pyFormex Documentation

Release 0.9.1

Benedict Verhegghe

October 15, 2013

CONTENTS

1 Introduction to pyFormex **1**
1.1 What is pyFormex? 1
1.2 License and Disclaimer 3
1.3 Installation 3
1.4 Using pyFormex 3
1.5 Getting Help 4

2 Installing pyFormex **5**
2.1 Choose installation type 5
2.2 Debian packages 7
2.3 Official release 7
2.4 Alpha release 10
2.5 Development version 11
2.6 BuMPix Live GNU/Linux system 12
2.7 Running pyFormex on non-Linux systems 12

3 pyFormex tutorial **15**
3.1 The philosophy 15
3.2 Getting started 16
3.3 Some basic Python concepts 19
3.4 Some basic NumPy concepts 21
3.5 The Formex data model 21
3.6 Creating a Formex 22
3.7 Concatenation and lists of Formices 29
3.8 Formex property numbers 30
3.9 Getting information about a Formex 32
3.10 Saving geometry 32
3.11 Saving images 33
3.12 Transforming a Formex 34
3.13 Converting a Formex to a Mesh model 37

4 pyFormex user guide **39**
4.1 Running pyFormex 39
4.2 Command line options 40
4.3 Running without the GUI 41
4.4 The Graphical User Interface 41
4.5 pyFormex scripting 46
4.6 Modeling Geometry with pyFormex 49

4.7 The Canvas . . . 50
4.8 Creating Images . . . 52
4.9 Using Projects . . . 53
4.10 Assigning properties to geometry . . . 53
4.11 Using Widgets . . . 60
4.12 pyFormex plugins . . . 60
4.13 Configuring pyFormex . . . 61

5 pyFormex example scripts 65
5.1 WireStent . . . 65
5.2 Operating on surface meshes . . . 80

6 pyFormex reference manual 83
6.1 Autoloaded modules . . . 83
6.2 Other pyFormex core modules . . . 158
6.3 pyFormex GUI modules . . . 236
6.4 pyFormex plugins . . . 310
6.5 pyFormex plugin menus . . . 486
6.6 pyFormex tools . . . 486

7 pyFormex FAQ 'n TRICKS 499
7.1 FAQ . . . 499
7.2 TRICKS . . . 500

8 pyFormex file formats 505
8.1 Introduction . . . 505
8.2 pyFormex Project File Format . . . 505
8.3 pyFormex Geometry File Format 1.6 . . . 506

9 BuMPix Live GNU/Linux system 509
9.1 What is BuMPix . . . 509
9.2 Obtain a BuMPix Live bootable medium . . . 509
9.3 Boot your BuMPix system . . . 511
9.4 FAQ . . . 511
9.5 Upgrade the pyFormex version on a BuMPix-0.6.1 USB stick . . . 512

10 GNU GENERAL PUBLIC LICENSE 515
10.1 Preamble . . . 515
10.2 Terms and Conditions . . . 516
10.3 How to Apply These Terms to Your New Programs . . . 524

11 About the pyFormex documentation 527
11.1 The people who did it . . . 527
11.2 How we did it . . . 527

Python Module Index 529

Index 531

CHAPTER

ONE

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 succesfully 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 stent [1] 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

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

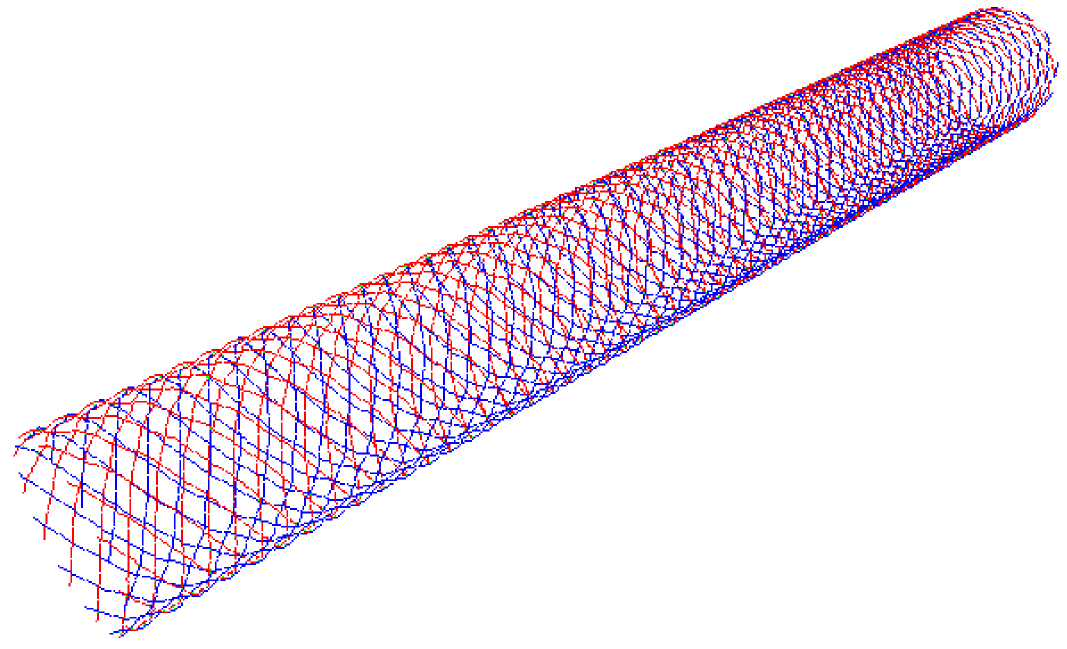


Figure 1.1: WireStent example

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.
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.

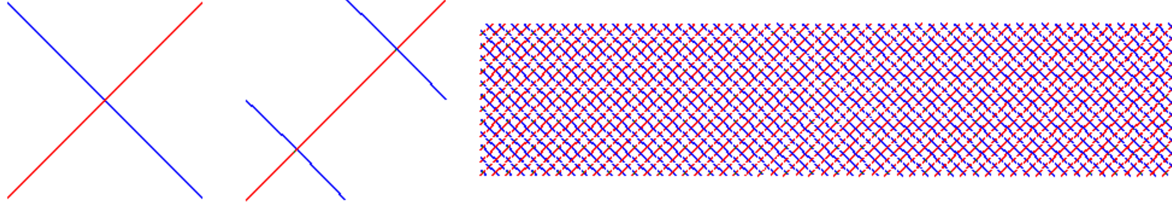


Figure 1.2: First three steps in building the WireStent example

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 mainframe 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 ©2004-2012 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.

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

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 change you will get helped quickly there. 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]

CHAPTER

TWO

INSTALLING PYFORMEX

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 Choose installation type

There are several ways to get a running installation of pyFormex, and you may choose the most appropriate method for you, depending on your needs, your current infrastructure and your computer knowledge. We will describe them in detail in this document, and advice you on which method might be the best in your case.

Note: pyFormex on non-Linux systems

pyFormex is being developed on GNU/Linux systems, and most users run it on a GNU/Linux platform. The *Running pyFormex on non-Linux systems* section holds information on how pyFormex can be run on other platforms.

Let's first give you an overview of the most important pros and cons of the different install methods.

Debian packages:

PROS	CONS
• Stable • Well supported • Easy install procedure • Automatic installation of all dependencies • Easy update procecure • Easy removal procecure • Site-wide install	• Debian GNU/Linux required [1] • Root access required • May be missing latest features

Official release:

[1] Installing the Debian packages may also work on Debian derivatives like Ubuntu and Mint.

PROS	CONS
• Stable • Well supported • Easy install procedure • Site-wide install	• GNU/Linux required • Root access required • Installation of dependencies required • May be missing latest features

Alpha release:

PROS	CONS
• Easy install procedure • Site-wide install	• GNU/Linux required • Root access required • Installation of dependencies required • Latests features

Development version:

PROS	CONS
• Latest features • No root access required • No installation required	• GNU/Linux required • Requires development tools • (Usually) single user install • Manual installation of dependencies (and root access) may be required • Less stable

BuMPix Live GNU/Linux system:

PROS	CONS
• No GNU/Linux required • No root access required • No installation required • Stable version • Easily portable • Upgradeable by installing development version	• Missing latest features • Somewhat slower loading

To sum it up:

- Unless you want to help with the development, or you absolutely need some of the latest features or bugfixes, or you just can not meet the requirements, the latest *Debian packages* or *Official release* source tarballs are what you want to go for. They give you the highest degree of stability and support and come packed in an archive, with an easy install procedure provided by your distributions package manager or included in the source tarball.

- If you need some recent feature of pyFormex that is not yet in an official release, you may be lucky to find it in some *Alpha release*.
- If the install procedures do not work for you, or you need the absolutely latest development code, you can run pyFormex directly from the anonymously checked out *Development version*.
- Finally, if you do not have enough permissions to install the dependencies, or if you do not have a GNU/Linux system in the first place, or if you just want to try out pyFormex without having to install anything, or if you want a portable system that can you take with you and run anywhere, choose for the *BuMPix Live GNU/Linux system* on USB stick.

2.2 Debian packages

If you are running Debian GNU/Linux, or have the opportunity to install it, then (by far) the most easy install method is to use the packages in the official Debian repositories. Currently pyFormex packages are available for Debian sid and wheezy releases. Be sure to also install the precompiled acceleration libraries:

```
apt-get install pyformex pyformex-lib
```

This single command will install pyFormex and all its dependencies. Some extra functionalities may be installable from a separate package:

```
apt-get install pyformex-extra
```

If you need a more recent version of pyFormex then the one available in the official repositories, you may try your luck with our local package repository. It contains debian format packages of intermediate releases and test packages for the official releases. To access our package repository from your normal package manager, add the following lines to your */etc/apt/sources.list*:

```
deb http://bumps.ugent.be/repos/debian/ sid main
deb-src http://bumps.ugent.be/repos/debian/ sid main
```

You can add the key used to sign the packages to the apt keyring with the following command:

```
wget -O - http://bumps.ugent.be/repos/pyformex-pubkey.gpg | apt-key add -
```

Then do `apt-get update`. Now you can use the above commands to install the latest alpha release.

2.3 Official release

pyFormex is software under development, and many users run it directly from the latest development sources. This holds a certain risk however, because the development version may at times become unstable or incompatible with previous versions and thus break your applications. At regular times we therefore create official releases, which provide a more stable and better documented and supported version, together with an easy install procedure.

If you can meet the requirements for using an officially packed release, and you can not use the *Debian packages*, this is the recommended way to install pyFormex. All the software packages needed to compile and run pyFormex can be obtained for free.

To install an official pyFormex release, you need a working GNU/Linux system, root (administrator) privileges to the system, and you need to make sure that the dependencies listed below are installed first on the system. Furthermore, you need the usual GNU development tools (gcc, make).

If you need to install a new GNU/Linux system from scratch, and have the choice to pick any distribution, we highly recommend Debian GNU/Linux or derivatives. This is because most of the pyFormex development is done on Debian systems, and we will give you precise install instructions for this system. Also, the Debian software repositories are amongst the most comprehensive to be found on the Internet. Furthermore, as of pyFormex version 0.8.6, we provide official *Debian packages*, making installation really a no-brainer.

Most popular GNU/Linux distributions provide appropriately packed recent versions of the dependencies, so that you can install them easily from the pacakge manager of your system. In case a package or version is not available for your system, you can always install it from source. We provide the websites where you can find the source packages.

2.3.1 Dependencies

In order to install an official release package of pyFormex, you need to have the following installed (and working) on your computer:

Python (http://www.python.org) Version 2.5 or higher (2.6 or 2.7 is recommended). Nearly all GNU/Linux distributions come with Python installed, so this should be no major obstacle.

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.

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

PyQt4 (http://www.riverbankcomputing.co.uk/pyqt/index.php) The Python bindings for Qt4.

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.

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 .

Installing dependencies on *Debian GNU/Linux*

Debian users should just have to install the packages `python-numpy`, `python-opengl` and `python-qt4-gl`. The latter will install `python-qt4` as dependency. Also, for compiling the acceleration library, you should install `python-dev` and `libglu1-mesa-dev`. This command will do it all:

```
apt-get install python-numpy python-opengl python-qt4-gl python-dev libglu1-mesa-dev
```

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

2.3.2 Download pyFormex

Official pyFormex releases can be downloaded from this website: Releases. As of the writing of this manual, the latest release is 0.8.6.

pyFormex is currently distributed in the form of a .tar.gz (tarball) archive. See *Install pyFormex* for how to proceed further with the downloaded file.

2.3.3 Install pyFormex

Once you have downloaded the tarball, unpack it with the command

```
tar xvzf pyformex-VERSION.tar.gz
```

where you replace `VERSION` with the correct version from the downloaded file. Then go to the created pyformex directory

```
cd pyformex-VERSION
```

and execute the following command with root privileges:

```
python setup.py install --prefix=/usr/local
```

This will install pyFormex under `/usr/local/`. You can change the prefix to install pyFormex in some other place.

The installation procedure installs everything into a single directory, and creates a symlink to the executable in `/usr/local/bin`. You can use the command

```
pyformex --whereami
```

to find out where pyFormex is installed.

Finally, a pyFormex tarball installation can usually be removed by giving the command

```
pyformex --remove
```

and answering 'yes' to the question. You may want to do this before installing a new version, especially if you install a new release of an already existing version.

2.3.4 Install additional software

pyFormex uses a large number of external software packages to enhance its functionality. Some of these packages are so essential, that they were listed as requirements. Others however are merely optional packages: interesting for those users who need them, but not essential for everybody. The user has the choice to install these extras or not.

Some external packages however do not come in an easy to install package, or the available packaged formats do not collaborate well with pyFormex. Therefore, we have created a set of dedicated install

script to ease the installation of these external packages. Currently, there is an install procedure for the following packages:

> **Warning:** We provide these installation procedures for your convenience, but take no responsibility for them working correctly.

gl2ps This package allows to save the OpenGL rendering to a file in vector format. Currently supported are `eps`, `pdf` and `svg`. Our install procedure provides the necessary Python interface and installs the gl2ps library at the same time.

gts This package (Gnu Triangluted Surfaces) implements a library of powerful functions for operating on triangulated surface models. It also delivers some example programs built with the library. The pyFormex `surface` plugin uses these for many of its functions. Debian users should install the packages `libgts-0.7.5`, `libgts-bin` and `libgts-dev` as dependencies.

tetgen This package provides a high quality tetrahedral mesher. pyFormex has some import and export functions for the specific `tetgen` file formats. Since `tetgen` is only distributed in source form, we provide this install procedure to help with the compile/install.

calpy Calpy is an experimental package that provides efficient Finite Element simulations through a Python interface. It does this by calling into a library of compiled Fortran routines. There is currently no distribution to the general public yet, but this install procedure grabs the source from our local FTP server, compiles the library and creates the Python interface. pyFormex comes with some examples that use Calpy as a simulatiopn tool.

To install any of these packages, proceed as follows. Go to the directory where you unpacked the pyFormex distribution: `cd pyformex-version`. Then go to the `pyformex/external` subdirectory, where you will find a subdirectory for each of the above packages. Go into the directory of the package you wish to install and execute the following commands (install may require root privileges):

```
make
make install
```

In some case there is no `Makefile` provided but an install script instead. Then you can just do:

```
./install.sh all
```

All these procedures will install under `/usr/local`. If you wish to change this, you will have to change the `Makefile` or install procedure. The install procedures can also be used to perform only part of the installation process. Thus, `./install.sh get unpack` will only download and unpack that package. See the README files and the install procedures themselves for more info.

2.4 Alpha release

Official releases are only created a couple of times per year, because they require a lot of testing. pyFormex is however developing fast, and as soon as an official release has been made, new features are already being included in the source repository. Sometimes, you may be in need of such a feature, that will only become officially available within several months. Between successive official releases, we create interim alpha releases. They are less well tested (meaning they may contain more bugs) and supported than the official ones, but they may just work well for you.

These alpha releases can be downloaded from the developer FTP site or from our local FTP server. The latter may be slower, but you may find there some old releases or release candidates that are not available on the official server. They install just like the *Official release*.

Again, as a Debian user, you may be extra lucky: we usually create Debian *Debian packages* from these alpha releases and make them available on our local package repository.

2.5 Development version

If the install procedures for the packaged releases do not work for you, or if you want to have the absolutely latest features and bug fixes, then you can run pyFormex directly from the development sources. Obviously, the pyFormex developers use this method, but there are also several normal users who prefer this, because it allows for easy updating to the latest version.

To run pyFormex from the development sources you need to have the same dependencies installed as for the *Official release*. Furthermore, you need the git revision control system. You can check whether you have it by trying the command `git`. If you do not have the command, you should first install it. Debian and Ubuntu users can just do `apt-get install git`.

Now you can anonymously check out the latest pyFormex version from the Source code repository at the Project page, using the command:

```
git clone git://git.savannah.nongnu.org/pyformex.git
```

This will create a directory `pyformex` with the full source.

