c = b.inverse()
>>> print(a,b,c)
Varray (4,2)
  [0 1]
  [2 0]
  [1 2]
  [4]
Varray (5,2)
  [0 1]
  [0 2]
  [1 2]
  []
  [3]
Varray (4,2)
  [0 1]
  [0 2]
  [1 2]
  [4]
<BLANKLINE>
```

Functions defined in module varray

`varray.inverseIndex (ind, sort=False, expand=False)`

Create the inverse of a 2D index array.

Parameters:

- *ind*: a Varray or a 2D index array. A 2D index array is a 2D integer array where only nonnegative values are significant and negative values are silently ignored. While in most cases all values in a row are unique, this is not a requirement. Degenerate elements may have the same node number appearing multiple times in the same row.
- *sort*: bool. If True, rows are sorted.
- *expand*: bool. If True, an `numpy.ndarray` is returned.

Returns the inverse index, as a Varray (default) or as an ndarray (if `expand` is True). If `sort` is True, rows are sorted.

Example

```
>>> a = inverseIndex([[0,1],[0,2],[1,2],[0,3]])
>>> print(a)
Varray (4,3)
  [0 1 3]
  [0 2]
  [1 2]
  [3]
<BLANKLINE>
```

6.2.10 adjacency — A class for storing and handling adjacency tables.

This module defines a specialized array class for representing adjacency of items of a single type. This is e.g. used in mesh models, to store the adjacent elements.

class `adjacency.Adjacency` (*data=[]*, *dtyp=None*, *copy=False*, *normalize=True*)

A class for storing and handling adjacency tables.

An adjacency table defines a neighbouring relation between elements of a single collection. The nature of the relation is not important, but should be a binary relation: two elements are either related or they are not.

Typical applications in pyFormex are the adjacency tables for storing elements connected by a node, or by an edge, or by a node but not by an edge, etcetera.

Conceptually the adjacency table corresponds with a graph. In graph theory however the data are usually stored as a set of tuples (a,b) indicating a connection between the elements a and b . In pyFormex elements are numbered consecutively from 0 to `nelems-1`, where `nelems` is the number of elements. If the user wants another numbering, he can always keep an array with the actual numbers himself. Connections between elements are stored in an efficient two-dimensional array, holding a row for each element. This row contains the numbers of the connected elements. Because the number of connections can be different for each element, the rows are padded with an invalid elements number (-1).

A normalized Adjacency is one where all rows do not contain duplicate nonnegative entries and are sorted in ascending order and where no column contains only -1 values. Also, since the adjacency is defined within a single collection, no row should contain a value higher than the maximum row index.

Parameters

- **data** (*int :term:*) – Data to initialize the Connectivity. The data should be 2-dim with shape $(nelems, ncon)$, where `nelems` is the number of elements and `ncon` is the maximum number of connections per element.
- **dtyp** (*float datatype, optional*) – Can be provided to force a specific int data type. If not, the datatype of `data` is used.
- **copy** (*bool, optional*) – If True, the data are copied. The default setting will try to use the original data if possible, e.g. if `data` is a correctly shaped and typed `numpy.ndarray`.
- **normalize** (*bool, optional*) – If True (default) the Adjacency will be normalized at creation time.
- **allow_self** (*bool, optional*) – If True, connections of elements with itself are allowed. The default (False) will remove self-connections when the table is normalized.

Warning: The `allow_self` parameter is currently inactive.

Examples

```
>>> A = Adjacency([[1, 2, -1],
...               [3, 2, 0],
...               [1, -1, 3],
...               [1, 2, -1],
...               [-1, -1, -1]])
>>> print(A)
[[-1  1  2]
 [ 0  2  3]
 [-1  1  3]
 [-1  1  2]
 [-1 -1 -1]]
>>> A.nelems()
5
>>> A.maxcon()
3
>>> Adjacency([[]])
Adjacency([], shape=(1, 0))
```

`nelems()`

Return the number of elements in the Adjacency table.

`maxcon()`

Return the maximum number of connections for any element.

This returns the row width of the Adjacency.

`sortRows()`

Sort an adjacency table.

This sorts the entries in each row of the adjacency table in ascending order and removes all columns containing only -1 values.

Returns *Adjacency* – An Adjacency with the same non-negative data but each row sorted in ascending order, and no column with only negative values. The number of rows is the same as the input, the number of columns may be lower.

Examples

```
>>> a = Adjacency([[ 0,  2,  1, -1],
...               [-1,  3,  1, -1],
...               [ 3, -1,  0,  1],
...               [-1, -1, -1, -1]])
>>> a.sortRows()
Adjacency([[[-1,  1,  2],
            [-1, -1,  3],
            [ 0,  1,  3],
            [-1, -1, -1]])
>>> a = Adjacency([[ 0,  2,  1, -1],
...               [-1,  3,  1, -1],
...               [ 3, -1,  0,  1],
...               [-1, -1, -1, -1]], normalize=False)
```

(continues on next page)

(continued from previous page)

```
>>> a.sortRows()
Adjacency([[ 0,  1,  2],
           [-1,  1,  3],
           [ 0,  1,  3],
           [-1, -1, -1]])
```

normalize()

Normalize an adjacency table.

A normalized adjacency table is one where each row:

- does not contain the row index itself,
- does not contain duplicates,
- is sorted in ascending order,

and that has no columns with all -1 values.

By default, an Adjacency gets normalized when it is constructed. Performing operations on an Adjacency may however leave it in a non-normalized state. Calling this method will normalize it again. This can obviously also be obtained by creating a new Adjacency with self as data.

Returns *Adjacency* – An Adjacency object with shape (self.shape[0],maxc), with maxc <= adj.shape[1]. A row *i* of the Adjacency contains the unique non-negative numbers except the value *i* of the same row *i* in the original, and is possibly padded with -1 values.

Examples

```
>>> a = Adjacency([[ 0,  0,  0,  1,  2,  5],
...               [-1,  0,  1, -1,  1,  3],
...               [-1, -1,  0, -1, -1,  2],
...               [-1, -1,  1, -1, -1,  3],
...               [-1, -1, -1, -1, -1, -1],
...               [-1, -1,  0, -1, -1,  5]],normalize=False)
>>> a.normalize()
Adjacency([[ 1,  2,  5],
           [-1,  0,  3],
           [-1, -1,  0],
           [-1, -1,  1],
           [-1, -1, -1],
           [-1, -1,  0]])
```

pairs()

Return all pairs of adjacent element.

Returns *int array* – An int array with two columns, where each row contains a pair of adjacent elements. The element number in the first column is always the smaller of the two element numbers.

Examples

```
>>> Adjacency([[ -1,  1], [ 0,  2], [-1,  0]]).pairs()
array([[ 0,  1],
       [ 1,  2]])
```

symdiff (*adj*)

Return the symmetric difference of two adjacency tables.

Parameters *adj* (*Adjacency*) – An *Adjacency* with the same number of rows as *self*.

Returns *Adjacency* – An adjacency table of the same length, where each row contains all the (nonnegative) numbers of the corresponding rows of *self* and *adj*, except those that occur in both.

Examples

```
>>> A = Adjacency([[ 1, 2,-1],
...               [ 3, 2, 0],
...               [ 1,-1, 3],
...               [ 1, 2,-1],
...               [-1,-1,-1]])
>>> B = Adjacency([[ 1, 2, 3],
...               [ 3, 4, 1],
...               [ 0,-1, 2],
...               [ 0, 3, 4],
...               [-1, 0,-1]])
>>> A.symdiff(B)
Adjacency([[ -1, -1, -1,  3],
           [-1,  0,  2,  4],
           [-1,  0,  1,  3],
           [ 0,  1,  2,  4],
           [-1, -1, -1,  0]])
```

frontGenerator (*startat=0, frontinc=1, partinc=1*)

Generator function returning the frontal elements.

This is a generator function and is normally not used directly, but via the *frontWalk()* method.

Parameters: see *frontWalk()*.

Returns *int array* – Int array with a value for each element. On the initial call, all values are -1, except for the elements in the initial front, which get a value 0. At each call a new front is created with all the elements that are connected to any of the current front and which have not yet been visited. The new front elements get a value equal to the last front's value plus the *frontinc*. If the front becomes empty and a new starting front is created, the front value is extra incremented with *partinc*.

Examples

```
>>> A = Adjacency([[ 1, 2,-1],
...               [ 3, 2, 0],
...               [ 1,-1, 3],
...               [ 1, 2,-1],
...               [-1,-1,-1]])
>>> for p in A.frontGenerator(): print(p)
[ 0 -1 -1 -1 -1]
[ 0  1  1 -1 -1]
[ 0  1  1  2 -1]
[0 1 1 2 4]
```

frontWalk (*startat=0, frontinc=1, partinc=1, maxval=-1*)

Walks through the elements by their node front.

A frontal walk is executed starting from the given element(s). A number of steps is executed, each step advancing the front over a given number of single pass increments. The step number at which an element is reached is recorded and returned.

Parameters

- **startat** (*int or list of ints*) – Initial element number(s) in the front.
- **frontinc** (*int*) – Increment for the front number on each frontal step.
- **partinc** (*int*) – Increment for the front number when the front gets empty and a new part is started.
- **maxval** (*int*) – Maximum frontal value. If negative (default) the walk will continue until all elements have been reached. If non-negative, walking will stop as soon as the frontal value reaches this maximum.

Returns *int array* – An array of ints specifying for each element in which step the element was reached by the walker.

Examples

```
>>> A = Adjacency([
...     [-1, 1, 2, 3],
...     [-1, 0, 2, 3],
...     [ 0, 1, 4, 5],
...     [-1, -1, 0, 1],
...     [-1, -1, 2, 5],
...     [-1, -1, 2, 4]])
>>> print(A.frontWalk())
[0 1 1 1 2 2]
```

front (*startat=0, add=False*)

Returns the elements of the first node front.

Parameters

- **startat** (*int or list of ints*) – Element number(s) or a list of element numbers. The list of elements to find the next front for.
- **add** (*bool, optional*) – If True, the *startat* elements will be included in the return value. The default (False) will only return the elements in the next front line.

Returns *int array* – A list of the elements that are connected to any of the nodes that are part of the *startat* elements.

Notes

This is equivalent to the first step of a *frontWalk()* with the same *startat* elements, and could thus also be obtained from `where(self.frontWalk(startat,maxval=1) == 1)[0]`.

Here however another implementation is used, which is more efficient for very large models: it avoids the creation of the large array as returned by *frontWalk*.

Examples

```
>>> a = Adjacency([[ 0,  0,  0,  1,  2,  5],
...               [-1,  0,  1, -1,  1,  3],
...               [-1, -1,  0, -1, -1,  2],
...               [-1, -1,  1, -1, -1,  3],
...               [-1, -1, -1, -1, -1, -1],
...               [-1, -1,  0, -1, -1,  5]])
>>> print(a.front())
[1 2 5]
>>> print(a.front([0,1]))
[2 3 5]
>>> print(a.front([0,1],add=True))
[0 1 2 3 5]
```

6.2.11 `simple` — Predefined geometries with a simple shape.

This module contains some functions, data and classes for generating Formex structures representing simple geometric shapes. You need to import this module in your scripts to have access to its contents.

Functions defined in module `simple`

`simple.shape` (*name*)

Return a Formex with one of the predefined named shapes.

This is a convenience function returning a plex-2 Formex constructed from one of the patterns defined in the `simple.Pattern` dictionary. Currently, the following pattern names are defined: 'line', 'angle', 'square', 'plus', 'cross', 'diamond', 'rtriangle', 'cube', 'star', 'star3d'. See the Pattern example.

`simple.randomPoints` (*n*, *bbox*=[[0.0, 0.0, 0.0], [1.0, 1.0, 1.0]])

Create *n* random points in a specified *bbox*.

`simple.regularGrid` (*x0*, *x1*, *nx*, *swapaxes*=None)

Create a regular grid of points between two points *x0* and *x1*.

Parameters:

- *x0*: n-dimensional float (usually 1D, 2D or 3D).
- *x1*: n-dimensional float with same dimension as *x0*.
- *nx*: n-dimensional int with same dimension as *x0* and *x1*. The space between *x0* and *x1* is subdivided in *nx[i]* equal parts along the axis *i*.
- *swapaxes*: bool. If False(default), the points are number first in the direction of the 0 axis, then the next axis,... If True, numbering starts in the direction of the highest axis. This is the legacy behavior.

Returns a rectangular grid of n-dimensional coordinates in an array with shape (*nx*[0]+1, *nx*[1]+1, ..., *ndim*).

Example:

```
>>> regularGrid(0.,1.,4)
array([[ 0. ],
       [ 0.25],
       [ 0.5 ],
       [ 0.75],
       [ 1.  ]])
```

(continues on next page)

(continued from previous page)

```

>>> regularGrid((0.,0.), (1.,1.), (3,2))
array([[ 0. , 0. ],
       [ 0.33, 0. ],
       [ 0.67, 0. ]],
      <BLANKLINE>
       [[ 1. , 0. ],
       [ 0. , 0.5 ],
       [ 0.33, 0.5 ]],
      <BLANKLINE>
       [[ 0.67, 0.5 ],
       [ 1. , 0.5 ],
       [ 0. , 1. ]],
      <BLANKLINE>
       [[ 0.33, 1. ],
       [ 0.67, 1. ],
       [ 1. , 1. ]])

```

`simple.point` ($x=0.0, y=0.0, z=0.0$)

Return a Formex which is a point, by default at the origin.

Each of the coordinates can be specified and is zero by default.

`simple.line` ($p1=[0.0, 0.0, 0.0], p2=[1.0, 0.0, 0.0], n=1$)

Return a Formex which is a line between two specified points.

$p1$: first point, $p2$: second point The line is split up in n segments.

`simple.rect` ($p1=[0.0, 0.0, 0.0], p2=[1.0, 0.0, 0.0], nx=1, ny=1$)

Return a Formex which is the circumference of a rectangle.

$p1$ and $p2$ are two opposite corner points of the rectangle. The edges of the rectangle are in planes parallel to the z -axis. There will always be two opposite edges that are parallel with the x -axis. The other two will only be parallel with the y -axis if both points have the same z -value, but in any case they will be parallel with the y - z plane.

The edges parallel with the x -axis are divide in nx parts, the other ones in ny parts.

`simple.rectangle` ($nx=1, ny=1, b=None, h=None, bias=0.0, diag=None$)

Return a Formex representing a rectangular surface.

The rectangle has a size(b,h) divided into (nx,ny) cells.

The default b/h values are equal to nx/ny , resulting in a modular grid. The rectangle lies in the (x,y) plane, with one corner at $[0,0,0]$. By default, the elements are quads. By setting $diag='u','d'$ or $'x'$, diagonals are added in l , resp. and both directions, to form triangles.

`simple.Cube` ()

Create the surface of a cube

Returns a TriSurface representing the surface of a unit cube. Each face of the cube is represented by two triangles.

`simple.circle` ($a1=2.0, a2=0.0, a3=360.0, r=None, n=None, c=None, eltype='line2'$)

A polygonal approximation of a circle or arc.

All points generated by this function lie on a circle with unit radius at the origin in the x - y -plane.

- $a1$: the angle enclosed between the start and end points of each line segment (dash angle).
- $a2$: the angle enclosed between the start points of two subsequent line segments (module angle). If $a2==0.0$, $a2$ will be taken equal to $a1$.

- a_3 : the total angle enclosed between the first point of the first segment and the end point of the last segment (arc angle).

All angles are given in degrees and are measured in the direction from x- to y-axis. The first point of the first segment is always on the x-axis.

The default values produce a full circle (approximately). If $a_3 < 360$, the result is an arc. Large values of a_1 and a_2 result in polygons. Thus *circle(120.)* is an equilateral triangle and *circle(60.)* is regular hexagon.

Remark that the default $a_2 == a_1$ produces a continuous line, while $a_2 > a_1$ results in a dashed line.

Three optional arguments can be added to scale and position the circle in 3D space:

- r : the radius of the circle
- n : the normal on the plane of the circle
- c : the center of the circle

`simple.polygon(n)`

A regular polygon with n sides.

Creates the circumference of a regular polygon with n sides, inscribed in a circle with radius 1 and center at the origin. The first point lies on the axis 0. All points are in the (0,1) plane. The return value is a plex-2 Formex. This function is equivalent to *circle(360./n)*.

`simple.polygonSector(n)`

Create one sector of a regular polygon with n sides

`simple.triangle()`

An equilateral triangle with base [0,1] on axis 0.

Returns an equilateral triangle with side length 1. The first point is the origin, the second points is on the axis 0. The return value is a plex-3 Formex.

`simple.quadraticCurve(x=None, n=8)`

Create a collection of curves.

x is a (3,3) shaped array of coordinates, specifying 3 points.

Return an array with $2*n+1$ points lying on the quadratic curve through the points x . Each of the intervals $[x_0, x_1]$ and $[x_1, x_2]$ will be divided in n segments.

`simple.sphere(ndiv=6, base='icosa', equiv='max')`

Create a triangulated approximation of a spherical surface.

A (possibly high quality) approximation of a spherical surface is constructed as follows. First a simple base triangulated surface is created. Its triangular facets are subdivided by dividing all edges in $ndiv$ parts. The resulting mesh is then projected on a sphere with unit radius. The higher $ndiv$ is taken, the better the approximation. For $ndiv=1$, the base surface is returned.

Parameters:

- $ndiv$: number of divisions along the edges of the base surface.
- $base$: the type of base surface. One of the following:
 - ‘icosa’: icosahedron (20 faces): this offers the highest quality with triangles of almost same size and shape.
 - ‘octa’: octahedron (8 faces): this model will have the same mesh on each of the quadrants. The coordinate planes do not cut any triangle. This model is this fit to be subdivided along coordinate planes.

Returns a TriSurface, representing a triangulated approximation of a spherical surface with radius 1 and center at the origin.

`simple.sphere3` (*nx, ny, r=1, bot=-90.0, top=90.0*)

Return a sphere consisting of surface triangles

A sphere with radius *r* is modeled by the triangles formed by a regular grid of *nx* longitude circles, *ny* latitude circles and their diagonals.

The two sets of triangles can be distinguished by their property number: 1: horizontal at the bottom, 2: horizontal at the top.

The sphere caps can be cut off by specifying top and bottom latitude angles (measured in degrees from 0 at north pole to 180 at south pole).

`simple.sphere2` (*nx, ny, r=1, bot=-90, top=90*)

Return a sphere consisting of line elements.

A sphere with radius *r* is modeled by a regular grid of *nx* longitude circles, *ny* latitude circles and their diagonals.

The 3 sets of lines can be distinguished by their property number: 1: diagonals, 2: meridionals, 3: horizontals.

The sphere caps can be cut off by specifying top and bottom latitude angles (measured in degrees from 0 at north pole to 180 at south pole).

`simple.connectCurves` (*curve1, curve2, n*)

Connect two curves to form a surface.

curve1, *curve2* are plex-2 Formices with the same number of elements. The two curves are connected by a surface of quadrilaterals, with *n* elements in the direction between the curves.

See also:

`Mesh.connect` ()

`simple.sector` (*r, t, nr, nt, h=0.0, diag=None*)

Constructs a Formex which is a sector of a circle/cone.

A sector with radius *r* and angle *t* is modeled by dividing the radius in *nr* parts and the angle in *nt* parts and then creating straight line segments. If a nonzero value of *h* is given, a conical surface results with its top at the origin and the base circle of the cone at *z=h*. The default is for all points to be in the (*x,y*) plane.

By default, a plex-4 Formex results. The central quads will collapse into triangles. If *diag='up'* or *diag='down'*, all quads are divided by an up directed diagonal and a plex-3 Formex results.

`simple.cylinder` (*D, L, nt, nl, D1=None, angle=360.0, bias=0.0, diag=None*)

Create a cylindrical, conical or truncated conical surface.

Returns a Formex representing (an approximation of) a cylindrical or (possibly truncated) conical surface with its axis along the *z*-axis. The resulting surface is actually a prism or pyramid, and only becomes a good approximation of a cylinder or cone for high values of *nt*.

Parameters:

- *D*: base diameter (at *z=0*) of the cylinder/cone,
- *L*: length (along *z*-axis) of the cylinder/cone,
- *nt*: number of elements along the circumference,
- *nl*: number of elements along the length,
- *D1*: diameter at the top (*z=L*) of the cylinder/cone: if unspecified, it is taken equal to *D* and a cylinder results. Setting either *D1* or *D* to zero results in a cone, other values will create a truncated cone.

- *diag*: by default, the elements are quads. Setting *diag* to ‘u’ or ‘d’ will put in an ‘up’ or ‘down’ diagonal to create triangles.

`simple.boxes(x)`

Create a set of cuboid boxes.

x: Coords with shape (nelems,2,3), usually with $x[:,0,:] < x[:,1,:]$

Returns a Formex with shape (nelems,8,3) and of type ‘hex8’, where each element is the cuboid box which has $x[:,0,:]$ as its minimum coordinates and $x[:,1,:]$ as the maximum ones. Note that the elements may be degenerate or reverted if the minimum coordinates are not smaller than the maximum ones.

This function can be used to visualize the `bboxes()` of a geometry.

`simple.boxes2d(x)`

Create a set of rectangular boxes.

Parameters:

- *x*: Coords with shape (nelems,2,3), usually with $x[:,0,:] < x[:,1,:]$ and $x[:,:,2] == 0$.

Returns a Formex with shape (nelems,4,3) and of type ‘quad4’, where each element is the rectangular box which has $x[:,0,:]$ as its minimum coordinates and $x[:,1,:]$ as the maximum ones. Note that the elements may be degenerate or reverted if the minimum coordinates are not smaller than the maximum ones.

This function is a 2D version of `bboxes()`.

`simple.cuboid(xmin=[0.0, 0.0, 0.0], xmax=[1.0, 1.0, 1.0], cs=None)`

Create a rectangular prism.

Create a rectangular prism from two opposite corners. The vertices are specified in the global or a given coordinate system. The faces are parallel to the coordinate planes.

Parameters:

- *xmin*: float(3): minimum coordinates
- *xmax*: float(3): maximum coordinates
- *cs*: CoordSys: if specified, the cuboid is constructed in this coordinate system, and then transformed back to global axes.

Returns a single element Formex with eltype ‘hex8’.

`simple.cuboid2d(xmin=[0.0, 0.0, 0.0], xmax=[1.0, 1.0, 0.0])`

Create a rectangle.

Creates a rectangle with sides parallel to the global y-axis and global xz-plane, and having the points *xmin* and *xmax* as opposite corner points.

Returns a single element Formex with eltype ‘quad4’.

`simple.boundingBox(obj, cs=None)`

Returns a cuboid that is the bounding box of some geometry

The boundingBox is computed in the specified coordinate system. The default is the global axes.

Returns a single hexahedral Formex object.

6.2.12 project — project.py

Functions for managing a project in pyFormex.

Classes defined in module project

class `project.Unpickler` (*f*, *try_resolve=True*)
Customized Unpickler class

find_class (*module*, *name*)
Return an object from a specified module.

If necessary, the module will be imported. Subclasses may override this method (e.g. to restrict unpickling of arbitrary classes and functions).

This method is called whenever a class or a function object is needed. Both arguments passed are str objects.

class `project.Project` (*filename=None*, *access='wr'*, *convert=True*, *signature='pyFormex 1.0.7 (release-1.0.6-141-gc3e5737d) DEVELOPMENT VERSION'*, *compression=5*, *binary=True*, *data={}*, *protocol=3*, ***kargs*)

Project: a persistent storage of pyFormex data.

A pyFormex Project is a regular Python dict that can contain named data of any kind, and can be saved to a file to create persistence over different pyFormex sessions.

The `Project` class is used by pyFormex for the `pyformex.PF` global variable that collects variables exported from pyFormex scripts. While projects are mostly handled through the pyFormex GUI, notably the *File* menu, the user may also create and handle his own Project objects from a script.

Because of the way pyFormex Projects are written to file, there may be problems when trying to read a project file that was created with another pyFormex version. Problems may occur if the project contains data of a class whose implementation has changed, or whose definition has been relocated. Our policy is to provide backwards compatibility: newer versions of pyFormex will normally read the older project formats. Saving is always done in the newest format, and these can generally not be read back by older program versions (unless you are prepared to do some hacking).

Warning: Compatibility issues.

Occasionally you may run into problems when reading back an old project file, especially when it was created by an unreleased (development) version of pyFormex. Because pyFormex is evolving fast, we can not test the full compatibility with every revision. You can file a support request on the [pyFormex support tracker](#), and we will try to add the required conversion code to pyFormex.

The project files are mainly intended as a means to easily save lots of data of any kind and to restore them in the same session or a later session, to pass them to another user (with the same or later pyFormex version), to store them over a medium time period. Occasionally opening and saving back your project files with newer pyFormex versions may help to avoid read-back problems over longer time.

For a problemless long time storage of Geometry type objects you may consider to write them to a pyFormex Geometry file (.pgf) instead, since this uses a stable ascii based format. It can however (currently) only store objects of class Geometry or one of its subclasses.

Parameters:

- *filename*: the name of the file where the Project data will be saved. If the file exists (and *access* is not *w*), it should be a previously saved Project and an attempt will be made to load the data from this file into the Project. If this fails, an error is raised.

If the file exists and *access* is *w*, it will be overwritten, destroying any previous contents.

If no filename is specified, a temporary file will be created when the Project is saved for the first time. The file will not be automatically deleted. The generated name can be retrieved from the filename attribute.

- *access*: One of 'wr' (default), 'rw', 'w' or 'r'. If the string contains an 'r' the data from an existing file will be read into the dict. If the string starts with an 'r', the file should exist. If the string contains a 'w', the data can be written back to the file. The 'r' access mode is thus a read-only mode.

access	File must exist	File is read	File can be written
r	yes	yes	no
rw	yes	yes	yes
wr	no	if it exists	yes
w	no	no	yes

- *convert*: if True (default), and the file is opened for reading, an attempt is made to open old projects in a compatibility mode, doing the necessary conversions to new data formats. If convert is set False, only the latest format can be read and older formats will generate an error.
- *signature*: A text that will be written in the header record of the file. This can e.g. be used to record format version info.
- *compression*: An integer from 0 to 9: compression level. For large data sets, compression leads to much smaller files. 0 is no compression, 9 is maximal compression. The default is 4.
- *binary*: if False and no compression is used, storage is done in an ASCII format, allowing to edit the file. Otherwise, storage uses a binary format. Using binary=False is deprecated.
- *data*: a dict-like object to initialize the Project contents. These data may override values read from the file.

Example

```
>>> d = dict(a=1,b=2,c=3,d=[1,2,3],e={'f':4,'g':5})
>>> P = Project()
>>> P.update(d)
>>> print(P) # doctest: +ELLIPSIS
Project name: None
  access: wr   mode: b   gzip:5
  signature: pyFormex ...
  contents: ['a', 'b', 'c', 'd', 'e']
<BLANKLINE>
>>> print(utils.dictStr(P)) # doctest: +ELLIPSIS
{'a': 1, 'b': 2, 'c': 3, 'd': [1, 2, 3], 'e': ...}
>>> with utils.TempFile() as tmp:
...     P.save(filename=tmp.path, quiet=True)
...     P.clear()
...     print(utils.dictStr(P))
...     P.load(quiet=True)
...     {}
>>> print(utils.dictStr(P)) # doctest: +ELLIPSIS
{'a': 1, 'b': 2, 'c': 3, 'd': [1, 2, 3], 'e': ...}
```

header_data()

Construct the data to be saved in the header.

save (filename=None, quiet=False)

Save the project to file.

readHeader (f, quiet=False)

Read the header from a project file.

f is the file opened for reading. Tries to read the header from different legacy formats, and if successful, adjusts the project attributes according to the values in the header. Returns the open file if successful.

load (*filename=None, try_resolve=True, quiet=False*)

Load a project from file.

The loaded definitions will update the current project.

convert (*filename=None*)

Convert an old format project file.

The project file is read, and if successful, is immediately saved. By default, this will overwrite the original file. If a filename is specified, the converted data are saved to that file. In both cases, access is set to 'wr', so the saved data can be read back immediately.

uncompress (*verbose=True*)

Uncompress a compressed project file.

The project file is read, and if successful, is written back in uncompressed format. This allows to make conversions of the data inside.

delete ()

Unrecoverably delete the project file.

Functions defined in module project

`project.find_class` (*module, name*)

Find a class whose name or module has changed

6.2.13 geomfile — Handling pyFormex Geometry Files

This module defines a class to work with files in the native pyFormex Geometry File Format.

class `geomfile.GeometryFile` (*filename, mode=None, compr=None, level=5, delete_temp=True, sep=' ', ifmt=" ", ffmt=" ", version=None*)

A class to handle files in the pyFormex Geometry File format.

The pyFormex Geometry File (PGF) format allows the persistent storage of most of the geometrical objects available in pyFormex using a format that is independent of the pyFormex version. It guarantees a possible read back in future versions. The format is simple and public, and hence also allows read back from other software.

See http://pyformex.org/doc/file_format for a full description of the file format(s). Older file formats are supported for reading.

Other than just geometry, the pyFormex Geometry File format can also store some attributes of the objects, like names and colors. Future versions will also allow to store field variables.

The `GeometryFile` class uses the `utils.File` class to access the files, and thus provides for transparent compression and decompression of the files. When making use of the compression, the PGF files will remain small even for complex models.

A PGF file starts with a specific header identifying the format and version. When opening a file for reading, the PGF header is read automatically, and the file is thus positioned ready for reading the objects. When opening a file for writing (not appending!), the header is automatically written, and the file is ready for writing objects.

In append mode however, nothing is currently done with the header. This means that it is possible to append to a file using a format different from that used to create the file initially. This is not a good practice, as it may hinder the proper read back of the data. Therefore, append mode should only be used when you are sure that your current pyFormex uses the same format as the already stored file. As a reminder, warning is written when opening the file in append mode.

The *filename*, *mode*, *compr*, *level* and *delete_temp* arguments are passed to the `utils.File` class. See `utils.File` for more details.

Parameters

- **filename** (*path_like*) – The name of the file to open. If the file name ends with ‘.gz’ or ‘.bz2’, transparent (de)compression will be used, as provided by the `utils.File` class.
- **mode** (*'rb', 'wb' or 'ab'*) – Specifies that the file should be opened in read, write or append mode respectively. If omitted, an existing file will be opened in read mode and a non-existing in write mode. Opening an existing file in ‘wb’ mode will overwrite the file, while opening it in ‘ab’ mode will allow to append to the file.
- **compr** (*'gz' or 'bz2'*) – The compression type to be used: gzip or bzip2. If the file name is ending with ‘.gz’ or ‘.bz2’, this is set automatically from the suffix.
- **level** (*int 1..9*) – Compression level for gzip/bzip2. Higher values result in smaller files, but require longer compression times. The default of 5 gives already a fairly good compression ratio.
- **delete_temp** (*bool*) – If True (default), the temporary files needed to do the (de)compression are deleted when the GeometryFile is closed.
- **sep** (*str*) – Separator string to be used when writing numpy arrays to the file. An empty string will make the arrays being written in binary format. Any other string will force text mode, and the `sep` string is used as a separator between subsequent array elements. See also `numpy.tofile()`.
- **ifmt** (*str*) – Format for integer items when writing in text mode.
- **ffmt** (*str*) – Format for float items when writing in test mode.
- **version** (*str*) – Version of PGF format to use when writing. Currently available are ‘1.9’, ‘2.0’, ‘2.1’. The default is ‘2.1’.

readline ()

Read a line from the file

writeline (*s*)

Write a text line to the file

reopen (*mode='rb'*)

Reopen the file, possibly changing the mode.

The default mode for the reopen is ‘rb’

close ()

Close the file.

writeHeader ()

Write the header of a pyFormex geometry file.

The header identifies the file as a pyFormex geometry file and sets the following global values:

- *version*: the version of the geometry file format
- *sep*: the default separator to be used when not specified in the data block

writeData (*data, sep*)

Write an array of data to a pyFormex geometry file.

If `fmt` is None, the data are written using `numpy.tofile`, with the specified separator. If `sep` is an empty string, the data block is written in binary mode, leading to smaller files. If `fmt` is specified, each

write (*geom, name=None, sep=None*)

Write a collection of Geometry objects to the Geometry File.

Parameters:

- *geom*: an object of one the supported Geometry data types or a list or dict of such objects, or a WebGL objdict. Currently exported geometry objects are `Coords`, `Formex`, `Mesh`, `PolyLine`, `BezierSpline`.

Returns the number of objects written.

writeGeometry (*geom*, *name=None*, *sep=None*)

Write a single Geometry object to the Geometry File.

Writes a single Geometry object to the Geometry File, using the specified name and separator.

Parameters:

- *geom*: a supported Geometry type object. Currently supported Geometry objects are `Coords`, `Formex`, `Mesh`, `PolyLine`, `BezierSpline`. Other types are skipped, and a message is written, but processing continues.
- *name*: string: the name of the object to be stored in the file. If not specified, and the object has an *attrib* dict containing a name, that value is used. Else an object name is generated from the file name. On readback, the object names are used as keys to store the objects in a dict.
- *sep*: string: defines the separator to be used for writing this object. If not specified, the value given in the constructor will be used. This argument allows to override it on a per object base.

Returns 1 if the object has been written, 0 otherwise.

writeAttrib (*attrib*)

Write the Attributes block of the Geometry

attrib: the Attributes dict of a Geometry object

Beware: this is work in progress. Not all Attributes can currently be stored in the PGF format.

writeMesh (*F*, *name=None*, *sep=None*, *objtype='Mesh'*)

Write a Mesh to a pyFormex geometry file.

This writes a header line with these attributes and arguments: *objtype*, *ncoords*, *nelems*, *nplex*, *props*(True/False), *eltype*, *normals*(True/False), *color*, *sep*, *name*. This is followed by the array data for: *coords*, *elems*, *prop*, *normals*, *color*

The *objtype* can/should be overridden for subclasses.

writeFormex (*F*, *name=None*, *sep=None*)

Write a Formex to the geometry file.

This is equivalent to `writeMesh(F,name,sep,objtype='Formex')`

writeTriSurface (*F*, *name=None*, *sep=None*)

Write a TriSurface to a pyFormex geometry file.

This is equivalent to `writeMesh(F,name,sep,objtype='TriSurface')`

writeCurve (*F*, *name=None*, *sep=None*, *objtype=None*, *extra=None*)

Write a Curve to a pyFormex geometry file.

This function writes any curve type to the geometry file. The *objtype* is automatically detected but can be overridden.

The following attributes and arguments are written in the header: *ncoords*, *closed*, *name*, *sep*. The following attributes are written as arrays: *coords*

writePolyLine (*F*, *name=None*, *sep=None*)

Write a PolyLine to a pyFormex geometry file.

This is equivalent to `writeCurve(F,name,sep,objtype='PolyLine')`

writeBezierSpline (*F*, *name=None*, *sep=None*)

Write a BezierSpline to a pyFormex geometry file.

This is equivalent to `writeCurve(F,name,sep,objtype='BezierSpline')`

writeNurbsCurve (*F*, *name=None*, *sep=None*, *extra=None*)

Write a NurbsCurve to a pyFormex geometry file.

This function writes a NurbsCurve instance to the geometry file.

The following attributes and arguments are written in the header: `ncoords`, `nknots`, `closed`, `name`, `sep`. The following attributes are written as arrays: `coords`, `knots`

writeNurbsSurface (*F*, *name=None*, *sep=None*, *extra=None*)

Write a NurbsSurface to a pyFormex geometry file.

This function writes a NurbsSurface instance to the geometry file.

The following attributes and arguments are written in the header: `ncoords`, `nknotsu`, `nknotsv`, `closedu`, `closedv`, `name`, `sep`. The following attributes are written as arrays: `coords`, `knotsu`, `knotsv`

read (*count=-1*, *warn_version=True*)

Read objects from a pyFormex Geometry File.

This function reads objects from a Geometry File until the file ends, or until *count* objects have been read. The File should have been opened for reading.

A count may be specified to limit the number of objects read.

Returns a dict with the objects read. The keys of the dict are the object names found in the file. If the file does not contain object names, they will be autogenerated from the file name.

Note that PGF files of version 1.0 are no longer supported. The use of formats 1.1 to 1.5 is deprecated, and users are urged to upgrade these files to a newer format. Support for these formats may be removed in future.

decode (*s*)

Decode the announcement line.

Returns a dict with the interpreted values of the line.

readHeader (*s=None*)

Read the header of a pyFormex geometry file.

Without argument, reads a line from the file and interpretes it as a header line. This is normally used to read the first line of the file. A string *s* may be specified to interpret further lines as a header line.

doHeader (*version='1.1'*, *sep=*"", ***kargs*)

Read the header of a pyFormex geometry file.

Sets detected default values

readGeometry (*objtype='Formex'*, *name=None*, *nelems=None*, *ncoords=None*, *nplex=None*, *props=None*, *eltype=None*, *normals=None*, *color=None*, *colormap=None*, *closed=None*, *degree=None*, *nknots=None*, *sep=None*, ***kargs*)

Read a geometry record of a pyFormex geometry file.

If an object was succesfully read, it is set in `self.geometry`

readField (*field=None*, *fldtype=None*, *shape=None*, *sep=None*, ***kargs*)

Read a Field defined on the last read geometry.

readAttrib (*attrib=None*, ***kargs*)

Read an Attributes dict defined on the last read geometry.

readFormex (*nelems, nplex, props, eltype, sep*)

Read a Formex from a pyFormex geometry file.

The coordinate array for $nelems * nplex$ points is read from the file. If present, the property numbers for $nelems$ elements are read. From the coords and props a Formex is created and returned.

readMesh (*ncoords, nelems, nplex, props, eltype, normals, sep, objtype='Mesh'*)

Read a Mesh from a pyFormex geometry file.

The following arrays are read from the file: - a coordinate array with $ncoords$ points, - a connectivity array with $nelems$ elements of plexitude $nplex$, - if present, a property number array for $nelems$ elements.

Returns the Mesh constructed from these data, or a subclass if an *objtype* is specified.

readPolyLine (*ncoords, closed, sep*)

Read a Curve from a pyFormex geometry file.

The coordinate array for $ncoords$ points is read from the file and a Curve of type *objtype* is returned.

readBezierSpline (*ncoords, closed, degree, sep*)

Read a BezierSpline from a pyFormex geometry file.

The coordinate array for $ncoords$ points is read from the file and a BezierSpline of the given degree is returned.

readNurbsCurve (*ncoords, nknots, closed, sep*)

Read a NurbsCurve from a pyFormex geometry file.

The coordinate array for $ncoords$ control points and the $nknots$ knot values are read from the file. A NurbsCurve of degree $p = nknots - ncoords - 1$ is returned.

readNurbsSurface (*ncoords, nuknots, nvknots, uclosed, vclosed, sep*)

Read a NurbsSurface from a pyFormex geometry file.

The coordinate array for $ncoords$ control points and the $nuknots$ and $nvknots$ values of $uknots$ and $vknots$ are read from the file. A NurbsSurface of degree $pu = nuknots - ncoords - 1$ and $p_v = nvknots - ncoords - 1$ is returned.

readLegacy (*count=-1*)

Read the objects from a pyFormex Geometry File format ≤ 1.7 .

This function reads all the objects of a Geometry File. The File should have been opened for reading, and the header should have been read previously.

A count may be specified to limit the number of objects read.

Returns a dict with the objects read. The keys of the dict are the object names found in the file. If the file does not contain object names, they will be autogenerated from the file name.

oldReadBezierSpline (*ncoords, nparts, closed, sep*)

Read a BezierSpline from a pyFormex geometry file version 1.3.

The coordinate array for $ncoords$ points and control point array for $(nparts, 2)$ control points are read from the file. A BezierSpline is constructed and returned.

rewrite ()

Convert the geometry file to the latest format.

The conversion is done by reading all objects from the geometry file and writing them back. Parts that could not be successfully read will be skipped.

6.2.14 geomtools — Basic geometrical operations.

This module defines some basic operations on simple geometrical entities such as points, lines, planes, vectors, segments, triangles, circles.

The operations include intersection, projection, distance computing.

Many of the functions in this module are exported as methods on the Coords and Geometry classes and subclasses.

Renamed functions:

intersectionPointsLWL	-> intersectLineWithLine
intersectionTimesLWL	-> intersectLineWithLine
intersectionPointsLWP	-> intersectLineWithPlane
intersectionTimesLWP	-> intersectLineWithPlane

Planned renaming of functions:

areaNormals	
degenerate	
levelVolumes	
smallestDirection	
distance	-> distance
closest	
closestPair	
projectedArea	
polygonNormals	
averageNormals	
triangleInCircle	
triangleCircumCircle	
triangleBoundingCircle	
triangleObtuse	
lineIntersection	-> to be removed (or intersect2DLineWithLine)
displaceLines	
segmentOrientation	
rotationAngle	
anyPerpendicularVector	
perpendicularVector	
projectionVOV	-> projectVectorOnVector
projectionVOP	-> projectVectorOnPlane
pointsAtLines	-> pointOnLine
pointsAtSegments	-> pointOnSegment
intersectionTimesSWP	-> remove or intersectLineWithLineTimes2
intersectionSWP	-> intersectSegmentWithPlaneTimes
intersectionPointsSWP	-> intersectSegmentWithPlane
intersectionTimesLWT	-> intersectLineWithTriangleTimes
intersectionPointsLWT	-> intersectLineWithTriangle
intersectionTimesSWT	-> intersectSegmentWithTriangleTimes
intersectionPointsSWT	-> intersectSegmentWithTriangle
intersectionPointsPWP	-> intersectThreePlanes
intersectionLinesPWP	-> intersectTwoPlanes (or intersectPlaneWithPlane)
intersectionSphereSphere	-> intersectTwoSpheres (or intersectSphereWithSphere)
faceDistance	->
edgeDistance	->
vertexDistance	->
baryCoords	

(continues on next page)

```
insideSimplex
insideTriangle
```

Functions defined in module geomtools

`geomtools.pointsAtLines` (q, m, t)

Return the points of lines (q,m) at parameter values t .

Parameters:

- $q, 'm'$: ($\dots,3$) shaped arrays of points and vectors, defining a single line or a set of lines. The vectors do not need to have unit length.
- t : array of parameter values, broadcast compatible with q and m . Parametric value 0 is at point q , parametric value 1 is at $q+m$.

Returns a Coords array with the points at parameter values t .

`geomtools.pointsAtSegments` (S, t)

Return the points of line segments S at parameter values t .

Parameters:

- S : ($\dots,2,3$) shaped array, defining a single line segment or a set of line segments.
- t : array of parameter values, broadcast compatible with S .

Returns an array with the points at parameter values t .

`geomtools.intersectLineWithLine` ($q1, m1, q2, m2, mode='all', times=False$)

Find the common perpendicular of lines ($q1,m1$) and lines ($q2,m2$)

Return the intersection points of lines ($q1,m1$) and lines ($q2,m2$) with the perpendiculars between them. For intersecting lines, the corresponding points will coincide.

Parameters:

- $q_i, 'm_i'$ ($i=1,2$): ($nq_i,3$) shaped arrays of points and vectors defining nq_i lines.
- $mode$: 'all' or 'pair':
 - if 'all', the intersection of all lines $q1,m1$ with all lines $q2,m2$ is computed; $nq1$ and $nq2$ can be different.
 - if 'pair': $nq1$ and $nq2$ should be equal (or 1) and the intersection of pairs of lines are computed (using broadcasting for length 1 data).
- $times$: bool: if True, returns the parameter values of the intersection points instead of the intersection points themselves.

Returns two Coords arrays with the intersection points on lines ($q1,m1$) and ($q2,m2$) respectively (or, if $times$ is True, two float arrays with the parameter values of these points). The size of the Coords arrays is ($nq1,nq2,3$) for mode 'all' and ($nq1,3$) for mode 'pair'. With $times$ True, the size of the returned parameter arrays is ($nq1,nq2,3$) for mode 'all' and size ($nq1,3$) for mode 'pair'.

Note: taking the intersection of two parallel lines will result in nan values. These are not removed from the result. The user can do it if he so wishes.

Example:

```
>>> q, m = [[0, 0, 0], [0, 0, 1], [0, 0, 3]], [[1, 0, 0], [1, 1, 0], [0, 1, 0]]
>>> p, n = [[2., 0., 0.], [0., 0., 0.]], [[0., 1., 0.], [0., 0., 1.]]
```

```

>>> x1,x2 = intersectLineWithLine(q,m,p,n)
>>> print(x1)
[[[ 2.  0.  0.]
 [ 0.  0.  0.]]
<BLANKLINE>
[[ 2.  2.  1.]
 [ 0.  0.  1.]]
<BLANKLINE>
[[ nan nan nan]
 [ 0.  0.  3.]]]
>>> print(x2)
[[[ 2.  0.  0.]
 [ 0.  0.  0.]]
<BLANKLINE>
[[ 2.  2.  0.]
 [ 0.  0.  1.]]
<BLANKLINE>
[[ nan nan nan]
 [ 0.  0.  3.]]]

```

```

>>> x1,x2 = intersectLineWithLine(q[:2],m[:2],p,n,mode='pair')
>>> print(x1)
[[ 2.  0.  0.]
 [ 0.  0.  1.]]
>>> print(x2)
[[ 2.  0.  0.]
 [ 0.  0.  1.]]

```

```

>>> t1,t2 = intersectLineWithLine(q,m,p,n,times=True)
>>> print(t1)
[[ 2. -0.]
 [ 2. -0.]
 [ nan -0.]]
>>> print(t2)
[[ -0. -0.]
 [ 2.  1.]
 [ nan  3.]]

```

```

>>> t1,t2 = intersectLineWithLine(q[:2],m[:2],p,n,mode='pair',times=True)
>>> print(t1)
[ 2. -0.]
>>> print(t2)
[-0.  1.]

```

`geomtools.intersectLineWithPlane` (*q, m, p, n, mode='all', times=False*)

Find the intersection of lines (q,m) and planes (p,n)

Return the intersection points of lines (q,m) and planes (p,n).

Parameters:

- *q, 'm'*: (nq,3) shaped arrays of points and vectors (*mode=all*) or broadcast compatible arrays (*mode=pair*), defining a single line or a set of lines.
- *p, 'n'*: (np,3) shaped arrays of points and normals (*mode=all*) or broadcast compatible arrays (*mode=pair*), defining a single plane or a set of planes.
- *mode*: *all* to calculate the intersection of each line (q,m) with all planes (p,n) or *pair* for pairwise intersec-

tions.

Returns a (nq,np) shaped (*mode=all*) array of parameter values t, such that the intersection points are given by $q+t*m$.

Notice that the result will contain an INF value for lines that are parallel to the plane.

Example:

```
>>> q,m = [[0,0,0],[0,1,0],[0,0,3]], [[1,0,0],[0,1,0],[0,0,1]]
>>> p,n = [[1.,1.,1.],[1.,1.,1.]], [[1.,1.,0.],[1.,1.,1.]]
```

```
>>> t = intersectLineWithPlane(q,m,p,n,times=True)
>>> print(t)
[[ 2.  3.]
 [ 1.  2.]
 [ inf  0.]]
>>> x = intersectLineWithPlane(q,m,p,n)
>>> print(x)
[[[ 2.  0.  0.]
   [ 3.  0.  0.]]
<BLANKLINE>
 [[ 0.  2.  0.]
   [ 0.  3.  0.]]
<BLANKLINE>
 [[ nan nan inf]
   [ 0.  0.  3.]]]
>>> x = intersectLineWithPlane(q[:2],m[:2],p,n,mode='pair')
>>> print(x)
[[ 2.  0.  0.]
 [ 0.  3.  0.]]
```

`geomtools.intersectionTimesSWP` (*S, p, n, mode='all'*)

Return the intersection of line segments *S* with planes (*p,n*).

This is like `intersectionTimesLWP`, but the lines are defined by two points instead of by a point and a vector.

Parameters:

- *S*: (nS,2,3) shaped array (*mode=all*) or broadcast compatible array (*mode=pair*), defining one or more line segments.
- *p, 'n'*: (np,3) shaped arrays of points and normals (*mode=all*) or broadcast compatible arrays (*mode=pair*), defining a single plane or a set of planes.
- *mode*: *all* to calculate the intersection of each line segment *S* with all planes (*p,n*) or *pair* for pairwise intersections.

Returns a (nS,np) shaped (*mode=all*) array of parameter values t, such that the intersection points are given by $(1-t)*S[\dots,0,:] + t*S[\dots,1,:]$.

`geomtools.intersectionSWP` (*S, p, n, mode='all', return_all=False, atol=0.0*)

Return the intersection points of line segments *S* with planes (*p,n*).

Parameters:

- *S*: (nS,2,3) shaped array, defining a single line segment or a set of line segments.
- *p, 'n'*: (np,3) shaped arrays of points and normals, defining a single plane or a set of planes.
- *mode*: *all* to calculate the intersection of each line segment *S* with all planes (*p,n*) or *pair* for pairwise intersections.

- *return_all*: if True, all intersection points of the lines along the segments are returned. Default is to return only the points that lie on the segments.
- *atol*: float tolerance of the points inside the line segments.

Return values if *return_all==True*:

- *t*: (nS,NP) parametric values of the intersection points along the line segments.
- *x*: the intersection points themselves (nS,nP,3).

Return values if *return_all==False*:

- *t*: (n,) parametric values of the intersection points along the line segments ($n \leq nS * nP$)
- *x*: the intersection points themselves (n,3).
- *wl*: (n,) line indices corresponding with the returned intersections.
- *wp*: (n,) plane indices corresponding with the returned intersections

`geomtools.intersectionPointsSWP (S, p, n, mode='all', return_all=False, atol=0.0)`
 Return the intersection points of line segments S with planes (p,n) within tolerance atol.

This is like `intersectionSWP ()` but does not return the parameter values. It is equivalent to:

```
intersectionSWP (S,p,n,mode,return_all) [1:]
```

`geomtools.intersectionTimesLWT (q, m, F, mode='all')`
 Return the intersection of lines (q,m) with triangles F.

Parameters:

- *q*, 'm': (nq,3) shaped arrays of points and vectors (*mode=all*) or broadcast compatible arrays (*mode=pair*), defining a single line or a set of lines.
- *F*: (nF,3,3) shaped array (*mode=all*) or broadcast compatible array (*mode=pair*), defining a single triangle or a set of triangles.
- *mode*: *all* to calculate the intersection of each line (q,m) with all triangles F or *pair* for pairwise intersections.

Returns a (nq,nF) shaped (*mode=all*) array of parameter values *t*, such that the intersection points are given $q+tm$.

`geomtools.intersectionPointsLWT (q, m, F, mode='all', return_all=False)`
 Return the intersection points of lines (q,m) with triangles F.

Parameters:

- *q*, 'm': (nq,3) shaped arrays of points and vectors, defining a single line or a set of lines.
- *F*: (nF,3,3) shaped array, defining a single triangle or a set of triangles.
- *mode*: *all* to calculate the intersection points of each line (q,m) with all triangles F or *pair* for pairwise intersections.
- *return_all*: if True, all intersection points are returned. Default is to return only the points that lie inside the triangles.

Returns If *return_all==True*, a (nq,nF,3) shaped (*mode=all*) array of intersection points, else, a tuple of intersection points with shape (n,3) and line and plane indices with shape (n), where $n \leq nq * nF$.

`geomtools.intersectionTimesSWT` (*S*, *F*, *mode*='all')

Return the intersection of lines segments *S* with triangles *F*.

Parameters:

- *S*: (nS,2,3) shaped array (*mode*=all) or broadcast compatible array (*mode*=pair), defining a single line segment or a set of line segments.
- *F*: (nF,3,3) shaped array (*mode*=all) or broadcast compatible array (*mode*=pair), defining a single triangle or a set of triangles.
- *mode*: all to calculate the intersection of each line segment *S* with all triangles *F* or *pair* for pairwise intersections.

Returns a (nS,nF) shaped (*mode*=all) array of parameter values *t*, such that the intersection points are given by $(1-t)*S[\dots,0,:] + t*S[\dots,1,:]$.

`geomtools.intersectionPointsSWT` (*S*, *F*, *mode*='all', *return_all*=False)

Return the intersection points of lines segments *S* with triangles *F*.

Parameters:

- *S*: (nS,2,3) shaped array, defining a single line segment or a set of line segments.
- *F*: (nF,3,3) shaped array, defining a single triangle or a set of triangles.
- *mode*: all to calculate the intersection points of each line segment *S* with all triangles *F* or *pair* for pairwise intersections.
- *return_all*: if True, all intersection points are returned. Default is to return only the points that lie on the segments and inside the triangles.

Returns If *return_all*==True, a (nS,nF,3) shaped (*mode*=all) array of intersection points, else, a tuple of intersection points with shape (n,3) and line and plane indices with shape (n), where $n \leq nS*nF$.

`geomtools.intersectionPointsPWP` (*p1*, *n1*, *p2*, *n2*, *p3*, *n3*, *mode*='all')

Return the intersection points of planes (p1,n1), (p2,n2) and (p3,n3).

Parameters:

- *pi*, 'ni' (i=1...3): (npi,3) shaped arrays of points and normals (*mode*=all) or broadcast compatible arrays (*mode*=pair), defining a single plane or a set of planes.
- *mode*: all to calculate the intersection of each plane (p1,n1) with all planes (p2,n2) and (p3,n3) or *pair* for pairwise intersections.

Returns a (np1,np2,np3,3) shaped (*mode*=all) array of intersection points.

`geomtools.intersectionLinesPWP` (*p1*, *n1*, *p2*, *n2*, *mode*='all')

Return the intersection lines of planes (p1,n1) and (p2,n2).

Parameters:

- *pi*, 'ni' (i=1...2): (npi,3) shaped arrays of points and normals (*mode*=all) or broadcast compatible arrays (*mode*=pair), defining a single plane or a set of planes.
- *mode*: all to calculate the intersection of each plane (p1,n1) with all planes (p2,n2) or *pair* for pairwise intersections.

Returns a tuple of (np1,np2,3) shaped (*mode*=all) arrays of intersection points *q* and vectors *m*, such that the intersection lines are given by $q+t*m$.

`geomtools.intersectionSphereSphere` (R, r, d)

Intersection of two spheres (or two circles in the x,y plane).

Computes the intersection of two spheres with radii R , resp. r , having their centres at distance $d \leq R+r$. The intersection is a circle with its center on the segment connecting the two sphere centers at a distance x from the first sphere, and having a radius y . The return value is a tuple x,y .

`geomtools.projectPointOnPlane` ($X, p, n, mode='all'$)

Return the projection of points X on planes (p,n) .

Parameters:

- X : a $(nx,3)$ shaped array of points.
- p, n : $(np,3)$ shaped arrays of points and normals defining np planes.
- $mode$: 'all' or 'pair':
 - if 'all', the projection of all points on all planes is computed; nx and np can be different.
 - if 'pair': nx and np should be equal (or 1) and the projection of pairs of point and plane are computed (using broadcasting for length 1 data).

Returns a float array of size $(nx,np,3)$ for mode 'all', or size $(nx,3)$ for mode 'pair'.

Example:

```
>>> X = Coords([[0., 1., 0.], [3., 0., 0.], [4., 3., 0.]])
>>> p,n = [[2., 0., 0.], [0., 1., 0.]], [[1., 0., 0.], [0., 1., 0.]]
>>> print(projectPointOnPlane(X,p,n))
[[[ 2.  1.  0.]
   [ 0.  1.  0.]]
<BLANKLINE>
 [[ 2.  0.  0.]
   [ 3.  1.  0.]]
<BLANKLINE>
 [[ 2.  3.  0.]
   [ 4.  1.  0.]]]
>>> print(projectPointOnPlane(X[:2],p,n,mode='pair'))
[[ 2.  1.  0.]
 [ 3.  1.  0.]]
```

`geomtools.projectPointOnPlaneTimes` ($X, p, n, mode='all'$)

Return the projection of points X on planes (p,n) .

This is like `projectPointOnPlane()` but instead of returning the projected points, returns the parametric values t along the lines (X,n) , such that the projection points can be computed from $X+t*n$.

Parameters: see `projectPointOnPlane()`.

Returns a float array of size (nx,np) for mode 'all', or size $(nx,)$ for mode 'pair'.

Example:

```
>>> X = Coords([[0., 1., 0.], [3., 0., 0.], [4., 3., 0.]])
>>> p,n = [[2., 0., 0.], [0., 1., 0.]], [[1., 0., 0.], [0., 1., 0.]]
>>> print(projectPointOnPlaneTimes(X,p,n))
[[ 2.  0.]
 [-1.  1.]
 [-2. -2.]]
>>> print(projectPointOnPlaneTimes(X[:2],p,n,mode='pair'))
[ 2.  1.]
```

`geomtools.projectPointOnLine` (*X*, *p*, *n*, *mode*='all')

Return the projection of points *X* on lines (*p*,*n*).

Parameters:

- *X*: a (nx,3) shaped array of points.
- *p*, *n*: (np,3) shaped arrays of points and vectors defining *np* lines.
- *mode*: 'all' or 'pair':
 - if 'all', the projection of all points on all lines is computed; *nx* and *np* can be different.
 - if 'pair': *nx* and *np* should be equal (or 1) and the projection of pairs of point and line are computed (using broadcasting for length 1 data).

Returns a float array of size (nx,np,3) for mode 'all', or size (nx,3) for mode 'pair'.

Example:

```
>>> X = Coords([[0., 1., 0.], [3., 0., 0.], [4., 3., 0.]])
>>> p, n = [[2., 0., 0.], [0., 1., 0.]], [[0., 2., 0.], [1., 0., 0.]]
>>> print(projectPointOnLine(X, p, n))
[[[ 2.  1.  0.]
   [ 0.  1.  0.]]
<BLANKLINE>
 [[ 2.  0.  0.]
   [ 3.  1.  0.]]
<BLANKLINE>
 [[ 2.  3.  0.]
   [ 4.  1.  0.]]]
>>> print(projectPointOnLine(X[:2], p, n, mode='pair'))
[[ 2.  1.  0.]
 [ 3.  1.  0.]
```

`geomtools.projectPointOnLineTimes` (*X*, *p*, *n*, *mode*='all')

Return the projection of points *X* on lines (*p*,*n*).

This is like `projectPointOnLine()` but instead of returning the projected points, returns the parametric values *t* along the lines (*X*,*n*), such that the projection points can be computed from $p+t*n$.

Parameters: see `projectPointOnLine()`.

Returns a float array of size (nx,np) for mode 'all', or size (nx,) for mode 'pair'.

Example:

```
>>> X = Coords([[0., 1., 0.], [3., 0., 0.], [4., 3., 0.]])
>>> p, n = [[2., 0., 0.], [0., 1., 0.]], [[0., 1., 0.], [1., 0., 0.]]
>>> print(projectPointOnLineTimes(X, p, n))
[[ 1.  0.]
 [ 0.  3.]
 [ 3.  4.]]
>>> print(projectPointOnLineTimes(X[:2], p, n, mode='pair'))
[ 1.  3.]
```

`geomtools.distanceFromLine` (*X*, *lines*, *mode*='all')

Return the distance of points *X* from lines (*p*,*n*).

Parameters:

- *X*: a (nx,3) shaped array of points.
- *lines*: one of the following definitions of the line(s):

- a tuple (p,n), where both p and n are (np,3) shaped arrays of respectively points and vectors defining *np* lines;
- an (np,2,3) shaped array containing two points of each line.
- *mode*: ‘all’ or ‘pair’:
 - if ‘all’, the distance of all points to all lines is computed; *nx* and *np* can be different.
 - if ‘pair’: *nx* and *np* should be equal (or 1) and the distance of pairs of point and line are computed (using broadcasting for length 1 data).

Returns a float array of size (nx,np) for mode ‘all’, or size (nx) for mode ‘pair’.

Example:

```
>>> X = Coords([[0.,1.,0.],[3.,0.,0.],[4.,3.,0.]])
>>> L = Line([[2.,0.,0.],[0.,1.,0.]],[[0.,3.,0.],[1.,0.,0.]])
>>> print(distanceFromLine(X,L))
[[ 2. 0.]
 [ 1. 1.]
 [ 2. 2.]]
>>> print(distanceFromLine(X[:2],L,mode='pair'))
[ 2. 1.]
>>> L = Line([[2.,0.,0.],[2.,2.,0.]],[[0.,1.,0.],[1.,1.,0.]])
>>> print(distanceFromLine(X,L))
[[ 2. 0.]
 [ 1. 1.]
 [ 2. 2.]]
>>> print(distanceFromLine(X[:2],L,mode='pair'))
[ 2. 1.]
```

`geomtools.pointNearLine` (*X*, *p*, *n*, *atol*, *nproc*=1)

Find the points from X that are near to lines (p,n).

Finds the points from X that are closer than *atol* to any of the lines (p,n).

Parameters:

- *X*: a (nx,3) shaped array of points.
- *p*, *n*: (np,3) shaped arrays of points and vectors defining *np* lines.
- *atol*: float or (nx,) shaped float array of pointwise tolerances

Returns a list of arrays with sorted point indices for each of the lines.

Example:

```
>>> X = Coords([[0.,1.,0.],[3.,0.,0.],[4.,3.,0.]])
>>> p,n = [[2.,0.,0.],[0.,1.,0.]],[[0.,3.,0.],[1.,0.,0.]]
>>> print(pointNearLine(X,p,n,1.5))
[array([1]), array([0, 1])]
>>> print(pointNearLine(X,p,n,1.5,2))
[array([1]), array([0, 1])]
```

`geomtools.faceDistance` (*X*, *Fp*, *return_points*=False)

Compute the closest perpendicular distance to a set of triangles.

X is a (nX,3) shaped array of points. *Fp* is a (nF,3,3) shaped array of triangles.

Note that some points may not have a normal with footpoint inside any of the facets.

The return value is a tuple OKpid,OKdist,OKpoints where:

- OKpid is an array with the point numbers having a normal distance;
- OKdist is an array with the shortest distances for these points;
- OKpoints is an array with the closest footpoints for these points and is only returned if return_points = True.

`geomtools.edgeDistance` (*X*, *Ep*, *return_points=False*)

Compute the closest perpendicular distance of points *X* to a set of edges.

X is a (nX,3) shaped array of points. *Ep* is a (nE,2,3) shaped array of edge vertices.

Note that some points may not have a normal with footpoint inside any of the edges.

The return value is a tuple OKpid,OKdist,OKpoints where:

- OKpid is an array with the point numbers having a normal distance;
- OKdist is an array with the shortest distances for these points;
- OKpoints is an array with the closest footpoints for these points and is only returned if return_points = True.

`geomtools.vertexDistance` (*X*, *Vp*, *return_points=False*)

Compute the closest distance of points *X* to a set of vertices.

X is a (nX,3) shaped array of points. *Vp* is a (nV,3) shaped array of vertices.

The return value is a tuple OKdist,OKpoints where:

- OKdist is an array with the shortest distances for the points;
- OKpoints is an array with the closest vertices for the points and is only returned if return_points = True.

`geomtools.areaNormals` (*x*)

Compute the area and normal vectors of a collection of triangles.

x is an (ntri,3,3) array with the coordinates of the vertices of ntri triangles.

Returns a tuple (areas, normals) with the areas and the normals of the triangles. The area is always positive. The normal vectors are normalized.

`geomtools.degenerate` (*area*, *normals*)

Return a list of the degenerate faces according to area and normals.

area, *normals* are equal sized arrays with the areas and normals of a list of faces, such as the output of the `areaNormals()` function.

A face is degenerate if its area is less or equal to zero or the normal has a nan (not-a-number) value.

Returns a list of the degenerate element numbers as a sorted array.

`geomtools.hexVolume` (*x*)

Compute the volume of hexahedrons.

Parameters:

- *x*: float array (nelems,8,3)

Returns a float array (nelems) with the approximate volume of the hexahedrons formed by each 8-tuple of vertices. The volume is obtained by dividing the hexahedron in 24 tetrahedrons and using the formulas from <http://www.osti.gov/scitech/servlets/purl/632793>

Example:

```

>>> from pyformex.elements import Hex8
>>> X = Coords(Hex8.vertices).reshape(-1, 8, 3)
>>> print(hexVolume(X))
[ 1.]

```

`geomtools.levelVolumes(x)`

Compute the level volumes of a collection of elements.

`x` is an $(\text{nelems}, \text{nplex}, 3)$ array with the coordinates of the `nplex` vertices of `nelems` elements, with `nplex` equal to 2, 3 or 4.

If `nplex == 2`, returns the lengths of the straight line segments. If `nplex == 3`, returns the areas of the triangle elements. If `nplex == 4`, returns the signed volumes of the tetraeder elements. Positive values result if vertex 3 is at the positive side of the plane defined by the vertices (0,1,2). Negative volumes are reported for tetraeders having reversed vertex order.

For any other value of `nplex`, raises an error. If succesful, returns an $(\text{nelems},)$ shaped float array.

`geomtools.inertialDirections(x)`

Return the directions and dimension of a Coords based of inertia.

- `x`: a Coords-like array

Returns a tuple of the principal direction vectors and the sizes along these directions, ordered from the smallest to the largest direction.

`geomtools.smallestDirection(x, method='inertia', return_size=False)`

Return the direction of the smallest dimension of a Coords

- `x`: a Coords-like array
- `method`: one of 'inertia' or 'random'
- `return_size`: if True and `method` is 'inertia', a tuple of a direction vector and the size along that direction and the cross directions; else, only return the direction vector.

`geomtools.largestDirection(x, return_size=False)`

Return the direction of the largest dimension of a Coords.

- `x`: a Coords-like array
- `return_size`: if True and `method` is 'inertia', a tuple of a direction vector and the size along that direction and the cross directions; else, only return the direction vector.

`geomtools.distance(X, Y)`

Returns the distance of all points of `X` to those of `Y`.

Parameters:

- `X`: $(\text{nX}, 3)$ shaped array of points.
- `Y`: $(\text{nY}, 3)$ shaped array of points.

Returns an (nX, nT) shaped array with the distances between all points of `X` and `Y`.

`geomtools.closest(X, Y=None, return_dist=False)`

Find the point of `Y` closest to each of the points of `X`.

Parameters:

- `X`: $(\text{nX}, 3)$ shaped array of points
- `Y`: $(\text{nY}, 3)$ shaped array of points. If None, `Y` is taken equal to `X`, allowing to search for the closest point in a single set. In the latter case, the point itself is excluded from the search (as otherwise that would obviously be the closest one).

- *return_dist*: bool. If True, also returns the distances of the closest points.

Returns:

- *ind*: (nX,) int array with the index of the closest point in Y to the points of X
- *dist*: (nX,) float array with the distance of the closest point. This is equal to `length(X-Y[ind])`. It is only returned if *return_dist* is True.

`geomtools.closestPair(X, Y)`

Find the closest pair of points from X and Y.

Parameters:

- *X*: (nX,3) shaped array of points
- *Y*: (nY,3) shaped array of points

Returns a tuple (i,j,d) where i,j are the indices in X,Y identifying the closest points, and d is the distance between them.

`geomtools.projectedArea(x, dir)`

Compute projected area inside a polygon.

Parameters:

- *x*: (npoints,3) Coords with the ordered vertices of a (possibly nonplanar) polygonal contour.
- *dir*: either a global axis number (0, 1 or 2) or a direction vector consisting of 3 floats, specifying the projection direction.

Returns a single float value with the area inside the polygon projected in the specified direction.

Note that if the polygon is planar and the specified direction is that of the normal on its plane, the returned area is that of the planar figure inside the polygon. If the polygon is nonplanar however, the area inside the polygon is not defined. The projected area in a specified direction is, since the projected polygon is a planar one.

`geomtools.polygonNormals(x)`

Compute normals in all points of polygons in x.

x is an (nel,nplex,3) coordinate array representing nel (possibly nonplanar) polygons.

The return value is an (nel,nplex,3) array with the unit normals on the two edges ending in each point.

`geomtools.averageNormals(coords, elems, atNodes=False, treshold=None)`

Compute average normals at all points of elems.

coords is a (ncoords,3) array of nodal coordinates. elems is an (nel,nplex) array of element connectivity.

The default return value is an (nel,nplex,3) array with the averaged unit normals in all points of all elements. If *atNodes* == True, a more compact array with the unique averages at the nodes is returned.

`geomtools.triangleInCircle(x)`

Compute the incircles of the triangles x

The incircle of a triangle is the largest circle that can be inscribed in the triangle.

x is a Coords array with shape (ntri,3,3) representing ntri triangles.

Returns a tuple r,C,n with the radii, Center and unit normals of the incircles.

Example:

```
>>> X = Formex(Coords([[1.,0.,0.]])).rosette(3,120.)
>>> print(X)
{[1.0,0.0,0.0], [-0.5,0.866025,0.0], [-0.5,-0.866025,0.0]}
```

(continues on next page)

(continued from previous page)

```

>>> radius, center, normal = triangleInCircle(X.coords.reshape(-1,3,3))
>>> print(radius)
[ 0.5]
>>> print(center)
[[ 0. 0. 0.]]

```

`geomtools.triangleCircumCircle` (*x*, *bounding=False*)

Compute the circumcircles of the triangles *x*

x is a Coords array with shape (ntri,3,3) representing ntri triangles.

Returns a tuple r,C,n with the radii, Center and unit normals of the circles going through the vertices of each triangle.

If *bounding=True*, this returns the triangle bounding circle.

`geomtools.triangleBoundingCircle` (*x*)

Compute the bounding circles of the triangles *x*

The bounding circle is the smallest circle in the plane of the triangle such that all vertices of the triangle are on or inside the circle. If the triangle is acute, this is equivalent to the triangle's circumcircle. If the triangle is obtuse, the longest edge is the diameter of the bounding circle.

x is a Coords array with shape (ntri,3,3) representing ntri triangles.

Returns a tuple r,C,n with the radii, Center and unit normals of the bounding circles.

`geomtools.triangleObtuse` (*x*)

Checks for obtuse triangles

x is a Coords array with shape (ntri,3,3) representing ntri triangles.

Returns an (ntri) array of True/False values indicating whether the triangles are obtuse.

`geomtools.lineIntersection` (*P0, N0, P1, N1*)

Finds the intersection of 2 (sets of) lines.

This relies on the lines being pairwise coplanar.

`geomtools.displaceLines` (*A, N, C, d*)

Move all lines (A,N) over a distance *a* in the direction of point *C*.

A,N are arrays with points and directions defining the lines. *C* is a point. *d* is a scalar or a list of scalars. All line elements of *F* are translated in the plane (line,C) over a distance *d* in the direction of the point *C*. Returns a new set of lines (A,N).

`geomtools.segmentOrientation` (*vertices, vertices2=None, point=None*)

Determine the orientation of a set of line segments.

vertices and *vertices2* are matching sets of points. *point* is a single point. All arguments are Coords objects.

Line segments run between corresponding points of *vertices* and *vertices2*. If *vertices2* is *None*, it is obtained by rolling the *vertices* one position forward, thus corresponding to a closed polygon through the *vertices*). If *point* is *None*, it is taken as the center of *vertices*.

The orientation algorithm checks whether the line segments turn positively around the point.

Returns an array with +1/-1 for positive/negative oriented segments.

`geomtools.rotationAngle` (*A, B, m=None, angle_spec=0.017453292519943295*)

Return rotation angles and vectors for rotations of *A* to *B*.

A and *B* are (n,3) shaped arrays where each line represents a vector. This function computes the rotation from each vector of *A* to the corresponding vector of *B*. If *m* is *None*, the return value is a tuple of an (n,) shaped

array with rotation angles (by default in degrees) and an (n,3) shaped array with unit vectors along the rotation axis. If *m* is a (n,3) shaped array with vectors along the rotation axis, the return value is a (n,) shaped array with rotation angles. The returned angle is then the angle between the planes formed by the axis and the vectors. Specify `angle_spec=RAD` to get the angles in radians.

`geomtools.anyPerpendicularVector (A)`

Return arbitrary vectors perpendicular to vectors of A.

A is a (n,3) shaped array of vectors. The return value is a (n,3) shaped array of perpendicular vectors.

The returned vector is always a vector in the x,y plane. If the original is the z-axis, the result is the x-axis.

`geomtools.perpendicularVector (A, B)`

Return vectors perpendicular on both A and B.

`geomtools.projectionVOV (A, B)`

Return the projection of vector of A on vector of B.

`geomtools.projectionVOP (A, n)`

Return the projection of vector of A on plane of B.

`geomtools.baryCoords (S, P)`

Compute the barycentric coordinates of points P wrt. simplexes S.

S is a (nel,npdex,3) shaped array of n-simplexes (n=npdex-1): - 1-simplex: line segment - 2-simplex: triangle - 3-simplex: tetrahedron P is a (npts,3), (npts,nel,3) or (npts,1,3) shaped array of points.

The return value is a (npdex,npts,nel) shaped array of barycentric coordinates.

`geomtools.insideSimplex (BC, bound=True)`

Check if points are in simplexes.

BC is an array of barycentric coordinates (along the first axis), which sum up to one. If `bound = True`, a point lying on the boundary is considered to be inside the simplex.

`geomtools.insideTriangle (x, P, method='bary')`

Checks whether the points P are inside triangles x.

x is a Coords array with shape (ntri,3,3) representing ntri triangles. P is a Coords array with shape (npts,ntri,3) representing npts points in each of the ntri planes of the triangles. This function checks whether the points of P fall inside the corresponding triangles.

Returns an array with (npts,ntri) bool values.

6.2.15 inertia — Compute inertia related quantities of geometrical models.

Inertia related quantities of a geometrical model comprise: the total mass, the center of mass, the inertia tensor, the principal axes of inertia.

This module defines some classes to store the inertia data:

- *Tensor*: a general second order tensor
- *Inertia*: a specialized second order tensor for inertia data

This module also provides the basic functions to compute the inertia data of collections of simple geometric data: points, lines, triangles, tetrahedrons.

The preferred way to compute inertia data of a geometric model is through the `Geometry.inertia()` methods.

Classes defined in module inertia

`class inertia.Tensor`

A second order symmetric(!) tensor in 3D vector space.

This is a new class under design. Only use for development!

The Tensor class provides conversion between full matrix (3,3) shape and contracted vector (6,) shape. It can e.g. be used to store an inertia tensor or a stress or strain tensor. It provides methods to transform the tensor to other (cartesian) axes.

Parameters:

- *data*: *array_like* (float) of shape (3,3) or (6,)
- *symmetric*: bool. If True (default), the tensor is forced to be symmetric by averaging the off-diagonal elements.
- *cs*: CoordSys. The coordinate system of the tensor.

Properties: a Tensor T has the following properties:

- T.xx, T.xy, T.xz, T.yx, T.yy, T.yz, T.zx, T.zy, T.zz: aliases for the nine components of the tensor
- T.contracted: the (6,) shaped contracted array with independent values of the tensor
- T.tensor: the full tensor as an (3,3) array

Discussion:

- inertia and stres/strain tensors transform in the same way on rotations of axes. But differently on translations! Should we therefore store the purpose of the tensor??
 - Propose to leave it to the user to know what he is doing.
 - Propose to have a separate class Inertia derived from Tensor, which implements computing the inertia tensor and translation.
- should we allow non-symmetrical tensors? Then what with principal?
 - Propose to silently allow non-symm. Result of functions is what it is. Again, suppose the user knows what he is doing.

Example

```
>>> t = Tensor([1,2,3,4,5,6])
>>> print(t)
[[ 1.  6.  5.]
 [ 6.  2.  4.]
 [ 5.  4.  3.]]
>>> print(t.contracted)
[ 1.  2.  3.  4.  5.  6.]
>>> s = Tensor(t)
>>> print(s)
[[ 1.  6.  5.]
 [ 6.  2.  4.]
 [ 5.  4.  3.]]
```

`contracted`

Returned the symmetric tensor data as a numpy array with shape (6,)

tensor

Returned the tensor data as a numpy array with shape (3,3)

sym

Return the symmetric part of the tensor.

asym

Return the antisymmetric part of the tensor.

principal (*sort=True, right_handed=True*)

Returns the principal values and axes of the inertia tensor.

Parameters:

- *sort*: bool. If True (default), the return values are sorted in order of decreasing principal values. Otherwise they are unsorted.
- *right_handed*: bool. If True (default), the returned axis vectors are guaranteed to form a right-handed coordinate system. Otherwise, left-handed systems may result)

Returns a tuple (prin,axes) where

- *prin*: is a (3,) array with the principal values,
- *axes*: is a (3,3) array with the rotation matrix that rotates the global axes to the principal axes. This also means that the rows of axes are the unit vectors along the principal directions.

Example:

```
>>> t = Tensor([-19., 4.6, -8.3, 11.8, 6.45, -4.7 ])
>>> p,a = t.principal()
>>> print(p)
[ 11.62 -9.   -25.32]
>>> print(a)
[[-0.03  0.86  0.5 ]
 [-0.62  0.38 -0.69]
 [-0.78 -0.33  0.53]]
```

rotate (*rot*)

Transform the tensor on coordinate system rotation.

Note: for an inertia tensor, the inertia should have been computed around axes through the center of mass. See also `translate`.

Example:

```
>>> t = Tensor([-19., 4.6, -8.3, 11.8, 6.45, -4.7 ])
>>> p,a = t.principal()
>>> print(t.rotate(np.linalg.linalg.inv(a)))
[[ 11.62  0.    0.   ]
 [ -0.   -9.   -0.   ]
 [  0.   -0.  -25.32]]
```

class inertia.Inertia

A class for storing the inertia tensor of an array of points.

Parameters:

- *X*: a `Coords` with shape (npoints,3). Shapes (... ,3) are accepted but will be reshaped to (npoints,3).
- *mass*: optional, (npoints,) float array with the mass of the points. If omitted, all points have mass 1.

The result is a tuple of two float arrays:

- the center of gravity: shape (3,)
- the inertia tensor: shape (6,) with the following values (in order): Ixx, Iyy, Izz, Iyz, Izx, Ixy

Example:

```
>>> from .elements import Tet4
>>> X = Tet4.vertices
>>> print(X)
[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
>>> I = X.inertia()
>>> print(I)
[[ 1.5  0.25  0.25]
 [ 0.25  1.5  0.25]
 [ 0.25  0.25  1.5 ]]
>>> print(I.ctr)
[ 0.25  0.25  0.25]
>>> print(I.mass)
4.0
>>> print(I.translate(-I.ctr))
[[ 2.  0.  0.]
 [ 0.  2.  0.]
 [ 0.  0.  2.]]
```

translate (*trl*, *toG=False*)

Return the inertia tensor around axes translated over vector *trl*.

Parameters:

- *trl*: arraylike (3,). Distance vector from the center of mass to the new reference point.
- *toG*: bool. If False (default) the inertia tensor is translated to the the new reference point, otherwise it will be translated to its center of mass

translateTo (*ref*, *toG=False*)

Return the inertia tensor around axes translated to the reference point *ref*.

Parameters:

- *ref*: arraylike (3,). The new reference point coordinates.
- *toG*: bool. If False (default) the inertia tensor is translated to the the new reference point, otherwise it will be translated to its center of mass

toCS (*cs*)

Transform the coordinates to another CoordSys.

Functions defined in module inertia

inertia.point_inertia (*X*, *mass=None*, *center_only=False*)

Compute the total mass, center of mass and inertia tensor mass points.

Parameters:

- *X*: a CoordSys with shape (npoints,3). Shapes (... ,3) are accepted but will be reshaped to (npoints,3).
- *mass*: optional, (npoints,) float array with the mass of the points. If omitted, all points have mass 1.
- *center_only*: bool: if True, only returns the total mass and center of mass.

Returns a tuple (M,C,I) where M is the total mass of all points, C is the center of mass, and I is the inertia tensor in the central coordinate system, i.e. a coordinate system with axes parallel to the global axes but origin at the (computed) center of mass. If *center_only* is True, returns the tuple (M,C) only. On large models this is more effective in case you do not need the inertia tensor.

`inertia.surface_volume(x, pt=None)`

Return the volume inside a 3-plex Formex.

- *x*: an (ntri,3,3) shaped float array, representing ntri triangles.
- *pt*: a point in space. If unspecified, it is taken equal to the origin of the global coordinate system ([0.,0.,0.]).

Returns an (ntri) shaped array with the volume of the tetrahedrons formed by the triangles and the point *pt*. Triangles with an outer normal pointing away from *pt* will generate positive tetrahedral volumes, while triangles having *pt* at the side of their positive normal will generate negative volumes. In any case, if *x* represents a closed surface, the algebraic sum of all the volumes is the total volume inside the surface.

`inertia.surface_volume_inertia(x, center_only=False)`

Return the inertia of the volume inside a 3-plex Formex.

- *x*: an (ntri,3,3) shaped float array, representing ntri triangles.

This uses the same algorithm as `tetrahedral_inertia` using [0.,0.,0.] as the 4-th point for each tetrahedron.

Returns a tuple (V,C,I) where V is the total volume, C is the center of mass (3,) and I is the inertia tensor (6,) of the tetrahedral model.

Example:

```
>>> from .simple import sphere
>>> S = sphere(4).toFormex()
>>> V,C,I = surface_volume_inertia(S.coords)
>>> print(V,C,I)
4.04701 [-0. -0. -0.] [ 1.58  1.58  1.58 -0.  0.  0. ]
```

`inertia.tetrahedral_volume(x)`

Compute the volume of tetrahedrons.

- *x*: an (ntet,4,3) shaped float array, representing ntet tetrahedrons.

Returns an (ntet,) shaped array with the volume of the tetrahedrons. Depending on the ordering of the points, this volume may be positive or negative. It will be positive if point 4 is on the side of the positive normal formed by the first 3 points.

`inertia.tetrahedral_inertia(x, density=None, center_only=False)`

Return the inertia of the volume of a 4-plex Formex.

Parameters:

- *x*: an (ntet,4,3) shaped float array, representing ntet tetrahedrons.
- *density*: optional mass density (ntet,) per tetrahedron
- *center_only*: bool. If True, returns only the total volume, total mass and center of gravity. This may be used on large models when only these quantities are required.

Returns a tuple (V,M,C,I) where V is the total volume, M is the total mass, C is the center of mass (3,) and I is the inertia tensor (6,) of the tetrahedral model.

Formulas for inertia were based on F. Tonon, J. Math & Stat, 1(1):8-11,2005

Example:

```

>>> x = Coords([
...     [ 8.33220, -11.86875,  0.93355 ],
...     [ 0.75523,  5.00000, 16.37072 ],
...     [ 52.61236,  5.00000, -5.38580 ],
...     [ 2.000000,  5.00000,  3.00000 ],
...     ])
>>> F = Formex([x])
>>> print(tetrahedral_center(F.coords))
[ 15.92  0.78  3.73]
>>> print(tetrahedral_volume(F.coords))
[ 1873.23]
>>> print(*tetrahedral_inertia(F.coords))
1873.23 1873.23 [ 15.92  0.78  3.73] [ 43520.32 194711.28 191168.77  4417.66 -
↪46343.16 11996.2 ]

```

`inertia.tetrahedral_center` (*x*, *density=None*)

Compute the center of mass of a collection of tetrahedrons.

- *x*: an (ntet,4,3) shaped float array, representing ntet tetrahedrons.
- *density*: optional mass density (ntet,) per tetrahedron. Default 1.

Returns a (3,) shaped array with the center of mass.

`inertia.inertia` (*X*, *mass=None*, *center_only=False*)

Compute the total mass, center of mass and inertia tensor mass points.

Parameters:

- *X*: a Coords with shape (npoints,3). Shapes (... ,3) are accepted but will be reshaped to (npoints,3).
- *mass*: optional, (npoints,) float array with the mass of the points. If omitted, all points have mass 1.
- *center_only*: bool: if True, only returns the total mass and center of mass.

Returns a tuple (M,C,I) where M is the total mass of all points, C is the center of mass, and I is the inertia tensor in the central coordinate system, i.e. a coordinate system with axes parallel to the global axes but origin at the (computed) center of mass. If *center_only* is True, returns the tuple (M,C) only. On large models this is more effective in case you do not need the inertia tensor.

6.2.16 fileread — Read geometry from file in a whole number of formats.

This module defines basic routines to read geometrical data from a file and the specialized importers to read files in a number of well known standardized formats.

The basic routines are very versatile as well as optimized (using the version in the pyFormex C-library) and allow to easily create new exporters for other formats.

Functions defined in module fileread

`fileread.getParams` (*line*)

Strip the parameters from a comment line

`fileread.readNodes` (*fil*)

Read a set of nodes from an open mesh file

`fileread.readElems` (*fil*, *nplex*)

Read a set of elems of plexitude nplex from an open mesh file

`fileread.readEsets` (*fil*)

Read the eset data of type generate

`fileread.readMeshFile` (*fn*)

Read a nodes/elems model from file.

Returns a dict:

- *coords*: a Coords with all nodes
- *elems*: a list of Connectivities
- *esets*: a list of element sets

`fileread.extractMeshes` (*d*)

Extract the Meshes read from a .mesh file.

`fileread.convertInp` (*fn*)

Convert an Abaqus .inp to a .mesh set of files

`fileread.readInpFile` (*filename*)

Read the geometry from an Abaqus/Calculix .inp file

This is a replacement for the `convertInp/readMeshFile` combination. It uses the `ccxinp` plugin to provide a direct import of the Finite Element meshes from an Abaqus or Calculix input file. Currently still experimental and limited in functionality (aimed primarily at Calculix). But also many simple meshes from Abaqus can already be read.

Returns an dict.

`fileread.read_off` (*fn*)

Read an OFF surface mesh.

The mesh should consist of only triangles! Returns a nodes,elems tuple.

`fileread.read_gts` (*fn*)

Read a GTS surface mesh.

Return a coords,edges,faces tuple.

`fileread.read_stl_bin` (*fn*)

Read a binary stl.

Parameters *fn* (*str*) – Name of the file to read, holding binary STL data.

Returns *Coords* (*ntri,4,3*) – A Coords with *ntri* triangles. Each triangle consists of 4 items: the first one is the normal, the other three are the coordinates of the vertices.

`fileread.read_gambit_neutral` (*fn*, *eltype='tri'*)

Read a triangular/hexahedral surface mesh in Gambit neutral format.

eltype = 'tri' for triangular, 'hex' for hexahedral mesh. The .neu file nodes are numbered from 1! Returns a nodes,elems tuple.

6.2.17 filewrite — Write geometry to file in a whole number of formats.

This module defines both the basic routines to write geometrical data to a file and the specialized exporters to write files in a number of well known standardized formats.

The basic routines are very versatile as well as optimized (using the version in the pyFormex C-library) and allow to easily create new exporters for other formats.

Functions defined in module filewrite

`filewrite.writeData (fil, data, sep="", fmt=None, end="")`

Write an array of numerical data to an open file.

Parameters:

- *fil*: an open file object
- *data*: a numerical array of int or float type
- *sep*: a string to be used as separator in case no *fmt* is specified. If an empty string, the data are written in binary mode. This is the default. For any other string, the data are written in ascii mode with the specified string inserted as separator between any two items, and a newline appended at the end. In both cases, the data are written using the *numpy.tofile* function.
- *fmt*: a format string compatible with the array data type. If specified, the *sep* argument is ignored and the data are written according to the specified format. This uses the pyFormex functions *misc.tofile_int32* or *misc.tofile_float32*, which have accelerated versions in the pyFormex C library. This also means that the data arrays will be forced to type *float32* or *int32* before writing.

The format string should contain a valid format converter for a single data item in both Python and C. They should also contain the necessary spacing or separator. Examples are `'%5i'` for int data and `'%f,'` or `'%10.3e'` for float data. The array will be converted to a 2D array, keeping the length of the last axis. Then all elements will be written row by row using the specified format string, and the *end* string will be added after each row.

- *end*: a string to be written at the end of the data block (if no *fmt*) or at the end of each row (with *fmt*). The default value is a newline character.

`filewrite.writeIData (data, fil, fmt, ind=1)`

Write an indexed array of numerical data to an open file.

ind = i: autoindex from i array: use these indices

`filewrite.writeOFF (fn, mesh)`

Write a mesh of polygons to a file in OFF format.

Parameters:

- *fn*: file name, by preference ending on `'.off'`
- *mesh*: a Mesh

`filewrite.writeOBJ (fn, mesh, name=None)`

Write a mesh of polygons to a file in OBJ format.

Parameters:

- *fn*: file name, by preference ending on `'.obj'`
- *mesh*: a Mesh
- *name*: name of the Mesh to be written into the file. If None, and the Mesh has an `.attrib.name`, that name will be used.

`filewrite.writePLY (fn, mesh, comment=None)`

Write a mesh to a file in PLY format.

Parameters:

- *fn*: file name, by preference ending on `'.ply'`
- *mesh*: a Mesh

- *comment*: an extra comment to add in the file header.

`filewrite.writeGTS` (*fn*, *coords*, *edges*, *faces*)

Write a mesh of triangles to a file in GTS format.

Parameters:

- *fn*: file name, by preference ending on ‘.gts’
- *coords*: float array with shape (*ncoords*,3), with the coordinates of *ncoords* vertices
- *edges*: int array with shape (*nedges*,2), with the definition of *nedges* edges in function of the vertex indices
- *faces*: int array with shape (*nfaces*,3), with the definition of *nfaces* triangles in function of the edge indices

`filewrite.writeSTL` (*f*, *x*, *n=None*, *binary=False*, *color=None*)

Write a collection of triangles to an STL file.

Parameters:

- *fn*: file name, by preference ending with ‘.stl’ or ‘.stla’
- *x*: (*ntriangles*,3,3) shaped array with the vertices of the triangles
- *n*: (*ntriangles*,3) shaped array with the normals of the triangles. If not specified, they will be calculated.
- *binary*: if True, the output file format will be a binary STL. The default is an ascii STL. Note that creation of a binary STL requires the external program ‘admesh’.
- *color*: a single color can be passed to a binary STL and will be stored in the header.

`filewrite.write_stl_bin` (*fn*, *x*, *color=None*)

Write a binary stl.

Parameters:

- *x*: (*ntri*,4,3) float array describin *ntri* triangles. The first item of each triangle is the normal, the other three are the vertices.
- *color*: (4,) int array with values in the range 0..255. These are the red, green, blue and alpha components of the color. This is a single color for all the triangles, and will be stored in the header of the STL file.

`filewrite.write_stl_asc` (*fn*, *x*)

Write a collection of triangles to an ascii .stl file.

Parameters:

- *fn*: file name, by preference ending with ‘.stl’ or ‘.stla’
- *x*: (*ntriangles*,3,3) shaped array with the vertices of the triangles

6.2.18 `multi` — Framework for multi-processing in pyFormex

This module contains some functions to perform multiprocessing inside pyFormex in a unified way.

Functions defined in module `multi`

`multi.splitArgs` (*args*, *mask=None*, *nproc=-1*, *close=False*)

Split data blocks over multiple processors.

Parameters:

- *args*: a list or tuple of data blocks. All items in the list that need to be split should be arrays with the same first dimension.

- *mask*: list of bool, with same length as *args*. It flags which items in the *args* list are to be split. If not specified, all array type items will be split.
- *nproc*: number of processors intended. If negative (default), it is set equal to the number of processors detected.
- *close*: bool. If True, the elements where the arrays are split are included in both blocks delimited by the element.

Returns a list of *nproc* tuples. Each tuple contains the same number of items as the input *args* and in the same order, whereby the (nonmasked) arrays are replaced by a slice of the array along its first axis, and the masked and non-array items are replicated as is.

This function uses `arraytools.splitar()` for the splitting of the arrays.

Example:

```
>>> splitArgs([np.arange(5), 'abcde'], nproc=3)
[(array([0, 1]), 'abcde'), (array([2]), 'abcde'), (array([3, 4]), 'abcde')]
>>> for i in splitArgs([np.eye(5), '=>', np.arange(5)], nproc=3):
...     print("%s %s %s" % i)
[[ 1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.]] => [0 1]
[[ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  1.]] => [3 4]
>>> for i in splitArgs([np.eye(5), '=>', np.arange(5)], mask=[1, 0, 0], nproc=3):
...     print("%s %s %s" % i)
[[ 1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.]] => [0 1 2 3 4]
[[ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  1.]] => [0 1 2 3 4]
```

`multi.dofunc` (*arg*)

Helper function for the multitask function.

It expects a tuple with (function,args) as single argument.

`multi.multitask` (*tasks*, *nproc=-1*)

Perform tasks in parallel.

Runs a number of tasks in parallel over a number of subprocesses.

Parameters:

- *tasks*: a list of (function,args) tuples, where function is a callable and args is a tuple with the arguments to be passed to the function.
- 'nproc': the number of subprocesses to be started. This may be different from the number of tasks to run: processes finishing a task will pick up a next one. There is no benefit in starting more processes than the number of tasks or the number of processing units available. The default will set *nproc* to the minimum of these two values.

`multi.worker` (*input*, *output*)

Helper function for the multitask function.

This is the function executed by any of the processes started by the multitask function. It takes tuples (function,args) from the input queue, computes the results of the call function(args), and pushes these results on the output queue.

Parameters:

- *input*: Queue holding the tasks to be performed
- *output*: Queue where the results are to be delivered

`multi.multitask2 (tasks, nproc=-1)`

Perform tasks in parallel.

Runs a number of tasks in parallel over a number of subprocesses.

Parameters:

- *tasks* : a list of (function,args) tuples, where function is a callable and args is a tuple with the arguments to be passed to the function.
- ‘*nproc*’: the number of subprocesses to be started. This may be different from the number of tasks to run: processes finishing a task will pick up a next one. There is no benefit in starting more processes than the number of tasks or the number of processing units available. The default will set *nproc* to the minimum of these two values.

6.2.19 software — software.py

A module to help with detecting required software and helper software, and to check the versions of it.

This module is currently experimental. It contains some old functions moved here from `utils.py`.

Functions defined in module software

`software.checkVersion (name, version, external=False)`

Checks a version of a program/module.

name is either a module or an external program whose availability has been registered. Default is to treat *name* as a module. Add `external=True` for a program.

Return value is -1, 0 or 1, depending on a version found that is <, == or > than the requested values. This should normally understand version numbers in the format 2.10.1 Returns -2 if no version found.

`software.hasModule (name, check=False)`

Test if we have the named module available.

Returns a nonzero (version) string if the module is available, or an empty string if it is not.

By default, the module is only checked on the first call. The result is remembered in the `the_version` dict. The optional argument `check==True` forces a new detection.

`software.requireModule (name, version=None, comp='ge')`

Ensure that the named Python module/version is available.

Checks that the specified module is available, and that its version number is not lower than the specified version. If no version is specified, any version is ok.

The default comparison operator ‘*ge*’ can be replaced with one of: ‘*eq*’, ‘*ge*’, ‘*gt*’, ‘*le*’, ‘*lt*’, ‘*ne*’.

Returns if the required module/version could be loaded, else an error is raised.

`software.checkAllModules ()`

Check the existence of all known modules.

`software.checkModule (name, ver=(), fatal=False, quiet=False)`

Check if the named Python module is available, and record its version.

ver is a tuple of:

- `modname`: name of the module to test import
- `vername`: name of the module holding the version string
- more fields are consecutive attributes leading to the version string

The obtained version string is returned, empty if the module could not be loaded. The (name,version) pair is also inserted into the `the_version` dict.

If `fatal=True`, pyFormex will abort if the module can not be loaded.

`software.hasExternal` (*name*, *force=False*)

Test if we have the external command 'name' available.

Returns a nonzero string if the command is available, or an empty string if it is not.

The external command is only checked on the first call. The result is remembered in the `the_external` dict.

`software.requireExternal` (*name*)

Ensure that the named external program is available.

If the module is not available, an error is raised.

`software.checkAllExternals` ()

Check the existence of all known externals.

Returns a dict with all the known externals, detected or not. The detected ones have a non-zero value, usually the version number.

`software.checkExternal` (*name*, *command=None*, *answer=None*, *quiet=False*)

Check if the named external command is available on the system.

`name` is the generic command name, `command` is the command as it will be executed to check its operation, `answer` is a regular expression to match positive answers from the command. `answer` should contain at least one group. In case of a match, the contents of the match will be stored in the `the_external` dict with `name` as the key. If the result does not match the specified answer, an empty value is inserted.

Usually, `command` will contain an option to display the version, and the `answer` re contains a group to select the version string from the result.

As a convenience, we provide a list of predeclared external commands, that can be checked by their name alone.

`software.Shaders` ()

Return a list of the available GPU shader programs.

Shader programs are in the `pyformex/glsl` directory and consist at least of two files: 'vertex_shader_SHADER.c' and 'fragment_shader_SHADER.c'. This function will return a list of all the SHADER filename parts currently available. The default shader programs do not have the '_SHADER' part and will not be contained in this list.

`software.detectedSoftware` (*all=True*)

Return a dict with all detected helper software

`software.formatDict` (*d*, *indent=4*)

Format a dict in nicely formatted Python source representation.

Each (key,value) pair is formatted on a line of the form:

```
key = value
```

If all the keys are strings containing only characters that are allowed in Python variable names, the resulting text is a legal Python script to define the items in the dict. It can be stored on a file and executed.

This format is the storage format of the `Config` class.

`software.compareVersion` (*has, want*)

Check whether a detected version matches the requirements.

has is the version string detected. *want* is the required version string, possibly preceded by one of the doubly underscored comparison operators: `__gt__`, etc. If no comparison operator is specified, `'__eq__'` is assumed.

Note that any tail behind *x.y.z* version is considered to be later version than *x.y.z*.

Returns the result of the comparison: True or False .. rubric:: Examples

```
>>> compareVersion('2.7', '2.4.3')
False
>>> compareVersion('2.7', '>2.4.3')
True
>>> compareVersion('2.7', '>= 2.4.3')
True
>>> compareVersion('2.7', '>= 2.7-rc3')
False
>>> compareVersion('2.7-rc4', '>= 2.7-rc3')
True
```

`software.checkDict` (*has, want*)

Check that software dict has has the versions required in *want*

`software.checkSoftware` (*req, report=False*)

Check that we have the matching components

Returns True or False. If *report=True*, also returns a string with a full report.

`software.registerSoftware` (*req*)

Register the current values of required software

`software.soft2config` (*soft*)

Convert software collection to config

`software.config2soft` (*conf*)

Convert software collection from config

`software.storeSoftware` (*soft, fn, mode='pretty'*)

Store the software collection on file.

`software.readSoftware` (*fn, mode='python'*)

Read the software collection from file.

- *mode* = 'pretty': readable, editable
- *mode* = 'python': readable, editable
- *mode* = 'config': readable, editable
- *mode* = 'pickle': binary

6.3 pyFormex GUI modules

These modules create the components of the pyFormex GUI. They are located under `pyformex/gui`. They depend on the Qt4 framework.

6.3.1 `gui.widgets` — A collection of custom widgets used in the pyFormex GUI

The widgets in this module were primarily created in function of the pyFormex GUI. The user can apply them to change the GUI or to add interactive widgets to his scripts. Of course he can also use all the Qt widgets directly.

Classes defined in module `gui.widgets`

class `gui.widgets.InputItem`(*name*, **args*, ***kargs*)

A single input item.

This is the base class for widgets holding a single input item. A single input item is any item that is treated as a unit and referred to by a single name.

This base class is rarely used directly. Most of the components of an `InputDialog` are subclasses of hereof, each specialized in some form of input data or representation. There is e.g. an `InputInteger` class to input an integer number and an `InputString` for the input of a string. The base class groups the functionality that is common to the different input widgets.

The `InputItem` widget holds a horizontal layout box (`QHBoxLayout`) to group its its components. In most cases there are just two components: a label with the name of the field, and the actual input field. Other components, such as buttons or sliders, may be added. This is often done in subclasses.

The constructor has one required argument: *name*. Other (optional) positional parameters are passed to the `QtWidgets.QWidget` constructor. The remaining keyword parameters are options that somehow change the default behavior of the `InputItem` class.

Parameters:

- *name*: the name used to identify the item. It should be unique for all `InputItems` in the same `InputDialog`. It will be used as a key in the dictionary that returns all the input values in the dialog. It will also be used as the label to display in front of the input field, in case no *text* value was specified.
- *text*: if specified, this text will be displayed in the label in front of the input field. This allows for showing descriptive texts for the input fields in the dialog, while keeping short and simple names for the items in the programming. *text* can be set to an empty string to suppress the creation of a label in front of the input field. This is useful if the input field widget itself already provides a label (see e.g. `InputBool`). *text* can also be a `QPixmap`, allowing for icons to be used as labels.
- *buttons*: a list of (label,function) tuples. For each tuple a button will be added after the input field. The button displays the text and when pressed, the specified function will be executed. The function takes no arguments.
- *data*: any extra data that you want to be stored into the widget. These data are not displayed, but can be useful in the functioning of the widget.
- *enabled*: boolean. If `False`, the `InputItem` will not be enabled, meaning that the user can not enter any values there. Disabled fields are usually displayed in a greyed-out fashion.
- *readonly*: boolean. If `True`, the data are read-only and can not be changed by the user. Unlike disabled items, they are displayed in a normal fashion.
- *tooltip*: A descriptive text which is only shown when the user pauses the cursor for some time on the widget. It can be used to give more comprehensive explanation to first time users.
- *spacer*: string. Only the characters 'l', 'r' and 'c' are relevant. If the string contains an 'l', a spacer is inserted before the label. If the string contains an 'r', a spacer is inserted after the input field. If the string contains a 'c', a spacer is inserted between the label and the input field.

Subclasses should have an `__init__()` method which first constructs a proper widget for the input field, and stores it in the attribute `self.input`. Then the baseclass should be properly initialized, passing any optional parameters:

```
self.input = SomeInputWidget()
InputItem.__init__(self, name, *args, **kwargs)
```

Subclasses should also override the following default methods of the `InputItem` base class:

- `text()`: if the subclass calls the superclass `__init__()` method with a value `text=' '`. This method should return the value of the displayed text.
- `value()`: if the value of the input field is not given by `self.input.text()`, i.e. in most cases. This method should return the value of the input field.
- `setValue(val)`: always, unless the field is readonly. This method should change the value of the input widget to the specified value.

Subclasses are allowed to NOT have a `self.input` attribute, IFF they redefine both the `value()` and the `setValue()` methods.

Subclasses can set validators on the input, like:

```
self.input.setValidator(QtGui.QIntValidator(self.input))
```

Subclasses can define a `show()` method e.g. to select the data in the input field on display of the dialog.

name()
Return the name of the `InputItem`.

text()
Return the displayed text of the `InputItem`.

value()
Return the widget's value.

setValue(val)
Change the widget's value.

class `gui.widgets.InputInfo` (*name, value, *args, **kwargs*)
An unchangeable input field with a label in front.

It is just like an `InputString`, but the text can not be edited. The value should be a simple string without newlines.

There are no specific options.

value()
Return the widget's value.

class `gui.widgets.InputLabel` (*name, value, *args, **kwargs*)
An unchangeable information field.

The value is displayed as a string, but may contain more complex texts.

By default, the text format will be guessed to be either plain text, `ReStructuredText` or `html`. Specify `plain=True` to display in plain text.

setValue(val)
Change the widget's value.

class `gui.widgets.InputString` (*name, value, max=None, *args, **kwargs*)
A string input field with a label in front.

If the type of value is not a string, the input string will be eval'ed before returning.

Options:

- *max*: the maximum number of characters in the string.

show (*args)

Select all text on first display.

value ()

Return the widget's value.

class `gui.widgets.InputText` (*name, value, *args, **kargs*)

A scrollable text input field with a label in front.

By default, the text format will be guessed to be either plain text, ReStructuredText or html.

Specify `plain=True` to display in plain text.

If the type of value is not a string, the input text will be eval'ed before returning.

show (*args)

Select all text on first display.

value ()

Return the widget's value.

setValue (*val*)

Change the widget's value.

class `gui.widgets.InputBool` (*name, value, *args, **kargs*)

A boolean input item.

Creates a new checkbox for the input of a boolean value.

Displays the name next to a checkbox, which will initially be set if value evaluates to True. (Does not use the label) The value is either True or False, depending on the setting of the checkbox.

Options:

- *func*: an optional function to be called whenever the value is changed. The function receives the input field as argument. With this argument, the fields attributes like name, value, text, can be retrieved.

text ()

Return the displayed text.

value ()

Return the widget's value.

setValue (*val*)

Change the widget's value.

class `gui.widgets.InputList` (*name, default=[], choices=[], sort=False, single=False, check=False, fast_sel=False, maxh=-1, *args, **kargs*)

A list selection InputItem.

A list selection is a widget allowing the selection of zero, one or more items from a list.

choices is a list/tuple of possible values. *default* is the initial/default list of selected items. Values in *default* that are not in the *choices* list, are ignored. If *default* is None or an empty list, nothing is selected initially.

By default, the user can select multiple items and the return value is a list of all currently selected items. If *single* is True, only a single item can be selected.

If *maxh*==*-1*, the widget gets a fixed height to precisely take the number of items in the list. If *maxh*>=0, the widget will get scrollbars when the height is not sufficient to show all items. With *maxh*>0, the item will get the specified height (in pixels), while *maxh*==0 will try to give the widget the required height to show all items

If `check` is `True`, all items have a checkbox and only the checked items are returned. This option sets `single==False`.

setSelected (*selected*, *flag=True*)

Mark the specified items as selected or not.

setChecked (*selected*, *flag=True*)

Mark the specified items as checked or not.

value ()

Return the widget's value.

setValue (*val*)

Change the widget's value.

setAll ()

Mark all items as selected/checked.

setNone ()

Mark all items as not selected/checked.

class `gui.widgets.InputCombo` (*name*, *value*, *choices=[]*, *onselect=None*, *func=None*, **args*, ***kargs*)

A combobox `InputItem`.

A combobox is a widget allowing the selection of an item from a drop down list.

`choices` is a list/tuple of possible values. `value` is the initial/default choice. If `value` is not in the `choices` list, it is prepended.

The choices are presented to the user as a combobox, which will initially be set to the default value.

An optional `onselect` function may be specified, which will be called whenever the current selection changes. The function is passed the selected option string

value ()

Return the widget's value.

setValue (*val*)

Change the widget's current value.

setChoices (*choices*)

Change the widget's choices.

This also sets the current value to the first in the list.

class `gui.widgets.InputRadio` (*name*, *value*, *choices=[]*, *direction='h'*, **args*, ***kargs*)

A radiobuttons `InputItem`.

Radio buttons are a set of buttons used to select a value from a list.

`choices` is a list/tuple of possible values. `value` is the initial/default choice. If `value` is not in the `choices` list, it is prepended. If `value` is `None`, the first item of `choices` is taken as the default.

The choices are presented to the user as a `hbox` with radio buttons, of which the default will initially be pressed. If `direction == 'v'`, the options are in a `vbox`.

value ()

Return the widget's value.

setValue (*val*)

Change the widget's value.

class `gui.widgets.InputPush` (*name, value=None, choices=[], direction='h', count=0, icon=None, iconsonly=False, func=None, *args, **kargs*)

A pushbuttons InputItem.

Creates pushbuttons for the selection of a value from a list.

`choices` is a list/tuple of possible values. `value` is the initial/default choice. If `value` is not in the `choices` list, it is prepended. If `value` is `None`, the first item of `choices` is taken as the default.

The choices are presented to the user as a hbox with push buttons, of which the default will initially be selected. If `direction == 'v'`, the options are in a vbox.

Extra parameters:

- *func*: the function to call when the button is clicked. The function receives the input field as argument. From this argument, the fields attributes like `name`, `value`, `text`, can be retrieved. The function should return the value to be set, or `None` if it is to be unchanged. If no function is specified, the value can not be changed.

setText (*text, index=0*)

Change the text on button index.

setIcon (*icon, index=0*)

Change the icon on button index.

value ()

Return the widget's value.

setValue (*val*)

Change the widget's value.

doFunc ()

Set the value by calling the button's `func`

class `gui.widgets.InputInteger` (*name, value, *args, **kargs*)

An integer input item.

Options:

- *min, max*: range of the scale (integer)

show ()

Select all text on first display.

value ()

Return the widget's value.

setValue (*val*)

Change the widget's value.

class `gui.widgets.InputFloat` (*name, value, *args, **kargs*)

A float input item.

show ()

Select all text on first display.

value ()

Return the widget's value.

setValue (*val*)

Change the widget's value.

```
class gui.widgets.InputTable (name, value, chead=None, rhead=None, celltype=None,
rowtype=None, coltype=None, edit=True, resize=None, autowidth=True, *args, **kwargs)
```

An input item for tabular data.

- *value*: a 2-D array of items, with *nrow* rows and *ncol* columns.

If *value* is a numpy array, the Table will use the ArrayModel: editing the data will directly change the input data array; all items are of the same type; the size of the table can not be changed.

Else a TableModel is used. Rows and columns can be added to or removed from the table. Item type can be set per row or per column or for the whole table.

- *autowidth*:
- additionally, all keyword parameters of the TableModel or ArrayModel may be passed

```
value ()
```

Return the widget's value.

```
class gui.widgets.InputSlider (name, value, *args, **kwargs)
```

An integer input item using a slider.

Options:

- *min, max*: range of the scale (integer)
- *ticks*: step for the tick marks (default range length / 10)
- *func*: an optional function to be called whenever the value is changed. The function receives the input field as argument. With this argument, the fields attributes like name, value, text, can be retrieved.
- *tracking*: bool. If True (default), *func* is called repeatedly while the slider is being dragged. If False, *func* is only called when the user releases the slider.

```
class gui.widgets.InputFSlider (name, value, *args, **kwargs)
```

A float input item using a slider.

Options:

- *min, max*: range of the scale (integer)
- *scale*: scale factor to compute the float value
- *ticks*: step for the tick marks (default range length / 10)
- *func*: an optional function to be called whenever the value is changed. The function receives the input field as argument. With this argument, the fields attributes like name, value, text, can be retrieved.
- *tracking*: bool. If True (default), *func* is called repeatedly while the slider is being dragged. If False, *func* is only called when the user releases the slider.

```
class gui.widgets.InputPoint (name, value, ndim=3, *args, **kwargs)
```

A 2D/3D point/vector input item.

The default gives fields x, y and z. With ndim=2, only x and y.

```
value ()
```

Return the widget's value.

```
setValue (val)
```

Change the widget's value.

```
class gui.widgets.InputIVector (name, value, *args, **kwargs)
```

A vector of int values.

value ()
Return the widget's value.

setValue (val)
Change the widget's value.

class `gui.widgets.InputButton (name, value, *args, **kargs)`
A button input item.

The button input field is a button displaying the current value. Clicking on the button executes a function responsible for changing the value.

Extra parameters:

- *func*: the function to call when the button is clicked. The function receives the input field as argument. From this argument, the fields attributes like name, value, text, can be retrieved. The function should return the value to be set, or None if it is to be unchanged. If no function is specified, the value can not be changed.

doFunc ()
Set the value by calling the button's func

class `gui.widgets.InputColor (name, value, *args, **kargs)`
A color input item. Creates a new color input field with a label in front.

The color input field is a button displaying the current color. Clicking on the button opens a color dialog, and the returned value is set in the button.

Options:

- *func*: an optional function to be called whenever the value is changed. The function receives the input field as argument. With this argument, the fields attributes like name, value, text, can be retrieved.

setValue (value)
Change the widget's value.

class `gui.widgets.InputFont (name, value, *args, **kargs)`
An input item to select a font.

class `gui.widgets.InputFilename (name, value, filter='*', exist=False, preview=None, **kargs)`
A filename input item.

This is a button input field displaying a file name. Clicking on the button pops up a file dialog.

Extra parameters:

- *filter*: the filter for the filenames to accept. See `askFilename`.
- *exist*: boolean. Specifies whether the file should exist, or a new one can be created (default).
- *preview*: an `ImageView` widget, or another widget having a `showImage` method. This can be used with image files to show a preview of the selected file. In most cases the preview widget is inserted in a dialog directly below the `InputFilename` field.

changeFilename ()
Pop up a `FileDialog` to change the filename

class `gui.widgets.InputFile (name, value, pattern='*', exist=False, multi=False, dir=False, compr=False, *args, **kargs)`

An input item to select a file.

The following arguments are passed to the `FileDialog` widget: value,pattern,exist,multi,dir,compr.

value ()
Return the widget's value.

setValue (*value*)

Change the widget's value.

class `gui.widgets.InputWidget` (*name, value, *args, **kargs*)

An input item containing any other widget.

The widget should have:

- a `results` attribute that is set to a dict with the resulting input values when the widget's `acceptData()` is called.
- an `acceptData()` method, that sets the widgets results dict.
- a `setValue(dict)` method that sets the widgets values to those specified in the dict.

The return value of this item is an `OrderedDict`.

text ()

Return the displayed text.

value ()

Return the widget's value.

setValue (*val*)

Change the widget's value.

class `gui.widgets.InputGroup` (*name, *args, **kargs*)

A boxed group of `InputItems`.

value ()

Return the widget's value.

setValue (*val*)

Change the widget's value.

class `gui.widgets.InputHBox` (*name, hbox, *args, **kargs*)

A column in a `hbox` input form.

class `gui.widgets.InputTab` (*name, tab, *args, **kargs*)

A tab page in an input form.

class `gui.widgets.InputForm`

An input form.

The input form is a layout box in which the items are layed out vertically. The layout can also contain any number of tab widgets in which items can be layed out using tab pages.

class `gui.widgets.InputDialog` (*items, caption=None, parent=None, flags=None, actions=None, default=None, store=None, prefix="", autoprefix=False, flat=None, modal=None, enablers=[], scroll=False, button-sattop=False, size=None, align_right=False*)

A dialog widget to interactively set the value of one or more items.

Overview

The pyFormex user has full access to the Qt4 framework on which the GUI was built. Therefore he can built input dialogs as complex and powerful as he can imagine. However, directly dealing with the Qt4 libraries requires some skills and, for simple input widgets, more effort than needed.

The `InputDialog` class presents a unified system for quick and easy creation of common dialog types. The provided dialog can become quite sophisticated with tabbed pages, groupboxes and custom widgets. Both modal and modeless (non-modal) dialogs can be created.

Items

Each basic input item is a dictionary, where the fields have the following meaning:

- **name**: the name of the field,
- **value**: the initial or default value of the field,
- **itemtype**: the type of values the field can accept,
- **options**: a dict with options for the field.
- **text**: if specified, the text value will be displayed instead of the name. The name value will remain the key in the return dict. Use this field to display a more descriptive text for the user, while using a short name for handling the value in your script.
- **buttons**:
- **tooltip**:
- **min**:
- **max**:
- **scale**:
- **func**:

For convenience, simple items can also be specified as a tuple. A tuple (key,value) will be transformed to a dict {'key':key, 'value':value}.

Other arguments

- **caption**: the window title to be shown in the window decoration
- **actions**: a list of action buttons to be added at the bottom of the input form. By default, a Cancel and Ok button will be added, to either reject or accept the input values.
- **default**: the default action
- **parent**: the parent widget (by default, this is the pyFormex main window)
- **autoprefix**: if True, the names of items inside tabs and group boxes will get prefixed with the tab and group names, separated with a '/'.
 If False, the results will be structured: the value of a tab or a group is a dictionary with the results of its fields. The default value is equal to the value of autoprefix.
- **flat**: if True, the results are returned in a single (flat) dictionary, with keys being the specified or autoprefixed ones. If False, the results will be structured: the value of a tab or a group is a dictionary with the results of its fields. The default value is equal to the value of autoprefix.
- **flags**:
- **modal**:
- **enablers**: a list of tuples (key,value,key1,...) where the first two items indicate the key and value of the enabler, and the next items are keys of fields that are enabled when the field key has the specified value. Currently, key should be a field of type boolean, [radio], combo or group. Also, any input field should only have one enabler!

add_items (*items, form, prefix=""*)

Add input items to form.

items is a list of input item data layout is the widget layout where the input widgets will be added

add_group (*form, prefix, name, items, **extra*)

Add a group of input items.

add_hbox (*form, prefix, name, items, **extra*)

Add a hbox of input items.

add_tab (*form, prefix, name, items, **extra*)

Add a Tab page of input items.

add_input (*form, prefix, **item*)

Add a single input item to the form.

timeout ()

Hide the dialog and set the result code to TIMEOUT

timedOut ()

Returns True if the result code was set to TIMEOUT

show (*timeout=None, timeoutfunc=None, modal=False*)

Show the dialog.

For a non-modal dialog, the user has to call this function to display the dialog. For a modal dialog, this is implicitly executed by `getResults()`.

If a timeout is given, start the timeout timer.

acceptData (*result=PySide2.QtWidgets.QDialog.DialogCode.Accepted*)

Update the dialog's return value from the field values.

This function is connected to the 'accepted()' signal. Modal dialogs should normally not need to call it. In non-modal dialogs however, you can call it to update the results without having to raise the accepted() signal (which would close the dialog).

updateData (*d*)

Update a dialog from the data in given dictionary.

d is a dictionary where the keys are field names in the dialog. The values will be set in the corresponding input items.

getResults (*timeout=None*)

Get the results from the input dialog.

This function is used to present a modal dialog to the user (i.e. a dialog that must be ended before the user can continue with the program. The dialog is shown and user interaction is processed. The user ends the interaction either by accepting the data (e.g. by pressing the OK button or the ENTER key) or by rejecting them (CANCEL button or ESC key). On accept, a dictionary with all the fields and their values is returned. On reject, an empty dictionary is returned.

If a timeout (in seconds) is given, a timer will be started and if no user input is detected during this period, the input dialog returns with the default values set. A value 0 will timeout immediately, a negative value will never timeout. The default is to use the global variable `input_timeout`.

The `result()` method can be used to find out how the dialog was ended. Its value will be one of ACCEPTED, REJECTED or TIMEOUT.

class `gui.widgets.ListWidget` (*maxh=0*)

A customized QListWidget with ability to compute its required size.

class `gui.widgets.TableModel` (*data, chead=None, rhead=None, celltype=None, rowtype=None, coltype=None, edit=True, resize=None*)

A model representing a two-dimensional array of items.

- *data*: any tabular data organized in a fixed number of rows and columns. This means that an item at row *i* and column *j* can be addressed as `data[i][j]`. Thus it can be a list of lists, or a list of tuples or a 2D numpy array. The data will always be returned as a list of lists though. Unless otherwise specified by the use of a *celltype*, *coltype* or *rowtype* argument, all items are converted to strings and will be returned as strings. Item storage order is row by row.
- *thead*: optional list of (ncols) column headers

- *rhead*: optional list of (nrows) row headers
- *celltype*: callable: if specified, it is used to map all items. This is only used if neither *rowtype* nor *coltype* are specified. If unspecified, it will be set to 'str', unless *data* is a numpy array, in which case it will be set to the datatype of the array.
- *rowtype*: list of nrows callables: if specified, the items of each row are mapped by the corresponding callable. This overrides *celltype* and is only used if *coltype* is not specified.
- *coltype*: list of ncols callables: if specified, the items of each column are mapped by the corresponding callable. This overrides *celltype* and *rowtype*.
- *edit*: bool: if True (default), the table is editable. Set to False to make the data readonly.
- *resize*: bool: if True, the table can be resized: rows and columns can be added or removed. If False, the size of the table can not be changed. The default value is equal to the value of *edit*. If *coltype* is specified, the number of columns can not be changed. If *rowtype* is specified, the number of rows can not be changed.

makeEditable (*edit=True, resize=None*)

Make the table editable or not.

- *edit*: bool: makes the items in the table editable or not.
- *resize*: bool: makes the table resizable or not. If unspecified, it is set equal to the *edit*.

rowCount (*parent=None*)

Return number of rows in the table

columnCount (*parent=None*)

Return number of columns in the table

data (*index, role*)

Return the data at the specified index

cellType (*r, c*)

Return the type of the item at the specified position

setCellData (*r, c, value*)

Set the value of an individual table element.

This changes the stored data, not the interface.

setData (*index, value, role=PySide2.QtCore.Qt.ItemDataRole.EditRole*)

Set the value of an individual table element.

headerData (*col, orientation, role*)

Return the header data for the specified row or column

insertRows (*row=None, count=None*)

Insert row(s) in table

removeRows (*row=None, count=None*)

Remove row(s) from table

flags (*index*)

Return the TableModel flags.

class `gui.widgets.ArrayModel` (*data, chead=None, rhead=None, edit=True*)

A model representing a two-dimensional numpy array.

- *data*: a numpy array with two dimensions.
- *thead, rhead*: column and row headers. The default will show column and row numbers.
- *edit*: if True (default), the data can be edited. Set to False to make the data readonly.

makeEditable (*edit=True*)

Make the table editable or not.

- *edit*: bool: makes the items in the table editable or not.

rowCount (*parent=None*)

Return number of rows in the table

columnCount (*parent=None*)

Return number of columns in the table

cellType (*r, c*)

Return the type of the item at the specified position

setData (*index, value, role=PySide2.QtCore.Qt.ItemDataRole.EditRole*)

Set the value of an individual table element.

headerData (*col, orientation, role*)

Return the header data for the specified row or column

flags (*index*)

Return the TableModel flags.

class `gui.widgets.Table` (*data, chead=None, rhead=None, label=None, celltype=None, rowtype=None, coltype=None, edit=True, resize=None, parent=None, autowidth=True*)

A widget to show/edit a two-dimensional array of items.

- *data*: a 2-D array of items, with *nrow* rows and *ncol* columns.

If *data* is a numpy array, the Table will use the ArrayModel: editing the data will directly change the input data array; all items are of the same type; the size of the table can not be changed.

Else a TableModel is used. Rows and columns can be added to or removed from the table. Item type can be set per row or per column or for the whole table.

- *label*: currently unused (intended to display an optional label in the upper left corner if both *thead* and *rhead* are specified).
- *parent*:
- *autowidth*:
- additionally, all other parameters for the initialization of the TableModel or ArrayModel may be passed

colWidths ()

Return the width of the columns in the table

rowHeights ()

Return the height of the rows in the table

update ()

Update the table.

This method should be called to update the widget when the data of the table have changed. If *autowidth* is True, this will also adjust the column widths.

value ()

Return the Table's value.

class `gui.widgets.FileDialog` (*path='.', pattern='', exist=False, multi=False, dir=False, compr=False, button=None, sidebar=None, caption=None, native=False, **kargs*)

A file selection dialog.

The FileDialog dialog is a special purpose complex dialog widget that allows to interactively select a file or directory from the file system, possibly even multiple files, create new files or directories.

Parameters:

- *path*: the path shown on initial display of the dialog. It should be an existing path in the file system. The default is '.' for the current directory.
- *pattern*: a string or a list of strings: specifies one or more UNIX glob patterns, used to limit the set of displayed filenames to those matching the glob. Each string can contain multiple globs, and an explanation string can be place in front:

```
'Image files (*.png *.jpg)'
```

The *pattern* argument is passed to the `utils.fileDescription()` function, together with the *compr* argument, to generate the actual pattern set. This allows the creation of filters for common file types with a minimal input.

If a list of multiple strings is given, a combo box will allow the user to select between one of them.

- *exist*: bool: if True, the filename must exist. The default will allow any new filename to be created.
- *multi*: bool: if True, multiple files can be selected. The default is to allow only a single file.
- *dir*: bool: if True, only directories can be selected. If *dir* evaluates to True, but is not the value True, either a directory or a filename can be selected.
- *compr*: bool: if True, compressed files of the specified type will be selectable as well. This is passed together with the *pattern* argument to the `utils.fileDescription()` to generate the actual patterns.
- *button*: string: the label to be displayed on the accept button. The default is set to 'Save' if new files are allowed or 'Open' if only existing files can be selected.

setFilters (*patterns*)

Set filter based on name patterns.

Parameters *patterns* (*list of str*) – Each string has the format 'DESCRIPTION (PATTERNS)' where DESCRIPTION is a text describing the file type and PATTERNS is one or more filename matching patterns, separated by blanks.

See also:

`utils.fileDescriptions()` predefined patterns for most common file types.

value ()

Return the selected value

getResults (*timeout=None*)

Ask the user to fill in the dialog.

Returns a dict. If the user accepts the results, the dict has a single entry with key 'fn' and the selected filename(s) as value. If the user hits CANCEL or ESC, an empty dict is returned.

getFilename (*timeout=None*)

Ask for filename(s) by user interaction.

Returns *Path | list of Paths | None* – The filename(s) selected by the user if the user accepts the selection. Returns None if the user hits CANCEL or ESC.

```
class gui.widgets.GeometryFileDialog (path=None, pattern=None, exist=False,  
mode='binary', compression=4, access=None, de-  
fault=None, convert=True, **kargs)
```

A file selection dialog specialized for opening pgf files.

getResults (*timeout=None*)

Ask the user to fill in the dialog.

Returns a Dict or dict. If the user accepts the results, a Dict with the following entries is returned: *fn*, *acc*, and optional *mod*, *cpr*, *cvt*. If the user hits CANCEL or ESC, an empty dict is returned.

class `gui.widgets.ProjectSelection` (*path=None, pattern=None, exist=False, compression=4, protocol=None, access=None, default=None, convert=True*)

A file selection dialog specialized for opening projects.

getResults ()

Ask the user to fill in the dialog.

Returns a dict. If the user accepts the results, the dict has a single entry with key 'fn' and the selected filename(s) as value. If the user hits CANCEL or ESC, an empty dict is returned.

class `gui.widgets.SaveImageDialog` (*path=None, pattern=None, exist=False, multi=False*)

A dialog for saving to an image file.

The dialog contains the normal file selection widget plus some extra fields to set the Save Image parameters:

- *Grab Screen*: If checked, the image will be cropped from the screen video buffers. This is implied by the 'Whole Window' and 'Add Border' options
- *Whole Window*: If checked, the whole pyFormex main window will be saved. If unchecked, only the current OpenGL viewport is saved.

getResults ()

Ask the user to fill in the dialog.

Returns a dict. If the user accepts the results, the dict has a single entry with key 'fn' and the selected filename(s) as value. If the user hits CANCEL or ESC, an empty dict is returned.

class `gui.widgets.ListSelection` (*choices, caption='ListSelection', default=[], single=False, check=False, sort=False, *args, **kwargs*)

A dialog for selecting one or more items from a list.

This is a convenient class which constructs an input dialog with a single input item: an `InputList`. It allows the user to select one or more items from a list. The constructor supports all arguments of the `InputDialog` and the `InputList` classes. The return value is the value of the `InputList`, not the result of the `InputDialog`.

setValue (*selected*)

Mark the specified items as selected.

value ()

Return the selected items.

getResults ()

Show the modal dialog and return the list of selected values.

If the user cancels the selection operation, the return value is `None`. Else, the result is always a list, possibly empty or with a single value.

class `gui.widgets.GenericDialog` (*widgets, title=None, parent=None, actions=[('OK',)], default='OK'*)

A generic dialog widget.

The dialog is formed by a number of widgets stacked in a vertical box layout. At the bottom is a horizontal button box with possible actions.

- *widgets*: a list of widgets to include in the dialog
- *title*: the window title for the dialog

- *parent*: the parent widget. If None, it is set to pf.GUI.
- *actions*: the actions to include in the bottom button box. By default, an 'OK' button is displayed to close the dialog. Can be set to None to avoid creation of a button box.
- *default*: the default action, 'OK' by default.

class `gui.widgets.MessageBox` (*text, format=""*, *level='info'*, *actions=['OK']*, *default=None*, *timeout=None*, *modal=None*, *parent=None*, *check=None*)

A message box is a widget displaying a short text for the user.

The message box displays a text, an optional icon depending on the level and a number of push buttons.

- *text*: the text to be shown. This can be either plain text or html or reStructuredText.
- *format*: the text format: either 'plain', 'html' or 'rest'. Any other value will try automatic recognition. Recognition of plain text and html is automatic. A text is autorecognized to be reStructuredText if its first line starts with '..' and is followed by a blank line.
- *level*: defines the icon that will be shown together with the text. If one of 'question', 'info', 'warning' or 'error', a matching icon will be shown to hint the user about the type of message. Any other value will suppress the icon.
- *actions*: a list of strings. For each string a pushbutton will be created which can be used to exit the dialog and remove the message. By default there is a single button labeled 'OK'.

When the MessageBox is displayed with the `getResults()` method, a modal dialog is created, i.e. the user will have to click a button or hit the ESC key before he can continue.

If you want a modeless dialog, allowing the user to continue while the message stays open, use the `show()` method to display it.

addCheck (*text*)

Add a check field at the bottom of the layout.

getResults ()

Display the message box and wait for user to click a button.

This will show the message box as a modal dialog, so that the user has to click a button (or hit the ESC key) before he can continue. Returns the text of the button that was clicked or an empty string if ESC was hit.

class `gui.widgets.TextBox` (*text, format=None*, *actions=[('OK',)]*, *modal=None*, *parent=None*, *caption=None*, *mono=False*, *timeout=None*, *flags=None*)

Display a text and wait for user response.

Possible choices are 'OK' and 'CANCEL'. The function returns True if the OK button was clicked or 'ENTER' was pressed, False if the 'CANCEL' button was pressed or ESC was pressed.

class `gui.widgets.ButtonBox` (*name=""*, *actions=None*, *default=None*, *parent=None*, *spacer=False*, *stretch=[-1, -1]*, *cmargin=(2, 2, 2)*)

A box with action buttons.

- *name*: a label to be displayed in front of the button box. An empty string will suppress it.
- *actions*: a list of (button label, button function) tuples. The button function can be a normal callable function, or one of the values `widgets.ACCEPTED` or `widgets.REJECTED`. In the latter case, *parent* should be specified.
- *default*: name of the action to set as the default. If no default is given, it will be set to the LAST button.
- *parent*: the parent dialog holding this button box. It should be specified if one of the buttons actions is not specified or is `widgets.ACCEPTED` or `widgets.REJECTED`.

setText (*text*, *index=0*)

Change the text on button index.

setIcon (*icon*, *index=0*)

Change the icon on button index.

class `gui.widgets.CoordsBox` (*ndim=3*, *readonly=False*, **args*)

A widget displaying the coordinates of a point.

getValues ()

Return the current x,y,z values as a list of floats.

setValues (*values*)

Set the three values of the widget.

class `gui.widgets.ImageView` (*image=None*, *maxheight=None*, *parent=None*)

A widget displaying an image.

showImage (*image*, *maxheight=None*)

Show an image in the viewer.

image: either a filename or an existing QImage instance. If a filename, it should be an image file that can be read by the QImage constructor. Most image formats are understood by QImage. The variable `gui.image.image_formats_qtr` provides a list.

Functions defined in module `gui.widgets`

`gui.widgets.pyformexIcon` (*icon*)

Create a pyFormex icon.

Returns a QIcon with an image taken from the pyFormex icons directory. *icon* is the basename of the image file (.xpm or .png).

`gui.widgets.objSize` (*object*)

Return the width and height of an object.

Returns a tuple w,h for any object that has width and height methods.

`gui.widgets.maxWinSize` ()

Return the maximum widget size.

The maximum widget size is the maximum size for a window on the screen. The available size may be smaller than the physical screen size (e.g. it may exclude the space for docking panels).

`gui.widgets.addTimeout` (*widget*, *timeout=None*, *timeoutfunc=None*)

Add a timeout to a widget.

This enables calling a function or a widget method after a specified time has elapsed.

Parameters

- **widget** (*QWidget*) – The widget to set the timeout function for.
- **timeoutfunc** (*callable*, *optional*) – Function to be called after the widget times out. If None, and the widget has a *timeout* method, that will be used.
- **timeout** (*float*, *optional*) – The time in seconds to wait before calling the timeout function. If None, it will be set to to the global `widgets.input_timeout`.

Notes

If timeout is positive, a timer is installed into the widget which will call the *timeoutfunc* after *timeout* seconds have elapsed. The *timeoutfunc* can be any callable, but usually will emit a signal to make the widget accept or reject the input. The *timeoutfunc* will not be called if the widget is destructed before the timer has finished.

`gui.widgets.defaultItemType (item)`

Guess the InputItem type from the value

`gui.widgets.simpleInputItem (name, value=None, itemtype=None, **kargs)`

A convenience function to create an InputItem dictionary

`gui.widgets.groupInputItem (name, items=[], **kargs)`

A convenience function to create an InputItem dictionary

`gui.widgets.hboxInputItem (name, items=[], **kargs)`

A convenience function to create an InputItem dictionary

`gui.widgets.tabInputItem (name, items=[], **kargs)`

A convenience function to create an InputItem dictionary

`gui.widgets.convertInputItem (item)`

Convert InputItem item to a dict or a widget.

This function tries to convert some old style or sloppy InputItem item to a proper InputItem item dict.

The conversion does the following:

- if *item* is a dict, it is considered a proper item and returned as is.
- if *item* is a QWidget, it is also returned as is.
- if *item* is a tuple or a list, conversion with `simpleInputItem` is tried, using the item items as arguments.
- if neither succeeds, an error is raised.

`gui.widgets.inputAny (name, value, itemtype, **options)`

Create an InputItem of any type, depending on the arguments.

Arguments: only name, value and itemtype are required

- name: name of the item, also the key for the return value
- value: initial value,
- itemtype: one of the available itemtypes

`gui.widgets.updateDialogItems (data, newdata)`

Update the input data fields with new data values

- data: a list of dialog items, as required by an InputDialog.
- newdata: a dictionary with new values for (some of) the items.

The data items with a name occurring as a key in *newdata* will have their value replaced with the corresponding value in *newdata*, unless this value is *None*.

The user should make sure to set only values of the proper type!

`gui.widgets.fileUrls (files)`

Transform a list of local file names to urls

`gui.widgets.selectFont ()`

Ask the user to select a font.

A font selection dialog widget is displayed and the user is requested to select a font. Returns a font if the user exited the dialog with the *OK* button. Returns *None* if the user clicked *CANCEL*.

`gui.widgets.getColor (col=None, caption=None)`

Create a color selection dialog and return the selected color.

`col` is the initial selection. If a valid color is selected, its string name is returned, usually as a hex `#RRGGBB` string. If the dialog is canceled, *None* is returned.

`gui.widgets.updateText (widget, text, format="")`

Update the text of a text display widget.

- *widget*: a widget that has the `setText()`, `setPlainText()` and `setHtml()` methods to set the widget's text. Examples are `QMessageBox` and `QTextEdit`.
- *text*: a multiline string with the text to be displayed.
- *format*: the format of the text. If empty, autorecognition will be tried. Currently available formats are: `plain`, `html` and `rest` (`reStructuredText`).

This function allows to display other text formats besides the plain text and html supported by the widget. Any format other than `plain` or `html` will be converted to one of these before sending it to the widget. Currently, we convert the following formats:

- `rest` (`reStructuredText`): If the `:mod:docutils` is available, *rest* text is converted to *html*, otherwise it will be displayed as plain text. A text is autorecognized as `reStructuredText` if its first line starts with `..`. Note: If you add a `..` line to your text to have it autorecognized as `reST`, be sure to have it followed with a blank line, or your first paragraph could be turned into comments.

`gui.widgets.addActionButtons (layout, actions=[('Cancel',), ('OK',)], default=None, parent=None)`

Add a set of action buttons to a layout

`layout` is a `QLayout`

`actions` is a list of tuples (name,) or (name,function). If a function is specified, it will be executed on pressing the button. If no function is specified, and name is one of 'ok' or 'cancel' (case is ignored), the button will be bound to the dialog's 'accept' or 'reject' slot. If `actions` is *None* (default), it will be set to the default `[('Cancel',), ('OK',)]`.

`default` is the name of the action to set as the default. If no default is given, it is set to the `LAST` button.

Returns a list of `QPushButton`s.

`gui.widgets.setCustomColors (col)`

Set `QColorDialog` Custom colors.

`col` is a list of max. 16 color values (any values accepted by `colors.RGBcolor`)

6.3.2 `gui.menu` — Menus for the pyFormex GUI.

This modules implements specialized classes and functions for building the pyFormex GUI menu system.

Classes defined in module `gui.menu`

class `gui.menu.BaseMenu (title='AMenu', parent=None, before=None, items=None)`

A general menu class.

A hierarchical menu that keeps a list of its item names and actions. The item names are normalized by removing all '&' characters and converting the result to lower case. It thus becomes easy to search for an existing item in a menu.

This class is not intended for direct use, but through subclasses. Subclasses should implement at least the following methods:

- `addSeparator()`
- `insertSeperator(before)`
- `addAction(text,action)`
- `insertAction(before,text,action)`
- `addMenu(text,menu)`
- `insertMenu(before,text,menu)`

`QtWidgets.Menu` and `QtWidgets.MenuBar` provide these methods.

`actionList ()`

Return a list with the current actions.

`actionsLike (clas)`

Return a list with the current actions of given class.

`subMenus ()`

Return a list with the submenus

`index (text)`

Return the index of the specified item in the actionlist.

If the requested item is not in the actionlist, -1 is returned.

`action (text)`

Return the action with specified text.

First, a normal action is tried. If none is found, a separator is tried.

See also `item()`.

`item (text)`

Return the item with specified text.

For a normal action or a separator, an action is returned. For a menu action, a menu is returned.

`nextitem (text)`

Returns the name of the next item.

This can be used to replace the current item with another menu. If the item is the last, None is returned.

`removeItem (item)`

Remove an item from this menu.

`insert_sep (before=None)`

Create and insert a separator

`insert_menu (menu, before=None)`

Insert an existing menu.

`insert_action (action, before=None)`

Insert an action.

`create_insert_action (name, val, before=None)`

Create and insert an action.

`insertItems (items, before=None, debug=False)`

Insert a list of items in the menu.

Parameters:

- *items*: a list of menuitem tuples. Each item is a tuple of two or three elements: (text, action, options):
 - *text*: the text that will be displayed in the menu item. It is stored in a normalized way: all lower case and with ‘&’ removed.
 - *action*: can be any of the following:
 - * a Python function or instance method : it will be called when the item is selected,
 - * a string with the name of a function/method,
 - * a list of Menu Items: a popup Menu will be created that will appear when the item is selected,
 - * an existing Menu,
 - * None : this will create a separator item with no action.
 - *options*: optional dictionary with following honoured fields:
 - * *icon*: the name of an icon to be displayed with the item text. This name should be that of one of the icons in the pyFormex configured icon dirs.
 - * *shortcut*: is an optional key combination to select the item.
 - * *tooltip*: a text that is displayed as popup help.
- *before*: if specified, should be the text *or* the action of one of the items in the Menu (not the items list!): the new list of items will be inserted before the specified item.

class `gui.menu.Menu` (*title='UserMenu', parent=None, before=None, tearoff=False, items=None*)
 A popup/pulldown menu.

class `gui.menu.MenuBar` (*title='TopMenuBar'*)
 A menu bar allowing easy menu creation.

class `gui.menu.DAction` (*name, icon=None, data=None, signal=None*)
 A DAction is a QAction that emits a signal with a string parameter.

When triggered, this action sends a signal (default ‘CLICKED’) with a custom string as parameter. The connected slot can then act depending on this parameter.

class `gui.menu.ActionList` (*actions=[], function=None, menu=None, toolbar=None, icons=None, text=None*)
 Menu and toolbar with named actions.

An action list is a list of strings, each connected to some action. The actions can be presented in a menu and/or a toolbar. On activating one of the menu or toolbar buttons, a given signal is emitted with the button string as parameter. A fixed function can be connected to this signal to act dependent on the string value.

add (*name, icon=None, text=None*)
 Add a new name to the actions list and create a matching DAction.

If the actions list has an associated menu or toolbar, a matching button will be inserted in each of these. If an icon is specified, it will be used on the menu and toolbar. The icon is either a filename or a QIcon object. If text is specified, it is displayed instead of the action’s name.

remove (*name*)
 Remove an action by name

removeAll ()
 Remove all actions from self

names ()
 Return an ordered list of names of the action items.

toolbar (*name*)

Create a new toolbar corresponding to the menu.

6.3.3 gui.colorscales — Mapping numerical values into colors.

This module contains some definitions useful in the mapping of numerical values into colors. This is typically used to provide a visual representation of numerical values (e.g. a temperature plot). See the 'postproc' plugin for some applications in the representation of results from Finite Element simulation programs.

- **ColorScale**: maps scalar float values into colors.
- **ColorLegend**: subdivides a ColorScale into a number of subranges (which are graphically represented by the `pyformex.opengl.decores.ColorLegend` class).
- **Palette**: a dict with some predefined palettes that can be used to create ColorScale instances. The values in the dict are tuples of three colors, the middle one possibly being None (see ColorScale initialization for more details). The keys are strings that can be used in the ColorScale initialization instead of the corresponding value. Currently, the following palettes are defined: 'RAINBOW', 'IRAINBOW', 'HOT', 'RGB', 'BGR', 'RWB', 'BWR', 'RWG', 'GWR', 'GWB', 'BWG', 'BW', 'WB', 'PLASMA', 'VIRIDIS', 'INFERNO', 'MAGMA'.

Classes defined in module gui.colorscales

class `gui.colorscales.ColorScale` (*palet='RAINBOW', minval=0.0, maxval=1.0, midval=None, exp=1.0, exp2=None*)

Mapping floating point values into colors.

The ColorScale maps a range of float values `minval..maxval` or `minval..midval..maxval` into the corresponding color from the specified palette.

Parameters:

- *palet*: the color palette to be used. It is either a string or a tuple of three colors. If a string, it should be one of the keys of the `colorscales.Palette` dict (see above). The full list of available strings can be seen in the *ColorScale* example. If a tuple of three colors, the middle one may be specified as None, in which case it will be set to the mean value of the two other colors. Each color is a tuple of 3 float values, corresponding to the RGB components of the color. Although OpenGL RGB values are limited to the range 0.0 to 1.0, it is perfectly legal to specify color component values outside this range here. OpenGL will however clip the resulting colors to the 0..1 range. This feature can effectively be used to construct color ranges displaying a wider variation of colors. For example, the built in 'RAINBOW' palette has a value `((-2., 0., 2.), (0., 2., 0.), (2., 0., -2.))`. After clipping these will correspond to the colors blue, yellow, red respectively.
- *minval*: float: the minimum value of the scalar range. This value and all lower values will be mapped to the first color of the palette.
- *maxval*: float: the maximum value of the scalar range. This value and all higher values will be mapped to the last (third) color of the palette.
- *midval*: float: a value in the scalar range that will correspond to the middle color of the palette. It defaults to the mean value of *minval* and *maxval*. It can be specified to allow unequal scaling of both subranges of the scalar values. This is often used to set a middle value 0.0 when the values can have both negative and positive values but with rather different maximum values in both directions.
- *exp, exp2*: float: exponent to allow non-linear mapping. The defaults provide a linear mapping between numerical values and colors, or rather a bilinear mapping if the (midval, middle color) is not a linear combination of the endpoint mappings. Still, there are cases where the user wants a nonlinear mapping, e.g. to have more visual accuracy in the higher or lower (absolute) values of the (sub)range(s). Therefore, the values are first linearly scaled to the -1..1 range, and then mapped through the nonlinear function

`arraytools.stuur()`. The effect is that with both `exp > 1.0`, more colors are used in the neighbourhood of the lowest value, while with `exp < 1.0`, more colors are use around the highest value. When both `exp` and `exp2`, the first one holds for the upper halfrange, the second for the lower one. Setting both values `> 1.0` thus has the effect of using more colors around the *midval*.

See example: `ColorScale`

scale (*val*)

Scale a value to the range -1..1.

Parameters:

- *val*: float: numerical value to be scaled.

If the `ColorScale` has only one exponent, values in the range `self.minval..self.maxval` are scaled to the range -1..+1.

If two exponents were specified, scaling is done independently in the intervals `minval..midval` and `midval..maxval`, mapped resp. using `exp2` and `exp` onto the intervals -1..0 and 0..1.

color (*val*)

Return the color representing a value *val*.

Parameters:

- *val*: float: numerical value to be scaled.

The returned color is a tuple of three float RGB values. Values may be out of the range 0..1 if any of the palette defining colors is.

The resulting color is obtained by first scaling the value to the -1..1 range using the *scale* method, and then using that result to linearly interpolate between the color values of the palette.

class `gui.colorscales.ColorLegend` (*colorscale*, *n*)

A `ColorLegend` divides a `ColorScale` in a number of subranges.

Parameters:

- *colorscale*: a `ColorScale` instance
- *n*: a positive integer

For a `ColorScale` without `midval`, the full range is divided in *n* subranges; for a scale with `midval`, each of the two ranges is divided in *n*/2 subranges. In each case the legend has *n* subranges limited by *n*+1 values. The *n* colors of the legend correspond to the middle value of each subrange.

See also `opengl.decors.ColorLegend`.

overflow (*oflow=None*)

Raise a runtime error if *oflow* is `None`, else return *oflow*.

color (*val*)

Return the color representing a value *val*.

The color is that of the subrange holding the value. If the value matches a subrange limit, the lower range color is returned. If the value falls outside the `colorscale` range, a runtime error is raised, unless the corresponding `underflowcolor` or `overflowcolor` attribute has been set, in which case this attribute is returned. Though these attributes can be set to any not `None` value, it will usually be set to some color value, that will be used to show overflow values. The returned color is a tuple of three RGB values in the range 0-1.

6.3.4 `gui.viewport` — Interactive OpenGL Canvas embedded in a Qt4 widget.

This module implements user interaction with the OpenGL canvas defined in module `canvas`. `QtCanvas` is a single interactive OpenGL canvas, while `MultiCanvas` implements a dynamic array of multiple canvases.

Classes defined in module `gui.viewport`

class `gui.viewport.CursorShapeHandler` (*widget*)
A class for handling the mouse cursor shape on the Canvas.

setCursorShape (*shape*)
Set the cursor shape to *shape*

setCursorShapeFromFunc (*func*)
Set the cursor shape to *shape*

class `gui.viewport.CanvasMouseHandler`
A class for handling the mouse events on the Canvas.

getMouseFunc ()
Return the mouse function bound to `self.button` and `self.mod`

class `gui.viewport.QtCanvas` (**args, **kargs*)
A canvas for OpenGL rendering.

This class provides interactive functionality for the OpenGL canvas provided by the `canvas.Canvas` class.

Interactivity is highly dependent on Qt4. Putting the interactive functions in a separate class makes it easier to use the Canvas class in non-interactive situations or combining it with other GUI toolsets.

The `QtCanvas` constructor may have positional and keyword arguments. The positional arguments are passed to the `QtOpenGL.QGLWidget` constructor, while the keyword arguments are passed to the `canvas.Canvas` constructor.

getSize ()
Return the size of this canvas

saneSize (*width=-1, height=-1*)
Return a cleverly resized canvas size.

Computes a new size for the canvas, while trying to keep its current aspect ratio. Specified positive values are returned unchanged.

Parameters

- **width** (*int*) – Requested width of the canvas. If ≤ 0 , it is automatically computed from height and canvas aspect ratio, or set equal to canvas width.
- **height** (*int*) – Requested height of the canvas. If ≤ 0 , it is automatically computed from width and canvas aspect ratio, or set equal to canvas height.

Returns

- **width** (*int*) – Adjusted canvas width.
- **height** (*int*) – Adjusted canvas height.

changeSize (*width, height*)
Resize the canvas to (width x height).

If a negative value is given for either width or height, the corresponding size is set equal to the maximum visible size (the size of the central widget of the main window).

Note that this may not have the expected result when multiple viewports are used.

image (*w=-1, h=-1, remove_alpha=True*)

Return the current OpenGL rendering in an image format.

Parameters

- **w** (*int*) – Requested width of the image (in pixels). If ≤ 0 , automatically computed from height and canvas aspect ratio, or set equal to canvas width.
- **h** (*int*) – Requested height of the image (in pixels). If ≤ 0 , automatically computed from width and canvas aspect ratio, or set equal to canvas height.
- **remove_alpha** (*bool*) – If True (default), the alpha channel is removed from the image.

Returns **qim** (*QImage*) – The current OpenGL rendering as a QImage of the specified size.

Notes

The returned image can be written directly to an image file with `qim.save(filename)`.

See also:

[`rgb\(\)`](#) returns the canvas rendering as a numpy ndarray

rgb (*w=-1, h=-1, remove_alpha=True*)

Return the current OpenGL rendering in an array format.

Parameters

- **w** (*int*) – Requested width of the image (in pixels). If ≤ 0 , automatically computed from height and canvas aspect ratio, or set equal to canvas width.
- **h** (*int*) – Requested height of the image (in pixels). If ≤ 0 , automatically computed from width and canvas aspect ratio, or set equal to canvas height.
- **remove_alpha** (*bool*) – If True (default), the alpha channel is removed from the image.

Returns **ar** (*array*) – The current OpenGL rendering as a numpy array of type uint. Its shape is (w,h,3) if `remove_alpha` is True (default) or (w,h,4) if `remove_alpha` is False.

See also:

[`image\(\)`](#) return the current rendering as an image

outline (*size=(0, 0), profile='luminance', level=0.5, bgcolor=None, nproc=None*)

Return the outline of the current rendering

Parameters:

- *size*: a tuple of ints (w,h) specifying the size of the image to be used in outline detection. A non-positive value will be set automatically from the current canvas size or aspect ratio.
- *profile*: the function used to translate pixel colors into a single value. The default is to use the luminance of the pixel color.
- *level*: isolevel at which to construct the outline.
- *bgcolor*: a color that is to be interpreted as background color and will get a pixel value -0.5.
- *nproc*: number of processors to be used in the image processing. Default is to use as many as available.

Returns the outline as a Formex of plexitude 2.

Note:

- ‘luminance’ is currently the only profile implemented.
- *bgcolor* is currently experimental.

setCursorShape (*shape*)

Set the cursor shape to shape

setCursorShapeFromFunc (*func*)

Set the cursor shape to shape

getMouseFunc ()

Return the mouse function bound to self.button and self.mod

mouse_rectangle (*x, y, action*)

Process mouse events during interactive rectangle zooming.

On PRESS, record the mouse position. On MOVE, create a rectangular zoom window. On RELEASE, zoom to the picked rectangle.

start_selection (*mode, filter, pickable=None*)

Start an interactive picking mode.

If selection mode was already started, mode is disregarded and this can be used to change the filter method.

wait_selection ()

Wait for the user to interactively make a selection.

finish_selection ()

End an interactive picking mode.

accept_selection (*clear=False*)

Accept or cancel an interactive picking mode.

If clear == True, the current selection is cleared.

cancel_selection ()

Cancel an interactive picking mode and clear the selection.

pick (*mode='actor', oneshot=False, func=None, filter=None, pickable=None, _rect=None*)

Interactively pick objects from the viewport.

- *mode*: defines what to pick: one of ['actor', 'element', 'point', 'number', 'edge']
- *oneshot*: if True, the function returns as soon as the user ends a picking operation. The default is to let the user modify his selection and only to return after an explicit cancel (ESC or right mouse button).
- *func*: if specified, this function will be called after each atomic pick operation. The Collection with the currently selected objects is passed as an argument. This can e.g. be used to highlight the selected objects during picking.
- *filter*: defines what elements to retain from the selection: one of [None, 'single', 'closest', 'connected'].
 - None (default) will return the complete selection.
 - ‘closest’ will only keep the element closest to the user.
 - ‘connected’ will only keep elements connected to
 - * the closest element (set picked)
 - * what is already in the selection (add picked).

Currently this only works when picking mode is 'element' and for Actors having a partitionBy-Connection method.

Returns a (possibly empty) Collection with the picked items. After return, the value of the `pf.canvas.selection_accepted` variable can be tested to find how the picking operation was exited: True means accepted (right mouse click, ENTER key, or OK button), False means canceled (ESC key, or Cancel button). In the latter case, the returned Collection is always empty.

pickNumbers (**args, **kargs*)

Go into number picking mode and return the selection.

mouse_rect_pick_events (*rect=None*)

Create the events for a mouse rectangle pick.

Parameters *rect* (*tuple of ints, optional*) – A tuple (x0,y0,x1,y1) specifying the top left corner and the bottom right corner of the rectangular area to be picked. Values are in pixels relative to the canvas widget. If not provided, the whole canvas area will be picked.

Returns

list – A nested list of events. The list contains two sublists. The first holds the events to make the rectangle pick:

- Press the left button mouse at (x0,y0).
- Move the mouse while holding the left button pressed to (x1,y1).
- Release the left mouse button at (x1,y1).

The second sublist holds the events to accept the picked area:

- Press the right mouse button at (x1,y1).
- Release the right mouse button at (x1,y1).

idraw (*mode='point', npoints=-1, zplane=0.0, func=None, coords=None, preview=False*)

Interactively draw on the canvas.

This function allows the user to interactively create points in 3D space and collects the subsequent points in a Coords object. The interpretation of these points is left to the caller.

- *mode*: one of the drawing modes, specifying the kind of objects you want to draw. This is passed to the specified *func*.
- *npoints*: If -1, the user can create any number of points. When ≥ 0 , the function will return when the total number of points in the collection reaches the specified value.
- *zplane*: the depth of the z-plane on which the 2D drawing is done.
- *func*: a function that is called after each atomic drawing operation. It is typically used to draw a preview using the current set of points. The function is passed the current Coords and the *mode* as arguments.
- *coords*: an initial set of coordinates to which the newly created points should be added. If specified, *npoints* also counts these initial points.
- *preview*: **Experimental** If True, the preview function will also be called during mouse movement with a pressed button, allowing to preview the result before a point is created.

The drawing operation is finished when the number of requested points has been reached, or when the user clicks the right mouse button or hits 'ENTER'. The return value is a (n,3) shaped Coords array. To know in which way the drawing was finished check `pf.canvas.draw_accepted`: True means mouse right click / ENTER, False means ESC button on keyboard.

start_draw (*mode, zplane, coords*)
Start an interactive drawing mode.

finish_draw ()
End an interactive drawing mode.

accept_draw (*clear=False*)
Cancel an interactive drawing mode.

If `clear == True`, the current drawing is cleared.

cancel_draw ()
Cancel an interactive drawing mode and clear the drawing.

mouse_draw (*x, y, action*)
Process mouse events during interactive drawing.

On PRESS, do nothing. On MOVE, do nothing. On RELEASE, add the point to the point list.

drawLinesInter (*mode='line', oneshot=False, func=None*)
Interactively draw lines on the canvas.

- `oneshot`: if True, the function returns as soon as the user ends a drawing operation. The default is to let the user draw multiple lines and only to return after an explicit cancel (ESC or right mouse button).
- `func`: if specified, this function will be called after each atomic drawing operation. The current drawing is passed as an argument. This can e.g. be used to show the drawing.

When the drawing operation is finished, the drawing is returned. The return value is a (n,2,2) shaped array.

start_drawing (*mode*)
Start an interactive line drawing mode.

wait_drawing ()
Wait for the user to interactively draw a line.

finish_drawing ()
End an interactive drawing mode.

accept_drawing (*clear=False*)
Cancel an interactive drawing mode.

If `clear == True`, the current drawing is cleared.

cancel_drawing ()
Cancel an interactive drawing mode and clear the drawing.

edit_drawing (*mode*)
Edit an interactive drawing.

dynarot (*x, y, action*)
Perform dynamic rotation operation.

This function processes mouse button events controlling a dynamic rotation operation. The action is one of PRESS, MOVE or RELEASE.

dynapan (*x, y, action*)
Perform dynamic pan operation.

This function processes mouse button events controlling a dynamic pan operation. The action is one of PRESS, MOVE or RELEASE.

dynazoom (*x, y, action*)
Perform dynamic zoom operation.

This function processes mouse button events controlling a dynamic zoom operation. The action is one of PRESS, MOVE or RELEASE.

wheel_zoom (*delta*)

Zoom by rotating a wheel over an angle delta

emit_done (*x, y, action*)

Emit a DONE event by clicking the mouse.

This is equivalent to pressing the ENTER button.

emit_cancel (*x, y, action*)

Emit a CANCEL event by clicking the mouse.

This is equivalent to pressing the ESC button.

draw_state_rect (*x, y*)

Store the pos and draw a rectangle to it.

mouse_pick (*x, y, action*)

Process mouse events during interactive picking.

On PRESS, record the mouse position. On MOVE, create a rectangular picking window. On RELEASE, pick the objects inside the rectangle.

draw_state_line (*x, y*)

Store the pos and draw a line to it.

mouse_draw_line (*x, y, action*)

Process mouse events during interactive drawing.

On PRESS, record the mouse position. On MOVE, draw a line. On RELEASE, add the line to the drawing.

mousePressEvent (*e*)

Process a mouse press event.

mouseMoveEvent (*e*)

Process a mouse move event.

mouseReleaseEvent (*e*)

Process a mouse release event.

wheelEvent (*e*)

Process a wheel event.

class `gui.viewport.MultiCanvas` (*parent=None*)

An OpenGL canvas with multiple viewports and QT interaction.

The MultiCanvas implements a central QT widget containing one or more QtCanvas widgets.

changeLayout (*nvps=None, ncols=None, nrows=None, pos=None, rstretch=None, cstretch=None*)

Change the lay-out of the viewports on the OpenGL widget.

nvps: number of viewports *ncols*: number of columns *nrows*: number of rows *pos*: list holding the position and span of each viewport `[[row,col,rowspan,colspan],...]` *rstretch*: list holding the stretch factor for each row *cstretch*: list holding the stretch factor for each column (rows/columns with a higher stretch factor take more of the available space) Each of this parameters is optional.

If a number of viewports is given, viewports will be added or removed to match the requested number. By default they are laid out rowwise over two columns.

If *ncols* is an int, viewports are laid out rowwise over *ncols* columns and *nrows* is ignored. If *ncols* is None and *nrows* is an int, viewports are laid out columnwise over *nrows* rows. Alternatively, the *pos* argument can be used to specify the layout of the viewports.

newView (*shared=True, settings=None*)

Create a new viewport.

If *shared* is *True*, and the *MultiCanvas* already has one or more viewports, the new viewport will share display lists and textures with the first viewport. Since *pyFormex* is not using display lists (anymore) and textures are needed to display text, the value defaults to *True*, and all viewports will share the same textures, unless a viewport is created with a specified value for *shared*: it can either be another viewport to share textures with, or a value *False* or *None* to not share textures with any viewport. In the latter case you will not be able to use text display, unless you initialize the textures yourself.

settings: can be a legal *CanvasSettings* to initialize the viewport. Default is to copy settings of the current viewport.

Returns the created viewport, which is an instance of *QtCanvas*.

addView ()

Add a new viewport to the widget

setCurrent (*canv*)

Make the specified viewport the current one.

canv can be either a viewport or viewport number.

viewIndex (*view*)

Return the index of the specified view

currentView ()

Return the index of the current view

showWidget (*w*)

Show the view *w*.

removeView ()

Remove the last view

link (*vp, to*)

Link viewport *vp* to *to*

config ()

Return the full configuration needed to restore this *MultiCanvas*.

Currently only works on single viewport.

save (*filename*)

Save the canvas settings to file

Currently only works on single viewport.

loadConfig (*config*)

Reset the viewports size, layout and cameras from a *Config* dict

load (*filename*)

Load the canvas settings from file

Functions defined in module *gui.viewport*

gui.viewport.**dotpr** (*v, w*)

Return the dot product of vectors *v* and *w*

gui.viewport.**length** (*v*)

Return the length of the vector *v*

`gui.viewport.projection(v, w)`

Return the (signed) length of the projection of vector *v* on vector *w*.

`gui.viewport.setOpenGLFormat()`

Set the correct OpenGL format.

On a correctly installed system, the default should do well. The default OpenGL format can be changed by command line options:

```
--dri    : use the Direct Rendering Infrastructure, if available
--nodri  : do not use the DRI
--opengl : set the opengl version
--(no)multisample
```

`gui.viewport.OpenGLFormat(fmt=None)`

Report information about the OpenGL format.

`gui.viewport.OpenGLSupportedVersions(flags)`

Return the supported OpenGL version.

flags is the return value of `QGLFormat.OpenGLVersionFlag()`

Returns a list with tuple (*k,v*) where *k* is a string describing an Opengl version and *v* is True or False.

`gui.viewport.OpenGLVersions(fmt=None)`

Report information about the supported OpenGL versions.

`gui.viewport.drawDot(x, y)`

Draw a dot at canvas coordinates (*x,y*).

`gui.viewport.drawLine(x1, y1, x2, y2)`

Draw a straight line from (*x1,y1*) to (*x2,y2*) in canvas coordinates.

`gui.viewport.drawGrid(x1, y1, x2, y2, nx, ny)`

Draw a rectangular grid of lines

The rectangle has (*x1,y1*) and (*x2,y2*) as opposite corners. There are (*nx,ny*) subdivisions along the (*x,y*)-axis. So the grid has (*nx+1*) * (*ny+1*) lines. *nx=ny=1* draws a rectangle. *nx=0* draws 1 vertical line (at *x1*). *nx=-1* draws no vertical lines. *ny=0* draws 1 horizontal line (at *y1*). *ny=-1* draws no horizontal lines.

`gui.viewport.drawRect(x1, y1, x2, y2)`

Draw the circumference of a rectangle.

6.3.5 `gui.image` — Saving OpenGL renderings to image files.

This module defines some functions that can be used to save the OpenGL rendering and the pyFormex GUI to image files. There are even provisions for automatic saving to a series of files and creating a movie from these images.

Functions defined in module `gui.image`

`gui.image.imageFormats(mode='w')`

Return a list of the valid image formats.

image formats are lower case strings as 'png', 'gif', 'ppm', 'eps', etc. The available image formats are derived from the installed software.

Parameters:

- *mode*: 'w' or 'r', for formats that can be written or read.

`gui.image.initialize()`
Initialize the image module.

`gui.image.checkImageFormat(fmt, verbose=True)`
Checks image format; if verbose, warn if it is not.

Returns the image format, or None if it is not OK.

`gui.image.imageFormatFromExt(ext)`
Determine the image format from an extension.

The extension may or may not have an initial dot and may be in upper or lower case. The format is equal to the extension characters in lower case. If the supplied extension is empty, the default format 'png' is returned.

`gui.image.save_canvas(canvas, fn, fmt=None, quality=-1, size=None, alpha=False)`
Save the rendering on canvas as an image file.

canvas specifies the qtcanvas rendering window. fn is the name of the file fmt is the image file format. The default is taken from the filename.

`gui.image.save_window(filename, format, quality=-1, windowname=None, crop=None)`
Save a window as an image file.

This function needs a filename AND format. If a window is specified, the named window is saved. Else, the main pyFormex window is saved.

If crop is given, the specified rectangle is cropped. If crop='canvas', windowname is set to main pyFormex window and crop to canvas rectangle.

In all cases, the GUI and current canvas are raised.

`gui.image.save_window_rect(filename, format, quality=-1, window='root', crop=None)`
Save a rectangular part of the screen to a an image file.

crop: (x,y,w,h)

`gui.image.save(filename=None, window=False, multi=False, hotkey=True, autosave=False, border=False, grab=False, format=None, quality=-1, size=None, verbose=False, alpha=True, rootcrop=None)`

Saves an image to file or Starts/stops multisave mode.

With a filename and multi==False (default), the current viewport rendering is saved to the named file.

With a filename and multi==True, multisave mode is started. Without a filename, multisave mode is turned off. Two subsequent calls starting multisave mode without an intermediate call to turn it off, do not cause an error. The first multisave mode will implicitly be ended before starting the second.

In multisave mode, each call to saveNext() will save an image to the next generated file name. File-names are generated by incrementing a numeric part of the name. If the supplied filename (after removing the extension) has a trailing numeric part, subsequent images will be numbered continuing from this number. Otherwise a numeric part '-000' will be added to the filename.

If window is True, the full pyFormex window is saved. If window and border are True, the window decorations will be included. If window is False, only the current canvas viewport is saved.

If grab is True, the external 'import' program is used to grab the window from the screen buffers. This is required by the border and window options.

If hotkey is True, a new image will be saved by hitting the 'S' key. If autosave is True, a new image will be saved on each execution of the 'draw' function. If neither hotkey nor autosave are True, images can only be saved by executing the saveNext() function from a script.

If no format is specified, it is derived from the filename extension. fmt should be one of the valid formats as returned by imageFormats()

If `verbose=True`, error/warnings are activated. This is usually done when this function is called from the GUI.

`gui.image.saveImage` (*filename=None, window=False, multi=False, hotkey=True, autosave=False, border=False, grab=False, format=None, quality=-1, size=None, verbose=False, alpha=True, rootcrop=None*)

Saves an image to file or Starts/stops multisave mode.

With a filename and `multi=False` (default), the current viewport rendering is saved to the named file.

With a filename and `multi=True`, multisave mode is started. Without a filename, multisave mode is turned off. Two subsequent calls starting multisave mode without an intermediate call to turn it off, do not cause an error. The first multisave mode will implicitly be ended before starting the second.

In multisave mode, each call to `saveNext()` will save an image to the next generated file name. File-names are generated by incrementing a numeric part of the name. If the supplied filename (after removing the extension) has a trailing numeric part, subsequent images will be numbered continuing from this number. Otherwise a numeric part '-000' will be added to the filename.

If `window` is `True`, the full pyFormex window is saved. If `window` and `border` are `True`, the window decorations will be included. If `window` is `False`, only the current canvas viewport is saved.

If `grab` is `True`, the external 'import' program is used to grab the window from the screen buffers. This is required by the `border` and `window` options.

If `hotkey` is `True`, a new image will be saved by hitting the 'S' key. If `autosave` is `True`, a new image will be saved on each execution of the 'draw' function. If neither `hotkey` nor `autosave` are `True`, images can only be saved by executing the `saveNext()` function from a script.

If no format is specified, it is derived from the filename extension. `fmt` should be one of the valid formats as returned by `imageFormats()`

If `verbose=True`, error/warnings are activated. This is usually done when this function is called from the GUI.

`gui.image.saveNext` ()

In multisave mode, saves the next image.

This is a quiet function that does nothing if multisave was not activated. It can thus safely be called on regular places in scripts where one would like to have a saved image and then either activate the multisave mode or not.

`gui.image.changeBackgroundColorXPM` (*fn, color*)

Changes the background color of an .xpm image.

This changes the background color of an .xpm image to the given value. `fn` is the filename of an .xpm image. `color` is a string with the new background color, e.g. in web format ('#FFF' or '#FFFFFF' is white). A special value 'None' may be used to set a transparent background. The current background color is selected from the lower left pixel.

`gui.image.saveIcon` (*fn, size=32, transparent=True*)

Save the current rendering as an icon.

`gui.image.autoSaveOn` ()

Returns `True` if `autosave` multisave mode is currently on.

Use this function instead of directly accessing the `autosave` variable.

`gui.image.createMovie` (*files, encoder='convert', outfn='output', **kargs*)

Create a movie from a saved sequence of images.

Parameters:

- *files*: a list of filenames, or a string with one or more filenames separated by whitespace. The filenames can also contain wildcards interpreted by the shell.
- *encoder*: string: the external program to be used to create the movie. This will also define the type of output file, and the extra parameters that can be passed. The external program has to be installed on the computer. The default is *convert*, which will create animated gif. Other possible values are ‘mencoder’ and ‘ffmpeg’, creating meg4 encode movies from jpeg input files.
- *outfn*: string: output file name (not including the extension). Default is output.

Other parameters may be passed and may be needed, depending on the converter program used. Thus, for the default ‘convert’ program, each extra keyword parameter will be translated to an option ‘-keyword value’ for the command.

Example:

```
createMovie('images*.png', delay=1, colors=256)
```

will create an animated gif ‘output.gif’.

`gui.image.saveMovie` (*filename, format, windowname=None*)
Create a movie from the pyFormex window.

6.3.6 `gui.imageView` — A general image viewer

Part of this code was borrowed from the TrollTech Qt examples.