Now you can directly run pyFormex from the created `pyformex` directory:

```
cd pyformex
pyformex/pyformex
```

The first time you run the command, it will start with compiling the pyFormex acceleration libraries. When that has finished, the pyFormex GUI will start, running directly from your checked out source. The next time you run the command, the library will not be recompiled, unless some updates have been made to the files, making the already compiled versions out of date.

You can make the `pyformex/pyformex` command executable from anywhere by creating a symlink under one of the directories in your `PATH` environment variable. Many GNU/Linux distributions add `/home/USER/bin` to the user's path. Thus the following command is suitable in most cases:

```
ln -sfn BASEDIR/pyformex/pyformex /home/USER/bin
```

where `BASEDIR` is the full path to the directory where you checked out the source.

The pyFormex repository contains a lot of files that are only needed and interesting for the pyFormex developers. As a normal user you may want to remove this extra overhead in your copy. To do so, run the *sparse_checkout* script from the checkout directory:

```
sh sparse_checkout
```

You can update your pyFormex installation at any time to the latest version by issuing the command

```
git pull
```

in your `BASEDIR` directory. You can even roll back to any older revision of pyFormex. Just remember that after updating your sources, the compiled libraries could be out of sync with your new sources. Normally pyFormex will rebuild the libraries the next time you start it. If you ever want to rebuild the libraries without starting the `pyformex` program, you can use the command `make lib` from inside `BASEDIR`.

2.5.1 Using the older Subversion repository

To run pyFormex from the development sources you need to have the same dependencies installed as for the *Official release*. Furthermore, you need the Subversion revision control system. You can check whether you have it by trying the command `svn help`. If you do not have the command, you should install Subversion first. Debian and Ubuntu users can just do `apt-get install subversion`.

Now you can anonymously check out the latest pyFormex version from the SVN Source code repository at the Project page. If you are not a pyFormex developer, the suggested commands for this checkout are:

```
svn co svn://svn.savannah.nongnu.org/pyformex/trunk --depth files MYDIR
svn update --depth infinity MYDIR/pyformex
```

In these commands you should replace `MYDIR` with the path name of a directory where you have write permission. Many users choose `~/pyformex` as the checkout directory, but this is not required. You can even check out different versions under different path names. If you leave out the `MYDIR` from the above command, a new directory `trunk` will be created in the current path.

Instead of the above two commands, you could also use the following single command to check out the whole trunk, but that would download a lot of extra files which are only useful for pyFormex developers, not for normal users

```
svn co svn://svn.savannah.nongnu.org/pyformex/trunk MYDIR
```

Now change into the created `MYDIR` directory, where you can execute the command `pyformex/pyformex` and proceed as explained above for a checkout of the git repository.

2.6 BuMPix Live GNU/Linux system

If you do not have access to a running GNU/Linux system, or if the above installation methods fail for some unknown reason (remember, you can ask for help on the pyFormex Support tracker), you can still run pyFormex by using a Bumpix Live GNU/Linux system. *Bumpix Live* is a full featured Debian GNU/Linux system including pyFormex that can be run from a single removable medium such as a CD or a USB key. Installation instructions can be found in *BuMPix Live GNU/Linux system*.

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.

Further guidelines for using the BuMPix system can be found in *BuMPix Live GNU/Linux system*.

2.7 Running pyFormex on non-Linux systems

pyFormex is being developed on GNU/Linux platforms, and most of its users run pyFormex on a GNU/Linux system. Because of that, there is no installation procedure to run pyFormex natively on

other systems.

Currently, the easiest way to run pyFormex on a non-Linux system is by using the *BuMPix Live GNU/Linux system*. We use this frequently with large groups of students in classes having only Windows PCs. We also have some professional users who could no install GNU/Linux due to corporate regulations, that are working this way.

Another possibility is to run a virtual GNU/Linux instance on the platform. There is currently quite good virtualization software available for nearly any platform.

However, as all the essential underlying packages on which pyFormex is built are available for many other platforms (including Windows, Mac), it is (in theory) possible to get pyFormex to run natively on non-Linux platforms. There has already been a successful attempt with a rather old version, but with recent versions nobody has (yet) taken the bother to try it.

Note: pyFormex on Windows Lately there have been some successful attempts to get the basic functionality of pyFormex running on Windows. Thomas Praet has compiled this document on how to proceed. Submit a request on the Support tracker if you need any help.

There may be a few things that have to be changed to successfully run pyFormex on other platforms (especially on Windows), but it should all be rather obvious. If you have some programming experience on your platform, and you want to give it a try, please do so. We certainly are willing to help where we can. And we are certainly interested in feedback about your attempt, whether successful or not.

CHAPTER

THREE

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 hypar 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.

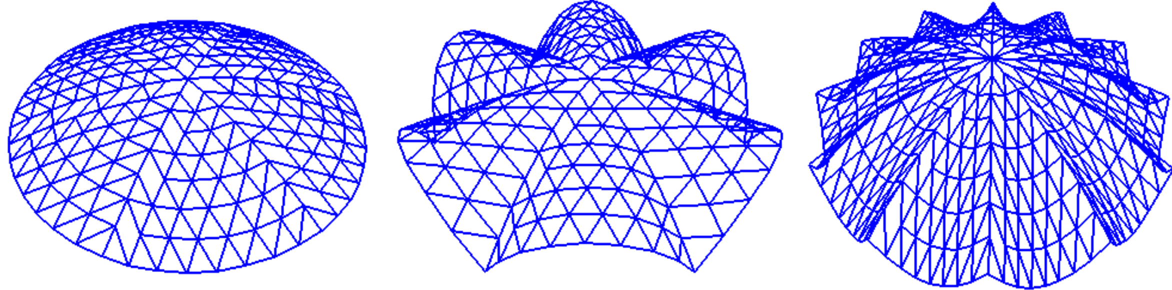


Figure 3.1: Same script, different domes

- 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.

3.2 Getting started

- Start the pyFormex GUI by entering the command `pyformex` in a terminal. Depending on your instalation, 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  # pyformex script/app template
2  #
3  ##
4  ##  Copyright (C) 2011 John Doe (j.doe@somewhere.org)
5  ##  Distributed under the GNU General Public License version 3 or later.
6  ##
7  ##  This program is free software: you can redistribute it and/or modify
8  ##  it under the terms of the GNU General Public License as published by
9  ##  the Free Software Foundation, either version 3 of the License, or
10 ##  (at your option) any later version.
11 ##
12 ##  This program is distributed in the hope that it will be useful,
13 ##  but WITHOUT ANY WARRANTY; without even the implied warranty of
14 ##  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
15 ##  GNU General Public License for more details.
16 ##
17 ##  You should have received a copy of the GNU General Public License
18 ##  along with this program.  If not, see http://www.gnu.org/licenses/.
19 ##
20
21 """pyFormex Script/App Template
22
```

```
23 This is a template file to show the general layout of a pyFormex
24 script or app.
25
26 A pyFormex script is just any simple Python source code file with
27 extension '.py' and is fully read and execution at once.
28
29 A pyFormex app can be a '.py' of '.pyc' file, and should define a function
30 'run()' to be executed by pyFormex. Also, the app should import anything that
31 it needs.
32
33 This template is a common structure that allows the file to be used both as
34 a script or as an app, with almost identical behavior.
35
36 For more details, see the user guide under the `Scripting` section.
37
38 The script starts by preference with a docstring (like this),
39 composed of a short first line, then a blank line and
40 one or more lines explaining the intention of the script.
41 """
42 from __future__ import print_function
43
44 from gui.draw import *  # for an app we need to import explicitely
45
46 def run():
47     """Main function.
48
49     This is executed on each run.
50     """
51     print("This is the pyFormex template script/app")
52
53
54 # Initialization code
55
56 print("This is the initialization code of the pyFormex template script/app")
57
58
59 # The following is to make script and app behavior alike
60 if __name__ == 'draw':
61     print("Running as a script")
62     run()
63
64
65 # 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 typical 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 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 becaused 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 own 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
7  for light in [ 'green','yellow','red','black',None]:
8      if light == 'red':
9          print 'stop'
10     elif light == 'yellow':
11         print 'brake'
12     elif light == 'green':
13         print 'drive'
14     else:
15         print 'THE LIGHT IS BROKEN!'
16
17 appreciation = {
```

```
18      0: 'not driving',
19      30:'slow',
20      60:'normal',
21      90:'dangerous',
22      120:'suicidal'
23      }
24
25  for i in range(5):
26      speed = 30*i
27      print "%s. Driving at speed %s is %s" % (i,speed,appreciation[speed])
28
29  # 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. 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 The 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 the *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.

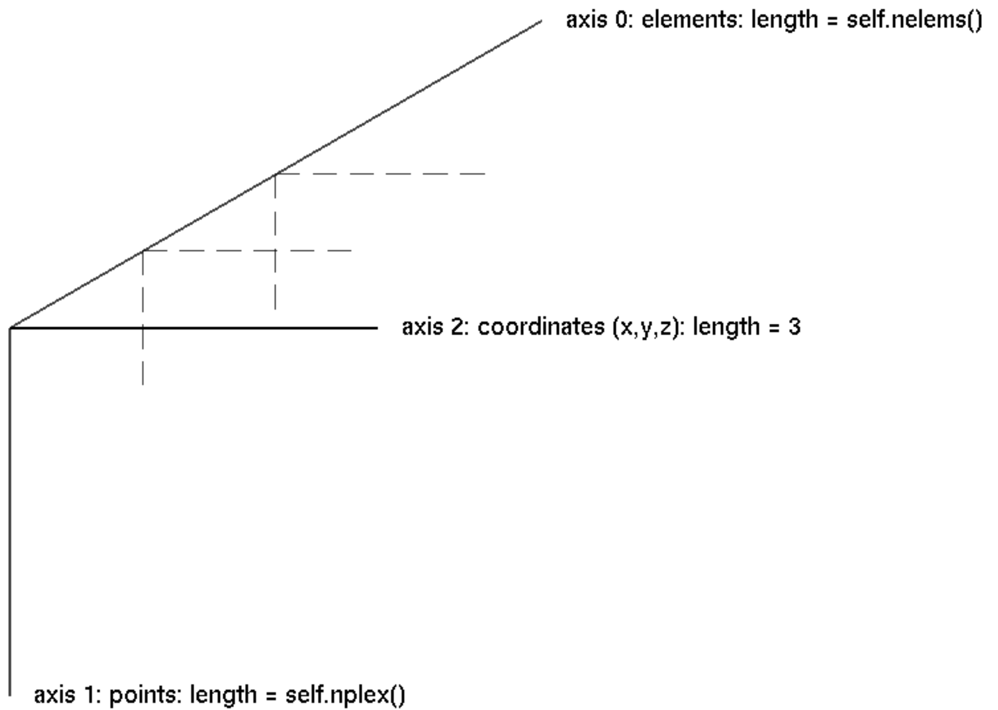


Figure 3.2: The structure of a Formex

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*.

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 interprete 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.

Figure 3.3: 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.

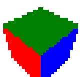
But wait a minute! Does this represent a square surface, or just the four lines constituting the circumference of the square? Actually, it is a square surface, but since the pyFormex GUI by default displays in wireframe mode, unless you have changed it, you will only see the border of the square. You 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

Figure 3.4: A square.

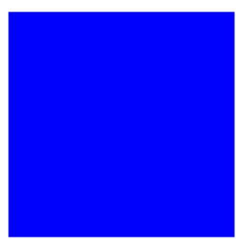
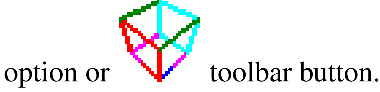


Figure 3.5: The square in smooth rendering.

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:

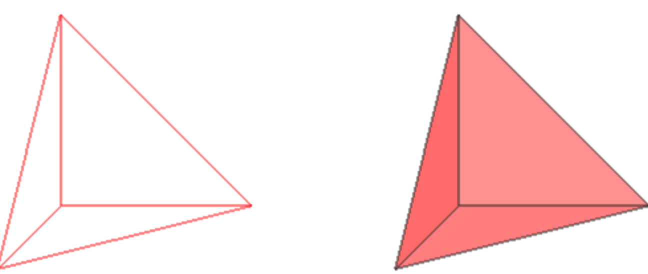
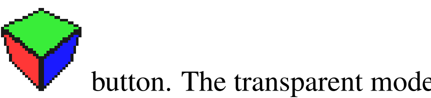


Figure 3.6: The tetrahedron in wireframe and smoothwire (transparent) rendering

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 rigth button will zoomin 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 explicitely 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.

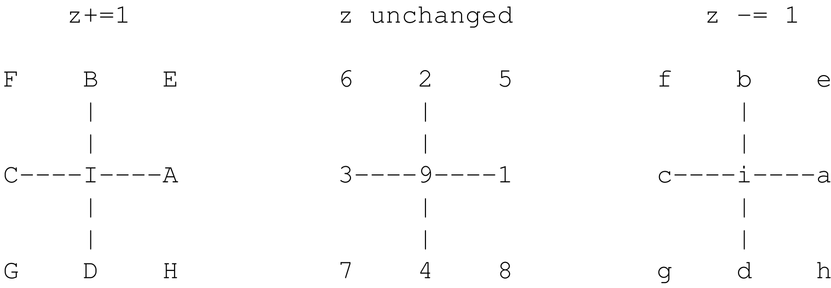
The points are created from a string input, interpreting each character as a code specifying how to move from the previous point to the new point. The start position on entry is the origin [0.,0.,0.].

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 'ABCDEFGHI', 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.

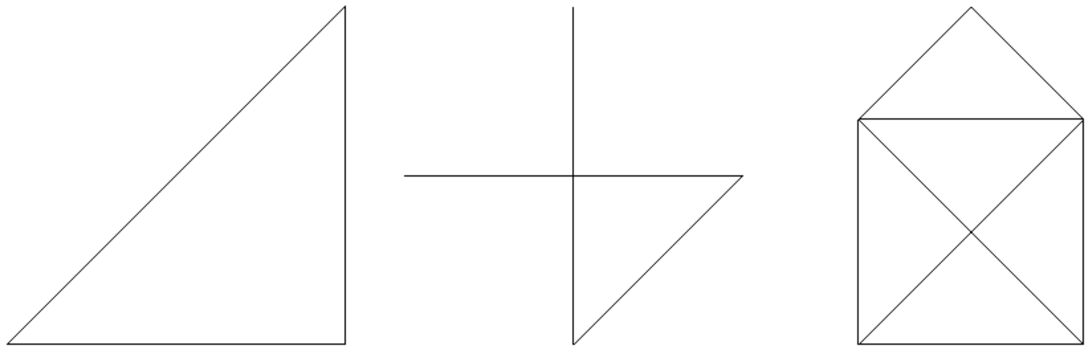


Figure 3.7: Images generated from the patterns '127', '11722' and '22584433553388'

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')

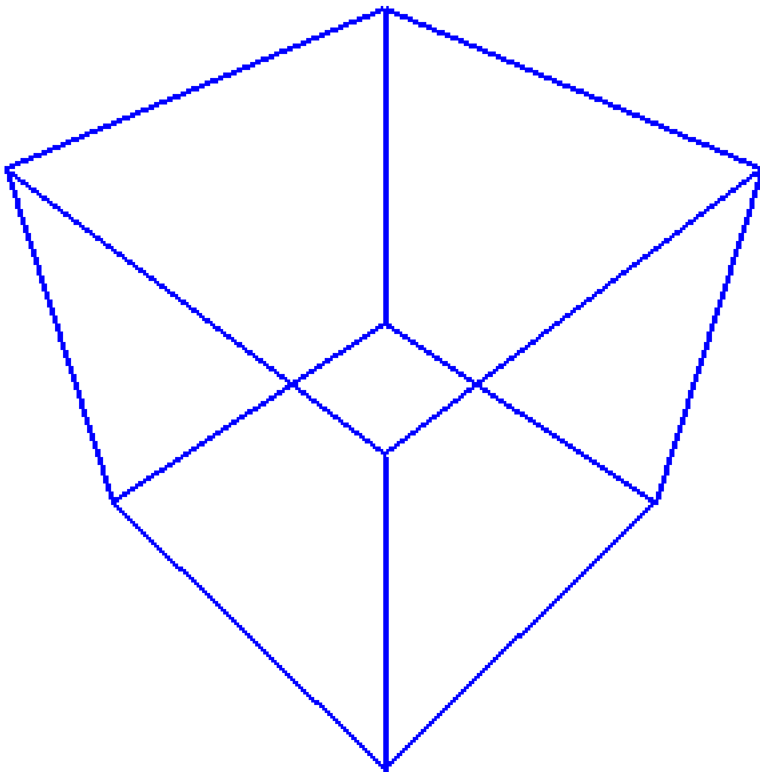


Figure 3.8: A wireframe cube

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 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*.