```
class gui.imageView.ImageView (parent=None, path=None)
    pyFormex image viewer
```

The pyFormex image viewer was shaped after the Image Viewer from the TrollTech Qt documentation.

It can be use as stand alone application, as well as from inside pyFormex. The viewer allows browsing through directories, selecting an image to be displayed. The image can be resized to fit a window.

Parameters:

- *parent*: The parent Qt4 widget (the Qt4 app in the stand alone case).
- *path*: string: the full path to the image to be initially displayed.

6.3.7 `gui.appMenu` — Menu with pyFormex apps.

Classes defined in module `gui.appMenu`

```
class gui.appMenu.AppMenu (title, dir=None, files=None, mode='app', ext=None, recursive=None,  
                           max=0, autoplay=False, toplevel=True, parent=None, before=None,  
                           runall=True)
```

A menu of pyFormex applications in a directory or list.

This class creates a menu of pyFormex applications or scripts collected from a directory or specified as a list of modules. It is used in the pyFormex GUI to create the examples menu, and for the apps history. The pyFormex apps can then be run from the menu or from the button toolbar. The user may use this class to add his own apps/scripts into the pyFormex GUI.

Apps are simply Python modules that have a ‘run’ function. Only these modules will be added to the menu. Only files that are recognized by `utils.is_pyFormex()` as being pyFormex scripts will be added to the menu.

The constructor takes the following arguments:

- *title*: the top level label for the menu
- *dir*: an optional directory path. If specified, and no *files* argument is specified, all Python files in *dir* that do not start with either '.' or '_', will be considered for inclusion in the menu. If mode=='app', they will only be included if they can be loaded as a module. If mode=='script', they will only be included if they are considered a pyFormex script by `utils.is_pyFormex`. If *files* is specified, *dir* will just be prepended to each file in the list.
- *files*: an explicit list of file names of pyFormex scripts. If no *dir* nor *ext* arguments are given, these should be the full path names to the script files. Otherwise, *dir* is prepended and *ext* is appended to each filename.
- *ext*: an extension to be added to each filename. If *dir* was specified, the default extension is '.py'. If no *dir* was specified, the default extension is an empty string.
- *recursive*: if True, a cascading menu of all pyFormex scripts in the directory and below will be constructed. If only *dir* and no *files* are specified, the default is True
- *max*: if specified, the list of files will be truncated to this number of items. Adding more files to the menu will then be done at the top and the surplus number of files will be dropped from the bottom of the list.

The defaults were thus chosen to be convenient for the three most frequent uses of this class:

```
AppMenu('My Apps', dir="/path/to/my/appdir")
```

creates a menu with all pyFormex apps in the specified path and its subdirectories.

```
ApptMenu('My Scripts', dir="/path/to/my/scriptsdir", mode='scripts')
```

creates a menu with all pyFormex scripts in the specified path and its subdirectories.

```
AppMenu('History', files=["/my/script1.py", "/some/other/script.pye"],
↪mode='script', recursive=False)
```

is typically used to create a history menu of previously visited script files.

With the resulting file list, a menu is created. Selecting a menu item will make the corresponding file the current script and unless the *autoplay* configuration variable was set to False, the script is executed.

Furthermore, if the menu is a toplevel one, it will have the following extra options:

- *Classify scripts*
- *Remove catalog*
- *Reload scripts*

The first option uses the keyword specifications in the scripts docstring to make a classification of the scripts according to keywords. See the `scriptKeywords()` function for more info. The second option removes the classification. Both options are especially useful for the pyFormex examples.

The last option reloads a ScriptMenu. This can be used to update the menu when you created a new script file.

getFiles()

Get a list of scripts in self.dir

filterFiles(files)

Filter a list of scripts

loadFiles(files=None)

Load the app/script files in this menu

fileName(script)

Return the full pathname for a script.

fullAppName (*app*)

Return the pkg.module name for an app.

run (*action*)

Run the selected app.

This function is executed when the menu item is selected.

runApp (*app, play=True*)

Set/Run the specified app.

Set the specified app as the current app, and run it if play==True.

runAll (*startfrom='A', stopat='I', count=-1, recursive=True, random=False*)

Run all apps with a name in the range [startfrom,stopat].

Runs the apps with a name \geq *startfrom* and $<$ *stopat*. The default will run all apps starting with a capital (like the examples). Specify None to disable the limit. If count is positive, at most count scripts are executed. If recursive is True, also the files in submenu are played. If random is True, the files in any submenu are shuffled before running.

runAllNext (*offset=1, count=-1*)

Run a sequence of apps, starting with the current plus offset.

If a positive count is specified, at most count scripts will be run. A nonzero offset may be specified to not start with the current script.

runCurrent ()

Run the current app, or the first if none was played yet.

runNextApp ()

Run the next app, or the first if none was played yet.

runRandom ()

Run a random script.

reload ()

Reload the scripts from dir.

This is only available if a directory path was specified and no files.

add (*name, strict=True, skipconfig=True*)

Add a new filename to the front of the menu.

This function is used to add app/scripts to the history menus. By default, only legal pyFormex apps or scripts can be added, and scripts from the user config will not be added. Setting strict and or skipconfig to False will skip the filter(s).

Functions defined in module gui.appMenu

`gui.appMenu.sortSets` (*d*)

Turn the set values in d into sorted lists.

- *d*: a Python dictionary

All the values in the dictionary are checked. Those that are of type *set* are converted to a sorted list.

`gui.appMenu.classify` (*appdir, pkg, nmax=0*)

Classify the files in submenus according to keywords.

`gui.appMenu.splitAlpha` (*strings, n, ignorecase=True*)

Split a series of strings in alphabetic collections.

The strings are split over a series of bins in alphabetical order. Each bin can contain strings starting with multiple successive characters, but not more than *n* items. Items starting with the same character are always in the same bin. If any starting character occurs more than *n* times, the maximum will be exceeded.

- *files*: a list of strings start with an upper case letter ('A'-'Z')
- *n*: the desired maximum number of items in a bin.

Returns: a tuple of

- *labels*: a list of strings specifying the range of start characters (or the single start character) for the bins
- *groups*: a list with the contents of the bins. Each item is a list of sorted strings starting with one of the characters in the corresponding label

`gui.appMenu.createAppMenu(mode='app', parent=None, before=None)`

Create the menu(s) with pyFormex apps

This creates a menu with all examples distributed with pyFormex. By default, this menu is put in the top menu bar with menu label 'Examples'.

The user can add his own app directories through the configuration settings. In that case the 'Examples' menu and menus for all the configured app paths will be gathered in a top level popup menu labeled 'Apps'.

The menu will be placed in the top menu bar before the specified item. If a menu item named 'Examples' or 'Apps' already exists, it is replaced.

`gui.appMenu.reloadMenu(mode='app')`

Reload the named menu.

6.3.8 `gui.toolbar` — Toolbars for the pyFormex GUI.

This module defines the functions for creating the pyFormex window toolbars.

Functions defined in module `gui.toolbar`

`gui.toolbar.addButton(toolbar, tooltip, icon, func, repeat=False, toggle=False, checked=False, icon0=None, enabled=True)`

Add a button to a toolbar.

- *toolbar*: the toolbar where the button will be added
- *tooltip*: the text to appear as tooltip
- *icon*: name of the icon to be displayed on the button,
- *func*: function to be called when the button is pressed,
- *repeat*: if True, the *func* will repeatedly be called if button is held down.
- *toggle*: if True, the button is a toggle and stays in depressed state until pressed again.
- *checked*: initial state for a toggle button.
- *icon1*: for a toggle button, icon to display when button is not checked.

`gui.toolbar.removeButton(toolbar, button)`

Remove a button from a toolbar.

`gui.toolbar.addActionButtons(toolbar)`

Add the script action buttons to the toolbar.

`gui.toolbar.addCameraButtons (toolbar)`
 Add the camera buttons to a toolbar.

`gui.toolbar.toggleButton (attr, state=None)`
 Update the corresponding viewport attribute.
 This does not update the button state.

`gui.toolbar.updateButton (button, attr)`
 Update the button to correct state.

`gui.toolbar.updateWireButton ()`
 Update the wire button to correct state.

`gui.toolbar.updateTransparencyButton ()`
 Update the transparency button to correct state.

`gui.toolbar.updateLightButton ()`
 Update the light button to correct state.

`gui.toolbar.updateNormalsButton (state=True)`
 Update the normals button to correct state.

`gui.toolbar.updatePerspectiveButton ()`
 Update the normals button to correct state.

`gui.toolbar.addTimeoutButton (toolbar)`
 Add or remove the timeout button, depending on cfg.

`gui.toolbar.timeout (onoff=None)`
 Programmatically toggle the timeout button

6.4 pyFormex plugins

Plugin modules extend the basic pyFormex functions to variety of specific applications. Apart from being located under the `pyformex/plugins` path, these modules are in no way different from other pyFormex modules.

6.4.1 `plugins.bifmesh` — Vascular Sweeping Mesher

Functions defined in module `plugins.bifmesh`

`plugins.bifmesh.structuredQuadMeshGrid (sgx=3, sgy=3, isopquad=None)`
 it returns nodes (2D) and elems of a structured quadrilateral grid. nodes and elements are both ordered first vertically (y) and then orizontally (x). This function is the equivalent of `simple.rectangularGrid` but on the mesh level.

`plugins.bifmesh.structuredHexMeshGrid (dx, dy, dz, isophex='hex64')`
 it builds a structured hexahedral grid with nodes and elements both numbered in a structured way: first along z, then along y, and then along x. The resulting hex cells are oriented along z. This function is the equivalent of `simple.rectangularGrid` but for a mesh. Additionally, dx,dy,dz can be either integers or div (1D list or array). In case of list/array, first and last numbers should be 0.0 and 1.0 if the desired grid has to be inside the region 0.,0.,0. to 1.,1.,1. TODO: include other options to get the control points for other isoparametric transformation for hex.

`plugins.bifmesh.findBisectrixUsingPlanes (cpx, centx)`
 it returns a bisectrix-points at each point of a Polygon (unit vector of the bisectrix). All the bisectrix-points are on the side of centx (inside the Polygon), regardless to the concavity or convexity of the angle, thus avoiding

the problem of collinear or concave segments. The points will point towards the centx if the centx is offplane. It uses the lines from intersection of 2 planes.

`plugins.bifmesh.cpBoundaryLayer` (*BS, centr, issection0=False, bl_rel=0.2*)

it takes n points of a nearly circular section (for the isop transformation, n should be 24, 48 etc) and find the control points needed for the boundary layer. The center of the section has to be given separately. `-issection0` needs to be True only for the section-0 of each branch of a bifurcation, which has to share the control points with the other branches. So it must be False for all other sections and single vessels. This implementation for the bl (separated from the inner lumen) is needed to ensure an optimal mesh quality at the boundary layer in terms of angular skewness, needed for WSS calculation.

`plugins.bifmesh.cpQuarterLumen` (*lumb, centp, edgesq=0.75, diag=0.848528137423857, verbos=False*)

control points for 1 quarter of lumen mapped in quad regions. `lumb` is a set of points on a quarter of section. `centp` is the center of the section. The number of poin I found that `edgesq=0.75, diag=0.6*2**0.5` give the better mapping. Also possible `edgesq=0.4, diag=0.42*2**0.5`. Currently, it is not perfect if the section is not planar.

`plugins.bifmesh.visualizeSubmappingQuadRegion` (*sqr, timewait=None*)

visualilze the control points (-1,16,3) in each submapped region and check the quality of the region (which will be inherited by the mesh crosssectionally)

`plugins.bifmesh.cpOneSection` (*hc, oc=None, isBranchingSection=False, verbos=False*)

`hc` is a numbers of points on the boundary line of 1 almost circular section. `oc` is the center point of the section. It returns 3 groups of control points: for the inner part, for the transitional part and for the boundary layer of one single section

`plugins.bifmesh.cpAllSections` (*HC, OC, start_end_branching=[False, False]*)

control points of all sections divided in 3 groups of control points: for the inner part, for the transitional part and for the boundary layer. if `start_end_branching` is [True,True] the first and the last section are considered bifurcation sections and therefore meshed differently.

`plugins.bifmesh.cpStackQ16toH64` (*cpq16*)

sweeping trick: from sweeping sections longitudinally to mapping hex64: ittakes -1,16,3 (cp of the quad16) and groups them in -1,64,3 (cp of the hex63) but slice after slice: [0,1,2,3],[1,2,3,4],[2,3,4,5],... It is a trick to use the hex64 for sweeping along an arbitrary number of sections.

`plugins.bifmesh.mapHexLong` (*mesh_block, cpvr*)

map a structured mesh (`n_block, e_block, cp_block` are in `mesh_block`) into a volume defined by the control points `cpvr` (# regions longitudinally, # regions in 1 cross sectionsm, 64, 3). `cp_block` are the control points of the mesh block. It returns nodes and elements. Nodes are repeated in subsequently mapped regions ! TRICK: in order to make the mapping working for an arbitrary number of sections the following trick is used: of the whole `mesh_block`, only the part located between the points 1-2 is meshed and mapped between 2 slices only. Thus, the other parts 0-1 and 2-3 are not mapped. To do so, the first and the last slice need to be meshed separately: `n_start` 0-1 and `n_end` 2-3.

`plugins.bifmesh.mapQuadLong` (*mesh_block, cpvr*)

TRICK: in order to make the mapping working for an arbitrary number of sections the following trick is used: of the whole `mesh_block`, only the part located between the points 1-2 is meshed and mapped between 2 slices only. Thus, the other parts 0-1 and 2-3 are not mapped. To do so, the first and the last slice need to be meshed separately: `n_start` 0-1 and `n_end` 2-3.

6.4.2 `plugins.cameratools` — Camera tools

Some extra tools to handle the camera.

Functions defined in module `plugins.cameratools`

`plugins.cameratools.showCameraTool()`
Show the camera settings dialog.

This function pops up a dialog where the user can interactively adjust the current camera settings.

The function can also be called from the `Camera->Settings` menu.

6.4.3 `plugins.ccxdat` —

Functions defined in module `plugins.ccxdat`

`plugins.ccxdat.readDispl(fil, nnodes, nres)`
Read displacements from a Calculix .dat file

`plugins.ccxdat.readStress(fil, nelems, ngp, nres)`
Read stresses from a Calculix .dat file

`plugins.ccxdat.readResults(fn, DB, nnodes, nelems, ngp)`
Read Calculix results file for nnodes, nelems, ngp
Add results to the specified DB

`plugins.ccxdat.createResultDB(model)`
Create a results database for the given FE model

`plugins.ccxdat.addFeResult(DB, step, time, result)`
Add an FeResult for a time step to the result DB
This is currently 2D only

`plugins.ccxdat.computeAveragedNodalStresses(M, data, gprule)`
Compute averaged nodal stresses from GP stresses in 2D quad8 mesh

6.4.4 `plugins.ccxinp` —

Functions defined in module `plugins.ccxinp`

`plugins.ccxinp.abq_eltype(eltype)`
Analyze an Abaqus element type and return eltype characteristics.

Returns a dictionary with:

- `type`: the element base type
- `ndim`: the dimensionality of the element
- `nplex`: the plexitude (number of nodes)
- `mod`: a modifier string

Currently, all these fields are returned as strings. We should probably change `ndim` and `nplex` to an int.

`plugins.ccxinp.pyf_eltype(d)`
Return the best matching pyFormex element type for an abq/ccx element
`d` is an element groupdict obtained by scanning the element name.

`plugins.ccxinp.startPart(name)`
Start a new part.

`plugins.ccxinp.readCommand` (*line*)
 Read a command line, return the command and a dict with options

`plugins.ccxinp.do_HEADING` (*opts, data*)
 Read the nodal data

`plugins.ccxinp.do_PART` (*opts, data*)
 Set the part name

`plugins.ccxinp.do_SYSTEM` (*opts, data*)
 Read the system data

`plugins.ccxinp.do_NODE` (*opts, data*)
 Read the nodal data

`plugins.ccxinp.do_ELEMENT` (*opts, data*)
 Read element data

`plugins.ccxinp.readInput` (*fn*)
 Read an input file (.inp)

Returns an object with the following attributes:

- *heading*: the heading read from the .inp file
- *parts*: a list with parts.

A part is a dict and can contain the following keys:

- *name*: string: the part name
- *coords*: float (nnod,3) array: the nodal coordinates
- *nodid*: int (nnod,) array: node numbers; default is arange(nnod)
- *elems*: int (nelems,nplex) array: element connectivity
- *elid*: int (nelems,) array: element numbers; default is arange(nelems)

6.4.5 `plugins.curve` — Definition of curves in pyFormex.

This module defines classes and functions specialized for handling one-dimensional geometry in pyFormex. These may be straight lines, polylines, higher order curves and collections thereof. In general, the curves are 3D, but special cases may be created for handling plane curves.

Classes defined in module `plugins.curve`

class `plugins.curve.Curve`
 Base class for curve type classes.

This is a virtual class intended to be subclassed. It defines the common definitions for all curve types. The subclasses should at least define the following attributes and methods or override them if the defaults are not suitable.

Attributes:

- *coords*: coordinates of points defining the curve
- *nparts*: number of parts (e.g. straight segments of a polyline)
- *closed*: is the curve closed or not
- *range*: [min,max], range of the parameter: default 0..1

Methods:

- *sub_points(t,j)*: returns points at parameter value t,j
- *sub_directions(t,j)*: returns direction at parameter value t,j
- *pointsOn()*: the defining points placed on the curve
- *pointsOff()*: the defining points placed off the curve (control points)
- *parts(j,k)*:
- *approx(nseg=None,ndiv=None,chordal=None)*:

Furthermore it may define, for efficiency reasons, the following methods:

- *sub_points_2*
- *sub_directions_2*

nelems ()

Return the number of elements in the Geometry.

Returns *int* – The number of elements in the Geometry. This is an abstract method that should be reimplemented by the derived class.

endPoints ()

Return start and end points of the curve.

Returns a Coords with two points, or None if the curve is closed.

sub_points (t,j)

Return the points at values t in part j

t can be an array of parameter values, j is a single segment number.

sub_points_2 (t,j)

Return the points at values,parts given by zip(t,j)

t and j can both be arrays, but should have the same length.

sub_directions (t,j)

Return the directions at values t in part j

t can be an array of parameter values, j is a single segment number.

sub_directions_2 (t,j)

Return the directions at values,parts given by zip(t,j)

t and j can both be arrays, but should have the same length.

localParam (t)

Split global parameter value in part number and local parameter

Parameter values are floating point values. Their integer part is interpreted as the curve segment number, and the decimal part goes from 0 to 1 over the segment.

Returns a tuple of arrays i,t, where i are the (integer) part numbers and t the local parameter values (between 0 and 1).

pointsAt (t, return_position=False)

Return the points at parameter values t.

Parameter values are floating point values. Their integer part is interpreted as the curve segment number, and the decimal part goes from 0 to 1 over the segment.

Returns a Coords with the coordinates of the points.

If `return_position` is `True`, also returns the part numbers on which the point are lying and the local parameter values.

directionsAt (*t*)

Return the directions at parameter values *t*.

Parameter values are floating point values. Their integer part is interpreted as the curve segment number, and the decimal part goes from 0 to 1 over the segment.

subPoints (*div=10, extend=[0.0, 0.0]*)

Return a sequence of points on the Curve.

- *div*: int or a list of floats (usually in the range [0.,1.]) If *div* is an integer, a list of floats is constructed by dividing the range [0.,1.] into *div* equal parts. The list of floats then specifies a set of parameter values for which points at in each part are returned. The points are returned in a single Coords in order of the parts.

The extend parameter allows to extend the curve beyond the endpoints. The normal parameter space of each part is [0.0 .. 1.0]. The extend parameter will add a curve with parameter space [-extend[0] .. 0.0] for the first part, and a curve with parameter space [1.0 .. 1 + extend[0]] for the last part. The parameter step in the extensions will be adjusted slightly so that the specified extension is a multiple of the step size. If the curve is closed, the extend parameter is disregarded.

split (*split=None*)

Split a curve into a list of partial curves

`split` is a list of integer values specifying the node numbers where the curve is to be split. As a convenience, a single int may be given if the curve is to be split at a single node, or `None` to split at all nodes.

Returns a list of open curves of the same type as the original.

length ()

Return the total length of the curve.

This is only available for curves that implement the ‘lengths’ method.

atApproximate (*nseg=None, ndiv=None, equidistant=False, npre=None*)

Return parameter values for approximating a Curve with a PolyLine.

Parameters:

- *nseg*: number of segments of the resulting PolyLine. The number of returned parameter values is *nseg* if the curve is closed, else *nseg+1*. Only used if *ndiv* is not specified.
- *ndiv*: positive integer or a list thereof. If a single integer, it specifies the number of straight segments in each part of the curve, and is thus equivalent with *nseg = ndiv * self.nparts*. If a list of integers, its length should be equal to the number of parts in the curve and each integer in the list specifies the number of segments the corresponding part.
- *equidistant*: bool, only used if *ndiv* is not specified. If `True`, the points are spaced almost equidistantly over the curve. If `False` (default), the points are spread equally over the parameter space.
- *npre*: integer: only used when *equidistant* is `True`: number of segments per part of the curve used in the pre-approximation. This pre-approximation is currently required to compute curve lengths.

Examples

```
>>> PL = PolyLine([[0,0,0], [1,0,0], [1,1,0]])
>>> print(PL.atApproximate(nseg=6))
[ 0.    0.33  0.67  1.    1.33  1.67  2. ]
```

(continues on next page)

(continued from previous page)

```

>>> print (PL.atApproximate (ndiv=3))
[ 0.    0.33  0.67  1.    1.33  1.67  2. ]
>>> print (PL.atApproximate (ndiv=(2,4)))
[ 0.    0.5   1.    1.25  1.5   1.75  2. ]
>>> print (PL.atApproximate (ndiv=(2,)))
[ 0.    0.5   1. ]

```

atChordal (*chordal*, *at=None*)

Return parameter values to approximate within given chordal error.

Parameters:

- *chordal*: relative tolerance on the distance of the chord to the curve. The tolerance is relative to the curve's `charLength()`.
- *at*: list of floats: list of parameter values that need to be included in the result. The list should contain increasing values in the curve's parameter range. If not specified, a default is set assuring that the curve is properly approximated. If you specify this yourself, you may end up with bad approximations due to bad choice of the initial values.

Returns a list of parameter values that create a PolyLine approximate for the curve, such that the chordal error is everywhere smaller than the give value. The chordal error is defined as the distance from a point of the curve to the chord.

approxAt (*at*)

Create a PolyLine approximation with specified parameter values.

Parameters:

- *at*: a list of parameter values in the curve parameter range. The Curve points at these parameter values are connected with straight segments to form a PolyLine approximation.

approx (*nseg=None*, *ndiv=None*, *chordal=0.02*, *equidistant=False*, *npre=None*)

Approximate a Curve with a PolyLine of n segments

If neither *nseg* nor *ndiv* are specified (default), a chordal method is used limiting the chordal distance of the curve to the PolyLine segments.

Parameters:

- *nseg*: integer: number of straight segments of the resulting PolyLine. Only used if *ndiv* is not specified.
- *ndiv*: positive integer or a list thereof. If a single integer, it specifies the number of straight segments in each part of the curve, and is thus equivalent with $nseg = ndiv * self.nparts$. If a list of integers, its length should be equal to the number of parts in the curve and each integer in the list specifies the number of segments the corresponding part.
- *chordal*: float: accuracy of the approximation when using the 'chordal error' method. This is the case if neither *nseg* nor *ndiv* are specified (the default). The value is relative to the curve's `charLength()`.
- *equidistant*: bool, only used if *nseg* is specified and *ndiv* is not. If True the $nseg+1$ points are spaced almost equidistantly over the curve. If False (default), the points are spread equally over the parameter space.
- *npre*: integer, only used if the *chordal* method is used or if *nseg* is not None and *equidistant* is True: the number of segments per part of the curve (like *ndiv*) used in a pre-approximation. If not specified, it is set to the degree of the curve for the *chordal* method (1 for PolyLine), and to 100 in the *equidistant* method (where the pre-approximation is currently used to compute accurate curve lengths).

approximate (*nseg=None, ndiv=None, chordal=0.02, equidistant=False, npre=None*)

Approximate a Curve with a PolyLine of n segments

If neither *nseg* nor *ndiv* are specified (default), a chordal method is used limiting the chordal distance of the curve to the PolyLine segments.

Parameters:

- *nseg*: integer: number of straight segments of the resulting PolyLine. Only used if *ndiv* is not specified.
- *ndiv*: positive integer or a list thereof. If a single integer, it specifies the number of straight segments in each part of the curve, and is thus equivalent with $nseg = ndiv * self.nparts$. If a list of integers, its length should be equal to the number of parts in the curve and each integer in the list specifies the number of segments the corresponding part.
- *chordal*: float: accuracy of the approximation when using the ‘chordal error’ method. This is the case if neither *nseg* nor *ndiv* are specified (the default). The value is relative to the curve’s `charLength()`.
- *equidistant*: bool, only used if *nseg* is specified and *ndiv* is not. If True the $nseg+1$ points are spaced almost equidistantly over the curve. If False (default), the points are spread equally over the parameter space.
- *npre*: integer, only used if the *chordal* method is used or if *nseg* is not None and *equidistant* is True: the number of segments per part of the curve (like *ndiv*) used in a pre-approximation. If not specified, it is set to the degree of the curve for the *chordal* method (1 for PolyLine), and to 100 in the *equidistant* method (where the pre-approximation is currently used to compute accurate curve lengths).

frenet (*ndiv=None, nseg=None, chordal=0.01, upvector=None, avgdir=True, compensate=False*)

Return points and Frenet frame along the curve.

A PolyLine approximation for the curve is constructed, using the `Curve.approx()` method with the arguments *ndiv*, *nseg* and *chordal*. Then Frenet frames are constructed with `PolyLine._movingFrenet()` using the remaining arguments. The resulting PolyLine points and Frenet frames are returned.

Parameters:

- *upvector*: (3,) vector: a vector normal to the (tangent,normal) plane at the first point of the curve. It defines the binormal at the first point. If not specified it is set to the shorted distance through the set of 10 first points.
- *avgdir*: bool or array. If True (default), the tangential vector is set to the average direction of the two segments ending at a node. If False, the tangent vectors will be those of the line segment starting at the points. The tangential vector can also be set by the user by specifying an array with the matching number of vectors.
- *compensate*: bool: If True, adds a compensation algorithm if the curve is closed. For a closed curve the moving Frenet algorithm can be continued back to the first point. If the resulting binormal does not coincide with the starting one, some torsion is added to the end portions of the curve to make the two binormals coincide.

This feature is off by default because it is currently experimental and is likely to change in future. It may also form the base for setting the starting as well as the ending binormal.

Returns:

- *X*: a Coords with *npts* points on the curve
- *T*: normalized tangent vector to the curve at *npts* points
- *N*: normalized normal vector to the curve at *npts* points

- *B*: normalized binormal vector to the curve at *npts* points

position (*geom*, *csys=None*)

Position a Geometry object along a path.

Parameters:

- *geom*: Geometry or Coords.
- *csys*: CoordSys.

For each point of the curve, a copy of the Geometry/Coords object is created, and positioned thus that the specified *csys* (default the global axes) coincides with the curve's frenet axes at that point.

Returns a list of Geometry/Coords objects.

sweep (*mesh*, *eltype=None*, *csys=None*)

Sweep a mesh along the curve, creating an extrusion.

Parameters:

- *mesh*: Mesh-like object. This is usually a planar object that is swept in the direction normal to its plane.
- *eltype*: string. Name of the element type on the returned Meshes.

Returns a Mesh obtained by sweeping the given Mesh over a path. The returned Mesh has double plexitude of the original. If *path* is a closed Curve connect back to the first.

Note: Sweeping nonplanar objects and/or sweeping along very curly curves may result in physically impossible geometries.

See also:

[*sweep2* \(\)](#)

sweep2 (*coords*, *origin=(0.0, 0.0, 0.0)*, *scale=None*, ***kargs*)

Sweep a Coords object along a curve, returning a series of copies.

At each point of the curve a copy of the coords is created, possibly scaled, and rotated to keep same orientation with respect to the curve.

Parameters

- **coords** (*Coords object*) – The Coords object to be copied, possibly scaled and positioned at each point of the curve.
- **origin** (float *array_like* (3,)) – The local origin in the Coords. This is the point that will be positioned at the curve's points. It is also the center of scaling if *scale* is provided.
- **scale** (float *array_like*, optional) – If provided, it should have the shape (npts,3). For each point of the curve, it specifies the three scaling factors to be used in the three coordinate directions on the *coords* for the copy that is to be placed at that point.
- ****kargs** (*optional keyword arguments*) – Extra keyword arguments that are passed to [*positionCoordsObj* \(\)](#).

Returns *A sequence of the transformed Coords objects.*

See also:

[*sweep* \(\)](#)

toFormex (*args, **kargs)

Convert a curve to a Formex.

This creates a polyline approximation as a plex-2 Formex. This is mainly used for drawing curves that do not implement their own drawing routines.

The method can be passed the same arguments as the *approx* method.

toNurbs ()

Convert a curve to a NurbsCurve.

This is currently only implemented for BezierSpline and PolyLine

setProp (p=None)

Create or destroy the property number for the Curve.

A curve can have a single integer as property number. If it is set, derived curves and approximations will inherit it. Use this method to set or remove the property.

Parameters:

- *p*: integer or None. If a value None is given, the property is removed from the Curve.

class plugins.curve.**PolyLine** (coords=[], control=None, closed=False)

A class representing a series of straight line segments.

coords is a (npts,3) shaped array of coordinates of the subsequent vertices of the polyline (or a compatible data object). If *closed* == True, the polyline is closed by connecting the last point to the first. This does not change the vertex data.

The *control* parameter has the same meaning as *coords* and is added for symmetry with other Curve classes. If specified, it will override the *coords* argument.

close (atol=0.0)

Close a PolyLine.

If the PolyLine is already closed, this does nothing. Else it is closed by one of two methods, depending on the distance between the start and end points of the PolyLine:

- if the distance is smaller than *atol*, the last point is removed and the last segment now connects the penultimate point with the first one, leaving the number of segments unchanged and the number of points decreased by one;
- if the distance is not smaller than *atol*, a new segment is added connecting the last point with the first, resulting in a curve with one more segment and the number of points unchanged. Since the default value for *atol* is 0.0, this is the default behavior for all PolyLines.

Returns True if the PolyLine was closed without adding a segment.

The return value can be used to reopen the PolyLine while keeping all segments (see *open*()).

Warning: This method changes the PolyLine inplace.

open (keep_last=False)

Open a closed PolyLine.

If the PolyLine is not closed, this does nothing.

Else, the PolyLine is opened in one of two ways:

- if *keep_last* is True, the PolyLine is opened by adding a last point equal to the first, and keeping the number of segments unchanged;

- if False, the PolyLine is opened by removing the last segment, keeping the number of points unchanged. This is the default behavior.

There is no return value.

If a closed PolyLine is opened with `keep_last=True`, the first and last point will coincide. In order to close it again, a positive `atol` value needs to be used in `close()`.

Warning: This method changes the PolyLine inplace.

nelems ()

Return the number of elements in the Geometry.

Returns *int* – The number of elements in the Geometry. This is an abstract method that should be reimplemented by the derived class.

toFormex ()

Return the PolyLine as a Formex.

toMesh ()

Convert the PolyLine to a plex-2 Mesh.

The returned Mesh is equivalent with the PolyLine.

sub_points (*t, j*)

Return the points at values *t* in part *j*

sub_points_2 (*t, j*)

Return the points at value,part pairs (*t,j*)

sub_directions (*t, j*)

Return the unit direction vectors at values *t* in part *j*.

vectors ()

Return the vectors of each point to the next one.

The vectors are returned as a Coords object. If the curve is not closed, the number of vectors returned is one less than the number of points.

directions (*return_doubles=False*)

Returns unit vectors in the direction of the next point.

This directions are returned as a Coords object with the same number of elements as the point set.

If two subsequent points are identical, the first one gets the direction of the previous segment. If more than two subsequent points are equal, an invalid direction (NaN) will result.

If the curve is not closed, the last direction is set equal to the penultimate.

If `return_doubles` is True, the return value is a tuple of the direction and an index of the points that are identical with their follower.

avgDirections (*return_doubles=False*)

Returns the average directions at points.

For each point the returned direction is the average of the direction from the preceding point to the current, and the direction from the current to the next point.

If the curve is open, the first and last direction are equal to the direction of the first, resp. last segment.

Where two subsequent points are identical, the average directions are set equal to those of the segment ending in the first and the segment starting from the last.

cosAngles ()

Return the cosinus of the angles between subsequent segments.

Returns an array of floats in the range [-1.0..1.0]. The value at index *i* is the cosinus of the angle between the segment *i* and the segment *i-1*. The number of floats is always equal to the number of points.

If the curve is not closed, the first value is the cosinus of the angle between the last and the first segment, while the last value is always equal to 1.0.

Where a curve has two subsequent coincident points, the value for the the first point will be 1.0. Where a curve has more than two subsequent coincident points, NAN values will result.

Examples:

```
>>> C1 = PolyLine('01567')
>>> C2 = PolyLine('01567',closed=True)
>>> C3 = PolyLine('015674')
>>> print(C1.cosAngles())
[-0.71  0.71  0.    0.    1.   ]
>>> print(C2.cosAngles())
[ 0.    0.71  0.    0.    0.71]
>>> print(C3.cosAngles())
[ 0.    0.71  0.    0.    0.71  1.   ]
```

splitByAngle (cosangle=0.0, angle=None, angle_spec=0.017453292519943295)

Split a curve at points with high change of direction.

Parameters:

- *cosangle*: float: threshold value for the cosinus of the angle between subsequent segment vectors. The curve will be split at all points where the value of `cosAngles()` is (algebraically) lower than or equal to this threshold value. The default value will split the curve where the direction changes with more than 90 degrees. A value 1.0 will split the curve at all points. A value of -1.0 will only split where the curve direction exactly reverses.
- *angle*: float: if specified, *cosangle* will be computed from `cosd(angle,angle_spec)`
- *angle_spec*: float: units used for the angle. This is the number of radians for 1 unit.

Returns a list of PolyLines, where each PolyLine has limited direction changes. Major changes in direction occur between the PolyLines.

roll (n)

Roll the points of a closed PolyLine.

lengths ()

Return the length of the parts of the curve.

cumLengths ()

Return the cumulative length of the curve for all vertices.

relLengths ()

Return the relative length of the curve for all vertices.

atLength (div)

Returns the parameter values at given relative curve length.

div is a list of relative curve lengths (from 0.0 to 1.0). As a convenience, a single integer value may be specified, in which case the relative curve lengths are found by dividing the interval [0.0,1.0] in the specified number of subintervals.

The function returns a list with the parameter values for the points at the specified relative lengths.

reverse ()

Return the same curve with the parameter direction reversed.

parts (j, k)

Return a PolyLine containing only segments j to k (k not included).

j and k should be in the range [0:nparts]. For an open curve $j < k$. For a closed curve a value $k \leq j$ is allowed, to get parts spanning the point 0 as a single (open) PolyLine.

The resulting PolyLine is always open.

cutWithPlane (p, n, side=)

Return the parts of the PolyLine at one or both sides of a plane.

If side is '+' or '-', return a list of PolyLines with the parts at the positive or negative side of the plane.

For any other value, returns a tuple of two lists of PolyLines, the first one being the parts at the positive side.

p is a point specified by 3 coordinates. n is the normal vector to a plane, specified by 3 components.

append (PL, fuse=True, smart=False, **kargs)

Concatenate two open PolyLines.

This combines two open PolyLines into a single one. Closed PolyLines cannot be concatenated.

Parameters:

- *PL*: an open PolyLine, to be appended at the end of the current.
- *fuse*: bool. If True, the last point of *self* and the first point of *PL* will be fused to a single point if they are close. Extra parameters may be added to tune the fuse operation. See the `Coords.fuse()` method. The *ppb* parameter is not allowed.
- *smart*: bool. If True, *PL* will be connected to *self* by the endpoint that is closest to the last point of *self*, thus possibly reversing the sense of *PL*.

The same result (with the default parameter values) can also be achieved using operator syntax: *PolyLine1* + *PolyLine2*.

static concatenate (PLlist, **kargs)

Concatenate a list of *PolyLine* objects.

Parameters:

- *PLlist*: a list of open PolyLines.
- Other parameters are like in `append()`

Returns a PolyLine which is the concatenation of all the PolyLines in *PLlist*

insertPointsAt (t, return_indices=False)

Insert new points at parameter values t.

Parameter values are floating point values. Their integer part is interpreted as the curve segment number, and the decimal part goes from 0 to 1 over the segment.

Returns a PolyLine with the new points inserted. Note that the parameter values of the points will have changed. If *return_indices* is True, also returns the indices of the inserted points in the new PolyLine.

splitAt (t)

Split a PolyLine at parametric values t

Parameter values are floating point values. Their integer part is interpreted as the curve segment number, and the decimal part goes from 0 to 1 over the segment.

Returns a list of open PolyLines.

splitAtLength (*L*)

Split a PolyLine at relative lengths *L*.

This is a convenience function equivalent with:

```
self.splitAt(self.atLength(L))
```

refine (*maxlen*)

Insert extra points in the PolyLine to reduce the segment length.

Parameters:

- *maxlen*: float: maximum length of the segments. The value is relative to the total curve length.

Returns a PolyLine which is geometrically equivalent to the input PolyLine.

vertexReductionDP (*tol, maxlen=None, keep=[0, -1]*)

Douglas-Peucker vertex reduction.

For any subpart of the PolyLine, if the distance of all its vertices to the line connecting the endpoints is smaller than *tol*, the internal points are removed.

Returns a bool array flagging the vertices to be kept in the reduced form.

coarsen (*tol=0.01, maxlen=None, keep=[0, -1]*)

Reduce the number of points of the PolyLine.

Parameters:

- *tol*: maximum relative distance of vertices to remove from the chord of the segment. The value is relative to the total curve length.
- *keep*: list of vertices to keep during the coarsening process. (Not implemented yet).

Returns a Polyline with a reduced number of points.

class `plugins.curve.Line` (*coords*)

A Line is a PolyLine with exactly two points.

Parameters:

- *coords*: compatible with (2,3) shaped float array

class `plugins.curve.BezierSpline` (*coords=None, deriv=None, curl=0.3333333333333333, control=None, closed=False, degree=3, endzerocurv=False*)

A class representing a Bezier spline curve of degree 1, 2 or 3.

A Bezier spline of degree *d* is a continuous curve consisting of *nparts* successive parts, where each part is a Bezier curve of the same degree. Currently pyFormex can model linear, quadratic and cubic BezierSplines. A linear BezierSpline is equivalent to a PolyLine, which has more specialized methods than the BezierSpline, so it might be more sensible to use a PolyLine instead of the linear BezierSpline.

A Bezier curve of degree *d* is determined by *d+1* control points, of which the first and the last are on the curve, while the intermediate *d-1* points are not. Since the end point of one part is the begin point of the next part, a BezierSpline is described by *ncontrol=d*nparts+1* control points if the curve is open, or *ncontrol=d*nparts* if the curve is closed.

The constructor provides different ways to initialize the full set of control points. In many cases the off-curve control points can be generated automatically.

Parameters:

- *coords* : *array_like* (npoints,3) The points that are on the curve. For an open curve, npoints=nparts+1, for a closed curve, npoints = nparts. If not specified, the on-curve points should be included in the *control* argument.
- *deriv* : *array_like* (npoints,3) or (2,3) or a list of 2 values one of which can be None and the other is a shape(3,) arraylike. If specified, it gives the direction of the curve at all points or at the endpoints only for a shape (2,3) array or only at one of the endpoints for a list of shape(3,) arraylike and a None type. For points where the direction is left unspecified or where the specified direction contains a *NaN* value, the direction is calculated as the average direction of the two line segments ending in the point. This will also be used for points where the specified direction contains a value *NaN*. In the two endpoints of an open curve however, this average direction can not be calculated: the two control points in these parts are set coincident.
- *curl* : float The curl parameter can be set to influence the curliness of the curve in between two subsequent points. A value curl=0.0 results in straight segments. The higher the value, the more the curve becomes curled.
- *control* : array(nparts,d-1,3) or array(ncontrol,3) If *coords* was specified and $d > 1$, this should be a (nparts,d-1,3) array with the intermediate control points, $d-1$ for each part. If *coords* was not specified, this should be the full array of *ncontrol* control points for the curve.
If not specified, the control points are generated automatically from the *coords*, *deriv* and *curl* arguments. If specified, they override these parameters.
- *closed* : boolean If True, the curve will be continued from the last point back to the first to create a closed curve.
- *degree* : int (1, 2 or 3) Specifies the degree of the curve. Default is 3.
- *endzeroconv* : boolean or tuple of two booleans. Specifies the end conditions for an open curve. If True, the end curvature will be forced to zero. The default is to use maximal continuity of the curvature. The value may be set to a tuple of two values to specify different conditions for both ends. This argument is ignored for a closed curve.

pointsOn ()

Return the points on the curve.

This returns a Coords object of shape [nparts+1]. For a closed curve, the last point will be equal to the first.

pointsOff ()

Return the points off the curve (the control points)

This returns a Coords object of shape [nparts,ndegree-1], or an empty Coords if degree <= 1.

part (*j*, *k=None*)

Returns the points defining parts [j:k] of the curve.

If *k* is None, it is set equal to *j*+1, resulting in a single part with degree+1 points.

sub_points (*t*, *j*)

Return the points at values *t* in part *j*.

sub_directions (*t*, *j*)

Return the unit direction vectors at values *t* in part *j*.

sub_curvature (*t*, *j*)

Return the curvature at values *t* in part *j*.

length_intgrnd (*t*, *j*)

Return the arc length integrand at value *t* in part *j*.

lengths ()

Return the length of the parts of the curve.

parts (*j, k*)

Return a curve containing only parts *j* to *k* (*k* not included).

The resulting curve is always open.

atLength (*l, approx=20*)

Returns the parameter values at given relative curve length.

Parameters:

- *l*: list of relative curve lengths (from 0.0 to 1.0). As a convenience, a single integer value may be specified, in which case the relative curve lengths are found by dividing the interval [0.0,1.0] in the specified number of subintervals.
- *approx*: int or None. If not None, an approximate result is returned obtained by approximating the curve first by a PolyLine with *approx* number of line segments per curve segment. This is currently the only implemented method, and specifying None will fail.

The function returns a list with the parameter values for the points at the specified relative lengths.

insertPointsAt (*t, split=False*)

Insert new points on the curve at parameter values *t*.

Parameters:

- *t*: float: parametric value where the new points will be inserted. Parameter values are floating point values. Their integer part is interpreted as the curve segment number, and the decimal part goes from 0 to 1 over the segment.
 - Currently there can only be one new point in each segment *
- *split*: bool: if True, this method behaves like the `split()` method. Users should use the latter method instead of the *split* parameter.

Returns a single BezierSpline (default). The result is equivalent with the input curve, but has more points on the curve (and more control points. If *split* is True, the behavior is that of the `split()` method.

splitAt (*t*)

Split a BezierSpline at parametric values.

Parameters:

- *t*: float: parametric value where the new points will be inserted. Parameter values are floating point values. Their integer part is interpreted as the curve segment number, and the decimal part goes from 0 to 1 over the segment.
 - Currently there can only be one new point in each segment *

Returns a list of `len(t)+1` open BezierSplines of the same degree as the input.

toMesh ()

Convert the BezierSpline to a Mesh.

For degrees 1 or 2, the returned Mesh is equivalent with the BezierSpline, and will have element type 'line1', resp. 'line2'.

For degree 3, the returned Mesh will currently be a quadratic approximation with element type 'line2'.

extend (*extend=[1.0, 1.0]*)

Extend the curve beyond its endpoints.

This function will add a Bezier curve before the first part and/or after the last part by applying de Casteljau's algorithm on this part.

reverse ()

Return the same curve with the parameter direction reversed.

class `plugins.curve.Contour` (*coords, elems*)

A class for storing a contour.

The contour class stores a continuous (usually closed) curve which consists of a sequence of strokes, each stroke either being a straight segment or a quadratic or cubic Bezier curve. A stroke is thus defined by 2, 3 or 4 points. The contour is defined by a list of points and a Varray of element connectivity. This format is well suited to store contours of scalable fonts. The contours are in that case usually 2D.

endPoints ()

Return start and end points of the curve.

Returns a Coords with two points, or None if the curve is closed.

part (i)

Returns the points defining part j of the curve.

stroke (i)

Return curve for part i

sub_points (t, j)

Return the points at values t in part j

t can be an array of parameter values, j is a single segment number.

sub_directions (t, j)

Return the directions at values t in part j

t can be an array of parameter values, j is a single segment number.

atChordal (*chordal=0.01, at=None*)

Return parameter values to approximate within given chordal error.

Parameters:

- **chordal**: relative tolerance on the distance of the chord to the curve. The tolerance is relative to the curve's `charLength()`.
- **at**: list of floats: list of parameter values that need to be included in the result. The list should contain increasing values in the curve's parameter range. If not specified, a default is set assuring that the curve is properly approximated. If you specify this yourself, you may end up with bad approximations due to bad choice of the initial values.

Returns a list of parameter values that create a PolyLine approximate for the curve, such that the chordal error is everywhere smaller than the give value. The chordal error is defined as the distance from a point of the curve to the chord.

toMesh ()

Convert the Contour to a Mesh.

class `plugins.curve.CardinalSpline` (*coords, tension=0.0, closed=False, endzerocurv=False*)

A class representing a cardinal spline.

Create a natural spline through the given points.

The Cardinal Spline with given tension is a Bezier Spline with curl :math: curl = (1 - tension) / 3 The separate class name is retained for compatibility and convenience. See `CardinalSpline2` for a direct implementation (it misses the end intervals of the point set).

class `plugins.curve.CardinalSpline2` (*coords, tension=0.0, closed=False*)
 A class representing a cardinal spline.

sub_points (*t, j*)
 Return the points at values *t* in part *j*
t can be an array of parameter values, *j* is a single segment number.

class `plugins.curve.NaturalSpline` (*coords, closed=False, endzerocurv=False*)
 A class representing a natural spline.

sub_points (*t, j*)
 Return the points at values *t* in part *j*
t can be an array of parameter values, *j* is a single segment number.

class `plugins.curve.Arc3` (*coords*)
 A class representing a circular arc.

sub_points (*t, j*)
 Return the points at values *t* in part *j*
t can be an array of parameter values, *j* is a single segment number.

class `plugins.curve.Arc` (*coords=None, center=None, radius=None, angles=None, angle_spec=0.017453292519943295*)
 A class representing a circular arc.

The arc can be specified by 3 points (begin, center, end) or by center, radius and two angles. In the latter case, the arc lies in a plane parallel to the x,y plane. If specified by 3 colinear points, the plane of the circle will be parallel to the x,y plane if the points are in such plane, else the plane will be parallel to the z-axis.

sub_points (*t, j*)
 Return the points at values *t* in part *j*
t can be an array of parameter values, *j* is a single segment number.

sub_directions (*t, j*)
 Return the directions at values *t* in part *j*
t can be an array of parameter values, *j* is a single segment number.

approx (*ndiv=None, chordal=0.001*)
 Return a PolyLine approximation of the Arc.
 Approximates the Arc by a sequence of inscribed straight line segments.
 If *ndiv* is specified, the arc is divided in precisely *ndiv* segments.
 If *ndiv* is not given, the number of segments is determined from the chordal distance tolerance. It will guarantee that the distance of any point of the arc to the chordal approximation is less or equal than *chordal* times the radius of the arc.

class `plugins.curve.Spiral` (*turns=2.0, nparts=100, rfunc=None*)
 A class representing a spiral curve.

Functions defined in module `plugins.curve`

`plugins.curve.circle` ()
 Create a spline approximation of a circle.
 The returned circle lies in the x,y plane, has its center at (0,0,0) and has a radius 1.
 In the current implementation it is approximated by a bezier spline with curl 0.375058 through 8 points.

`plugins.curve.arc2points(x0, x1, R, pos='-')`

Create an arc between two points

Given two points `x0` and `x1`, this constructs an arc with radius `R` through these points. The two points should have the same `z`-value. The arc will be in a plane parallel with the `x-y` plane and wind positively around the `z`-axis when moving along the arc from `x0` to `x1`.

If `pos == '-'`, the center of the arc will be at the left when going along the chord from `x0` to `x1`, creating an arc smaller than a half-circle. If `pos == '+'`, the center of the arc will be at the right when going along the chord from `x0` to `x1`, creating an arc larger than a half-circle.

If `R` is too small, an exception is raised.

`plugins.curve.binomial(n, k)`

Compute the binomial coefficient $C_{n,k}$.

This computes the binomial coefficient $C_{n,k} = \text{fac}(n) // \text{fac}(k) // \text{fac}(n-k)$.

Example:

```
>>> print([ binomial(3,i) for i in range(4) ])
[1, 3, 3, 1]
```

`plugins.curve.binomialCoeffs(p)`

Compute all binomial coefficients for a given degree `p`.

Returns an array of `p+1` values.

For efficiency reasons, the computed values are stored in the module, in a dict with `p` as key. This allows easy and fast lookup of already computed values.

Example:

```
>>> print(binomialCoeffs(4))
[1 4 6 4 1]
```

`plugins.curve.bezierPowerMatrix(p)`

Compute the Bezier to power curve transformation matrix for degree `p`.

Bezier curve representations can be converted to power curve representations using the coefficients in this matrix.

Returns a `(p+1,p+1)` shaped array with a zero upper triangular part.

For efficiency reasons, the computed values are stored in the module, in a dict with `p` as key. This allows easy and fast lookup of already computed values.

Example:

```
>>> print(bezierPowerMatrix(4))
[[ 1.  0.  0.  0.  0.]
 [ -4.  4.  0.  0.  0.]
 [  6. -12.  6.  0.  0.]
 [ -4.  12. -12.  4.  0.]
 [  1.  -4.  6. -4.  1.]]
```

```
>>> 4 in _bezier_power_matrix
True
```

`plugins.curve.convertFormexToCurve(self, closed=False)`

Convert a Formex to a Curve.

The following Formices can be converted to a Curve: - plex 2 : to PolyLine - plex 3 : to BezierSpline with degree=2 - plex 4 : to BezierSpline

`plugins.curve.positionCoordsObj` (*objects, path, normal=0, upvector=2, avgdir=False, enddir=None*)

Position a sequence of Coords objects along a path.

At each point of the curve, a copy of the Coords object is created, with its origin in the curve's point, and its normal along the curve's direction. In case of a PolyLine, directions are pointing to the next point by default. If `avgdir==True`, average directions are taken at the intermediate points `avgdir` can also be an array like sequence of shape (N,3) to explicitly set the directions for ALL the points of the path

Missing end directions can explicitly be set by `enddir`, and are by default taken along the last segment. `enddir` is a list of 2 array like values of shape (3). one of the two can also be an empty list. If the curve is closed, endpoints are treated as any intermediate point, and the user should normally not specify `enddir`.

The return value is a sequence of the repositioned Coords objects.

6.4.6 `plugins.datareader` — Numerical data reader

Functions defined in module `plugins.datareader`

`plugins.datareader.splitFloat` (*s*)

Match a floating point number at the beginning of a string

If the beginning of the string matches a floating point number, a list is returned with the float and the remainder of the string; if not, None is returned. Example: `splitFloat('123e4rt345e6')` returns `[1230000.0, 'rt345e6']`

`plugins.datareader.readData` (*s, type, strict=False*)

Read data from a line matching the 'type' specification.

This is a powerful function for reading, interpreting and converting numerical data from a string. Fields in the string *s* are separated by commas. The 'type' argument is a list where each element specifies how the corresponding field should be interpreted. Available values are 'int', 'float' or some unit ('kg', 'm', etc.). If the type field is 'int' or 'float', the data field is converted to the matching type. If the type field is a unit, the data field should be a number and a unit separated by space or not, or just a number. If it is just a number, its value is returned unchanged (as float). If the data contains a unit, the number is converted to the requested unit. It is an error if the datafield holds a non-conformable unit. The function returns a list of ints and/or floats (without the units). If the number of data fields is not equal to the number of type specifiers, the returned list will correspond to the shortest of both and the surplus data or types are ignored, UNLESS the `strict` flag has been set, in which case a `RuntimError` is raised. Example:

```
readData('12, 13, 14.5e3, 12 inch, 1hr, 31kg ', ['int','float','kg','cm','s'])
```

```
returns [12, 13.0, 14500.0, 30.48, 3600.0]
```

```
..warning
```

```
You need to have the GNU ``units`` command installed for the unit
conversion to work.
```

6.4.7 `plugins.dxf` — Read/write geometry in DXF format.

This module allows to import and export some simple geometrical items in DXF format.

Classes defined in module plugins.dxf

class `plugins.dxf.DxfExporter` (*filename*, *terminator='n'*)
Export geometry in DXF format.

While we certainly do not want to promote proprietary software, some of our users occasionally needed to export some model in DXF format. This class provides a minimum of functionality.

write (*s*)

Write a string to the dxf file.

The string does not include the line terminator.

out (*code*, *data*)

Output a string data item to the dxf file.

code is the group code, *data* holds the data

close ()

Finalize and close the DXF file

section (*name*)

Start a new section

endSection ()

End the current section

entities ()

Start the ENTITIES section

layer (*layer*)

Export the layer

vertex (*x*, *layer=0*)

Export a vertex.

x is a (3,) shaped array

line (*x*, *layer=0*)

Export a line.

x is a (2,3) shaped array

polyline (*x*, *layer=0*)

Export a polyline.

x is a (nvertices,3) shaped array

arc (*C*, *R*, *a*, *layer=0*)

Export an arc.

Functions defined in module plugins.dxf

`plugins.dxf.importDXF` (*filename*)

Import (parts of) a DXF file into pyFormex.

This function scans a DXF file for recognized entities and imports those entities as pyFormex objects. It is only a very partial importer, but has proven to be already very valuable for many users.

filename: name of a DXF file. The return value is a list of pyFormex objects.

Importing a DXF file is done in two steps:

- First the DXF file is scanned and the recognized entities are formatted into a text with standard function calling syntax. See `readDXF()`.
- Then the created text is executed as a Python script, producing equivalent pyFormex objects. See `convertDXF()`.

`plugins.dxf.readDXF(filename)`

Read a DXF file and extract the recognized entities.

filename: name of a .DXF file.

Returns a multiline string with one line for each recognized entity, in a format that can directly be used by `convertDXF()`.

This function requires the external program `dxfparser` which comes with the pyFormex distribution. It currently recognizes entities of type 'Arc', 'Line', 'Polyline', 'Vertex'.

`plugins.dxf.convertDXF(text)`

Convert a textual representation of a DXF format to pyFormex objects.

text [a multiline text representation of the contents of a DXF file.] This text representation can e.g. be obtained by the function `readDXF()`. It contains lines defining DXF entities. A small example:

```
Arc(0.0,0.0,0.0,1.0,-90.,90.)
Arc(0.0,0.0,0.0,3.0,-90.,90.)
Line(0.0,-1.0,0.0,0.0,1.0,0.0)
Polyline(0)
Vertex(0.0,3.0,0.0)
Vertex(-2.0,3.0,0.0)
Vertex(-2.0,-7.0,0.0)
Vertex(0.0,-7.0,0.0)
Vertex(0.0,-3.0,0.0)
```

Each line of the text defines a single entity or starts a multiple component entity. The text should be well aligned to constitute a proper Python script. Currently, the only defined entities are 'Arc', 'Line', 'Polyline', 'Vertex'.

Returns a list of pyFormex objects corresponding to the text. The returned objects are of the following type:

function name	object
Arc	<code>plugins.curve.Arc</code>
Line	<code>plugins.curve.Line</code>
Polyline	<code>plugins.curve.PolyLine</code>

No object is returned for the `Vertex` function: they define the vertices of a `PolyLine`.

`plugins.dxf.collectByType(entities)`

Collect the dxf entities by type.

`plugins.dxf.toLines(coll, chordal=0.01, arcddiv=None)`

Convert the dxf entities in a dxf collection to a plex-2 Formex

This converts Lines, Arcs and PolyLines to plex-2 elements and collects them in a single Formex. The `chordal` and `arcddiv` parameters are passed to `Arc.approx()` to set the accuracy for the approximation of the Arc by line segments.

`plugins.dxf.exportDXF(filename, F)`

Export a Formex to a DXF file

Currently, only plex-2 Formices can be exported to DXF.

`plugins.dxf.exportDxf` (*filename, coll*)

Export a collection of dxf parts a DXF file

coll is a list of dxf objects

Currently, only dxf objects of type 'Line' and 'Arc' can be exported.

`plugins.dxf.exportDxfText` (*filename, parts*)

Export a set of dxf entities to a .dxf text file.

6.4.8 `plugins.export` — Classes and functions for exporting geometry in various formats.

- `ObjFile`: wavefront .obj format
- `PlyFile`: .ply format

These classes do not necessarily implement all features of the specified formats. They are usually fitted to export and sometimes imports simple mesh format geometries.

Classes defined in module `plugins.export`

class `plugins.export.ObjFile` (*filename*)

Export a mesh in OBJ format.

This class exports a mesh in Wavefront OBJ format (see https://en.wikipedia.org/wiki/Wavefront_.obj_file).

The class instantiator takes a filename as parameter. Normally, the file name should end in '.obj', but anything is accepted. The `ObjFile` instance is a context manager.

Usage:

```
with ObjFile(PATH_TO_OBJFILE) as fil:
    fil.write(MESH)
```

write (*mesh, name=None*)

Write a mesh to file in .obj format.

mesh is a `Mesh` instance or another object having compatible coords and elems attributes. *name* is an optional name of the object.

class `plugins.export.PlyFile` (*filename, binary=False*)

Export a mesh in PLY format.

This class exports a mesh in Polygon File Format (see [https://en.wikipedia.org/wiki/PLY_\(file_format\)](https://en.wikipedia.org/wiki/PLY_(file_format))).

The class instantiator takes a filename as parameter. Normally, the file name should end in '.ply', but anything is accepted. The `PlyFile` instance is a context manager.

Usage:

```
with PlyFile(PATH_TO_PLYFILE) as fil:
    fil.write(MESH)
```

write (*mesh, comment=None, color_table=None*)

Write a mesh to file in .ply format.

Parameters:

- *mesh*: a `Mesh` instance or another object having compatible coords and elems attributes.

- *comment*: an extra comment written into the output file.
- *color_table*: BEWARE! THIS IS SUBJECT TO CHANGES!

color_table currently is a list of 2 elements. The first entry is a string that can assume 2 values 'v' or 'e', to indicate whether the color table represents nodal or element values. The second entry is an array of shape (ncoords,3) or (nelems,3) of integer RGB values between 0 and 255. If RGB values are passed as float between 0 and 1, they will be converted to RGB integers.

6.4.9 `plugins.fe` — Finite Element Models in pyFormex.

Finite element models are geometrical models that consist of a unique set of nodal coordinates and one of more sets of elements.

Classes defined in module `plugins.fe`

class `plugins.fe.Model` (*coords=None, elems=None, meshes=None, fuse=True*)

Contains all FE model data.

meshes ()

Return the parts as a list of meshes

nnodes ()

Return the number of nodes in the model.

nelems ()

Return the number of elements in the model.

ngroups ()

Return the number of element groups in the model.

mplex ()

Return the plexitude of all the element groups in the model.

Returns a list of integers.

splitElems (*elems*)

Splits a set of element numbers over the element groups.

Parameters:

- *elems*: a list of element numbers in the range 0..self.nelems()

Returns two lists of element sets, the first in global numbering, the second in group numbering. Each item contains the element numbers from the given set that belong to the corresponding group.

elemNrs (*group, elems=None*)

Return the global element numbers for elements in group

Parameters:

- *group*: group number
- *elems*: list of local element numbers. If omitted, the list of all the elements in that group is used.

Returns a list with corresponding global element numbers.

getElems (*sets*)

Return the definitions of the elements in sets.

sets should be a list of element sets with length equal to the number of element groups. Each set contains element numbers local to that group.

As the elements can be grouped according to plexitude, this function returns a list of element arrays matching the element groups in `self.elems`. Some of these arrays may be empty.

renumber (*old=None, new=None*)

Renumber a set of nodes.

`old` and `new` are equally sized lists with unique node numbers, all smaller than the number of nodes in the model. The old numbers will be renumbered to the new numbers. If one of the lists is `None`, a range with the length of the other is used. If the lists are shorter than the number of nodes, the remaining nodes will be numbered in an unspecified order. If both lists are `None`, the nodes are renumbered randomly.

This function returns a tuple (`old,new`) with the full renumbering vectors used. The first gives the old node numbers of the current numbers, the second gives the new numbers corresponding with the old ones.

class `plugins.fe.FEModel` (*meshes*)

A Finite Element Model.

This class is intended to collect all data concerning a Finite Element Model. In due time it may replace the `Model` class. Currently it only holds geometrical data, but will probably be expanded later to include a property database holding material data, boundary conditions, loading conditions and simulation step data.

While the `Model` class stores the geometry in a single `coords` block and multiple `elems` blocks, the new `FEModel` class uses a list of `Meshes`. The `Meshes` do not have to be compact though, and thus all `Meshes` in the `FEModel` could use the same `coords` block, resulting in an equivalent model as the old `Model` class. But the `Meshes` may also use different `coords` blocks, allowing to accommodate better to versatile applications.

nelems ()

Return the number of elements in the model.

Functions defined in module `plugins.fe`

`plugins.fe.mergedModel` (*meshes, **kargs*)

Returns the `fe Model` obtained from merging individual meshes.

The input arguments are (`coords,elems`) tuples. The return value is a merged `fe Model`.

`plugins.fe.sortElemsByLoadedFace` (*ind*)

Sort a set of face loaded elements by the loaded face local number

`ind` is a (`nelems,2`) array, where `ind[:,0]` are element numbers and `ind[:,1]` are the local numbers of the loaded faces

Returns a dict with the loaded face number as key and a list of element numbers as value.

For a typical use case, see the `FePlast` example.

6.4.10 `plugins.fe_abq` — Export finite element models in Abaqus™ input file format.

This module provides functions and classes to export finite element models from `pyFormex` in the Abaqus™ input format (`.inp`). The exporter handles the mesh geometry as well as model, node and element properties gathered in a `PropertyDB` database (see module `properties`).

While this module provides only a small part of the Abaqus input file format, it suffices for most standard jobs. While we continue to expand the interface, depending on our own necessities or when asked by third parties, we do not intend to make this into a full implementation of the Abaqus input specification. If you urgently need some missing function, there is always the possibility to edit the resulting text file or to import it into the Abaqus environment for further processing.

The module provides two levels of functionality: on the lowest level, there are functions that just generate a part of an Abaqus input file, conforming to the Abaqus™ Keywords manual.

Then there are higher level functions that read data from the property module and write them to the Abaqus input file and some data classes to organize all the data involved with the finite element model.

Classes defined in module `plugins.fe_abq`

class `plugins.fe_abq.Command` (*cmd*, **args*, ***kargs*)

A class to format a keyword block in an INP file.

Parameters (all are optional, except for *cmd*):

- *cmd*: string, starting with an Abaqus keyword. The *cmd* string may include already formatted options. It will be converted to upper case and put as is after the initial '*' character.
- *options*: string. String to be added to the command line after any other *args* and *kargs* keyword options have been formatted (even those added later with the `add()` method). If the string does not start with a comma, a ', ' will be interposed.
- *data*: list-like or list of tuple. Specifies the data to be put below the command line. A list of tuples will be formatted with one tuple per line. Any other data type will be transformed to a flat sequence and be formatted with maximum 8 values per line. Each individual item is converted to str type, so the data sequence may contain numerical (float or int) as well as string data.
- *extra*: string: will be added as is below the command and data. This may be a multiline string and can be used to add complete preformatted sections to the output.
- *args*: any other non-keyword arguments are added as options to the command line, after conversion to string and upper case.
- *kargs*: any other keyword arguments are added to the command line as options of the form 'KEY=value'. The keys are converted to upper case, the values not.

After initial construction, the `add()` method can be used to add more parts to the command.

Remark: there is no check whether a certain option contains multiple occurrences of the same option.

Examples

```
>>> cmd1 = Command('Key', 'material=plastic', options='material=steel', data=[1, 2, 3,
↪4.0, 'last'], extra='*AnotherKey, opt=optionalSet1, \n 1, 4.7')
>>> cmd1.add(material='wood')
>>> print(cmd1)
*KEY, MATERIAL=PLASTIC, MATERIAL=wood, material=steel
1, 2, 3, 4.0, last
*AnotherKey, opt=optionalSet1,
 1, 4.7
<BLANKLINE>
```

add (**args*, ***kargs*)

Add more parts to the Command.

This method takes all the parameters as the initializer, except for *cmd*. Specifying *data* will overwrite any previously defined data for the command. All other parameters will be added to the already existing.

out

The formatted command as a string

```
class plugins.fe_abq.Output (kind="", vars='PRESELECT', set=None, typeset=None,
                             type='FIELD', options="", extra="", variable=None, keys=None,
                             *args, **kargs)
```

A request for output to .odb.

Parameters:

- *type*: 'FIELD' (default), 'HISTORY' or ''.
- *kind*: string: one of '', 'N', 'NODE', 'E', 'ELEMENT', 'ENERGY', 'CONTACT'. 'N' and 'E' are abbreviations for 'NODE', 'ELEMENT', respectively.
- *vars*: 'ALL', 'PRESELECT' or, if *kind* != None, a list of output identifiers compatible with the specified *kind*.
- *set*: a single set name or a list of node/element set names. This can not be specified for *kind*==''. If no set is specified, the default is 'Nall' for *kind*=='NODE' and 'Eall' for *kind*=='ELEMENT'
- *typeset*: string: it is equal to the corresponding abaqus parameter to identify the kind of set. If not specified, the default is 'NSET' for *kind*=='NODE' , 'CONTACT' and 'ELSET' for *kind*=='ELEMENT' , 'ENERGY'
- *options*: (opt) options string to be added to the keyword line.
- *extra*: (opt) extra string to be added below the keyword line and optional data.

Examples:

```
>>> out1 = Output (type='field')
>>> print (out1.fmt ())
*OUTPUT, FIELD, VARIABLE=PRESELECT

>>> out0 = Output (vars=None)
>>> out2 = Output (type='field', kind='e', vars=['S','SP'])
>>> print (out0.fmt ()+out2.fmt ())
*OUTPUT, FIELD
*ELEMENT OUTPUT, ELSET=Eall
S, SP

>>> out3 = Output (type='field', kind='e', vars=['S','SP'], set=['set1','set2'])
>>> print (out0.fmt ()+out3.fmt ())
*OUTPUT, FIELD
*ELEMENT OUTPUT, ELSET=set1
S, SP
*ELEMENT OUTPUT, ELSET=set2
S, SP

>>> out4 = Output (type='', diagnostic='yes')
>>> print (out4.fmt ())
*OUTPUT, DIAGNOSTIC=yes
```

fmt ()

Format an output request.

Return a string with the formatted output command.

```
class plugins.fe_abq.Result (kind, keys, set=None, output='FILE', freq=1, time=False, **kargs)
```

A request for output of results on nodes or elements.

Parameters:

- *kind*: 'NODE' or 'ELEMENT' (first character suffices)
- *keys*: a list of output identifiers (compatible with kind type)
- *set*: a single item or a list of items, where each item is either a property number or a node/element set name for which the results should be written. If no set is specified, the default is 'Nall' for kind=='NODE' and 'Eall' for kind=='ELEMENT'
- *output* is either FILE (for .fil output) or PRINT (for .dat output)(Abaqus/Standard only)
- *freq* is the output frequency in increments (0 = no output)

Extra keyword arguments are available: see the *writeNodeResults* and *writeElemResults* methods for details.

```
class plugins.fe_abq.AbqData (model, prop, nprop=None, eprop=None, steps=[], res=[], out=[],
                               initial=None, eofs=1, nofs=1, extra="")
```

Contains all data required to write the Abaqus input file.

- *model* : a `Model` instance.
- *prop* : the *Property* database.
- *nprop* : the node property numbers to be used for by-prop properties.
- *eprop* : the element property numbers to be used for by-prop properties.
- *steps* : a list of *Step* instances.
- *res* : a list of *Result* instances to be applied on all steps.
- *out* : a list of *Output* instances to be applied on all steps.
- *initial* : a tag or alist of the initial data, such as boundary conditions. The default is to apply ALL boundary conditions initially. Specify a (possibly non-existing) tag to override the default.
- *eofs* : integer defining the element offset for the abaqus numbering. Default value is 1.
- *nofs* : integer defining the node offset for the abaqus numbering. Default value is 1.
- *extra* : string to be added at model level.