Figure 3.9: 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 diffference: the + operator does not changing 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.

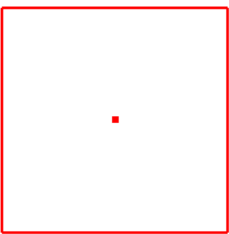


Figure 3.10: A square and its center point.

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 appearence 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).

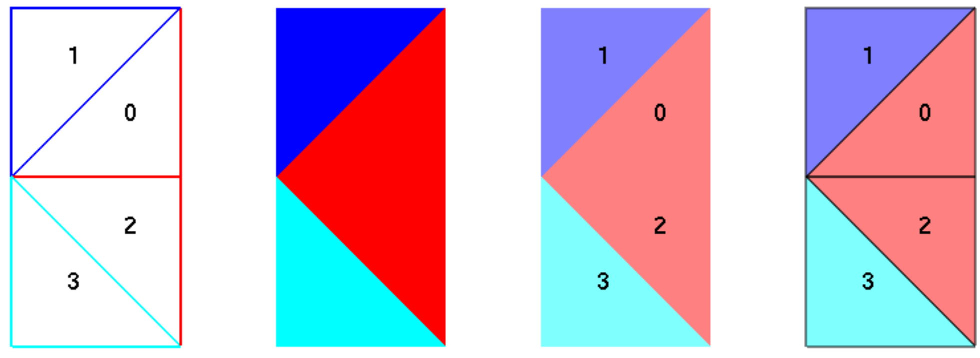
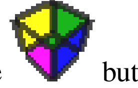


Figure 3.11: A Formex with property numbers drawn as colors. From left to right: wireframe, flat, flat (transparent), flatwire (transparent).

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 attibrute 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
`F.nelems()`	The number of elements in the Formex
`F.nplex()`	The plexitude of the Formex (the number of points in each element of the Formex)
`F.bbox()`	The bounding box of the Formex
`F.center()`	The center of the bbox of the Formex
`F.sizes()`	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* buttton. 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. If you have installed `gl2ps` (see *Install additional software*), you can even save in a number of vector formats, such as `eps` or `svg`.

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	bgcolor()	*Viewport* → *Background Color*
Line width	linewidth()	*Viewport* → *LineWidth*
Canvas Size	canvasSize()	*Viewport* → *Canvas Size*
Render Mode	wireframe(), flat(), flatwire(), smooth(), smoothwire()	*Viewport* → *Render Mode*
Transparency	transparent()	

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:

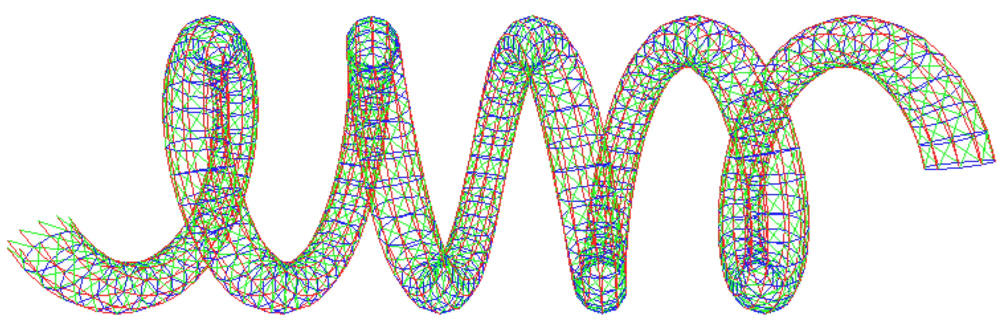


Figure 3.12: A helical frame structure (Helix example)

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.

```
# $Id: 4560b6c on Wed Oct 2 17:10:50 2013 +0200 by Benedict Verhegghe $   *** pyformex *
##
##  This file is part of pyFormex 0.9.1  (Tue Oct 15 21:05:25 CEST 2013)
##  pyFormex is a tool for generating, manipulating and transforming 3D
##  geometrical models by sequences of mathematical operations.
##  Home page: http://pyformex.org
##  Project page:  http://savannah.nongnu.org/projects/pyformex/
##  Copyright 2004-2013 (C) Benedict Verhegghe (benedict.verhegghe@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/.
##
"""Helix example from pyFormex"""
m = 36 # number of cells along helix
n = 10 # number of cells along circular cross section
reset()
setDrawOptions({'clear':True})
F = Formex('l:164'),[1,2,3]); draw(F)
F = F.replic(m,1.,0); draw(F)
F = F.replic(n,1.,1); draw(F)
F = F.translate(2,1.); draw(F,view='iso')
F = F.cylindrical([2,1,0],[1.,360./n,1.]); draw(F)
F = F.replic(5,m*1.,2); draw(F)
F = F.rotate(-10.,0); draw(F)
F = F.translate(0,5.); draw(F)
F = F.cylindrical([0,2,1],[1.,360./m,1.]); draw(F)
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 (lines 26-27) 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.*

Lines 28 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 29 makes sure the subsequent drawing instructions will remove the previous step from the canvas.

In line 30 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])
```

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)
```

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

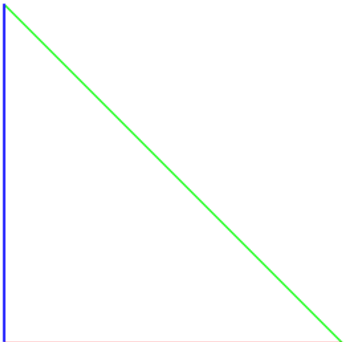


Figure 3.13: The basic Formex

Figure 3.14: Replicated in x-direction

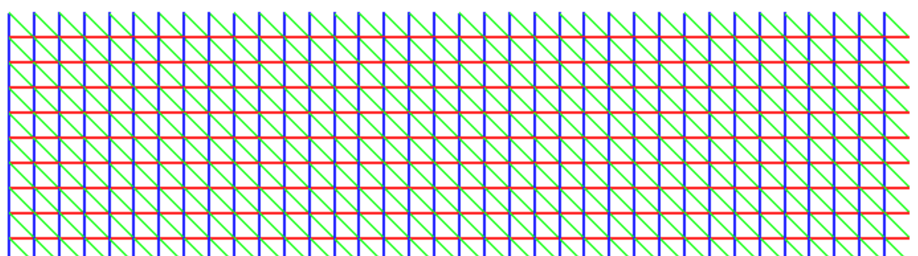


Figure 3.15: Replicated in y-direction

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.

```
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 lenght 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 then 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)
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.

CHAPTER

FOUR

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`.

```
Usage
=====
  pyformex [<options>] [ [ scriptname [scriptargs] ] ...]

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

Options
=======
--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)
--interactive           Go into interactive mode after processing the command
                        line parameters. This is implied by the --gui option.
--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 usage of an OpenGL version. The version
                        should be specified as a string 'a.b'. The default is
                        1.0
--uselib                Use the pyFormex C lib if available. This is the
                        default.
--nouselib              Do not use the pyFormex C-lib.
--commands              Use the commands module to execute external commands.
                        Default is to use the subprocess module.
--config=CONFIG         Use file CONFIG for settings
--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)
                        strings corresponding with the attributes of the
                        DebugLevels class. The individual values are OR-ed
                        together to produce a final debug value. 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
                        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.
--newviewports          Use the new multiple viewport canvas implementation.
                        This is an experimental feature only intended for
                        developers.
--testmodule=TESTMODULE
                        Run the docstring tests for module TESTMODULE.
```

```
                        TESTMODULE is the name of the module, using . as path
                        separator.
--testcamera            Print camera settings whenever they change.
--testexecutor          Test alternate executor: only for developers!
--memtrack              Track memory for leaks. This is only for developers.
--fastnurbs             Test C library nurbs drawing: only for developers!
--pyside                Use the PySide bindings for QT4 libraries
--pyqt4                 Use the PyQt4 bindings for QT4 libraries
--listfiles             List the pyformex Python source files.
--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 interprete this as an extended regular
                        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
--version               show program's version number and exit
--help, -h              show this help message and exit
```

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 interprete 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 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*.

4.4.1 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.

4.4.2 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

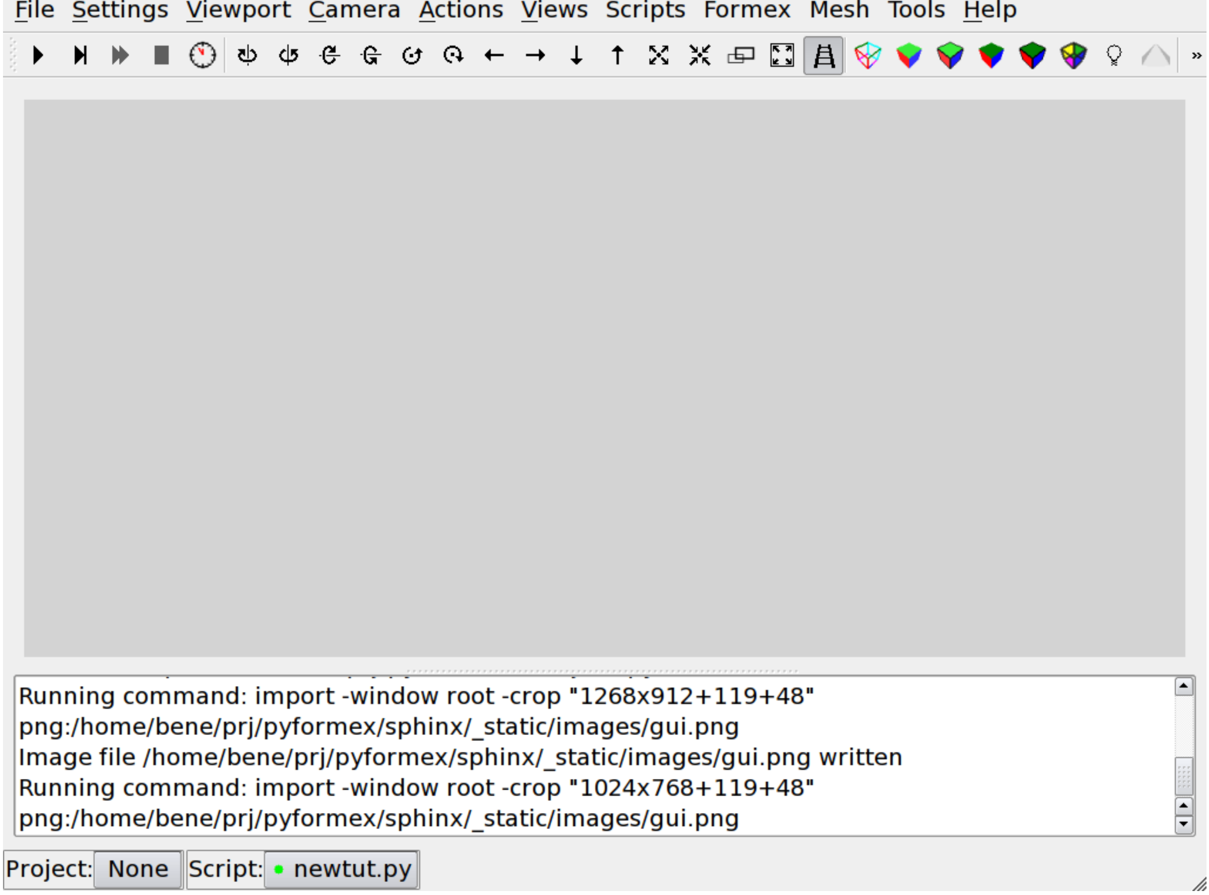


Figure 4.1: The pyFormex main window

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 import 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.

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.

4.4.3 The *File* menu

4.4.4 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.

4.4.5 The viewport menu

4.4.6 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 ove 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 workinhg of mouse rotation.

pan Pressing the middle (or simultanuous 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).

4.4.7 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 .pyformexrc 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 no 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.

4.5 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 automize 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 fist 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 obeing 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.

4.5.1 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.

4.5.2 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 explicitely 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.

4.5.3 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.

```
# pyformex script/app template
#
##
##  Copyright (C) 2011 John Doe (j.doe@somewhere.org)
##  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/.
##

"""pyFormex Script/App Template

This is a template file to show the general layout of a pyFormex
script or app.

A pyFormex script is just any simple Python source code file with
extension '.py' and is fully read and execution at once.

A pyFormex app can be a '.py' of '.pyc' file, and should define a function
'run()' to be executed by pyFormex. Also, the app should import anything that
it needs.

This template is a common structure that allows the file to be used both as
a script or as an app, with almost identical behavior.

For more details, see the user guide under the `Scripting` section.

The script starts by preference with a docstring (like this),
composed of a short first line, then a blank line and
one or more lines explaining the intention of the script.
"""
from __future__ import print_function

from gui.draw import *  # for an app we need to import explicitely

def run():
    """Main function.

    This is executed on each run.
    """
    print("This is the pyFormex template script/app")


```

```
54 # Initialization code
55
56 print("This is the initialization code of the pyFormex template script/app")
57
58
59 # The following is to make script and app behavior alike
60 if __name__ == 'draw':
61     print("Running as a script")
62     run()
63
64
65 # 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.

4.6 Modeling Geometry with pyFormex

Warning: This document still needs to be written!

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.

4.6.1 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 though [1], 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 vaned 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 modleing 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]

[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.

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

4.6.2 The Formex model

4.6.3 The Mesh model

4.6.4 Analytical models

4.7 The Canvas

4.7.1 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 dispaying 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.

4.7.2 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 elementof 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 properies 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.

4.7.3 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 decribed 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 justing 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.

4.7.4 Other canvas items

Actors

Marks

Decorations

4.7.5 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 (*nvps=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.

4.8 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.

4.8.1 Save a rendering as image

> *A picture tells a thousand words*

4.9 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.

4.9.1 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.

4.10 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 considerer 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 used 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()
```

4.10.1 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 choosen 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

  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 readible layout, offsetting each lower level dictionary two more postions 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 though 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 assigment 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' filed 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 posess 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.

4.10.2 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

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.

4.10.3 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 distincted by their kind field. General property records have kind=”, node properties haven 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.

4.10.4 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:

cload 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 mulitple 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 interprete 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.

4.10.5 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.

4.10.6 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 interprete 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 togehter 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

4.11 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.

4.11.1 The askItems functions

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

4.11.2 The input dialog

4.11.3 The user menu

4.11.4 Other widgets

4.12 pyFormex plugins

Abstract

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

4.12.1 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 comprehense 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 .

4.12.2 How to create a plugin.

4.13 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 thay can easily be found. Some of these items can even be set interactivley 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 tought it was necessary to provide already some information on how to configure. Be aware though that important changes to this system will likely occur.

4.13.1 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/pyformexrc: 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/pyformex: 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.
- /.pyformexrc: this is where the user normally stores his own default settings.
- CURRENT-DIR/.pyformex: if the current working directory from which is started contains a file named .pyformex, 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!

4.13.2 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).

4.13.3 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',myscriptsdir)]`: 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. Possioble 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.

CHAPTER

FIVE

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  # $Id: 954dbe2 on Mon Mar 25 13:11:26 2013 +0100 by Benedict Verhegghe $   *** pyformex
2  ##
3  ##  This file is part of pyFormex 0.9.1  (Tue Oct 15 21:05:25 CEST 2013)
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 2004-2013 (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.
```

```
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 frome 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 cilinder
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=lambda x: 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
```

```
        self.cell1 = cell1.translate([1.,1.,0.])
        self.cell2 = cell2.translate([-1.,-1.,0.])
        # the base pattern cell1+cell2 now has size [-2,-2]..[2,2]
        # Create the full pattern by replication
        dx = 4.
        dy = 4.
        F = (self.cell1+self.cell2).replic2(nx,ny,dx,dy)
        # fold it into a cylinder
        self.F = F.translate([0.,0.,r]).cylindrical(
            dir=[2,0,1],scale=[1.,360./(nx*dx),p/nx/dy])
        self.ny = ny

    def all(self):
        """Return the Formex with all bar elements."""
        return self.F


if __name__ == "draw":

    # show an example

    wireframe()
    reset()

    D = 10.
    L = 80.
    d = 0.2
    n = 12
    b = 30.
    res = askItems([['Diameter',D],
                    ['Length',L],
                    ['WireDiam',d],
                    ['NWires',n],
                    ['Pitch',b]])

    if not res:
        exit()

    D = float(res['Diameter'])
    L = float(res['Length'])
    d = float(res['WireDiam'])
    n = int(res['NWires'])
    if (n % 2) != 0:
        warning('Number of wires must be even!')
        exit()
    b = float(res['Pitch'])

    H = DoubleHelixStent(D,L,d,n,b).all()
    clear()
    draw(H,view='iso')

    # and save it in a lot of graphics formats
    if ack("Do you want to save this image (in lots of formats) ?"):
        for ext in [ 'bmp', 'jpg', 'pbm', 'png', 'ppm', 'xbm', 'xpm',
                     'eps', 'ps', 'pdf', 'tex' ]:
            image.save('WireStent.'+ext)

```

```
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 straigth 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*,.

Figure 5.1: A straight 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 seperate 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)

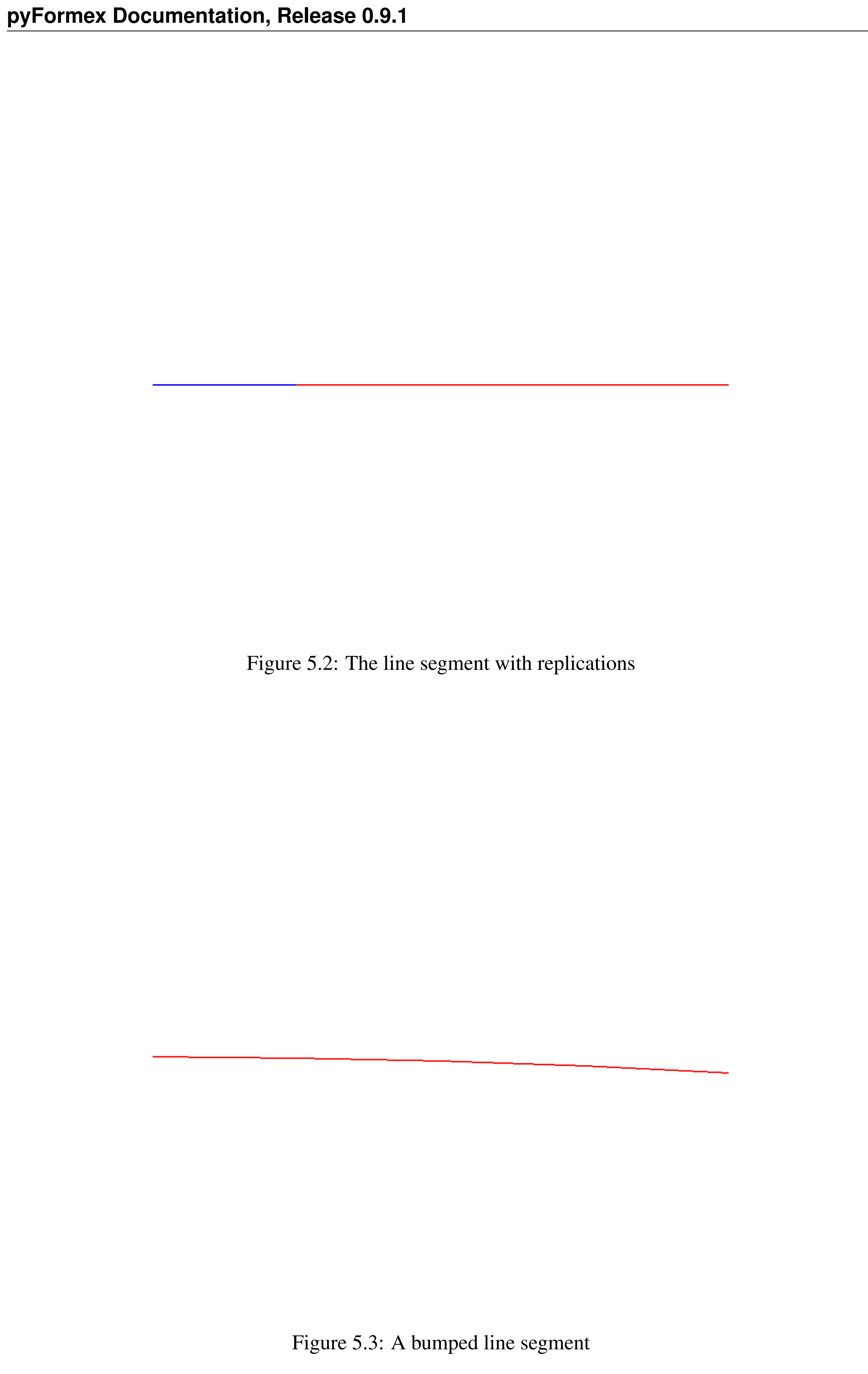


Figure 5.2: The line segment with replications

Figure 5.3: A bumped line segment

Figure 5.4: Rescaled bumped strut

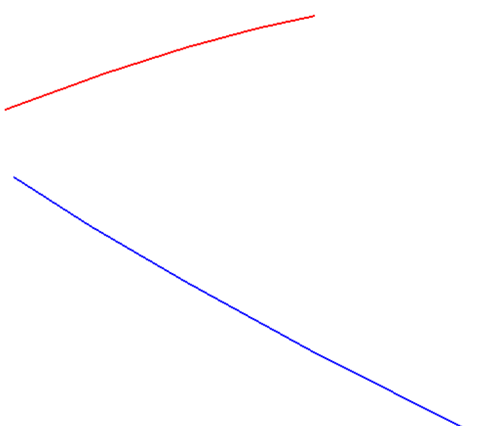


Figure 5.5: Mirrored and skewed bumped strut

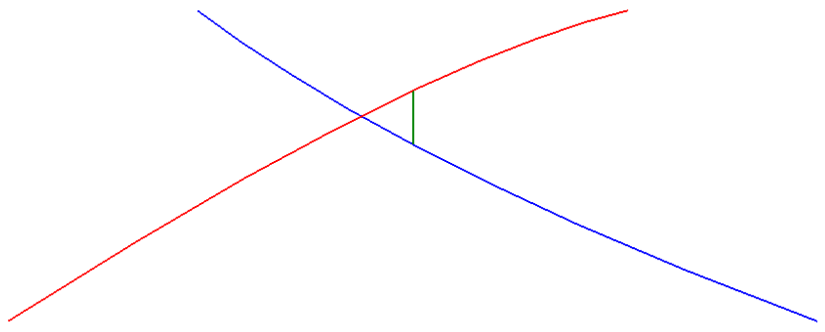


Figure 5.6: Unit cell of crossing wires and connectors

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)

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 nx and ny replications with steps dx and dy in respectively, the default X- and Y-direction.

(lines 90-93)

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)

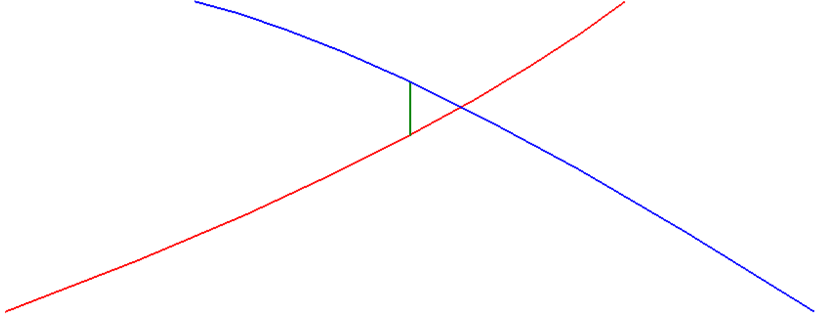


Figure 5.7: Mirrored unit cell

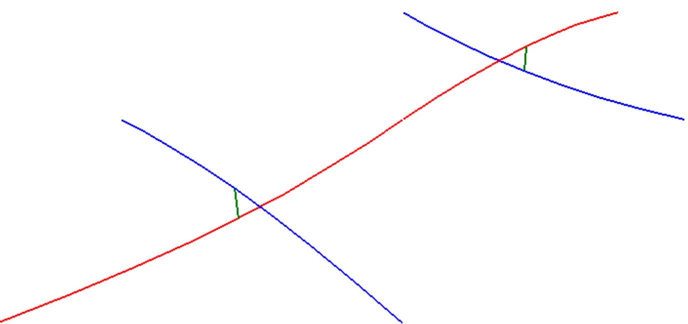


Figure 5.8: Completed base module

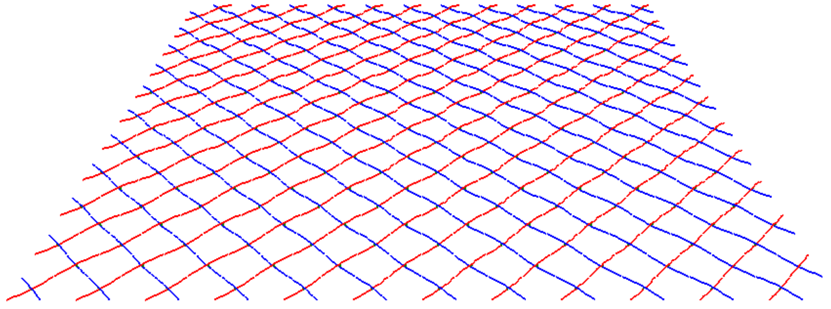


Figure 5.9: Full planar topology

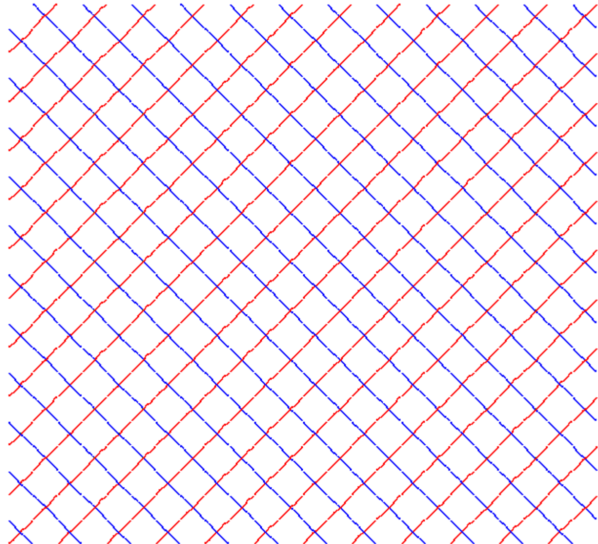


Figure 5.10: Orthogonal view of the full planar topology

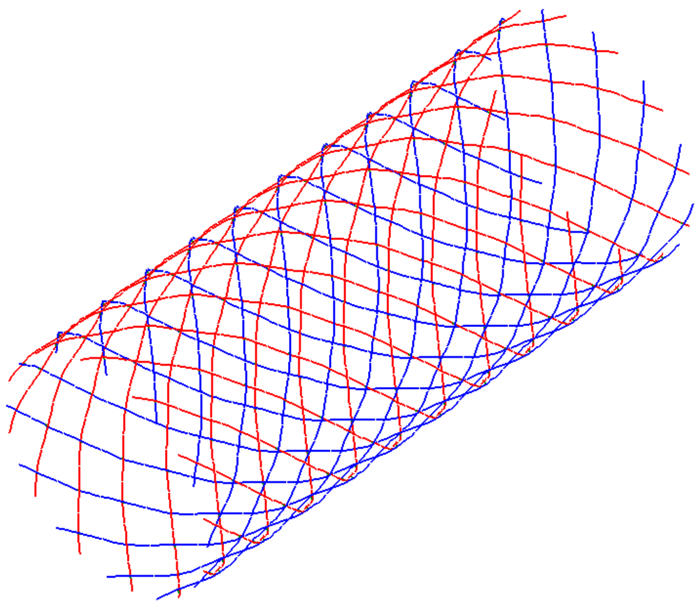


Figure 5.11: Cylindrical stent

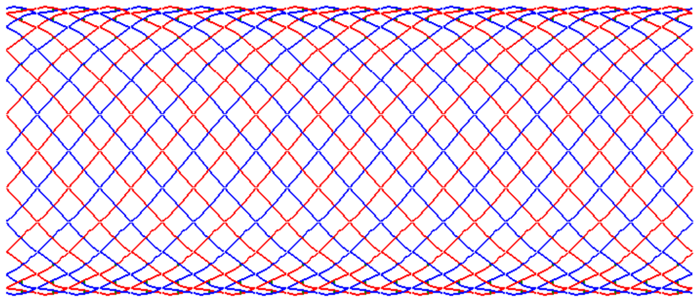


Figure 5.12: Orthogonal view of the cylindrical stent

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* : 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 a the `DoubleHelixStent` class in the (`WireStent.py`) script, it can easily be imported for e.g. this purpose.

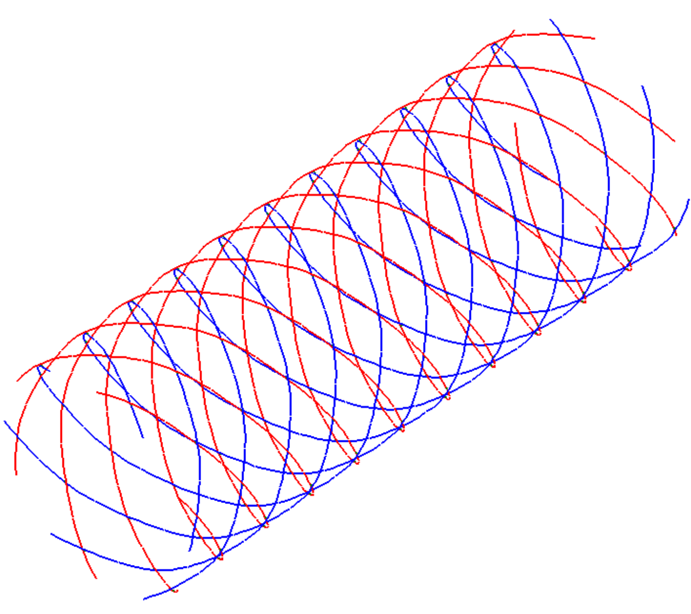


Figure 5.13: Stent variant with $De = 16, nx = 6, \beta = 25$

```
# $Id: 954dbe2 on Mon Mar 25 13:11:26 2013 +0100 by Benedict Verhegghe $   *** pyformex
##
##  This file is part of pyFormex 0.9.1  (Tue Oct 15 21:05:25 CEST 2013)
##  pyFormex is a tool for generating, manipulating and transforming 3D
##  geometrical models by sequences of mathematical operations.
##  Home page: http://pyformex.org
##  Project page:  http://savannah.nongnu.org/projects/pyformex/
##  Copyright 2004-2013 (C) Benedict Verhegghe (benedict.verhegghe@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/.
##
```