```
write (jobname=None, group_by_eset=True, group_by_group=False, subsets=True, comment=None,
        header="", create_part=False)
```

Write an Abaqus input (INP) file.

- *jobname*: relative or absolute path name of the exported Abaqus INP file. If the name does not end in '.inp', this extension will be appended. If no name is specified, the output is written to sys.stdout.
- *comment*: A text to be included at the top of the INP file, right after the 'created by pyFormex' line. The text can be a multiline string or a function returning such string. Any other object will be dumped to a string in JSON format. All lines of the resulting text are prepended with '** ' before inclusion in the INP file, so Abaqus will recognize them as comments.
- *header*: A text like comments, but this one will be inserted in the header section of the INP file, and not marked as comments. This is commonly used to add information that should appear in the result files.
- *create_part* : if True, the model will be created as an Abaqus Part, followed by an assembly of that part.

Functions defined in module `plugins.fe_abq`

`plugins.fe_abq.abqInputNames` (*job*)

Returns corresponding jobname and input filename.

Parameters *job* (*str* or *Path*) – The job name or input file name, with or without a directory part, with or without a suffix `‘.inp’`.

Returns

- **jobname** (*str*) – The basename of job, without the suffix (stem).
- **filename** (*Path*) – The absolute path name of the input file.

Examples

```
>>> jobname, filename = abqInputNames('myjob.inp')
>>> jobname, filename.relative_to(Path.cwd())
('myjob', Path('myjob.inp'))
>>> jobname, filename = abqInputNames('/home/user/mydict/myjob.inp')
>>> print(jobname, filename)
myjob /home/user/mydict/myjob.inp
>>> jobname, filename = abqInputNames('mydict/myjob')
>>> jobname, filename.relative_to(Path.cwd())
('myjob', Path('mydict/myjob.inp'))
```

`plugins.fe_abq.nsetName` (*p*)

Determine the name for writing a node set property.

`plugins.fe_abq.esetName` (*p*)

Determine the name for writing an element set property.

`plugins.fe_abq.fmtData2d` (*data*, *linesep=‘\n’*)

Format 2D data.

data: list.

Each item of data is formatted using `fmtData1D` and the resulting strings are joined with `linesep`.

Examples

```
>>> print(fmtData2d([('set0', 1, 5.0), ('set1', 2, 10.0)]))
set0, 1, 5.0
set1, 2, 10.0
```

`plugins.fe_abq.fmtData` (*data*, *linesep=‘\n’*)

Format the data section

If data is a list of tuples, or data is a 2D array, each item/row of data will be formatted on a separate line. Any other data will be formatted as a 1D sequence with 8 items per line. Lines are separated with `linesep`.

Examples:

```
>>> print(fmtData([1, 2, 3, 4.0, 'last']))
1, 2, 3, 4.0, last
>>> print(fmtData([(1, 2), (3, 4.0, 'last')]))
1, 2
3, 4.0, last
```

`plugins.fe_abq.fmtKeyword(keyword, options="", data=None, extra="", *args, **kargs)`

Format any keyword block in INP file.

- *keyword*: string, keyword command, possibly including options
- *options*: string or Options. The argument is first converted to str. If the result starts with a comma, it is added as is to the command line; otherwise it is added with interposition of ‘, ‘.
- *data*: numerical data: will be formatted with maximum 8 values per line, or a list of tuples: each tuple will be formatted on a line
- *extra*: string: will be added as is below the command and data

All other arguments will be formatted with `fmtOptions` on the command line, between *keyword* and *options*.

Examples:

`plugins.fe_abq.fmtOption(key, value)`

Format a single option.

`plugins.fe_abq.fmtOptions(**kargs)`

Format the options of an Abaqus command line.

Each key,value pair in the argument list is formatted into a string ‘KEY=value’, or just ‘KEY’ if the value is an empty string. The key is always converted to upper case, and any underscore in the key is replaced with a space. The resulting strings are joined with ‘, ‘ between them.

Returns a comma-separated string of ‘keyword’ or ‘keyword=value’ fields. The string has a leading but no trailing comma.

Note that if you specified two arguments whose keyword only differs by case, both will appear in the output with the same keyword. Also note that the order in which the options appear is unspecified.

Examples

```
>>> print(fmtOptions(var_a = 123., var_B = '123.', Var_C = ''))
, VAR A=123.0, VAR B=123., VAR C
```

`plugins.fe_abq.fmtWatermark()`

Format the pyFormex watermark.

The pyFormex watermark contains the version used to create the output. This should always be the first line of the output file.

```
>>> print(fmtWatermark())
** Abaqus input file created by pyFormex 1.0.7 (http://pyformex.org)
**
<BLANKLINE>
```

`plugins.fe_abq.fmtComment(text=None)`

Format the initial comment section.

```
>>> print(fmtComment("This is a comment\n of two lines."))
**This is a comment
** of two lines.
<BLANKLINE>
```

`plugins.fe_abq.fmtHeading(text="")`

Format the heading section.

Any specified text will be included in the heading section.

```
>>> print (fmtHeading("This is the heading"))
**
*HEADING
This is the heading
<BLANKLINE>
```

`plugins.fe_abq.fmtSectionHeading` (*text*=")
Format a section heading of the Abaqus input file.

`plugins.fe_abq.fmtPart` (*name*='Part-1')
Start a new Part.

`plugins.fe_abq.fmtMaterial` (*mat*)
Write a material section.

mat is the property dict of the material. The following keys are recognized and output accordingly. The keys labeled (opt) are optional.

- *name*: if specified, and a material with this name has already been written, this function does nothing.
- *elasticity*: one of 'LINEAR', 'HYPERELASTIC', 'ANISOTROPIC HYPERELASTIC', 'USER' or another string specifying a valid material command. Default is 'LINEAR'. Defines the elastic behavior class of the material. The required and recognized keys depend on this parameter (see below).
- *constants*: list of floats or None. The material constants to be used in the model. For 'LINEAR' elasticity, these may alternatively be specified by other keywords (see below).
- *options* (opt): string: will be added to the material command as is.
- *extra* (opt): (multiline) string: will be added to the material data as is.
- *plastic* (opt): arraylike, float, shape (N,2). Definition of the material plasticity law. Each row contains a tuple of a yield stress value and the corresponding equivalent plastic strain.
- *damping* (opt): tuple (alpha,beta). Adds damping into the material. See Abaqus manual for the meaning of alpha and beta. Either of them can be a float or None.
- *field* (opt): boolean. If True, a keyword "USER DEFINED FIELD" is added.

Recognized keys depending on model:

'LINEAR': allows the specification of the material constants using the following keys:

- *young_modulus*: float
- *shear_modulus* (opt): float
- *poisson_ratio* (opt): float: if not specified, it is calculated from the above two values.

'HYPERELASTIC': has a required key 'model':

- *model*: one of 'OGDEN', 'POLYNOMIAL' or 'REDUCED POLYNOMIAL'. The number of constants to be specified depends on the model and the order. The temperature should not be included in the temperature.
- *order* (opt): order of the model. If omitted, it is calculated from the number of constants specified.
- *temp* (opt): temperature at which these constants are valid. If omitted, temperature is set to 0.0 degrees.
- *testdata* (opt): list. The first item of the list is one of 'UNIFORM', 'PLANAR', 'BIAXIAL'. The second item of the list is the array of values of the test data shaped as they should be written in the input file. The third item is optional. It is an integer representing the smooth factor of the data (see Abaqus manual).

'ANISOTROPIC HYPERELASTIC': has a required key 'model':

- model: one of 'FUNG-ANISOTROPIC', 'FUNG-ORTHOTROPIC', 'HOLZAPFEL' or 'USER'.
- depvar (opt): see below ('USER')
- localdir (opt): int: number of local directions.

'USER':

- depvar (opt): list. The first item in the list is the number of solution dependent variables. Further items are optional and are to specify output names for (some of) solution dependent state variables. Each item is a tuple of a variable number and the corresponding variable name. See the examples below.

Example:

```
>>> steel = {
...     'name': 'steel',
...     'young_modulus': 207000,
...     'poisson_ratio': 0.3,
...     'density': 7.85e-9,
...     'plastic': [(200.,0.), (900.,0.5)],
... }
>>> from pyformex.attributes import Attributes
>>> print (fmtMaterial (Attributes (steel)))
*MATERIAL, NAME=steel
*ELASTIC
207000.0, 0.3
*DENSITY
7.85e-09
*PLASTIC
200.0, 0.0
900.0, 0.5
<BLANKLINE>
```

```
>>> intima = {
...     'name': 'intima',
...     'density': 0.1,
...     'elasticity': 'hyperelastic',
...     'model': 'reduced polynomial',
...     'constants': [6.79E-03, 5.40E-01, -1.11, 10.65, -7.27, 1.63, 0.0, 0.0, 0.
↪0, 0.0, 0.0, 0.0]
... }
>>> print (fmtMaterial (Attributes (intima)))
*MATERIAL, NAME=intima
*HYPERELASTIC, REDUCED POLYNOMIAL, N=6
0.00679, 0.54, -1.11, 10.65, -7.27, 1.63, 0.0, 0.0
0.0, 0.0, 0.0, 0.0, 0.0
*DENSITY
0.1
<BLANKLINE>
```

```
>>> nitinol = {
...     'name': 'ABQ_SUPER_ELASTIC_N3D',
...     'elasticity': 'user',
...     'density': 6.5e-9,
...     'depvar' : [24, (1, 'VAR1'), (3, 'DAMAGE')],
...     'constants' : [10000., 0.3, 5000, 0.3, 0.03, 6.5, 200., 300.,
...                    0.0, 6.5, 260., 100.,0., 0.03, ]
... }
>>> print (fmtMaterial (Attributes (nitinol)))
```

(continues on next page)

(continued from previous page)

```
*MATERIAL, NAME=ABQ_SUPER_ELASTIC_N3D
*DEPVAR
24
1, VAR1
3, DAMAGE
*USER MATERIAL, CONSTANTS=14
10000.0, 0.3, 5000, 0.3, 0.03, 6.5, 200.0, 300.0
0.0, 6.5, 260.0, 100.0, 0.0, 0.03
*DENSITY
6.5e-09
<BLANKLINE>
```

`plugins.fe_abq.fmtConnectorElasticity` (*elas*)
Format connector elasticity behavior.

`plugins.fe_abq.fmtConnectorStop` (*stop*)
Format connector stop behavior.

`plugins.fe_abq.elementClass` (*eltype*)
Find the general element class for Abaqus eltype

`plugins.fe_abq.fmtSolidSection` (*section, setname*)
Format a solid section for the named element set.

Parameters:

- *section*: dict: section properties
- *setname*: string: element set name for which these section properties are to be applied.

Recognized keys in the section dict:

- orientation
- thickness

See also *class:Command* for accepted parameters.

Examples

```
>>> sec1 = dict(matname='steel', thickness=0.12)
>>> from pyformex.attributes import Attributes
>>> print (fmtSolidSection(Attributes(sec1), 'section1'))
*SOLID SECTION, ELSET=section1, MATERIAL=steel
0.12
<BLANKLINE>
```

`plugins.fe_abq.fmtSolid2dSection` (*section, setname*)
Format a solid section for the named element set.

Parameters:

- *section*: dict: section properties
- *setname*: string: element set name for which these section properties are to be applied.

Recognized keys in the section dict:

- orientation
- thickness

See also *class:Command* for accepted parameters.

Examples

```
>>> sec1 = dict(matname='steel',thickness=0.12)
>>> from pyformex.attributes import Attributes
>>> print (fmtSolidSection(Attributes(sec1), 'section1'))
*SOLID SECTION, ELSET=section1, MATERIAL=steel
0.12
<BLANKLINE>
```

`plugins.fe_abq.fmtSolid3dSection` (*section*, *setname*)

Format a solid section for the named element set.

Parameters:

- *section*: dict: section properties
- *setname*: string: element set name for which these section properties are to be applied.

Recognized keys in the section dict:

- orientation
- thickness

See also *class:Command* for accepted parameters.

Examples

```
>>> sec1 = dict(matname='steel',thickness=0.12)
>>> from pyformex.attributes import Attributes
>>> print (fmtSolidSection(Attributes(sec1), 'section1'))
*SOLID SECTION, ELSET=section1, MATERIAL=steel
0.12
<BLANKLINE>
```

`plugins.fe_abq.fmtShellSection` (*section*, *setname*)

Format a SHELL SECTION keyword.

Parameters: see *func:fmtSolidSection*.

Recognized keys in the section dict:

- thickness
- offset (opt): for contact surface SPOS or 0.5, SNEG or -0.5
- transverseshearstiffness (opt):
- poisson (opt): ?? CLASH WITH MATERIAL ??
- thicknessmodulus (opt):

`plugins.fe_abq.fmtMembraneSection` (*section*, *setname*)

Format a MEMBRANE SECTION keyword.

Parameters: see *func:fmtSolidSection*.

Recognized keys in the section dict:

- thickness: float

`plugins.fe_abq.fmtSurfaceSection` (*section, setname*)

Format a SURFACE SECTION keyword.

Parameters: see *func:fmtSolidSection*.

Recognized keys in the section dict:

- density (opt): float

`plugins.fe_abq.fmtBeamSection` (*section, setname*)

Format a beam section for the named element set.

Note that there are two Beam section keywords:

- BEAM SECTION
- BEAM GENERAL SECTION: this is formatted by a separate function, currently selected if no material name is specified

Parameters: see *func:fmtSolidSection*.

The following holds for the BEAM SECTION:

Recognized keys:

- sectiontype: 'ARBITRARY', 'BOX', 'CIRC', 'HEX', 'I', 'L', 'PIPE', 'RECT', 'TRAPEZOID' or 'EL-BOW'
- material:
- data: list of tuples: data corresponding with the sectiontype. See Abaqus manual.
- transverseshearstiffness (opt):

`plugins.fe_abq.fmtGeneralBeamSection` (*section, setname*)

Format a general beam section for the named element set.

This specifies a beam section when numerical integration over the section is not required and the material constants are specified directly. See also *func:fmtBeamSection*.

Parameters: see *func:fmtSolidSection*.

Recognized keys:

- all sectiontypes:
 - sectiontype (opt): 'GENERAL' (default), 'CIRC' or 'RECT'
 - data (opt): list of tuples: section data (see Abaqus Manual). For some sections, the data may be specified by other arguments instead (see below).
 - density (opt): density of the material (required in Abaqus/Explicit)
 - transverseshearstiffness (opt):

Data specifying keys (only used if 'data' is not specified):

- sectiontype GENERAL:
 - cross_section
 - moment_inertia_11
 - moment_inertia_12
 - moment_inertia_22
 - torsional_constant

- sectiontype CIRC:
 - radius
- sectiontype RECT:
 - width, height
- sectiontype GENERAL, CIRC or RECT: - orientation (opt): a vector with the direction cosines of the 1 axis - young_modulus - shear_modulus or poisson_ratio

`plugins.fe_abq.fmtFrameSection` (*section, setname*)

Format a frame section for the named element set.

Parameters: see *func:fmtSolidSection*.

Recognized keys:

- all sectiontypes:
 - sectiontype (opt): ‘GENERAL’ (default), ‘CIRC’ or ‘RECT’
 - young_modulus
 - shear_modulus
 - density (opt): density of the material
 - yield_stress (opt): yield stress of the material
 - orientation (opt): a vector with the direction cosines of the 1 axis
- sectiontype GENERAL:
 - cross_section
 - moment_inertia_11
 - moment_inertia_12
 - moment_inertia_22
 - torsional_constant
- sectiontype CIRC:
 - radius
- sectiontype RECT:
 - width
 - height

`plugins.fe_abq.fmtTrussSection` (*section, setname*)

Format a truss section for the named element set.

Parameters: see *func:fmtSolidSection*.

Recognized keys:

- all sectiontypes:
 - sectiontype (opt): ‘GENERAL’ (default), ‘CIRC’ or ‘RECT’
 - material
- sectiontype GENERAL:
 - cross_section

- sectiontype CIRC:
 - radius
- sectiontype RECT:
 - width
 - height

`plugins.fe_abq.fmtConnectorSection` (*section, setname*)

Format a connector section.

Parameters: see *func:fmtSolidSection*.

Recognized keys:

- sectiontype: ‘JOIN’, ‘HINGE’, ...
- behavior (opt): connector behavior name
- orientation (opt): connector orientation
- elimination (opt): ‘NO’ (default), ‘YES’

`plugins.fe_abq.fmtSpringOrDashpot` (*eltype, section, setname*)

Format a section of type spring or dashpot.

Parameters (see also *func:fmtSolidSection*):

- *eltype*: ‘SPRING’ or ‘DASHPOT’

Recognized keys:

- stiffness: float: spring or dashpot stiffness. For SPRING type, this is the force per relative displacement; for DASHPOT type, the force per relative velocity.

`plugins.fe_abq.fmtSpringSection` (*section, setname*)

Shorthand for *fmtSpringOrDashpot*(“SPRING”, *section, setname*)

`plugins.fe_abq.fmtDashpotSection` (*section, setname*)

Shorthand for *fmtSpringOrDashpot*(“SPRING”, *section, setname*)

`plugins.fe_abq.fmtMassSection` (*section, setname*)

Format a section of type mass.

Parameters: see *func:fmtSolidSection*

Recognized keys:

- mass: float: mass magnitude

`plugins.fe_abq.fmtInertiaSection` (*section, setname*)

Format a section of type inertia.

Parameters: see *func:fmtSolidSection*

Recognized keys:

- inertia: list of six floats: [I11, I22, I33, I12, I13, I23]

`plugins.fe_abq.fmtRigidSection` (*section, setname*)

Format rigid body sectiontype.

Parameters: see *func:fmtSolidSection*

Recognized keys:

- refnode: string (set name) or integer (node number)

- density (opt): float:
- density (opt): float:

`plugins.fe_abq.fmtTransform` (*csys*, *setname*)
 Format a coordinate transform for the given set.

- *setname* is the name of a node set
- *csys* is a CoordSystem.

`plugins.fe_abq.fmtOrientation` (*prop*)
 Format the orientation.

Optional:

- definition
- system: coordinate system
- a: a first point
- b: a second point

`plugins.fe_abq.fmtSurface` (*prop*)
 Format the surface definitions.

Parameters:

- *prop*: a property record containing the key *surftype*.

Recognized keys:

- set: string, list of strings or list of integers.
 - string : name of an existing set.
 - list of integers: list of elements/nodes of the surface.
 - list of strings: list of existing set names.
- name: string. The surface name.
- **surftype: string. Can assume values ‘ELEMENT’ or ‘NODE’, other abaqus surface types or an empty string for special cases that do not need a**
- **label (opt): string, or a list of strings storing the abaqus face or edge identifier** It is only required for `surftype == ‘ELEMENT’`.
- options (opt): string that is added as is to the command line.

Example:

```
# This allow specifying a surface from an existing set of surface elements
P.Prop(set='quad_set' ,name='quad_surface',surftype='element',label='SPOS')

# This allow specifying a surface from already existing sets of brick elements
# using different label identifiers per each set

P.Prop(set=['hex_set1', 'hex_set2'] ,name='quad_surface',surftype='element',
↪label=['S1','S2'])

#This allows to use different identifiers for the different elements in the_
↪surface
Prop(name='mysurf',set=[0,1,2,6],surftype='element',label=['S1','S2','S1','S3'])
```

(continues on next page)

(continued from previous page)

```
will get exported to Abaqus as::

*SURFACE, NAME=mysurf, TYPE=element
 1, S1
 2, S2,
 3, S1
 7, S3
```

`plugins.fe_abq.fmtAnalyticalSurface` (*prop*)

Format the analytical surface rigid body.

Required:

- `nodeset`: refnode.
- `name`: the surface name
- `surftype`: 'ELEMENT' or 'NODE'
- `label`: face or edge identifier (only required for `surftype = 'NODE'`)

Example:

```
P.Prop(name='AnalySurf', nodeset = 'REFNOD', analyticalsurface='')
```

`plugins.fe_abq.fmtSurfaceInteraction` (*prop*)

Format the interactions.

Required:

- `name`

Optional:

- `cross_section` (for node based interaction)
- `friction` : friction coeff or 'rough'
- `surface behavior`: no separation
- `surface behavior`: pressureoverclosure

`plugins.fe_abq.fmtContact` (*prop*)

Format a contact property record.

Parameters:

- *prop*: a Property record having a key 'contact'

Recognized keys:

- `contact`: string, one of 'GENERAL' or 'PAIR'
- `interaction`: string, the name of a surface interaction

For 'GENERAL' contact:

- `include`: tuple or list of tuples (`surf1, surf2`). `surf1` and `surf2` are the names of defined surfaces or an empty string. If `surf1` is empty, a surface containing all exterior surfaces defined in the model is used. If `surf2` is empty, it is equal to `surf1` a case of self-contact results. If both are empty, self contact between all surfaces is modeled.
- `exclude` (opt): tuple or list of tuples (`surf1, surf2`). Specifies names of surfaces on which no contact is to be modeled.

For 'PAIR' contact:

- include: tuple or list of tuples (slave, master). slave and master are the names of defined surfaces. Each tuple can optionally contain one or two more values, with the orientation of the tangential slip directions for the slave, resp. master surface. They are strings with the name of defined orientations.

Examples

```
>>> PDB = PropertyDB()
>>> p1 = PDB.Prop(contact='pair', include=('s1', 'm1'), interaction='i1')
>>> print(fmtContact(p1))
*CONTACT PAIR, INTERACTION=I1
s1, m1
<BLANKLINE>
```

```
>>> p2 = PDB.Prop(contact='pair', include=[('s1', 'm1'), ('s2', 'm2')], interaction='i1
↔')
>>> print(fmtContact(p2))
*CONTACT PAIR, INTERACTION=I1
s1, m1
s2, m2
<BLANKLINE>
```

```
>>> g1 = PDB.Prop(contact='general', include=('s1', 'm1'), interaction='i1')
>>> print(fmtContact(g1))
*CONTACT
*CONTACT INCLUSIONS
s1, m1
*CONTACT PROPERTY ASSIGNMENT
s1, m1, i1
<BLANKLINE>
```

```
>>> g2 = PDB.Prop(contact='general', include=[('s1', 'm1'), ('s2', 'm2')], interaction=
↔'i1')
>>> print(fmtContact(g2))
*CONTACT
*CONTACT INCLUSIONS
s1, m1
s2, m2
*CONTACT PROPERTY ASSIGNMENT
s1, m1, i1
s2, m2, i1
<BLANKLINE>
```

`plugins.fe_abq.fmtContactPair(prop)`

Format the contact pair.

Required:

- master: master surface
- slave: slave surface
- interaction: interaction properties : name or Dict

Example:


```
P.Prop(name='contact0', interaction=Interaction(name='contactprop',
surfacebehavior=True, pressureoverclosure=['hard', 'linear', 0.0, 0.0, 0.001]),
master='quadtubeINTSURF1', slave='hexstentEXTSURF', contacttype='NODE TO SURFACE
↔')
```

`plugins.fe_abq.fmtConstraint` (*prop*)

Format Tie constraint

Required:

- name
- adjust (yes or no)
- slave
- master

Optional:

- type (surf2surf, node2surf)
- positiontolerance
- no rotation
- tiednset (it cannot be used in combination with positiontolerance)

Example:

```
P.Prop(constraint='1', name='constr1', adjust='no',
master='hexstentbarSURF', slave='hexstentEXTSURF', type='NODE TO SURFACE')
```

`plugins.fe_abq.fmtInitialConditions` (*prop*)

Format initial conditions

Required:

- type
- nodes
- data

Example:

```
P.Prop(initialcondition='', nodes='Nall', type='TEMPERATURE', data=37.)
```

`plugins.fe_abq.fmtEquation` (*prop*)

Format multi-point constraint using an equation

Required:

- equation: list of tuples (node,dof,coeff), where
 - node: is a node number or node set name
 - dof: is the number of the degree of freedom (0 based)
 - coeff: is the coefficient for this variable in the equation

Example:

```

>>> P = PropertyDB()
>>> eq1 = P.Prop(equation=[(9,1,1.), (32,2,-1.)])
>>> eq2 = P.Prop(equation=[('seta',0,1)])

The first equation forces the Z-displacement of node 32 to be equal
to the Y-displacement of node 9. The second forces the sum of the
X-displacement of all nodes in node set 'seta' to be equal to zero.

>>> print (fmtEquation(eq1)+fmtEquation(eq2))
*EQUATION
2
10, 2, 1.0
33, 3, -1.0
*EQUATION
1
seta, 1, 1

```

`plugins.fe_abq.fmtInertia` (*prop*)
Format rotary inertia

Required:

- `inertia` : inertia tensor `i11`, `i22`, `i33`, `i12`, `i13`, `i23`
- `set` : name of the element set on which inertia is applied

`plugins.fe_abq.fmtBoundary` (*prop*)
Format nodal boundary conditions.

`prop` is a node property record having the 'bound' key.

Recognized keys:

- `bound` : either of:
 - a string, representing a standard set of boundary conditions
 - a list of 6 integer values (0 or 1) corresponding with the 6 degrees of freedom `UX`, `UY`, `UZ`, `RX`, `RY`, `RZ`. The dofs corresponding to the 1's will be restrained (given a value 0.0).
 - a list of tuples (`dofid`, `value`) : this allows for nonzero boundary values to be specified. NOTE: this can also be achieved by a 'displ' keyword (see `writeDisplacements`) and that is the preferred way of specifying nonzero boundary conditions.
- `op` (opt): 'MOD' (default) or 'NEW'. By default, the boundary conditions are applied as a modification of the existing boundary conditions, i.e. initial conditions and conditions from previous steps remain in effect. The user can set `op='NEW'` to remove the previous conditions. This will remove ALL conditions of the same type.
- `ampl` (opt): string: specifies the name of an amplitude that is to be multiplied with the values to have the time history of the variable. Only relevant if `bound` specifies nonzero values. Its use is discouraged (see above).
- `options` (opt): string that is added as is to the command line.

Examples:

```

# The following are equivalent
P.nodeProp(tag='init', set='setA', name='pinned_nodes', bound='pinned')
P.nodeProp(tag='init', set='setA', name='pinned_nodes', bound=[1, 1, 1, 0, 0, 0])

```

(continues on next page)

(continued from previous page)

```
# this is possible, but discouraged
P.nodeProp(tag='init',set='setB',name='forced_displ',bound=[(1,3.0)])
# it is better to use:
P.nodeProp(tag='step0',set='setB',name='forced_displ',displ=[(1,3.0)])
```

`plugins.fe_abq.writeSection` (*fil, prop*)

Write an element section.

prop is an element property record with a section and eltype attribute

`plugins.fe_abq.writeNodes` (*fil, nodes, name='Nall', nofs=1*)

Write nodal coordinates.

The nodes are added to the named node set. If a name different from 'Nall' is specified, the nodes will also be added to a set named 'Nall'. The *nofs* specifies an offset for the node numbers. The default is 1, because Abaqus numbering starts at 1.

`plugins.fe_abq.writeElems` (*fil, elems, type, name='Eall', eid=None, eofs=1, nofs=1*)

Write element group of given type.

elems is the list with the element node numbers. The elements are added to the named element set. If a name different from 'Eall' is specified, the elements will also be added to a set named 'Eall'. The *eofs* and *nofs* specify offsets for element and node numbers. The default is 1, because Abaqus numbering starts at 1. If *eid* is specified, it contains the element numbers increased with *eofs*.

`plugins.fe_abq.writeSet` (*fil, type, name, set, ofs=1*)

Write a named set of nodes or elements (type=NSET|ELSET)

set: an ndarray. *set* can be a list of node/element numbers, in which case the *ofs* value will be added to them, or a list of names the name of another already defined set.

`plugins.fe_abq.writeDistribution` (*fil, prop*)

Write a distribution table.

Parameters:

- *prop*: a Property record having with the key *distribution*.

Recognized keys:

- **distribution**: string. Name of the distribution.
- **location**:string. can assume values 'ELEMENT','NODE' or 'NONE'
- **table**: arraylike. **The array needs to be passed as should be written in the** abaqus data line. Every row is a new line and it is in the form [element_or_node_number,data1,data2, ...]. NB For the first line used in Abaqus as default, the element_or_node_number should be an empty string.
- **format**: list of strings. **Every string is an abaqus word to be used in the distribution table.** See Abaqus documentation for allowed *words*.
- **options** (opt): string that is added as is to the command line.

The name of the distribution table (required) is derived from the name of the distribution.

`plugins.fe_abq.writeDisplacements` (*fil, prop, btype*)

Write prescribed displacements, velocities or accelerations

Parameters:

- *prop* is a list of node property records containing one (or more) of the keys 'displ', 'veloc' 'accel'.
- *btype* is the boundary type: one of 'displacement', 'velocity' or 'acceleration'

Recognized property keys:

- `displ`, `veloc`, `accel`: each is optional and is a list of tuples (`dofid`, `value`), for respectively the displacement, velocity or acceleration. A special value 'reset' may also be specified to reset the prescribed condition for these variables.
- `op` (opt): 'MOD' (default) or 'NEW'. By default, the boundary conditions are applied as a modification of the existing boundary conditions, i.e. initial conditions and conditions from previous steps remain in effect. The user can set `op='NEW'` to remove the previous conditions. This will remove ALL conditions of the same type.
- `ampl` (opt): string: specifies the name of an amplitude that is to be multiplied with the values to have the time history of the variable.
- `options` (opt): string that is added to the command line.

`plugins.fe_abq.fmtLoad(key, prop)`

Format a load input block.

Parameters:

- `key`: load type, one of: 'cload', 'dload' or 'dsload'
- `prop`: a node property record containing the specified key.

Recognized keys: - `op` (opt): string: 'MOD' or 'NEW'. See *func:writeDisplacements*. - `ampl` (opt): string: amplitude name. See *func:writeDisplacements*. - `options` (opt): string: see *func:fmtKeyWord* - `extra` (opt): string: see *func:fmtKeyWord*

For load type 'cload':

- `set`: node set on which to apply the load
- `cload`: list of tuples (`dofid`, `magnitude`)

For load type 'dload':

- `set`: element set on which to apply the load
- `dload`: ElemLoad or data

For load type 'dsload':

- `surface`: string: name of surface on which to apply the load
- `dsload`: ElemLoad or data

`plugins.fe_abq.writeAmplitude(fil, prop)`

Write Amplitude.

Parameters:

-`prop`: list of property records having an attribute *amplitude*.

Recognized keys:

- `name`: string. The name of the amplitude.
- `amplitude`: class Amplitude (see `plugins.property`).
- `options` (opt): string that is added as is to the command line.

Examples:

```
P=PropertyDB()      t=[0,1]      a=[0,0.5]      amp      =      Amplitude(data=column_stack([t,a]))
P.Prop(amplitude=amp,name='amp11',options='definition=TABULAR,smooth=0.1')
```

`plugins.fe_abq.writeNodeResult` (*fil, kind, keys, set='Nall', output='FILE', freq=1, global-axes=False, lastmode=None, summary=False, total=False*)

Write a request for nodal result output to the .fil or .dat file.

- *keys*: a list of NODE output identifiers
- *set*: a single item or a list of items, where each item is either a property number or a node set name for which the results should be written
- *output* is either FILE (for .fil output) or PRINT (for .dat output)(Abaqus/Standard only)
- *freq* is the output frequency in increments (0 = no output)

Extra arguments:

- *globalaxes*: If 'YES', the requested output is returned in the global axes. Default is to use the local axes wherever defined.

Extra arguments for output='PRINT':

- *summary*: if True, a summary with minimum and maximum is written
- *total*: if True, sums the values for each key

'Remark that the *kind* argument is not used, but is included so that we can easily call it with a *Results* dict as arguments.'

`plugins.fe_abq.writeElemResult` (*fil, kind, keys, set='Eall', output='FILE', freq=1, pos=None, summary=False, total=False*)

Write a request for element result output to the .fil or .dat file.

- *keys*: a list of ELEMENT output identifiers
- *set*: a single item or a list of items, where each item is either a property number or an element set name for which the results should be written
- *output* is either FILE (for .fil output) or PRINT (for .dat output)(Abaqus/Standard only)
- *freq* is the output frequency in increments (0 = no output)

Extra arguments:

- *pos*: Position of the points in the elements at which the results are written. Should be one of:
 - 'INTEGRATION POINTS' (default)
 - 'CENTROIDAL'
 - 'NODES'
 - 'AVERAGED AT NODES'

Non-default values are only available for ABAQUS/Standard.

Extra arguments for output='PRINT':

- *summary*: if True, a summary with minimum and maximum is written
- *total*: if True, sums the values for each key

Remark: the *kind* argument is not used, but is included so that we can easily call it with a *Results* dict as arguments

`plugins.fe_abq.writeFileOutput` (*fil, resfreq=1, timemarks=False*)

Write the FILE OUTPUT command for Abaqus/Explicit

`plugins.fe_abq.exportMesh(filename, mesh, eltype, header=)`

Export a finite element mesh in Abaqus .inp format.

This is a convenience function to quickly export a mesh to Abaqus without having to go through the whole setup of a complete finite element model. This just writes the nodes and elements specified in the mesh to the file with the specified name. If the mesh has different properties, the elements with the same properties, will be grouped in the same element set. The resulting file can then be imported in Abaqus/CAE or manual be edited to create a full model. If an eltype is specified, it will override the value stored in the mesh. This should be used to set a correct Abaqus element type matching the mesh.

6.4.11 `plugins.fe_post` — A class for holding results from Finite Element simulations.

`class plugins.fe_post.FeResult(name='__FePost__', datasize={'COORD': 3, 'S': 6, 'U': 3})`

Finite Element Results Database.

This class can hold a collection of results from a Finite Element simulation. While the class was designed for the post-processing of Abaqus (tm) results, it can be used more generally to store results from any program performing simulations over a mesh.

pyFormex comes with an included program `postabq` that scans an Abaqus .fil output file and translates it into a pyFormex script. Use it as follows:

```
postabq job.fil > job.py
```

Then execute the created script `job.py` from inside pyFormex. This will create an `FeResult` instance with all the recognized results.

The structure of the `FeResult` class very closely follows that of the Abaqus results database. There are some attributes with general info and with the geometry (mesh) of the domain. The simulation results are divided in 'steps' and inside each step in 'increments'. Increments are usually connected to incremental time and so are often the steps, though it is up to the user to interpret the time. Steps could just as well be different unrelated simulations performed over the same geometry.

In each step/increment result block, individual values can be accessed by result codes. The naming mostly follows the result codes in Abaqus, but components of vector/tensor values are number starting from 0, as in Python and pyFormex.

Result codes:

- *U*: displacement vector
- *U0, U1, U2* : x, y, resp. z-component of displacement
- *S*: stress tensor
- *S0 .. S5*: components of the (symmetric) stress tensor: 0..2 : x, y, z normal stress 3..5 : xy, yz, zx shear stress

Increment (*step, inc, **kargs*)

Add a new step/increment to the database.

This method can be used to add a new increment to an existing step, or to add a new step and set the initial increment, or to just select an existing step/inc combination. If the step/inc combination is new, a new empty result record is created. The result record of the specified step/inc becomes the current result.

Export ()

Align on the last increment and export results

do_nothing (**kargs)

A do nothing function to stand in for as yet undefined functions.

TotalEnergies (**kargs)

A do nothing function to stand in for as yet undefined functions.

OutputRequest (**kargs)

A do nothing function to stand in for as yet undefined functions.

Coordinates (**kargs)

A do nothing function to stand in for as yet undefined functions.

Displacements (**kargs)

A do nothing function to stand in for as yet undefined functions.

Unknown (**kargs)

A do nothing function to stand in for as yet undefined functions.

setStepInc (step, inc=1)

Set the database pointer to a given step,inc pair.

This sets the step and inc attributes to the given values, and puts the corresponding results in the R attribute. If the step.inc pair does not exist, an empty results dict is set.

getSteps ()

Return all the step keys.

getIncs (step)

Return all the incs for given step.

nextStep ()

Skips to the start of the next step.

nextInc ()

Skips to the next increment.

The next increment is either the next increment of the current step, or the first increment of the next step.

prevStep ()

Skips to the start of the previous step.

prevInc ()

Skips to the previous increment.

The previous increment is either the previous increment of the current step, or the last increment of the previous step.

getres (key, domain='nodes')

Return the results of the current step/inc for given key.

The key may include a component to return only a single column of a multicolumn value.

printSteps ()

Print the steps/increments/resultcodes for which we have results.

6.4.12 plugins.flavia —

(C) 2010 Benedict Verhegghe.

Functions defined in module `plugins.flavia`

`plugins.flavia.readMesh` (*fn*)
Read a flavia mesh file.
Returns a list of Meshes if succesful.

`plugins.flavia.readCoords` (*fil, ndim*)
Read a set of coordinates from a flavia file

`plugins.flavia.readElems` (*fil, nplex*)
Read a set of coordinates from a flavia file

`plugins.flavia.readResults` (*fn, nnodes, ndim*)
Read a flavia results file for an ndim mesh.

`plugins.flavia.readResult` (*fil, nvalues, nres*)
Read a set of results from a flavia file

`plugins.flavia.createFeResult` (*model, results*)
Create an FeResult from meshes and results

`plugins.flavia.readFlavia` (*meshfile, resfile*)
Read flavia results files
Currently we only read matching pairs of meshfile,resfile files.

6.4.13 `plugins.imagearray` — Convert bitmap images into numpy arrays.

This module contains functions to convert bitmap images into numpy arrays and vice versa. There are functions to read image from file into arrays, and to save image arrays to files. There is even a class that reads a full stack of Dicom images into a 3D numpy array.

Some of this code was based on ideas found on the PyQwt mailing list.

Classes defined in module `plugins.imagearray`

class `plugins.imagearray.DicomStack` (*files, zsort=True, reverse=False*)
A stack of DICOM images.

Note: This class is only available if you have pydicom installed.

The `DicomStack` class stores a collection of DICOM images and provides conversion to `ndarray`.

The `DicomStack` is initialized by a list of file names. All input files are scanned for DICOM image data and non-DICOM files are skipped. From the DICOM files, the pixel and slice spacing are read, as well as the origin. The DICOM files are sorted in order of ascending `origin_z` value.

While reading the DICOM files, the following attributes are set:

- *files*: a list of the valid DICOM files, in order of ascending z-value.
- *pixel_spacing*: a dict with an (x,y) pixel spacing tuple as key and a list of indices of the matching images as value.
- *slice_thickness*: a list of the slice thicknesses of the images, in the same order as *files*.
- *xy_origin*: a dict with an (x,y) position the pixel (0,0) as key and a list of indices of the matching images as value.

- *z_origin*: a list of the z positions of the images, in the same order as *files*.
- *rejected*: list of rejected files.

class Object

_A dummy object to allow attributes

nfiles ()

Return the number of accepted files.

files ()

Return the list of accepted filenames

rejected ()

Return the list of rejected filenames

pspacing ()

Return the pixel spacing and slice thickness.

Return the pixel spacing (x,y) and the slice thickness (z) in the accepted images.

Returns: a float array of shape (nfiles,3).

origin ()

Return the origin of all accepted images.

Return the world position of the pixel (0,0) in all accepted images.

Returns: a float array of shape (nfiles,3).

zvalues ()

Return the zvalue of all accepted images.

The zvalue is the z value of the origin of the image.

Returns: a float array of shape (nfiles).

zspacing ()

Check for constant slice spacing.

spacing ()

Return uniform spacing parameters, if uniform.

image (i)

Return the image at index i

pixar (raw=False)

Return the DicomStack as an array

Returns all images in a single array. This is only succesful if all accepted images have the same size.

Functions defined in module plugins.imagearray

plugins.imagearray.**image2array** (*filename, mode=None, flip=True*)

Read an image file into a numpy array.

Parameters

- **filename** (*path_like*) – The name of the file containing the image.
- **mode** (*str, optional*) – If provided, the image will be converted to the specified mode. The mode is a string defining the color channels to be returned. Typical values are '1' (black/white), 'L' (grayscale), 'LA' (gray with alpha), 'RGB' (three colors), 'RGBA' (three colors plus alpha). Default is to return the image as it is stored.

- **flip** (*bool*) – If True, the vertical axis will be inverted. This is the default, because image scanlines are stored from top to bottom, while opengl uses a vertical axis running from bottom to top. Set this to False when using images as fonts in `FontTexture`, because the `FontTexture` engine itself takes account of the top-down image storage.

`plugins.imagearray.array2image` (*ar, filename*)

Save an image stored in a numpy array to file.

Parameters

- **ar** (*array*) – Array holding the pixel data. This is typically an Int8 type array with shape (height, width, 3) where the last axis holds the RGB color data.
- **filename** (*path_like*) – The name of the file where the image is to be saved.

`plugins.imagearray.resizeImage` (*image, w=0, h=0*)

Load and optionally resize a QImage.

Parameters

- **image** (*qimage_like*) – QImage, or data that can be converted to a QImage, e.g. the name of a raster image file.
- **w** (*int, optional*) – If provided and >0, the image will be resized to this width.
- **h** (*int, optional*) – If provided and >0, the image will be resized to this height.

Returns *QImage* – A QImage with the requested size (if provided).

`plugins.imagearray.qimage2numpy` (*image, resize=(0, 0), order='RGBA', flip=True, indexed=None*)

Transform a QImage to a `numpy.ndarray`.

Parameters

- **image** (*qimage_like*) – A QImage or any data that can be converted to a QImage, e.g. the name of an image file, in any of the formats supported by Qt. The image can be a full color image or an indexed type. Only 32bit and 8bit images are currently supported.
- **resize** (*tuple of ints*) – A tuple of two integers (width,height). Positive value will force the image to be resized to this value.
- **order** (*str*) – String with a permutation/subset of the characters 'RGBA', defining the order in which the colors are returned. Default is RGBA, so that `result[...0]` gives the red component. Note however that QImage stores in ARGB order. You may also specify a subset of the 'RGBA' characters, in which case you will only get some of the color components. An often used value is 'RGB' to get the colors without the alpha value.
- **flip** (*bool*) – If True, the image scanlines are flipped upside down. This is practical because image files are usually stored in top down order, while OpenGL uses an upwards positive direction, requiring a flip to show the image upright.
- **indexed** (*bool or None.*) – If True, the result will be an indexed image where each pixel color is an index into a color table. Non-indexed image data will be converted.

If False, the result will be a full color array specifying the color of each pixel. Indexed images will be expanded to a full color array.

If None (default), no conversion is done and the resulting data are dependent on the image format. In all cases both a color and a colortable will be returned, but the latter will be None for non-indexed images.

Returns

- **colors** (*array*) – An int8 array with shape (height,width,ncomp), holding the components of the color of each pixel. Order and number of components is as specified by the `order` argument. The default is 4 components in order ‘RGBA’.

This value is only returned if the image was not an indexed type or if `indexed=False` was specified, in which case indexed images will be converted to full color.

- **colorindex** (*array*) – An int array with shape (height,width) holding color indices into `colortable`, which stores the actual color values.

This value is only returned if the image was an indexed type or if `indexed=True` was specified, in which case nonindexed images will be converted to using color index and table.

- **colortable** (*array or None.*) – An int8 array with shape (ncolors,ncomp). This array stores all the colors used in the image, in the order specified by `order`

This value is only returned if `indexed` is not `False`. Its value will be `None` if `indexed` is `None` (default) and the image was not indexed.

Notes

This table summarizes the return values for each of the possible values of the `indexed` argument combined with indexed or nonindexed image data:

arg	non-indexed image	indexed image
<code>indexed=None</code>	colors, None	colorindex, colortable
<code>indexed=False</code>	colors	colors
<code>indexed=True</code>	colorindex, colortable	colorindex, colortable

`plugins.imagearray.numpy2qimage` (*array*)

Convert a 2D or 3D integer numpy array into a QImage

Parameters **array** (*2D or 3D int array*) – If the input array is 2D, the array is converted into a gray image. If the input array is 3D, the last axis should have length 3 or 4 and represents the color channels in order RGB or RGBA.

Notes

This is equivalent to calling `gray2qimage()` for a 2D array and `rgb2qimage()` for a 3D array.

`plugins.imagearray.gray2qimage` (*gray*)

Convert a 2D numpy array to gray QImage.

Parameters **gray** (*uint8 array (height,width)*) – Array with the grey values of an image with size (height,width).

Returns *QImage* – A QImage with the corresponding gray image. The image format will be indexed.

`plugins.imagearray.rgb2qimage` (*rgb*)

Convert a 3D numpy array into a 32-bit QImage.

Parameters **rgb** (*int array (height,width,ncomp)*) – Int array with the pixel values of the image. The data can have 3 (RGB) or 4 (RGBA) components.

Returns *QImage* – A QImage with size (height,width) in the format RGB32 (3 components) or ARGB32 (4 components).

`plugins.imagearray.qimage2glcolor (image, resize=(0, 0))`

Convert a bitmap image to corresponding OpenGL colors.

Parameters

- **image** (*qimage_like*) – A QImage or any data that can be converted to a QImage, e.g. the name of an image file, in any of the formats supported by Qt. The image can be a full color image or an indexed type. Only 32bit and 8bit images are currently supported.
- **resize** (*tuple of ints*) – A tuple of two integers (width,height). Positive value will force the image to be resized to this value.

Returns *float array (w,h,3)* – Array of float values in the range 0.0 to 1.0, containing the OpenGL colors corresponding to the image RGB colors. By default the image is flipped upside-down because the vertical OpenGL axis points upwards, while bitmap images are stored downwards.

`plugins.imagearray.removeAlpha (qim)`

Remove the alpha component from a QImage.

Directly saving a QImage grabbed from the OpenGL buffers always results in an image with transparency. See <https://savannah.nongnu.org/bugs/?36995> .

This function will remove the alpha component from the QImage, so that it can be saved with opaque objects.

Note: we did not find a way to do this directly on the QImage, so we go through a conversion to a numpy array and back.

`plugins.imagearray.saveGreyImage (a, f, flip=True)`

Save a 2D int array as a grey image.

Parameters

- **a** (*int array*) – Int array (height,width) with values in the range 0..255. These are the grey values of the pixels.
- **f** (*str*) – Name of the file to write the image to.
- **flip** (*bool*) – If True (default), the vertical axis is flipped, so that images are stored starting at the top. If your data already have the vertical axis downwards, use flip=False.

Note: This stores the image as an RGB image with equal values for all three color components.

`plugins.imagearray.dicom2numpy (files)`

Easy conversion of a set of dicom images to a numpy array.

Parameters **files** (list of *path_like*) – List of file names containing the subsequent DICOM images in the stack. The file names should be in the correct order.

Returns

- **pixar** (*int array (nfiles, height, width)*) – Pixel array with the nfiles images.
- *spacing*

6.4.14 `plugins.isopar` — Isoparametric transformations

Classes defined in module `plugins.isopar`

class `plugins.isopar.Isopar` (*eltype, coords, oldcoords*)

A class representing an isoparametric transformation

`eltype` is one of the keys in `Isopar.isodata` `coords` and `oldcoords` can be either arrays, `Coords` or `Formex` instances, but should be of equal shape, and match the number of atoms in the specified transformation type

The following three formulations are equivalent

```
trf = Isopar(eltype, coords, oldcoords)
G = F.isopar(trf)

trf = Isopar(eltype, coords, oldcoords)
G = trf.transform(F)

G = isopar(F, eltype, coords, oldcoords)
```

transform(*X*)

Apply isoparametric transform to a set of coordinates.

Returns a `Coords` array with same shape as *X*

Functions defined in module `plugins.isopar`

`plugins.isopar.evaluate` (*atoms*, *x*, *y=0*, *z=0*)

Build a matrix of functions of `coords`.

- *atoms*: a list of text strings representing a mathematical function of *x*, and possibly of *y* and *z*.
- *x*, *y*, *z*: a list of *x*- (and optionally *y*-, *z*-) values at which the *atoms* will be evaluated. The lists should have the same length.

Returns a matrix with *nvalues* rows and *natoms* columns.

`plugins.isopar.exponents` (*n*, *layout='lag'*)

Create tuples of polynomial exponents.

This function creates the exponents of polynomials in 1 to 3 dimensions which can be used to construct interpolation function over lagrangian, triangular or serendipity grids.

Parameters:

- *n*: a tuple of 1 to 3 integers, specifying the degree of the polynomials in the *x* up to *z* directions. For a lagrangian layout, this is one less than the number of points in each direction.
- *layout*: string, specifying the layout of grid and the selection of monomials to be used. Should be one of 'lagrangian', 'triangular', 'serendipity' or 'border'. The string can be abbreviated to its first 3 characters.

Returns an integer array of shape (ndim,npoints), where ndim = len(n) and npoints depends on the layout:

- lagrangian: npoints = prod(n). The point layout is a rectangular lagrangian grid form by *n*[*i*] points in direction *i*. As an example, specifying *n*=(3,2) uses a grid of 3 points in *x*-direction and 2 points in *y*-direction.
- triangular: requires that all values in *n* are equal. For ndim=2, the number of points is $n*(n+1)//2$.
- border: this is like the lagrangian grid with all internal points removed. For ndim=2, we have npoints = $2 * \text{sum}(n) - 4$. For ndim=3 we have npoints = $2 * \text{sum}(nx*ny+ny*nz+nz*nx) - 4 * \text{sum}(n) + 8$. Thus *n*=(3,3,3) will yield $2*3*3*3 - 4*(3+3+3) + 8 = 26$
- serendipity: tries to use only the corner and edge nodes, but uses a convex domain of the monomials. This may require some nodes inside the faces or the volume. Currently works up to (4,4) in 2D or (3,3,3) in 3D.

`plugins.isopar.interpoly` (*n*, *layout='lag'*)

Create an interpolation polynomial

parameters are like for exponents.

Returns a Polynomial that can be used for interpolation over the element.

6.4.15 `plugins.isosurface` — Isosurface: surface reconstruction algorithms

This module contains the marching cube algorithm.

Some of the code is based on the example by Paul Bourke from <http://paulbourke.net/geometry/polygonise/>

Functions defined in module `plugins.isosurface`

`plugins.isosurface.isosurface` (*data*, *level*, *nproc=-1*, *tet=0*)

Create an isosurface through data at given level.

- *data*: (nx,ny,nz) shaped array of data values at points with coordinates equal to their indices. This defines a 3D volume [0,nx-1], [0,ny-1], [0,nz-1]
- *level*: data value at which the isosurface is to be constructed
- *nproc*: number of parallel processes to use. On multiprocessor machines this may be used to speed up the processing. If ≤ 0 , the number of processes will be set equal to the number of processors, to achieve a maximal speedup.

Returns an (ntr,3,3) array defining the triangles of the isosurface. The result may be empty (if level is outside the data range).

`plugins.isosurface.isoline` (*data*, *level*, *nproc=-1*)

Create an isocontour through data at given level.

- *data*: (nx,ny,nz) shaped array of data values at points with coordinates equal to their indices. This defines a 2D area [0,nx-1], [0,ny-1]
- *level*: data value at which the isocontour is to be constructed
- *nproc*: number of parallel processes to use. On multiprocessor machines this may be used to speed up the processing. If ≤ 0 , the number of processes will be set equal to the number of processors, to achieve a maximal speedup.

Returns an (nseg,2,2) array defining the 2D coordinates of the segments of the isocontour. The result may be empty (if level is outside the data range).

6.4.16 `plugins.lima` — Lindenmayer Systems

Classes defined in module `plugins.lima`

class `plugins.lima.Lima` (*axiom=""*, *rules={}*)

A class for operations on Lindenmayer Systems.

status ()

Print the status of the Lima

addRule (*atom*, *product*)

Add a new rule (or overwrite an existing)

translate (*rule, keep=False*)

Translate the product by the specified rule set.

If `keep=True` is specified, atoms that do not have a translation in the rule set, will be kept unchanged. The default (`keep=False`) is to remove those atoms.

Functions defined in module `plugins.lima`

`plugins.lima.lima` (*axiom, rules, level, turtlecmds, glob=None*)

Create a list of connected points using a Lindenmayer system.

`axiom` is the initial string, `rules` are translation rules for the characters in the string, `level` is the number of generations to produce, `turtlecmds` are the translation rules of the final string to turtle cmds, `glob` is an optional list of globals to pass to the turtle script player.

This is a convenience function for quickly creating a drawing of a single generation member. If you intend to draw multiple generations of the same Lima, it is better to use the `grow()` and `translate()` methods directly.

6.4.17 `plugins.neu_exp` — Gambit neutral file exporter.

This module contains some functions to export pyFormex mesh models to Gambit neutral files.

Functions defined in module `plugins.neu_exp`

`plugins.neu_exp.writeHeading` (*fil, nodes, elems, nbsets=0, heading=""*)

Write the heading of the Gambit neutral file.

nbsets: number of boundary condition sets (border patches).

`plugins.neu_exp.writeNodes` (*fil, nodes*)

Write nodal coordinates.

`plugins.neu_exp.writeElems` (*fil, elems*)

Write element connectivity.

`plugins.neu_exp.writeGroup` (*fil, elems*)

Write group of elements.

`plugins.neu_exp.writeBCsets` (*fil, bcsets, elgeotype*)

Write boundary condition sets of faces.

Parameters:

- *bcsets*: a dict where the values are `BorderFace` arrays (see below).
- *elgeotype*: element geometry type: 4 for hexahedrons, 6 for tetrahedrons.

`BorderFace` array: A set of border faces defined as a (n,2) shaped int array: each row contains an element number (`enr`) and face number (`fnr`).

There are 2 ways to construct the `BorderFace` arrays:

find border both as mesh and `enr/fnr` and keep correspondence:

```
brde, brdfaces = M.getFreeEntities(level=-1, return_indices=True)
brd = Mesh(M.coords, brde)

.. note: This needs further explanation. Gianluca?
```

matchFaces: Given a volume mesh **M** and a surface meshes **S**, being (part of) the border of **M**, Border-Face array for the surface **S** can be obtained from:

```
bf = M.matchFaces(S)[1]
```

To define other boundary types: Value - Boundary Entity Type,

0 UNSPECIFIED, 1 AXIS, 2 CONJUGATE, 3 CONVECTION, 4 CYCLIC, 5 DEAD, 6 ELEMENT_SIDE, 7 ESPECIES, 8 EXHAUST_FAN, 9 FAN, 10 FREE_SURFACE, 11 GAP, 12 INFLOW, 13 INLET, 14 INLET_VENT, 15 INTAKE_FAN, 16 INTERFACE, 17 INTERIOR, 18 INTERNAL, 19 LIVE, 20 MASS_FLOW_INLET, 21 MELT, 22 MELT_INTERFACE, 23 MOVING_BOUNDARY, 24 NODE, 25 OUTFLOW, 26 OUTLET, 27 OUTLET_VENT, 28 PERIODIC, 29 PLOT, 30 POROUS, 31 POROUS_JUMP, 32 PRESSURE, 33 PRESSURE_FAR_FIELD, 34 PRESSURE_INFLOW, 35 PRESSURE_INLET, 36 PRESSURE_OUTFLOW, 37 PRESSURE_OUTLET, 38 RADIATION, 39 RADIATOR , 40 RECIRCULATION_INLET, 41 RECIRCULATION_OUTLET, 42 SLIP, 43 SREACTION, 44 SURFACE, 45 SYMMETRY, 46 TRACTION, 47 TRAJECTORY, 48 VELOCITY, 49 VELOCITY_INLET, 50 VENT, 51 WALL, 52 SPRING

See also http://combust.hit.edu.cn:8080/fluent/Gambit13_help/modeling_guide/mg0b.htm#mg0b01 for the description of the neu file syntax.

`plugins.neu_exp.write_neu` (*fil, mesh, bcsets=None, heading='generated with pyFormex'*)

Export a mesh as .neu file (For use in Gambit/Fluent)

- *fil*: file name
- *mesh*: pyFormex Mesh
- *heading*: heading text to be shown in the gambit header
- *bcsets*: dictionary of 2D arrays: {'name1': brdfaces1, ... }, see writeBCsets

6.4.18 plugins.nurbs — Using NURBS in pyFormex.

The `nurbs` module defines functions and classes to manipulate NURBS curves and surface in pyFormex.

Classes defined in module plugins.nurbs

class `plugins.nurbs.Coords4`

A collection of points represented by their homogeneous coordinates.

While most of the pyFormex implementation is based on the 3D Cartesian coordinates class `Coords`, some applications may benefit from using homogeneous coordinates. The class `Coords4` provides some basic functions and conversion to and from cartesian coordinates. Through the conversion, all other pyFormex functions, such as transformations, are available.

`Coords4` is implemented as a float type `numpy.ndarray` whose last axis has a length equal to 4. Each set of 4 values (x,y,z,w) along the last axis represents a single point in 3D space. The cartesian coordinates of the point are obtained by dividing the first three values by the fourth: (x/w, y/w, z/w). A zero w-value represents a point at infinity. Converting such points to `Coords` will result in Inf or NaN values in the resulting object.

The float datatype is only checked at creation time. It is the responsibility of the user to keep this consistent throughout the lifetime of the object.

Just like `Coords`, the class `Coords4` is derived from `numpy.ndarray`.

Parameters:

data: *array_like* If specified, data should evaluate to an array of floats, with the length of its last axis not larger than 4. When equal to four, each tuple along the last axis represents a single point in homogeneous coordinates. If smaller than four, the last axis will be expanded to four by adding values zero in the second and third position and values 1 in the last position. If no data are given, a single point (0,0,0) will be created.

w: *array_like* If specified, the w values are used to denormalize the homogeneous data such that the last component becomes w.

dtyp: **data-type** The datatype to be used. If not specified, the datatype of *data* is used, or the default `Float` (which is equivalent to `numpy.float32`).

copy: **boolean** If `True`, the data are copied. By default, the original data are used if possible, e.g. if a correctly shaped and typed `numpy.ndarray` is specified.

normalize ()

Normalize the homogeneous coordinates.

Two sets of homogeneous coordinates that differ only by a multiplicative constant refer to the same points in cartesian space. Normalization of the coordinates is a way to make the representation of a single point unique. Normalization is done so that the last component (w) is equal to 1.

The normalization of the coordinates is done in place.

Warning: Normalizing points at infinity will result in Inf or NaN values.

deNormalize (w)

Denormalizes the homogeneous coordinates.

This multiplies the homogeneous coordinates with the values w. w normally is a constant or an array with shape `self.shape[:-1] + (1,)`. It then multiplies all 4 coordinates of a point with the same value, thus resulting in a denormalization while keeping the position of the point unchanged.

The denormalization of the coordinates is done in place. If the `Coords4` object was normalized, it will have precisely w as its 4-th coordinate value after the call.

toCoords ()

Convert homogeneous coordinates to cartesian coordinates.

Returns:

A `Coords` object with the cartesian coordinates of the points. Points at infinity (w=0) will result in Inf or NaN value. If there are no points at infinity, the resulting `Coords` point set is equivalent to the `Coords4` one.

npoints ()

Return the total number of points.

ncoords ()

Return the total number of points.

x ()

Return the x-plane

y ()

Return the y-plane

z ()

Return the z-plane

w()

Return the w-plane

bbox()

Return the bounding box of a set of points.

Returns the bounding box of the cartesian coordinates of the object.

actor(kargs)**

Graphical representation

class plugins.nurbs.**Geometry4**

This is a preliminary class intended to provide some transforms in 4D

class plugins.nurbs.**KnotVector** (*data=None, val=None, mul=None*)

A knot vector

A knot vector is sequence of float values sorted in ascending order. Values can occur multiple times. In they typical use case for this class (Nurbs) most values do indeed occur multiple times, and the multiplicity of the values is an important quantity. Therefore, the knot vector is stored in two arrays of the same length:

- *v*: the unique float values, a strictly ascending sequence
- *m*: the multiplicity of each of the values

Example:

```
>>> K = KnotVector([0.,0.,0.,0.5,0.5,1.,1.,1.])
>>> print(K.val)
[ 0.  0.5  1. ]
>>> print(K.mul)
[3 2 3]
>>> print(K)
KnotVector: 0.0(3), 0.5(2), 1.0(3)
>>> print([(v,m) for v,m in zip(K.val,K.mul)])
[(0.0, 3), (0.5, 2), (1.0, 3)]
>>> print(K.values())
[ 0.  0.  0.  0.5  0.5  1.  1.  1. ]
>>> K.index(0.5)
1
>>> K.span(1.0)
5
>>> K.mult(0.5)
2
>>> K.mult(0.7)
0
>>> K[4],K[-1]
(0.5, 1.0)
>>> K[4:6]
array([ 0.5,  1. ])
```

nknots()

Return the total number of knots

values()

Return the full list of knot values

index(u)

Find the index of knot value u.

If the value does not exist, a ValueError is raised.

mult (*u*)

Return the multiplicity of knot value *u*.

Returns an int with the multiplicity of the knot value *u*, or 0 if the value is not in the `KNotVector`.

span (*u*)

Find the (first) index of knot value *u* in the full knot values vector.

If the value does not exist, a `ValueError` is raised.

copy ()

Return a copy of the `KnotVector`.

Changing the copy will not change the original.

reverse ()

Return the reverse knot vector.

Example:

```
>>> print(KnotVector([0,0,0,1,3,6,6,8,8,8]).reverse().values())
[ 0.  0.  0.  2.  2.  5.  7.  8.  8.  8.]
```

class `plugins.nurbs.NurbsCurve` (*control*, *degree=None*, *wts=None*, *knots=None*, *closed=False*, *blended=True*)

A NURBS curve

The Nurbs curve is defined by `nctrl` control points, a degree (≥ 1) and a knot vector with `nknots = nctrl+degree+1` parameter values.

Parameters:

- *control*: Coords-like (`nctrl,3`): the vertices of the control polygon.
- *degree*: int: the degree of the Nurbs curve. If not specified, it is derived from the length of the knot vector (*knots*).
- *wts*: float array (`nctrl`): weights to be attributed to the control points. Default is to attribute a weight 1.0 to all points. Using different weights allows for more versatile modeling (like perfect circles and arcs.)
- *knots*: `KnotVector` or an ascending list of `nknots` float values. The values are only defined upon a multiplicative constant and will be normalized to set the last value to 1. If *degree* is specified, default values are constructed automatically by calling `genKnotVector()`. If no knots are given and no degree is specified, the degree is set to the `nctrl-1` if the curve is blended. If not blended, the degree is not set larger than 3.
- *closed*: bool: determines whether the curve is closed. Default `False`. The use of closed `NurbsCurves` is currently very limited.
- *blended*: bool: determines that the curve is blended. Default is `True`. Set `blended==False` to define a nonblended curve. A nonblended curve is a chain of independent curves, Bezier curves if the weights are all ones. See also `decompose()`. The number of control points should be a multiple of the degree, plus one. This parameter is only used if no knots are specified.

knots

Return the full list of knot values

nctrl ()

Return the number of control points

nknots ()

Return the number of knots

order ()

Return the order of the Nurbs curve

urange ()

Return the parameter range on which the curve is defined.

Returns a (2,) float array with the minimum and maximum parameter value for which the curve is defined.

isClamped ()

Return True if the NurbsCurve uses a clamped knot vector.

A clamped knot vector has a multiplicity $p+1$ for the first and last knot. All our generated knot vectors are clamped.

isUniform ()

Return True if the NurbsCurve has a uniform knot vector.

A uniform knot vector has a constant spacing between the knot values.

isRational ()

Return True if the NurbsCurve is rational.

The curve is rational if the weights are not constant. The curve is polygonal if the weights are constant.

Returns True for a rational curve, False for a polygonal curve.

isBlended ()

Return True if the NurbsCurve is blended.

An clamped NurbsCurve is unblended (or decomposed) if it consists of a chain of independent Bezier curves. Such a curve has multiplicity p for all internal knots and $p+1$ for the end knots of an open curve. Any other NurbsCurve is blended.

Returns True for a blended curve, False for an unblended one.

Note: for testing whether an unclamped curve is blended or not, first clamp it.

bbox ()

Return the bounding box of the NURBS curve.

copy ()

Return a (deep) copy of self.

Changing the copy will not change the original.

pointsAt (u)

Return the points on the Nurbs curve at given parametric values.

Parameters:

- *u*: (nu,) shaped float array, parametric values at which a point is to be placed. Note that valid points are only obtained for parameter values in the range `self.range()`.

Returns (nu,3) shaped Coords with nu points at the specified parametric values.

derivs (u, d=1)

Returns the points and derivatives up to *d* at parameter values *u*

Parameters:

- *u*: either of:
 - int: number of points (npts) at which to evaluate the points and derivatives. The points will be equally spaced in parameter space.
 - float array (npts): parameter values at which to compute points and derivatives.

- *d*: int: highest derivative to compute.

Returns a float array of shape (d+1,npts,3).

frenet (*u*)

Compute Frenet vectors, curvature and torsion at parameter values *u*

Parameters:

- *u*: either of:
 - int: number of points (npts) at which to evaluate the points and derivatives. The points will be equally spaced in parameter space.
 - float array (npts): parameter values at which to compute points and derivatives.

Returns a float array of shape (d+1,npts,3).

Returns a tuple of arrays at *nu* parameter values *u*:

- *T*: normalized tangent vector (nu,3)
- *N*: normalized normal vector (nu,3)
- *B*: normalized binormal vector (nu,3)
- *k*: curvature of the curve (nu)
- *t*: torsion of the curve (nu)

curvature (*u*, *torsion=False*)

Compute Frenet vectors, curvature and torsion at parameter values *u*

Parameters:

- *u*: either of:
 - int: number of points (npts) at which to evaluate the points and derivatives. The points will be equally spaced in parameter space.
 - float array (npts): parameter values at which to compute points and derivatives.
- *torsion*: bool. If True, also returns the torsion in the curve.

If *torsion* is False (default), returns a float array with the curvature at parameter values *u*. If *torsion* is True, also returns a float array with the torsion at parameter values *u*.

knotPoints (*multiple=False*)

Returns the points at the knot values.

If *multiple* is True, points are returned with their multiplicity. The default is to return all points just once.

insertKnots (*u*)

Insert a set of knots in the Nurbs curve.

u is a vector with knot parameter values to be inserted into the curve. The control points are adapted to keep the curve unchanged.

Returns:

A Nurbs curve equivalent with the original but with the specified knot values inserted in the knot vector, and the control points adapted.

requireKnots (*val*, *mul*)

Insert knots until the required multiplicity reached.

Inserts knot values only if they are currently not there or their multiplicity is lower than the required one.

Parameters:

- *val*: list of float (nval): knot values required in the knot vector.
- *mul*: list of int (nval): multiplicities required for the knot values *u*.

Returns:

A Nurbs curve equivalent with the original but where the knot vector is guaranteed to contain the values in *u* with at least the corresponding multiplicity in *m*. If all requirements were already fulfilled at the beginning, returns self.

subCurve (*u1*, *u2*)

Extract the subcurve between parameter values *u1* and *u2*

Parameters:

- *u1*, *u2*: two parameter values (*u1* < *u2*), delimiting the part of the curve to extract. These values do not have to be knot values.

Returns a NurbsCurve containing only the part between *u1* and *u2*.

clamp ()

Clamp the knot vector of the curve.

A clamped knot vector starts and ends with multiplicities *p*-1. See also *isClamped()*.

Returns self if the curve is already clamped, else returns an equivalent curve with clamped knot vector.

Note: The use of unclamped knot vectors is deprecated. This method is provided only as a convenient method to import curves from legacy systems using unclamped knot vectors.

unclamp ()

Unclamp the knot vector of the curve.

An unclamped knot vector starts and ends with multiplicities *p*-1. See also *isClamped()*.

Returns self if the curve is already clamped, else returns an equivalent curve with clamped knot vector.

Note: The use of unclamped knot vectors is deprecated. This method is provided as a convenient method to export curves to legacy systems that only handle unclamped knot vectors.

unblend ()

Decomposes a curve in subsequent Bezier curves.

Returns an equivalent unblended Nurbs.

See also *toBezier()*

decompose ()

Decomposes a curve in subsequent Bezier curves.

Returns an equivalent unblended Nurbs.

See also *toBezier()*

toCurve (*force_Bezier=False*)

Convert a (nonrational) NurbsCurve to a BezierSpline or PolyLine.

This decomposes the curve in a chain of Bezier curves and converts the chain to a BezierSpline or PolyLine.

This only works for nonrational NurbsCurves, as the BezierSpline and PolyLine classes do not allow homogeneous coordinates required for rational curves.

Returns a BezierSpline or PolyLine (if degree is 1) that is equivalent with the NurbsCurve.

See also *unblend()* which decomposes both rational and nonrational NurbsCurves.

toBezier()

Convert a (nonrational) NurbsCurve to a BezierSpline.

This is equivalent with toCurve(force_Bezier=True) and returns a BezierSpline in all cases.

removeKnot (*u, m, tol=1e-05*)

Remove a knot from the knot vector of the Nurbs curve.

u: knot value to remove *m*: how many times to remove (if negative, remove maximally)

Returns:

A Nurbs curve equivalent with the original but with a knot vector where the specified value has been removed *m* times, if possible, or else as many times as possible. The control points are adapted accordingly.

removeAllKnots (*tol=1e-05*)

Remove all removable knots

Parameters:

- *tol*: float: acceptable error (distance between old and new curve).

Returns an equivalent (if *tol* is small) NurbsCurve with all extraneous knots removed.

blend (*tol=1e-05*)

Remove all removable knots

Parameters:

- *tol*: float: acceptable error (distance between old and new curve).

Returns an equivalent (if *tol* is small) NurbsCurve with all extraneous knots removed.

elevateDegree (*t=1*)

Elevate the degree of the Nurbs curve.

t: how much to elevate the degree

Returns:

A Nurbs curve equivalent with the original but of a higher degree.

reduceDegree (*t=1*)

Reduce the degree of the Nurbs curve.

t: how much to reduce the degree (max. = degree-1)

Returns:

A Nurbs curve approximating the original but of a lower degree.

projectPoint (*P, eps1=1e-05, eps2=1e-05, maxit=20, nseed=20*)

Project a given point on the Nurbs curve.

This can also be used to determine the parameter value of a point lying on the curve.

Parameters:

- *P*: Coords-like (npts,3): one or more points in space.

Returns a tuple (u,X):

- *u*: float: parameter value of the base point X of the projection of P on the NurbsCurve.
- *X*: Coords (3,): the base point of the projection of P on the NurbsCurve.

The algorithm is based on the from The Nurbs Book.

approx (*ndiv=None, nseg=None, **kargs*)

Return a PolyLine approximation of the Nurbs curve

If no *nseg* is given, the curve is approximated by a PolyLine through equidistant *ndiv+1* point in parameter space. These points may be far from equidistant in Cartesian space.

If *nseg* is given, a second approximation is computed with *nseg* straight segments of nearly equal length. The lengths are computed based on the first approximation with *ndiv* segments.

actor (***kargs*)

Graphical representation

reverse ()

Return the reversed Nurbs curve.

The reversed curve is geometrically identical, but start and end point are interchanged and parameter values increase in the opposite direction.

class `plugins.nurbs.NurbsSurface` (*control, degree=(None, None), wts=None, knots=(None, None), closed=(False, False), blended=(True, True)*)

A NURBS surface

The Nurbs surface is defined as a tensor product of NURBS curves in two parametrical directions *u* and *v*. The control points form a grid of (*nctrlu*,*nctrlv*) points. The other data are like those for a NURBS curve, but need to be specified as a tuple for the (*u,v*) directions.

The knot values are only defined upon a multiplicative constant, equal to the largest value. Sensible default values are constructed automatically by a call to the `genKnotVector()` function.

If no knots are given and no degree is specified, the degree is set to the number of control points - 1 if the curve is blended. If not blended, the degree is not set larger than 3.

Warning: This is a class under development!

urange ()

Return the *u*-parameter range on which the curve is defined.

Returns a (2,) float array with the minimum and maximum parameter value *u* for which the curve is defined.

vrangle ()

Return the *v*-parameter range on which the curve is defined.

Returns a (2,) float array with the minimum and maximum parameter value *v* for which the curve is defined.

bbox ()

Return the bounding box of the NURBS surface.

pointsAt (*u*)

Return the points on the Nurbs surface at given parametric values.

Parameters:

- *u*: (nu,2) shaped float array: *nu* parametric values (*u,v*) at which a point is to be placed.

Returns (nu,3) shaped Coords with *nu* points at the specified parametric values.

derivs (*u, m*)

Return points and derivatives at given parametric values.

Parameters:

- *u*: (nu,2) shaped float array: *nu* parametric values (*u,v*) at which the points and derivatives are evaluated.

- *m*: tuple of two int values (mu,mv). The points and derivatives up to order mu in u direction and mv in v direction are returned.

Returns:

(nu+1,nv+1,nu,3) shaped Coords with *nu* points at the specified parametric values. The slice (0,0,:,:) contains the points.

approx (*ndiv=None, **kargs*)

Return a Quad4 Mesh approximation of the Nurbs surface

Parameters:

- *ndiv*: number of divisions of the parametric space.

actor (***kargs*)

Graphical representation

Functions defined in module plugins.nurbs

`plugins.nurbs.genKnotVector` (*nctrl, degree, blended=True, closed=False*)

Compute sensible knot vector for a Nurbs curve.

A knot vector is a sequence of non-decreasing parametric values. These values define the *knots*, i.e. the points where the analytical expression of the Nurbs curve may change. The knot values are only meaningful upon a multiplicative constant, and they are usually normalized to the range [0.0..1.0].

A Nurbs curve with *nctrl* points and of given *degree* needs a knot vector with *nknots* = *nctrl*+*degree*+1 values. A *degree* curve needs at least *nctrl* = *degree*+1 control points, and thus at least *nknots* = 2*(*degree*+1) knot values.

To make an open curve start and end in its end points, it needs knots with multiplicity *degree*+1 at its ends. Thus, for an open blended curve, the default policy is to set the knot values at the ends to 0.0, resp. 1.0, both with multiplicity *degree*+1, and to spread the remaining *nctrl* - *degree* - 1 values equally over the interval.

For a closed (blended) curve, the knots are equally spread over the interval, all having a multiplicity 1 for maximum continuity of the curve.

For an open unblended curve, all internal knots get multiplicity *degree*. This results in a curve that is only one time continuously derivable at the knots, thus the curve is smooth, but the curvature may be discontinuous. There is an extra requirement in this case: *nctrl* should be a multiple of *degree* plus 1.

Returns a KnotVector instance.

Example:

```
>>> print(genKnotVector(7,3))
KnotVector: 0.0(4), 0.25(1), 0.5(1), 0.75(1), 1.0(4)
>>> print(genKnotVector(7,3,blended=False))
KnotVector: 0.0(4), 1.0(3), 2.0(4)
>>> print(genKnotVector(3,2,closed=True))
KnotVector: 0.0(1), 0.2(1), 0.4(1), 0.6(1), 0.8(1), 1.0(1)
```

`plugins.nurbs.globalInterpolationCurve` (*Q, degree=3, strategy=0.5*)

Create a global interpolation NurbsCurve.

Given an ordered set of points *Q*, the `globalInterpolationCurve` is a NURBS curve of the given *degree*, passing through all the points.

Returns:

A NurbsCurve through the given point set. The number of control points is the same as the number of input points.

Warning: Currently there is the limitation that two consecutive points should not coincide. If they do, a warning is shown and the double points will be removed.

The procedure works by computing the control points that will produce a NurbsCurve with the given points occurring at predefined parameter values. The strategy to set this values uses a parameter as exponent. Different values produce (slightly) different curves. Typical values are:

0.0: equally spaced (not recommended) 0.5: centripetal (default, recommended) 1.0: chord length (often used)

`plugins.nurbs.NurbsCircle` ($O=[0.0, 0.0, 0.0]$, $r=1.0$, $X=[1.0, 0.0, 0.0]$, $Y=[0.0, 1.0, 0.0]$, $ths=0.0$, $the=360.0$)

Create a NurbsCurve representing a perfect circle or arc.

Parameters:

- O : float (3,): center of the circle
- r : float: radius
- X : unit vector in the plane of the circle
- Y : unit vector in the plane of the circle and perpendicular to X
- ths : start angle, measured from the X axis, counterclockwise in X - Y plane
- the : end angle, measured from the X axis

Returns a NurbsCurve that is a perfect circle or arc.

`plugins.nurbs.toCoords4` (x)

Convert cartesian coordinates to homogeneous

x: Coords Array with cartesian coordinates.

Returns a Coords4 object corresponding to the input cartesian coordinates.

`plugins.nurbs.pointsOnBezierCurve` (P, u)

Compute points on a Bezier curve

Parameters:

P is an array with $n+1$ points defining a Bezier curve of degree n . u is a vector with nu parameter values between 0 and 1.

Returns:

An array with the nu points of the Bezier curve corresponding with the specified parametric values. **ERROR:** currently u is a single paramtric value!

See also: examples BezierCurve, Casteljau

`plugins.nurbs.deCasteljau` (P, u)

Compute points on a Bezier curve using deCasteljau algorithm

Parameters:

P is an array with $n+1$ points defining a Bezier curve of degree n . u is a single parameter value between 0 and 1.

Returns:

A list with point sets obtained in the subsequent deCasteljau approximations. The first one is the set of control points, the last one is the point on the Bezier curve.

This function works with Coords as well as Coords4 points.

`plugins.nurbs.splitBezierCurve (P, u)`

Split a Bezier curve at parametric values

Parameters:

P is an array with n+1 points defining a Bezier curve of degree n. u is a single parameter value between 0 and 1.

Returns two arrays of n+1 points, defining the Bezier curves of degree n obtained by splitting the input curve at parametric value u. These results can be used with the control argument of BezierSpline to create the corresponding curve.

`plugins.nurbs.frenet (d1, d2, d3=None)`

Compute Frenet vectors, curvature and torsion.

Parameters:

- *d1*: first derivative at *npts* points of a nurbs curve
- *d2*: second derivative at *npts* points of a nurbs curve
- *d3*: (optional) third derivative at *npts* points of a nurbs curve

The derivatives of the nurbs curve are normally obtained from `NurbsCurve.deriv()`.

Returns:

- *T*: normalized tangent vector to the curve at *npts* points
- *N*: normalized normal vector to the curve at *npts* points
- *B*: normalized binormal vector to the curve at *npts* points
- *k*: curvature of the curve at *npts* points
- *t*: (only if *d3* was specified) torsion of the curve at *npts* points

Curvature is found from $|d1 \times d2| / |d1|^{**3}$

6.4.19 `plugins.objects` — Selection of objects from the global dictionary.

This is a support module for other pyFormex plugins.

Classes defined in module `plugins.objects`

class `plugins.objects.Objects (clas=None, like=None, filter=None, namelist=[])`

A selection of objects from the pyFormex Globals().

The class provides facilities to filter the global objects by their type and select one or more objects by their name(s). The values of these objects can be changed and the changes can be undone.

object_type ()

Return the type of objects in this selection.

set (*names*)

Set the selection to a list of names.

namelist can be a single object name or a list of names. This will also store the current values of the variables.

append (*name, value=None*)

Add a name,value to a selection.

If no value is given, its current value is used. If a value is given, it is exported.

clear ()

Clear the selection.

listAll ()

Return a list with all selectable objects.

This lists all the global names in pyformex.PF that match the class and/or filter (if specified).

remember (*copy=False*)

Remember the current values of the variables in selection.

If *copy==True*, the values are copied, so that the variables' current values can be changed inplace without affecting the remembered values.

changeValues (*newvalues*)

Replace the current values of selection by new ones.

The old values are stored locally, to enable undo operations.

This is only needed to change the values of objects that can not be changed inplace!

undoChanges ()

Undo the last changes of the values.

check (*single=False, warn=True*)

Check that we have a current selection.

Returns the list of Objects corresponding to the current selection. If *single==True*, the selection should hold exactly one Object name and a single Object instance is returned. If there is no selection, or more than one in case of *single==True*, an error message is displayed and None is returned

odict ()

Return the currently selected items as a dictionary.

Returns an OrderedDict with the currently selected objects in the order of the selection.names.

ask (*mode='multi'*)

Show the names of known objects and let the user select one or more.

mode can be set to 'single' to select a single item. Return a list with the selected names, possibly empty (if nothing was selected by the user), or None if there is nothing to choose from. This also sets the current selection to the selected names, unless the return value is None, in which case the selection remains unchanged.

ask1 ()

Select a single object from the list.

Returns the object, not its name!

forget ()

Remove the selection from the globals.

keep ()

Remove everything except the selection from the globals.

printval ()

Print the selection.

printbbox ()

Print the bbox of the current selection.

writeToFile (*filename*)
Write objects to a geometry file.

readFromFile (*filename*)
Read objects from a geometry file.

class `plugins.objects.DrawableObjects` (***kargs*)
A selection of drawable objects from the `globals()`.

This is a subclass of `Objects`. The constructor has the same arguments as the `Objects` class, plus the following:

- *annotations*: a set of functions that draw annotations of the objects. Each function should take an object name as argument, and draw the requested annotation for the named object. If the object does not have the annotation, it should be silently ignored. Default annotation functions available are:
 - `draw_object_name`
 - `draw_elem_numbers`
 - `draw_nodes`
 - `draw_node_numbers`
 - `draw_bbox`

No annotation functions are activated by default.

ask (*mode='multi'*)
Interactively sets the current selection.

drawChanges ()
Draws old and new version of a Formex with different colors.

old and new can be a either Formex instances or names or lists thereof. old are drawn in yellow, new in the current color.

undoChanges ()
Undo the last changes of the values.

toggleAnnotation (*f*, *onoff=None*)
Toggle the display of an annotation On or Off.

If given, *onoff* is True or False. If no *onoff* is given, this works as a toggle.

drawAnnotation (*f*)
Draw some annotation for the current selection.

removeAnnotation (*f*)
Remove the annotation *f*.

editAnnotations (*ontop=None*)
Edit the annotation properties

Currently only changes the *ontop* attribute for all drawn annotations. Values: True, False or '' (toggle). Other values have no effect.

hasAnnotation (*f*)
Return the status of annotation *f*

setProp (*prop=None*)
Set the property of the current selection.

prop should be a single integer value or None. If None is given, a value will be asked from the user. If a negative value is given, the property is removed. If a selected object does not have a `setProp` method, it is ignored.

delProp ()

Delete the property of the current selection.

This will reset the *prop* attribute of all selected objects to None.

Functions defined in module `plugins.objects`

`plugins.objects.draw_object_name (n)`

Draw the name of an object at its center.

`plugins.objects.draw_elem_numbers (n)`

Draw the numbers of an object's elements.

`plugins.objects.draw_nodes (n)`

Draw the nodes of an object.

`plugins.objects.draw_node_numbers (n)`

Draw the numbers of an object's nodes.

`plugins.objects.draw_free_edges (n)`

Draw the feature edges of an object.

`plugins.objects.draw_bbox (n)`

Draw the bbox of an object.

6.4.20 `plugins.partition` — Partitioning tools

Functions defined in module `plugins.partition`

`plugins.partition.prepare (V)`

Prepare the surface for slicing operation.

`plugins.partition.colorCut (F, P, N, prop)`

Color a Formex in two by a plane (P,N)

`plugins.partition.splitProp (G, name)`

Partition a Formex according to its prop values and export the results.

If G has property numbers, the structure is split and according to the property values, and the (compact) parts are exported with names 'name-propnumber'.

`plugins.partition.partition (Fin, prop=0)`

Interactively partition a Formex.

By default, the parts will get properties 0,1,... If prop >= 0, the parts will get incremental props starting from prop.

Returns the cutplanes in an array with shape (ncuts,2,3), where (i,0,:) is a point in the plane i and (i,1,:) is the normal vector on the plane i .

As a side effect, the properties of the input Formex will be changed to flag the parts between successive cut planes by incrementing property values. If you wish to restore the original properties, you should copy them (or the input Formex) before calling this function.

6.4.21 `plugins.plot2d` — `plot2d.py`

Generic 2D plotting functions for pyFormex.

Functions defined in module `plugins.plot2d`

`plugins.plot2d.showStepPlot` (*x*, *y*, *xlabel=""*, *ylabel=""*, *label=""*, *title=None*)
 Show a step plot of *x,y* data.

`plugins.plot2d.showHistogram` (*x*, *y*, *cumulative=False*, *xlabel=""*, *ylabel=""*, *label=""*, *title=""*)
 Show a histogram of *x,y* data.

`plugins.plot2d.createHistogram` (*data*, *cumulative=False*, ***kargs*)
 Create a histogram from data

6.4.22 `plugins.polygon` — Polygonal facets.

Classes defined in module `plugins.polygon`

class `plugins.polygon.Polygon` (*border*, *normal=2*, *holes=[]*)

A Polygon is a flat surface bounded by a closed PolyLine.

The border is specified as a Coords object with shape (*nvertex,3*) specifying the vertex coordinates in order. While the Coords are 3d, only the first 2 components are used.

nelems ()

Return the number of elements in the Geometry.

Returns *int* – The number of elements in the Geometry. This is an abstract method that should be reimplemented by the derived class.

npoints ()

Return the number of points and edges.

vectors ()

Return the vectors from each point to the next one.

angles ()

Return the angles of the line segments with the x-axis.

externalAngles ()

Return the angles between subsequent line segments.

The returned angles are the change in direction between the segment ending at the vertex and the segment leaving. The angles are given in degrees, in the range $]-180,180]$. The sum of the external angles is always (a multiple of) 360. A convex polygon has all angles of the same sign.

isConvex ()

Check if the polygon is convex and turning anticlockwise.

Returns:

- +1 if the Polygon is convex and turning anticlockwise,
- -1 if the Polygon is convex, but turning clockwise,
- 0 if the Polygon is not convex.

internalAngles ()

Return the internal angles.

The returned angles are those between the two line segments at each vertex. The angles are given in degrees, in the range $]-180,180]$. These angles are the complement of the

reverse ()

Return the Polygon with reversed order of vertices.

fill()

Fill the surface inside the polygon with triangles.

Returns a TriSurface filling the surface inside the polygon.

area()

Compute area inside a polygon.

Functions defined in module `plugins.polygon`

`plugins.polygon.projected(X, N)`

Returns 2-D coordinates of a set of 3D coordinates.

The returned 2D coordinates are still stored in a 3D Coords object. The last coordinate will however (approximately) be zero.

6.4.23 `plugins.polynomial` — Polynomials

This module defines the class `Polynomial`, representing a polynomial in n variables.

Classes defined in module `plugins.polynomial`

class `plugins.polynomial.Polynomial` (*exp, coeff=None*)

A polynomial in $ndim$ dimensions.

Parameters:

- *exp*: ($nterms, ndim$) int array with the exponents of each of the $ndim$ variables in the $nterms$ terms of the polynomial.
- *coeff*: ($nterms$,) float array with the coefficients of the terms. If not specified, all coefficients are set to 1.

Example:

```
>>> p = Polynomial([(0,0), (1,0), (1,1), (0,2)], (2,3,-1,-1))
>>> print(p.atoms())
['1', 'x', 'x*y', 'y**2']
>>> print(p.human())
2.0 + 3.0*x -1.0*x*y -1.0*y**2
>>> print(p.evalAtoms([[1,2], [3,0], [2,1]]))
[[ 1.  1.  2.  4.]
 [ 1.  3.  0.  0.]
 [ 1.  2.  2.  1.]]
>>> print(p.eval([[1,2], [3,0], [2,1]]))
[-1.  11.  5.]
```

degrees()

Return the degree of the polynomial in each of the dimensions.

The degree is the maximal exponent for each of the dimensions.

degree()

Return the total degree of the polynomial.

The degree is the sum of the degrees for all dimensions.

evalAtoms1 (*x*)

Evaluate the monomials at the given points

x is an (npoints,ndim) array of points where the polynomial is to be evaluated. The result is an (npoints,nterms) array of values.

evalAtoms (*x*)

Evaluate the monomials at the given points

x is an (npoints,ndim) array of points where the polynomial is to be evaluated. The result is an (npoints,nterms) array of values.

eval (*x*)

Evaluate the polynomial at the given points

x is an (npoints,ndim) array of points where the polynomial is to be evaluated. The result is an (npoints,) array of values.

atoms (*symbol='xyz'*)

Return a human representation of the monomials

human (*symbol='xyz'*)

Return a human representation

Functions defined in module `plugins.polynomial`

`plugins.polynomial.polynomial` (*atoms, x, y=0, z=0*)

Build a matrix of functions of coords.

- *atoms*: a list of text strings representing a mathematical function of *x*, and possibly of *y* and *z*.
- *x, y, z*: a list of *x*- (and optionally *y*-, *z*-) values at which the *atoms* will be evaluated. The lists should have the same length.

Returns a matrix with *nvalues* rows and *natoms* columns.

`plugins.polynomial.monomial` (*exp, symbol='xyz'*)

Compute the monomials for the given exponents

- *exp*: a tuple of integer exponents
- *symbol*: a string of at least the same length as *exp*

Returns a string representation of a monomial created by raising the symbols to the corresponding exponent.

Example:

```
>>> monomial((2,1))
'x**2*y'
```

6.4.24 `plugins.postproc` — Postprocessing functions

Postprocessing means collecting a geometrical model and computed values from a numerical simulation, and render the values on the domain.

Functions defined in module `plugins.postproc`

`plugins.postproc.frameScale` (*nframes=10, cycle='up', shape='linear'*)

Return a sequence of scale values between -1 and +1.

`nframes` : the number of steps between 0 and -1/+1 values.

`cycle`: determines how subsequent cycles occur:

'up': ramping up

'updown': ramping up and down

'revert': ramping up and down then reverse up and down

`shape`: determines the shape of the amplitude curve:

'linear': linear scaling

'sine': sinusoidal scaling

6.4.25 `plugins.properties` — General framework for attributing properties to geometrical elements.

Properties can really be just about any Python object. Properties can be attributed to a set of geometrical elements.

Classes defined in module `plugins.properties`

class `plugins.properties.Database` (*data*={})

A class for storing properties in a database.

readDatabase (*filename*, **args*, ***kargs*)

Import all records from a database file.

For now, it can only read databases using flatkeydb. *args* and *kargs* can be used to specify arguments for the FlatDB constructor.

class `plugins.properties.MaterialDB` (*data*={})

A class for storing material properties.

class `plugins.properties.SectionDB` (*data*={})

A class for storing section properties.

class `plugins.properties.ElemSection` (*section*=None, *material*=None, *orientation*=None, ***kargs*)

Properties related to the section of an element.

An element section property can hold the following sub-properties:

section the geometric properties of the section. This can be a dict or a string. If it is a string, its value is looked up in the global section database. The section dict should at least have a key 'sectiontype', defining the type of section.

Currently the following sectiontype values are known by module `fe_abq` for export to Abaqus/Calculix:

- 'solid' : a solid 2D or 3D section,
- 'circ' : a plain circular section,
- 'rect' : a plain rectangular section,
- 'pipe' : a hollow circular section,
- 'box' : a hollow rectangular section,
- 'I' : an I-beam,
- 'general' : anything else (automatically set if not specified).

- 'rigid' : a rigid body

The other possible (useful) keys in the section dict depend on the sectiontype. Again for `fe_abq`:

- for sectiontype 'solid' : thickness
- the sectiontype 'general': `cross_section`, `moment_inertia_11`, `moment_inertia_12`, `moment_inertia_22`, `torsional_constant`
- for sectiontype 'circ': radius
- for sectiontype 'rigid': `refnode`, `density`, `thickness`

material the element material. This can be a dict or a string. Currently known keys to `fe_abq.py` are: `young_modulus`, `shear_modulus`, `density`, `poisson_ratio` . (see `fmtMaterial` in `fe_abq`) It should not be specified for rigid sections.

orientation

- a Dict, or
- a list of 3 direction cosines of the first beam section axis.

addSection (*section*)

Create or replace the section properties of the element.

If 'section' is a dict, it will be added to the global SectionDB. If 'section' is a string, this string will be used as a key to search in the global SectionDB.

computeSection (*section*)

Compute the section characteristics of specific sections.

addMaterial (*material*)

Create or replace the material properties of the element.

If the argument is a dict, it will be added to the global MaterialDB. If the argument is a string, this string will be used as a key to search in the global MaterialDB.

class `plugins.properties.ElemLoad` (*label=None, value=None, dir=None*)

Distributed loading on an element.

class `plugins.properties.EdgeLoad` (*edge=-1, label=None, value=None*)

Distributed loading on an element edge.

class `plugins.properties.CoordSystem` (*csys, cdata*)

A class for storing coordinate systems.

class `plugins.properties.Amplitude` (*data, definition='TABULAR', atime='STEP TIME', smoothing=None*)

A class for storing an amplitude.

The amplitude is a list of tuples (time,value).

atime (amplitude time) can be either STEP TIME (default in Abaqus) or TOTAL TIME

smoothing (optional) is a float (from 0. to 0.5, suggested value 0.05) representing the fraction of the time interval before and after each time point during which the piecewise linear time variation will be replaced by a smooth quadratic time variation (avoiding infinite accelerations). Smoothing should be used in combination with TABULAR (set 0.05 as default value?)

class `plugins.properties.PropertyDB` (*mat="", sec=""*)

A database class for all properties.

This class collects all properties that can be set on a geometrical model.

This should allow for storing:

- materials
- sections
- any properties
- node properties
- elem properties
- model properties (current unused: use unnamed properties)

Materials and sections use their own database for storing. They can be specified on creating the property database. If not specified, default ones are created from the files distributed with pyFormex.

setMaterialDB (*aDict*)

Set the materials database to an external source

setSectionDB (*aDict*)

Set the sections database to an external source

print ()

Print the property database

Prop (*kind=""*, *tag=None*, *set=None*, *name=None*, ***kargs*)

Create a new property, empty by default.

A property can hold almost anything, just like any Dict type. It has however four predefined keys that should not be used for anything else than explained hereafter:

- **nr**: a unique id, that never should be set/changed by the user.
- **tag**: an identification tag used to group properties
- **name**: the name to be used for this set. Default is to use an automatically generated name.
- **set**: identifies the geometrical elements for which the defined properties will hold. This can be either:
 - a single number,
 - a list of numbers,
 - the name of an already defined set,
 - a list of such names.

Besides these, any other fields may be defined and will be added without checking.

getProp (*kind=""*, *rec=None*, *tag=None*, *attr=[]*, *noattr=[]*, *delete=False*)

Return all properties of type kind matching tag and having attr.

kind is either 'e', 'n', 'e' or 'm' If rec is given, it is a list of record numbers or a single number. If a tag or a list of tags is given, only the properties having a matching tag attribute are returned.

attr and noattr are lists of attributes. Only the properties having all the attributes in attr and none of the properties in noattr are returned. Attributes whose value is None are treated as non-existing.

If delete==True, the returned properties are removed from the database.

delProp (*kind=""*, *rec=None*, *tag=None*, *attr=[]*)

Delete properties.

This is equivalent to getProp() but the returned properties are removed from the database.

nodeProp (*prop=None*, *set=None*, *name=None*, *tag=None*, *load=None*, *bound=None*, *displ=None*, *veloc=None*, *accel=None*, *csys=None*, *ampl=None*, ***kargs*)

Create a new node property, empty by default.

A node property can contain any combination of the following fields:

- tag: an identification tag used to group properties (this is e.g. used to flag Step, increment, load case, ...)
- set: a single number or a list of numbers identifying the node(s) for which this property will be set, or a set name. If None, the property will hold for all nodes.
- cload: a concentrated load: a list of 6 float values [FX,FY,FZ,MX,MY,MZ] or a list of (dofid,value) tuples.
- displ,veloc,accel: prescribed displacement, velocity or acceleration: a list of 6 float values [UX,UY,UZ,RX,RY,RZ] or a list of tuples (dofid,value)
- bound: a boundary condition: a string, a list of 6 codes (0/1), or a list of tuples (dofid, value)
- csys: a CoordSystem
- ampl: the name of an Amplitude

elemProp (*prop=None, grp=None, set=None, name=None, tag=None, section=None, eltype=None, dload=None, eload=None, ampl=None, **kargs*)

Create a new element property, empty by default.

An elem property can contain any combination of the following fields:

- tag: an identification tag used to group properties (this is e.g. used to flag Step, increment, load case, ...)
- set: a single number or a list of numbers identifying the element(s) for which this property will be set, or a set name. If None, the property will hold for all elements.
- grp: an elements group number (default None). If specified, the element numbers given in set are local to the specified group. If not, elements are global and should match the global numbering according to the order in which element groups will be specified in the Model.
- eltype: the element type (currently in Abaqus terms).
- section: an ElemSection specifying the element section properties.
- dload: an ElemLoad specifying a distributed load on the element.
- ampl: the name of an Amplitude

Functions defined in module `plugins.properties`

`plugins.properties.setMaterialDB` (*mat*)

Set the global materials database.

If *mat* is a MaterialDB, it will be used as the global MaterialDB. Else, a new global MaterialDB will be created, initialized from the argument *mat*.

`plugins.properties.setSectionDB` (*sec*)

Set the global sections database.

If *sec* is a SectionDB, it will be used as the global SectionDB. Else, a new global SectionDB will be created, initialized from the argument *sec*.

`plugins.properties.checkIdValue` (*values*)

Check that a variable is a list of (id,value) tuples

id should be convertible to an int, *value* to a float. If ok, return the values as a list of (int,float) tuples.

`plugins.properties.checkArrayOrIdValue` (*values*)

Check that a variable is a list of values or (id,value) tuples

This convenience function checks that the argument is either:

- a list of 6 float values (or convertible to it), or
- a list of (id,value) tuples where id is convertible to an int, value to a float.

If ok, return the values as a list of (int,float) tuples.

Examples

```
>>> checkArrayOrIdValue([0,3,4,0,0,0])
[(1, 3.0), (2, 4.0)]
>>> checkArrayOrIdValue([(1,3.0), (2,4.0)])
[(1, 3.0), (2, 4.0)]
>>> checkArrayOrIdValue([(0,1.0), (2,4.0), (3,5.0), (4,4), (5,3.0), (1,4.0)])
[(0, 1.0), (2, 4.0), (3, 5.0), (4, 4.0), (5, 3.0), (1, 4.0)]
```

`plugins.properties.checkArrayOrIdValueOrEmpty` (*values*)

Check that a variable is a list of values or (id,value) tuples or empty.

This convenience function checks that the argument is either:

- a list of 6 float values (or convertible to it), or
- a list of (id,value) tuples where id is convertible to an int, value to a float.
- something representing an empty list

If ok, return the values as a list of (int,float) tuples.

Examples

```
>>> checkArrayOrIdValueOrEmpty([0,3,4,0,0,0])
[(1, 3.0), (2, 4.0)]
>>> checkArrayOrIdValueOrEmpty([(1,3.0), (2,4.0)])
[(1, 3.0), (2, 4.0)]
>>> checkArrayOrIdValueOrEmpty([])
[]
```

`plugins.properties.checkString` (*a, valid*)

Check that a string *a* has one of the valid values.

This is case insensitive, and returns the upper case string if valid. Else, an error is raised.

`plugins.properties.FindListItem` (*l, p*)

Find the item *p* in the list *l*.

If *p* is an item in the list (not a copy of it!), this returns its position. Else, -1 is returned.

Matches are found with a 'is' function, not an '=='. Only the first match will be reported.

`plugins.properties.RemoveListItem` (*l, p*)

Remove the item *p* from the list *l*.

If *p* is an item in the list (not a copy of it!), it is removed from the list. Matches are found with a 'is' comparison. This is different from the normal Python list.remove() method, which uses '=='. As a result, we can find complex objects which do not allow '==', such as ndarrays.

6.4.26 `plugins.pyformex_gts` — Operations on triangulated surfaces using GTS functions.

This module provides access to GTS from inside pyFormex.

Functions defined in module `plugins.pyformex_gts`

`plugins.pyformex_gts.boolean` (*self, surf, op, check=False, verbose=False*)

Perform a boolean operation with another surface.

Boolean operations between surfaces are a basic operation in free surface modeling. Both surfaces should be closed orientable non-intersecting manifolds. Use the `check()` method to find out.

The boolean operations are set operations on the enclosed volumes: `union('+')`, `difference('-')` or `intersection('*')`.

Parameters

- **surf** (*TriSurface*) – Another *TriSurface* that is a closed manifold surface.
- **op** ('+', '-', '*') – The boolean operation to perform: `union('+')`, `difference('-')` or `intersection('*')`.
- **check** (*bool*) – If True, a check is done that the surfaces are not self-intersecting; if one of them is, the set of self-intersecting faces is written (as a *GtsSurface*) on standard output
- **verbose** (*bool*) – If True, print statistics about the surface.

Returns *TriSurface* – A closed manifold *TriSurface* that is the volume union, difference or intersection of self with surf.

Note: This method uses the external command 'gtsset' and will not run if it is not installed (available from `pyformex/extras`).

`plugins.pyformex_gts.intersection` (*self, surf, check=False, verbose=False*)

Return the intersection curve of two surfaces.

Boolean operations between surfaces are a basic operation in free surface modeling. Both surfaces should be closed orientable non-intersecting manifolds. Use the `check()` method to find out.

Parameters:

- *surf*: a closed manifold surface
- *check*: boolean: check that the surfaces are not self-intersecting; if one of them is, the set of self-intersecting faces is written (as a *GtsSurface*) on standard output
- *verbose*: boolean: print statistics about the surface

Returns: a list of intersection curves.

`plugins.pyformex_gts.inside` (*self, pts, atol='auto', multi=True*)

Test which of the points *pts* are inside the surface.

Parameters:

- *pts*: a (usually 1-plex) Formex or a data structure that can be used to initialize a Formex.

Returns an integer array with the indices of the points that are inside the surface. The indices refer to the onedimensional list of points as obtained from `pts.points()`.

6.4.27 `plugins.section2d` — Some functions operating on 2D structures.

This is a plugin for pyFormex. (C) 2002 Benedict Verhegghe

See the Section2D example for an example of its use.

Classes defined in module `plugins.section2d`

class `plugins.section2d.PlaneSection` (*F*)

A class describing a general 2D section.

The 2D section is the area inside a closed curve in the (x,y) plane. The curve is described by a finite number of points and by straight segments connecting them.

Functions defined in module `plugins.section2d`

`plugins.section2d.sectionChar` (*F*)

Compute characteristics of plane sections.

The plane sections are described by their circumference, consisting of a sequence of straight segments. The segment end point data are gathered in a plex-2 Formex. The segments should form a closed curve. The z-value of the coordinates does not have to be specified, and will be ignored if it is. The resulting path through the points should rotate positively around the z axis to yield a positive surface.

The return value is a dict with the following characteristics:

- *L* : circumference,
- *A* : enclosed surface,
- *Sx* : first area moment around global x-axis
- *Sy* : first area moment around global y-axis
- *Ixx* : second area moment around global x-axis
- *Iyy* : second area moment around global y-axis
- *Ixy* : product moment of area around global x,y-axes

`plugins.section2d.extendedSectionChar` (*S*)

Computes extended section characteristics for the given section.

S is a dict with section basic section characteristics as returned by `sectionChar()`. This function computes and returns a dict with the following:

- *xG*, *yG* : coordinates of the center of gravity G of the plane section
- *IGxx*, *IGyy*, *IGxy* : second area moments and product around axes through G and parallel with the global x,y-axes
- *alpha* : angle(in radians) between the global x,y axes and the principal axes (X,Y) of the section (X and Y always pass through G)
- *IXX*, *IYY* : principal second area moments around X,Y respectively. (The second area product is always zero.)

`plugins.section2d.princTensor2D` (*Ixx*, *Iyy*, *Ixy*)

Compute the principal values and directions of a 2D tensor.

Returns a tuple with three values:

- *alpha* : angle (in radians) from x-axis to principal X-axis
- *IXX,IYY* : principal values of the tensor

6.4.28 `plugins.sectionize` — `sectionize.py`

Create, measure and approximate cross section of a Formex.

Functions defined in module `plugins.sectionize`

`plugins.sectionize.connectPoints` (*F*, *close=False*)

Return a Formex with straight segments connecting subsequent points.

F can be a Formex or data that can be turned into a Formex (e.g. an (n,3) array of points). The result is a plex-2 Formex connecting the subsequent points of *F* or the first point of subsequent elements in case the plexitude of *F* > 1. If *close=True*, the last point is connected back to the first to create a closed polyline.

`plugins.sectionize.centerline` (*F*, *dir*, *nx=2*, *mode=2*, *th=0.2*)

Compute the centerline in the direction *dir*.

`plugins.sectionize.createSegments` (*F*, *ns=None*, *th=None*)

Create segments along 0 axis for sectionizing the Formex *F*.

`plugins.sectionize.sectionize` (*F*, *segments*, *th=0.1*, *visual=True*)

Sectionize a Formex in planes perpendicular to the segments.

F is any Formex. *segments* is a plex-2 Formex.

Planes are chosen in each center of a segment, perpendicular to that segment. Then parts of the Formex *F* are selected in the neighbourhood of each plane. Each part is then approximated by a circle in that plane.

th is the relative thickness of the selected part of the Formex. If *th* = 0.5, that part will be delimited by two planes in the endpoints of and perpendicular to the segments.

`plugins.sectionize.drawCircles` (*sections*, *ctr*, *diam*)

Draw circles as approximation of Formices.

6.4.29 `plugins.tetgen` — Interface with `tetgen`

A collection of functions to read/write tetgen files and to run the tetgen program

tetgen is a quality tetrahedral mesh generator and a 3D Delaunay triangulator. See <http://tetgen.org>

Functions defined in module `plugins.tetgen`

`plugins.tetgen.readNodeFile` (*fn*)

Read a tetgen .node file.

Returns a tuple as described in `readNodesBlock()`.

`plugins.tetgen.readEleFile` (*fn*)

Read a tetgen .ele file.

Returns a tuple as described in `readElemsBlock`.

`plugins.tetgen.readFaceFile` (*fn*)

Read a tetgen .face file.

Returns a tuple as described in `readFacesBlock`.

`plugins.tetgen.readSmeshFile` (*fn*)

Read a tetgen .smesh file.

Returns an array of triangle elements.

`plugins.tetgen.readPolyFile` (*fn*)

Read a tetgen .poly file.

Returns an array of triangle elements.

`plugins.tetgen.readSurface` (*fn*)

Read a tetgen surface from a .node/.face file pair.

The given filename is either the .node or .face file. Returns a tuple of (nodes,elems).

`plugins.tetgen.skipComments` (*fil*)

Skip comments and blank lines on a tetgen file.

Reads from a file until the first non-comment and non-empty line. Then returns the non-empty, non-comment line, stripped from possible trailing comments. Returns None if end of file is reached.

`plugins.tetgen.stripLine` (*line*)

Strip blanks, newline and comments from a line of text.

`plugins.tetgen.getInts` (*line, nint*)

Read a number of ints from a line, adding zero for omitted values.

line is a string with blanks separated integer values. Returns a list of *nint* integers. The trailing ones are set to zero if the strings contains less values.

`plugins.tetgen.addElem` (*elems, nrs, e, n, nplex*)

Add an element to a collection.

`plugins.tetgen.readNodesBlock` (*fil, npts, ndim, nattr, nbmark*)

Read a tetgen nodes block.

Returns a tuple with:

- *coords*: Coords array with nodal coordinates
- *nrs*: node numbers
- *attr*: node attributes
- *bmrk*: node boundary marker

The last two may be None.

`plugins.tetgen.readElemsBlock` (*fil, nelems, nplex, nattr*)

Read a tetgen elems block.

Returns a tuple with:

- *elems*: Connectivity of type 'tet4' or 'tet10'
- *nrs*: the element numbers
- *attr*: the element attributes

The last can be None.

`plugins.tetgen.readFacesBlock` (*fil, nelems, nbmark*)

Read a tetgen faces block.

Returns a a tuple with:

- `elems`: Connectivity of type 'tri3'
- `nrs`: face numbers
- `bmrk`: face boundary marker

The last can be None.

`plugins.tetgen.readSmeshFacetsBlock` (*fil, nfacets, nbmark*)

Read a tetgen .smesh facets bock.

Returns a tuple of dictionaries with plexitudes as keys:

- `elems`: for each plexitude a Connectivity array
- `nrs`: for each plexitude a list of element numbers in corresponding elems

`plugins.tetgen.readNeigh` (*fn*)

Read a tetgen .neigh file.

Returns an array containing the tetrahedra neighbours:

`plugins.tetgen.writeNodes` (*fn, coords, offset=0*)

Write a tetgen .node file

`plugins.tetgen.writeSmesh` (*fn, facets, coords=None, holes=None, regions=None*)

Write a tetgen .smesh file.

Currently it only writes the facets of a triangular surface mesh. Coords should be written independently to a .node file.

`plugins.tetgen.writeTmesh` (*fn, elems, offset=0*)

Write a tetgen .ele file.

Writes elements of a tet4 mesh.

`plugins.tetgen.writeSurface` (*fn, coords, elems*)

Write a tetgen surface model to .node and .smesh files.

Parameters

- **`fn`** (*path_like*) – Filename of the files to which the model sohould be exported. The provided file name is either the .node or the .smesh filename, or else it is the basename where .node and .smesh extensions will be appended.
- **`coords`** (*Coords*) – The vertices in the surface model.
- **`elems`** (*array*) – The element definitions in function of the vertex numbers.

`plugins.tetgen.writeTetMesh` (*fn, coords, elems*)

Write a tetgen tetrahedral mesh model to .node and .ele files.

The provided file name is either the .node or the .smesh filename, or else it is the basename where .node and .ele extensions will be appended.

`plugins.tetgen.runTetgen` (*fn, options=""*)

Run tetgen mesher on the specified file.

The input file is a closed triangulated surface. tetgen will generate a volume tetraeder mesh inside the surface, and create a new approximation of the surface as a by-product.

`plugins.tetgen.readTetgen` (*fn*)

Read a tetgen model.

This reads a file created by the ‘tetgen’ tetrahedral mesher and returns corresponding pyFormex objects.

Parameters *fn* (*path_like*) – Path to a tetgen file. This can be one of .node, .ele, .face, .smesh or .poly files.

Returns *dict* – If the suffix of *fn* is one of .node, .ele or .face, the dict contains one item with key ‘tetgen.SUFFIX’. For suffix .node, the value is a Coords read from the .node file, for suffix .ele or .face, the value is a Mesh with elems read from the .ele or .face, and coords read from the corresponding .node. If the suffix is .smesh or .poly, the dict contains a number of Meshes, with keys ‘Mesh-NPLEX’ where NPLEX is the plexitude of the corresponding Mesh.

`plugins.tetgen.tetgenConvexHull` (*pts*)

Tetralize the convex hull of some points.

Parameters *pts* (*Coords*) – A collection of points to find the convex hull of.

Returns

- **convexhull** (*TriSurface*) – The smallest *TriSurface* enclosing all the points.
- **tetmesh** (*Mesh*) – A tetraeder *Mesh* filling the convex hull.
- *If all points are on the same plane there is no convex hull.*

`plugins.tetgen.checkSelfIntersectionsWithTetgen` (*self*, *verbose=False*)

check self intersections using tetgen

Returns pairs of intersecting triangles

`plugins.tetgen.tetMesh` (*surfacefile*, *quality=2.0*, *volume=None*, *outputdir=None*)

Create a tetrahedral mesh inside a surface

This uses the external program ‘tetgen’ to create a tetrahedral mesh inside a closed manifold surface.

Parameters

- **surfacefile** (*path_like*) – Path to a file holding a surface model. The file can be either a .off or .stl.
- **quality** (*float*) – The quality of the output tetrahedral mesh. The value is a constraint on the circumradius-to-shortest-edge ratio. The default (2.0) already provides a high quality mesh. Providing a larger value will reduce quality but increase speed. With *quality=None*, no quality constraint will be imposed.
- **volume** (*float*, *optional*) – If provided, applies a maximum tetrahedron volume constraint.
- **outputdir** (*path_like*) – Path to an existing directory where the results will be placed. The default is to use the directory where the input file resides.

Returns *Mesh* – A tetrahedral *Mesh* (eltype=‘tet4’) filling the input surface, provided the tetgen program finished successfully.

6.4.30 `plugins.tools` — `tools.py`

Graphic Tools for pyFormex.

Functions defined in module `plugins.tools`

`plugins.tools.getObjectItems` (*obj, items, mode*)
Get the specified items from object.

`plugins.tools.getCollection` (*K*)
Returns a collection.

`plugins.tools.growCollection` (*K, **kargs*)
Grow the collection with *n* frontal rings.

K should be a collection of elements. This should work on any objects that have a `growSelection` method.

`plugins.tools.partitionCollection` (*K*)
Partition the collection according to node adjacency.

The actor numbers will be connected to a collection of property numbers, e.g. 0 [1 [4,12] 2 [6,20]], where 0 is the actor number, 1 and 2 are the property numbers and 4, 12, 6 and 20 are the element numbers.

`plugins.tools.getPartition` (*K, prop*)
Remove all partitions with property not in *prop*.

`plugins.tools.exportObjects` (*obj, name, single=False*)
Export a list of objects under the given name.

If *obj* is a list, and *single=True*, each element of the list is exported as a single item. The items will be given the names *name-0*, *name-1*, etc. Else, the *obj* is exported as is under the name.

`plugins.tools.actorDialog` (*actorids*)
Create an actor dialog for the specified actors (by index)

6.4.31 `plugins.turtle` — Turtle graphics for pyFormex

This module was mainly aimed at the drawing of Lindenmayer products (see `plugins.lima` and the Lima example).

The idea is that a turtle can be moved in 2D from one position to another, thereby creating a line between start and endpoint or not.

The current state of the turtle is defined by

- `pos`: the position as a 2D coordinate pair (x,y),
- `angle`: the moving direction as an angle (in degrees) with the x-axis,
- `step`: the speed, as a discrete step size.

The start conditions are: `pos=(0,0)`, `step=1.`, `angle=0.`

The followin example turtle script creates a unit square:

```
fd();ro(90);fd();ro(90);fd();ro(90);fd()
```

Functions defined in module `plugins.turtle`

`plugins.turtle.sind` (*arg*)
Return the sine of an angle in degrees.

`plugins.turtle.cosd` (*arg*)
Return the cosine of an angle in degrees.

`plugins.turtle.reset()`

Reset the turtle graphics engine to start conditions.

This resets the turtle's state to the starting conditions `pos=(0,0)`, `step=1.`, `angle=0.`, removes everything from the state save stack and empties the resulting path.

`plugins.turtle.push()`

Save the current state of the turtle.

The turtle state includes its position, step and angle.

`plugins.turtle.pop()`

Restore the turtle state to the last saved state.

`plugins.turtle.fd(d=None, connect=True)`

Move forward over a step *d*, with or without drawing.

The direction is the current direction. If *d* is not given, the step size is the current step.

By default, the new position is connected to the previous with a straight line segment.

`plugins.turtle.mv(d=None)`

Move over step *d* without drawing.

`plugins.turtle.ro(a)`

Rotate over angle *a*. The new direction is incremented with *a*

`plugins.turtle.go(p)`

Go to position *p* (without drawing).

While the *mv* method performs a relative move, this is an absolute move. *p* is a tuple of (x,y) values.

`plugins.turtle.st(d)`

Set the step size.

`plugins.turtle.an(a)`

Set the angle

`plugins.turtle.play(scr, glob=None)`

Play all the commands in the script *scr*

The script is a string of turtle commands, where each command is ended with a semicolon (;).

If a dict *glob* is specified, it will be update with the turtle module's `globals()` after each turtle command.

6.4.32 `plugins.units` — A Python wrapper for unit conversion of physical quantities.

This module uses the standard UNIX program 'units' (available from <http://www.gnu.org/software/units/units.html>) to do the actual conversions. Obviously, it will only work on systems that have this program available.

If you really insist on running another OS lacking the units command, have a look at <http://home.tiscali.be/be052320/Unum.html> and make an implementation based on unum. If you GPL it and send it to me, I might include it in this distribution.

Classes defined in module `plugins.units`

`class plugins.units.UnitsSystem(system='international')`

A class for handling and converting units of physical quantities.

The units class provides two built-in consistent units systems: `International()` and `Engineering()`. `International()` returns the standard International Standard units. `Engineering()` returns a consistent engineering system, which is very practical for use in mechanical engineering. It uses 'mm' for length and 'MPa' for pressure and stress. To keep it consistent however, the density is rather unpractical: 't/mm³'. If you want to use t/m³, you can make a custom units system. Beware with non-consistent unit systems though! The better practice is to allow any unit to be specified at input (and eventually requested for output), and to convert everything internally to a consistent system. Apart from the units for usual physical quantities, Units stores two special purpose values in its units dictionary: 'model' : defines the length unit used in the geometrical model 'problem' : defines the unit system to be used in the problem. Defaults are: model='m', problem='international'.

Add (*un*)

Add the units from dictionary un to the units system

Predefined (*system*)

Returns the predefined units for the specified system

International ()

Returns the international units system.

Engineering ()

Returns a consistent engineering units system.

Read (*filename*)

Read units from file with specified name.

The units file is an ascii file where each line contains a couple of words separated by a colon and a blank. The first word is the type of quantity, the second is the unit to be used for this quantity. Lines starting with '#' are ignored. A 'problem: system' line sets all units to the corresponding value of the specified units system.

Get (*ent*)

Get units list for the specified entities.

If ent is a single entity, returns the corresponding unit if an entry ent exists in the current system or else returns ent unchanged. If ent is a list of entities, returns a list of corresponding units. Example: with the default units system:

```
Un = UnitsSystem()
Un.Get(['length', 'mass', 'float'])
```

returns: ['m', 'kg', 'float']

Functions defined in module plugins.units

`plugins.units.convertUnits` (*From, To*)

Converts between conformable units.

This function converts the units 'From' to units 'To'. The units should be conformable. The 'From' argument can (and usually does) include a value. The return value is a string with the converted value without units. Thus: `convertUnits('3.45 kg','g')` will return '3450'. This function is merely a wrapper around the GNU 'units' command, which should be installed for this function to work.

Examples

```
>>> convertUnits('25.4cm', 'in')
'10'
```

(continues on next page)

(continued from previous page)

```

>>> convertUnits('31e6mg', 'kg')
'31'
>>> convertUnits('1 lightyear', 'km')
'9.4607305e+12'
>>> convertUnits('21000 kN/cm**2', 'MPa')
'210000'

```

6.4.33 `plugins.web` — Tools to access files from the web

This module provides some convenience functions to access files from the web. It is currently highly biased to downloading 3D models from archive3d.net, unzipping the files, and displaying the .3ds models included.

Functions defined in module `plugins.web`

`plugins.web.download` (*url*, *tgt*)
Download a file with a known url to tgt.

`plugins.web.find3ds` (*fn*)
List the .3ds files in a zip file

`plugins.web.show3ds` (*fn*)
Import and show a 3ds file
Import a 3ds file, render it, and export it to the GUI

`plugins.web.show3dzip` (*fn*)
Show the .3ds models in a zip file.
fn: a zip file containing one or more .3ds models.

`plugins.web.download3d` (*fid*, *fn*)
Download a file from archive3d.net by id number.

`plugins.web.show3d` (*fid*)
Show a model from the archive3d.net database.
fid: string: the archive3d id number

6.4.34 `plugins.webgl` — View and manipulate 3D models in your browser.

This module defines some classes and function to help with the creation of WebGL models. A WebGL model can be viewed directly from a compatible browser (see <http://en.wikipedia.org/wiki/WebGL>).

A WebGL model typically consists out of an HTML file and a Javascript file, possibly also some geometry data files. The HTML file is loaded in the browser and starts the Javascript program, responsible for rendering the WebGL scene.

Classes defined in module `plugins.webgl`

class `plugins.webgl.WebGL` (*name='Scene1'*, *scripts=None*, *bgcolor='white'*, *title=None*, *description=None*, *keywords=None*, *author=None*, *htmlheader=None*, *jsheader=None*, *pgfheader=None*, *sep=""*, *dataformat='.pgf.gz'*, *urlprefix=None*, *gui=True*, *cleanup=False*)

A class to export a 3D scene to WebGL.

Exporting a WebGL model creates:

- a HTML file calling some Javascript files. This file can be viewed in a WebGL enabled browser (Firefox, Chrome, Safari)
- a Javascript file describing the model. This file relies on other Javascript files to do the actual rendering and provide control menus. The default rendering script is <https://net.feops.com/public/webgl/fewgl-0.2.js> and the gui toolkit is https://net.feops.com/public/webgl/xtk_xdat.gui.js. The user can replace them with scripts of his choice.
- a number of geometry files in pyFormex PGF format. These are normally created automatically by the `exportScene` method. The user can optionally add other files.

An example of its usage can be found in the WebGL example.

Parameters:

- *name*: string: the base name for the created HTML and JS files.
- *scripts*: a list of URLs pointing to scripts that will be needed for the rendering of the scene.
- *bgcolor*: string: the background color of the rendered page. This can be the name of a color or a hexadecimal WEB color string like '#FOA0F0'.
- *title*: string: an optional title to be set in the .html file. If not specified, the *name* is used.
- *description*, *keywords*, *author*: strings: if specified, these will be added as meta tags to the generated .html file. The first two have defaults if not specified.
- *htmlheader*: string: a string to be included in the header section of the exported html file. It should be a legal html string.
- *jsheader*: string: a string to be included at the start of the javascript file. It should be a legal javascript text. Usually, it is only used to insert comments, in which case all lines should start with `'/'`, or the whole text should be included in a `'/'`, `'/'` pair.
- *dataformat*: string: the extension of the data files used for storing the geometry. This should be an extension supported by the webgl rendering script. The default (<https://net.feops.com/public/webgl/fewgl-0.2.js>) will always support pyFormex native formats `'pgf'` and `'pgf.gz'`. Some other formats (like `.stl`) may also be supported, but the use of the native formats is preferred, because they are a lot smaller. Default is to use the compressed `'pgf.gz'` format.
- *urlprefix*: string: if specified, this string gets prepended to the exported .js filename in the .html file, and to the datafiles in the exported .js file. This can be used to serve the models from a web server, where the html code is generated dynamically and the model script and data can not be kept in the same location as the html.
- *gui*: bool: if True, a gui will be added to the model, allowing some features to be changed interactively.
- *cleanup*: bool: if True, files in the output directory (the current work directory) starting with the specified base name and having a name structure used by the exported, will be deleted from the file system. Currently, this includes files with name patterns `NAME.html*`, `NAME.js*`, `NAME_*.pgf*` and `NAME_*.stl*`.

addScene (*name=None*, *camera=True*)

Add the current OpenGL scene to the WebGL model.

This method adds all the geometry in the current viewport to the WebGL model. By default it will also add the current camera and export the scene as a completed WebGL model with the given name.

Parameters:

- *name*: a string with the name of the model for the current scene. When multiple scenes are exported, they will be identified by this name. This name is also used as the basename for the exported geometry files. For a single scene export, the name may be omitted, and will then be set equal to the name of

the WebGL exporter. If no name is specified, the model is not exported. This allows the user to add more scenes to the same model, and then to explicitly export it with `exportScene()`.

- *camera*: bool or dict : if True, sets the current viewport camera as the camera in the WebGL model. If False the default camera values of WebGL will be used. If dict the camera values will be taken from the dictionary.

addActor (*actor*)

Add an actor to the model.

The actor's drawable objects are added to the WebGL model as a list. The actor's controller attributes are added to the controller gui.

setCamera (***kargs*)

Set the camera position and direction.

This takes two (optional) keyword parameters:

- *position*=: specify a list of 3 coordinates. The camera will be positioned at that place, and be looking at the origin. This should be set to a proper distance from the scene to get a decent result on first display.
- *upvector*=: specify a list of 3 components of a vector indicating the upwards direction of the camera. The default is [0.,1.,0.].

format_actor (*actor*)

Export an actor in Javascript format for fewgl.

format_gui_controller (*name, attr*)

Format a single controller

format_gui ()

Create the controller GUI script

start_js (*name*)

Export the start of the js file.

Parameters:

- *name*: string: the name of the model that will be shown by default when displaying the webgl html page

This function should only be executed once, before any other export functions. It is called automatically by the first call to `exportScene`.

exportScene (*name=None*)

Export the current OpenGL scene to the WebGL model.

Parameters:

- *name*: string: the name of the model in the current scene. When multiple scenes are exported, they will be identified by this name. This name is also used as the basename for the exported geometry files. For a single scene export, the name may be omitted, and will then be set equal to the name of the WebGL exporter.

export (*body="", createdby=-1*)

Finish the export by creating the html file.

Parameters:

- *body*: an HTML section to be put in the body
- *createdby*: int. If not zero, a logo 'Created by pyFormex' will appear on the page. If > 0, it specifies the width of the logo field. If < 0, the logo will be displayed on its natural width.

Returns the full path of the created html file.

Functions defined in module `plugins.webgl`

- `plugins.webgl.saneSettings` (*k*)
Sanitize sloppy settings for JavaScript output
- `plugins.webgl.properties` (*o*)
Return properties of an object
properties are public attributes (not starting with an `'_'`) that are not callable.
- `plugins.webgl.createdBy` (*width=0*)
Return a div html element with the created by pyFormex logo
- `plugins.webgl.createWebglHtml` (*name*, *scripts=[]*, *bghcolor='white'*, *body=""*, *description='WebGL model'*, *keywords='pyFormex, WebGL'*, *author=""*, *title='pyFormex WebGL model'*, *header=""*)
Create a html file for a WebGL model.
Returns the absolute pathname to the created HTML file.
- `plugins.webgl.surface2webgl` (*S*, *name*, *caption=None*)
Create a WebGL model of a surface
- *S*: TriSurface
 - *name*: basename of the output files
 - *caption*: text to use as caption
- `plugins.webgl.createMultiWebGL` ()
Creates a multimodel WebGL from single WebGL models

6.5 pyFormex OpenGL modules

These modules are responsible for rendering the 3D models and depend on OpenGL. Currently, pyFormex contains two rendering engines. The new engine's modules are located under `pyformex/opengl`.

6.5.1 `opengl.camera` — OpenGL camera handling

Python OpenGL framework for pyFormex

This OpenGL framework is intended to replace (in due time) the current OpenGL framework in pyFormex.

(C) 2013 Benedict Verheghe and the pyFormex project.

Classes defined in module `opengl.camera`

class `opengl.camera.Camera` (*focus=(0.0, 0.0, 0.0)*, *angles=(0.0, 0.0, 0.0)*, *dist=1.0*, *fovy=45.0*, *aspect=1.3333333333333333*, *clip=(0.01, 100.0)*, *perspective=True*, *area=(0.0, 0.0, 1.0, 1.0)*, *locked=False*, *keep_aspect=True*, *tracking=False*)

A camera for 3D model rendering.

The Camera class holds all the camera parameters related to the rendering of a 3D scene onto a 2D canvas. These includes parameters related to camera position and orientation, as well as lens related parameters (opening angle,

front and back clipping planes). The class provides the required matrices to transform the 3D world coordinates to 2D canvas coordinates, as well as a wealth of methods to change the camera settings in a convenient way so as to simulate smooth camera manipulation.

The basic theory of camera handling and 3D rendering can be found in a lot of places on the internet, especially in OpenGL related places. However, while the pyFormex rendering engine is based on OpenGL, the way it stores and handles the camera parameters is more sophisticated than what is usually found in popular tutorials on OpenGL rendering. Therefore we give here a extensive description of how the pyFormex camera handling and 3D to 2D coordinate transformation works.

Camera position and orientation:

The camera viewing line is defined by two points: the position of the camera and the center of the scene the camera is looking at. We use the center of the scene as the origin of a local coordinate system to define the camera position. For convenience, this could be stored in spherical coordinates, as a distance value and two angles: longitude and latitude. Furthermore, the camera can also rotate around its viewing line. We can define this by a third angle, the twist. From these four values, the needed translation vector and rotation matrix for the scene rendering may be calculated.

Inversely however, we can not compute a unique set of angles from a given rotation matrix (this is known as 'gimball lock'). As a result, continuous (smooth) camera rotation by e.g. mouse control requires that the camera orientation be stored as the full rotation matrix, rather than as three angles. Therefore we store the camera position and orientation as follows:

- *ctr*: [*x,y,z*] : the reference point of the camera: this is always a point on the viewing axis. Usually, it is set to the center of the scene you are looking at.
- *dist*: distance of the camera to the reference point.
- *rot*: a 3x3 rotation matrix, rotating the global coordinate system thus that the z-direction is oriented from center to camera.

These values have influence on the Modelview matrix.

Camera lens settings:

The lens parameters define the volume that is seen by the camera. It is described by the following parameters:

- *fovy*: the vertical lens opening angle (Field Of View Y),
- *aspect*: the aspect ratio (width/height) of the lens. The product *fovy* * *aspect* is the horizontal field of view.
- *near, far*: the position of the front and back clipping planes. They are given as distances from the camera and should both be strictly positive. Anything that is closer to the camera than the *near* plane or further away than the *far* plane, will not be shown on the canvas.

Camera methods that change these values will not directly change the Modelview matrix. The `loadModelview()` method has to be called explicitly to make the settings active.

These values have influence on the Projection matrix.

Methods that change the camera position, orientation or lens parameters will not directly change the related Modelview or Projection matrix. They will just flag a change in the camera settings. The changes are only activated by a call to the `loadModelview()` or `loadProjection()` method, which will test the flags to see whether the corresponding matrix needs a rebuild.

The default camera is at distance 1.0 of the center point [0.,0.,0.] and looking in the -z direction. Near and far clipping planes are by default set to 0.1, resp 10 times the camera distance.

Properties:

- *modelview*: Matrix4: the OpenGL Modelview transformation matrix
- *projection*: Matrix4: the OpenGL Projection transformation matrix

settings

Dict to allow save/load of camera

This dict contains all data that allow save and restore of the camera to exactly the same settings (on the same size of Canvas).

modelview

Return the current modelview matrix.

This will recompute the modelview matrix if any camera position parameters have changed.

projection

Return the current projection matrix.

This will recompute the projection matrix if any camera lens parameters have changed.

viewport

Return the camera viewport.

This property can not be changed directly. It should be changed by resizing the parent canvas.

focus

Return the camera reference point (the focus point).

dist

Return the camera distance.

The camera distance is the distance between the camera eye and the camera focus point.

perspective

Return the perspective flag.

If the perspective flag is True, the camera uses a perspective projection. If it is False, the camera uses orthogonal projection.

rot

Return the camera rotation matrix.

angles

Return the camera angles.

Returns a tuple (longitude, latitude, twist) in local camera axes.

upvector

Return the camera up vector

axis

Return a unit vector along the camera axis.

The camera axis points from the focus towards the camera.

setAngles (*angles*, *axes=None*)

Set the rotation angles.

Parameters

- **angles** (*tuple of floats*) – A tuple of three angles (long,lat,twist) in degrees. A value None is also accepted, but has no effect.
- **axes** (*if specified, any number of rotations can be applied.*) –

eye

Return the position of the camera.

lock (*onoff=True*)

Lock/unlock a camera.

When a camera is locked, its position and lens parameters can not be changed. This can e.g. be used in multiple viewports layouts to create fixed views from different angles.

report ()

Return a report of the current camera settings.

dolly (*val*)

Move the camera eye towards/away from the scene center.

This has the effect of zooming. A value > 1 zooms out, a value < 1 zooms in. The resulting enlargement of the view will approximately be 1/val. A zero value will move the camera to the center of the scene. The front and back clipping planes may need adjustment after a dolly operation.

pan (*val, axis=0*)

Rotate the camera around axis through its eye.

The camera is rotated around an axis through the eye point. For axes 0 and 1, this will move the focus, creating a panning effect. The default axis is parallel to the y-axis, resulting in horizontal panning. For vertical panning (axis=1) a convenience alias tilt is created. For axis = 2 the operation is equivalent to the rotate operation.

tilt (*val*)

Rotate the camera up/down around its own horizontal axis.

The camera is rotated around and perpendicular to the plane of the y-axis and the viewing axis. This has the effect of a vertical pan. A positive value tilts the camera up, shifting the scene down. The value is specified in degrees.

move (*dx, dy, dz*)

Move the camera over translation (dx,dy,dz) in global coordinates.

The focus of the camera is moved over the specified translation vector. This has the effect of moving the scene in opposite direction.

setLens (*fovy=None, aspect=None*)

Set the field of view of the camera.

We set the field of view by the vertical opening angle fovy and the aspect ratio (width/height) of the viewing volume. A parameter that is not specified is left unchanged.

resetArea ()

Set maximal camera area.

Resets the camera window area to its maximum values corresponding to the fovy setting, symmetrical about the camera axes.

setArea (*hmin, vmin, hmax, vmax, relative=True, focus=False, center=False, clip=True*)

Set the viewable area of the camera.

Note: Use relative=False and clip=False if you want to set the zoom exactly as in previously recorded values.

zoomArea (*val=0.5, area=None*)

Zoom in/out by shrinking/enlarging the camera view area.

The zoom factor is relative to the current setting. Values smaller than 1.0 zoom in, larger values zoom out.

transArea (*dx, dy*)

Pan by moving the camera area.

dx and *dy* are relative movements in fractions of the current area size.

setClip (*near, far*)

Set the near and far clipping planes

setTracking (*onoff=True*)

Enable/disable coordinate tracking using the camera

setProjection ()

Set the projection matrix.

This computes and sets the camera's projection matrix, depending on the current camera settings. The projection can either be an orthogonal or a perspective one.

The computed matrix is saved as the camera's projection matrix, and the `lensChanged` attribute is set to `False`.

The matrix can be retrieved from the projection attribute, and can be loaded in the GL context with `loadProjection()`.

This function does nothing if the camera is locked.

loadProjection ()

Load the Projection matrix.

If lens parameters of the camera have been changed, the current Projection matrix is rebuilt. Then, the current Projection matrix of the camera is loaded into the OpenGL engine.

pickMatrix (*rect, viewport=None*)

Return a picking matrix.

The picking matrix confines the scope of the normalized device coordinates to a rectangular subregion of the camera viewport. This means that values in the range -1 to +1 are inside the rectangle.

Parameters:

- *rect*: a tuple of 4 floats (*x,y,w,h*) defining the picking region center (*x,y*) and size (*w,h*)
- *viewport*: a tuple of 4 int values (*xmin,ymin,xmax,ymax*) defining the size of the viewport. This is normally left unspecified and set to the camera viewport.

eyeToClip (*x*)

Transform a vertex from eye to clip coordinates.

This transforms the vertex using the current Projection matrix.

It is equivalent with multiplying the homogeneous coordinates with the Projection matrix, but is done here in an optimized way.

clipToEye (*x*)

Transform a vertex from clip to eye coordinates.

This transforms the vertex using the inverse of the current Projection matrix.

It is equivalent with multiplying the homogeneous coordinates with the inverse Projection matrix, but is done here in an optimized way.

lookAt (*focus=None, eye=None, up=None*)

Set the Modelview matrix to look at the specified focus point.

The Modelview matrix is set with the camera positioned at eye and looking at the focus points, while the camera up vector is in the plane of the camera axis (focus-eye) and the specified up vector.

If any of the arguments is left unspecified, the current value will be used.

rotate (*val*, *vx*, *vy*, *vz*)

Rotate the camera around current camera axes.

setModelview (*m=None*, *angles=None*)

Set the Modelview matrix.

The Modelview matrix can be set from one of the following sources:

- if *mat* is specified, it is a 4x4 matrix with a valuable Modelview transformation. It will be set as the current camera Modelview matrix.
- else, if *angles* is specified, it is a sequence of tuples (angle, axis) each of which define a rotation of the camera around an axis through the focus point. The camera Modelview matrix is set from the current camera focus, the current camera distance, and the specified angles/axes. This option is typically used to change the viewing direction of the camera, while keeping the focus point and camera distance.
- else, if the `viewChanged` flag is set, the camera Modelview matrix is set from the current camera focus, the current camera distance, and the current camera rotation matrix. This option is typically used after changing the camera focus point and/or distance, while keeping the current viewing angles.
- else, the current Modelview matrix remains unchanged.

In all cases, if a modelview callback was set, it is called, and the `viewChanged` flag is cleared.

loadModelview ()

Load the Modelview matrix.

If camera positioning parameters have been changed, the current Modelview matrix is rebuild. Then, the current Modelview matrix of the camera is loaded into the OpenGL engine.

toEye (*x*)

Transform a vertex from world to eye coordinates.

This transforms the vertex using the current Modelview matrix.

It is equivalent with multiplying the homogeneous coordinates with the Modelview matrix, but is done here in an optimized way.

toWorld (*x*)

Transform a vertex from eye to world coordinates.

This transforms the vertex using the inverse of the current Modelview matrix.

It is equivalent with multiplying the homogeneous coordinates with the inverse Modelview matrix, but is done here in an optimized way.

toWindow (*x*)

Convert normalized device coordinates to window coordinates

fromWindow (*x*)

Convert window coordinates to normalized device coordinates

toNDC (*x*, *rect=None*)

Convert world coordinates to normalized device coordinates.

The normalized device coordinates (NDC) have *x* and *y* values in the range -1 to +1 for points that are falling within the visible region of the camera.

Parameters:

- *x*: Coords with the world coordinates to be converted

- *rect*: optional, a tuple of 4 values (x,y,w,h) specifying a rectangular subregion of the camera's viewport. The default is the full camera viewport.

The return value is a Coords. The z-coordinate provides depth information.

toNDC1 (*x*, *rect=None*)

This is like toNDC without the perspective divide

This function is useful to compute the vertex position of a 3D point as computed by the vertex shader.

project (*x*)

Map the world coordinates (x,y,z) to window coordinates.

unproject (*x*)

Map the window coordinates x to object coordinates.

inside (*x*, *rect=None*, *return_depth=False*)

Test if points are visible inside camera.

Parameters:

- *rect*: optional, a tuple of 4 values (x,y,w,h) specifying a rectangular subregion of the camera's viewport. The default is the full camera viewport.
- *return_depth*: if True, also returns the the z-depth of the points.

Returns a boolean array with value 1 (True) for the points that are projected inside the rectangular are of the camera. If *return_depth* is True, a second array with the z-depth value of all the points is returned.

config ()

Return a Config with the settings to be saved for a restore

save (*filename*)

Save the camera settings to file

loadConfig (*config*)

Load the camera settings from a Config or dict

apply (*config*)

Load the camera settings from a Config or dict

load (*filename*)

Load the camera settings from file

loadModelView ()

Load the Modelview matrix.

If camera positioning parameters have been changed, the current Modelview matrix is rebuild. Then, the current Modelview matrix of the camera is loaded into the OpenGL engine.

Functions defined in module `opengl.camera`

`opengl.camera.gl_projection` ()

Get the OpenGL projection matrix

`opengl.camera.gl_modelview` ()

Get the OpenGL modelview matrix

`opengl.camera.gl_viewport` ()

Get the OpenGL viewport

`opengl.camera.gl_loadmodelview` (*m*)

Load the OpenGL modelview matrix

`opengl.camera.gl_loadprojection` (*m*)
Load the OpenGL projection matrix

`opengl.camera.gl_depth` (*x, y*)
Read the depth value of the pixel at (*x,y*)

`opengl.camera.normalize` (*x, w*)
Normalized coordinates inside a window.

Parameters:

- *x*: an (np,nc) array with coordinates.
- *w*: a (2,nc) array with minimal and width of the window that will be mapped to the range -1..1.

Returns an array with the *x* values linearly remapped thus that values *w*[0] become -1 and values *w*[0]+*w*[1] become +1.

`opengl.camera.denormalize` (*x, w*)
Map normalized coordinates to fit a window

Parameters:

- *x*: an (np,nc) array with normalized coordinates.
- *w*: a (2,nc) array with minimal and width values of the window.

Returns an array with the *x* values linearly remapped thus that values -1 coincide with the minimum window values and +1 with the minimum+width values.

`opengl.camera.perspective_matrix` (*left, right, bottom, top, near, far*)
Create a perspective Projection matrix.

`opengl.camera.orthogonal_matrix` (*left, right, bottom, top, near, far*)
Create an orthogonal Projection matrix.

`opengl.camera.pick_matrix` (*x, y, w, h, viewport*)
Create a pick Projection matrix

6.5.2 `opengl.canvas` — This implements an OpenGL drawing widget for painting 3D scenes.

Classes defined in module `opengl.canvas`

class `opengl.canvas.Light` (*ambient=0.0, diffuse=0.0, specular=0.0, position=[0.0, 0.0, 1.0], enabled=True*)
A class representing an OpenGL light.

The light can emit 3 types of light: ambient, diffuse and specular, which can have different color and are all off by default.

class `opengl.canvas.LightProfile` (*ambient, lights*)
A lightprofile contains all the lighting parameters.

Currently this consists off: - *ambient*: the global ambient lighting (currently a float) - *lights*: a list of 1 to 4 Lights

class `opengl.canvas.CanvasSettings` (***kargs*)
A collection of settings for an OpenGL Canvas.

The canvas settings are a collection of settings and default values affecting the rendering in an individual viewport. There are two type of settings:

- mode settings are set during the initialization of the canvas and can/should not be changed during the drawing of actors and decorations;
- default settings can be used as default values but may be changed during the drawing of actors/decorations: they are reset before each individual draw instruction.

Currently the following mode settings are defined:

- `bgmode`: the viewport background color mode
- `bgcolor`: the viewport background color: a single color or a list of colors (max. 4 are used).
- `bgimage`: background image filename
- `slcolor`: the highlight color
- `alphablend`: boolean (transparency on/off)

The list of default settings includes:

- `fgcolor`: the default drawing color
- `bkcolor`: the default backface color
- `colormap`: the default color map to be used if color is an index
- `bkormap`: the default color map to be used if `bkcolor` is an index
- `smooth`: boolean (smooth/flat shading)
- `lighting`: boolean (lights on/off)
- `culling`: boolean
- `transparency`: float (0.0..1.0)
- `avgnormals`: boolean
- `wiremode`: integer -3..3
- `pointsize`: the default size for drawing points
- `marksize`: the default size for drawing markers
- `linewidth`: the default width for drawing lines

Any of these values can be set in the constructor using a keyword argument. All items that are not set, will get their value from the configuration file(s).

reset (*d={}*)

Reset the CanvasSettings to its defaults.

The default values are taken from the configuration files. An optional dictionary may be specified to override (some of) these defaults.

update (*d, strict=True*)

Update current values with the specified settings

Returns the sanitized update values.

classmethod checkDict (*dict, strict=True*)

Transform a dict to acceptable settings.

activate ()

Activate the default canvas settings in the GL machine.

class `opengl.canvas.Canvas` (*settings={}*)

A canvas for OpenGL rendering.

The Canvas is a class holding all global data of an OpenGL scene rendering. It always contains a Scene with the actors and decorations that are drawn on the canvas. The canvas has a Camera holding the viewing parameters needed to project the scene on the canvas. Settings like colors, line types, rendering mode and the lighting information are gathered in a CanvasSettings object. There are attributes for some special purpose decorations (Triade, Grid) that can not be handled by the standard drawing and Scene changing functions.

The Canvas class does not however contain the viewport size and position. The class is intended as a mixin to be used by some OpenGL widget class that will hold this information (such as the QtCanvas class).

Important note: the Camera object is not initialized by the class initialization. The derived class initialization should therefore explicitly call the *initCamera* method before trying to draw anything to the canvas. This is because initializing the camera requires a working OpenGL format, which is only established by the derived OpenGL widget.

sceneBbox ()

Return the bbox of all actors in the scene

resetDefaults (*dict={}*)

Return all the settings to their default values.

setAmbient (*ambient*)

Set the global ambient lighting for the canvas

setMaterial (*matname*)

Set the default material light properties for the canvas

resetLighting ()

Change the light parameters

resetOptions ()

Reset the Drawing options to default values

setOptions (*d*)

Set the Drawing options to some values

setRenderMode (*mode, lighting=None*)

Set the rendering mode.

This sets or changes the rendermode and lighting attributes. If lighting is not specified, it is set depending on the rendermode.

If the canvas has not been initialized, this merely sets the attributes `self.rendermode` and `self.settings.lighting`. If the canvas was already initialized (it has a camera), and one of the specified settings is different from the existing, the new mode is set, the canvas is re-initialized according to the newly set mode, and everything is redrawn with the new mode.

setWireMode (*state, mode=None*)

Set the wire mode.

This toggles the drawing of edges on top of 2D and 3D geometry. State is either True or False, mode is 1, 2 or 3 to switch:

1: all edges 2: feature edges 3: border edges

If no mode is specified, the current wiremode is used. A negative value inverses the state.

setToggle (*attr, state*)

Set or toggle a boolean settings attribute

Furthermore, if a Canvas method `do_ATTR` is defined, it will be called with the old and new toggle state as a parameter.

do_wiremode (*state, oldstate*)

Change the wiremode

do_alphablend (*state, oldstate*)

Toggle alphablend on/off.

do_lighting (*state, oldstate*)

Toggle lights on/off.

setLineWidth (*lw*)

Set the linewidth for line rendering.

setLineStipple (*repeat, pattern*)

Set the linestipple for line rendering.

setPointSize (*sz*)

Set the size for point drawing.

setBackground (*color=None, image=None*)

Set the color(s) and image.

Change the background settings according to the specified parameters and set the canvas background accordingly. Only (and all) the specified parameters get a new value.

Parameters:

- *color*: either a single color, a list of two colors or a list of four colors.
- *image*: an image to be set.

createBackground ()

Create the background object.

setFgColor (*color*)

Set the default foreground color.

setSlColor (*color*)

Set the highlight color.

setTriade (*pos='lb', siz=100, triade=None*)

Set the Triade for this canvas.

Display the Triade on the current viewport. The Triade is a reserved Actor displaying the orientation of the global axes. It has special methods to show/hide it. See also: `removeTriade()`, `hasTriade()`

Parameters:

- *pos*: string of two characters. The characters define the horizontal (one of 'l', 'c', or 'r') and vertical (one of 't', 'c', 'b') position on the camera's viewport. Default is left-bottom.
- *siz*: float: intended size (in pixels) of the triade.
- *triade*: None or Geometry: defines the Geometry to be used for representing the global axes.

If None, use the previously set triade, or set a default if no previous.

If Geometry, use this to represent the axes. To be useful and properly displayed, the Geometry's bbox should be around `[(-1,-1,-1),(1,1,1)]`. Drawing attributes may be set on the Geometry to influence the appearance. This allows to fully customize the Triade.

removeTriade ()

Remove the Triade from the canvas.

Remove the Triade from the current viewport. The Triade is a reserved Actor displaying the orientation of the global axes. It has special methods to draw/undraw it. See also: `setTriade()`, `hasTriade()`

hasTriade()

Check if the canvas has a Triade displayed.

Return True if the Triade is currently displayed. The Triade is a reserved Actor displaying the orientation of the global axes. See also: `setTriade()`, `removeTriade()`

setGrid(*grid=None*)

Set the canvas Grid for this canvas.

Display the Grid on the current viewport. The Grid is a 2D grid fixed to the canvas. See also: `removeGrid()`, `hasGrid()`

Parameters:

- *grid*: None or Actor: The Actor to be displayed as grid. This will normal be a Grid2D Actor.

removeGrid()

Remove the Grid from the canvas.

Remove the Grid from the current viewport. The Grid is a 2D grid fixed to the canvas. See also: `setGrid()`, `hasGrid()`

hasGrid()

Check if the canvas has a Grid displayed.

Return True if the Grid is currently displayed. The Grid is a 2D grid fixed to the canvas. See also: `setGrid()`, `removeGrid()`

clearCanvas()

Clear the canvas to the background color.

drawit(*a*)

_Perform the drawing of a single item

setDefault()

Activate the canvas settings in the GL machine.

overrideMode(*mode*)

Override some settings

reset()

Reset the rendering engine.

The rendering machine is initialized according to self.settings: - self.rendermode: one of - self.lighting

glinit()

Initialize the rendering engine.

glupdate()

Flush all OpenGL commands, making sure the display is updated.

draw_sorted_objects(*objects, alphablend*)

Draw a list of sorted objects through the fixed pipeline.

If alphablend is True, objects are separated in opaque and transparent ones, and the opaque are drawn first. Inside each group, ordering is preserved. Else, the objects are drawn in the order submitted.

display()

(Re)display all the actors in the scene.

This should e.g. be used when actors are added to the scene, or after changing camera position/orientation or lens.

begin_2D_drawing ()

Set up the canvas for 2D drawing on top of 3D canvas.

The 2D drawing operation should be ended by calling `end_2D_drawing`. It is assumed that you will not try to change/refresh the normal 3D drawing cycle during this operation.

end_2D_drawing ()

Cancel the 2D drawing mode initiated by `begin_2D_drawing`.

setCamera (bbox=None, angles=None, orient='xy')

Sets the camera looking under angles at `bbox`.

This function sets the camera parameters to view the specified `bbox` volume from the specified viewing direction.

Parameters:

- *bbox*: the `bbox` of the volume looked at
- *angles*: the camera angles specifying the viewing direction. It can also be a string, the key of one of the predefined camera directions

If no angles are specified, the viewing direction remains constant. The scene center (camera focus point), camera distance, fovy and clipping planes are adjusted to make the whole `bbox` viewed from the specified direction fit into the screen.

If no `bbox` is specified, the following remain constant: the center of the scene, the camera distance, the lens opening and aspect ratio, the clipping planes. In other words the camera is moving on a spherical surface and keeps focusing on the same point.

If both are specified, then first the scene center is set, then the camera angles, and finally the camera distance.

In the current implementation, the lens fovy and aspect are not changed by this function. Zoom adjusting is performed solely by changing the camera distance.

project (x, y, z, locked=False)

Map the object coordinates (x,y,z) to window coordinates.

unproject (x, y, z, locked=False)

Map the window coordinates (x,y,z) to object coordinates.

zoom (f, dolly=True)

Dolly zooming.

Zooms in with a factor *f* by moving the camera closer to the scene. This does not change the camera's FOV setting. It will change the perspective view though.

zoomRectangle (x0, y0, x1, y1)

Rectangle zooming.

Zooms in/out by changing the area and position of the visible part of the lens. Unlike `zoom()`, this does not change the perspective view.

x0,y0,x1,y1 are pixel coordinates of the lower left and upper right corners of the area of the lens that will be mapped on the canvas viewport. Specifying values that lead to smaller width/height will zoom in.

zoomCentered (w, h, x=None, y=None)

Rectangle zooming with specified center.

This is like `zoomRectangle`, but the zoom rectangle is specified by its center and size, which may be more appropriate when using off-center zooming.

zoomAll ()

Rectangle zoom to make full scene visible.

saveBuffer ()

Save the current OpenGL buffer

showBuffer ()

Show the saved buffer

add_focus_rectangle (*color=(1.0, 0.2, 0.4)*)

Draw the focus rectangle.

draw_cursor (*x, y*)

draw the cursor

pick_actors ()

Set the list of actors inside the pick_window.

pick_parts (*obj_type, store_closest=False*)

Set the list of actor parts inside the pick_window.

obj_type can be 'element', 'face', 'edge' or 'point'. 'face' and 'edge' are only available for Mesh type geometry.

The picked object numbers are stored in `self.picked`. If `store_closest==True`, the closest picked object is stored in as a tuple (`[actor,object]`, `distance`) in `self.picked_closest`

A list of actors from which can be picked may be given. If so, the resulting keys are indices in this list. By default, the full actor list is used.

pick_elements ()

Set the list of actor elements inside the pick_window.

pick_points ()

Set the list of actor points inside the pick_window.

pick_edges ()

Set the list of actor edges inside the pick_window.

pick_faces ()

Set the list of actor faces inside the pick_window.

pick_numbers ()

Return the numbers inside the pick_window.

highlightActors (*K*)

Highlight a selection of actors on the canvas.

K is Collection of actors as returned by the `pick()` method. `colormap` is a list of two colors, for the actors not in, resp. in the Collection *K*.

removeHighlight ()

Remove a highlight or a list thereof from the 3D scene.

Without argument, removes all highlights from the scene.

Functions defined in module `opengl.canvas`

`opengl.canvas.gl_pickbuffer` ()

Return a list of the 2nd numbers in the openGL pick buffer.

`opengl.canvas.glLineStipple` (*factor, pattern*)

Set the line stipple pattern.

When drawing lines, OpenGL can use a stipple pattern. The stipple is defined by two values: a pattern (on/off) of maximum 16 bits, used on the pixel level, and a multiplier factor for each bit.

If factor <= 0, the stippling is disabled.

`opengl.canvas.glSmooth` (*smooth=True*)

Enable smooth shading

`opengl.canvas.glFlat` ()

Disable smooth shading

`opengl.canvas.onOff` (*onoff*)

Convert On/Off strings to a boolean

`opengl.canvas.glEnable` (*facility, onoff*)

Enable/Disable an OpenGL facility, depending on onoff value

facility is an OpenGL facility. onoff can be True or False to enable, resp. disable the facility, or None to leave it unchanged.

`opengl.canvas.glViewport` ()

Return the current OpenGL Viewport

Returns a tuple x,y,w,h specifying the position and size of the current OpenGL viewport (in pixels).

`opengl.canvas.extractCanvasSettings` (*d*)

Split a dict in canvas settings and other items.

Returns a tuple of two dicts: the first one contains the items that are canvas settings, the second one the rest.

6.5.3 `opengl.decors` — Decorations for the OpenGL canvas.

This module contains a collection of predefined decorations that can be useful additions to a geometry scene rendering.

Classes defined in module `opengl.decors`

class `opengl.decors.BboxActor` (*bbox, **kargs*)

Draws a bounding bbox.

A bounding box is a hexaeder in global axes. The hexaeder is drawn in wireframe mode with default color black.

class `opengl.decors.Rectangle` (*x1, y1, x2, y2, **kargs*)

A 2D-rectangle on the canvas.

class `opengl.decors.Line` (*x1, y1, x2, y2, **kargs*)

A 2D-line on the canvas.

Parameters:

- *x1, y1, x2, y2*: floats: the viewport coordinates of the endpoints of the line
- *kargs*: keyword arguments to be passed to the `Actor`.

class `opengl.decors.Lines` (*data, color=None, linewidth=None, **kargs*)

A collection of straight lines on the canvas.

Parameters:

- *data*: data that can initialize a 2-plex Formex: the viewport coordinates of the 2 endpoints of the n lines. The third coordinate is ignored.
- *kargs*: keyword arguments to be passed to the `Actor`.

class `opengl.decor.Grid2D` (*x1*, *y1*, *x2*, *y2*, *nx=1*, *ny=1*, *lighting=False*, *rendertype=2*, ***kargs*)
 A 2D-grid on the canvas.

class `opengl.decor.ColorLegend` (*colorscale*, *ncolors*, *x*, *y*, *w*, *h*, *ngrid=0*, *linewidth=None*, *nlabel=-1*, *size=18*, *font=None*, *textcolor=None*, *dec=2*, *scale=0*, *lefttext=False*, ***kargs*)

A labeled colorscale legend.

When showing the distribution of some variable over a domain by means of a color encoding, the viewer expects some labeled colorscale as a guide to decode the colors. The `ColorLegend` decoration provides such a color legend. This class only provides the visual details of the scale. The conversion of the numerical values to the matching colors is provided by the `colorscale.ColorScale` class.

Parameters:

- *colorscale*: a `colorscale.ColorScale` instance providing conversion between numerical values and colors
- *ncolors*: int: the number of different colors to use.
- *x,y,w,h*: four integers specifying the position and size of the color bar rectangle
- *ngrid*: int: number of intervals for the grid lines to be shown. If > 0, grid lines are drawn around the color bar and between the *ngrid* intervals. If = 0, no grid lines are drawn. If < 0 (default), the value is set equal to the number of colors or to 0 if this number is higher than 50.
- *linewidth*: float: width of the grid lines. If not specified, the current canvas line width is used.
- *nlabel*: int: number of intervals for the labels to be shown. If > 0, labels will be displayed at *nlabel* interval borders, if possible. The number of labels displayed thus will be *nlabel*+1, or less if the labels would otherwise be too close or overlapping. If 0, no labels are shown. If < 0 (default), a default number of labels is shown.
- *size*: font size to be used for the labels
- *font*: font to be used for the labels. It can be a `texttext.FontTexture` or a string with the path to a monospace .ttf font. If unspecified, the default font is used.
- *dec*: int: number of decimals to be used in the labels
- *scale*: int: exponent of 10 for the scaling factor of the label values. The displayed values will be equal to the real values multiplied with $10^{**scale}$.
- *lefttext*: bool: if True, the labels will be drawn to the left of the color bar. The default is to draw the labels at the right.

Some practical guidelines:

- Large numbers of colors result in a quasi continuous color scheme.
- With a high number of colors, grid lines disturb the image, so either use *ngrid=0* or *ngrid=* to only draw a border around the colors.
- With a small number of colors, set *ngrid = len(colorlegend.colors)* to add gridlines between each color. Without it, the individual colors in the color bar may seem to be not constant, due to an optical illusion. Adding the grid lines reduces this illusion.
- When using both grid lines and labels, set both *ngrid* and *nlabel* to the same number or make one a multiple of the other. Not doing so may result in a very confusing picture.

- The best practice is to either use a low number of colors (≤ 20) and the default `ngrid` and `nlabel`, or to use a high number of colors (≥ 200) and the default values or a low value for `nlabel`.

The *ColorScale* example script provides opportunity to experiment with different settings.

Functions defined in module `opengl.decor`s

`opengl.decor`s.**Grid** ($nx=(1, 1, 1)$, $ox=(0.0, 0.0, 0.0)$, $dx=(1.0, 1.0, 1.0)$, $lines='b'$, $planes='b'$, $linecolor=(0.0, 0.0, 0.0)$, $planecolor=(1.0, 1.0, 1.0)$, $alpha=0.3$, ****kargs**)
Creates a (set of) grid(s) in (some of) the coordinate planes.

Parameters:

- *nx*: a list of 3 integers, specifying the number of divisions of the grid in the three coordinate directions. A zero value may be specified to avoid the grid to extend in that direction. Thus, setting the last value to zero will result in a planar grid in the xy-plane.
- *ox*: a list of 3 floats: the origin of the grid.
- *dx*: a list of 3 floats: the step size in each coordinate direction.
- *planes*: one of 'first', 'box', 'all', 'no' (the string can be shortened to the first character): specifies how many planes are drawn in each direction: 'f' only draws the first, 'b' draws the first and the last, resulting in a box, 'a' draws all planes, 'n' draws no planes.

Returns a list with up to two Meshes: the planes, and the lines.

6.5.4 `opengl.drawable` — OpenGL rendering objects for the new OpenGL2 engine.

Classes defined in module `opengl.drawable`

class `opengl.drawable.Drawable` (*parent*, ****kargs**)

Base class for objects that can be rendered by the OpenGL engine.

This is the basic drawable object in the pyFormex OpenGL rendering engine. It collects all the data that are needed to properly described any object to be rendered by the OpenGL shader programs. It has a multitude of optional attributes allowing it to describe many very different objects and rendering situations.

This class is however not intended to be directly used to construct an object for rendering. The *Actor* class and its multiple subclasses should be used for that purpose. The Actor classes provide an easier and more logical interface, and more powerful at the same time, since they can be compound: one Actor can hold multiple Drawables.

The elementary objects that can be directly drawn by the shader programs are more simple, yet very diverse. The Drawable class collects all the data that are needed by the OpenGL engine to do a proper rendering of the object. It this represents a single, versatile interface of the Actor classes with the GPU shader programs.

The versatility comes from the *Attributes* base class, with an unlimited set of attributes. Any undefined attribute just returns None. Some of the most important attributes are described hereafter:

- *rendertype*: int: the type of rendering process that will be applied by the rendering engine to the Drawable:
 - 0: A full 3D Actor. The Drawable will be rendered in full 3D with** all active capabilities, such as camera rotation, projection, rendermode, lighting. The full object undergoes the camera transformations, and thus will appear as a 3D object in space. The object's vertices are defined in 3D world coordinates. Used in: *Actor*.

- 1: A 2D object (often a text or an image) inserted at a 3D position.** The 2D object always keeps its orientation towards the camera. When the camera changes, the object can change its position on the viewport, but the object itself looks the same. This can be used to add annotations to (parts of) a 3D object. The object is defined in viewport coordinates, the insertion points are in 3D world coordinates. Used in: `texttext.Text`.
- 2: A 2D object inserted at a 2D position. Both object and position** are defined in viewport coordinates. The object will take a fixed position on the viewport. This can be used to add decorations to the viewport (like color legends and background images). Used in: `decors.ColorLegend`.
- 3: Like 2, but with special purpose. These Drawables are not part of** the user scene, but used for system purposes (like setting the background color, or adding an elastic rectangle during mouse picking). Used in: `Canvas.createBackground()`.
- 1: Like 1, but with different insertion points for the multiple items** in the object. Used to place a list of marks at a list of points. Used in: `texttext.Text`.
- 2: A 3D object inserted at a 2D position. The 3D object will rotate** when the camera changes directions, but it will always be located on the same position of the viewport. This is normally used to display a helper object showing the global axis directions. Used in: `decors.Triade`.

The initialization of a Drawable takes a single parameter: *parent*, which is the Actor that created the Drawable. All other parameters should be keyword arguments, and are stored as attributes in the Drawable.

Methods:

- *prepare*. . . : creates sanitized and derived attributes/data. Its action depends on current canvas settings: `mode`, `transparent`, `avgnormals`
- *render*: push values to shader and render the object: depends on canvas and renderer.
- *pick*: fake render to be used during pick operations
- *str*: format the full data set of the Drawable

prepareColor ()

Prepare the colors for the shader.

changeVertexColor (color)

Change the vertex color buffer of the object.

This is experimental!!! Just make sure that the passed data have the correct shape!

prepareTexture ()

Prepare texture and texture coords

prepareSubelems ()

Create an index buffer to draw subelements

This is always used for `nplex > 3`, but also to draw the edges for `nplex=3`.

render (renderer)

Render the geometry of this object

pick (renderer)

Pick the geometry of this object

class `opengl.drawable.BaseActor (**kargs)`

Base class for all drawn objects (Actors) in pyFormex.

This defines the interface for all drawn objects, but does not implement any drawn objects. Drawable objects should be instantiated from the derived classes. Currently, we have the following derived classes:

Actor: a 3-D object positioned and oriented in the 3D scene. Defined in `actors.py`.

Mark: an object positioned in 3D scene but not undergoing the camera axis rotations and translations. It will always appear the same to the viewer, but will move over the screen according to its 3D position. Defined in marks.py.

Decor: an object drawn in 2D viewport coordinates. It will unchangeably stick on the viewport until removed. Defined in decors.py.

The BaseActor class is just an Attributes dict storing all the rendering parameters, and providing defaults from the current canvas drawoptions for the essential parameters that are not specified.

Additional parameters can be set at init time or later using the update method. The specified parameters are sanitized before being stored.

Arguments processed by the base class:

- *marksize*: force to float and also copied as *pointsize*

setLineWidth (*linewidth*)

Set the linewidth of the Drawable.

setLineStipple (*linestipple*)

Set the linewidth of the Drawable.

setColor (*color=None, colormap=None, ncolors=1*)

Set the color of the Drawable.

setTexture (*texture*)

Set the texture data of the Drawable.

class `opengl.drawable.Actor` (*obj, **kargs*)

Proposal for drawn objects

`__init__`: store all static values: attributes, geometry, vbo's prepare: creates sanitized and derived attributes/data
render: push values to shader and render the object

`__init__` is only dependent on input attributes and geometry

prepare may depend on current canvas settings: mode, transparent, avgnormals

render depends on canvas and renderer

If the actor does not have a name, it will be given a default one.

The actor has the following attributes, initialized or computed on demand

coords

Return the fused coordinates of the object

elems

Return the original elems of the object

fcoords

Return the full coordinate set of the object

ndim

Return the dimensionality of the object.

faces

Return the elems of the object as they will be drawn

This returns a 2D index in a single element. All elements should have compatible node numberings.

edges

Return the edges of the object as they will be drawn

This returns a 2D index in a single element. All elements should have compatible node numberings.

b_normals

Return individual normals at all vertices of all elements

b_avgnormals

Return averaged normals at the vertices

prepare (*canvas*)

Prepare the attributes for the renderer.

This sanitizes and completes the attributes for the renderer. Since the attributes may be dependent on the rendering mode, this method is called on each mode change.

changeMode (*canvas*)

Modify the actor according to the specified mode

fullElems ()

Return an elems index for the full coords set

subElems (*nSel=None, esel=None*)

Create an index for the drawable subelems

This index always refers to the full coords (fcoords).

The esel selects the elements to be used (default all).

The nsel selects (possibly multiple) parts from each element. The selector is 2D (nsubelems, nsubplex). It is applied on all selected elements

If both esel and esel are None, returns None

highlighted ()

Return True if the Actor is highlighted.

The highlight can be full (self.highlight=1) or partial (self._highlight is not None).

removeHighlight ()

Remove the highlight for the current actor.

Remove the highlight (whether full or partial) from the actor.

setHighlight ()

Add full highlighting of the actor.

This makes the whole actor being drawn in the highlight color.

addHighlightElements (*sel=None*)

Add a highlight for the selected elements. Default is all.

addHighlightPoints (*sel=None*)

Add a highlight for the selected points. Default is all.

okColor (*color, colormap=None*)

Compute a color usable by the shader.

The shader (currently) only supports 3*float type of colors:

- None
- single color (separate for front and back faces)
- vertex colors

setAlpha (*alpha, bkalpha=None*)

Set the Actors alpha value.

setTexture (*texture, texcoords=None, texmode=None*)

Set the texture data of the Drawable.

render (*renderer*)

Render the geometry of this object

inside (*camera, rect=None, mode='actor', sel='any', return_depth=False*)

Test whether the actor is rendered inside rect of camera.

Parameters:

- *camera*: a Camera that has been set up properly. Usually it will be the current canvas camera, `pf.canvas.camera`.
- *rect*: a tuple of 4 values (x,y,w,h) specifying a rectangular subregion of the camera's viewport. The default is the full camera viewport.
- *mode*: the testing mode. Currently defined modes:
 - 'actor' (default): test if the actor is (partly) inside
 - 'element': test which elements of the actor are inside
 - 'point': test which vertices of the actor are inside
- *sel*: either 'all' or 'any'. This is not used with 'point' mode. It specifies whether all or any of the points of the actor, element, ... should be inside the rectangle in order to be flagged as a positive.

The return value depends on the mode:

- 'actor': True or False
- 'element': the indices of the elements inside
- 'point': the indices of the vertices inside

If *return_depth* is True, a second value is returned, with the z-depth value of all the objects inside.

`opengl.drawable.GeomActor`
alias of `opengl.drawable.Actor`

6.5.5 `opengl.matrix` — `opengl/matrix.py`

Python OpenGL framework for pyFormex

This OpenGL framework is intended to replace (in due time) the current OpenGL framework in pyFormex.

(C) 2013 Benedict Verhegghe and the pyFormex project.

Classes defined in module `opengl.matrix`

class `opengl.matrix.Vector4`

One or more homogeneous coordinates

class `opengl.matrix.Matrix4`

A 4x4 transformation matrix for homogeneous coordinates.

The matrix is to be used with post-multiplication on row vectors (i.e. OpenGL convention).

- *data*: if specified, should be a (4,4) float array or compatible. Else a 4x4 identity matrix is created.

Example:

```
>>> I = Matrix4()
>>> print(I)
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
```

We can first scale and then rotate, or first rotate and then scale (with another scaling factor):

```
>>> a = I.scale([4.,4.,4.]).rotate(45.,[0.,0.,1.])
>>> b = I.rotate(45.,[0.,0.,1.]).scale([2.,2.,2.])
>>> print(a)
[[ 0.  4.  0.  0.]
 [-4.  0.  0.  0.]
 [ 0.  0.  8.  0.]
 [ 0.  0.  0.  1.]]
>>> (a==b).all()
True
```

gl()

Get the transformation matrix as a ‘ready-to-use’-gl version.

Returns the (4,4) Matrix as a rowwise flattened array of type float32.

Example:

```
>>> Matrix4().gl()
Matrix4([ 1.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  0.,  1.,  0.,  0.,
          0.,  0.,  0.,  1.])
```

identity()

Reset the matrix to a 4x4 identity matrix.

translate(vector)

Translate a 4x4 matrix by a (3,) vector.

- *vector*: (3,) float array: the translation vector

Changes the Matrix in place and also returns the result

Example:

```
>>> Matrix4().translate([1.,2.,3.])
Matrix4([[ 1.,  0.,  0.,  0.],
         [ 0.,  1.,  0.,  0.],
         [ 0.,  0.,  1.,  0.],
         [ 1.,  2.,  3.,  1.]])
```

rotate(angle, axis=None)

Rotate a Matrix4.

The rotation can be specified by

- an angle and axis,
- a 3x3 rotation matrix,
- a 4x4 transformation matrix (Matrix4).

Parameters:

- **angle: float: the rotation angle.** A 3x3 or 4x4 matrix may be give instead, to directly specify the roation matrix.
- **axis: int or (3,) float: the axis to rotate around**

Changes the Matrix in place and also returns the result.

Example:

```
>>> Matrix4().rotate(90., [0., 1., 0.])
Matrix4([[ 0.,  0., -1.,  0.],
         [ 0.,  1.,  0.,  0.],
         [ 1.,  0.,  0.,  0.],
         [ 0.,  0.,  0.,  1.]])
```

scale (*vector*)

Scale a 4x4 matrix by a (3,) vector.

- **vector: (3,) float array: the scaling vector**

Changes the Matrix in place and also returns the result

Example:

```
>>> Matrix4().scale([1., 2., 3.])
Matrix4([[ 1.,  0.,  0.,  0.],
         [ 0.,  2.,  0.,  0.],
         [ 0.,  0.,  3.,  0.],
         [ 0.,  0.,  0.,  1.]])
```

swapRows (*row1, row2*)

Swap two rows.

- **row1, row2: index of the rows to swap**

swapCols (*col1, col2*)

Swap two columns.

- **col1, col2: index of the columns to swap**

inverse ()

Return the inverse matrix

transform (*x*)

Transform a vertex using this matrix.

- **x: a (3,) or (4,) vector.**

If the vector has length 4, it holds homogeneous coordinates, and the result is the dot product of the vector with the Matrix: $x * M$. If the vector has length 3, the 4th homogeneous coordinate is assumed to be 1, and the product is computed in an optimized way.

invtransform (*x*)

Transform a vertex with the inverse of this matrix.

- **x: a (3,) or (4,) vector.**

transinv ()

Return the transpose of the inverse.

Functions defined in module `opengl.matrix`

6.5.6 `opengl.objectdialog` — Interactive modification of the rendered scene

This module contains some tools that can be used to interactively modify the rendered scene.

Functions defined in module `opengl.objectdialog`

`opengl.objectdialog.objectDialog` (*obj=None, unlike='object_'*)

Create an interactive object dialog for drawn objects

obj is a single drawn object or a list thereof. If no objects are specified, the current scene actors are used. A drawn object is the return value of a `draw()` function call.

6.5.7 `opengl.renderer` — `opengl/renderer.py`

Python OpenGL framework for pyFormex

This OpenGL framework is intended to replace (in due time) the current OpenGL framework in pyFormex.

(C) 2013 Benedict Verhegghe and the pyFormex project.

Functions defined in module `opengl.renderer`

`opengl.renderer.front` (*actorlist*)

Return the actors from the list that have `ontop=True`

`opengl.renderer.back` (*actorlist*)

Return the actors from the list that have `ontop=False`

6.5.8 `opengl.sanitize` — Sanitize data before rendering.

The pyFormex drawing functions were designed to require a minimal knowledge of OpenGL and rendering principles in general. They allow the user to specify rendering attributes in a simple, user-friendly, sometimes even sloppy way. This module contains some functions to sanitize the user input into the more strict attributes required by the OpenGL rendering engine.

These functions are generally not intended for direct used by the user, but for use in the `opengl` rendering functions.

Functions defined in module `opengl.sanitize`

`opengl.sanitize.saneFloat` (*value*)

Return a float value or None.

If *value* can be converted to float, the float is returned, else None.

`opengl.sanitize.saneLineWidth` (*value*)

Return a float value or None.

If *value* can be converted to float, the float is returned, else None.

`opengl.sanitize.saneLineStipple` (*stipple*)

Return a sane line stipple tuple.

A line stipple tuple is a tuple (factor,pattern) where pattern defines which pixels are on or off (maximum 16 bits), factor is a multiplier for each bit.

`opengl.sanitize.saneColor` (*color=None*)

Return a sane color array derived from the input color.

A sane color is one that will be usable by the draw method. The input value of color can be either of the following:

- None: indicates that the default color will be used,
- a single color value in a format accepted by `colors.GLcolor`,
- a tuple or list of such colors,
- a (3,) shaped array of RGB values, ranging from 0.0 to 1.0,
- an (n,3) shaped array of RGB values,
- a (4,) shaped array of RGBA values, ranging from 0.0 to 1.0,
- an (n,4) shaped array of RGBA values,
- an (n,) shaped array of integer color indices.

The return value is one of the following: - None, indicating no color (current color will be used), - a float array with shape (3/4,), indicating a single color, - a float array with shape (n,3/4), holding a collection of colors, - an integer array with shape (n,), holding color index values.

!! Note that a single color can not be specified as integer RGB values. A single list of integers will be interpreted as a color index ! Turning the single color into a list with one item will work though. `[[0, 0, 255]]` will be the same as `['blue']`, while `[0,0,255]` would be a color index with 3 values.

```
>>> print (saneColor('red'))
[ 1.  0.  0.]
```

```
>>> print (saneColor('grey90'))
[ 0.9  0.9  0.9]
```

`opengl.sanitize.saneColorArray` (*color, shape*)

Makes sure the shape of the color array is compatible with shape.

Parameters:

- *color*: set of colors
- *shape*: (nelems,nplex) tuple

A compatible `color.shape` is equal to `shape` or has either or both of its dimensions equal to 1. Compatibility is enforced in the following way:

- if `color.shape[1] != nplex` and `color.shape[1] != 1`: take out first plane in direction 1
- if `color.shape[0] != nelems` and `color.shape[0] != 1`: repeat the plane in direction 0 nelems times

`opengl.sanitize.saneColorSet` (*color=None, colormap=None, shape=(1,)*)

Return a sane set of colors.

A sane set of colors is one that guarantees correct use by the draw functions. This means either - no color (None) - a single color - at least as many colors as the shape argument specifies - a color index and a color map with enough colors to satisfy the index. The return value is a tuple `color,colormap`. `colormap` will be None, unless `color` is an integer array, meaning a color index.

6.5.9 `opengl.scene` — This implements an OpenGL drawing widget for painting 3D scenes.

Classes defined in module `opengl.scene`

class `opengl.scene.ItemList` (*scene*)

A list of drawn objects of the same kind.

This is used to collect the Actors, Decorations and Annotations in a scene. Currently the implementation does not check that the objects are of the proper type or are not occurring multiple times.

add (*item*)

Add an item or a list thereof to a ItemList.

delete (*items*, *sticky=True*)

Remove item(s) from an ItemList.

Parameters:

- *items*: a single item or a list or tuple of items
- *sticky*: bool: if True, also sticky items are removed. The default is to not remove sticky items. Sticky items are items having an attribute `sticky=True`.

clear (*sticky=False*)

Clear the list.

Parameters:

- *sticky*: bool: if True, also sticky items are removed. The default is to not remove sticky items. Sticky items are items having an attribute `sticky=True`.

class `opengl.scene.Scene` (*canvas=None*)

An OpenGL scene.

The Scene is a class holding a collection of Actors, Decorations and Annotations. It can also have a background.

bbox

Return the bounding box of the scene.

The bounding box is normally computed automatically as the box enclosing all Actors in the scene. Decorations and Annotations are not included. The user can however set the `bbox` himself, in which case that value will be used. It can also be set to the special value `None` to force recomputing the `bbox` from all Actors.

set_bbox (*bb*)

Set the bounding box of the scene.

This can be used to set the scene bounding box to another value than the one autocomputed from all actors.

`bb` is a (2,3) shaped array specifying a bounding box. A special value `None` may be given to force recomputing the `bbox` from all Actors.

changeMode (*canvas*, *mode=None*)

This function is called when the rendering mode is changed

This method should be called to update the actors on a rendering mode change.

addAny (*actor*)

Add any actor type or a list thereof.

This will add any actor/annotation/decoration item or a list of any such items to the scene. This is the preferred method to add an item to the scene, because it makes sure that each item is added to the proper list.

removeAny (*actor*)

Remove a list of any actor/highlights/annotation/decoration items.

This will remove the items from any of the canvas lists in which the item appears. *itemlist* can also be a single item instead of a list. If *None* is specified, all items from all lists will be removed.

drawn (*obj*)

List the graphical representations of the given object.

Returns a list with the actors that point to the specified object.

removeDrawn (*obj*)

Remove all graphical representations of the given object.

Removes all actors returned by `drawn(obj)()`.

clear (*sticky=False*)

Clear the whole scene

removeHighlight (*actors=None*)

Remove the highlight from the actors in the list.

If no *actors* list is specified, all the highlights are removed.

highlighted ()

List highlighted actors

removeHighlighted ()

Remove the highlighted actors.

Functions defined in module `opengl.scene`

`opengl.scene.sane_bbox` (*bb*)

Return a sane nonzero *bb*.

bb should be a (2,3) float array or compatible. Returns a (2,3) float array where the values of the second row are guaranteed larger than the first. A value 1 is added in the directions where the input *bb* has zero size. Also, any NaNs will be transformed to numbers.

6.5.10 `opengl.shader` — OpenGL shader programs

Python OpenGL framework for pyFormex

This module is responsible for loading the shader programs and its data onto the graphics system.

(C) 2013 Benedict Verheghe and the pyFormex project.

Classes defined in module `opengl.shader`

class `opengl.shader.Shader` (*vshader=None, fshader=None, attributes=None, uniforms=None*)

An OpenGL shader consisting of a vertex and a fragment shader pair.

Class attributes:

- `_vertexshader` : the vertex shader source. By default, a basic shader supporting vertex positions and vertex colors is defined

- *_fragmentshader* : the fragment shader source. By default, a basic shader supporting fragment colors is defined.
- *attributes*: the shaders' vertex attributes.
- *uniforms*: the shaders' uniforms.

locations (*func, keys*)

Create a dict with the locations of the specified keys

uniformInt (*name, value*)

Load a uniform integer or boolean into the shader

uniformFloat (*name, value*)

Load a uniform float into the shader

uniformVec3 (*name, value*)

Load a uniform vec3[n] into the shader

The value should be a 1D array or list.

uniformMat4 (*name, value*)

Load a uniform mat4 into the shader

uniformMat3 (*name, value*)

Load a uniform mat3 into the shader

bind (*picking=False*)

Bind the shader program

unbind ()

Unbind the shader program

loadUniforms (*D*)

Load the uniform attributes defined in D

D is a dict with uniform attributes to be loaded into the shader program. The attributes can be of type int, float, or vec3.

Functions defined in module `opengl.shader`

`opengl.shader.defaultShaders` ()

Determine the default shader programs

6.5.11 `opengl.texttext` — Text rendering on the OpenGL canvas.

This module uses textures on quads to render text on an OpenGL canvas. It is dependent on freetype and the Python bindings freetype-py.

Classes defined in module `opengl.texttext`

class `opengl.texttext.FontTexture` (*filename, size, save=False*)

A Texture class for text rendering.

The `FontTexture` class is a texture containing the most important characters of a font. This texture can then be used to draw text on geometry. In the current implementation only the characters with ASCII ordinal in the range 32..127 are put in the texture.

Parameters

- **filename** (*str* or *Path*) – The name of the font file to be used. It should be the full path of an existing monospace font on the system.
- **size** (*float*) – Intended font height. The actual height might differ a bit.
- **save** (*bool*) – If True and *filename* is a font file (.ttf), the generated texture image will be saved as a .png image for later reload.

generateFromFont (*filename, size, save=False*)

Initialize a FontTexture

activate (*mode=None*)

Bind the texture and make it ready for use.

Returns the texture id.

texCoords (*char*)

Return the texture coordinates for a character or string.

Parameters:

- *char*: integer or ascii string. If an integer, it should be in the range 32..127 (printable ASCII characters). If a string, all its characters should be ASCII printable characters (have an ordinal value in the range 32..127).

If *char* is an integer, returns a tuple with the texture coordinates in the FontTexture corresponding with the specified character. This is a sequence of four (x,y) pairs corresponding respectively with the lower left, lower right, upper right, upper left corners of the character in the texture. Note that values for the lower corners are higher than those for the upper corners. This is because the FontTextures are (currently) stored from top to bottom, while opengl coordinates are from bottom to top.

If *char* is a string of length *ntext*, returns a float array with shape (ntext,4,2) holding the texture coordinates needed to display the given text on a grid of quad4 elements.

classmethod default (*size=24*)

Set and return the default FontTexture.

class `opengl.texttext.Text` (*text, pos, gravity=None, size=18, width=None, font=None, lineskip=1.0, grid=None, texmode=4, **kargs*)

A text drawn at a 2D or 3D position.

Parameters:

- *text*: string: the text to display. If not a string, the string representation of the object will be drawn. Newlines in the string are supported. After a newline, the remainder of the string is continued from a lower vertical position and the initial horizontal position. The vertical line offset is determined from the font.
- *pos*: a 2D or 3D position. If 2D, the values are measured in pixels. If 3D, it is a point in global 3D space. The text is drawn in 2D, inserted at the specified position.
- *gravity*: a string that determines the adjusting of the text with respect to the insert position. It can be a combination of one of the characters 'N' or 'S' to specify the vertical position, and 'W' or 'E' for the horizontal. The default(empty) string will center the text.
- *size*: float: size (height) of the font. This is the displayed height. The used font can have a different height and is scaled accordingly.
- *width*: float: width of the font. This is the displayed width of a single character (currently only monospace fonts are supported). The default is set from the *size* and the aspect ratio of the font. Setting this to a different value allows the creation of condensed and expanded font types. Condensed fonts are often used to save space.

- *font*: *FontTexture* or string. The font to be used. If a string, it is the filename of an existing monospace font on the system.
- *lineskip*: float: distance in pixels between subsequent baselines in case of multi-line text. Multi-line text results when the input *text* contains newlines.
- *grid*: raster geometry for the text. This is the geometry where the generate text will be rendered on as a texture. The default is a grid of rectangles of size (*width*, 'size') which are juxtaposed horizontally. Each rectangle will be rendered with a single character on it.

class `opengl.texttext.TextArray` (*val, pos, prefix=""*, ***kargs*)

An array of texts drawn at a 2D or 3D positions.

The text is drawn in 2D, inserted at the specified (2D or 3D) position, with alignment specified by the gravity (see class *Text*).

Parameters:

- *text*: a list of N strings: the texts to display. If an item is not a string, the string representation of the object will be drawn.
- *pos*: either an [N,2] or [N,3] shaped array of 2D or 3D positions. If 2D, the values are measured in pixels. If 3D, it is a point in global 3D space.
- *prefix*: string. If specified, it is prepended to all drawn strings.

Other parameters can be passed to the *Text* class.

class `opengl.texttext.Mark` (*pos, tex, size, opak=False, ontop=True*, ***kargs*)

A 2D drawing inserted at a 3D position of the scene.

The minimum attributes and methods are:

- *pos* : 3D point where the mark will be drawn

6.5.12 `opengl.texture` — Texture rendering.

This module defines tools for texture rendering in pyFormex.

class `opengl.texture.Texture` (*image, mode=1, format=GL_RGBA, texformat=GL_RGBA*)

An OpenGL 2D Texture.

Parameters

- **image** (*array_like* or *path_like*) – Image data: either raw image data (unsigned byte RGBA data) or the name of an image file with such data.
- **format** – Format of the image data.
- **texformat** – Format of the texture data.

activate (*mode=None, fltr=0*)

Render-time texture environment setup

6.6 pyFormex plugin menus

Plugin menus are optionally loadable menus for the pyFormex GUI, providing specialized interactive functionality. Because these are still under heavy development, they are currently not documented. Look to the source code or try them out in the GUI. They can be loaded from the File menu option and switched on permanently from the Settings menu.

Currently available:

- geometry_menu
- formex_menu
- surface_menu
- tools_menu
- draw2d_menu
- nurbs_menu
- dxf_tools
- jobs menu
- postproc_menu
- bifmesh_menu

6.7 pyFormex tools

The main pyformex path contains a number of modules that are not considered to be part of the pyFormex core, but are rather tools that were used in the implementation of other modules, but can also be useful elsewhere.

6.7.1 mydict — Extensions to Python’s built-in dict class.

Dict is a dictionary with default values and alternate attribute syntax. CDict is a Dict with lookup cascading into the next level Dict’s if the key is not found in the CDict itself.

The classes have been largely reimplemented in 2019, because many of the reasons for choosing the old implementation are not valid anymore because of improved features of Python. And the migration to Python3 showed the shortcomings of an implementation that was too hackishly meddling with Python internals. Now we have a much cleaner and simpler implementation while still preserving the intended functionality.

(C) 2005,2008,2019 Benedict Verhegghe Distributed under the GNU GPL version 3 or later

Classes defined in module mydict

class mydict.Dict (*default_factory=None, *args, **kwargs*)
A dictionary with default lookup and attribute and call syntax.

Dict is a dictionary class providing extended functionality over the builtin Python `dict` class:

- Items can not only be accessed with dictionary key syntax, but also with attribute and call syntax. Thus, if *C* is a *Dict*, an item with key ‘foo’ can be obtained in any of the following equivalent `C['foo']`, `C.foo`, `C('foo')`. The attribute syntax can of course only be used if the key is a string that is valid as Python variable. The first two can also be used to set the value in the Dict: `C['foo'] = bar`, `C.foo = bar`. The call syntax does not allow setting a value. See however the `Attributes` subclass for a dict that does allow `C(foo=bar)` to set values.
- A `default_factory` function can be provided to handle missing keys. The function is then passed the missing key and whatever it returns is returned as the value for that key. Note that this is different from Python’s `defaultdict`, where the `default_factory` does not take an argument and can only return a constant value. Without `default_factory`, the Dict will still produce a `KeyError` on missing keys.

- Because the Dict accepts call syntax to lookup keys, another Dict instance can perfectly well be used as default_factory. You can even create a chain of Dict's to lookup missing keys.
- An example default_factory is provided which returns None for any key. Using this default_factory will make the Dict accept any key and just return None for the nonexistent keys.

Parameters

- **default_factory** (*callable, optional*) – If provided, missing keys will be looked up by a call to the default_factory.
- **kargs** (*args,*) – Any other arguments are passed to the dict initialization.

Notes

Watch for this syntax quirks when initializing a Dict:

- Dict(): an empty Dict without default_factory
- Dict(callable): an empty Dict with callable as default_factory
- Dict(noncallable): a Dict without default_factory initialized with data from noncallable (e.g. a dict or a sequence of tuples)
- Dict(callable, noncallable): a Dict with callable as default_factory and initialized with data from noncallable.

And since a Dict is callable, but a dict is not, Dict(some_dict) will initialize the data in the Dict, while Dict(some_Dict) will use some_dict as defaults, but stay empty itself.

If a default_factory is provided, all lookup mechanisms (key, attribute, call) will use it for missing keys. The proper way to test whether a key actually exists in the Dict (and not in the default_factory) is using the 'key in Dict' operation.

Examples

```
>>> A = Dict()
>>> A
Dict({})
>>> A['a'] = 0
>>> A.b = 1
>>> # A(c=2) # not allowed
>>> A
Dict({'a': 0, 'b': 1})
>>> sorted(A.keys())
['a', 'b']
>>> 'c' in A
False
>>> A.default_factory is None # A does have a default_factory attribute
True
```

Now a Dict with a default_factory.

```
>>> B = Dict(Dict.returnNone, A) # The dict is initialized with the values of A
>>> B
Dict({'a': 0, 'b': 1})
>>> sorted(B.keys())
['a', 'b']
```

(continues on next page)

(continued from previous page)

```

>>> 'c' in B
False
>>> B['c'] is None # lookup nonexistent key returns None
True
>>> B.c is None
True
>>> B('c') is None
True
>>> B.get('c','Surprise?') # get() does not use default_factory!
'Surprise?'

```

Using another Dict as default_factory

```

>>> C = Dict(A,{'b':5, 'c':6}) # Now A is used for missing keys
>>> C
Dict({'b': 5, 'c': 6})
>>> sorted(C.keys())
['b', 'c']
>>> 'a' in C, 'b' in C, 'c' in C # 'a' is not in C
(False, True, True)
>>> C['a'], C['b'], C['c'] # but still returns a value
(0, 5, 6)
>>> C('a'), C('b'), C('c') # also with call syntax
(0, 5, 6)
>>> C.a, C.b, C.c # or as attributes
(0, 5, 6)
>>> hasattr(C,'a') # hasattr uses default_factory
True
>>> getattr(C,'a',None) # getattr too
0
>>> 'a' in C # the proper way to test if key exists in C
False
>>> C['b'], C.__missing__('b') # masked entries of A can also be accessed
(5, 1)

```

static returnNone (*args, **kargs)
Always returns None.

update (data={}, **kargs)
Add a dictionary to the Dict object.

The data can be a dict or Dict type object.

class mydict.CDict (default_factory=<function Dict.returnNone>, *args, **kargs)
A cascading Dict: missing keys are looked up in all dict values.

This is a *Dict* subclass with an extra lookup mechanism for missing keys: any value in the Dict that implements the Mapping mechanism (such as dict, Dict and their subclasses) will be tested for the missing key, and the first one found will be returned. If there is no Mapping value providing the key, and a default_factory exists, the key will finally be looked up in the default_factory as well.

Parameters

- **default_factory** (callable, optional) – If provided, missing keys will be looked up by a call to the default_factory.
- **kargs** (args,) – Any other arguments are passed to the dict initialization.

Notes

If there are multiple Mapping values in the CDict containing the missing key, it is undefined which one will be used to return a value. CDict's are therefore usually used where the Mapping values have themselves separate sets of keys.

If any of the values in the CDict is also a CDict, the lookup of missing keys will continue in the Mapping values of that CDict. This results in a cascading lookup as long as the Mapping types in the values as CDict (and not Dict or dict). This is another way of creating chained lookup mechanisms (besides using a Dict as default_factory).

Examples

```
>>> C = CDict(a=0, d={'a':1, 'b':2})
>>> C
CDict({'a': 0, 'd': {'a': 1, 'b': 2}})
>>> C['a']
0
>>> C['d']
{'a': 1, 'b': 2}
>>> C['b']           # looked up in C['d']
2
>>> 'b' in C
False
```

Now a CDict with a default_factory.

```
>>> C = CDict(CDict.returnNone,a=0, d={'a':1, 'b':2})
>>> C
CDict({'a': 0, 'd': {'a': 1, 'b': 2}})
>>> 'c' in C, C['c']      # No 'c', but returns None
(False, None)
```

Functions defined in module mydict

`mydict.formatDict(d, indent=4)`

Format a dict in nicely formatted Python source representation.

Each (key,value) pair is formatted on a line of the form:

```
key = value
```

If all the keys are strings containing only characters that are allowed in Python variable names, the resulting text is a legal Python script to define the items in the dict. It can be stored on a file and executed.

`mydict.returnNone(*args, **kargs)`

Always returns None.

6.7.2 attributes — Attributes

This module defines a general class for adding extra attributes to other objects without cluttering the name space.

`class attributes.Attributes(default_factory=<function Dict.returnNone>, *args, **kargs)`

A general class for holding attributes.

This class is a versatile *Mapping* class for objects that need a customizable set of attributes, while avoiding a wildly expanding name space.

The class derives from `Dict` and therefore has key lookup via normal dict key mechanism or via attribute syntax or via function call. It also provides a `default_factory` to lookup missing keys.

The difference with the `Dict` class are:

- The function call can also be used to populate or update the contents of the Mapping.
- By default, a `default_factory` is set returning `None` for any missing key.
- Giving an attribute the value `None` removes it from the Mapping.

Parameters

- **default_factory** (*callable, optional*) – If provided, missing keys will be looked up by a call to the `default_factory`.
- **kargs** (*args,*) – Any other arguments are passed to the dict initialization.

Notes

While setting a single item to `None` will remove the item from the Attributes, `None` values can be entered using the `update()` method.

The parameter order is different from previous implementation of this class. This was done for consistency with the `Dict` and `CDict` classes.

Examples

```
>>> A = Attributes()
>>> A
Dict({})
>>> A(color='red', alpha=0.7, ontop=True)
>>> A
Dict({'color': 'red', 'alpha': 0.7, 'ontop': True})
>>> A['alpha'] = 0.8
>>> A.color = 'blue'
>>> A.ontop = None      # remove 'ontop'
>>> A
Dict({'color': 'blue', 'alpha': 0.8})
>>> A = Attributes({'alpha': 0.8, 'color': 'blue'})
>>> A.ontop is None
True
```

Create another Attributes with A as default, override color:

```
>>> B = Attributes(default_factory=A, color='green')
>>> B
Dict({'color': 'green'})
>>> B.color, B.alpha      # alpha found in A
('green', 0.8)
>>> A.clear()
>>> A
Dict({})
>>> A['alpha'], A.alpha, A('alpha')      # all mechanisms still working
(None, None, None)
```

(continues on next page)

(continued from previous page)

```

>>> B['alpha'], B.alpha, B('alpha')
(None, None, None)
>>> B(color=None,alpha=1.0)           # remove and change in 1 operation
>>> B
Dict({'alpha': 1.0})
>>> B.update(color=None)             # update can be used to enter None values.
>>> B
Dict({'alpha': 1.0, 'color': None})
>>> B['alpha'] = None
>>> B
Dict({'color': None})

```

6.7.3 config — A general yet simple configuration class.

(C) 2005, 2019 Benedict Verhegghe

Distributed under the GNU GPL version 3 or later

Why I wrote this simple class because I wanted to use Python expressions in my configuration files. This is so much more fun than using .INI style config files. While there are some other Python config modules available on the web, I couldn't find one that suited my needs and my taste: either they are intended for more complex configuration needs than mine, or they do not work with the simple Python syntax I expected.

What Our Config class is just a normal Python dictionary which can hold anything. Fields can be accessed either as dictionary lookup (`config['foo']`) or as object attributes (`config.foo`). The class provides a function for reading the dictionary from a flat text (multiline string or file). I will always use the word 'file' hereafter, because that is what you usually will read the configuration from. Your configuration file can have named sections. Sections are stored as other Python dicts inside the top Config dictionary. The current version is limited to one level of sectioning.

Classes defined in module config

class `config.Config` (*data*={}, *default*=<function Dict.returnNone>)

A configuration class allowing Python expressions in the input.

The Config class is a subclass of the Dict mapping, which provides access to items by either dict-style key lookup `config['foo']`, attribute syntax `config.foo` or call syntax `config('foo')`. Furthermore, the Dict class allows a `default_factory` function to lookup in another Dict. This allows a chain of Config objects: `session_prefs -> user_prefs -> factory_defaults`.

The Config class is different from its parent Dict class in two ways:

- Keys should only be strings that are valid Python variable names, except that they can also contain a '/' character. The '/' splits up the key in two parts: the first part becomes a key in the Config, and its value is a dict where the values are stored with the second part of the key. This allows for creating sections in the configuration. Currently only one level of sectioning is allowed (keys can only have a single '/' character).
- A Config instance can be initialized or updated with a text in Python source style. This provides an easy way to store configurations in files with Python style. The Config class also provides a way to export its contents to such a Python source text: updated configurations can thus be written back to a configuration file. See Notes below for details.

Parameters

- **data** (*dict or multiline string, optional*) – Data to initialize the Config. If a dict, all keys should follow the rules for valid config keys formulated above. If a multiline string, it should be an executable Python source text, with the limitations and exceptions outlined in the Notes below.
- **default** (*Config object, optional*) – If provided, this object will be used as default lookup for missing keys.

Notes

The configuration object can be initialized from a dict or from a multiline string. Using a dict is obvious: one only has to obey the restriction that keys should be valid Python variable names.

The format of the multiline config text is described hereafter. This is also the format in which config files are written and can be loaded.

All config lines should have the format: `key = value`, where `key` is a string and `value` is a Python expression. The first '=' character on the line is the delimiter between key and value. Blanks around both the key and the value are stripped. The value is then evaluated as a Python expression and stored in a variable with name specified by the key. This variable is available for use in subsequent configuration lines. It is an error to use a variable before it is defined. The key,value pair is also stored in the Config dictionary, unless the key starts with an underscore ('_'): this provides for local variables.

Lines starting with '#' are comments and are ignored, as are empty and blank lines. Lines ending with '' are continued on the next line. A line starting with '[' starts a new section. A section is nothing more than a Python dictionary inside the Config dictionary. The section name is delimited by '[' and ']'. All subsequent lines will be stored in the section dictionary instead of the toplevel dictionary.

All other lines are executed as Python statements. This allows e.g. for importing modules.

Whole dictionaries can be inserted at once in the Config with the `update()` function.

All defined variables while reading config files remain available for use in the config file statements, even over multiple calls to the `read()` function. Variables inserted with `addSection()` will not be available as individual variables though, but can be accessed as `self['name']`.

As far as the resulting Config contents is concerned, the following are equivalent:

```
C.update({'key':'value'})
C.read("key='value'\n")
```

There is an important difference though: the second line will make a variable `key` (with value 'value') available in subsequent Config `read()` method calls.

Examples

```
>>> C = Config('''# A simple config example
...     aa = 'bb'
...     bb = aa
...     [cc]
...     aa = 'aa'      # yes ! comments are allowed
...     _n = 3        # local: will get stripped
...     rng = list(range(_n))
...     ''')
>>> C
Dict({'aa': 'bb', 'bb': 'bb', 'cc': Dict({'aa': 'aa', 'rng': [0, 1, 2]})})
>>> C['aa']
```

(continues on next page)

(continued from previous page)

```
'bb'
>>> C['cc']
Dict({'aa': 'aa', 'rng': [0, 1, 2]})
>>> C['cc/aa']
'aa'
```

Create a new Config with default lookup in C

```
>>> D = Config(default=C)
>>> D
Dict({})
>>> D['aa']          # Get from C
'bb'
>>> D['cc']          # Get from C
Dict({'aa': 'aa', 'rng': [0, 1, 2]})
>>> D['cc/aa']       # Get from C
'aa'
>>> D.get('cc/aa','zorro') # but get method does not cascade!
'zorro'
```

Setting values in D will store them in D while C remains unchanged.

```
>>> D['aa'] = 'wel'
>>> D['dd'] = 'hoe'
>>> D['cc/aa'] = 'ziedewel'
>>> D
Dict({'aa': 'wel', 'dd': 'hoe', 'cc': Dict({'aa': 'ziedewel'})})
>>> C
Dict({'aa': 'bb', 'bb': 'bb', 'cc': Dict({'aa': 'aa', 'rng': [0, 1, 2]})})
>>> D['cc/aa']
'ziedewel'
>>> D['cc']
Dict({'aa': 'ziedewel'})
>>> D['cc/rng']
[0, 1, 2]
>>> 'ee' in D
False
>>> 'cc/ee' in D
False
>>> D['cc/bb'] = 'ok'
>>> list(D.keys())
['aa', 'dd', 'cc', 'cc/aa', 'cc/bb']
>>> del D['aa']
>>> del D['cc/aa']
>>> list(D.keys())
['dd', 'cc', 'cc/bb']
>>> del D['cc']
>>> list(D.keys())
['dd']
```

update (*data*={}, *name*=None, *removeLocals*=False)

Add a dictionary to the Config object.

The data, if specified, should be a valid Python dict. If no name is specified, the data are added to the top dictionary and will become attributes. If a name is specified, the data are added to the named attribute, which should be a dictionary. If the name does not specify a dictionary, an empty one is created, deleting the existing attribute.

If a name is specified, but no data, the effect is to add a new empty dictionary (section) with that name.

If `removeLocals` is set, keys starting with `'_'` are removed from the data before updating the dictionary and not included in the config. This behaviour can be changed by setting `removeLocals` to `false`.

load (*filename*, *debug=False*)

Read a configuration from a file in Config format.

Parameters **filename** (*path_like*) – Path of a text file in Config format.

Returns *Config* – Returns the Config self, update with the settings read from the specified file.

read (*txt*, *debug=False*)

Read a configuration from a file or text

txt is a sequence of strings. Any type that allows a loop like `for line in txt:` to iterate over its text lines will do. This could be an open file, or a multiline text after splitting on `'\n'`.

The function will try to react intelligently if a string is passed as argument. If the string contains at least one `'\n'`, it will be interpreted as a multiline string and be splitted on `'\n'`. Else, the string will be considered and a file with that name will be opened. It is an error if the file does not exist or can not be opened.

The function returns self, so that you can write: `cfg = Config()`.

write (*filename*, *header='# Config written by pyFormex -*- PYTHON -*-\n\n'*, *trailer='\n# End of config\n'*)

Write the config to the given file

The configuration data will be written to the file with the given name in a text format that is both readable by humans and by the `Config.read()` method.

The header and trailer arguments are strings that will be added at the start and end of the outputfile. Make sure they are valid Python statements (or comments) and that they contain the needed line separators, if you want to be able to read it back.

keys (*descend=True*)

Return the keys in the config.

By default this descends one level of Dicts.

Functions defined in module config

`config.formatDict` (*d*)

Format a dict in Python source representation.

Each (key,value) pair is formatted on a line of the form:

```
key = value
```

If all the keys are strings containing only characters that are allowed in Python variable names, the resulting text is a legal Python script to define the items in the dict. It can be stored on a file and executed.

This format is the storage format of the Config class.

6.7.4 collection — Tools for handling collections of elements belonging to multiple parts.

This module defines the Collection class.

class `collection.Collection` (*object_type=None*)

A collection is a set of (int,int) tuples.

The first part of the tuple has a limited number of values and are used as the keys in a dict. The second part can have a lot of different values and is implemented as an integer array with unique values. This is e.g. used to identify a set of individual parts of one or more OpenGL actors.

Examples:

```
>>> a = Collection()
>>> a.add(range(7), 3)
>>> a.add(range(4))
>>> a.remove([2,4], 3)
>>> print(a)
-1 [0 1 2 3]; 3 [0 1 3 5 6];
>>> a.add([[2,0], [2,3], [-1,7], [3,88]])
>>> print(a)
-1 [0 1 2 3 7]; 2 [0 3]; 3 [ 0 1 3 5 6 88];
>>> a[2] = [1,2,3]
>>> print(a)
-1 [0 1 2 3 7]; 2 [1 2 3]; 3 [ 0 1 3 5 6 88];
>>> a[2] = []
>>> print(a)
-1 [0 1 2 3 7]; 3 [ 0 1 3 5 6 88];
>>> a.set([[2,0], [2,3], [-1,7], [3,88]])
>>> print(a)
-1 [7]; 2 [0 3]; 3 [88];
>>> print(a.keys())
[-1 2 3]
>>> print([k for k in a])
[-1, 2, 3]
>>> 2 in a
True
```

add (*data, key=-1*)

Add new data to the collection.

data can be a 2d array with (key,val) tuples or a 1-d array of values. In the latter case, the key has to be specified separately, or a default value will be used.

data can also be another Collection, if it has the same object type.

set (*data, key=-1*)

Set the collection to the specified data.

This is equivalent to clearing the corresponding keys before adding.

remove (*data, key=-1*)

Remove data from the collection.

get (*key, default=[]*)

Return item with given key or default.

keys ()

Return a sorted array with the keys

items ()

Return an iterator over (key,value) pairs.

6.7.5 `olist` — Some convenient shortcuts for common list operations.

While most of these functions look (and work) like set operations, their result differs from using Python builtin Sets in that they preserve the order of the items in the lists.

Classes defined in module `olist`

class `olist.List` (**args*)

A versatile list class.

This class extends the builtin list type with automatic calling of a method for all items in the list. Any method other than the ones defined here will return a new List with the method applied to each of the items, using the same arguments.

As an example, `List([a,b]).len()` will return `List([a.len(),b.len()])`

```
>>> L = List(['first', 'second'])
>>> L.upper()
['FIRST', 'SECOND']
>>> L.startswith('f')
[True, False]
```

Functions defined in module `olist`

`olist.lrange` (**args*)

Return a range as a list.

This is a convenience function for compatibility between Python2 and Python3. In Python2 the `range()` function returns a list, in Python3 it doesn't. Use `lrange` instead of `range` whenever you need the whole list of numbers (thus not in a for loop).

`olist.roll` (*a, n=1*)

Roll the elements of a list *n* positions forward (backward if *n* < 0)

```
>>> roll(lrange(5), 2)
[2, 3, 4, 0, 1]
```

`olist.union` (*a, b*)

Return a list with all items in *a* or in *b*, in the order of *a, b*.

```
>>> union(lrange(3), lrange(1, 4))
[0, 1, 2, 3]
```

`olist.difference` (*a, b*)

Return a list with all items in *a* but not in *b*, in the order of *a*.

```
>>> difference(lrange(3), lrange(1, 4))
[0]
```

`olist.symdifference` (*a, b*)

Return a list with all items in *a* or *b* but not in both.

```
>>> symdifference(lrange(3), lrange(1, 4))
[0, 3]
```

`olist.intersection(a, b)`

Return a list with all items in a and in b, in the order of a.

```
>>> intersection(lrange(3), lrange(1, 4))
[1, 2]
```

`olist.concatenate(a)`

Concatenate a list of lists.

```
>>> concatenate([lrange(3), lrange(1, 4)])
[0, 1, 2, 1, 2, 3]
```

`olist.flatten(a, recurse=False)`

Flatten a nested list.

By default, lists are flattened one level deep. If `recurse=True`, flattening recurses through all sublists.

```
>>> flatten([[3., 2], 6.5], [5], 6, 'hi')
[[3.0, 2], 6.5, 5, 6, 'hi']
>>> flatten([[3., 2], 6.5], [5], 6, 'hi', True)
[3.0, 2, 6.5, 5, 6, 'hi']
```

`olist.group(a, n)`

Group a list by sequences of maximum n items.

Parameters:

- *a*: list
- *n*: integer

Returns a list of lists. Each sublist has length n, except for the last one, which may be shorter.

Examples

```
>>> group([3.0, 2, 6.5, 5, 'hi'], 2)
[[3.0, 2], [6.5, 5], ['hi']]
```

`olist.select(a, b)`

Return a subset of items from a list.

Returns a list with the items of a for which the index is in b.

```
>>> select(range(2, 6), [1, 3])
[3, 5]
```

`olist.remove(a, b)`

Returns the complement of `select(a,b)`.

```
>>> remove(range(2, 6), [1, 3])
[2, 4]
```

`olist.toFront(l, i)`

Add or move i to the front of list l

l is a list. If i is in the list, it is moved to the front of the list. Else i is added at the front of the list.

This changes the list inplace and does not return a value.

```
>>> L = lrange(5)
>>> toFront(L, 3)
>>> L
[3, 0, 1, 2, 4]
```

6.7.6 path — Object oriented filesystem paths.

This module defines the Path class which is used throughout pyFormex for handling filesystem paths.

The Path class provides object oriented handling of filesystem paths. It has many similarities to the classes in the `pathlib` module of Python3. But there are some important differences and these are mostly inspired by the specialized requirements of pyFormex as opposed to the more general requirements of Python.

Our Path class derives from Python's `str`. This was almost a necessity, because pyFormex uses a lot of external programs with their own file naming rules. Very often we need string methods to build the proper file names. The constant switching between Path and str is a real hindrance in using the Python `pathlib` classes. Because our Path is a str, it can be used immediately in all places and all functions as before, which allowed for a gradual transition from using `os.path` and other modules to our Path.

Unlike `pathlib`, we do not discern between pure paths and concrete paths. The basic usage of a file path in pyFormex is to use the file. There is hardly a need for file paths besides concrete paths.

While pyFormex is currently almost exclusively used on Linux, the Path class makes it rather straightforward to port it to other OSes, as all the path related code is concentrated in a single module.

Despite the differences, there are also quite some similarities with the `pathlib` classes. Most of the methods are named likewise, so that even changing to the use of `pathlib` could be done inside this single module.

Our Path class has however a lot more methods available than those of `pathlib`. Since we are not bound to `pathlib`, we can as well move all path and file related functionality into this module and extend the Path class with any interesting functionality that has common use.

In order for this module to be of general use however, we have kept things that are strictly pyFormex oriented out of this module

Classes defined in module path

class `path.Path`

A filesystem path which also behaves like a str.

A Path instance represents a valid path to a file in the filesystem, existing or not. Path is thus a subclass of str that can only represent strings that are valid as file paths. The constructor will always normalize the path.

Parameters `args` (*path_like*, ...) – One or more path components that will be concatenated to form the new Path. Each component can be a str or a Path. It can be relative or absolute. If multiple absolute components are specified, the last one is used.

The following all create the same Path:

```
>>> Path('/pyformex/gui/menu')
Path('/pyformex/gui/menu')
>>> Path('/pyformex', 'gui', 'menu')
Path('/pyformex/gui/menu')
>>> Path('/pyformex', Path('gui'), 'menu')
Path('/pyformex/gui/menu')
```

But this is different:

```
>>> Path('/pyformex', '/gui', 'menus')
Path('/gui/menus')
```

Spurious slashes and single and double dots are collapsed:

```
>>> Path('/pyformex//gui//menus')
Path('/pyformex/gui/menus')
>>> Path('/pyformex./gui/menus')
Path('/pyformex/gui/menus')
>>> Path('/pyformex../gui/menus')
Path('/gui/menus')
```

Note: The collapsing of double dots is different from the `pathlib` behavior. Our `Path` class follows the `os.path.normpath()` behavior here.

Operators: The slash operator helps create child paths, similarly to `os.path.join()`. The plus operator can be used to add a trailing part without a slash separator. The equal operator allows comparing paths.

```
>>> p = Path('/etc') / 'init.d' / 'apache2'
>>> p
Path('/etc/init.d/apache2')
>>> p + '.d'
Path('/etc/init.d/apache2.d')
>>> p1 = Path('/etc') + '/init.d/apache2'
>>> p1 == p
True
```

Note: Unlike the `pathlib`, our `Path` class does not provide the possibility to join a `str` and a `Path` with a slash operator: the first component must be a `Path`.

Properties: The following properties give access to different components of the `Path`:

- `parts`: a tuple with the various parts of the `Path`,
- `parent`: the parent directory of the `Path`
- `parents`: a tuple with the subsequent parent `Paths`
- `name`: a string with the final component of the `Path`
- `suffix`: the file extension of the final component, if any
- `stem`: the final component without its suffix
- `lsuffix`: the suffix in lower case
- `ftype`: the suffix in lower case and without the leading dot

Note: We currently do not have the following properties available with `pathlib`: `drive`, `root`, `anchor`, `suffixes`

Examples

```

>>> Path('/a/b')
Path('/a/b')
>>> Path('a/b/c')
Path('a/b/c')
>>> Path('a/b/.c')
Path('a/b/c')
>>> Path('a/b/../c')
Path('a/c')
>>> Path('a/b/...c')
Path('a/b/...c')
>>> Path('//a/b')
Path('//a/b')
>>> Path('///a/b')
Path('/a/b')
>>> p = Path('/etc/init.d/')
>>> p.parts
('/', 'etc', 'init.d')
>>> p.parent
Path('/etc')
>>> p.parents
(Path('/etc'), Path('/'))
>>> p0 = Path('pyformex/gui/menus')
>>> p0.parts
('pyformex', 'gui', 'menus')
>>> p0.parents
(Path('pyformex/gui'), Path('pyformex'), Path('.'))
>>> Path('../pyformex').parents
(Path('..'), Path('.'))
>>> p.name
'init.d'
>>> p.stem
'init'
>>> p.suffix
'.d'
>>> p1 = Path('Aa.Bb')
>>> p1.suffix, p1.ksuffix, p1.ftype
('.Bb', '.bb', 'bb')
>>> p.exists()
True
>>> p.is_dir()
True
>>> p.is_file()
False
>>> p.is_symlink()
False
>>> p.is_absolute()
True
>>> Path('/var/run').is_symlink()
True

```

parts

Split the Path in its components.

Returns *tuple of str* – The various components of the Path

Examples

```
>>> Path('/a/b/c/d').parts
('/', 'a', 'b', 'c', 'd')
>>> Path('a/b//d').parts
('a', 'b', 'd')
>>> Path('a/b/./d').parts
('a', 'b', 'd')
>>> Path('a/b/../d').parts
('a', 'd')
>>> Path('a/b/.../d').parts
('a', 'b', '...', 'd')
```

parents

Return the parents of the Path.

Returns *tuple of Path* – The subsequent parent directories of the Path

parent

Return the parent directory.

Returns *Path* – The parent directory of the Path.

name

Return the final path component.

Returns *str* – The final component of the Path.

stem

Return the final path component without its suffix.

Returns *str* – The final component of the Path without its *suffix*.

Examples

```
>>> Path('aA.bB').stem
'aA'
```

suffix

Return the file extension of the Path component.

The file extension is the last substring of the final component starting at a dot that is neither the start nor the end of the component.

Returns *str* – The file extension of the Path, including the leading dot.

Examples

```
>>> Path('aA.bB').suffix
'.bB'
```

lsuffix

Return the file extension in lower case.

Returns *str* – The suffix of the Path, converted to lower case.

Examples

```
>>> Path('aA.bb').lsuffix
'.bb'
```

ftype

Return the file extension in lower case and with the leading dot.

Returns *str* – The lsuffix of the Path without the leading dot.

Examples

```
>>> Path('aA.bbB').ftype
'bb'
```

exists()

Return True if the Path exists

is_dir()

Return True if the Path exists and is a directory

is_file()

Return True if the Path exists and is a file

is_symlink()

Return True if the Path exists and is a symlink

is_badlink()

Return True if the Path exists and is a bad symlink

A bad symlink is a symlink that points to a non-existing file

is_absolute()

Return True if the Path is absolute.

The Path is absolute if it start with a '/'.

```
>>> Path('/a/b').is_absolute()
True
>>> Path('a/b').is_absolute()
False
```

with_name(name)

Return a new Path with the filename changed.

Parameters *name* (*str*) – Name to replace the last component of the Path

Returns *Path* – A Path where the last component has been changed to *name*.

Examples

```
>>> Path('data/testrun.inp').with_name('testimg.png')
Path('data/testimg.png')
```

with_suffix(suffix)

Return a new Path with the suffix changed.

Parameters `suffix` (*str*) – Suffix to replace the last component’s suffix. The replacement string does not need to start with a dot. See Examples.

Returns *Path* – A Path where the suffix of the last component has been changed to `suffix`.

Examples

```
>>> Path('data/testrun.inp').with_suffix('.cfg')
Path('data/testrun.cfg')
>>> Path('data/testrun.inp').with_suffix('_bis.inp')
Path('data/testrun_bis.inp')
```

`absolute()`

Return an absolute version of the path.

Returns *Path* – The absolute filesystem path of the Path. This also works if the Path does not exist. It does not resolve symlinks.

See also:

`resolve()` return an absolute path resolving any symlinks.

Examples

```
>>> Path('.').absolute() # doctest: +ELLIPSIS
Path('/home/.../pyformex')
>>> Path('something').absolute() # doctest: +ELLIPSIS
Path('/home/.../pyformex/something')
```

`resolve()`

Return absolute path resolving all symlinks.

Returns *Path* – The absolute filesystem path of the Path, resolving all symlinks. This also works if any of the Path components does not exist.

Examples

```
>>> Path('/var/run').resolve()
Path('/run')
>>> Path('something/inside').resolve() # doctest: +ELLIPSIS
Path('/home/.../pyformex/something/inside')
```

`expanduser()`

Expand the ~ and ~user in Path.

Returns *Path* – The Path with ~ and ~user expanded.

Examples

```
>>> Path('~').expanduser() # doctest: +ELLIPSIS
Path('/home/...')
>>> Path('~root').expanduser()
Path('/root')
```

as_uri()

Return the Path as an URI.

Returns *str* – A string starting with ‘file://’ followed by the resolved absolute path of the Path. Also ~ and ~user are expanded.

Examples

```
>>> Path('~user/some/file.html').as_uri()
'file:///home/user/some/file.html'
```

samefile(other_file)

Test whether two pathnames reference the same actual file

Parameters *other* (*path_like*) – Another file path to compare with.

Returns *bool* – True if the other file is actually the same file as self.

Examples

```
>>> Path.home().samefile(Path('~').expanduser())
True
```

commonprefix(*other)

Return the longest common leading part in a list of paths.

Parameters **other* (one or more *path_like*) – Other file path(s) to compare with.

Returns *Path* – The longest common leading part in all the file paths.

Examples

```
>>> p = Path('/etc/password')
>>> q = Path('/etc/profile')
>>> p.commonprefix(p, q, '/etc/pam.d')
Path('/etc/p')
>>> p.commonprefix(p, q, '/etc/group')
Path('/etc')
>>> p.commonpath(p, q, '/etc/pam.d')
Path('/etc')
```

commonpath(*other)

Return the longest common sub-path in a list of paths.

Parameters **other* (one or more *path_like*) – Other file path(s) to compare with.

Returns *Path* – The longest common sub-path in all the file paths.

Examples

```
>>> p = Path('/etc/password')
>>> p.commonpath(p, '/etc/pam.d')
Path('/etc')
```

joinpath (**other*)

Join two or more path components.

Parameters **other* (one or more *path_like*) – Other path components to join to self.

Notes

This alternative to using the slash operator is especially useful if the components are a computed and/or long sequence.

Examples

```
>>> home = Path.home()
>>> p1 = home.joinpath('.config', 'pyformex', 'pyformex.conf')
>>> p2 = home / '.config' / 'pyformex' / 'pyformex.conf'
>>> p1 == p2
True
```

relative_to (*other*)

Return a relative path version of a path.

Parameters *other* (*path_like*) – Another file path to compare with.

Returns *Path* – Path relative to *other* pointing to same file as self.

See also:

absolute() make a Path absolute

Examples

```
>>> p1 = Path('/usr/local/bin')
>>> p2 = p1.relative_to('/usr/bin')
>>> p2
Path('../local/bin')
>>> p2.absolute() # doctest: +ELLIPSIS
Path('/home/.../local/bin')
```

mkdir (*mode=509, parents=False, exist_ok=False*)

Create a directory.

Parameters

- **mode** (*int*) – The mode to be set on the created directory.
- **parents** (*bool*) – If True, nonexistent intermediate directories will also be created. The default requires all parent directories to exist.
- **exist_ok** (*bool*) – If True and the target already exist and is a directory, it will be silently accepted. The default (False) will raise an exception if the target exists.

rmdir ()

Remove an empty directory.

unlink ()

Remove an existing file.

remove ()

Remove a file, but silently ignores non-existing

removeTree (top=True)

Recursively delete a directory tree.

Parameters **top** (*bool*) – If True (default), the top level directory will be removed as well. If False, the top level directory will be kept, and only its contents will be removed.

move (dst)

Rename or move a file or directory

Parameters **dst** (*path_like*) – Destination path.

Returns *Path* – The new Path.

Notes

Changing a directory component will move the file. Moving a file across file system boundaries may not work. If the destination is an existing file, it will be overwritten.

copy (dst)

Copy the file under another name.

Parameters **dst** (*path_like*) – Destination path.

symlink (dst)

Create a symlink for this Path.

Parameters **dst** (*path_like*) – Path of the symlink, which will point to self if successfully created.

touch ()

Create an empty file or update an existing file's timestamp.

If the file does not exist, it is create as an empty file. If the file exists, it remains unchanged but its time of last modification is set to the current time.

truncate ()

Create an empty file or truncate an existing file.

If the file does not exist, it is create as an empty file. If the file exists, its contents is erased.

chmod (mode)

Change the access permissions of a file.

Parameters **mode** (*int*) – Permission mode to set on the file. This is usually given as an octal number reflecting the access mode bitfield. Typical values in a trusted environment are 0o664 for files and 0o775 for directories. If you want to deny all access for others, the values are 0o660 and 0o770 respectively.

stat ()

Return the full stat info for the file.

Returns *stat_result* – The full stat results for the file Path.

Notes

The return value can be interrogated using Python's stat module. Often used values can also be got from Path methods *mtime()*, *size()*, *owner()*, *group()*.

mtime()
Return the (UNIX) time of last change

size()
Return the file size in bytes

owner()
Return the login name of the file owner.

group()
Return the group name of the file gid.

open(mode='r', buffering=-1, encoding=None, errors=None)
Open the file pointed to by the Path.

Parameters are like in Python's built-in `python.open()` function.

read_bytes()
Open the file in bytes mode, read it, and close the file.

write_bytes(data)
Open the file in bytes mode, write to it, and close the file.

read_text(encoding=None, errors=None)
Open the file in text mode, read it, and close the file.

write_text(text, encoding=None, errors=None)
Open the file in text mode, write to it, and close the file.

Examples

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
>>> p.read_text()
'Text file contents'
```

walk()
Recursively walk through a directory.

This walks top-down through the directory, yielding tuples `root, dirs, files`, like `os.walk()`.

scandir()
Returns all entries in a directory

dirs()
List the subdirectories in a directory path.

Returns *list of str* – A list of the names of all directory type entries in the Path. If the Path is not a directory or not accessible, an exception is raised.

files()
List the files in a directory path.

Returns *list of str* – A list of the names of all file type entries in the Path. If the Path is not a directory or not accessible, an exception is raised.

symlinks()
Returns all symlinks in a directory.

glob (*recursive=False*)

Return a list of paths matching a pathname pattern.

For a Path including wildcards (`,` `?`, `[]`, `*`), finds all existing files (of any type, including directories) matching the Path. The `**` wildcard matches all files and any number of subdirectories.

Returns *list of Path* – A sorted list of existing files matching the pattern.

Note: This method works differently from `pathlib.Path.glob()`. The wildcards are part of the calling input Path and operation is like that of `glob.glob()`.

Parameters **recursive** (*bool*) – If True, operation is recursive and a `**` wildcard matches all files and zero or any subdirectories.

See also:

[`listTree\(\)`](#) find files matching regular expressions

Examples

```
>>> Path('/etc/init.d').glob()
[Path('/etc/init.d')]
>>> Path('pyformex/pr*.py').glob()
[Path('pyformex/process.py'), Path('pyformex/project.py')]
>>> Path('pyformex/**/pa*.py').glob()
[Path('pyformex/plugins/partition.py')]
>>> Path('pyformex/**/pa*.py').glob(recursive=True)
[Path('pyformex/path.py'), Path('pyformex/plugins/partition.py')]
>>> Path('**/q*.py').glob()
[]
>>> Path('**/q*.py').glob(recursive=True)
[Path('pyformex/gui/qtutils.py')]
```

listTree (*listdirs=False*, *topdown=True*, *sorted=False*, *excludedirs=[]*, *excludefiles=[]*, *includedirs=[]*, *includefiles=[]*, *symlinks=True*)

List recursively all files matching some regular expressions.

This scans a directory tree for all files matching specified regular expressions.

Parameters

- **listdirs** (*bool*) – If True, matching directories are listed as well. The default is to only list files.
- **topdown** (*bool*) – If True (default) the directories are scanned top down, and files are listed in that order.
- **sorted** (*bool*) – If True, directories on the same level and files within a directory, will be sorted. The default is to treat items on the same level in an undefined order.
- **excludedirs** (*re* or list of *re*'s) – Regular expression(s) for dirnames to exclude from the tree scan.
- **excludefiles** (*re*) – Regular expression(s) for filenames to exclude from the file list.
- **includedirs** (*re* or list of *re*'s, optional) – Regular expression(s) for dirnames to include in the tree scan.

- **includefiles** (*re*) – Regular expression(s) for filenames to include in the file list.
- **symlinks** (*bool*) – If True, symlinks are included in the listed results. If False, symlinks are removed.

Returns *list of str* – The list of all existing file names (and possibly directory names) under the Path that satisfy the provided patterns. An exception is raised if the Path is not an existing directory.

Notes

If neither exclude nor include patterns are provided, all subdirectories are scanned and all files are reported. If only exclude patterns are provided, all directories and files except those matching any of the exclude patterns. If only include patterns are provided, only those matching at least one of the patterns are included. If both exclude and include patterns are provided, items are only listed if they match at least one of the include patterns but none of the exclude patterns

The use of `excludedirs` and/or `includedirs` forces top down handling.

filetype (*compressed*=[`'.gz'`, `'.bz2'`])

Return a normalized file type based on the filename suffix.

The returned suffix is in most cases the part of the filename starting at the last dot. However, if the thus obtained suffix is one of the specified compressed types (default: `.gz` or `.bz2`) and the file contains another dot that is not at the start of the filename, the returned suffix starts at the penultimate dot. This allows for transparent handling of compressed files.

Parameters **compressed** (*list of str*) – List of suffixes that are considered compressed file types.

Returns *str* – The filetype. This is the file suffix converted to lower case and without the leading dot. If the suffix is included in `compressed`, the returned suffix also includes the preceding suffix part, if any.

See also:

[`fctype\(\)`](#) the file type without accounting for compressed types

Examples

```
>>> Path('pyformex').filetype()
''
>>> Path('pyformex.pgf').filetype()
'pgf'
>>> Path('pyformex.pgf.gz').filetype()
'pgf.gz'
>>> Path('pyformex.gz').filetype()
'gz'
>>> Path('abcd/pyformex.GZ').filetype()
'gz'
```

ftype_compr (*compressed*=[`'.gz'`, `'.bz2'`])

Return the file type and compression based on suffix.

Parameters **compressed** (*list of str*) – List of suffixes that are considered compressed file types.

Returns

- **ftype** (*str*) – File type based on the last suffix if it is not a compression type, or on the penultimate suffix if the file is compressed.
- **compr** (*str*) – Compression type. This is the last suffix if it is one of the compressed types, or an empty string otherwise.

Examples

```
>>> Path('pyformex').ftype_compr()
('', '')
>>> Path('pyformex.pgf').ftype_compr()
('pgf', '')
>>> Path('pyformex.pgf.gz').ftype_compr()
('pgf', 'gz')
>>> Path('pyformex.gz').ftype_compr()
('gz', '')
```

classmethod `cwd()`

Return the current working directory.

classmethod `home()`

Return the user's home directory.

Functions defined in module `path`

`path.matchAny` (*target*, **regexps*)

Check whether *target* matches any of the regular expressions.

Parameters

- **target** (*str*) – String to match with the regular expressions.
- ***regexp** (*sequence of regular expressions.*) – The regular expressions to match the target string.

Returns *bool* – True, if target matches at least one of the provided regular expressions. False if no matches.

Examples

```
>>> matchAny('test.jpg', '.*\.png', '.*\.jpg')
True
>>> matchAny('test.jpeg', '.*\.png', '.*\.jpg')
False
>>> matchAny('test.jpg')
False
```

6.7.7 `flatkeydb` — Flat Text File Database.

A simple database stored as a flat text file.

(C) 2005 Benedict Verhegge.

Distributed under the GNU GPL version 3 or later.

Classes defined in module flatkeydb

class flatkeydb.FlatDB(*req_keys=[]*, *comment='#'*, *key_sep='='*, *beginrec='beginrec'*, *endrec='endrec'*, *strip_blanks=True*, *strip_quotes=True*, *check_func=None*)

A database stored as a dictionary of dictionaries.

Each record is a dictionary where keys and values are just strings. The field names (keys) can be different for each record, but there is at least one field that exists for all records and will be used as the primary key. This field should have unique values for all records.

The database itself is also a dictionary, with the value of the primary key as key and the full record as value.

On constructing the database a list of keys must be specified that will be required for each record. The first key in this list will be used as the primary key. Obviously, the list must at least have one required key.

The database is stored in a flat text file. Each field (key,value pair) is put on a line by itself. Records are delimited by a (beginrec, endrec) pair. The beginrec marker can be followed by a (key,value) pair on the same line. The endrec marker should be on a line by itself. If endrec is an empty string, each occurrence of beginrec will implicitly end the previous record.

Lines starting with the comment string are ignored. They can occur anywhere between or inside records. Blank lines are also ignored (except they serve as record delimiter if endrec is empty)

Thus, with the initialization:

```
FlatDB(req_keys=['key1'], comment = 'com', key_sep = '=',
beginrec = 'rec', endrec = '')
```

the following is a legal database:

```
com This is a comment
com
rec key1=val1
    key2=val2
rec
com Yes, this starts another record
    key1=val3
    key3=val4
```

The *readFile* function can even be instructed to ignore anything not between a (beginrec,endrec) pair. This allows for multiple databases being stored on the same file, even with records intermixed.

Keys and values can be any strings, except that a key can not begin nor end with a blank, and can not be equal to any of the comment, beginrec or endrec markers. Whitespace around the key is always stripped. By default, this is also done for the value (though this can be switched off.) If *strip_quotes* is True (default), a single pair of matching quotes surrounding the value will be stripped off. Whitespace is stripped before stripping the quotes, so that by including the value in quotes, you can keep leading and trailing whitespace in the value.

A record checking function can be specified. It takes a record as its argument. It is called whenever a new record is inserted in the database (or an existing one is replaced). Before calling this *check_func*, the system will already have checked that the record is a dictionary and that it has all the required keys.

Two error handlers may be overridden by the user:

- *record_error_handler(record)* is called when the record does not pass the checks;
- *key_error_handler(key)* is called when a duplicat key is encountered.

The default for both is to raise an error. Overriding is done by changing the instance attribute.

newRecord ()

Returns a new (empty) record.

The new record is a temporary storage. It should be added to the database by calling `append(record)`. This method can be overridden in subclasses.

checkKeys (*record*)

Check that record has the required keys.

checkRecord (*record*)

Check a record.

This function checks that the record is a dictionary type, that the record has the required keys, and that `check_func(record)` returns True (if a *check_func* was specified). If the record passes, just return True. If it does not, call the *record_error_handler* and (if it returns) return False. This method can safely be overridden in subclasses.

record_error_handler (*record*)

Error handler called when a check error on record is discovered.

Default is to raise a runtime error. This method can safely be overridden in subclasses.

key_error_handler (*key*)

Error handler called when a duplicate key is found.

Default is to raise a runtime error. This method can safely be overridden in subclasses.

insert (*record*)

Insert a record to the database, overwriting existing records.

This is equivalent to `__setitem__` but using the value stored in the the primary key field of the record as key for storing the record. This is also similar to `append()`, but overwriting an old record with the same primary key.

append (*record*)

Add a record to the database.

Since the database is a dictionary, keys are unique and appending a record with an existing key is not allowed. If you want to overwrite the old record, use `insert()` instead.

splitKeyValue (*line*)

Split a line in key,value pair.

The field is split on the first occurrence of the *key_sep*. Key and value are then stripped of leading and trailing whitespace. If there is no *key_sep*, the whole line becomes the key and the value is an empty string. If the *key_sep* is the first character, the key becomes an empty string.

parseLine (*line*)

Parse a line of the flat database file.

A line starting with the comment string is ignored. Leading whitespace on the remaining lines is ignored. Empty (blank) lines are ignored, unless the ENDREC mark was set to an empty string, in which case they count as an end of record if a record was started. Lines starting with a 'BEGINREC' mark start a new record. The remainder of the line is then reparsed. Lines starting with an 'ENDREC' mark close and store the record. All lines between the BEGINREC and ENDREC should be field definition lines of the type 'KEY [= VALUE]'. This function returns 0 if the line was parsed correctly. Else, the variable `self.error_msg` is set.

parse (*lines*, *ignore=False*, *filename=None*)

Read a database from text.

lines is an iterater over text lines (e.g. a text file or a multiline string splitted on 'n') Lines starting with a comment string are ignored. Every record is delimited by a (beginrec,endrec) pair. If *ignore* is True, all lines that are not between a (beginrec,endrec) pair are simply ignored. Default is to raise a RuntimeError.

readFile (*filename, ignore=False*)

Read a database from file.

Lines starting with a comment string are ignored. Every record is delimited by a (beginrec,endrec) pair. If ignore is True, all lines that are not between a (beginrec,endrec) pair are simply ignored. Default is to raise a RuntimeError.

writeFile (*filename, mode='w', header=None*)

Write the database to a text file.

Default mode is 'w'. Use 'a' to append to the file. The header is written at the start of the database. Make sure to start each line with a comment marker if you want to read it back!

match (*key, value*)

Return a list of records matching key=value.

This returns a list of primary keys of the matching records.

Functions defined in module flatkeydb

`flatkeydb.firstWord` (*s*)

Return the first word of a string.

Words are delimited by blanks. If the string does not contain a blank, the whole string is returned.

`flatkeydb.unQuote` (*s*)

Remove one level of quotes from a string.

If the string starts with a quote character (either single or double) and ends with the SAME character, they are stripped of the string.

`flatkeydb.splitKeyValue` (*s, key_sep*)

Split a string in a (key,value) on occurrence of key_sep.

The string is split on the first occurrence of the substring key_sep. Key and value are then stripped of leading and trailing whitespace. If there is no key_sep, the whole string becomes the key and the value is an empty string. If the string starts with key_sep, the key becomes an empty string.

`flatkeydb.ignore_error` (*dummy*)

This function can be used to override the default error handlers.

The effect will be to ignore the error (duplicate key, invalid record) and to not add the affected data to the database.

6.7.8 timer — A timer class.

class `timer.Timer` (*start=None, tag=None*)

A class for measuring elapsed time.

A Timer object measures elapsed real time since a specified time, which by default is the time of the creation of the Timer. A builtin tag can be used for identifying subsequent measurements.

Parameters:

- *start*: a datetime object. If not specified, the time of the creation of the Timer is used.
- *tag*: a string used as an identifier for the current measurement. It is displayed when printing the Timer value

Example:

```

>>> import time
>>> t = Timer()
>>> time.sleep(1.234)
>>> r = t.read()
>>> print(r.days, r.seconds, r.microseconds) # doctest: +ELLIPSIS
0 1 23...
>>> t.seconds() # doctest: +ELLIPSIS
1.23...
>>> t.seconds(rounded=True)
1

```

Note that the precise result of microseconds will be slightly larger than the input sleep time. The precise result is unknown, therefore ellipses are shown.

```

>>> tim = Timer(tag="First")
>>> time.sleep(0.13)
>>> print(tim)
Timer: First: 0.13 sec.
>>> tim.reset(tag="Next")
>>> time.sleep(0.13)
>>> print(tim)
Timer: Next: 0.13 sec.

```

reset (*start=None, tag=None*)

(Re)Start the timer.

Sets the start time of the timer to the specified value, or to the current time if not specified. Sets the tag to the specified value or to 'None'.

Parameters:

- *start*: a datetime object. If not specified, the current time as returned by `datetime.now()` is used.
- *tag*: a string used as an identifier for the current measurement

read (*reset=False*)

Read the timer.

Returns the elapsed time since the last reset (or the creation of the timer) as a `datetime.timedelta` object.

If `reset=True`, the timer is reset to the time of reading.

seconds (*reset=False, rounded=False*)

Return the timer readings in seconds.

The default return value is a rounded integer number of seconds. With `rounded == False`, a floating point value with granularity of 1 microsecond is returned.

If `reset=True`, the timer is reset at the time of reading.

report (*format='%.2f sec.'*)

Report the elapsed time in seconds since last reset

6.7.9 track — track.py

Functions for creating classes with access tracking facilities. This can e.g. be use to detect if the contents of a list or dict has been changed.

Variables defined in module track

`track.track_methods = ['__setitem__', '__setslice__', '__delitem__', 'update', 'append', '...`
List of names of methods that can possibly change an object of type dict or list.

Classes defined in module track

class `track.TrackedDict`

Tracked dict class

clear () → None. Remove all items from D.

copy () → a shallow copy of D

fromkeys ()

Create a new dictionary with keys from iterable and values set to value.

get ()

Return the value for key if key is in the dictionary, else default.

items () → a set-like object providing a view on D's items

keys () → a set-like object providing a view on D's keys

pop (**kw)

Wrapper function for a class method.

popitem (**kw)

Wrapper function for a class method.

setdefault (**kw)

Wrapper function for a class method.

update (**kw)

Wrapper function for a class method.

values () → an object providing a view on D's values

class `track.TrackedList`

Tracked list class

append (**kw)

Wrapper function for a class method.

clear ()

Remove all items from list.

copy ()

Return a shallow copy of the list.

count ()

Return number of occurrences of value.

extend (**kw)

Wrapper function for a class method.

index ()

Return first index of value.

Raises ValueError if the value is not present.

insert (**kw)

Wrapper function for a class method.

pop (**kw)
Wrapper function for a class method.

remove (**kw)
Wrapper function for a class method.

reverse ()
Reverse *IN PLACE*.

sort ()
Stable sort *IN PLACE*.

Functions defined in module track

`track.track_decorator` (*func*)
Create a wrapper function for tracked class methods.

The wrapper function increases the ‘hits’ attribute of the class and then executes the wrapped method. Note that the class is passed as the first argument.

`track.track_class_factory` (*cls*, *name*=”, *methods*=[‘__setitem__’, ‘__setslice__’, ‘__delitem__’, ‘update’, ‘append’, ‘extend’, ‘add’, ‘insert’, ‘pop’, ‘popitem’, ‘remove’, ‘setdefault’, ‘__iadd__’])
Create a wrapper class with method tracking facilities.

Tracking a class counts the number of times any of the specified class methods has been used. Given an input class, this will return a class with tracking facilities. The tracking occurs on all the specified methods. The default will

Parameters

- **cls** (*class*) – The class to be tracked.
- **name** (*str*, *optional*) – The name of the class wrapper. If not provided, the name will be ‘Tracked’ followed by the capitalized class name.
- **methods** (*list of str*) – List of class method names that should be taken into account in the tracking. Calls to any of these methods will increment the number of hits. The methods should be owned by the class itself, not by a parent class. The default list will track all methods which could possibly change a ‘list’ or a ‘dict’.

Returns *class* – A class wrapping the input class and tracking access to any of the specified methods. The class has an extra attribute ‘hits’ counting the number accesses to one of these methods. This value can be reset to zero to track changes after some breakpoint.