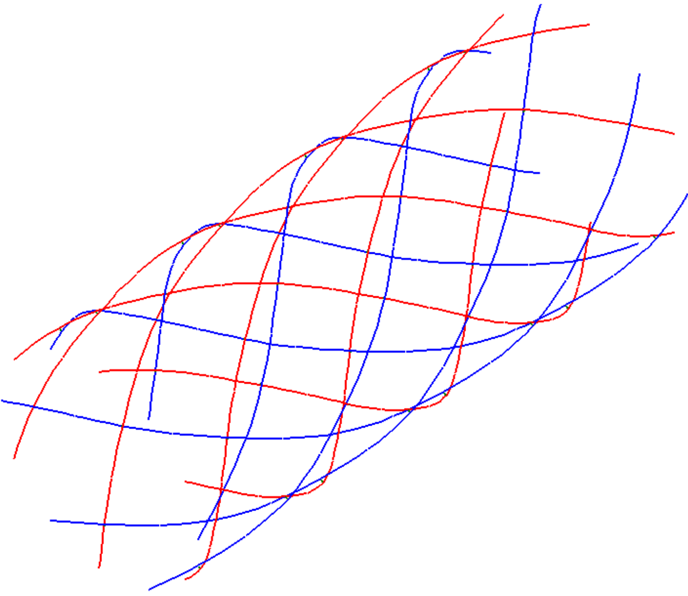


Figure 5.14: Stent variant with $De = 16, nx = 6, \beta = 50$

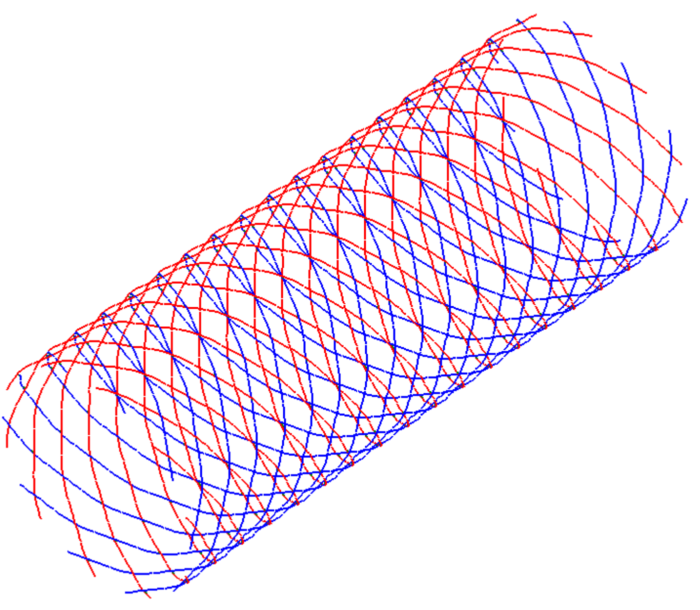


Figure 5.15: Stent variant with $De = 16, nx = 10, \beta = 25$

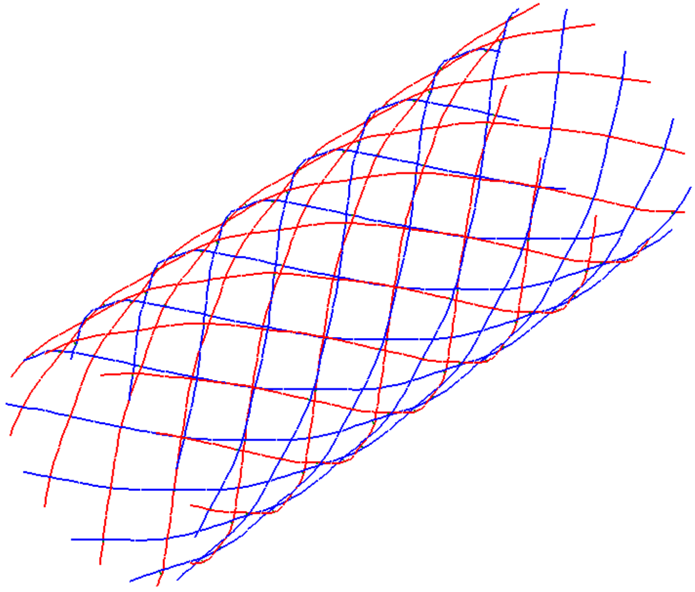


Figure 5.16: Stent variant with $De = 16, nx = 10, \beta = 50$

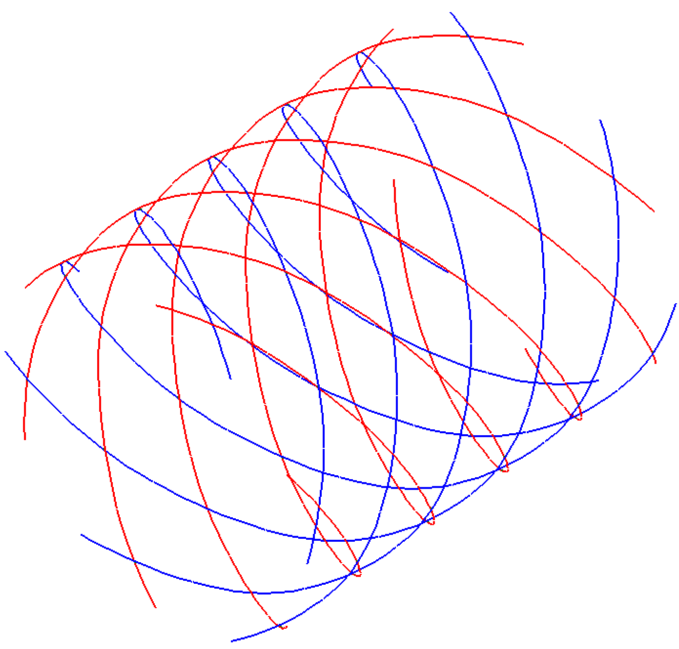


Figure 5.17: Stent variant with $De = 32, nx = 6, \beta = 25$

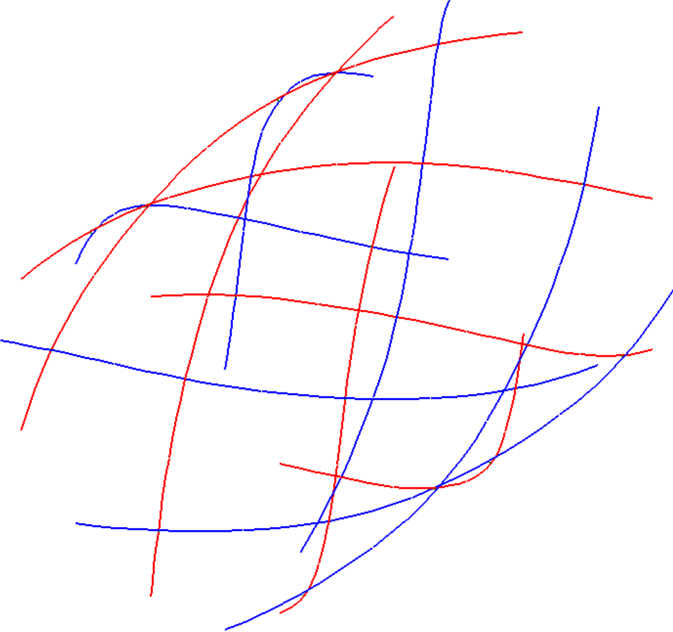


Figure 5.18: Stent variant with $De = 32, nx = 6, \beta = 50$

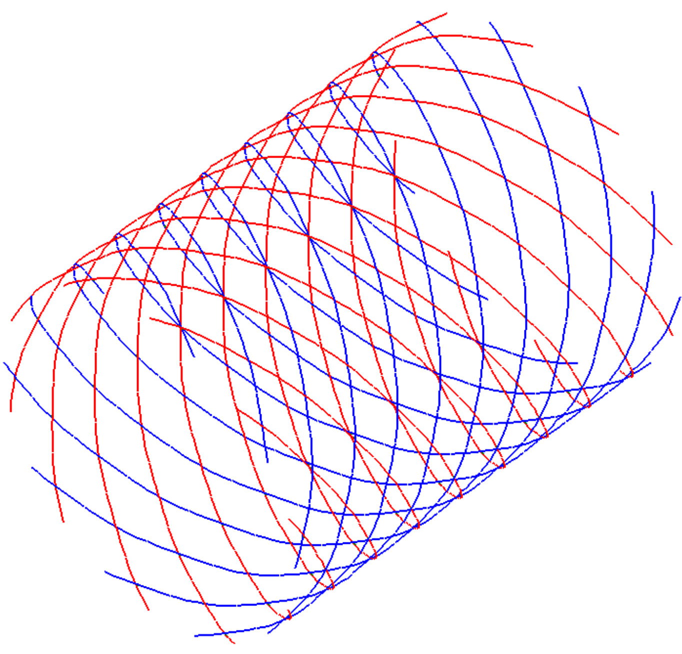


Figure 5.19: Stent variant with $De = 32, nx = 10, \beta = 25$

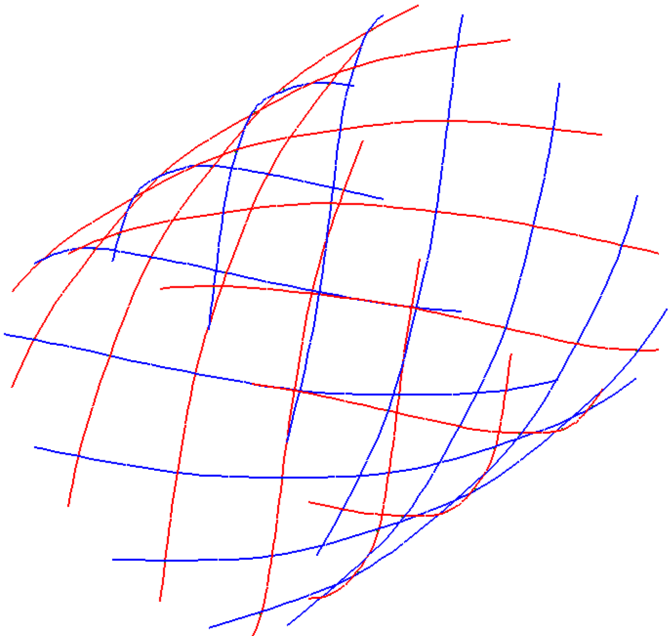


Figure 5.20: Stent variant with $De = 32, nx = 10, \beta = 50$

```
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.

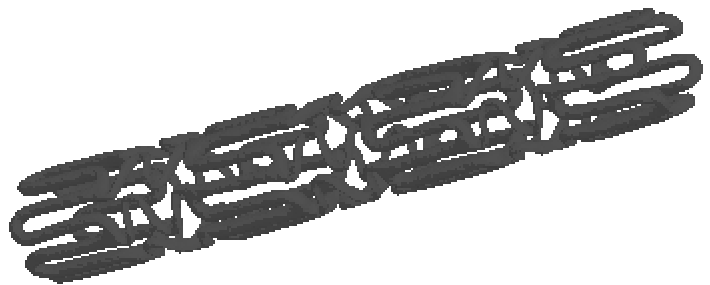


Figure 5.21: Triangulated mesh of a Cypher® stent

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.])
```

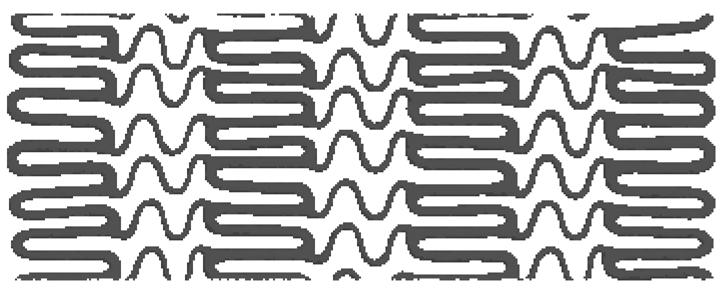


Figure 5.22: Result of the unroll operation

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*.

Figure 5.23: 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.

Figure 5.24: Circumference of a stent cell

CHAPTER

SIX

PYFORMEX REFERENCE MANUAL

Abstract

This is the reference manual for pyFormex 0.9.1. 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 explicitely 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. `Geometry`, `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`**

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

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

`Coords` is implemented as a subclass of `numpy.ndarray`, and thus inherits all its methods. The last axis of the `Coords` always has a length equal to 3. Each set of 3 values along the last axis represents a single point in 3D cartesian space. 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.

A new Coords object is created with the following syntax

```
Coords(data=None,dtyp=Float,copy=False)
```

Parameters:

- *data*: array_like of type float. The last axis should have a length of 1, 2 or 3, bu will always be expanded to 3. If no data are specified, an empty Coords with shape (0,3) is created.
- *dtyp*: the float datatype to be used. It 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. The default setting will try to use the original data if possible, e.g. if *data* is a correctly shaped and typed `numpy.ndarray`.

Example:

```
>>> Coords([1.,0.])
Coords([ 1.,  0.,  0.], dtype=float32)
```

`points``()`

Returns the `Coords` object as a simple set of points.

This reshapes the array to a 2-dimensional array, flattening the structure of the points.

`pshape``()`

Returns the shape of the `Coords` object.

This is the shape of the NumPy array with the last axis removed. The full shape of the `Coords` array can be obtained from its shape attribute.

`npoints``()`

Return the total number of points.

`ncoords``()`

Return the total number of points.

x()

Returns the X-coordinates of all points.

Returns an array with all the X-coordinates in the Coords. The returned array has the same shape as the Coords array along its first ndim-1 axes. This is equivalent with

```
asarray(self[...,0])
```

y()

Return the Y-coordinates of all points.

Returns an array with all the Y-coordinates in the Coords. The returned array has the same shape as the Coords array along its first ndim-1 axes. This is equivalent with

```
asarray(self[...,1])
```

z()

Returns the Z-coordinates of all points.

Returns an array with all the Z-coordinates in the Coords. The returned array has the same shape as the Coords array along its first ndim-1 axes. This is equivalent with

```
asarray(self[...,2])
```

bbox()

Returns the bounding box of a set of points.

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

Returns a Coords object with shape(2,3): the first point contains the minimal coordinates, the second has the maximal ones.

Example:

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

apt(*align*)

Returns an alignment point of a Coords.

Alignment point are points whose coordinates are either the minimal value, the maximal value or the middle value for the Coords. Combining the three values with the three dimensions, a Coords has in 27 (3**3) alignment points. The corner points of the bounding box are a subset of these.

The 27 points are addressed by an alignment string of three characters, one for each direction. Each character should be one of the following

- '-': use the minimal value for that coordinate,
- '+': use the minimal value for that coordinate,
- '0': use the middle value for that coordinate.

Any other character will set the corresponding coordinate to zero.

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

Example:

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

center()

Returns 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.

Example:

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

See also: centroid()

average(*wts=None*, *axis=0*)

Returns a (weighted) average of the Coords.

The average of a Coords is a Coords with one axis less than the original, obtained by averaging all the points along that axis. The weights array can either be 1-D (in which case its length must be the size along the given axis) or of the same shape as a. Weights can be specified as a 1-D array with the length of that axis, or as an array with the same shape as the Coords. The sum of the weights (along the specified axis if not 1-D) will generally be equal to 1.0. If wts=None, then all points are assumed to have a weight equal to one divided by the length of the specified axis.

Example:

```
>>> X = Coords([[[0.,0.,0.],[1.,0.,0.],[2.,0.,0.]],                    [[4.,0.,0
>>> print(X.average())
[[ 2.   0.   0.]
 [ 3.   0.   0.]
 [ 4.   0.   0.]]
>>> print(X.average(axis=1))
[[ 1.   0.   0.]
 [ 5.   0.   0.]]
>>> print(X.average(wts=[0.5,0.25,0.25],axis=1))
[[ 0.75  0.    0.  ]
 [ 4.75  0.    0.  ]]
```

centroid()

Returns the centroid of the Coords.

The centroid of a Coords is the point whose coordinates are the mean values of all points. The return value is a (3,) shaped Coords object.

Example:

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

See also: center()

sizes()

Returns the sizes of the Coords.

Returns an array with the length of the bbox along the 3 axes.

Example:

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

dsize()

Returns an estimate of the global size of the `Coords`.

This estimate is the length of the diagonal of the bbox().

Example:

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

bsphere()

Returns the diameter of the bounding sphere of the `Coords`.

The bounding sphere 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.

Example:

```
>>> print(Coords([[[0.,0.,0.],[3.,0.,0.],[0.,3.,0.]]]).bsphere())
2.12132024765
```

bboxes()

Returns the bboxes of all elements in the coords array.

The returned array has shape (...,2,3). Along the -2 axis are stored the minimal and maximal values of the Coords along that axis.

inertia(*mass=None*)

Returns inertia related quantities of the `Coords`.

Parameters:

- *mass*: float array with ncoords weight values. The default is to attribute a weight 1.0 to each point.

Returns a tuple of:

- *center*: the center of mass: shape (3,)
- *axes*: the principal axes of the inertia tensor: shape (3,3)
- *principal*: the (prinicipal) moments of inertia: shape (3,)
- *tensor*: the full inertia tensor in the global axes: shape (3,3)

distanceFromPlane(*p, n*)

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

Parameters:

- *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 float array with shape `self.pshape()` with the distance of each point to the plane through p and having normal n. Distance values are positive if the point is on the side of the plane indicated by the positive normal.

Example:

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

distanceFromLine (*p*, *n*)

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

p,n are (1,3) or (npts,3) arrays defining 1 or npts lines

Parameters:

- *p*: is a point on the line specified by 3 coordinates.
- *n*: is a vector specifying the direction of the line through p.

The return value is a [...] shaped array with the distance of each point to the line through p with direction n. All distance values are positive or zero.

Example:

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

distanceFromPoint (*p*)

Returns the distance of all points from the point p.

p is a single point specified by 3 coordinates.

The return value is a [...] shaped array with the distance of each point to point p. All distance values are positive or zero.

Example:

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

closestToPoint (*p*)

Returns the point closest to point p.

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

Returns the extreme distances from the plane p,n.

Parameters:

- *n*: the direction can be specified by a 3 component vector or by a single integer 0..2 designing one of the coordinate axes.
- *p*: is any point in space. If not specified, it is taken as the center() of the Coords.

The return value is a tuple of two float values specifying the extreme distances from the plane p,n.

directionalExtremes (*n*, *p=None*)

Returns extremal planes in the direction n.

n and *p* have the same meaning as in *directionalSize*.

The return value is a list of two points on the line (p,n), such that the planes with normal n through these points define the extremal planes of the Coords.

directionalWidth(*n*)

Returns the width of a Coords in the given direction.

The direction can be specified by a 3 component vector or by a single integer 0..2 designating one of the coordinate axes.

The return value is the thickness of the object in the direction n.

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

Flag points having coordinates between min and max.

Tests the position of the points of the `Coords` with respect to one or two 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 cooordinate axes, but a general direction may be used as well.

Parameters:

- *dir*: either a global axis number (0, 1 or 2) or a direction vector consisting of 3 floats. It specifies the direction in which the distances are measured. Default is the 0 (or x) direction.
- *min*, *max*: position of the minimum and maximum clipping planes. If *dir* was specified as an integer (0,1,2), this is a single float value corresponding with the coordinate in that axis direction. Else, it is a point in the clipping plane with normal direction *dir*. One of the two clipping planes may be left unspecified.

Returns:

A 1D integer array with same length as the number of points. For each point the value is 1 (True) if the point is above the minimum clipping plane and below the maximum clipping plane, or 0 (False) otherwise. An unspecified clipping plane corresponds with an infinitely low or high value. The return value can directly be used as an index to obtain a `Coords` with the points satisfying the test (or not). See the examples below.