Examples

```
>>> TrackedDict = track_class_factory(dict)
>>> D = TrackedDict({'a':1, 'b':2})
>>> print(D.hits)
0
>>> D['c'] = 3
>>> print(D.hits)
1
>>> D.hits = 0
>>> print(D.hits)
0
>>> D.update({'d':1, 'b':3})
```

(continues on next page)

(continued from previous page)

```
>>> del D['a']
>>> print(D.hits)
2
```


PYFORMEX FAQ 'N TRICKS

Date Jun 17, 2019

Version 1.0.7

Author Benedict Verheghe <benedict.verheghe@ugent.be>

Abstract

This chapter answers some frequently asked questions about pyFormex and present some nice tips to solve common problems. If you have some question that you want answered, or want to present a original solution to some problem, feel free to communicate it to us (by preference via the pyFormex [Support tracker](#)) and we'll probably include it in the next version of this FAQ.

7.1 FAQ

1. How was the pyFormex logo created?

We used the GNU Image Manipulation Program ([GIMP](#)). It has a wide variety of scripts to create logos. With newer versions (≥ 2.6) use the menu *Fille*→*Create*→*Logos*→*Alien-neon*. With older versions (≤ 2.4) use *Xtra*→*Script-Fu*→*Logos*→*Alien-neon*.

In the Alien Neon dialog specify the following data:

```
Text: pyFormex
Font Size: 150
Font: Blippo-Heavy
Glow Color: 0xFF3366
Background Color: 0x000000
Width of Bands: 2
Width of Gaps: 2
Number of Bands: 7
Fade Away: Yes
```

Press *OK* to create the logo. Then switch off the background layer and save the image in PNG format. Export the image with *Save Background Color* option switched off!

2. How was the pyFormex favicon created?

With FTGL, save as icon, handedited .xpm in emacs to set background color to None (transparent), then converted to .png and .ico with convert.

3. Why is pyFormex written in Python?

Because

- it is very easy to learn (See the [Python](#) website)
- it is extremely powerful (More on [Python](#) website)

Being a scripting language without the need for variable declaration, it allows for quick program development. On the other hand, Python provides numerous interfaces with established compiled libraries, so it can be surprisingly fast.

4. Is an interpreted language like Python fast enough with large data models?

See the *question above*.

Note: We should add something about NumPy and the pyFormex C-library.

7.2 TRICKS

1. Use your script path as the current working directory

Start your script with the following:

```
chdir(__file__)
```

When executing a script, pyFormex sets the name of the script file in a variable `__file__` passed with the global variables to the execution environment of the script.

2. Import modules from your own script directories

In order for Python to find the modules in non-standard locations, you should add the directory path of the module to the `sys.path` variable.

A common example is a script that wants to import modules from the same directory where it is located. In that case you can just add the following two lines to the start of your script:

```
import os, sys
sys.path.insert(0, os.dirname(__file__))
```

3. Automatically load plugin menus on startup

Plugin menus can be loaded automatically on pyFormex startup, by adding a line to the `[gui]` section of your configuration file (`~/ .pyformexrc`):

```
[gui]
plugins = ['surface_menu', 'formex_menu']
```

4. Automatically execute your own scripts on startup

If you create your own pugin menus for pyFormex, you cannot autoload them like the regular plugin menus from the distribution, because they are not in the plugin directory of the installation. Do not be tempted to put your own files under the installation directory (even if you can acquire the permissions to do so), because on removal or reinstall your files might be deleted! You can however automatically execute your own scripts by adding their full path names in the `autorun` variable of your configuration file

```
autorun = '/home/user/myscripts/startup/'
```

This script will then be run when the pyFormex GUI starts up. You can even specify a list of scripts, which will be executed in order. The autorun scripts are executed as any other pyFormex script, before any scripts specified on the command line, and before giving the input focus to the user.

5. Multiple viewports with unequal size

The multiple viewports are ordered in a grid layout, and you can specify relative sizes for the different columns and/or rows of viewports. You can use `setColumnStretch` and `setRowStretch` to give the columns a relative stretch compared to the other ones. The following example produces 4 viewports in a 2x2 layout, with the right column(1) having double width of the left one(0), while the bottom row has a height equal to 1.5 times the height of the top row

```
layout (4)
pf.GUI.viewports.setColumnStretch(0,1)
pf.GUI.viewports.setColumnStretch(1,2)
pf.GUI.viewports.setRowStretch(0,2)
pf.GUI.viewports.setRowStretch(1,3)
```

6. Activate pyFormex debug messages from your script

```
import pyformex
pyformex.options.debug = True
```

7. Get a list of all available image formats

```
import gui.image
print image.imageFormats()
```

8. Create a movie from a sequence of recorded images

The `multisave` option allows you to easily record a series of images while working with pyFormex. You may want to turn this sequence into a movie afterwards. This can be done with the `mencoder` and/or `ffmpeg` programs. The internet provides comprehensive information on how to use these video encoders.

If you are looking for a quick answer, however, here are some of the commands we have often used to create movies.

- Create MNPG movies from PNG To keep the quality of the PNG images in your movie, you should not encode them into a compressed format like MPEG. You can use the MPNG codec instead. Beware though that uncompressed encodings may lead to huge video files. Also, the MNPG is (though freely available), not installed by default on Windows machines.

Suppose you have images in files `image-000.png`, `image-001.png`, First, you should get the size of the images (they all should have the same size). The command

```
file image*.png
```

will tell you the size. Then create movie with the command

```
mencoder mf://image-*.png -mf w=796:h=516:fps=5:type=png -ovc copy -oac copy -
↳o movie1.avi
```

Fill in the correct width(w) and height(h) of the images, and set the frame rate(fps). The result will be a movie `movie1.avi`.

- Create a movie from (compressed) JPEG images. Because the compressed format saves a lot of space, this will be the preferred format if you have lots of image files. The quality of the compressed image movie will suffer somewhat, though.

```
ffmpeg -r 5 -b 800 -i image-%03d.jpg movie.mp4
```

9. Install the gl2ps extension

Note: This belongs in *Installing pyFormex*

Saving images in EPS format is done through the gl2ps library, which can be accessed from Python using wrapper functions. Recent versions of pyFormex come with an installation script that will also generate the required Python interface module.

Warning: The older `python-gl2ps-1.1.2.tar.gz` available from the web is no longer supported

You need to have the OpenGL header files installed in order to do this (on Debian: `apt-get install libgl1-mesa-dev`).

10. Permission denied error when running calpy simulation

If you have no write permission in your current working directory, running a calpy simulation will result in an error like this:

```
fil = file(self.tempfilename, 'w')
IOError
:
[Errno 13] Permission denied: 'calpy.tmp.part-0'
```

You can fix this by changing your current working directory to a path where you have write permission (e.g. your home directory). You can do this using the *File->Change workdir* menu option. The setting will be saved when you leave pyFormex (but other scripts might change the setting again).

11. Reading back old Project (.pyf) files

When the implementation of some pyFormex class changes, or when the location of a module is changed, an error may result when trying to read back old Project (.pyf) files. While in principle it is possible to create the necessary interfaces to read back the old data and transform them to new ones, our current policy is to not do this by default for all classes and all changes. That would just require too much resources for maybe a few or no cases occurring. We do provide here some guidelines to help you with solving the problems yourself. And if you are not able to fix it, just file a support request at our [Support tracker](#) and we will try to help you.

If the problem is with a changed implementation of a class, it can usually be fixed by adding an appropriate `__set_state__` method to the class. Currently we have this for Formex and Mesh classes. Look at the code in `formex.py` and `mesh.py` respectively.

If the problem comes from a relocation of a module (e.g. the mesh module was moved from plugins to the pyFormex core), you may get an error like this:

```
AttributeError: 'NoneType' object has no attribute 'Mesh'
```

The reason is that the path recorded in the Project file pointed to the old location of the mesh module under `plugins` while the mesh module is now in the top `pyformex` directory. This can be fixed in two ways:

- The easy (but discouraged) way is to add a symbolic link in the old position, linking to the new one. We do not encourage to use this method, because it sustains the dependency on legacy versions.
- The recommended way is to convert your Project file to point to the new path. To take care of the above relocation of the mesh module, you could e.g. use the following command to convert your `old.pyf` to a `new.pyf` that can be properly read. It just replaces the old module path (`plugins.mesh`) with the current path (`mesh`):

```
sed 's|plugins.mesh|mesh|'g old.pyf >new.pyf
```

12. Call a user function from a dialog widget

The pyFormex Dialog class provides a reduced interface to the Qt widget system. This helps users in creating quite complex interactive dialogs with a minimum effort. Sometimes however the user wants to change something in the working of some part of the Dialog. Since the full Qt system is accessible to pyFormex, it is quite easy to do such customizations.

As an example, suppose the user wants to customize the ‘fslider’ input item. The default allows to bind a user function to the slider, and to specify whether this function will be called during slider movement (tracking is True) or only at the end of the movement (tracking is False). Now suppose the user wants to bind different functions during slider movement (tracking True) and at the end of movement (when the slider is released). He could then use the provided function binding for the tracking function, and bind another function to the slider widget’s ‘sliderReleased’ signal. The outline to do such a binding would be like the following:

```
def myfunc(*args,**kwargs):
    """My function invoked on release of the slider"
    ...

dialog = Dialog(items=[ _I('fieldname', itemtype='fslider', ...) ])
item = dialog['fieldname']
item.slider.sliderReleased.connect(myfunc)
```

First, the function to be called is defined. The dialog is constructed and contains an ‘fslider’ type input item. The InputItem instance can be found from the Dialog by indexing with the field name. This instance will actually be an InputFSlider subclass of InputItem, and have an attribute ‘slider’ which is the QSlider widget. The last line then connects the ‘sliderReleased’ signal of that widget to the user function.

PYFORMEX FILE FORMATS

Date Jun 17, 2019

Version 1.0.7

Author Benedict Verhegghe <benedict.verhegghe@ugent.be>

Abstract

This document describes the native file formats used by pyFormex. There are currently two file formats: the pyFormex Project File (.pyf) and the pyFormex Geometry File (.pgf/.formex).

8.1 Introduction

pyFormex uses two native file formats to save data on a persistent medium: the pyFormex Project File (.pyf) and the pyFormex Geometry File (.pgf).

A Project File can store any pyFormex data and is the preferred way to store your data for later reuse within pyFormex. The data in the resulting file can normally not be used by humans and can only be easily restored by pyFormex itself.

The pyFormex Geometry File on the other hand can be used to exchange data between pyFormex projects or with other software. Because of its plain text format, the data can be read and even edited by humans. You may also wish to save data in this format to make them accessible to the need for pyFormex, or to bridge incompatible changes in pyFormex.

Because the geometrical data in pyFormex can be quite voluminous, the format has been chosen so as to allow efficient read and write operations from inside pyFormex. If you want a nicer layout and efficiency is not your concern, you can use the `fprint()` method of the geometry object.

8.2 pyFormex Project File Format

A pyFormex project file is just a pickled Python dictionary stored on file, possibly with compression. Any pyFormex objects can be exported and stored on the project file. The resulting file is normally not readable for humans and because all the class definitions of the exported data have to be present, the file can only be read back by pyFormex itself.

The format of the project file is therefore currently not further documented. See *Using Projects* for the use of project files from within pyFormex.

8.3 pyFormex Geometry File Format 1.6

This describes the pyFormex Geometry File Format (PGF) version 1.6 as drafted on 2013-03-10 and being used in pyFormex 0.9.0. The version numbering is such that implementations of a later version are able to read an older version with the same major numbering. Thus, the 1.6 version can still read version 1.5 files.

The preferred filename extension for pyFormex geometry files is `.pgf`, though this is not a requirement.

8.3.1 General principles

The PGF format consists of a sequence of records of two types: comment lines and data blocks. A record always ends with a newline character, but not all newline characters are record separators: data blocks may include multiple newlines as part of the data.

Comment records are ascii and start with a `#` character. Comment records are mostly used to announce the type and amount of data in the following data block(s). This is done by comment line containing a sequence of `'key=value'` statements, separated by semicolons (`;`).

Data blocks can be either ascii or binary, and are always announced by specially crafted comment lines preceding them. Note that even binary data blocks get a newline character at the end, to mark the end of the record.

8.3.2 Detailed layout

The pyFormex Geometry File starts with a header comment line identify the file type and version, and possibly specifying some global variables. For the version 1.6 format the first line may look like:

```
# pyFormex Geometry File (http://pyformex.org) version='1.6'; sep=' '
```

The version number is used to read back legacy formats in newer versions of pyFormex. The `sep = ' '` defines the default data separator for data blocks that do not specify it (see below).

The remainder of the file is a sequence of comment lines announcing data blocks, followed by those data blocks. The announcement line provides information about the number, type and size of data blocks that follow. This makes it possible to write and read the data using high speed functions (like `numpy.tofile` and `numpy.fromfile`) and without having to test any contents of the data. The data block information in the announcement line is provided by a number of `'key=value'` strings separated with a semicolon and optional whitespace.

Object type specific fields

For each object type that can be stored, there are some required fields and data blocks. In the examples below, `<int>` stands for an integer number, `<str>` for a string, and `<bool>` for either `True` or `False`.

- Formex: the announcement provides at least:

```
# objtype='Formex'; nelems=<int>; nplex=<int>
```

The data block following this line should contain exactly `nelems*nplex*3` floating point values: the 3 coordinates of the `nplex` points of the `nelems` elements of the Formex.

- Mesh: the announcement contains at least:

```
# objtype='Mesh'; ncoords=<int>; nelems=<int>; nplex=<int>
```

In this case two data blocks will follow: first `ncoords*3` float values with the coordinates of the nodes; then a block with `nelems*nplex` integer values: the connectivity table of the mesh.

- Curve:

Optional fields

The announcement line may contain other fields, usually to define extra attributes for the object:

- *props*=<bool> : If the value is True, another data block with *nelems* integer values follows. These are the property numbers of the object.
- *eltype*=<str> : Can also have the special value None. If specified and not None, it will be used to set the element type of the object.
- *name*=<str> : Name of the object. If specified, pyFormex will use this value as a key when returning the restored object.
- *sep*=<str> : This field defines how the data are stored. If it is not defined, the value from the file header is used.
 - An empty string means that the data blocks are written in binary. Floating point values are stored as little-endian 4byte floats, while integer values are stored as 4 byte integers.
 - Any other string makes the data being written in ascii mode, with the specified string used as a separator between any two values. When reading a PGF file, extra whitespace and newlines appearing around the separator are silently ignored.

8.3.3 Example

The following pyFormex script creates a PGF file containing two objects, a Formex with one square, and a Mesh with two triangles:

```
F = Formex('4:0123')
M = Formex('3:112.34').setProp(1).toMesh()
writeGeomFile('test.pgf', [F,M], sep=', ')
```

The Mesh has property numbers defined on it, the Formex doesn't. The data are written in ascii mode with ', ' as separator. Here is the resulting contents of the file 'test.pgf':

```
# pyFormex Geometry File (http://pyformex.org) version='1.6'; sep=', '
# objtype='Formex'; nelems=1; nplex=4; props=False; eltype=None; sep=', '
0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0
# objtype='Mesh'; ncoords=4; nelems=2; nplex=3; props=True; eltype='tri3'; sep=', '
1.0, 0.0, 0.0, 2.0, 0.0, 0.0, 1.0, 1.0, 0.0, 2.0, 1.0, 0.0
0, 1, 3, 3, 2, 0
1, 1
```

This file contains two objects: a Formex and a Mesh. The Formex has 1 element of plexitude 4 and no property numbers. Following its announcement is a single data block with $1 \times 4 \times 3 = 12$ coordinate values. The Mesh contains 2 elements of plexitude 3, has element type 'tri3' and contains property numbers. Following the announcement are three data blocks: first the 4×3 nodal coordinates, then the $2 \times 3 = 6$ entries in the connectivity table, and finally 2 property numbers.

BUMPIX LIVE GNU/LINUX SYSTEM

Abstract

This document gives a short introduction on the BuMPix Live GNU/Linux system and how to use it to run pyFormex directly on nearly any computer system without having to install it.

9.1 What is BuMPix

Bumpix Live is a fully featured GNU/Linux system including pyFormex that can be run from a single removable medium such as a CD or a USB key. BuMPix is still an experimental project, but new versions are already produced at regular intervals. While those are primarily intended for our students, the install images are made available for download on the [Bumpix Live GNU/Linux FTP server](#), so that anyone can use them.

All you need to use the [Bumpix Live GNU/Linux](#) is some proper PC hardware: the system boots and runs from the removable medium and leaves everything that is installed on the hard disk of the computer untouched.

Because the size of the image (since version 0.4) exceeds that of a CD, we no longer produce CD-images (.iso) by default, but some older images remain available on the server. New (reduced) CD images will only be created on request. On the other hand, USB-sticks of 2GB and larger have become very affordable and most computers nowadays can boot from a USB stick. USB sticks are also far more easy to work with than CD's: you can create a persistent partition where you can save your changes, while a CD can not be changed.

You can easily take your USB stick with you wherever you go, plug it into any available computer, and start or continue your previous pyFormex work. Some users even prefer this way to run pyFormex for that single reason. The Live system is also an excellent way to test and see what pyFormex can do for you, without having to install it. Or to demonstrate pyFormex to your friends or colleagues.

9.2 Obtain a BuMPix Live bootable medium

9.2.1 Download BuMPix

The numbering scheme of the BuMPix images is independent from the pyFormex numbering. Just pick the [latest BuMPix image](#) to get the most recent pyFormex available on USB stick. After you downloaded the .img file, write it to a USB stick as an image, not as file! Below, you find instructions on how to do this on a GNU/Linux system or on a Windows platform.

Warning: Make sure you've got the device designation correct, or you might end up overwriting your whole hard disk!

Also, be aware that the USB stick will no longer be usable to store your files under Windows.

9.2.2 Create the BuMPix USB stick under GNU/Linux

If you have an existing GNU/Linux system available, you can write the downloaded image to the USB-stick using the command:

```
dd if=bumpix-VERSION.img of=USBDEV
```

where `bumpix-VERSION.img` is the downloaded file and `USBDEV` is the device corresponding to your USB key. This should be `/dev/sda` or `/dev/sdb` or, generally, `/dev/sd?` where `?` is a single character from `a-z`. The value you should use depends on your hardware. You can find out the correct value by giving the command `dmesg` after you have plugged in the USB key. You will see messages mentioning the correct `[sd?]` device.

The `dd` command above will overwrite everything on the specified device, so copy your files off the stick before you start, and make sure you've got the device designation correct.

9.2.3 Create the BuMPix USB stick under Windows

If you have no GNU/Linux machine available to create the USB key, there are ways to do this under Windows as well. We recommend to use `dd` for Windows. You can then proceed as follows.

- Download `dd for Windows` to a folder, say `C:\download\ddWrite`.
- Download the latest BuMPix image to the same folder.
- Mount the target USB stick and look for the number of the mounted USB. This can be done with the command `c:\download\ddWrite dd --list`. Look at the description (Removable media) and the size to make sure you've got the correct harddisk designation (e.g. `harddisk1`).
- Write the image to the USB stick with the command, substituting the harddisk designation found above:

```
dd if=c:\download\ddwrite\bumpix-0.4-b1.img of=\\?\device\harddisk1\partition0_
↪bs=1M --progress
```

The `dd` command above will overwrite everything on the specified device, so copy your files off the stick before you start, and make sure you've got the device designation correct.

9.2.4 Buy a USB stick with BuMPix

Alternatively,

- if you do not succeed in properly writing the image to a USB key, or
- if you just want an easy solution without any install troubles, or
- if you want to financially support the further development of pyFormex, or
- if you need a large number of pyFormex USB installations,

you may be happy to know that we can provide ready-made BuMPix USB sticks with the `pyformex.org` logo at a cost hardly exceeding that of production and distribution. If you think this is the right choice for you, just [email us](#) for a quotation.



9.3 Boot your BuMPix system

Once the image has been written, reboot your computer from the USB stick. You may have to change your BIOS settings or use the boot menu to do that. On success, you will have a full GNU/Linux system running, containing pyFormex ready to use. There is even a start button in the toolbar at the bottom.

Warning: More detailed documentation on how to use the system is currently under preparation. For now, feel free to [email us](#) if you have any problems or urgent questions. But first check that your question is not solved in the FAQ below.

9.4 FAQ

A collection of hints and answers to frequently asked questions.

- 1) The initial user name is *user* and the password *live*.
- 2) On shutdown/reboot, the system pauses with the advice to remove the USB stick before hitting *ENTER* to proceed. We advice not to do this (especially when running in *PERSISTENT* mode): instead first hit *ENTER* and remove the USB stick when the screen goes black.

- 3) BuMPix 0.7.0 may contain a few user configuration files with incorrect owner settings. As a result some XFCE configuration may not be permanent. To solve the problem, you should run the following command in a terminal

```
sudo chown -R user:user /home/user
```

- 4) For BuMPix 0.7.0 (featuring pyFormex 0.8.4) with XFCE desktop, some users have reported occasional problems with starting the window manager. Windows remain undecorated and the mouse cursor keeps showing the *BUSY* symbol. This is probably caused by an improper previous shutdown and can be resolved as follows: open a terminal and enter the command `xm4`. That will start up the window manager for your current session and most likely will also remove the problem for your next sessions.
- 5) Install the latest pyFormex version from the SVN repository. The BuMPix stick contains a script `pyformex-svn` under the user's `bin` directory to install a pyFormex version directly from the SVN repository. However, the repository has been relocated to a new server and the script might still contain the old location. You can download a fixed script from <ftp://bumps.ugent.be/pub/pyformex/pyformex-svn>.

9.5 Upgrade the pyFormex version on a BuMPix-0.6.1 USB stick

This describes how you can upgrade (or downgrade) the pyFormex version on your BuMPix 0.6.1 USB key. You need to have network connection to do this.

- First, we need to fix some file ownerships. Open a Terminal and do the following

```
sudo -i
chown -R user:user /home/user
exit
```

- Then, add your own `bin` directory to the `PATH`:

```
echo 'export PATH=~/.bin:$PATH' >> ~/.bash_profile
```

- Change the configuration of your terminal. Click `Edit -> Profiles -> Edit -> Title and Command` and check the option 'Run command as a login shell'.
- Close the terminal and open a new one. Check that the previous operation went correct:

```
echo $PATH
```

- This should start with `~/home/user/bin`. If ok, then do:

```
cd bin
chmod +x pyformex-svn
ls
```

- You should now see a green 'pyformex-svn' script. Execute it as follows:

```
./pyformex-svn install makelib symlink
ls
```

- If everything went well, you should now also have a blue 'pyformex' link. Test it:

```
cd ..
pyformex
```

- The latest svn version of pyFormex should start. If ok, close it and you can make this the default version to start from the pyFormex button in the top panel. Right click on the button, then 'Properties'. Change the Command to:

```
bin/pyformex --redirect
```

- Now you should always have the updated pyformex running, from the command line as well as from the panel button. Next time you want to upgrade (or downgrade), you can just do:

```
cd pyformex-svn  
svn up
```

- or, for a downgrade, add a specific revision number:

```
svn up -r 1833
```


GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

10.1 Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

10.2 Terms and Conditions

10.2.1 0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

10.2.2 1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free

programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

10.2.3 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

10.2.4 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

10.2.5 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

10.2.6 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

- a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

- b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to “keep intact all notices”.
- c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
- d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation’s users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

10.2.7 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

- a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
- b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
- c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
- d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
- e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class

of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

10.2.8 7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
- c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
- e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

10.2.9 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

10.2.10 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10.2.11 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

10.2.12 11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

10.2.13 12. No Surrender of Others’ Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

10.2.14 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

10.2.15 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

10.2.16 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

10.2.17 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

10.2.18 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil

liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

End of Terms and Conditions

10.3 How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program> Copyright (C) <year> <name of author>
This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands show w and show c should show the appropriate parts of the General Public License. Of course, your program’s commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

ABOUT THE PYFORMEX DOCUMENTATION

Abstract

This document contains some meta information about the pyFormex documentation. You will learn nothing here about pyFormex. But if you are interested in knowing how the documentation is created and maintained, and who is responsible for this work, you will find some answers here.

11.1 The people who did it

Most of the manual was written by Benedict Verheghe, also the main author of pyFormex. There are contributions from Tim Neels, Matthieu De Beule and Peter Mortier.

11.2 How we did it

The documentation is written in `reStructuredText` and maintained with `Sphinx`.

GLOSSARY

Abstract

This glossary contains the description of terms used throughout the pyFormex documentation and that might not be immediately clear to first time users. Since it is expected that users will have at least a basic knowledge of Python, typical Python terms will not be explained here. Some NumPy terms are included.

array_like Any sequence that can be interpreted as an ndarray. This includes nested lists, tuples, scalars and existing arrays.

coords_like Either a Coords or data that can be used to initialize a Coords.

index An object that can be used as an index in a numpy ndarray. This includes Python style slicing and numpy advanced indexing.

level The dimensionality of a basic geometric entity. The highest level (3) are volumetric entities (cells). Surfaces (and faces of cells) are level 2. Lines (including face edges) are level 1. Finally, points are level 0.

mapping A container object that supports arbitrary key lookups. Examples include Python's dict, defaultdict, OrderedDict and pyFormex's Dict and CDict.

Mesh A geometric model where entities are represented by a combination of a table of coordinates of all points (nodes) and a connectivity table holding for each element the indices of the included nodes.

node A point in a *Mesh* type geometric model.

path_like An object that holds the path name of a file or directory. This includes str and pathlib.Path.

plexitude The number of points used to describe a basic geometric entity. For example, a straight line segment has plexitude 2, a triangle has plexitude 3, a quadrilateral and a tetrahedron have plexitude 4. And a point obviously has plexitude 1.

qimage_like A QImage, or data that can be converted to a QImage, e.g. the name of a raster image file.

re A Python regular expression.

seed Data that can be used as argument in the *smartSeed()* function. This means either a single int, a tuple containing an int and optionally one or two end attractors, or a sorted list of float values in the range 0.0 to 1.0.

PYTHON MODULE INDEX

a

adjacency, 350
arraytools, 157
attributes, 552

c

collection, 557
config, 554
connectivity, 294
coords, 73
coordsys, 278

e

elements, 311

f

field, 322
fileread, 385
filewrite, 386
flatkeydb, 573
formex, 127

g

geometry, 282
geomfile, 362
geomtools, 367
gui.appMenu, 425
gui.colorscales, 413
gui.draw, 216
gui.image, 422
gui.imageView, 425
gui.menu, 410
gui.toolbar, 428
gui.viewport, 415
gui.widgets, 393

i

inertia, 380

m

mesh, 235
multi, 388

mydict, 549

o

olist, 559
opengl.camera, 519
opengl.canvas, 526
opengl.colors, 232
opengl.decor, 533
opengl.drawable, 535
opengl.matrix, 539
opengl.objectdialog, 542
opengl.renderer, 542
opengl.sanitize, 542
opengl.scene, 544
opengl.shader, 545
opengl.texttext, 546
opengl.texture, 548

p

path, 561
plugins.bifmesh, 429
plugins.cameratools, 430
plugins.ccxdat, 431
plugins.ccxinp, 431
plugins.curve, 432
plugins.datareader, 448
plugins.dxf, 448
plugins.export, 451
plugins.fe, 452
plugins.fe_abq, 453
plugins.fe_post, 474
plugins.flavia, 475
plugins.imagearray, 476
plugins.isopar, 480
plugins.isosurface, 482
plugins.lima, 482
plugins.neu_exp, 483
plugins.nurbs, 484
plugins.objects, 495
plugins.partition, 498
plugins.plot2d, 498
plugins.polygon, 499

- plugins.polynomial, 500
- plugins.postproc, 501
- plugins.properties, 502
- plugins.pyformex_gts, 507
- plugins.section2d, 508
- plugins.sectionize, 509
- plugins.tetgen, 509
- plugins.tools, 512
- plugins.turtle, 513
- plugins.units, 514
- plugins.web, 516
- plugins.webgl, 516
- project, 359

S

- script, 212
- simple, 355
- software, 390

t

- timer, 576
- track, 577
- trisurface, 264

U

- utils, 325

V

- varray, 343

Symbols

`__add__()` (*formex.Formex* method), 138
`__getitem__()` (*mesh.Mesh* method), 239

A

`abat()` (*in module arraytools*), 185
`abq_eltype()` (*in module plugins.ccxinp*), 431
`AbqData` (*class in plugins.fe_abq*), 456
`abqInputNames()` (*in module plugins.fe_abq*), 457
`absolute()` (*path.Path* method), 566
`accept_draw()` (*gui.viewport.QtCanvas* method), 419
`accept_drawing()` (*gui.viewport.QtCanvas* method), 419
`accept_selection()` (*gui.viewport.QtCanvas* method), 417
`acceptData()` (*gui.widgets.InputDialog* method), 402
`ack()` (*in module gui.draw*), 216
`ack()` (*in module script*), 213
`action()` (*gui.menu.BaseMenu* method), 411
`ActionList` (*class in gui.menu*), 412
`actionList()` (*gui.menu.BaseMenu* method), 411
`actionsLike()` (*gui.menu.BaseMenu* method), 411
`activate()` (*opengl.canvas.CanvasSettings* method), 527
`activate()` (*opengl.texttext.FontTexture* method), 547
`activate()` (*opengl.texture.Texture* method), 548
`Actor` (*class in opengl.drawable*), 537
`actor()` (*plugins.nurbs.Coords4* method), 486
`actor()` (*plugins.nurbs.NurbsCurve* method), 492
`actor()` (*plugins.nurbs.NurbsSurface* method), 493
`actorDialog()` (*in module plugins.tools*), 513
`add()` (*collection.Collection* method), 558
`add()` (*gui.appMenu.AppMenu* method), 427
`add()` (*gui.menu.ActionList* method), 412
`add()` (*opengl.scene.ItemList* method), 544
`add()` (*plugins.fe_abq.Command* method), 454
`Add()` (*plugins.units.UnitsSystem* method), 515
`add_focus_rectangle()` (*opengl.canvas.Canvas* method), 532
`add_group()` (*gui.widgets.InputDialog* method), 401
`add_hbox()` (*gui.widgets.InputDialog* method), 401
`add_input()` (*gui.widgets.InputDialog* method), 402
`add_items()` (*gui.widgets.InputDialog* method), 401
`add_tab()` (*gui.widgets.InputDialog* method), 401
`addActionButtons()` (*in module gui.toolbar*), 428
`addActionButtons()` (*in module gui.widgets*), 410
`addActor()` (*plugins.webgl.WebGL* method), 518
`addAny()` (*opengl.scene.Scene* method), 544
`addAxis()` (*in module arraytools*), 162
`addButton()` (*in module gui.toolbar*), 428
`addCameraButtons()` (*in module gui.toolbar*), 428
`addCheck()` (*gui.widgets.MessageBox* method), 407
`addCompressedTypes()` (*in module utils*), 333
`added()` (*utils.DictDiff* method), 328
`addElem()` (*in module plugins.tetgen*), 510
`addFeResult()` (*in module plugins.ccxdat*), 431
`addField()` (*geometry.Geometry* method), 292
`addHighlightElements()` (*opengl.drawable.Actor* method), 538
`addHighlightPoints()` (*opengl.drawable.Actor* method), 538
`addMaterial()` (*plugins.properties.ElemSection* method), 503
`addMeanNodes()` (*mesh.Mesh* method), 252
`addNodes()` (*mesh.Mesh* method), 252
`addNoise()` (*coords.Coords* method), 111
`addNoise()` (*geometry.Geometry* method), 288
`addRule()` (*plugins.lima.Lima* method), 482
`addScene()` (*plugins.webgl.WebGL* method), 517
`addSection()` (*plugins.properties.ElemSection* method), 503
`addTimeout()` (*in module gui.widgets*), 408
`addTimeoutButton()` (*in module gui.toolbar*), 429
`addView()` (*gui.viewport.MultiCanvas* method), 421
`addViewport()` (*built-in function*), 48
`addViewport()` (*in module gui.draw*), 221
`Adjacency` (*class in adjacency*), 350
`adjacency` (*module*), 350
`adjacency()` (*connectivity.Connectivity* method), 301
`adjacency()` (*mesh.Mesh* method), 247
`adjacencyArrays()` (*in module trisurface*), 276
`adjacentElements()` (*connectivity.Connectivity* method), 302

adjacentTo() (*mesh.Mesh* method), 247
 adjust() (*coords.Coords* method), 115
 adjust() (*geometry.Geometry* method), 286
 affine() (*coords.Coords* method), 94
 affine() (*geometry.Geometry* method), 286
 align() (*coords.Coords* method), 91
 align() (*geometry.Geometry* method), 286
 align() (*in module coords*), 126
 Amplitude (*class in plugins.properties*), 503
 an() (*in module plugins.turtle*), 514
 angles (*opengl.camera.Camera* attribute), 521
 angles() (*plugins.polygon.Polygon* method), 499
 annotate() (*in module gui.draw*), 231
 anyPerpendicularVector() (*in module geom-
tools*), 380
 append() (*coords.Coords* method), 116
 append() (*flatkeydb.FlatDB* method), 575
 append() (*plugins.curve.PolyLine* method), 441
 append() (*plugins.objects.Objects* method), 495
 append() (*track.TrackedList* method), 578
 append() (*trisurface.TriSurface* method), 264
 apply() (*opengl.camera.Camera* method), 525
 AppMenu (*class in gui.appMenu*), 425
 approx() (*plugins.curve.Arc* method), 446
 approx() (*plugins.curve.Curve* method), 435
 approx() (*plugins.nurbs.NurbsCurve* method), 491
 approx() (*plugins.nurbs.NurbsSurface* method), 493
 approxAt() (*plugins.curve.Curve* method), 435
 approximate() (*plugins.curve.Curve* method), 435
 Arc (*class in plugins.curve*), 446
 arc() (*plugins.dxf.DxfExporter* method), 449
 arc2points() (*in module plugins.curve*), 446
 Arc3 (*class in plugins.curve*), 446
 arccosd() (*in module arraytools*), 172
 arcsind() (*in module arraytools*), 172
 arctand() (*in module arraytools*), 172
 arctand2() (*in module arraytools*), 173
 area() (*mesh.Mesh* method), 260
 area() (*plugins.polygon.Polygon* method), 500
 areaNormals() (*in module geomtools*), 376
 areaNormals() (*trisurface.TriSurface* method), 265
 areas() (*formex.Formex* method), 153
 areas() (*mesh.Mesh* method), 259
 areas() (*trisurface.TriSurface* method), 265
 argNearestValue() (*in module arraytools*), 196
 array2image() (*in module plugins.imagearray*), 478
 array2str() (*in module arraytools*), 168
 array_like, 609
 ArrayModel (*class in gui.widgets*), 403
 arraytools (*module*), 157
 as_uri() (*path.Path* method), 566
 asArray() (*formex.Formex* method), 137
 asCoords() (*formex.Formex* method), 137
 asFormex() (*formex.Formex* method), 137

asFormexWithProp() (*formex.Formex* method), 137
 ask() (*in module gui.draw*), 216
 ask() (*in module script*), 212
 ask() (*plugins.objects.DrawableObjects* method), 497
 ask() (*plugins.objects.Objects* method), 496
 ask1() (*plugins.objects.Objects* method), 496
 askDirname() (*in module gui.draw*), 219
 askFile() (*in module gui.draw*), 218
 askFilename() (*in module gui.draw*), 218
 askItems() (*in module gui.draw*), 217
 askNewFilename() (*in module gui.draw*), 219
 aspectRatio() (*trisurface.TriSurface* method), 268
 asPoints() (*formex.Formex* method), 140
 asym (*inertia.Tensor* attribute), 382
 atApproximate() (*plugins.curve.Curve* method),
434
 atba() (*in module arraytools*), 186
 atChordal() (*plugins.curve.Contour* method), 445
 atChordal() (*plugins.curve.Curve* method), 435
 atLength() (*plugins.curve.BezierSpline* method), 444
 atLength() (*plugins.curve.PolyLine* method), 440
 atoms() (*plugins.polynomial.Polynomial* method), 501
 attrib (*geometry.Geometry* attribute), 283
 attrib (*mesh.Mesh* attribute), 237
 Attributes (*class in attributes*), 552
 attributes (*module*), 552
 autoExport() (*in module script*), 213
 autoName() (*in module utils*), 340
 autorun, 58
 autoSaveOn() (*in module gui.image*), 424
 average() (*coords.Coords* method), 80
 averageNormals() (*in module geomtools*), 378
 avgDirections() (*plugins.curve.PolyLine* method),
439
 avgNodes() (*mesh.Mesh* method), 252
 avgVertexNormals() (*trisurface.TriSurface*
method), 265
 axes (*coordsys.CoordSys* attribute), 280
 axis (*opengl.camera.Camera* attribute), 521
 axis() (*coordsys.CoordSys* method), 280

B

b_avgnormals (*opengl.drawable.Actor* attribute), 538
 b_normals (*opengl.drawable.Actor* attribute), 537
 back() (*in module opengl.renderer*), 542
 baryCoords() (*in module geomtools*), 380
 BaseActor (*class in opengl.drawable*), 536
 BaseMenu (*class in gui.menu*), 410
 bbox (*opengl.scene.Scene* attribute), 544
 bbox() (*coords.Coords* method), 78
 bbox() (*geometry.Geometry* method), 284
 bbox() (*in module coords*), 123
 bbox() (*plugins.nurbs.Coords4* method), 486
 bbox() (*plugins.nurbs.NurbsCurve* method), 488

- bbox()** (*plugins.nurbs.NurbsSurface method*), 492
BboxActor (*class in opengl.decors*), 533
bboxes() (*coords.Coords method*), 83
bboxes() (*geometry.Geometry method*), 285
bboxes() (*mesh.Mesh method*), 242
bboxIntersection() (*in module coords*), 123
bboxPoint() (*coords.Coords method*), 79
bboxPoint() (*geometry.Geometry method*), 284
bboxPoints() (*coords.Coords method*), 79
begin_2D_drawing() (*opengl.canvas.Canvas method*), 530
bezierPowerMatrix() (*in module plugins.curve*), 447
BezierSpline (*class in plugins.curve*), 442
bgcolor() (*in module gui.draw*), 231
bind() (*opengl.shader.Shader method*), 546
binomial() (*in module plugins.curve*), 447
binomialCoeffs() (*in module plugins.curve*), 447
blend() (*plugins.nurbs.NurbsCurve method*), 491
boolean() (*in module plugins.pyformex_gts*), 507
boolean() (*trisurface.TriSurface method*), 275
border() (*trisurface.TriSurface method*), 267
borderEdgeNrs() (*trisurface.TriSurface method*), 266
borderEdges() (*trisurface.TriSurface method*), 266
borderMesh() (*mesh.Mesh method*), 246
borderNodeNrs() (*trisurface.TriSurface method*), 266
boundingBox() (*in module simple*), 359
boxes() (*coords.Coords method*), 113
boxes() (*in module simple*), 359
boxes2d() (*in module simple*), 359
breakpt() (*in module script*), 213
bsphere() (*coords.Coords method*), 82
bsphere() (*geometry.Geometry method*), 284
bump() (*coords.Coords method*), 102
bump() (*geometry.Geometry method*), 287
ButtonBox (*class in gui.widgets*), 407
- ## C
- Camera** (*class in opengl.camera*), 519
cancel_draw() (*gui.viewport.QtCanvas method*), 419
cancel_drawing() (*gui.viewport.QtCanvas method*), 419
cancel_selection() (*gui.viewport.QtCanvas method*), 417
Canvas (*class in opengl.canvas*), 527
CanvasMouseHandler (*class in gui.viewport*), 415
CanvasSettings (*class in opengl.canvas*), 526
canvasSize() (*in module gui.draw*), 232
cardanAngles() (*in module arraytools*), 180
CardinalSpline (*class in plugins.curve*), 445
CardinalSpline2 (*class in plugins.curve*), 445
cclip() (*geometry.Geometry method*), 291
CDict (*class in mydict*), 551
cellType() (*gui.widgets.ArrayModel method*), 404
cellType() (*gui.widgets.TableModel method*), 403
center() (*coords.Coords method*), 79
center() (*geometry.Geometry method*), 284
centered() (*coords.Coords method*), 91
centered() (*geometry.Geometry method*), 286
centerline() (*in module plugins.sectionize*), 509
centralCS() (*coords.Coords method*), 85
centroid() (*coords.Coords method*), 81
centroid() (*geometry.Geometry method*), 284
centroids() (*coords.Coords method*), 81
centroids() (*formex.Formex method*), 135
centroids() (*mesh.Mesh method*), 241
chain() (*connectivity.Connectivity method*), 308
chained() (*connectivity.Connectivity method*), 309
changeBackgroundColorXPM() (*in module gui.image*), 424
changed() (*utils.DictDiff method*), 328
changeFilename() (*gui.widgets.InputFilename method*), 399
changeLayout() (*gui.viewport.MultiCanvas method*), 420
changeMode() (*opengl.drawable.Actor method*), 538
changeMode() (*opengl.scene.Scene method*), 544
changeSize() (*gui.viewport.QtCanvas method*), 415
changeValues() (*plugins.objects.Objects method*), 496
changeVertexColor() (*opengl.drawable.Drawable method*), 536
chdir() (*in module script*), 214
check() (*plugins.objects.Objects method*), 496
check() (*trisurface.TriSurface method*), 272
checkAllExternals() (*in module software*), 391
checkAllModules() (*in module software*), 390
checkArray() (*in module arraytools*), 160
checkArray1D() (*in module arraytools*), 161
checkArrayOrIdValue() (*in module plugins.properties*), 505
checkArrayOrIdValueOrEmpty() (*in module plugins.properties*), 506
checkBorder() (*trisurface.TriSurface method*), 267
checkBroadcast() (*in module arraytools*), 159
checkDict() (*in module software*), 392
checkDict() (*opengl.canvas.CanvasSettings class method*), 527
checkExternal() (*in module software*), 391
checkFloat() (*in module arraytools*), 159
checkIdValue() (*in module plugins.properties*), 505
checkImageFormat() (*in module gui.image*), 423
checkInt() (*in module arraytools*), 159
checkKeys() (*flatkeydb.FlatDB method*), 575
checkModule() (*in module software*), 390

- checkPrintSyntax() (in module script), 213
 checkRecord() (flatkeydb.FlatDB method), 575
 checkSelfIntersections() (in module plugins.tetgen), 512
 checkSoftware() (in module software), 392
 checkString() (in module plugins.properties), 506
 checkUniqueNumbers() (in module arraytools), 161
 checkVersion() (in module software), 390
 checkWorkdir() (in module gui.draw), 219
 chmod() (path.Path method), 569
 circle() (in module plugins.curve), 446
 circle() (in module simple), 356
 circulize() (coords.Coords method), 102
 circulize() (geometry.Geometry method), 287
 circulize1() (formex.Formex method), 143
 clamp() (plugins.nurbs.NurbsCurve method), 490
 classify() (in module gui.appMenu), 427
 clear() (in module gui.draw), 232
 clear() (opengl.scene.ItemList method), 544
 clear() (opengl.scene.Scene method), 545
 clear() (plugins.objects.Objects method), 496
 clear() (track.TrackedDict method), 578
 clear() (track.TrackedList method), 578
 clearCanvas() (opengl.canvas.Canvas method), 530
 clip() (geometry.Geometry method), 291
 clipAtPlane() (mesh.Mesh method), 259
 clipToEye() (opengl.camera.Camera method), 523
 close() (geomfile.GeometryFile method), 363
 close() (plugins.curve.PolyLine method), 438
 close() (plugins.dxf.DxfExporter method), 449
 close() (trisurface.TriSurface method), 267
 close() (utils.File method), 326
 closeDialog() (in module gui.draw), 216
 closeGui() (in module gui.draw), 216
 closest() (in module geomtools), 377
 closestColorName() (in module opengl.colors), 235
 closestPair() (in module geomtools), 378
 closestToPoint() (coords.Coords method), 86
 coarsen() (plugins.curve.PolyLine method), 442
 coarsen() (trisurface.TriSurface method), 272
 col() (varray.Varray method), 347
 colindex() (varray.Varray method), 347
 collapseEdge() (trisurface.TriSurface method), 268
 collectByType() (in module plugins.dxf), 450
 Collection (class in collection), 557
 collection (module), 557
 collectOnLength() (in module arraytools), 191
 color() (gui.colorscales.ColorLegend method), 414
 color() (gui.colorscales.ColorScale method), 414
 colorCut() (in module plugins.partition), 498
 colorindex() (in module gui.draw), 231
 ColorLegend (class in gui.colorscales), 414
 ColorLegend (class in opengl.decor), 534
 colormap() (in module gui.draw), 231
 colorName() (in module opengl.colors), 234
 ColorScale (class in gui.colorscales), 413
 columnCount() (gui.widgets.ArrayModel method), 404
 columnCount() (gui.widgets.TableModel method), 403
 colWidths() (gui.widgets.Table method), 404
 combine() (connectivity.Connectivity method), 306
 Command (class in plugins.fe_abq), 454
 command() (in module utils), 329
 commonpath() (path.Path method), 567
 commonprefix() (path.Path method), 567
 comp() (field.Field method), 324
 compact() (mesh.Mesh method), 251
 compareVersion() (in module software), 391
 complement() (in module arraytools), 191
 computeAveragedNodalStresses() (in module plugins.ccxdat), 431
 computeSection() (plugins.properties.ElemSection method), 503
 concat() (in module arraytools), 166
 concatenate() (coords.Coords class method), 117
 concatenate() (formex.Formex class method), 138
 concatenate() (in module olist), 560
 concatenate() (mesh.Mesh class method), 258
 concatenate() (plugins.curve.PolyLine static method), 441
 Config (class in config), 554
 config (module), 554
 config() (gui.viewport.MultiCanvas method), 421
 config() (opengl.camera.Camera method), 525
 config2soft() (in module software), 392
 connect() (connectivity.Connectivity static method), 310
 connect() (in module formex), 154
 connect() (mesh.Mesh method), 256
 connectCurves() (in module simple), 358
 connectedElements() (mesh.Mesh method), 256
 connectedTo() (connectivity.Connectivity method), 300
 connectedTo() (mesh.Mesh method), 247
 Connectivity (class in connectivity), 294
 connectivity (module), 294
 connectPoints() (in module plugins.sectionize), 509
 continuousCurves() (in module mesh), 263
 Contour (class in plugins.curve), 445
 contracted (inertia.Tensor attribute), 381
 convert() (field.Field method), 324
 convert() (mesh.Mesh method), 253
 convert() (project.Project method), 362
 convertDXF() (in module plugins.dxf), 450

- convertField() (*geometry.Geometry* method), 293
 convertFormexToCurve() (*in module plugins.curve*), 447
 convertInp() (*in module fileread*), 386
 convertInputItem() (*in module gui.widgets*), 409
 convertPrintSyntax() (*in module script*), 213
 convertRandom() (*mesh.Mesh* method), 253
 convertUnits() (*in module plugins.units*), 515
 convexHull() (*coords.Coords* method), 120
 convexHull() (*geometry.Geometry* method), 285
 coord() (*formex.Formex* method), 134
 Coordinates() (*plugins.fe_post.FeResult* method), 475
 Coords (*class in coords*), 74
 coords (*coords.Coords* attribute), 77
 coords (*geometry.Geometry* attribute), 283
 coords (*mesh.Mesh* attribute), 236
 coords (*module*), 73
 coords (*opengl.drawable.Actor* attribute), 537
 Coords4 (*class in plugins.nurbs*), 484
 coords_like, 609
 CoordsBox (*class in gui.widgets*), 408
 CoordSys (*class in coordsys*), 278
 coordsys (*module*), 278
 CoordSystem (*class in plugins.properties*), 503
 copy() (*geometry.Geometry* method), 290
 copy() (*path.Path* method), 569
 copy() (*plugins.nurbs.KnotVector* method), 487
 copy() (*plugins.nurbs.NurbsCurve* method), 488
 copy() (*track.TrackedDict* method), 578
 copy() (*track.TrackedList* method), 578
 copyAxes() (*coords.Coords* method), 106
 copyAxes() (*geometry.Geometry* method), 288
 cosAngles() (*plugins.curve.PolyLine* method), 439
 cosd() (*in module arraytools*), 171
 cosd() (*in module plugins.turtle*), 513
 count() (*track.TrackedList* method), 578
 countLines() (*in module utils*), 338
 cpAllSections() (*in module plugins.bifmesh*), 430
 cpBoundaryLayer() (*in module plugins.bifmesh*), 430
 cpOneSection() (*in module plugins.bifmesh*), 430
 cpQuarterLumen() (*in module plugins.bifmesh*), 430
 cpStackQ16toH64() (*in module plugins.bifmesh*), 430
 create_insert_action() (*gui.menu.BaseMenu* method), 411
 createAppMenu() (*in module gui.appMenu*), 428
 createBackground() (*opengl.canvas.Canvas* method), 529
 createdBy() (*in module plugins.webgl*), 519
 createFeResult() (*in module plugins.flavia*), 476
 createHistogram() (*in module plugins.plot2d*), 499
 createMovie() (*in module gui.image*), 424
 createMultiWebGL() (*in module plugins.webgl*), 519
 createResultDB() (*in module plugins.ccxdat*), 431
 createSegments() (*in module plugins.sectionize*), 509
 createView() (*in module gui.draw*), 230
 createWebglHtml() (*in module plugins.webgl*), 519
 cselect() (*geometry.Geometry* method), 290
 cselectProp() (*geometry.Geometry* method), 292
 Cube() (*in module simple*), 356
 cubicEquation() (*in module arraytools*), 187
 cuboid() (*in module simple*), 359
 cuboid2d() (*in module simple*), 359
 cumLengths() (*plugins.curve.PolyLine* method), 440
 cumsum0() (*in module arraytools*), 189
 currentDialog() (*in module gui.draw*), 218
 currentView() (*gui.viewport.MultiCanvas* method), 421
 CursorShapeHandler (*class in gui.viewport*), 415
 curvature() (*in module trisurface*), 277
 curvature() (*plugins.nurbs.NurbsCurve* method), 489
 curvature() (*trisurface.TriSurface* method), 266
 Curve (*class in plugins.curve*), 432
 cutWithPlane() (*formex.Formex* method), 152
 cutWithPlane() (*plugins.curve.PolyLine* method), 441
 cutWithPlane() (*trisurface.TriSurface* method), 270
 cutWithPlane1() (*trisurface.TriSurface* method), 270
 cwd() (*path.Path* class method), 573
 cylinder() (*in module simple*), 358
 cylindrical() (*coords.Coords* method), 97
 cylindrical() (*geometry.Geometry* method), 287
- ## D
- DAction (*class in gui.menu*), 412
 data() (*gui.widgets.TableModel* method), 403
 Database (*class in plugins.properties*), 502
 deCasteljau() (*in module plugins.nurbs*), 494
 decode() (*geomfile.GeometryFile* method), 365
 decompose() (*plugins.nurbs.NurbsCurve* method), 490
 decorate() (*in module gui.draw*), 231
 default (*elements.ElementType* attribute), 315
 default() (*opengl.text.FontTexture* class method), 547
 defaultItemType() (*in module gui.widgets*), 409
 defaultMonoFont() (*in module utils*), 342
 defaultShaders() (*in module opengl.shader*), 546
 DEG (*in module arraytools*), 158
 degenerate() (*in module geomtools*), 376
 degenerate() (*trisurface.TriSurface* method), 268

- degree() (*plugins.polynomial.Polynomial method*), 500
- degrees() (*plugins.polynomial.Polynomial method*), 500
- delay() (*in module gui.draw*), 219
- delete() (*opengl.scene.ItemList method*), 544
- delete() (*project.Project method*), 362
- delField() (*geometry.Geometry method*), 293
- delProp() (*plugins.objects.DrawableObjects method*), 497
- delProp() (*plugins.properties.PropertyDB method*), 504
- denormalize() (*in module opengl.camera*), 526
- deNormalize() (*plugins.nurbs.Coords4 method*), 485
- deprecated() (*in module utils*), 328
- deprecated_by() (*in module utils*), 328
- deprecated_future() (*in module utils*), 329
- derivs() (*plugins.nurbs.NurbsCurve method*), 488
- derivs() (*plugins.nurbs.NurbsSurface method*), 492
- det2() (*in module arraytools*), 201
- det3() (*in module arraytools*), 202
- det4() (*in module arraytools*), 202
- detectedSoftware() (*in module software*), 391
- dialogAccepted() (*in module gui.draw*), 218
- dialogRejected() (*in module gui.draw*), 218
- dialogTimedOut() (*in module gui.draw*), 218
- dicom2numpy() (*in module plugins.imagearray*), 480
- DicomStack (*class in plugins.imagearray*), 476
- DicomStack.Object (*class in plugins.imagearray*), 477
- Dict (*class in mydict*), 549
- DictDiff (*class in utils*), 328
- dictStr() (*in module utils*), 342
- difference() (*in module olist*), 559
- directionalExtremes() (*coords.Coords method*), 87
- directionalExtremes() (*geometry.Geometry method*), 285
- directionalSize() (*coords.Coords method*), 86
- directionalSize() (*geometry.Geometry method*), 285
- directionalWidth() (*coords.Coords method*), 88
- directionalWidth() (*geometry.Geometry method*), 285
- directions() (*plugins.curve.PolyLine method*), 439
- directionsAt() (*plugins.curve.Curve method*), 434
- dirs() (*path.Path method*), 570
- diskSpace() (*in module utils*), 335
- displaceLines() (*in module geomtools*), 379
- Displacements() (*plugins.fe_post.FeResult method*), 475
- display() (*opengl.canvas.Canvas method*), 530
- dist (*opengl.camera.Camera attribute*), 521
- distance() (*in module geomtools*), 377
- distanceFromLine() (*coords.Coords method*), 85
- distanceFromLine() (*geometry.Geometry method*), 285
- distanceFromLine() (*in module geomtools*), 374
- distanceFromPlane() (*coords.Coords method*), 86
- distanceFromPlane() (*geometry.Geometry method*), 285
- distanceFromPoint() (*coords.Coords method*), 85
- distanceFromPoint() (*geometry.Geometry method*), 285
- distanceOfPoints() (*trisurface.TriSurface method*), 268
- do_after() (*in module gui.draw*), 220
- do_alphablend() (*opengl.canvas.Canvas method*), 529
- do_ELEMENT() (*in module plugins.ccxinp*), 432
- do_HEADING() (*in module plugins.ccxinp*), 432
- do_lighting() (*opengl.canvas.Canvas method*), 529
- do_NODE() (*in module plugins.ccxinp*), 432
- do_nothing() (*plugins.fe_post.FeResult method*), 474
- do_PART() (*in module plugins.ccxinp*), 432
- do_SYSTEM() (*in module plugins.ccxinp*), 432
- do_wiremode() (*opengl.canvas.Canvas method*), 529
- doFunc() (*gui.widgets.InputButton method*), 399
- doFunc() (*gui.widgets.InputPush method*), 397
- dofunc() (*in module multi*), 389
- doHeader() (*geomfile.GeometryFile method*), 365
- dolly() (*opengl.camera.Camera method*), 522
- dos2unix() (*in module utils*), 337
- dotpr() (*in module arraytools*), 175
- dotpr() (*in module gui.viewport*), 421
- download() (*in module plugins.web*), 516
- download3d() (*in module plugins.web*), 516
- draw() (*in module gui.draw*), 223
- draw_bbox() (*in module plugins.objects*), 498
- draw_cursor() (*opengl.canvas.Canvas method*), 532
- draw_elem_numbers() (*in module plugins.objects*), 498
- draw_free_edges() (*in module plugins.objects*), 498
- draw_node_numbers() (*in module plugins.objects*), 498
- draw_nodes() (*in module plugins.objects*), 498
- draw_object_name() (*in module plugins.objects*), 498
- draw_sorted_objects() (*opengl.canvas.Canvas method*), 530
- draw_state_line() (*gui.viewport.QtCanvas method*), 420
- draw_state_rect() (*gui.viewport.QtCanvas method*), 420
- Drawable (*class in opengl.drawable*), 535
- drawable() (*in module gui.draw*), 223
- DrawableObjects (*class in plugins.objects*), 497

- drawActor() (in module *gui.draw*), 229
- drawAnnotation() (plugins.objects.DrawableObjects method), 497
- drawAny() (in module *gui.draw*), 229
- drawAxes() (in module *gui.draw*), 228
- drawBbox() (in module *gui.draw*), 227
- drawChanges() (plugins.objects.DrawableObjects method), 497
- drawCircles() (in module *plugins.sectionize*), 509
- drawDot() (in module *gui.viewport*), 422
- drawField() (in module *gui.draw*), 229
- drawFreeEdges() (in module *gui.draw*), 227
- drawGrid() (in module *gui.viewport*), 422
- drawImage() (in module *gui.draw*), 229
- drawImage3D() (in module *gui.draw*), 228
- drawit() (*opengl.canvas.Canvas* method), 530
- drawLine() (in module *gui.viewport*), 422
- drawLinesInter() (*gui.viewport.QtCanvas* method), 419
- drawLinesInter() (in module *gui.draw*), 222
- drawMarks() (in module *gui.draw*), 226
- drawn() (*opengl.scene.Scene* method), 545
- drawn_as() (in module *gui.draw*), 223
- drawNumbers() (in module *gui.draw*), 227
- drawPrincipal() (in module *gui.draw*), 228
- drawPropNumbers() (in module *gui.draw*), 227
- drawRect() (in module *gui.viewport*), 422
- drawText() (in module *gui.draw*), 227
- drawText3D() (in module *gui.draw*), 228
- drawVectors() (in module *gui.draw*), 226
- drawVertexNumbers() (in module *gui.draw*), 227
- drawViewportAxes3D() (in module *gui.draw*), 228
- dsize() (*coords.Coords* method), 82
- dsize() (*geometry.Geometry* method), 284
- dualMesh() (*trisurface.TriSurface* method), 269
- DxfExporter (class in *plugins.dxf*), 449
- dynapan() (*gui.viewport.QtCanvas* method), 419
- dynarot() (*gui.viewport.QtCanvas* method), 419
- dynazoom() (*gui.viewport.QtCanvas* method), 419
- ## E
- edgeAdjacency() (*mesh.Mesh* method), 250
- edgeAngles() (*trisurface.TriSurface* method), 267
- edgeConnections() (*mesh.Mesh* method), 249
- edgeCosAngles() (*trisurface.TriSurface* method), 267
- edgeDistance() (in module *geomtools*), 376
- edgeLengths() (*trisurface.TriSurface* method), 267
- EdgeLoad (class in *plugins.properties*), 503
- edgeMesh() (*mesh.Mesh* method), 244
- edges (*opengl.drawable.Actor* attribute), 537
- edgeSignedAngles() (*trisurface.TriSurface* method), 267
- edit_drawing() (*gui.viewport.QtCanvas* method), 419
- editAnnotations() (plugins.objects.DrawableObjects method), 497
- editFile() (in module *gui.draw*), 217
- element, 13
- element() (*formex.Formex* method), 134
- element2str() (*formex.Formex* class method), 136
- elementClass() (in module *plugins.fe_abq*), 461
- elements (module), 311
- ElementType (class in *elements*), 311
- ElemLoad (class in *plugins.properties*), 503
- elemNrs() (*plugins.fe.Model* method), 452
- elemProp() (*plugins.properties.PropertyDB* method), 505
- Elms (class in *elements*), 317
- elems (*mesh.Mesh* attribute), 237
- elems (*opengl.drawable.Actor* attribute), 537
- ElemSection (class in *plugins.properties*), 502
- elevateDegree() (*plugins.nurbs.NurbsCurve* method), 491
- elName() (*formex.Formex* method), 133
- elName() (*mesh.Mesh* method), 239
- eltype (*formex.Formex* attribute), 131
- eltype (*mesh.Mesh* attribute), 238
- elType() (*formex.Formex* method), 132
- elType() (*mesh.Mesh* method), 239
- emit_cancel() (*gui.viewport.QtCanvas* method), 420
- emit_done() (*gui.viewport.QtCanvas* method), 420
- end_2D_drawing() (*opengl.canvas.Canvas* method), 531
- endPoints() (*plugins.curve.Contour* method), 445
- endPoints() (*plugins.curve.Curve* method), 433
- endSection() (*plugins.dxf.DxfExporter* method), 449
- Engineering() (*plugins.units.UnitsSystem* method), 515
- entities() (*plugins.dxf.DxfExporter* method), 449
- equal() (*utils.DictDiff* method), 328
- equalRows() (in module *arraytools*), 194
- error() (in module *gui.draw*), 216
- error() (in module *script*), 213
- esetName() (in module *plugins.fe_abq*), 457
- eval() (*plugins.polynomial.Polynomial* method), 501
- evalAtoms() (*plugins.polynomial.Polynomial* method), 501
- evalAtoms1() (*plugins.polynomial.Polynomial* method), 500
- evaluate() (in module *plugins.isopar*), 481
- execSource() (in module *utils*), 330
- exists() (*path.Path* method), 565
- exit() (in module *script*), 214
- exitGui() (in module *gui.draw*), 216
- expanduser() (*path.Path* method), 566

- exponents () (in module *plugins.isopar*), 481
 - export () (in module *script*), 212
 - Export () (*plugins.fe_post.FeResult* method), 474
 - export () (*plugins.webgl.WebGL* method), 518
 - export2 () (in module *script*), 212
 - exportDXF () (in module *plugins.dxf*), 450
 - exportDxf () (in module *plugins.dxf*), 450
 - exportDxfText () (in module *plugins.dxf*), 451
 - exportMesh () (in module *plugins.fe_abq*), 473
 - exportObjects () (in module *plugins.tools*), 513
 - exportScene () (*plugins.webgl.WebGL* method), 518
 - exportWebGL () (in module *gui.draw*), 222
 - extend () (*plugins.curve.BezierSpline* method), 444
 - extend () (*track.TrackedList* method), 578
 - extendedSectionChar () (in module *plugins.section2d*), 508
 - externalAngles () (*plugins.polygon.Polygon* method), 499
 - extractCanvasSettings () (in module *opengl.canvas*), 533
 - extractMeshes () (in module *fileread*), 386
 - extrude () (*elements.Elems* method), 321
 - extrude () (*formex.Formex* method), 148
 - extrude () (*mesh.Mesh* method), 257
 - eye (*opengl.camera.Camera* attribute), 521
 - eyeToClip () (*opengl.camera.Camera* method), 523
- ## F
- faceDistance () (in module *geomtools*), 375
 - faceMesh () (*mesh.Mesh* method), 244
 - faces (*opengl.drawable.Actor* attribute), 537
 - family () (*elements.ElementType* method), 316
 - fcoords (*opengl.drawable.Actor* attribute), 537
 - fd () (in module *plugins.turtle*), 514
 - featureEdges () (*trisurface.TriSurface* method), 269
 - FEModel (class in *plugins.fe*), 453
 - FeResult (class in *plugins.fe_post*), 474
 - fforward () (in module *gui.draw*), 219
 - fgcolor () (in module *gui.draw*), 231
 - Field (class in *field*), 322
 - field (module), 322
 - fieldReport () (*geometry.Geometry* method), 293
 - fields (*geometry.Geometry* attribute), 283, 292
 - fields (*mesh.Mesh* attribute), 237
 - File (class in *utils*), 325
 - fileDescription () (in module *utils*), 330
 - FileDialog (class in *gui.widgets*), 404
 - fileExtensions () (in module *utils*), 332
 - fileExtensionsFromFilter () (in module *utils*), 332
 - fileName () (*gui.appMenu.AppMenu* method), 426
 - fileread (module), 385
 - files () (*path.Path* method), 570
 - files () (*plugins.imagearray.DicomStack* method), 477
 - filetype () (*path.Path* method), 572
 - fileTypes () (in module *utils*), 333
 - fileUrls () (in module *gui.widgets*), 409
 - filewrite (module), 386
 - fill () (*plugins.polygon.Polygon* method), 499
 - fillBorder () (in module *trisurface*), 277
 - fillBorder () (*trisurface.TriSurface* method), 267
 - filterFiles () (*gui.appMenu.AppMenu* method), 426
 - filterWarning () (in module *utils*), 328
 - find3ds () (in module *plugins.web*), 516
 - find_class () (in module *project*), 362
 - find_class () (*project.Unpickler* method), 360
 - findAll () (in module *arraytools*), 198
 - findBisectrixUsingPlanes () (in module *plugins.bifmesh*), 429
 - findDuplicate () (*connectivity.Connectivity* method), 297
 - findEqualRows () (in module *arraytools*), 193
 - findFirst () (in module *arraytools*), 197
 - findIcon () (in module *utils*), 333
 - FindListItem () (in module *plugins.properties*), 506
 - finish_draw () (*gui.viewport.QtCanvas* method), 419
 - finish_drawing () (*gui.viewport.QtCanvas* method), 419
 - finish_selection () (*gui.viewport.QtCanvas* method), 417
 - firstWord () (in module *flatkeydb*), 576
 - fixNormals () (*trisurface.TriSurface* method), 271
 - fixVolumes () (*mesh.Mesh* method), 260
 - flags () (*gui.widgets.ArrayModel* method), 404
 - flags () (*gui.widgets.TableModel* method), 403
 - flare () (*coords.Coords* method), 104
 - flare () (*geometry.Geometry* method), 287
 - FlatDB (class in *flatkeydb*), 574
 - flatkeydb (module), 573
 - flatten () (in module *gui.draw*), 223
 - flatten () (in module *olist*), 560
 - Float (in module *arraytools*), 157
 - flyAlong () (in module *gui.draw*), 220
 - fmt () (*plugins.fe_abq.Output* method), 455
 - fmtAnalyticalSurface () (in module *plugins.fe_abq*), 467
 - fmtBeamSection () (in module *plugins.fe_abq*), 463
 - fmtBoundary () (in module *plugins.fe_abq*), 470
 - fmtComment () (in module *plugins.fe_abq*), 458
 - fmtConnectorElasticity () (in module *plugins.fe_abq*), 461
 - fmtConnectorSection () (in module *plugins.fe_abq*), 465

- fmtConnectorStop() (in module *plugins.fe_abq*), 461
 fmtConstraint() (in module *plugins.fe_abq*), 469
 fmtContact() (in module *plugins.fe_abq*), 467
 fmtContactPair() (in module *plugins.fe_abq*), 468
 fmtDashpotSection() (in module *plugins.fe_abq*), 465
 fmtData() (in module *plugins.fe_abq*), 457
 fmtDataId() (in module *arraytools*), 211
 fmtData2d() (in module *plugins.fe_abq*), 457
 fmtEquation() (in module *plugins.fe_abq*), 469
 fmtFrameSection() (in module *plugins.fe_abq*), 464
 fmtGeneralBeamSection() (in module *plugins.fe_abq*), 463
 fmtHeading() (in module *plugins.fe_abq*), 458
 fmtInertia() (in module *plugins.fe_abq*), 470
 fmtInertiaSection() (in module *plugins.fe_abq*), 465
 fmtInitialConditions() (in module *plugins.fe_abq*), 469
 fmtKeyword() (in module *plugins.fe_abq*), 457
 fmtLoad() (in module *plugins.fe_abq*), 472
 fmtMassSection() (in module *plugins.fe_abq*), 465
 fmtMaterial() (in module *plugins.fe_abq*), 459
 fmtMembraneSection() (in module *plugins.fe_abq*), 462
 fmtOption() (in module *plugins.fe_abq*), 458
 fmtOptions() (in module *plugins.fe_abq*), 458
 fmtOrientation() (in module *plugins.fe_abq*), 466
 fmtPart() (in module *plugins.fe_abq*), 459
 fmtRigidSection() (in module *plugins.fe_abq*), 465
 fmtSectionHeading() (in module *plugins.fe_abq*), 459
 fmtShellSection() (in module *plugins.fe_abq*), 462
 fmtSolid2dSection() (in module *plugins.fe_abq*), 461
 fmtSolid3dSection() (in module *plugins.fe_abq*), 462
 fmtSolidSection() (in module *plugins.fe_abq*), 461
 fmtSpringOrDashpot() (in module *plugins.fe_abq*), 465
 fmtSpringSection() (in module *plugins.fe_abq*), 465
 fmtSurface() (in module *plugins.fe_abq*), 466
 fmtSurfaceInteraction() (in module *plugins.fe_abq*), 467
 fmtSurfaceSection() (in module *plugins.fe_abq*), 462
 fmtTransform() (in module *plugins.fe_abq*), 466
 fmtTrussSection() (in module *plugins.fe_abq*), 464
 fmtWatermark() (in module *plugins.fe_abq*), 458
 focus (*opengl.camera.Camera* attribute), 521
 focus() (in module *gui.draw*), 220
 FontTexture (class in *opengl.texttext*), 546
 forceReST() (in module *utils*), 336
 forget() (in module *script*), 212
 forget() (*plugins.objects.Objects* method), 496
 forgetAll() (in module *script*), 212
 format_actor() (*plugins.webgl.WebGL* method), 518
 format_gui() (*plugins.webgl.WebGL* method), 518
 format_gui_controller() (*plugins.webgl.WebGL* method), 518
 formatDict() (in module *config*), 557
 formatDict() (in module *mydict*), 552
 formatDict() (in module *software*), 391
 Formex, 13
 Formex (class in *formex*), 127
 formex (module), 127
 fprintf() (*coords.Coords* method), 77
 framedText() (in module *utils*), 337
 frameScale() (in module *plugins.postproc*), 501
 frenet() (in module *plugins.nurbs*), 495
 frenet() (*plugins.curve.Curve* method), 436
 frenet() (*plugins.nurbs.NurbsCurve* method), 489
 fromCS() (*coords.Coords* method), 95
 fromCS() (*geometry.Geometry* method), 286
 fromfile() (*coords.Coords* class method), 119
 fromfile() (*formex.Formex* class method), 154
 fromkeys() (*track.TrackedDict* method), 578
 fromstring() (*coords.Coords* class method), 118
 fromstring() (*formex.Formex* class method), 153
 fromWindow() (*opengl.camera.Camera* method), 524
 front() (*adjacency.Adjacency* method), 354
 front() (*connectivity.Connectivity* method), 304
 front() (in module *opengl.renderer*), 542
 frontGenerator() (*adjacency.Adjacency* method), 353
 frontGenerator() (*connectivity.Connectivity* method), 303
 frontWalk() (*adjacency.Adjacency* method), 353
 frontWalk() (*connectivity.Connectivity* method), 304
 frontWalk() (*mesh.Mesh* method), 248
 ftype (*path.Path* attribute), 565
 ftype_compr() (*path.Path* method), 572
 fullAppName() (*gui.appMenu.AppMenu* method), 426
 fullElems() (*opengl.drawable.Actor* method), 538
 fuse() (*coords.Coords* method), 114
 fuse() (*mesh.Mesh* method), 250

G

generateFromFont() (*opengl.texttext.FontTexture*

- method*), 547
- GenericDialog (*class in gui.widgets*), 406
- genKnotVector () (*in module plugins.nurbs*), 493
- GeomActor (*in module opengl.drawable*), 539
- Geometry (*class in geometry*), 282
- geometry (*module*), 282
- Geometry4 (*class in plugins.nurbs*), 486
- GeometryFile (*class in geomfile*), 362
- GeometryFileDialog (*class in gui.widgets*), 405
- geomfile (*module*), 362
- geomtools (*module*), 367
- get () (*collection.Collection method*), 558
- get () (*elements.ElementType static method*), 317
- Get () (*plugins.units.UnitsSystem method*), 515
- get () (*track.TrackedDict method*), 578
- getBorder () (*mesh.Mesh method*), 245
- getBorderElems () (*mesh.Mesh method*), 246
- getBorderMesh () (*mesh.Mesh method*), 246
- getBorderNodes () (*mesh.Mesh method*), 246
- getCells () (*elements.ElementType method*), 316
- getCells () (*mesh.Mesh method*), 243
- getcfg () (*in module script*), 212
- getCollection () (*in module plugins.tools*), 513
- getColor () (*in module gui.widgets*), 410
- getDocString () (*in module utils*), 338
- getDrawEdges () (*elements.ElementType method*), 316
- getDrawFaces () (*elements.ElementType method*), 316
- getEdges () (*elements.ElementType method*), 316
- getEdges () (*mesh.Mesh method*), 243
- getElemEdges () (*mesh.Mesh method*), 244
- getElemEdges () (*trisurface.TriSurface method*), 264
- getElement () (*elements.ElementType method*), 316
- getElems () (*mesh.Mesh method*), 243
- getElems () (*plugins.fe.Model method*), 452
- getEntities () (*elements.ElementType method*), 315
- getFaces () (*elements.ElementType method*), 316
- getFaces () (*mesh.Mesh method*), 243
- getField () (*geometry.Geometry method*), 293
- getFilename () (*gui.widgets.FileDialog method*), 405
- getFiles () (*gui.appMenu.AppMenu method*), 426
- getFreeEdgesMesh () (*mesh.Mesh method*), 245
- getFreeEntities () (*mesh.Mesh method*), 244
- getFreeEntitiesMesh () (*mesh.Mesh method*), 245
- getIncs () (*plugins.fe_post.FeResult method*), 475
- getInts () (*in module plugins.tetgen*), 510
- getLowerEntities () (*mesh.Mesh method*), 242
- getMouseFunc () (*gui.viewport.CanvasMouseHandler method*), 415
- getMouseFunc () (*gui.viewport.QtCanvas method*), 417
- getNodes () (*mesh.Mesh method*), 243
- getObjectItems () (*in module plugins.tools*), 513
- getParams () (*in module fileread*), 385
- getPartition () (*in module plugins.tools*), 513
- getPoints () (*elements.ElementType method*), 316
- getPoints () (*mesh.Mesh method*), 243
- getProp () (*plugins.properties.PropertyDB method*), 504
- getRectangle () (*in module gui.draw*), 220
- getres () (*plugins.fe_post.FeResult method*), 475
- getResults () (*gui.widgets.FileDialog method*), 405
- getResults () (*gui.widgets.GeometryFileDialog method*), 405
- getResults () (*gui.widgets.InputDialog method*), 402
- getResults () (*gui.widgets.ListSelection method*), 406
- getResults () (*gui.widgets.MessageBox method*), 407
- getResults () (*gui.widgets.ProjectSelection method*), 406
- getResults () (*gui.widgets.SaveImageDialog method*), 406
- getSize () (*gui.viewport.QtCanvas method*), 415
- getSteps () (*plugins.fe_post.FeResult method*), 475
- getValues () (*gui.widgets.CoordsBox method*), 408
- gl () (*opengl.matrix.Matrix4 method*), 540
- gl_depth () (*in module opengl.camera*), 526
- gl_loadmodelview () (*in module opengl.camera*), 525
- gl_loadprojection () (*in module opengl.camera*), 525
- gl_modelview () (*in module opengl.camera*), 525
- gl_pickbuffer () (*in module opengl.canvas*), 532
- gl_projection () (*in module opengl.camera*), 525
- gl_viewport () (*in module opengl.camera*), 525
- GLcolor () (*in module opengl.colors*), 233
- GLcolorA () (*in module opengl.colors*), 233
- glEnable () (*in module opengl.canvas*), 533
- glFlat () (*in module opengl.canvas*), 533
- glinit () (*opengl.canvas.Canvas method*), 530
- glLineStipple () (*in module opengl.canvas*), 532
- glob () (*path.Path method*), 570
- glob () (*utils.NameSequence method*), 328
- globalInterpolationCurve () (*in module plugins.nurbs*), 493
- Globals () (*in module script*), 212
- globFiles () (*in module utils*), 339
- glSmooth () (*in module opengl.canvas*), 533
- glupdate () (*opengl.canvas.Canvas method*), 530
- glViewport () (*in module opengl.canvas*), 533
- go () (*in module plugins.turtle*), 514
- golden_ratio (*in module arraytools*), 158
- gray2qimage () (*in module plugins.imagearray*), 479
- grepSource () (*in module utils*), 334
- GREY () (*in module opengl.colors*), 235

Grid() (in module *opengl.decor*), 535
 Grid2D (class in *opengl.decor*), 534
 gridpoints() (in module *arraytools*), 209
 group() (in module *olist*), 560
 group() (*path.Path* method), 570
 groupArgmin() (in module *arraytools*), 198
 groupInputItem() (in module *gui.widgets*), 409
 growAxis() (in module *arraytools*), 163
 growCollection() (in module *plugins.tools*), 513
 growSelection() (*mesh.Mesh* method), 249
 gts_refine() (*trisurface.TriSurface* method), 273
 gts_smooth() (*trisurface.TriSurface* method), 273
 gtsset() (*trisurface.TriSurface* method), 275
 gui.appMenu (module), 425
 gui.colorscale (module), 413
 gui.draw (module), 216
 gui.image (module), 422
 gui.imageViewer (module), 425
 gui.menu (module), 410
 gui.toolbar (module), 428
 gui.viewport (module), 415
 gui.widgets (module), 393
 gunzip() (in module *utils*), 338
 gzip() (in module *utils*), 337

H

hasAnnotation() (*plugins.objects.DrawableObjects* method), 497
 hasExternal() (in module *software*), 391
 hasGrid() (*opengl.canvas.Canvas* method), 530
 hasMatch() (*coords.Coords* method), 116
 hasModule() (in module *software*), 390
 hasTriade() (*opengl.canvas.Canvas* method), 530
 hboxInputItem() (in module *gui.widgets*), 409
 header_data() (*project.Project* method), 361
 headerData() (*gui.widgets.ArrayModel* method), 404
 headerData() (*gui.widgets.TableModel* method), 403
 hex8_els() (in module *mesh*), 262
 hex8_wts() (in module *mesh*), 262
 hexVolume() (in module *geomtools*), 376
 hicolor() (in module *gui.draw*), 231
 highlightActor() (in module *gui.draw*), 221
 highlightActors() (*opengl.canvas.Canvas* method), 532
 highlighted() (*opengl.drawable.Actor* method), 538
 highlighted() (*opengl.scene.Scene* method), 545
 histogram2() (in module *arraytools*), 203
 hits() (*connectivity.Connectivity* method), 301
 hits() (*mesh.Mesh* method), 252
 home() (*path.Path* class method), 573
 horner() (in module *arraytools*), 186
 hsorted() (in module *utils*), 338
 human() (*plugins.polynomial.Polynomial* method), 501
 humanSize() (in module *utils*), 335

hyperCylindrical() (*coords.Coords* method), 98
 hyperCylindrical() (*geometry.Geometry* method), 287