Example:

```
>>> 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.]]
>>> print(x[~t])
[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 0.  2.  0.]]
```

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

Formatted printing of a `Coords` object.

The supplied format should contain 3 formatting sequences for the three coordinates of a point.

set (*f*)
Set the coordinates from those in the given array.

scale (*scale*, *dir=None*, *center=None*, *inplace=False*)
Returns a copy scaled with scale[i] in direction i.

The scale should be a list of 3 scaling factors for the 3 axis directions, or a single scaling factor. In the latter case, dir (a single axis number or a list) may be given to specify the direction(s) to scale. The default is to produce a homothetic scaling. The center of the scaling, if not specified, is the global origin. If a center is specified, the result is equivalent to:

```
self.translate(-center).scale(scale,dir).translate(center)
```

Example:

```
>>> print(Coords([1.,1.,1.]).scale(2))
[ 2.  2.  2.]
>>> print(Coords([1.,1.,1.]).scale([2,3,4]))
[ 2.  3.  4.]
```

translate (*dir*, *step=None*, *inplace=False*)
Translate a `Coords` object.

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

Parameters:

- *dir*: specifies the direction and distance of the translation. It can be either
  - an axis number (0,1,2), specifying a unit vector in the direction of one of the coordinate axes.
  - a single translation vector,
  - an array of translation vectors, compatible with the Coords shape.
- *step*: If specified, the translation vector specified by *dir* will be multiplied with this value. It is commonly used with unit *dir* vectors to set the translation distance.

Example:

```
>>> 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.]
```

centered ()
Returns a centered copy of the Coords.

Returns a Coords which is a translation thus that the center coincides with the origin. This is equivalent with:

```
self.trl(-self.center())
```

align (*alignment='—', point=[0.0, 0.0, 0.0]*)
Align a Coords on a given point.

Alignment involves a translation such that the bounding box of the Coords object becomes aligned with a given point. By default this is the origin of the global axes. The requested alignment is determined by 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.

The default alignment string `'---'` results in a translation which puts all the points in the octant with all positive coordinate values. A string `'000'` will center the object around the origin, just like the (slightly faster) `centered()` method.

See also the `coords.align()` function.

rotate (*angle, axis=2, around=None*)
Returns a copy rotated over angle around axis.

The angle is specified in degrees. The axis is either one of (0,1,2) designating the global axes, or a vector specifying an axis through the origin. If no axis is specified, rotation is around the 2(z)-axis. This is convenient for working on 2D-structures.

As a convenience, the user may also specify a 3x3 rotation matrix, in which case the function rotate(mat) is equivalent to affine(mat).

All rotations are performed around the point [0.,0.,0.], unless a rotation origin is specified in the argument 'around'.

shear (*dir, dir1, skew, inplace=False*)
Returns a copy skewed in the direction dir of plane (dir,dir1).

The coordinate dir is replaced with (dir + skew * dir1).

reflect (*dir=0, pos=0.0, inplace=False*)
Reflect the coordinates in direction dir against plane at pos.

Parameters:

- *dir*: int: direction of the reflection (default 0)
- *pos*: float: offset of the mirror plane from origin (default 0.0)
- *inplace*: boolean: change the coordinates inplace (default False)

affine (*mat, vec=None*)
Perform a general affine transformation.

Parameters:

- *mat*: a 3x3 float matrix
- *vec*: a length 3 list or array of floats

The returned object has coordinates given by `self * mat + vec`. If *mat* is a rotation matrix, than the operation performs a rigid rotation of the object plus a translation.

position (*x*, *y*)
Position an object so that points x are aligned with y.

Parameters are as for `arraytools.trfMatrix()`

cylindrical (*dir=[0, 1, 2], scale=[1.0, 1.0, 1.0], angle_spec=0.017453292519943295*)
Converts from cylindrical to cartesian after scaling.

Parameters:

- *dir*: specifies which coordinates are interpreted as resp. distance(r), angle(theta) and height(z). Default order is [r,theta,z].
- *scale*: will scale the coordinate values prior to the transformation. (scale is given in order r,theta,z).

The resulting angle is interpreted in degrees.

toCylindrical (*dir=[0, 1, 2], angle_spec=0.017453292519943295*)
Converts from cartesian to cylindrical coordinates.

Parameters:

- *dir*: specifies which coordinates axes are parallel to respectively the cylindrical axes distance(r), angle(theta) and height(z). Default order is [x,y,z].

The angle value is given in degrees.

spherical (*dir=[0, 1, 2], scale=[1.0, 1.0, 1.0], angle_spec=0.017453292519943295, colat=False*)
Converts from spherical to cartesian after scaling.

Parameters:

- *dir*: specifies which coordinates are interpreted as resp. longitude(theta), latitude(phi) and distance(r).
- *scale*: will scale the coordinate values prior to the transformation.

Angles are interpreted in degrees. Latitude, i.e. the elevation angle, is measured from equator in direction of north pole(90). South pole is -90.

If colat=True, the third coordinate is the colatitude (90-lat) instead.

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 creation of virtually any surface.

Just like with spherical(), the input coordinates are interpreted as the longitude, latitude and distance in a spherical coordinate system.

Parameters:

- *dir*: specifies which coordinates are interpreted as resp.longitude(theta), latitude(phi) and distance(r). Angles are then interpreted in degrees. Latitude, i.e. the elevation angle, is measured from equator in direction of north pole(90). South pole is -90. If colat=True, the third coordinate is the colatitude (90-lat) instead.
- *scale*: will scale the coordinate values prior to the transformation.

- *n*, *e*: parameters define exponential transformations of the north_south (latitude), resp. the east_west (longitude) coordinates. Default values of 1 result in a circle.
- *k*: adds 'eggness' to the shape: a difference between the northern and southern hemisphere. Values > 0 enlarge the southern hemishpere and shrink the northern.

toSpherical (*dir=[0, 1, 2], angle_spec=0.0174532925199432957*)
Converts from cartesian to spherical coordinates.

Parameters:

- *dir*: specifies which coordinates axes are parallel to respectively the spherical axes distance(r), longitude(theta) and latitude(phi). Latitude is the elevation angle measured from equator in direction of north pole(90). South pole is -90. Default order is [0,1,2], thus the equator plane is the (x,y)-plane.

The returned angle values are given in degrees.

bump1 (*dir*, *a*, *func*, *dist*)
Returns a `Coords` with a one-dimensional bump.

Parameters:

- *dir*: specifies the axis of the modified coordinates;
- *a*: is the point that forces the bumping;
- *dist*: specifies the direction in which the distance is measured;
- *func*: is a function that calculates the bump intensity from distance and should be such that `func(0) != 0`.

bump2 (*dir*, *a*, *func*)
Returns a `Coords` with a two-dimensional bump.

Parameters:

- *dir*: specifies the axis of the modified coordinates;
- *a*: is the point that forces the bumping;
- *func*: is a function that calculates the bump intensity from distance !! func(0) should be different from 0.

bump (*dir*, *a*, *func*, *dist=None*)
Returns a `Coords` with a bump.

A bump is a modification of a set of coordinates by a non-matching point. It can produce various effects, but one of the most common uses is to force a surface to be indented by some point.

Parameters:

- *dir*: specifies the axis of the modified coordinates;
- *a*: is the point that forces the bumping;
- *func*: is a function that calculates the bump intensity from distance (!! func(0) should be different from 0)
- *dist*: is the direction in which the distance is measured : this can be one of the axes, or a list of one or more axes. If only 1 axis is specified, the effect is like function bump1 If 2 axes are specified, the effect is like bump2 This function can take 3 axes however.

Default value is the set of 3 axes minus the direction of modification. This function is then equivalent to bump2.

flare (*xf, f, dir=[0, 2], end=0, exp=1.0*)
Create a flare at the end of a `Coords` block.

The flare extends over a distance `xf` at the start (`end=0`) or end (`end=1`) in direction `dir[0]` of the coords block, and has a maximum amplitude of `f` in the `dir[1]` direction.

map (*func*)
Map a `Coords` by a 3-D function.

This is one of the versatile mapping functions.

Parameters:

- *func*: is a numerical function which takes three arguments and produces a list of three output values. The coordinates [x,y,z] will be replaced by func(x,y,z).

The function must be applicable to arrays, so it should only include numerical operations and functions understood by the numpy module. This method is one of several mapping methods. See also map1 and mapd.

Example:

```
>>> 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:

- *func*: is a numerical function which takes one argument and produces one result. The coordinate dir will be replaced by func(coord[x]). If no x is specified, x is taken equal to dir.

The function must be applicable on arrays, so it should only include numerical operations and functions understood by the numpy module. This method is one of several mapping methods. See also map and mapd.

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*: 0, 1 or 2: the coordinate that will be replaced with `func(d)`, where *d* is calculated as the distance to *point*.
- *func*: a numerical function which takes one float argument and produce one float result. The function must be applicable on arrays, so it should only include numerical operations and functions understood by the `numpy` module.
- *point*: the point to where the distance *d* is computed.
- *dist*: a list of coordinate directions that are used to compute the distances *d*. It can also be a single coordinate direction. The default is to use 3-D distances.

This method is one of several mapping methods. See also `map3()` and `map1()`.

Example:

```
E.mapd(2,lambda d:sqrt(10**2-d**2),E.center(),[0,1])
```

maps E on a sphere with radius 10.

egg (*k*)
Maps the coordinates to an egg-shape

replace (*i, j, other=None*)
Replace the coordinates along the axes i by those along j.

i and j are lists of axis numbers or single axis numbers. replace ([0,1,2],[1,2,0]) will roll the axes by 1. replace ([0,1],[1,0]) will swap axes 0 and 1. An optionally third argument may specify another `Coords` object to take the coordinates from. It should have the same dimensions.

swapAxes (*i, j*)
Swap coordinate axes i and j.

Beware! This is different from numpy's swapaxes() method !

rollAxes (*n=1*)
Roll the axes over the given amount.

Default is 1, thus axis 0 becomes the new 1 axis, 1 becomes 2 and 2 becomes 0.

projectOnPlane (*n=2, P=[0.0, 0.0, 0.0]*)
Project a `Coords` on a plane (or planes).

Parameters:

- *n*: the normal direction to the plane. It can be specified either by a list of three floats, or by a single integer (0, 1 or 2) to use one of the global axes.
- *P*: a point on the plane, by default the global origin. If an int, the plane is the coordinate plane perpendicular to the

---

Note: For planes parallel to a coordinate plane, it is far more efficient to specify the normal by an axis number than by a three component vector.

Note: This method will also work if any or both of P and n have a shape (ncoords,3), where ncoords is the total number of points in the `Coords`. This allows to project each point on an individual plane.

Returns a `Coords` with same shape as original, with all the points projected on the specified plane(s).

projectOnSphere (*radius=1.0, center=[0.0, 0.0, 0.0]*)
Project `Coords` on a sphere.

The default sphere is a unit sphere at the origin. The center of the sphere should not be part of the `Coords`.

projectOnCylinder (*radius=1.0, dir=0, center=[0.0, 0.0, 0.0]*)
Project `Coords` on a cylinder with axis parallel to a global axis.

The default cylinder has its axis along the x-axis and a unit radius. No points of the `Coords` should belong to the axis..

projectOnSurface (*S*, *dir=0*, *missing='error'*, *return_indices=False*)
Project the Coords on a triangulated surface.

The points of the Coords are projected in the specified direction dir onto the surface S.

Parameters:

- *S*: TriSurface: any triangulated surface
- *dir*: int or vector: specifies the direction of the projection
- *missing*: float value or a string. Specifies a distance to set the position of the projection point in cases where the projective line does not cut the surface. The sign of the distance is taken into account. If specified as a string, it should be one of the strings 'c', 'f', or 'm', possibly preceded by a '+' or '-'. The distance will then be taken equal to the closest, the furthest, or the mean distance of a point to its projection, and applied in positive or negative direction as specified. Any other value of missing will result in an error if some point does not have any projection. An error will also be raised if not a single point projection intersects the surface.
- *return_indices*: if True, also returns an index of the points that have a projection on the surface.

Returns:

A Coords with the same shape as the input. If *return_indices* is True, also returns an index of the points that have a projection on the surface. This index is a sequential one, no matter what the shape of the input Coords is.

isopar (*eltype*, *coords*, *oldcoords*)
Perform an isoparametric transformation on a Coords.

This is a convenience method to transform a Coords object through an isoparametric transformation. It is equivalent to:

```
Isopar(eltype,coords,oldcoords).transform(self)
```

See `plugins.isopar` for more details.

transformCS (*currentCS*, *initialCS=None*)
Perform a coordinate system transformation on the Coords.

This method transforms the Coords object by the transformation that turns the initial coordinate system into the current coordinate system.

currentCS and initialCS are (4,3) shaped Coords instances defining a coordinate system as described in `CoordinateSystem`. If initialCS is None, the global (x,y,z) axes are used.

E.g. the default initialCS and currentCS equal to:

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

result in a rotation of 90 degrees around the z-axis.

This is a convenience function equivalent to:

```
self.isopar('tet4',currentCS,initialCS)
```

addNoise (*rsize=0.05*, *asize=0.0*)
Add random noise to a Coords.

A random amount is added to eacho individual coordinate in the Coords. The difference of any coordinate from its original value will not be r than `asize+rsize*self.sizes().max()`. The default is to set it to 0.05 times the geometrical size of the structure.

replicate (*n*, *dir=0*, *step=None*)
Replicate a Coords n times with fixed step in any direction.

Returns a Coords object with shape *(n,) + self.shape*, thus having an extra first axis. Each component along the axis 0 is equal to the previous component translated over *(dir,step)*, where *dir* and *step* are interpreted just like in the `translate()` method. The first component along the axis 0 is identical to the original Coords.

split ()
Split the coordinate array in blocks along first axis.

The result is a sequence of arrays with shape self.shape[1:]. Raises an error if self.ndim < 2.

sort (*order=[0, 1, 2]*)
Sort points in the specified order of their coordinates.

The points are sorted based on their coordinate values. There is a maximum number of points (above 2 million) that can be sorted. If you need to to sort more, first split up your data according to the first axis.

Parameters:

- *order*: permutation of [0,1,2], specifying the order in which the subsequent axes are used to sort the points.

Returns:

An int array which is a permutation of range(self.npoints()). If taken in the specified order, it is guaranteed that no point can have a coordinate that is larger that the corresponding coordinate of the next point.

boxes (*ppb=1*, *shift=0.5*, *minsize=1e-05*)
Create a grid of equally sized boxes spanning the points x.

A regular 3D grid of equally sized boxes is created spanning all the points *x*. The size, position and number of boxes are determined from the specified parameters.

Parameters:

- *ppb*: int: mean 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): a 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: minimum absolute size of the boxes (same in each coordinate direction).

Returns a tuple of:

- *ox*: float array (3): minimal coordinates of the box grid,
- *dx*: float array (3): box size in the three axis directions,
- *nx*: in array (3): number of boxes in each of the coordinate directions.

fuse (*ppb=1*, *shift=0.5*, *rtol=1e-05*, *atol=1e-05*, *repeat=True*, *nodesperbox=None*)

Find (almost) identical nodes and return a compressed set.

This method finds the points that are very close and replaces them with a single point.

Returns a tuple of two arrays:

- *coords*: the unique points as a `Coords` object with shape (npoints,3),
- *elems*: an int array holding an index in the unique coordinates array for each of the original nodes. The shape of the index array is equal to the shape of the input coords array minus the last dimension (also given by self.pshape()).

The procedure works by first dividing the 3D space in a number of equally sized boxes, with a mean population of ppb. The boxes are numbered in the 3 directions and a unique integer scalar is computed, that is then used to sort the nodes. Then only nodes inside the same box are compared on almost equal coordinates, using the numpy allclose() function. Two coordinates are considered close if they are within a relative tolerance rtol or absolute tolerance atol. See numpy for detail. The default atol is set larger than in numpy, because pyformex typically runs with single precision. Close nodes are replaced by a single one.

Running the procedure once does not guarantee to find 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.

match (*coords*, ***kargs*)

Match points form another Coords object.

This method finds the points from *coords* that coincide with (or are very close to) points of *self*.

Parameters:

- *coords*: a Coords object
- ***kargs*: keyword arguments that you want to pass to the `fuse()` method.

This method works by concatenating the serialized point sets of both Coords and then fusing them.

Returns:

- *matches*: an Int array with shape (nmatches,2)
- *coords*: a Coords with the fused coordinate set
- *index*: an index with the position of each of the serialized points of the concatenation in the fused coordinate set. To find the index of the points of the orginal coordinate sets, split this index at the position self.npoints() and reshape the resulting parts to *self.pshape()*, resp. *coords.pshape()*.

append (*coords*)

Append coords to a Coords object.

The appended coords should have matching dimensions in all but the first axis.

Returns the concatenated Coords object, without changing the current.

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.

classmethod concatenate (*clas*, *L*, *axis=0*)
Concatenate a list of `Coords` object.

All `Coords` object in the list L should have the same shape except for the length of the specified axis. This function is equivalent to the numpy concatenate, but makes sure the result is a `Coords` object,and the default axis is the first one instead of the last.

The result is at least a 2D array, even when the list contains a single Coords with a single point.

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

classmethod fromstring (*clas*, *fil*, *sep=' '*, *ndim=3*, *count=-1*)
Create a `Coords` object with data from a string.

This convenience function uses the `numpy.fromstring()` function to read coordinates from a string.

Parameters:

- *fil*: a string containing a single sequence of float numbers separated by whitespace and a possible separator string.
- *sep*: the separator used between the coordinates. If not a space, all extra whitespace is ignored.
- *ndim*: 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*: total number of coordinates to read. This should be a multiple of 3. The default is to read all the coordinates in the string. count can be used to force an error condition if the string does not contain the expected number of values.

The return value is Coords object.

classmethod fromfile (*clas*, *fil*, ***kargs*)
Read a `Coords` from file.

This convenience function uses the numpy fromfile function to read the coordinates from file. You just have to make sure that the coordinates are read in order (X,Y,Z) for subsequent points, and that the total number of coordinates read is a multiple of 3.

interpolate (*X*, *div*)
Create interpolations between two `Coords`.

Parameters:

- *X*: a `Coords` with same shape as *self*.
- *div*: a list of floating point values, or an int. If an int is specified, a list with (div+1) values for *div* is created by dividing the interval [0..1] into *div* equal distances.

Returns:

A `Coords` with an extra (first) axis, containing the concatenation of the interpolations of *self* and *X* at all values in *div*. Its shape is (n,) + self.shape, where n is the number of values in *div*.

An interpolation of F and G at value v is a `Coords` H where each coordinate Hijk is obtained from: Fijk = Fijk + v * (Gijk-Fijk). Thus, X.interpolate(Y,[0.,0.5,1.0]) will contain all points of X and Y and all points with mean coordinates between those of X and Y.

F.interpolate(G,n) is equivalent with F.interpolate(G,arange(0,n+1)/float(n))

rot (*angle*, *axis=2*, *around=None*)

Returns a copy rotated over angle around axis.

The angle is specified in degrees. The axis is either one of (0,1,2) designating the global axes, or a vector specifying an axis through the origin. If no axis is specified, rotation is around the 2(z)-axis. This is convenient for working on 2D-structures.

As a convenience, the user may also specify a 3x3 rotation matrix, in which case the function rotate(mat) is equivalent to affine(mat).

All rotations are performed around the point [0.,0.,0.], unless a rotation origin is specified in the argument 'around'.

trl (*dir*, *step=None*, *inplace=False*)

Translate a `Coords` object.

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

Parameters:

- *dir*: specifies the direction and distance of the translation. It can be either
  - an axis number (0,1,2), specifying a unit vector in the direction of one of the coordinate axes.
  - a single translation vector,
  - an array of translation vectors, compatible with the Coords shape.
- *step*: If specified, the translation vector specified by *dir* will be multiplied with this value. It is commonly used with unit *dir* vectors to set the translation distance.

Example:

```
>>> 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.]
```

rep (*n*, *dir=0*, *step=None*)
Replicate a Coords n times with fixed step in any direction.

Returns a Coords object with shape *(n,) + self.shape*, thus having an extra first axis. Each component along the axis 0 is equal to the previous component translated over *(dir,step)*, where *dir* and *step* are interpreted just like in the `translate()` method. The first component along the axis 0 is identical to the original Coords.

Functions defined in module coords

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. Objects that do not have a `bbox()` method or whose `bbox()` method returns invalid values, are ignored.

Parameters:

- *objects*: a list of objects (which should probably have the method `bbox()`).

Returns:

A Coords object with two points: the first contains the minimal coordinate values, the second has the maximal ones of the overall bounding box.

Example:

```
>>> from formex import *
>>> bbox([Coords([-1.,1.,0.]),Formex('1:5')])
Coords([[-1.,  0.,  0.],
        [ 1.,  1.,  0.]], dtype=float32)
```

coords.**bboxIntersection** (*A*, *B*)
Compute the intersection of the bounding box of two objects.

A and B are objects having a bbox method. The intersection of the two bounding boxes is returned in boox format.

coords.**testBbox** (*A*, *bb*, *dirs=[0, 1, 2]*, *nodes='any'*)
Test which part of A is inside a given bbox, applied in directions dirs.

Parameters:

- *A*: is any object having bbox and a test method (Formex, Mesh).
- *bb*: is a bounding box, i.e. a (2,3) shape float array.
- *dirs*: is a list of the three coordinate axes or a subset thereof.
- *nodes*: has the same meaning as in Formex.test and Mesh.test.

The result is a bool array flagging the elements that are inside the given bounding box.

coords.**origin** ()
Return a single point with coordinates [0.,0.,0.].

Returns a `Coords` object with shape(3,) holding three zero coordinates.

coords.**pattern** (*s*, *aslist=False*)
Return a series of points lying on a regular grid.

This function creates a series of points that lie on a regular grid with unit step. These points are created from a string input, interpreting each character as a code specifying how to move to the next point. The start position is always the origin (0.,0.,0.).

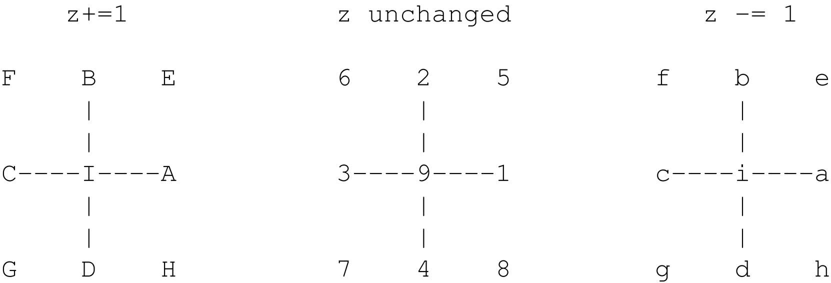
Currently the following codes are defined:

- 0: goto origin (0.,0.,0.)
- 1..8: move in the x,y plane
- 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 'ABCDEFGHI', resp. 'abcdefghi', correspond with '123456789' with an extra z +/-= 1. This gives the following schema:



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.

Parameters:

- *s*: string: with the characters generating subsequent points.
- *aslist*: bool: if True, the points are returned as lists of integer coordinates instead of a `Coords` object.

Returns a `Coords` with the generated points (default) or a list of tuples with 3 integer coordinates (if *aslist* is True).

Example:

```
>>> print(pattern('0123'))
[[ 0.  0.  0.]
 [ 1.  0.  0.]
 [ 1.  1.  0.]
 [ 0.  1.  0.]]
```

`coords.`**`xpattern`**(*s*, *nplex=1*)

Create a Coords object from a string pattern.

This is like pattern, but allows grouping the points into elements. First, the string is expanded to a list of points by calling pattern(s). Then the resulting list of points is transformed in a 2D table of points where each row has the length *nplex*.

If the number of points produced by *s* is not a multiple of *nplex*, an error is raised.

Example:

```
>>> print(xpattern('.12.34',3))
[[[ 0.  0.  0.]
  [ 1.  0.  0.]
  [ 1.  1.  0.]]

 [[ 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.

L is a list of geometrical objects (Coords or Geometry or subclasses thereof) and thus having an appropriate `align` method. align is a string of three characters, one for each coordinate direction, defining how the subsequent objects have to be aligned in that direction:

- `-` : 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 item

E.g., the string `'|--'` will juxtapose the objects in the x-direction, while aligning them on their minimal coordinates in the y- and z- direction.

An offset may be specified to create a space between the object, instead of juxtaposing them.

Returns: a list with the aligned objects.

`coords.`**`sweepCoords`**(*self, path, origin=[0.0, 0.0, 0.0], normal=0, upvector=2, avgdir=False, enddir=None, scalex=None, scaley=None, scalez=None*)

Sweep a Coords object along a path, returning a series of copies.

origin and normal define the local path position and direction on the mesh.

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 explicitely set the the directions for ALL the points of the path

Missing end directions can explicitely 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.

At each point of the curve, the original Coords object can be scaled in x and y direction by specifying scalex and scaley. The number of values specified in scalex and scaly should be equal to the number of points on the curve.

The return value is a sequence of the transformed Coords objects.

6.1.2 `formex` — Formex algebra in Python

This module defines the `Formex` class, which is the major class for representing geometry in pyFormex. The Formex class implements most functionality of Formex algebra in a consistent and easy to understand 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 the 3D space, that are structured into a set of elements all having the same number of points (e.g. a collection of line segments or a collection of triangles.)

The Formex basically contains (in its *coords* attribute) a `Coords` object, which is a Float type array with 3 axes (numbered 0,1,2). A scalar element of this array represents a coordinate.

A row along the last axis (2) is a set of 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 is always 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, a method exists to extract only a limited set of coordinates from the results, permitting to return to a 2-D environment

A plane along the axes 2 and 1 is a set of points or 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.

Finally, the whole Formex represents a collection of such elements.

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.

The Formex data can be initialized by another Formex, by a 2D or 3D coordinate list, 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 `coords.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 als 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('1:127')
```

The Formex constructor takes two optional arguments: prop and eltype. If a prop argument is specified, the setProp() function will be called to assign the specified properties. eltype can be used to specify a non-default element type. The default element type is derived from the plexitude as follows: 1 = point, 2 = line segment, 3 = triangle, 4 or more is a polygon. Specifying `eltype = 'tet4'` will e.g. interprete 4 point elements as a tetraeder.

Because the `Formex` class is derived from `Geometry`, the following `Formex` methods exist and return the value of the same method applied on the `coords` attribute: *x*, *y*, *z*, *bbox*, *center*, *centroid*, *sizes*, *dsize*, *bsphere*, *distanceFromPlane*, *distanceFromLine*, *distanceFromPoint*, *directionalSize*, *directionalWidth*, *directionalExtremes*, *__str__*. Refer to the correponding `Coords` method for their usage.

Also, the following `Coords` transformation methods can be directly applied to a `Formex` object or a derived class object. The return value is a new object identical to the original, except for the coordinates, which will have been transformed by the specified method. Refer to the corre- ponding `Coords` method for the usage of these methods: *scale*, *translate*, *rotate*, *shear*, *reflect*, *affine*, *cylindrical*, *hyperCylindrical*, *toCylindrical*, *spherical*, *superSpherical*, *toSpherical*, *bump*, *bump1*, *bump2*, *flare*, *map*, *map1*, *mapd*, *newmap*, *replace*, *swapAxes*, *rollAxes*, *projectOnSphere*, *projectOnCylinder*, *rot*, *trl*.

element (*i*)
    Return element i of the Formex

point (*i*, *j*)
    Return point j of element i

coord (*i*, *j*, *k*)
    Return coord k of point j of element i

nelems ()
    Return the number of elements in the formex.

nplex ()
    Return the number of points per element.

    Examples:

    1.unconnected points,

    2.straight line elements,

    3.triangles or quadratic line elements,

    4.tetraeders or quadrilaterals or cubic line elements.

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.

npoints()
Return the number of points in the formex.

This is the product of the number of elements in the formex with the number of nodes per element.

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, an equal to 2 for any higher plexitude (since the default is to interprete a higher plexitude as a polygon).

view()
Return the Formex coordinates as a numpy array (ndarray).

Since the ndarray object has a method view() returning a view on the ndarray, this method allows writing code that works with both Formex and ndarray instances. The results is always an ndarray.

getProp(*index=None*)
Return the property numbers of the element in index

maxProp()
Return the highest property value used, or None

propSet()
Return a list with unique property values on this Formex.

centroids()
Return the centroids of all elements of the Formex.

The centroid of an element is the point whose coordinates are the mean values of all points of the element. The return value is a Coords object with nelems points.

fuse(*repeat=True*, *ppb=1*, *rtol=1e-05*, *atol=None*)
Return a tuple of nodal coordinates and element connectivity.

A tuple of two arrays is returned. The first is float array with the coordinates of the unique nodes of the Formex. The second is an integer array with the node numbers connected by each element. The elements come in the same order as they are in the Formex, but the order of the nodes is unspecified. By the way, the reverse operation of `coords,elems=fuse(F)` is accomplished by `F=Formex(coords[elems])`

There is a (very small) probability that two very close nodes are not equivalenced by this procedure. Use it multiple times with different parameters to check. You can also set the rtol/atol parameters to influence the equivalence checking of two points. The default settting for atol is rtol * self.dsize()

toMesh(**args*, ***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, and elements are defined by a connectivity table with integers pointing to the corresponding vertex.

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.

The conversion will only work if the Formex represents a surface and its elements are triangles or quadrilaterals.

Returns a TriSurface if the conversion succeeds, else an error is raised. If the plexitude of the Formex is 3, the returned TriSurface is equivalent with the Formex.

info()
Return formatted information about a Formex.

classmethod **point2str**(*clas*, *point*)
Return a string representation of a point

classmethod **element2str**(*clas*, *elem*)
Return a string representation of an element

asFormex()
Return string representation of a Formex as in Formian.

Coordinates are separated by commas, points are separated by semicolons and grouped between brackets, elements are separated by commas and grouped between braces:

```
>>> F = Formex([[[1,0],[0,1]],[[0,1],[1,2]]])
>>> print(F)
{[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.

The string representation as done by asFormex() is followed by the words “with prop” and a list of the properties.

asArray()
Return string representation as a numpy array.

classmethod **setPrintFunction**(*clas*, *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 its own formatting method. This is a class method. It should be used asfollows: Formex.setPrintFunction(Formex.asArray).

append (*F*)

Append the members of Formex F to this one.

This function changes the original one! Use __add__ if you want to get a copy with the sum.

```
>>> F = Formex([[[1.0,1.0,1.0]]])
>>> G = F.append(F)
>>> print(F)
{[1.0,1.0,1.0], [1.0,1.0,1.0]}
```

classmethod concatenate (*clas*, *Flist*)

Concatenate all Formices in Flist.

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.

This is a class method, not an instance method!

```
>>> F = Formex([[[1,2,3]]],1)
>>> print(Formex.concatenate([F,F,F]))
{[1.0,2.0,3.0], [1.0,2.0,3.0], [1.0,2.0,3.0]}
```

Formex.concatenate([F,G,H]) is functionally equivalent with F+G+H. The latter is simpler to write for a list with a few elements. If the list becomes large, or the number of items in the list is not fixed, the concatenate method is easier (and faster). We made it a class method and not a global function, because that would interfere with NumPy's own concatenate function.

select (*idx*)

Return a Formex with only the elements selected by the parameter.

The parameter *idx* can be

- a single element number
- a list, or array, of element numbers
- a bool array of length self.nelems(), where True values flag the elements to be selected

See `cselect()` for the complementary operation.

cselect (*idx*)

Return a Formex without the elements selected by the parameter.

The parameter *idx* can be

- a single element number
- a list, or array, of element numbers
- a bool array of length self.nelems(), where True values flag the elements to be selected

This is the complementary operation of `select()`

selectNodes (*idx*)

Return a Formex which holds only some nodes of the parent.

idx is a list of node numbers to select. Thus, 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]) The returned Formex inherits the property of its parent.

asPoints()
Return a Formex containing only the points.

This is obviously a Formex with plexitude 1. It holds the same data as the original Formex, but in another shape: the number of points per element is 1, and the number of elements is equal to the total number of points. The properties are not copied over, since they will usually not make any sense.

The points() method returns the same data, but as a Coords object with a simple list of points.

remove(*F*)
Return a Formex where the elements in F have been removed.

This is also the subtraction of the current Formex with F. Elements are only removed if they have the same nodes in the same order. This is a slow operation: for large structures, you should avoid it where possible.

whereProp(*val*)
Return the numbers of the elements with property val.

val is either a single integer, or a list/array of integers. The return value is an array holding all the numbers of all the elements that have the property val, resp. one of the values in val.

If the Formex has no properties, a empty array is returned.

withProp(*val*)
Return a Formex which holds only the elements with property val.

val is either a single integer, or a list/array of integers. The return value is a Formex holding all the elements that have the property val, resp. one of the values in val. The returned Formex inherits the matching properties.

If the Formex has no properties, a copy with all elements is returned.

elbbox()
Return a Formex where each element is replaced by its bbox.

The returned Formex has two points for each element: the two corners of the bbox with the minimal and maximal coordinates.

removeDuplicate(*permutations=True*, *rtol=0.0001*, *atol=1e-06*)
Return a Formex which holds only the unique elements.

Two elements are considered equal when all its points are (nearly) coincident. By default any permutation of point order is also allowed.

Two coordinate value are considered equal if they are both small compared to atol or if their difference divided by the second value is small compared to rtol.