I

identity() (*opengl.matrix.Matrix4* method), 540
 idraw() (*gui.viewport.QtCanvas* method), 418
 ignore_error() (in module *flatkeydb*), 576
 image() (*gui.viewport.QtCanvas* method), 416
 image() (*plugins.imagearray.DicomStack* method), 477
 image2array() (in module *plugins.imagearray*), 477
 imageFormatFromExt() (in module *gui.image*), 423
 imageFormats() (in module *gui.image*), 422
 ImageView (class in *gui.widgets*), 408
 ImageViewer (class in *gui.imageViewer*), 425
 importDXF() (in module *plugins.dxf*), 449
 Increment() (*plugins.fe_post.FeResult* method), 474
 index, **609**
 index() (*gui.menu.BaseMenu* method), 411
 index() (*plugins.nurbs.KnotVector* method), 486
 index() (*track.TrackedList* method), 578
 index() (*varray.Varray* method), 347
 index1d() (*varray.Varray* method), 348
 Inertia (class in *inertia*), 382
 inertia (module), 380
 inertia() (*coords.Coords* method), 83
 inertia() (*geometry.Geometry* method), 285
 inertia() (in module *inertia*), 385
 inertia() (*trisurface.TriSurface* method), 266
 inertialDirections() (in module *geomtools*), 377
 info() (*formex.Formex* method), 136
 info() (*geometry.Geometry* method), 284
 info() (*mesh.Mesh* method), 240
 initialize() (in module *gui.image*), 422
 inputAny() (in module *gui.widgets*), 409
 InputBool (class in *gui.widgets*), 395
 InputButton (class in *gui.widgets*), 399
 InputColor (class in *gui.widgets*), 399
 InputCombo (class in *gui.widgets*), 396
 InputDialog (class in *gui.widgets*), 400
 InputFile (class in *gui.widgets*), 399
 InputFilename (class in *gui.widgets*), 399
 InputFloat (class in *gui.widgets*), 397
 InputFont (class in *gui.widgets*), 399
 InputForm (class in *gui.widgets*), 400
 InputFSlider (class in *gui.widgets*), 398
 InputGroup (class in *gui.widgets*), 400
 InputHBox (class in *gui.widgets*), 400
 InputInfo (class in *gui.widgets*), 394
 InputInteger (class in *gui.widgets*), 397
 InputItem (class in *gui.widgets*), 393
 InputIVector (class in *gui.widgets*), 398

- InputLabel (*class in gui.widgets*), 394
- InputList (*class in gui.widgets*), 395
- InputPoint (*class in gui.widgets*), 398
- InputPush (*class in gui.widgets*), 396
- InputRadio (*class in gui.widgets*), 396
- InputSlider (*class in gui.widgets*), 398
- InputString (*class in gui.widgets*), 394
- InputTab (*class in gui.widgets*), 400
- InputTable (*class in gui.widgets*), 397
- InputText (*class in gui.widgets*), 395
- InputWidget (*class in gui.widgets*), 400
- insert () (*flatkeydb.FlatDB method*), 575
- insert () (*track.TrackedList method*), 578
- insert_action () (*gui.menu.BaseMenu method*), 411
- insert_menu () (*gui.menu.BaseMenu method*), 411
- insert_sep () (*gui.menu.BaseMenu method*), 411
- insertItems () (*gui.menu.BaseMenu method*), 411
- insertKnots () (*plugins.nurbs.NurbsCurve method*), 489
- insertLevel () (*connectivity.Connectivity method*), 305
- insertLevel () (*elements.Elems method*), 319
- insertPointsAt () (*plugins.curve.BezierSpline method*), 444
- insertPointsAt () (*plugins.curve.PolyLine method*), 441
- insertRows () (*gui.widgets.TableModel method*), 403
- inside () (*in module arraytools*), 178
- inside () (*in module plugins.pyformex_gts*), 507
- inside () (*opengl.camera.Camera method*), 525
- inside () (*opengl.drawable.Actor method*), 539
- inside () (*trisurface.TriSurface method*), 273
- insideSimplex () (*in module geomtools*), 380
- insideTriangle () (*in module geomtools*), 380
- Int (*in module arraytools*), 157
- interleave () (*in module arraytools*), 164
- internalAngles () (*plugins.polygon.Polygon method*), 499
- International () (*plugins.units.UnitsSystem method*), 515
- interpolate () (*coords.Coords method*), 119
- interpolate () (*formex.Formex method*), 149
- interpolate () (*in module formex*), 155
- interpoly () (*in module plugins.isopar*), 481
- interrogate () (*in module utils*), 343
- intersection () (*in module olist*), 559
- intersection () (*in module plugins.pyformex_gts*), 507
- intersection () (*trisurface.TriSurface method*), 275
- intersectionLinesPWP () (*in module geomtools*), 372
- intersectionPointsLWT () (*in module geomtools*), 371
- intersectionPointsPWP () (*in module geomtools*), 372
- intersectionPointsSWP () (*in module geomtools*), 371
- intersectionPointsSWT () (*in module geomtools*), 372
- intersectionSphereSphere () (*in module geomtools*), 372
- intersectionSWP () (*in module geomtools*), 370
- intersectionTimesLWT () (*in module geomtools*), 371
- intersectionTimesSWP () (*in module geomtools*), 370
- intersectionTimesSWT () (*in module geomtools*), 371
- intersectionWithLines () (*mesh.Mesh method*), 259
- intersectionWithPlane () (*formex.Formex method*), 151
- intersectionWithPlane () (*trisurface.TriSurface method*), 270
- intersectLineWithLine () (*in module geomtools*), 368
- intersectLineWithPlane () (*in module geomtools*), 369
- inverse () (*connectivity.Connectivity method*), 300
- inverse () (*opengl.matrix.Matrix4 method*), 541
- inverse () (*varray.Varray method*), 349
- inverseDict () (*in module utils*), 342
- inverseIndex () (*in module arraytools*), 196
- inverseIndex () (*in module varray*), 349
- inverseUniqueIndex () (*in module arraytools*), 188
- invtransform () (*opengl.matrix.Matrix4 method*), 541
- is_absolute () (*path.Path method*), 565
- is_badlink () (*path.Path method*), 565
- is_dir () (*path.Path method*), 565
- is_file () (*path.Path method*), 565
- is_pyFormex () (*in module utils*), 338
- is_script () (*in module utils*), 338
- is_symlink () (*path.Path method*), 565
- is_valid_mono_font () (*in module utils*), 342
- isBlended () (*plugins.nurbs.NurbsCurve method*), 488
- isClamped () (*plugins.nurbs.NurbsCurve method*), 488
- isClosedManifold () (*trisurface.TriSurface method*), 267
- isConvex () (*plugins.polygon.Polygon method*), 499
- isConvexManifold () (*trisurface.TriSurface method*), 267
- isFloat () (*in module arraytools*), 158
- isInt () (*in module arraytools*), 158

- isManifold() (*trisurface.TriSurface* method), 266
 isNum() (*in module arraytools*), 158
 isoline() (*in module plugins.isosurface*), 482
 Isopar (*class in plugins.isopar*), 480
 isopar() (*coords.Coords* method), 111
 isopar() (*geometry.Geometry* method), 288
 isosurface() (*in module plugins.isosurface*), 482
 isqrt() (*in module arraytools*), 174
 isRational() (*plugins.nurbs.NurbsCurve* method), 488
 isroll() (*in module arraytools*), 192
 isUniform() (*plugins.nurbs.NurbsCurve* method), 488
 item() (*gui.menu.BaseMenu* method), 411
 ItemList (*class in opengl.scene*), 544
 items() (*collection.Collection* method), 558
 items() (*track.TrackedDict* method), 578
- ## J
- joinpath() (*path.Path* method), 567
- ## K
- keep() (*plugins.objects.Objects* method), 496
 key_error_handler() (*flatkeydb.FlatDB* method), 575
 keys() (*collection.Collection* method), 558
 keys() (*config.Config* method), 557
 keys() (*track.TrackedDict* method), 578
 killProcesses() (*in module utils*), 330
 knotPoints() (*plugins.nurbs.NurbsCurve* method), 489
 knots (*plugins.nurbs.NurbsCurve* attribute), 487
 KnotVector (*class in plugins.nurbs*), 486
- ## L
- largestByConnection() (*mesh.Mesh* method), 249
 largestDirection() (*in module geomtools*), 377
 lastCommandReport() (*in module utils*), 330
 layer() (*plugins.dxf.DxfExporter* method), 449
 layout() (*built-in function*), 47
 layout() (*in module gui.draw*), 221
 length() (*in module arraytools*), 175
 length() (*in module gui.viewport*), 421
 length() (*mesh.Mesh* method), 260
 length() (*plugins.curve.Curve* method), 434
 length() (*varray.Varray* method), 346
 length_intgrnd() (*plugins.curve.BezierSpline* method), 443
 lengths (*varray.Varray* attribute), 346
 lengths() (*formex.Formex* method), 152
 lengths() (*mesh.Mesh* method), 259
 lengths() (*plugins.curve.BezierSpline* method), 443
 lengths() (*plugins.curve.PolyLine* method), 440
 level, 609
 level() (*formex.Formex* method), 133
 level() (*geometry.Geometry* method), 284
 level() (*mesh.Mesh* method), 240
 levelSelector() (*elements.Elems* method), 319
 levelVolumes() (*in module geomtools*), 377
 levelVolumes() (*mesh.Mesh* method), 259
 Light (*class in opengl.canvas*), 526
 LightProfile (*class in opengl.canvas*), 526
 lights() (*in module gui.draw*), 232
 Lima (*class in plugins.lima*), 482
 lima() (*in module plugins.lima*), 483
 Line (*class in opengl.decor*), 533
 Line (*class in plugins.curve*), 442
 line() (*in module simple*), 356
 line() (*plugins.dxf.DxfExporter* method), 449
 line2_wts() (*in module mesh*), 261
 lineIntersection() (*in module geomtools*), 379
 Lines (*class in opengl.decor*), 533
 linestipple() (*in module gui.draw*), 232
 linewidth() (*in module gui.draw*), 232
 link() (*gui.viewport.MultiCanvas* method), 421
 linkViewport() (*built-in function*), 48
 linkViewport() (*in module gui.draw*), 221
 List (*class in olist*), 559
 list() (*elements.ElementType* class method), 316
 listall() (*elements.ElementType* class method), 317
 listAll() (*in module script*), 212
 listAll() (*plugins.objects.Objects* method), 496
 listAllFonts() (*in module utils*), 342
 listDegenerate() (*connectivity.Connectivity* method), 296
 listDuplicate() (*connectivity.Connectivity* method), 297
 listIconNames() (*in module utils*), 334
 listMonoFonts() (*in module utils*), 342
 listNonDegenerate() (*connectivity.Connectivity* method), 296
 ListSelection (*class in gui.widgets*), 406
 listTree() (*path.Path* method), 571
 listUnique() (*connectivity.Connectivity* method), 297
 ListWidget (*class in gui.widgets*), 402
 load() (*config.Config* method), 557
 load() (*gui.viewport.MultiCanvas* method), 421
 load() (*opengl.camera.Camera* method), 525
 load() (*project.Project* method), 362
 loadConfig() (*gui.viewport.MultiCanvas* method), 421
 loadConfig() (*opengl.camera.Camera* method), 525
 loadFiles() (*gui.appMenu.AppMenu* method), 426
 loadModelView() (*opengl.camera.Camera* method), 525

- loadModelview() (*opengl.camera.Camera method*), 524
 - loadProjection() (*opengl.camera.Camera method*), 523
 - loadUniforms() (*opengl.shader.Shader method*), 546
 - localParam() (*plugins.curve.Curve method*), 433
 - locations() (*opengl.shader.Shader method*), 546
 - lock() (*opengl.camera.Camera method*), 522
 - longestEdge() (*trisurface.TriSurface method*), 268
 - lookAt() (*opengl.camera.Camera method*), 523
 - lrange() (*in module olist*), 559
 - lsuffix (*path.Path attribute*), 564
 - luminance() (*in module opengl.colors*), 234
- ## M
- makeEditable() (*gui.widgets.ArrayModel method*), 403
 - makeEditable() (*gui.widgets.TableModel method*), 403
 - map() (*coords.Coords method*), 104
 - map() (*geometry.Geometry method*), 287
 - map1() (*coords.Coords method*), 105
 - map1() (*geometry.Geometry method*), 287
 - mapd() (*coords.Coords method*), 106
 - mapd() (*geometry.Geometry method*), 288
 - mapHexLong() (*in module plugins.bifmesh*), 430
 - mapping, 609
 - mapQuadLong() (*in module plugins.bifmesh*), 430
 - Mark (*class in opengl.texttext*), 548
 - maskedEdgeFrontWalk() (*mesh.Mesh method*), 248
 - match() (*coords.Coords method*), 115
 - match() (*flatkeydb.FlatDB method*), 576
 - matchAll() (*in module utils*), 330
 - matchAny() (*in module path*), 573
 - matchAny() (*in module utils*), 330
 - matchCentroids() (*mesh.Mesh method*), 251
 - matchCoords() (*mesh.Mesh method*), 250
 - matchCount() (*in module utils*), 330
 - matchMany() (*in module utils*), 330
 - matchNone() (*in module utils*), 330
 - MaterialDB (*class in plugins.properties*), 502
 - Matrix4 (*class in opengl.matrix*), 539
 - maxcon() (*adjacency.Adjacency method*), 351
 - maxnodes() (*connectivity.Connectivity method*), 295
 - maxProp() (*geometry.Geometry method*), 289
 - maxsize() (*coords.Coords method*), 82
 - maxWinSize() (*in module gui.widgets*), 408
 - meanNodes() (*mesh.Mesh method*), 252
 - memory_report() (*in module utils*), 343
 - Menu (*class in gui.menu*), 412
 - MenuBar (*class in gui.menu*), 412
 - mergedModel() (*in module plugins.fe*), 453
 - mergeMeshes() (*in module mesh*), 261
 - mergeNodes() (*in module mesh*), 260
 - Mesh, 609
 - Mesh (*class in mesh*), 235
 - mesh (*module*), 235
 - meshes() (*plugins.fe.Model method*), 452
 - MessageBox (*class in gui.widgets*), 407
 - minmax() (*in module arraytools*), 167
 - minroll() (*in module arraytools*), 192
 - mirror() (*formex.Formex method*), 143
 - mkdir() (*in module script*), 214
 - mkdir() (*path.Path method*), 568
 - mkpdir() (*in module script*), 215
 - Model (*class in plugins.fe*), 452
 - modelview (*opengl.camera.Camera attribute*), 521
 - moduleList() (*in module utils*), 334
 - monomial() (*in module plugins.polynomial*), 501
 - mouse_draw() (*gui.viewport.QtCanvas method*), 419
 - mouse_draw_line() (*gui.viewport.QtCanvas method*), 420
 - mouse_pick() (*gui.viewport.QtCanvas method*), 420
 - mouse_rect_pick_events() (*gui.viewport.QtCanvas method*), 418
 - mouse_rectangle() (*gui.viewport.QtCanvas method*), 417
 - mouseMoveEvent() (*gui.viewport.QtCanvas method*), 420
 - mousePressEvent() (*gui.viewport.QtCanvas method*), 420
 - mouseReleaseEvent() (*gui.viewport.QtCanvas method*), 420
 - move() (*opengl.camera.Camera method*), 522
 - move() (*path.Path method*), 569
 - movingAverage() (*in module arraytools*), 205
 - movingView() (*in module arraytools*), 204
 - mplex() (*plugins.fe.Model method*), 452
 - mtime() (*path.Path method*), 569
 - mult() (*plugins.nurbs.KnotVector method*), 486
 - multi (*module*), 388
 - MultiCanvas (*class in gui.viewport*), 420
 - multiplex() (*in module arraytools*), 164
 - multiplicity() (*in module arraytools*), 189
 - multitask() (*in module multi*), 389
 - multitask2() (*in module multi*), 390
 - multiWebGL() (*in module gui.draw*), 223
 - mv() (*in module plugins.turtle*), 514
 - mydict (*module*), 549
- ## N
- name (*path.Path attribute*), 564
 - name() (*elements.ElementType method*), 316
 - name() (*gui.widgets.InputItem method*), 394
 - named() (*in module script*), 212
 - names() (*gui.menu.ActionList method*), 412

- NameSequence (class in *utils*), 326
 NaturalSpline (class in *plugins.curve*), 446
 ncoords () (*coords.Coords* method), 78
 ncoords () (*formex.Formex* method), 132
 ncoords () (*mesh.Mesh* method), 240
 ncoords () (*plugins.nurbs.Coords4* method), 485
 nctrl () (*plugins.nurbs.NurbsCurve* method), 487
 ndarray, 13
 ndim (*opengl.drawable.Actor* attribute), 537
 ndim () (*formex.Formex* method), 132
 ndim () (*mesh.Mesh* method), 240
 nearestValue () (in module *arraytools*), 196
 nEdgeAdjacent () (*mesh.Mesh* method), 250
 nEdgeConnected () (*mesh.Mesh* method), 249
 nedges () (*elements.ElementType* method), 315
 nedges () (*mesh.Mesh* method), 240
 nedges () (*trisurface.TriSurface* method), 264
 nelems () (*adjacency.Adjacency* method), 351
 nelems () (*connectivity.Connectivity* method), 295
 nelems () (*formex.Formex* method), 131
 nelems () (*geometry.Geometry* method), 284
 nelems () (*mesh.Mesh* method), 240
 nelems () (*plugins.curve.Curve* method), 433
 nelems () (*plugins.curve.PolyLine* method), 439
 nelems () (*plugins.fe.FEModel* method), 453
 nelems () (*plugins.fe.Model* method), 452
 nelems () (*plugins.polygon.Polygon* method), 499
 newRecord () (*flatkeydb.FlatDB* method), 574
 newView () (*gui.viewport.MultiCanvas* method), 420
 next () (*utils.NameSequence* method), 327
 nextInc () (*plugins.fe_post.FEResult* method), 475
 nextitem () (*gui.menu.BaseMenu* method), 411
 nextStep () (*plugins.fe_post.FEResult* method), 475
 nfaces () (*elements.ElementType* method), 315
 nfaces () (*trisurface.TriSurface* method), 264
 nfiles () (*plugins.imagearray.DicomStack* method), 477
 ngroups () (*plugins.fe.Model* method), 452
 niceLogSize () (in module *arraytools*), 174
 niceNumber () (in module *arraytools*), 174
 nknots () (*plugins.nurbs.KnotVector* method), 486
 nknots () (*plugins.nurbs.NurbsCurve* method), 487
 nNodeAdjacent () (*mesh.Mesh* method), 250
 nNodeConnected () (*mesh.Mesh* method), 249
 nnodes () (*connectivity.Connectivity* method), 295
 nnodes () (*elements.ElementType* method), 315
 nnodes () (*mesh.Mesh* method), 240
 nnodes () (*plugins.fe.Model* method), 452
 nodalAvg () (in module *arraytools*), 211
 nodalSum () (in module *arraytools*), 210
 node, **609**
 nodeAdjacency () (*mesh.Mesh* method), 250
 nodeConnections () (*mesh.Mesh* method), 249
 nodeProp () (*plugins.properties.PropertyDB* method), 504
 nonManifoldEdgeNodes () (*mesh.Mesh* method), 250
 nonManifoldEdges () (*mesh.Mesh* method), 250
 nonManifoldEdges () (*trisurface.TriSurface* method), 266
 nonManifoldEdgesFaces () (*trisurface.TriSurface* method), 266
 nonManifoldNodes () (*mesh.Mesh* method), 250
 normalize () (*adjacency.Adjacency* method), 352
 normalize () (in module *arraytools*), 176
 normalize () (in module *opengl.camera*), 526
 normalize () (*plugins.nurbs.Coords4* method), 485
 nParents () (*connectivity.Connectivity* method), 300
 nplex () (*connectivity.Connectivity* method), 296
 nplex () (*elements.ElementType* method), 315
 nplex () (*formex.Formex* method), 131
 nplex () (*mesh.Mesh* method), 240
 npoints () (*coords.Coords* method), 78
 npoints () (*formex.Formex* method), 132
 npoints () (*mesh.Mesh* method), 240
 npoints () (*plugins.nurbs.Coords4* method), 485
 npoints () (*plugins.polygon.Polygon* method), 499
 nrows (*varray.Varray* attribute), 344, 346
 nsetName () (in module *plugins.fe_abq*), 457
 numpy2qimage () (in module *plugins.imagearray*), 479
 numsplit () (in module *utils*), 339
 NurbsCircle () (in module *plugins.nurbs*), 494
 NurbsCurve (class in *plugins.nurbs*), 487
 NurbsSurface (class in *plugins.nurbs*), 492
 nvertices () (*elements.ElementType* method), 315
 nViewports () (in module *gui.draw*), 221
- ## O
- o (*coordsys.CoordSys* attribute), 279, 280
 object_type () (*plugins.objects.Objects* method), 495
 objectDialog () (in module *opengl.objectdialog*), 542
 Objects (class in *plugins.objects*), 495
 ObjFile (class in *plugins.export*), 451
 objSize () (in module *gui.widgets*), 408
 odict () (*plugins.objects.Objects* method), 496
 offset () (*trisurface.TriSurface* method), 269
 okColor () (*opengl.drawable.Actor* method), 538
 oldReadBezierSpline () (*geomfile.GeometryFile* method), 366
 olist (module), 559
 onOff () (in module *opengl.canvas*), 533
 open () (*path.Path* method), 570
 open () (*plugins.curve.PolyLine* method), 438
 open () (*utils.File* method), 326

- opengl.camera (*module*), 519
 - opengl.canvas (*module*), 526
 - opengl.colors (*module*), 232
 - opengl.decor (*module*), 533
 - opengl.drawable (*module*), 535
 - opengl.matrix (*module*), 539
 - opengl.objectdialog (*module*), 542
 - opengl.renderer (*module*), 542
 - opengl.sanitize (*module*), 542
 - opengl.scene (*module*), 544
 - opengl.shader (*module*), 545
 - opengl.texttext (*module*), 546
 - opengl.texture (*module*), 548
 - OpenGLFormat () (*in module gui.viewport*), 422
 - OpenGLSupportedVersions () (*in module gui.viewport*), 422
 - OpenGLVersions () (*in module gui.viewport*), 422
 - order () (*plugins.nurbs.NurbsCurve method*), 487
 - origin (*coordsys.CoordSys attribute*), 280
 - origin () (*in module coords*), 124
 - origin () (*plugins.imagearray.DicomStack method*), 477
 - orthog () (*in module arraytools*), 178
 - orthogonal_matrix () (*in module opengl.camera*), 526
 - otherAxes () (*in module coords*), 123
 - out (*plugins.fe_abq.Command attribute*), 454
 - out () (*plugins.dxf.DxfExporter method*), 449
 - outline () (*gui.viewport.QtCanvas method*), 416
 - Output (*class in plugins.fe_abq*), 454
 - OutputRequest () (*plugins.fe_post.FeResult method*), 475
 - outside () (*trisurface.TriSurface method*), 274
 - overflow () (*gui.colorscale.ColorLegend method*), 414
 - overrideMode () (*opengl.canvas.Canvas method*), 530
 - owner () (*path.Path method*), 570
- P**
- pairs () (*adjacency.Adjacency method*), 352
 - pan () (*opengl.camera.Camera method*), 522
 - parallel () (*in module arraytools*), 177
 - parent (*path.Path attribute*), 564
 - parents (*path.Path attribute*), 564
 - parse () (*flatkeydb.FlatDB method*), 575
 - parseLine () (*flatkeydb.FlatDB method*), 575
 - part () (*plugins.curve.BezierSpline method*), 443
 - part () (*plugins.curve.Contour method*), 445
 - partition () (*in module plugins.partition*), 498
 - partitionByAngle () (*mesh.Mesh method*), 249
 - partitionByAngle () (*trisurface.TriSurface method*), 269
 - partitionByConnection () (*mesh.Mesh method*), 248
 - partitionCollection () (*in module plugins.tools*), 513
 - parts (*path.Path attribute*), 563
 - parts () (*plugins.curve.BezierSpline method*), 444
 - parts () (*plugins.curve.PolyLine method*), 441
 - Path (*class in path*), 561
 - path (*module*), 561
 - path_like, 609
 - pattern () (*in module coords*), 124
 - pause () (*in module gui.draw*), 220
 - peek () (*utils.NameSequence method*), 327
 - peel () (*mesh.Mesh method*), 246
 - percentile () (*in module arraytools*), 203
 - perimeters () (*trisurface.TriSurface method*), 268
 - perpendicularVector () (*in module geomtools*), 380
 - perspective (*opengl.camera.Camera attribute*), 521
 - perspective_matrix () (*in module opengl.camera*), 526
 - pick () (*gui.viewport.QtCanvas method*), 417
 - pick () (*in module gui.draw*), 221
 - pick () (*opengl.drawable.Drawable method*), 536
 - pick_actors () (*opengl.canvas.Canvas method*), 532
 - pick_edges () (*opengl.canvas.Canvas method*), 532
 - pick_elements () (*opengl.canvas.Canvas method*), 532
 - pick_faces () (*opengl.canvas.Canvas method*), 532
 - pick_matrix () (*in module opengl.camera*), 526
 - pick_numbers () (*opengl.canvas.Canvas method*), 532
 - pick_parts () (*opengl.canvas.Canvas method*), 532
 - pick_points () (*opengl.canvas.Canvas method*), 532
 - pickFocus () (*in module gui.draw*), 222
 - pickMatrix () (*opengl.camera.Camera method*), 523
 - pickNumbers () (*gui.viewport.QtCanvas method*), 418
 - pickNumbers () (*in module gui.draw*), 222
 - pickProps () (*in module gui.draw*), 222
 - pixar () (*plugins.imagearray.DicomStack method*), 477
 - PlaneSection (*class in plugins.section2d*), 508
 - play () (*in module gui.draw*), 219
 - play () (*in module plugins.turtle*), 514
 - playScript () (*in module script*), 213
 - plexitude, 609
 - plexitude, 13
 - plugins.bifmesh (*module*), 429
 - plugins.cameratools (*module*), 430
 - plugins.ccxdat (*module*), 431
 - plugins.ccxinp (*module*), 431
 - plugins.curve (*module*), 432
 - plugins.datareader (*module*), 448

- plugins.dxf (*module*), 448
- plugins.export (*module*), 451
- plugins.fe (*module*), 452
- plugins.fe_abq (*module*), 453
- plugins.fe_post (*module*), 474
- plugins.flavia (*module*), 475
- plugins.imagearray (*module*), 476
- plugins.isopar (*module*), 480
- plugins.isosurface (*module*), 482
- plugins.lima (*module*), 482
- plugins.neu_exp (*module*), 483
- plugins.nurbs (*module*), 484
- plugins.objects (*module*), 495
- plugins.partition (*module*), 498
- plugins.plot2d (*module*), 498
- plugins.polygon (*module*), 499
- plugins.polynomial (*module*), 500
- plugins.postproc (*module*), 501
- plugins.properties (*module*), 502
- plugins.pyformex_gts (*module*), 507
- plugins.section2d (*module*), 508
- plugins.sectionize (*module*), 509
- plugins.tetgen (*module*), 509
- plugins.tools (*module*), 512
- plugins.turtle (*module*), 513
- plugins.units (*module*), 514
- plugins.web (*module*), 516
- plugins.webgl (*module*), 516
- PlyFile (*class in plugins.export*), 451
- point () (*formex.Formex method*), 134
- point () (*in module simple*), 356
- point2str () (*formex.Formex class method*), 136
- point_inertia () (*in module inertia*), 383
- pointNearLine () (*in module geomtools*), 375
- points () (*coords.Coords method*), 77
- points () (*coordsys.CoordSys method*), 280
- points () (*geometry.Geometry method*), 284
- pointsAt () (*plugins.curve.Curve method*), 433
- pointsAt () (*plugins.nurbs.NurbsCurve method*), 488
- pointsAt () (*plugins.nurbs.NurbsSurface method*), 492
- pointsAtLines () (*in module geomtools*), 368
- pointsAtSegments () (*in module geomtools*), 368
- pointsize () (*in module gui.draw*), 232
- pointsOff () (*plugins.curve.BezierSpline method*), 443
- pointsOn () (*plugins.curve.BezierSpline method*), 443
- pointsOnBezierCurve () (*in module plugins.nurbs*), 494
- Polygon (*class in plugins.polygon*), 499
- polygon () (*in module simple*), 357
- polygonNormals () (*in module geomtools*), 378
- polygonSector () (*in module simple*), 357
- PolyLine (*class in plugins.curve*), 438
- polyline () (*plugins.dxf.DxfExporter method*), 449
- Polynomial (*class in plugins.polynomial*), 500
- polynomial () (*in module plugins.polynomial*), 501
- pop () (*in module plugins.turtle*), 514
- pop () (*track.TrackedDict method*), 578
- pop () (*track.TrackedList method*), 578
- popitem () (*track.TrackedDict method*), 578
- position () (*coords.Coords method*), 96
- position () (*geometry.Geometry method*), 287
- position () (*plugins.curve.Curve method*), 437
- positionCoordsObj () (*in module plugins.curve*), 448
- powers () (*in module arraytools*), 170
- Predefined () (*plugins.units.UnitsSystem method*), 515
- prefixDict () (*in module utils*), 340
- prefixText () (*in module utils*), 337
- prepare () (*in module plugins.partition*), 498
- prepare () (*opengl.drawable.Actor method*), 538
- prepareColor () (*opengl.drawable.Drawable method*), 536
- prepareSubelems () (*opengl.drawable.Drawable method*), 536
- prepareTexture () (*opengl.drawable.Drawable method*), 536
- prevInc () (*plugins.fe_post.FeResult method*), 475
- prevStep () (*plugins.fe_post.FeResult method*), 475
- principal () (*inertia.Tensor method*), 382
- principalCS () (*coords.Coords method*), 84
- principalCS () (*geometry.Geometry method*), 285
- principalSizes () (*coords.Coords method*), 84
- principalSizes () (*geometry.Geometry method*), 285
- princTensor2D () (*in module plugins.section2d*), 508
- print () (*plugins.properties.PropertyDB method*), 504
- printar () (*in module arraytools*), 169
- printbbox () (*plugins.objects.Objects method*), 496
- printMessage () (*in module gui.draw*), 219
- printSteps () (*plugins.fe_post.FeResult method*), 475
- printval () (*plugins.objects.Objects method*), 496
- processArgs () (*in module script*), 214
- Project (*class in project*), 360
- project (*module*), 359
- project () (*opengl.camera.Camera method*), 525
- project () (*opengl.canvas.Canvas method*), 531
- projected () (*in module plugins.polygon*), 500
- projectedArea () (*in module geomtools*), 378
- projection (*opengl.camera.Camera attribute*), 521
- projection () (*in module arraytools*), 177
- projection () (*in module gui.viewport*), 421
- projectionVOP () (*in module geomtools*), 380
- projectionVOV () (*in module geomtools*), 380
- projectName () (*in module utils*), 333

projectOnCylinder() (*coords.Coords method*), 109
 projectOnCylinder() (*geometry.Geometry method*), 288
 projectOnPlane() (*coords.Coords method*), 108
 projectOnPlane() (*geometry.Geometry method*), 288
 projectOnSphere() (*coords.Coords method*), 109
 projectOnSphere() (*geometry.Geometry method*), 288
 projectOnSurface() (*coords.Coords method*), 110
 projectPoint() (*plugins.nurbs.NurbsCurve method*), 491
 projectPointOnLine() (*in module geomtools*), 373
 projectPointOnLineTimes() (*in module geomtools*), 374
 projectPointOnPlane() (*in module geomtools*), 373
 projectPointOnPlaneTimes() (*in module geomtools*), 373
 ProjectSelection (*class in gui.widgets*), 406
 prop (*geometry.Geometry attribute*), 283
 prop (*mesh.Mesh attribute*), 237
 Prop() (*plugins.properties.PropertyDB method*), 504
 properties() (*in module plugins.webgl*), 519
 PropertyDB (*class in plugins.properties*), 503
 propSet() (*geometry.Geometry method*), 289
 pshape() (*coords.Coords method*), 77
 pspacing() (*plugins.imagearray.DicomStack method*), 477
 push() (*in module plugins.turtle*), 514
 pwdir() (*in module script*), 214
 pyf_etype() (*in module plugins.ccxinp*), 431
 pyformexIcon() (*in module gui.widgets*), 408

Q

qimage2glcolor() (*in module plugins.imagearray*), 479
 qimage2numpy() (*in module plugins.imagearray*), 478
 qimage_like, 609
 QtCanvas (*class in gui.viewport*), 415
 quad4_els() (*in module mesh*), 261
 quad4_wts() (*in module mesh*), 261
 quad9_els() (*in module mesh*), 261
 quad9_wts() (*in module mesh*), 261
 quadgrid() (*in module mesh*), 262
 quadraticCurve() (*in module simple*), 357
 quadrilateral() (*in module mesh*), 262
 quality() (*trisurface.TriSurface method*), 268
 quarterCircle() (*in module mesh*), 263
 quit() (*in module script*), 214

R

RAD (*in module arraytools*), 158
 randomNoise() (*in module arraytools*), 206
 randomPoints() (*in module simple*), 355
 re, 609
 reachableFrom() (*mesh.Mesh method*), 247
 read() (*config.Config method*), 557
 read() (*geometry.Geometry class method*), 293
 read() (*geomfile.GeometryFile method*), 365
 Read() (*plugins.units.UnitsSystem method*), 515
 read() (*timer.Timer method*), 577
 read() (*trisurface.TriSurface class method*), 264
 read_bytes() (*path.Path method*), 570
 read_gambit_neutral() (*in module fileread*), 386
 read_gts() (*in module fileread*), 386
 read_off() (*in module fileread*), 386
 read_stl() (*in module trisurface*), 277
 read_stl_bin() (*in module fileread*), 386
 read_text() (*path.Path method*), 570
 readArray() (*in module arraytools*), 170
 readAttrib() (*geomfile.GeometryFile method*), 365
 readBezierSpline() (*geomfile.GeometryFile method*), 366
 readCommand() (*in module plugins.ccxinp*), 431
 readCoords() (*in module plugins.flavia*), 476
 readData() (*in module plugins.datareader*), 448
 readDatabase() (*plugins.properties.Database method*), 502
 readDispl() (*in module plugins.ccxdat*), 431
 readDXF() (*in module plugins.dxf*), 450
 readEleFile() (*in module plugins.tetgen*), 509
 readElems() (*in module fileread*), 385
 readElems() (*in module plugins.flavia*), 476
 readElemsBlock() (*in module plugins.tetgen*), 510
 readEsets() (*in module fileread*), 385
 readFaceFile() (*in module plugins.tetgen*), 509
 readFacesBlock() (*in module plugins.tetgen*), 510
 readField() (*geomfile.GeometryFile method*), 365
 readFile() (*flatkeydb.FlatDB method*), 575
 readFlavia() (*in module plugins.flavia*), 476
 readFormex() (*geomfile.GeometryFile method*), 365
 readFromFile() (*plugins.objects.Objects method*), 497
 readGeometry() (*geomfile.GeometryFile method*), 365
 readGeomFile() (*in module script*), 215
 readHeader() (*geomfile.GeometryFile method*), 365
 readHeader() (*project.Project method*), 361
 readInpFile() (*in module fileread*), 386
 readInput() (*in module plugins.ccxinp*), 432
 readLegacy() (*geomfile.GeometryFile method*), 366
 readline() (*geomfile.GeometryFile method*), 363
 readMesh() (*geomfile.GeometryFile method*), 366
 readMesh() (*in module plugins.flavia*), 476

- readMeshFile() (in module *fileread*), 386
- readNeigh() (in module *plugins.tetgen*), 511
- readNodeFile() (in module *plugins.tetgen*), 509
- readNodes() (in module *fileread*), 385
- readNodesBlock() (in module *plugins.tetgen*), 510
- readNurbsCurve() (*geomfile.GeometryFile* method), 366
- readNurbsSurface() (*geomfile.GeometryFile* method), 366
- readPolyFile() (in module *plugins.tetgen*), 510
- readPolyLine() (*geomfile.GeometryFile* method), 366
- readResult() (in module *plugins.flavia*), 476
- readResults() (in module *plugins.ccxdat*), 431
- readResults() (in module *plugins.flavia*), 476
- readSmeshFacetsBlock() (in module *plugins.tetgen*), 511
- readSmeshFile() (in module *plugins.tetgen*), 510
- readSoftware() (in module *software*), 392
- readStress() (in module *plugins.ccxdat*), 431
- readSurface() (in module *plugins.tetgen*), 510
- readTetgen() (in module *plugins.tetgen*), 511
- record_error_handler() (*flatkeydb.FlatDB* method), 575
- rect() (in module *simple*), 356
- Rectangle (class in *opengl.decors*), 533
- rectangle() (in module *mesh*), 262
- rectangle() (in module *simple*), 356
- rectangleWithHole() (in module *mesh*), 262
- reduceDegenerate() (*elements.Elems* method), 320
- reduceDegree() (*plugins.nurbs.NurbsCurve* method), 491
- refine() (*plugins.curve.PolyLine* method), 442
- refine() (*trisurface.TriSurface* method), 271
- reflect() (*coords.Coords* method), 94
- reflect() (*geometry.Geometry* method), 286
- reflect() (*mesh.Mesh* method), 253
- refreshDict() (in module *utils*), 341
- register (*elements.ElementType* attribute), 315
- registerSoftware() (in module *software*), 392
- regularGrid() (in module *simple*), 355
- rejected() (*plugins.imagearray.DicomStack* method), 477
- relative_to() (*path.Path* method), 568
- relLengths() (*plugins.curve.PolyLine* method), 440
- reload() (*gui.appMenu.AppMenu* method), 427
- reloadMenu() (in module *gui.appMenu*), 428
- remember() (*plugins.objects.Objects* method), 496
- remove() (*collection.Collection* method), 558
- remove() (*formex.Formex* method), 140
- remove() (*gui.menu.ActionList* method), 412
- remove() (in module *olist*), 560
- remove() (*path.Path* method), 568
- remove() (*track.TrackedList* method), 579
- removeAll() (*gui.menu.ActionList* method), 412
- removeAllKnots() (*plugins.nurbs.NurbsCurve* method), 491
- removeAlpha() (in module *plugins.imagearray*), 480
- removeAnnotation() (*plugins.objects.DrawableObjects* method), 497
- removeAny() (*opengl.scene.Scene* method), 545
- removeButton() (in module *gui.toolbar*), 428
- removed() (*utils.DictDiff* method), 328
- removeDegenerate() (*connectivity.Connectivity* method), 296
- removeDegenerate() (*mesh.Mesh* method), 255
- removeDegenerate() (*trisurface.TriSurface* method), 268
- removeDict() (in module *utils*), 341
- removeDrawn() (*opengl.scene.Scene* method), 545
- removeDuplicate() (*connectivity.Connectivity* method), 298
- removeDuplicate() (*formex.Formex* method), 140
- removeDuplicate() (*mesh.Mesh* method), 255
- removeFlat() (*varray.Varray* method), 348
- removeGrid() (*opengl.canvas.Canvas* method), 530
- removeHighlight() (in module *gui.draw*), 221
- removeHighlight() (*opengl.canvas.Canvas* method), 532
- removeHighlight() (*opengl.drawable.Actor* method), 538
- removeHighlight() (*opengl.scene.Scene* method), 545
- removeHighlighted() (*opengl.scene.Scene* method), 545
- removeItem() (*gui.menu.BaseMenu* method), 411
- removeKnot() (*plugins.nurbs.NurbsCurve* method), 491
- RemoveListItem() (in module *plugins.properties*), 506
- removeNonManifold() (*trisurface.TriSurface* method), 267
- removeRows() (*gui.widgets.TableModel* method), 403
- removeTree() (*path.Path* method), 569
- removeTriade() (*opengl.canvas.Canvas* method), 529
- removeView() (*gui.viewport.MultiCanvas* method), 421
- removeViewport() (*built-in function*), 48
- removeViewport() (in module *gui.draw*), 221
- rename() (in module *script*), 212
- render() (*opengl.drawable.Actor* method), 539
- render() (*opengl.drawable.Drawable* method), 536
- renderMode() (in module *gui.draw*), 231
- renderModes() (in module *gui.draw*), 231
- renumber() (*connectivity.Connectivity* method), 299
- renumber() (*mesh.Mesh* method), 255
- renumber() (*plugins.fe.Model* method), 453

- renumberIndex () (in module arraytools), 188
- reopen () (geomfile.GeometryFile method), 363
- reopen () (utils.File method), 326
- reorder () (connectivity.Connectivity method), 298
- reorder () (mesh.Mesh method), 255
- reorderAxis () (in module arraytools), 163
- rep () (coords.Coords method), 122
- rep () (formex.Formex method), 145
- repeatValues () (in module arraytools), 165
- replay () (in module gui.draw), 219
- replic () (connectivity.Connectivity method), 307
- replic () (formex.Formex method), 146
- replic2 () (formex.Formex method), 147
- replicate () (coords.Coords method), 112
- replicate () (formex.Formex method), 144
- replicm () (formex.Formex method), 147
- repm () (formex.Formex method), 146
- report () (connectivity.Connectivity method), 296
- report () (mesh.Mesh method), 240
- report () (opengl.camera.Camera method), 522
- report () (timer.Timer method), 577
- requireExternal () (in module software), 391
- requireKnots () (plugins.nurbs.NurbsCurve method), 489
- requireModule () (in module software), 390
- reset () (in module gui.draw), 226
- reset () (in module plugins.turtle), 513
- reset () (opengl.canvas.Canvas method), 530
- reset () (opengl.canvas.CanvasSettings method), 527
- reset () (timer.Timer method), 577
- resetArea () (opengl.camera.Camera method), 522
- resetDefaults () (opengl.canvas.Canvas method), 528
- resetGUI () (in module gui.draw), 223
- resetLighting () (opengl.canvas.Canvas method), 528
- resetOptions () (opengl.canvas.Canvas method), 528
- resized () (geometry.Geometry method), 286
- resizeImage () (in module plugins.imagearray), 478
- resolve () (connectivity.Connectivity method), 307
- resolve () (path.Path method), 566
- Result (class in plugins.fe_abq), 455
- returnNone () (in module mydict), 552
- returnNone () (mydict.Dict static method), 551
- reverse () (coordsys.CoordSys method), 281
- reverse () (formex.Formex method), 143
- reverse () (mesh.Mesh method), 252
- reverse () (plugins.curve.BezierSpline method), 445
- reverse () (plugins.curve.PolyLine method), 440
- reverse () (plugins.nurbs.KnotVector method), 487
- reverse () (plugins.nurbs.NurbsCurve method), 492
- reverse () (plugins.polygon.Polygon method), 499
- reverse () (track.TrackedList method), 579
- reverseAxis () (in module arraytools), 164
- revolve () (mesh.Mesh method), 257
- rewrite () (geomfile.GeometryFile method), 366
- rgb () (gui.viewport.QtCanvas method), 416
- rgb2qimage () (in module plugins.imagearray), 479
- RGBA () (in module opengl.colors), 235
- RGBAcolor () (in module opengl.colors), 234
- RGBcolor () (in module opengl.colors), 234
- rmdir () (path.Path method), 568
- ro () (in module plugins.turtle), 514
- roll () (in module olist), 559
- roll () (plugins.curve.PolyLine method), 440
- rollAxes () (coords.Coords method), 108
- rollAxes () (geometry.Geometry method), 288
- rosette () (formex.Formex method), 147
- rot (coordsys.CoordSys attribute), 279, 280
- rot (opengl.camera.Camera attribute), 521
- rot () (coords.Coords method), 120
- rot () (geometry.Geometry method), 288
- rotate () (coords.Coords method), 92
- rotate () (coordsys.CoordSys method), 281
- rotate () (geometry.Geometry method), 286
- rotate () (inertia.Tensor method), 382
- rotate () (opengl.camera.Camera method), 524
- rotate () (opengl.matrix.Matrix4 method), 540
- rotationAngle () (in module geomtools), 379
- rotationMatrix () (in module arraytools), 179
- rotationMatrix3 () (in module arraytools), 180
- rotmat () (in module arraytools), 181
- rotMatrix () (in module arraytools), 183
- rotMatrix2 () (in module arraytools), 184
- row () (varray.Varray method), 346
- rowCount () (gui.widgets.ArrayModel method), 404
- rowCount () (gui.widgets.TableModel method), 403
- rowHeights () (gui.widgets.Table method), 404
- rowindex () (varray.Varray method), 347
- run () (gui.appMenu.AppMenu method), 427
- runAll () (gui.appMenu.AppMenu method), 427
- runAllNext () (gui.appMenu.AppMenu method), 427
- runAny () (in module script), 214
- runApp () (gui.appMenu.AppMenu method), 427
- runApp () (in module script), 213
- runCurrent () (gui.appMenu.AppMenu method), 427
- runNextApp () (gui.appMenu.AppMenu method), 427
- runRandom () (gui.appMenu.AppMenu method), 427
- runScript () (in module script), 213
- runTetgen () (in module plugins.tetgen), 511
- runtime () (in module script), 215

S

- samefile () (path.Path method), 567
- sameLength () (varray.Varray method), 348
- sane_bbox () (in module opengl.scene), 545
- saneColor () (in module opengl.sanitize), 543

- saneColorArray () (in module *opengl.sanitize*), 543
 saneColorSet () (in module *opengl.sanitize*), 543
 saneFloat () (in module *opengl.sanitize*), 542
 saneLineStipple () (in module *opengl.sanitize*), 542
 saneLineWidth () (in module *opengl.sanitize*), 542
 saneSettings () (in module *plugins.webgl*), 519
 saneSize () (*gui.viewport.QtCanvas* method), 415
 save () (*gui.viewport.MultiCanvas* method), 421
 save () (in module *gui.image*), 423
 save () (*opengl.camera.Camera* method), 525
 save () (*project.Project* method), 361
 save_canvas () (in module *gui.image*), 423
 save_window () (in module *gui.image*), 423
 save_window_rect () (in module *gui.image*), 423
 saveBuffer () (*opengl.canvas.Canvas* method), 532
 saveGreyImage () (in module *plugins.imagearray*), 480
 saveIcon () (in module *gui.image*), 424
 saveImage () (in module *gui.image*), 424
 SaveImageDialog (class in *gui.widgets*), 406
 saveMovie () (in module *gui.image*), 425
 saveNext () (in module *gui.image*), 424
 scale () (*coords.Coords* method), 89
 scale () (*geometry.Geometry* method), 286
 scale () (*gui.colorscale.ColorScale* method), 414
 scale () (*opengl.matrix.Matrix4* method), 541
 scandir () (*path.Path* method), 570
 Scene (class in *opengl.scene*), 544
 sceneBbox () (*opengl.canvas.Canvas* method), 528
 script (module), 212
 seconds () (*timer.Timer* method), 577
 section () (*plugins.dxf.DxfExporter* method), 449
 sectionChar () (in module *plugins.section2d*), 508
 SectionDB (class in *plugins.properties*), 502
 sectionize () (in module *plugins.sectionize*), 509
 sector () (in module *simple*), 358
 seed, 609
 seed () (in module *arraytools*), 208
 seed1 () (in module *arraytools*), 208
 segmentOrientation () (in module *geomtools*), 379
 select () (*geometry.Geometry* method), 290
 select () (in module *olist*), 560
 select () (*varray.Varray* method), 347
 selectDict () (in module *utils*), 341
 selectDictValues () (in module *utils*), 342
 selectFont () (in module *gui.widgets*), 409
 selectNodes () (*connectivity.Connectivity* method), 305
 selectNodes () (*elements.Elems* method), 319
 selectNodes () (*formex.Formex* method), 139
 selectNodes () (*mesh.Mesh* method), 252
 selectProp () (*geometry.Geometry* method), 291
 set () (*collection.Collection* method), 558
 set () (*coords.Coords* method), 89
 set () (*plugins.objects.Objects* method), 495
 set_bbox () (*opengl.scene.Scene* method), 544
 set_edit_mode () (in module *gui.draw*), 222
 set_material_value () (in module *gui.draw*), 232
 setAll () (*gui.widgets.InputList* method), 396
 setAlpha () (*opengl.drawable.Actor* method), 538
 setAmbient () (*opengl.canvas.Canvas* method), 528
 setAngles () (*opengl.camera.Camera* method), 521
 setArea () (*opengl.camera.Camera* method), 522
 setBackground () (*opengl.canvas.Canvas* method), 529
 setCamera () (*opengl.canvas.Canvas* method), 531
 setCamera () (*plugins.webgl.WebGL* method), 518
 setCellData () (*gui.widgets.TableModel* method), 403
 setChecked () (*gui.widgets.InputList* method), 396
 setChoices () (*gui.widgets.InputCombo* method), 396
 setClip () (*opengl.camera.Camera* method), 523
 setColor () (*opengl.drawable.BaseActor* method), 537
 setCoords () (*trisurface.TriSurface* method), 264
 setCurrent () (*gui.viewport.MultiCanvas* method), 421
 setCursorShape () (*gui.viewport.CursorShapeHandler* method), 415
 setCursorShape () (*gui.viewport.QtCanvas* method), 417
 setCursorShapeFromFunc () (*gui.viewport.CursorShapeHandler* method), 415
 setCursorShapeFromFunc () (*gui.viewport.QtCanvas* method), 417
 setCustomColors () (in module *gui.widgets*), 410
 setData () (*gui.widgets.ArrayModel* method), 404
 setData () (*gui.widgets.TableModel* method), 403
 setdefault () (*track.TrackedDict* method), 578
 setDefaults () (*opengl.canvas.Canvas* method), 530
 setDrawOptions () (in module *gui.draw*), 226
 setEdgesAndFaces () (*trisurface.TriSurface* method), 264
 setElems () (*trisurface.TriSurface* method), 264
 setEltype () (*mesh.Mesh* method), 238
 setFgColor () (*opengl.canvas.Canvas* method), 529
 setFilters () (*gui.widgets.FileDialog* method), 405
 setGrid () (in module *gui.draw*), 230
 setGrid () (*opengl.canvas.Canvas* method), 530
 setHighlight () (*opengl.drawable.Actor* method), 538
 setIcon () (*gui.widgets.ButtonBox* method), 408
 setIcon () (*gui.widgets.InputPush* method), 397
 setLens () (*opengl.camera.Camera* method), 522

- setLineStipple() (*opengl.canvas.Canvas method*), 529
- setLineStipple() (*opengl.drawable.BaseActor method*), 537
- setLineWidth() (*opengl.canvas.Canvas method*), 529
- setLineWidth() (*opengl.drawable.BaseActor method*), 537
- setMaterial() (*opengl.canvas.Canvas method*), 528
- setMaterialDB() (*in module plugins.properties*), 505
- setMaterialDB() (*plugins.properties.PropertyDB method*), 504
- setModelview() (*opengl.camera.Camera method*), 524
- setNone() (*gui.widgets.InputList method*), 396
- setNormals() (*mesh.Mesh method*), 239
- setOpenGLFormat() (*in module gui.viewport*), 422
- setOptions() (*opengl.canvas.Canvas method*), 528
- setPointSize() (*opengl.canvas.Canvas method*), 529
- setPrefs() (*in module script*), 214
- setPrintFunction() (*formex.Formex class method*), 138
- setProjection() (*opengl.camera.Camera method*), 523
- setProp() (*geometry.Geometry method*), 288
- setProp() (*plugins.curve.Curve method*), 438
- setProp() (*plugins.objects.DrawableObjects method*), 497
- setRenderMode() (*opengl.canvas.Canvas method*), 528
- setRow() (*varray.Varray method*), 347
- setSaneLocale() (*in module utils*), 336
- setSectionDB() (*in module plugins.properties*), 505
- setSectionDB() (*plugins.properties.PropertyDB method*), 504
- setSelected() (*gui.widgets.InputList method*), 396
- setShrink() (*in module gui.draw*), 226
- setSlColor() (*opengl.canvas.Canvas method*), 529
- setStepInc() (*plugins.fe_post.FeResult method*), 475
- setText() (*gui.widgets.ButtonBox method*), 407
- setText() (*gui.widgets.InputPush method*), 397
- setTexture() (*opengl.drawable.Actor method*), 538
- setTexture() (*opengl.drawable.BaseActor method*), 537
- settings (*opengl.camera.Camera attribute*), 521
- setToggle() (*opengl.canvas.Canvas method*), 528
- setTracking() (*opengl.camera.Camera method*), 523
- setTriade() (*in module gui.draw*), 230
- setTriade() (*opengl.canvas.Canvas method*), 529
- setValue() (*gui.widgets.InputBool method*), 395
- setValue() (*gui.widgets.InputColor method*), 399
- setValue() (*gui.widgets.InputCombo method*), 396
- setValue() (*gui.widgets.InputFile method*), 399
- setValue() (*gui.widgets.InputFloat method*), 397
- setValue() (*gui.widgets.InputGroup method*), 400
- setValue() (*gui.widgets.InputInteger method*), 397
- setValue() (*gui.widgets.InputItem method*), 394
- setValue() (*gui.widgets.InputIVector method*), 399
- setValue() (*gui.widgets.InputLabel method*), 394
- setValue() (*gui.widgets.InputList method*), 396
- setValue() (*gui.widgets.InputPoint method*), 398
- setValue() (*gui.widgets.InputPush method*), 397
- setValue() (*gui.widgets.InputRadio method*), 396
- setValue() (*gui.widgets.InputText method*), 395
- setValue() (*gui.widgets.InputWidget method*), 400
- setValue() (*gui.widgets.ListSelection method*), 406
- setValues() (*gui.widgets.CoordsBox method*), 408
- setView() (*in module gui.draw*), 230
- setWireMode() (*opengl.canvas.Canvas method*), 528
- Shader (*class in opengl.shader*), 545
- Shaders() (*in module software*), 391
- shallowCopy() (*mesh.Mesh method*), 240
- shape (*formex.Formex attribute*), 131
- shape (*varray.Varray attribute*), 344, 346
- shape() (*in module simple*), 355
- shape() (*mesh.Mesh method*), 240
- shape() (*trisurface.TriSurface method*), 264
- sharedNodes() (*connectivity.Connectivity method*), 307
- shear() (*coords.Coords method*), 93
- shear() (*geometry.Geometry method*), 286
- shortestEdge() (*trisurface.TriSurface method*), 268
- show() (*gui.widgets.InputDialog method*), 402
- show() (*gui.widgets.InputFloat method*), 397
- show() (*gui.widgets.InputInteger method*), 397
- show() (*gui.widgets.InputString method*), 395
- show() (*gui.widgets.InputText method*), 395
- show3d() (*in module plugins.web*), 516
- show3ds() (*in module plugins.web*), 516
- show3dzip() (*in module plugins.web*), 516
- showBuffer() (*opengl.canvas.Canvas method*), 532
- showCameraTool() (*in module plugins.cameratools*), 431
- showDoc() (*in module gui.draw*), 217
- showFile() (*in module gui.draw*), 217
- showHistogram() (*in module plugins.plot2d*), 499
- showHTML() (*in module gui.draw*), 223
- showImage() (*gui.widgets.ImageView method*), 408
- showInfo() (*in module gui.draw*), 216
- showLineDrawing() (*in module gui.draw*), 222
- showMessage() (*in module gui.draw*), 216
- showStepPlot() (*in module plugins.plot2d*), 499
- showText() (*in module gui.draw*), 216
- showURL() (*in module gui.draw*), 223

- showWidget() (*gui.viewport.MultiCanvas method*), 421
- shrink() (*formex.Formex method*), 142
- similarity() (*trisurface.TriSurface method*), 271
- simple (*module*), 355
- simpleInputItem() (*in module gui.widgets*), 409
- sind() (*in module arraytools*), 170
- sind() (*in module plugins.turtle*), 513
- size (*varray.Varray attribute*), 344, 346
- size() (*path.Path method*), 570
- sizes() (*coords.Coords method*), 81
- sizes() (*geometry.Geometry method*), 284
- skipComments() (*in module plugins.tetgen*), 510
- sleep() (*in module gui.draw*), 220
- slice() (*trisurface.TriSurface method*), 270
- slugify() (*in module utils*), 336
- smallestAltitude() (*trisurface.TriSurface method*), 268
- smallestDirection() (*in module geomtools*), 377
- smartSeed() (*in module arraytools*), 209
- smooth() (*mesh.Mesh method*), 258
- smooth() (*trisurface.TriSurface method*), 270
- smoothLaplaceHC() (*trisurface.TriSurface method*), 271
- smoothLowPass() (*trisurface.TriSurface method*), 271
- soft2config() (*in module software*), 392
- software (*module*), 390
- solveMany() (*in module arraytools*), 186
- sort() (*coords.Coords method*), 113
- sort() (*track.TrackedList method*), 579
- sort() (*varray.Varray method*), 348
- sortByColumns() (*in module arraytools*), 192
- sorted() (*varray.Varray method*), 348
- sortElemsByLoadedFace() (*in module plugins.fe*), 453
- sortRows() (*adjacency.Adjacency method*), 351
- sortSets() (*in module gui.appMenu*), 427
- sortSubsets() (*in module arraytools*), 190
- sourceFiles() (*in module utils*), 334
- spacing() (*plugins.imagearray.DicomStack method*), 477
- span() (*plugins.nurbs.KnotVector method*), 487
- sphere() (*in module simple*), 357
- sphere2() (*in module simple*), 358
- sphere3() (*in module simple*), 358
- spherical() (*coords.Coords method*), 99
- spherical() (*geometry.Geometry method*), 287
- Spiral (*class in plugins.curve*), 446
- splash image, 58
- split() (*coords.Coords method*), 112
- split() (*formex.Formex method*), 139
- split() (*plugins.curve.Curve method*), 434
- split() (*trisurface.TriSurface method*), 272
- split() (*varray.Varray method*), 348
- splitAlpha() (*in module gui.appMenu*), 427
- splitar() (*in module arraytools*), 166
- splitArgs() (*in module multi*), 388
- splitAt() (*plugins.curve.BezierSpline method*), 444
- splitAt() (*plugins.curve.PolyLine method*), 441
- splitAtLength() (*plugins.curve.PolyLine method*), 442
- splitBezierCurve() (*in module plugins.nurbs*), 495
- splitByAngle() (*plugins.curve.PolyLine method*), 440
- splitByConnection() (*mesh.Mesh method*), 249
- splitDegenerate() (*mesh.Mesh method*), 254
- splitDigits() (*in module utils*), 339
- splitElems() (*plugins.fe.Model method*), 452
- splitFileDescription() (*in module utils*), 331
- splitFloat() (*in module plugins.datareader*), 448
- splitKeyValue() (*flatkeydb.FlatDB method*), 575
- splitKeyValue() (*in module flatkeydb*), 576
- splitProp() (*geometry.Geometry method*), 292
- splitProp() (*in module plugins.partition*), 498
- splitRandom() (*mesh.Mesh method*), 252
- splitrange() (*in module arraytools*), 166
- st() (*in module plugins.turtle*), 514
- start_draw() (*gui.viewport.QtCanvas method*), 418
- start_drawing() (*gui.viewport.QtCanvas method*), 419
- start_js() (*plugins.webgl.WebGL method*), 518
- start_selection() (*gui.viewport.QtCanvas method*), 417
- startGui() (*in module script*), 215
- startPart() (*in module plugins.ccxinp*), 431
- stat() (*path.Path method*), 569
- stats() (*trisurface.TriSurface method*), 268
- status() (*plugins.lima.Lima method*), 482
- stem (*path.Path attribute*), 564
- stlConvert() (*in module trisurface*), 276
- stopatbreakpt() (*in module script*), 213
- storeSoftware() (*in module software*), 392
- stretch() (*in module arraytools*), 167
- stringar() (*in module arraytools*), 168
- stripLine() (*in module plugins.tetgen*), 510
- strNorm() (*in module utils*), 336
- stroke() (*plugins.curve.Contour method*), 445
- structuredHexMeshGrid() (*in module plugins.bifmesh*), 429
- structuredQuadMeshGrid() (*in module plugins.bifmesh*), 429
- stuur() (*in module arraytools*), 206
- sub_curvature() (*plugins.curve.BezierSpline method*), 443
- sub_directions() (*plugins.curve.Arc method*), 446

- sub_directions() (*plugins.curve.BezierSpline method*), 443
 - sub_directions() (*plugins.curve.Contour method*), 445
 - sub_directions() (*plugins.curve.Curve method*), 433
 - sub_directions() (*plugins.curve.PolyLine method*), 439
 - sub_directions_2() (*plugins.curve.Curve method*), 433
 - sub_points() (*plugins.curve.Arc method*), 446
 - sub_points() (*plugins.curve.Arc3 method*), 446
 - sub_points() (*plugins.curve.BezierSpline method*), 443
 - sub_points() (*plugins.curve.CardinalSpline2 method*), 446
 - sub_points() (*plugins.curve.Contour method*), 445
 - sub_points() (*plugins.curve.Curve method*), 433
 - sub_points() (*plugins.curve.NaturalSpline method*), 446
 - sub_points() (*plugins.curve.PolyLine method*), 439
 - sub_points_2() (*plugins.curve.Curve method*), 433
 - sub_points_2() (*plugins.curve.PolyLine method*), 439
 - subCurve() (*plugins.nurbs.NurbsCurve method*), 490
 - subDict() (*in module utils*), 340
 - subdivide() (*formex.Formex method*), 150
 - subdivide() (*mesh.Mesh method*), 253
 - subElems() (*opengl.drawable.Actor method*), 538
 - subMenus() (*gui.menu.BaseMenu method*), 411
 - subPoints() (*plugins.curve.Curve method*), 434
 - subsets() (*in module arraytools*), 189
 - suffix (*path.Path attribute*), 564
 - superSpherical() (*coords.Coords method*), 100
 - superSpherical() (*geometry.Geometry method*), 287
 - surface2webgl() (*in module plugins.webgl*), 519
 - surface_volume() (*in module inertia*), 384
 - surface_volume_inertia() (*in module inertia*), 384
 - surfaceType() (*trisurface.TriSurface method*), 266
 - swapAxes() (*coords.Coords method*), 107
 - swapaxes() (*coords.Coords method*), 76
 - swapAxes() (*geometry.Geometry method*), 288
 - swapCols() (*opengl.matrix.Matrix4 method*), 541
 - swapRows() (*opengl.matrix.Matrix4 method*), 541
 - sweep() (*mesh.Mesh method*), 257
 - sweep() (*plugins.curve.Curve method*), 437
 - sweep2() (*plugins.curve.Curve method*), 437
 - sym (*inertia.Tensor attribute*), 382
 - symdiff() (*adjacency.Adjacency method*), 352
 - symdifference() (*in module olist*), 559
 - symlink() (*path.Path method*), 569
 - symlinks() (*path.Path method*), 570
 - system() (*in module utils*), 329
- ## T
- tabInputItem() (*in module gui.widgets*), 409
 - Table (*class in gui.widgets*), 404
 - TableModel (*class in gui.widgets*), 402
 - tand() (*in module arraytools*), 171
 - TempDir (*class in utils*), 325
 - TempFile() (*in module utils*), 335
 - Tensor (*class in inertia*), 381
 - tensor (*inertia.Tensor attribute*), 381
 - test() (*coords.Coords method*), 88
 - test() (*formex.Formex method*), 141
 - test() (*mesh.Mesh method*), 258
 - testBbox() (*geometry.Geometry method*), 285
 - testDegenerate() (*connectivity.Connectivity method*), 296
 - tetgen() (*trisurface.TriSurface method*), 274
 - tetgenConvexHull() (*in module plugins.tetgen*), 512
 - tetMesh() (*in module plugins.tetgen*), 512
 - tetrahedral_center() (*in module inertia*), 385
 - tetrahedral_inertia() (*in module inertia*), 384
 - tetrahedral_volume() (*in module inertia*), 384
 - texCoords() (*opengl.textext.FontTexture method*), 547
 - Text (*class in opengl.textext*), 547
 - text() (*gui.widgets.InputBool method*), 395
 - text() (*gui.widgets.InputItem method*), 394
 - text() (*gui.widgets.InputWidget method*), 400
 - TextArray (*class in opengl.textext*), 548
 - TextBox (*class in gui.widgets*), 407
 - Texture (*class in opengl.texture*), 548
 - tilt() (*opengl.camera.Camera method*), 522
 - timedOut() (*gui.widgets.InputDialog method*), 402
 - timeEval() (*in module utils*), 338
 - timeout() (*gui.widgets.InputDialog method*), 402
 - timeout() (*in module gui.toolbar*), 429
 - Timer (*class in timer*), 576
 - timer (*module*), 576
 - toArray() (*varray.Varray method*), 348
 - toBezier() (*plugins.nurbs.NurbsCurve method*), 490
 - toCoords() (*plugins.nurbs.Coords4 method*), 485
 - toCoords4() (*in module plugins.nurbs*), 494
 - toCS() (*coords.Coords method*), 95
 - toCS() (*geometry.Geometry method*), 286
 - toCS() (*inertia.Inertia method*), 383
 - toCurve() (*mesh.Mesh method*), 241
 - toCurve() (*plugins.nurbs.NurbsCurve method*), 490
 - toCylindrical() (*coords.Coords method*), 99
 - toCylindrical() (*geometry.Geometry method*), 287
 - toEye() (*opengl.camera.Camera method*), 524
 - toFormex() (*elements.ElementType method*), 316
 - toFormex() (*mesh.Mesh method*), 241

- toFormex() (*plugins.curve.Curve method*), 437
toFormex() (*plugins.curve.PolyLine method*), 439
toFront() (*in module olist*), 560
toggleAnnotation() (*plugins.objects.DrawableObjects method*), 497
toggleButton() (*in module gui.toolbar*), 429
toLines() (*in module plugins.dxf*), 450
toList() (*varray.Varray method*), 348
toMesh() (*elements.ElementType method*), 316
toMesh() (*formex.Formex method*), 135
toMesh() (*mesh.Mesh method*), 241
toMesh() (*plugins.curve.BezierSpline method*), 444
toMesh() (*plugins.curve.Contour method*), 445
toMesh() (*plugins.curve.PolyLine method*), 439
toNDC() (*opengl.camera.Camera method*), 524
toNDC1() (*opengl.camera.Camera method*), 525
toNurbs() (*plugins.curve.Curve method*), 438
toolbar() (*gui.menu.ActionList method*), 412
toProp() (*geometry.Geometry method*), 289
toSpherical() (*coords.Coords method*), 102
toSpherical() (*geometry.Geometry method*), 287
toSurface() (*formex.Formex method*), 135
toSurface() (*mesh.Mesh method*), 241
TotalEnergies() (*plugins.fe_post.FeResult method*), 475
totalMemSize() (*in module utils*), 343
touch() (*path.Path method*), 569
toWindow() (*opengl.camera.Camera method*), 524
toWorld() (*opengl.camera.Camera method*), 524
track (*module*), 577
track_class_factory() (*in module track*), 579
track_decorator() (*in module track*), 579
track_methods (*in module track*), 578
TrackedDict (*class in track*), 578
TrackedList (*class in track*), 578
transArea() (*opengl.camera.Camera method*), 522
transform() (*opengl.matrix.Matrix4 method*), 541
transform() (*plugins.isopar.Isopar method*), 481
transformCS() (*coords.Coords method*), 96
transformCS() (*geometry.Geometry method*), 287
transinv() (*opengl.matrix.Matrix4 method*), 541
translate() (*coords.Coords method*), 90
translate() (*coordsys.CoordSys method*), 281
translate() (*geometry.Geometry method*), 286
translate() (*inertia.Inertia method*), 383
translate() (*opengl.matrix.Matrix4 method*), 540
translate() (*plugins.lima.Lima method*), 482
translatem() (*formex.Formex method*), 144
translateTo() (*inertia.Inertia method*), 383
transparent() (*in module gui.draw*), 232
trfmat() (*in module arraytools*), 182
triangle() (*in module simple*), 357
triangleBoundingCircle() (*in module geomtools*), 379
triangleCircumCircle() (*in module geomtools*), 379
triangleInCircle() (*in module geomtools*), 378
triangleObtuse() (*in module geomtools*), 379
triangleQuadMesh() (*in module mesh*), 263
TriSurface (*class in trisurface*), 264
trisurface (*module*), 264
trl (*coordsys.CoordSys attribute*), 279, 280
trl() (*coords.Coords method*), 121
trl() (*geometry.Geometry method*), 288
truncate() (*path.Path method*), 569
- ## U
- u (*coordsys.CoordSys attribute*), 279, 280
unbind() (*opengl.shader.Shader method*), 546
unblend() (*plugins.nurbs.NurbsCurve method*), 490
unchanged() (*utils.DictDiff method*), 328
unclamp() (*plugins.nurbs.NurbsCurve method*), 490
uncompress() (*project.Project method*), 362
underlineHeader() (*in module utils*), 336
undoChanges() (*plugins.objects.DrawableObjects method*), 497
undoChanges() (*plugins.objects.Objects method*), 496
undraw() (*in module gui.draw*), 229
uniformFloat() (*opengl.shader.Shader method*), 546
uniformInt() (*opengl.shader.Shader method*), 546
uniformMat3() (*opengl.shader.Shader method*), 546
uniformMat4() (*opengl.shader.Shader method*), 546
uniformParamValues() (*in module arraytools*), 207
uniformVec3() (*opengl.shader.Shader method*), 546
union() (*in module olist*), 559
unique() (*coords.Coords method*), 115
uniqueRows() (*in module arraytools*), 194
uniqueRowsIndex() (*in module arraytools*), 195
unitAttractor() (*in module arraytools*), 207
unitDivisor() (*in module arraytools*), 206
UnitsSystem (*class in plugins.units*), 514
unitVector() (*in module arraytools*), 179
unix2dos() (*in module utils*), 337
Unknown() (*plugins.fe_post.FeResult method*), 475
unlink() (*path.Path method*), 568
Unpickler (*class in project*), 360
unproject() (*opengl.camera.Camera method*), 525
unproject() (*opengl.canvas.Canvas method*), 531
unQuote() (*in module flatkeydb*), 576
update() (*config.Config method*), 556
update() (*gui.widgets.Table method*), 404
update() (*mydict.Dict method*), 551
update() (*opengl.canvas.CanvasSettings method*), 527
update() (*track.TrackedDict method*), 578
updateButton() (*in module gui.toolbar*), 429

updateData() (*gui.widgets.InputDialog method*), 402
 updateDialogItems() (*in module gui.widgets*), 409
 updateGUI() (*in module gui.draw*), 221
 updateLightButton() (*in module gui.toolbar*), 429
 updateNormalsButton() (*in module gui.toolbar*), 429
 updatePerspectiveButton() (*in module gui.toolbar*), 429
 updateText() (*in module gui.widgets*), 410
 updateTransparencyButton() (*in module gui.toolbar*), 429
 updateWireButton() (*in module gui.toolbar*), 429
 upvector (*opengl.camera.Camera attribute*), 521
 urange() (*plugins.nurbs.NurbsCurve method*), 488
 urange() (*plugins.nurbs.NurbsSurface method*), 492
 userName() (*in module utils*), 338
 utils (*module*), 325

V

v (*coordsys.CoordSys attribute*), 279, 280
 value() (*gui.widgets.FileDialog method*), 405
 value() (*gui.widgets.InputBool method*), 395
 value() (*gui.widgets.InputCombo method*), 396
 value() (*gui.widgets.InputFile method*), 399
 value() (*gui.widgets.InputFloat method*), 397
 value() (*gui.widgets.InputGroup method*), 400
 value() (*gui.widgets.InputInfo method*), 394
 value() (*gui.widgets.InputInteger method*), 397
 value() (*gui.widgets.InputItem method*), 394
 value() (*gui.widgets.InputIVector method*), 398
 value() (*gui.widgets.InputList method*), 396
 value() (*gui.widgets.InputPoint method*), 398
 value() (*gui.widgets.InputPush method*), 397
 value() (*gui.widgets.InputRadio method*), 396
 value() (*gui.widgets.InputString method*), 395
 value() (*gui.widgets.InputTable method*), 398
 value() (*gui.widgets.InputText method*), 395
 value() (*gui.widgets.InputWidget method*), 400
 value() (*gui.widgets.ListSelection method*), 406
 value() (*gui.widgets.Table method*), 404
 values() (*plugins.nurbs.KnotVector method*), 486
 values() (*track.TrackedDict method*), 578
 Varray (*class in varray*), 343
 varray (*module*), 343
 Vector4 (*class in opengl.matrix*), 539
 vectorPairAngle() (*in module arraytools*), 201
 vectorPairArea() (*in module arraytools*), 200
 vectorPairAreaNormals() (*in module arraytools*), 199
 vectorPairCosAngle() (*in module arraytools*), 201
 vectorPairNormals() (*in module arraytools*), 199
 vectors() (*plugins.curve.PolyLine method*), 439
 vectors() (*plugins.polygon.Polygon method*), 499

vectorTripleProduct() (*in module arraytools*), 200
 versaText() (*in module utils*), 337
 vertex() (*plugins.dxf.DxfExporter method*), 449
 vertexDistance() (*in module geomtools*), 376
 vertexReductionDP() (*plugins.curve.PolyLine method*), 442
 vertices() (*trisurface.TriSurface method*), 264
 view() (*formex.Formex method*), 134
 view() (*in module gui.draw*), 230
 viewIndex() (*gui.viewport.MultiCanvas method*), 421
 viewport (*opengl.camera.Camera attribute*), 521
 viewport() (*in module gui.draw*), 221
 visualizeSubmappingQuadRegion() (*in module plugins.bifmesh*), 430
 volume() (*mesh.Mesh method*), 260
 volume() (*trisurface.TriSurface method*), 265
 volumeInertia() (*trisurface.TriSurface method*), 265
 volumes() (*formex.Formex method*), 153
 volumes() (*mesh.Mesh method*), 260
 voxelize() (*trisurface.TriSurface method*), 274
 vrange() (*plugins.nurbs.NurbsSurface method*), 492

W

w (*coordsys.CoordSys attribute*), 279, 280
 w() (*plugins.nurbs.Coords4 method*), 485
 wait() (*in module gui.draw*), 219
 wait_drawing() (*gui.viewport.QtCanvas method*), 419
 wait_selection() (*gui.viewport.QtCanvas method*), 417
 walk() (*path.Path method*), 570
 warning() (*in module gui.draw*), 216
 warning() (*in module utils*), 328
 WEBcolor() (*in module opengl.colors*), 234
 WebGL (*class in plugins.webgl*), 516
 webgl() (*trisurface.TriSurface method*), 276
 wedge6_roll() (*in module mesh*), 263
 wedge6_tet4() (*in module mesh*), 263
 wheel_zoom() (*gui.viewport.QtCanvas method*), 420
 wheelEvent() (*gui.viewport.QtCanvas method*), 420
 where() (*varray.Varray method*), 347
 whereProp() (*geometry.Geometry method*), 290
 width (*varray.Varray attribute*), 344
 wireMode() (*in module gui.draw*), 232
 with_name() (*path.Path method*), 565
 with_suffix() (*path.Path method*), 565
 worker() (*in module multi*), 389
 write() (*config.Config method*), 557
 write() (*geometry.Geometry method*), 293
 write() (*geomfile.GeometryFile method*), 363
 write() (*plugins.dxf.DxfExporter method*), 449
 write() (*plugins.export.ObjFile method*), 451

- write() (*plugins.export.PlyFile method*), 451
 write() (*plugins.fe_abq.AbqData method*), 456
 write() (*trisurface.TriSurface method*), 265
 write_bytes() (*path.Path method*), 570
 write_neu() (*in module plugins.neu_exp*), 484
 write_stl_asc() (*in module filewrite*), 388
 write_stl_bin() (*in module filewrite*), 388
 write_text() (*path.Path method*), 570
 writeAmplitude() (*in module plugins.fe_abq*), 472
 writeArray() (*in module arraytools*), 169
 writeAttrib() (*geomfile.GeometryFile method*), 364
 writeBCsets() (*in module plugins.neu_exp*), 483
 writeBezierSpline() (*geomfile.GeometryFile method*), 364
 writeCurve() (*geomfile.GeometryFile method*), 364
 writeData() (*geomfile.GeometryFile method*), 363
 writeData() (*in module filewrite*), 387
 writeDisplacements() (*in module plugins.fe_abq*), 471
 writeDistribution() (*in module plugins.fe_abq*), 471
 writeElemResult() (*in module plugins.fe_abq*), 473
 writeElems() (*in module plugins.fe_abq*), 471
 writeElems() (*in module plugins.neu_exp*), 483
 writeFile() (*flatkeydb.FlatDB method*), 576
 writeFileOutput() (*in module plugins.fe_abq*), 473
 writeFormex() (*geomfile.GeometryFile method*), 364
 writeGeometry() (*geomfile.GeometryFile method*), 364
 writeGeomFile() (*in module script*), 215
 writeGroup() (*in module plugins.neu_exp*), 483
 writeGTS() (*in module filewrite*), 388
 writeHeader() (*geomfile.GeometryFile method*), 363
 writeHeading() (*in module plugins.neu_exp*), 483
 writeIData() (*in module filewrite*), 387
 writeline() (*geomfile.GeometryFile method*), 363
 writeMesh() (*geomfile.GeometryFile method*), 364
 writeNodeResult() (*in module plugins.fe_abq*), 472
 writeNodes() (*in module plugins.fe_abq*), 471
 writeNodes() (*in module plugins.neu_exp*), 483
 writeNodes() (*in module plugins.tetgen*), 511
 writeNurbsCurve() (*geomfile.GeometryFile method*), 365
 writeNurbsSurface() (*geomfile.GeometryFile method*), 365
 writeOBJ() (*in module filewrite*), 387
 writeOFF() (*in module filewrite*), 387
 writePLY() (*in module filewrite*), 387
 writePolyLine() (*geomfile.GeometryFile method*), 364
 writeSection() (*in module plugins.fe_abq*), 471
 writeSet() (*in module plugins.fe_abq*), 471
 writeSmesh() (*in module plugins.tetgen*), 511
 writeSTL() (*in module filewrite*), 388
 writeSurface() (*in module plugins.tetgen*), 511
 writeTetMesh() (*in module plugins.tetgen*), 511
 writeTmesh() (*in module plugins.tetgen*), 511
 writeToFile() (*plugins.objects.Objects method*), 496
 writeTriSurface() (*geomfile.GeometryFile method*), 364
- ## X
- x (*coords.Coords attribute*), 74, 76
 x (*geometry.Geometry attribute*), 283
 x() (*plugins.nurbs.Coords4 method*), 485
 xpattern() (*in module coords*), 125
 xy (*coords.Coords attribute*), 75, 76
 xy (*geometry.Geometry attribute*), 283
 xyz (*coords.Coords attribute*), 74, 76
 xyz (*geometry.Geometry attribute*), 283
 xz (*coords.Coords attribute*), 75, 76
 xz (*geometry.Geometry attribute*), 283
- ## Y
- y (*coords.Coords attribute*), 75, 76
 y (*geometry.Geometry attribute*), 283
 y() (*plugins.nurbs.Coords4 method*), 485
 yz (*coords.Coords attribute*), 75, 76
 yz (*geometry.Geometry attribute*), 283
- ## Z
- z (*coords.Coords attribute*), 75, 76
 z (*geometry.Geometry attribute*), 283
 z() (*plugins.nurbs.Coords4 method*), 485
 zipExtract() (*in module utils*), 336
 zipList() (*in module utils*), 336
 zoom() (*in module gui.draw*), 220
 zoom() (*opengl.canvas.Canvas method*), 531
 zoomAll() (*in module gui.draw*), 220
 zoomAll() (*opengl.canvas.Canvas method*), 531
 zoomArea() (*opengl.camera.Camera method*), 522
 zoomBbox() (*in module gui.draw*), 220
 zoomCentered() (*opengl.canvas.Canvas method*), 531
 zoomObj() (*in module gui.draw*), 220
 zoomRectangle() (*in module gui.draw*), 220
 zoomRectangle() (*opengl.canvas.Canvas method*), 531
 zspacing() (*plugins.imagearray.DicomStack method*), 477
 zvalues() (*plugins.imagearray.DicomStack method*), 477