If permutations is set False, two elements are not considered equal if one's points are a permutation of the other's.

unique(*permutations=True*, *rtol=0.0001*, *atol=1e-06*)
Return a Formex which holds only the unique elements.

Two elements are considered equal when all its points are (nearly) coincident. By default any permutation of point order is also allowed.

Two coordinate value are considered equal if they are both small compared to atol or if their difference divided by the second value is small compared to rtol.

If permutations is set False, two elements are not considered equal if one's points are a permutation of the other's.

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 Formex 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 Formex.

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 compaitble 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.

clip (*t*)
Return a Formex with all the elements where t>0.

t should be a 1-D integer array with length equal to the number of elements of the formex. The resulting Formex will contain all elements where t > 0. This is a convenience function for the user, equivalent to F.select(t>0).

cclip (*t*)
This is the complement of clip, returning a Formex where t<=0.

circulize (*angle*)
Transform a linear sector into a circular one.

A sector of the (0,1) plane with given angle, starting from the 0 axis, is transformed as follows: points on the sector borders remain in place. Points inside the sector are projected from the center on the circle through the intersection points of the sector border axes and the line through the point and perpendicular to the bisector of the angle. See Diamatic example.

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 Pi/6. This function is especially suited to create circular domains where all bars have nearly same length. See the Diamatic example.

shrink (*factor*)
Shrinks each element with respect to its own center.

Each element is scaled with the given factor in a local coordinate system with origin at the element center. The element center is the mean of all its nodes. The shrink operation

is typically used (with a factor around 0.9) in wireframe draw mode to show all elements disconnected. A factor above 1.0 will grow the elements.

reverse()

Return a Formex where all elements have been reversed.

Reversing an element means reversing the order of its points. This is equivalent to:

```
self.selectNodes(arange(self.nplex()-1,-1,-1))
```

mirror(*dir=2*, *pos=0*, *keep_orig=True*)

Reflect a Formex in one of the coordinate directions

This method behaves like reflect(), but adds the reflected part to the original. Setting keep_orig=False makes it behave just like reflect().

replicate(*n*, *dir=0*, *step=1.0*)

Replicate a Formex n times with fixed step in any direction.

Returns a Formex which is the concatenation of n copies, where each copy is equal to the previous one translated over *(dir,step)*, where *dir* and *step* are interpreted just like in the `translate()` method. The first of the copies is equal to the original.

rep(*n*, *dir=None*, *step=None*)

Like replicate, but allow repeated replication

n, dir and step are lists. Default values for dir are [0,1,2] and [1.0,1.0,1.0], cutoff at the length of the specified n.

replic(*n*, *step=1.0*, *dir=0*)

Return a Formex with n replications in direction dir with step.

The original Formex is the first of the n replicas.

replic2(*n1*, *n2*, *t1=1.0*, *t2=1.0*, *d1=0*, *d2=1*, *bias=0*, *taper=0*)

Replicate in two directions.

n1,n2 number of replications with steps t1,t2 in directions d1,d2 bias, taper : extra step and extra number of generations in direction d1 for each generation in direction d2

rosette(*n, angle, axis=2, point=[0.0, 0.0, 0.0]*)

Return a Formex with n rotational replications with angular step angle around an axis parallel with one of the coordinate axes going through the given point. axis is the number of the axis (0,1,2). point must be given as a list (or array) of three coordinates. The original Formex is the first of the n replicas.

ros(*n, angle, axis=2, point=[0.0, 0.0, 0.0]*)

Return a Formex with n rotational replications with angular step angle around an axis parallel with one of the coordinate axes going through the given point. axis is the number of the axis (0,1,2). point must be given as a list (or array) of three coordinates. The original Formex is the first of the n replicas.

translatem(**args*, ***kargs*)

Multiple subsequent translations in axis directions.

The argument list is a sequence of tuples (axis, step). Thus translatem((0,x),(2,z),(1,y)) is equivalent to translate([x,y,z]). This function is especially conveniant to translate in calculated directions.

extrude (*n*, *step=1.0*, *dir=0*)
Extrude a Formex in one of the axis directions.

Returns a Formex with doubled plexitude.

First the original Formex is translated over n steps of length step in direction dir. Then each pair of subsequent Formices is connected to form a higher plexitude structure.

Currently, this function correctly transforms: point1 to line2, line2 to quad4, tri3 to wedge6, quad4 to hex8.

See the 'connect' function for a more versatile tool.

divide (*div*)
Divide a plex-2 Formex at the values in div.

Replaces each member of the Formex by a sequence of members obtained by dividing the Formex at the relative values specified in div. The values should normally range from 0.0 to 1.0.

As a convenience, if an integer is specified for div, it is taken as a number of divisions for the interval [0..1].

This function only works on plex-2 Formices (line segments).

intersectionWithPlane (*p*, *n*, *atol=0*)
Return the intersection of a Formex with the plane (p,n) within tolerance atol.

Currently this only works for plex-2 and plex-3 Formices.

The intersection of the Formex with a plane specified by a point p and normal n is returned. For a plex-2 Formex (lines), the returned Formex will be of plexitude 1 (points). For a plex-3 Formex (triangles) the returned Formex has plexitude 2 (lines).

cutWithPlane (*p*, *n*, *side=''*, *atol=None*, *newprops=None*)
Cut a Formex with the plane(s) (p,n).

> **Warning:** This method currently only works for plexitude 2 or 3!

Parameters:

- *p*, *n*: a point and normal vector defining the cutting plane. In case of a Formex of plexitude 2, both *p* and *n* have shape (3,). In case of plexitude 3, *p* and/or *n* can be sequences of points, resp. vectors, allowing cutting with multiple planes. Both *p* and *n* can have shape (3) or (nplanes,3).
- *side*: either an empty string, or one of '+' or '-'. In the latter cases, only the part at the positive, resp. negative side of the plane (as defined by its normal) is returned. The (default) empty string makes both parts being returned as a tuple (pos,neg).

Returns:

The default return value is a tuple of two Formices of the same plexitude as the input: (Fpos,Fneg), where Fpos is the part of the Formex at the positive side of the plane (as defined by the normal vector), and Fneg is the part at the negative side. Elements of the input Formex that are lying completely on one side of the plane will return unaltered. Elements that are crossing the plane will be cut and split up into multiple parts.

When side = '+' or '-' (or 'positive'or 'negative'), only one of the sides is returned.

split (*n=1*)
Split a Formex in subFormices containing n elements.

The number of elements in the Formex should be a multiple of n. Returns a list of Formices each comprising n elements.

lengths ()
Compute the length of all elements of a 2-plex Formex.

The length of an element is the distance between its two points.

areas ()
Compute the areas of all elements of a 3-plex Formex.

The area of an element is the aread of the triangle formed by its three points.

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 points.

write (*fil*, *sep=' '*, *mode='w'*)
Write a Formex to file.

If fil is a string, a file with that name is opened. Else fil should be an open file. The Formex is then written to that file in a native format, using sep as separator between the coordinates. If fil is a string, the file is closed prior to returning.

classmethod **read** (*clas*, *fil*, *sep=' '*)
Read a Formex from file.

fil is a filename or a file object. If the file is in a valid Formex file format, the Formex is read and returned. Otherwise, None is returned. Valid Formex file formats are described in the manual.

classmethod **fromstring** (*clas*, *fil*, *sep=' '*, *nplex=1*, *ndim=3*, *count=-1*)
Create a `Formex` from coodinates in 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:

- *fil*: a string containing a single sequence of float numbers separated by whitespace and a possible separator string.
- *sep*: the separator used between the coordinates. If not a space, all extra whitespace is ignored.
- *ndim*: 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*: total number of coordinates to read. This should be a multiple of 3. The default is to read all the coordinates in the string. `count` can be used to force an error condition if the string does not contain the expected number of values.

The return value is a `Coords` object.

classmethod **fromfile** (*clas*, *fil*, *sep=' '*, *nplex=1*)
Read the coordinates of a Formex from a file

addNoise (**args*, ***kargs*)
Apply 'addNoise' transformation to the Geometry object.

See `coords.Coords.addNoise()` for details.

affine (**args*, ***kargs*)
Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

align (**args*, ***kargs*)
Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

bump (**args*, ***kargs*)
Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

bump1 (**args*, ***kargs*)
Apply 'bump1' transformation to the Geometry object.

See `coords.Coords.bump1()` for details.

bump2 (**args*, ***kargs*)
Apply 'bump2' transformation to the Geometry object.

See `coords.Coords.bump2()` for details.

centered (**args*, ***kargs*)
Apply 'centered' transformation to the Geometry object.

See `coords.Coords.centered()` for details.

cylindrical (**args*, ***kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See `coords.Coords.cylindrical()` for details.

egg (**args*, ***kargs*)
Apply 'egg' transformation to the Geometry object.

See `coords.Coords.egg()` for details.

flare (**args*, ***kargs*)
Apply 'flare' transformation to the Geometry object.

See `coords.Coords.flare()` for details.

hyperCylindrical (**args*, ***kargs*)
Apply 'hyperCylindrical' transformation to the Geometry object.

See `coords.Coords.hyperCylindrical()` for details.

isopar (**args*, ***kargs*)
Apply 'isopar' transformation to the Geometry object.

See `coords.Coords.isopar()` for details.

map (**args*, ***kargs*)
Apply 'map' transformation to the Geometry object.

See `coords.Coords.map()` for details.

map1 (**args*, ***kargs*)
Apply 'map1' transformation to the Geometry object.

See `coords.Coords.map1()` for details.

mapd (**args*, ***kargs*)
Apply 'mapd' transformation to the Geometry object.

See `coords.Coords.mapd()` for details.

position (**args*, ***kargs*)
Apply 'position' transformation to the Geometry object.

See `coords.Coords.position()` for details.

projectOnCylinder (**args*, ***kargs*)
Apply 'projectOnCylinder' transformation to the Geometry object.

See `coords.Coords.projectOnCylinder()` for details.

projectOnPlane (**args*, ***kargs*)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (**args*, ***kargs*)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

reflect (**args*, ***kargs*)
Apply 'reflect' transformation to the Geometry object.

See `coords.Coords.reflect()` for details.

replace (**args*, ***kargs*)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

rollAxes (**args*, ***kargs*)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

rot (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

rotate (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

scale (**args*, ***kargs*)
Apply 'scale' transformation to the Geometry object.

See `coords.Coords.scale()` for details.

shear (**args*, ***kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

spherical (**args*, ***kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args*, ***kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

swapAxes (**args*, ***kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

toCylindrical (**args*, ***kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

toSpherical (**args*, ***kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

transformCS (**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

translate (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

trl (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

setProp (*prop=None*, *blocks=None*)
Create or destroy the property array for the Geometry.

A property array is a rank-1 integer array with dimension 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. They play an import role when creating new geometry: new elements inherit the property number of their parent element. Properties are also preserved on most geometrical 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*: a single integer value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

  The special value 'range' will set the property numbers equal to the element number.

  A value None (default) removes the properties from the Geometry.

- *blocks*: a single integer value or a list/array of integer values. If the number of passed values is less than the length of *prop*, they wil be repeated. If you give more, they will be ignored. Every prop will be repeated the corresponding number of times specified in blocks.

toProp(*prop*)
Converts the argument into a legal set of properties for the object.

The conversion involves resizing the argument to a 1D array of length self.nelems(), and converting the data type to integer.

getCoords()
Get the coords data.

Returns the full array of coordinates stored in the Geometry object. Note that subclasses may store more points in this array than are used to define the geometry.

copy()
Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

splitProp(*prop=None*)
Partition a Geometry (Formex/Mesh) according to the values in prop.

Parameters:

- *prop*: an int array with length self.nelems(), or None. If None, the *prop* attribute of the Geometry is used.

Returns a list of Geometry objects of the same type as the input. Each object 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 their *prop* value.

It *prop* is None and the the object has no *prop* attribute, an empty list is returned.

nnodes()
Return the number of points in the formex.

This is the product of the number of elements in the formex with the number of nodes per element.

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.

`Functions defined in module formex`

formex.**connect**(*Flist*, *nodid=None*, *bias=None*, *loop=False*)
Return a Formex which connects the Formices in list.

Flist is a list of formices, nodid is an optional list of nod ids and bias is an optional list of element bias values. All lists should have the same length. The returned Formex has a plexitude equal to the number of formices in list. Each element of the Formex consist of a node from the corresponding element of each of the formices in list. By default this will be the first node of that element, but a nodid list may be given to specify the node id to be used for each of the formices. Finally, a list

of bias values may be given to specify an offset in element number for the subsequent formices. If loop==False, the order of the Formex will be the minimum order of the formices in Flist, each minus its respective bias. By setting loop=True however, each Formex will loop around if its end is encountered, and the order of the result is the maximum order in Flist.

`formex.`**`interpolate`**(*F*, *G*, *div*, *swap=False*, *concat=True*)
Create interpolations between two formices.

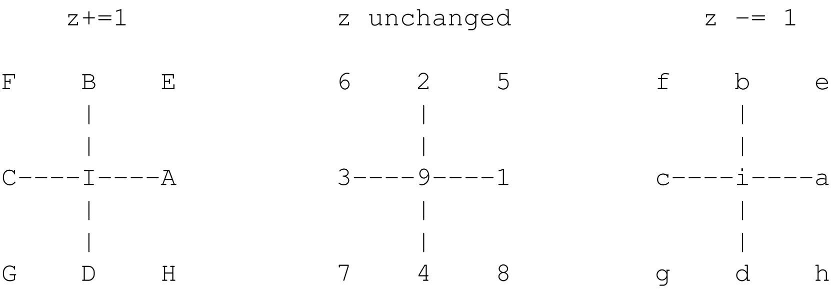
F and G are two Formices with the same shape. div is a list of floating point values. The result is the concatenation of the interpolations of F and G at all the values in div. An interpolation of F and G at value v is a Formex H where each coordinate Hijk is obtained from: Hijk = Fijk + v * (Gijk-Fijk). Thus, a Formex interpolate(F,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.

As a convenience, if an integer is specified for div, it is taken as a number of divisions for the interval [0..1]. Thus, interpolate(F,G,n) is equivalent with interpolate(F,G,arange(0,n+1)/float(n))

The swap argument sets the order of the elements in the resulting Formex. By default, if n interpolations are created of an m-element Formex, the element order is in-Formex first (n sequences of m elements). If swap==True, the order is swapped and you get m sequences of n interpolations.

`formex.`**`lpattern`**(*s*, *connect=True*)
Return a line segment pattern created from a string.

This function creates a list of line segments where all points lie on a regular grid with unit step. The first point of the list is [0,0,0]. Each character from the input string is interpreted as a code specifying how to move to the next point. Currently defined are the following codes: 1..8 move in the x,y plane 9 remains at the same place 0 = goto origin [0,0,0] + = go back to origin without creating a line segment When looking at the plane with the x-axis to the right, 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 'ABCDEFGHI', resp. 'abcdefghi', correspond with '123456789' with an extra z +/-= 1. This gives the following schema:



The special character '/' can be put before any character to make the move without inserting an element. The effect of any other character is undefined.

The resulting list is directly suited to initialize a Formex.

`formex.`**`pointsAt`**(*F*, *t*)
Return the points of a plex-2 Formex at times t.

F is a plex 2 Formex and t is an array with F.nelems() float values which are interpreted as local parameters along the edges of the Formex, such that the first node has value 0.0 and the last has value 1.0. The return value is a `coords.Coords` array with the points at values t.

`formex.`**`intersectionLinesWithPlane`**(*F*, *p*, *n*, *atol=0.0001*)
Return the intersection lines of a plex-3 Formex with plane (p,n).

F is a Formex of plexitude 3. p is a point specified by 3 coordinates. n is the normal vector to a plane, specified by 3 components. atol is a tolerance factor defining whether an edge is intersected by the plane.

formex.**cut2AtPlane**(*F*, *p*, *n*, *side='‘*, *atol=None*, *newprops=None*)
Returns all elements of the Formex cut at plane.

F is a Formex of plexitude 2. p is a point specified by 3 coordinates. n is the normal vector to a plane, specified by 3 components.

The return value is:

- with side = '+' or '-' or 'positive'or 'negative' : a Formex of the same plexitude with all elements located completely at the positive/negative side of the plane(s) (p,n) retained, all elements lying completely at the negative/positive side removed and the elements intersecting the plane(s) replaced by new elements filling up the parts at the positive/negative side.
- with side = '': two Formices of the same plexitude, one representing the positive side and one representing the negative side.

To avoid roundoff errors and creation of very small elements, a tolerance can be specified. Points lying within the tolerance distance will be considered lying in the plane, and no cutting near these points.

formex.**cut3AtPlane**(*F*, *p*, *n*, *side='‘*, *atol=None*, *newprops=None*)
Returns all elements of the Formex cut at plane(s).

F is a Formex of plexitude 3. p is a point or a list of points. n is the normal vector to a plane or a list of normal vectors. Both p and n have shape (3) or (npoints,3).

The return value is:

- with side='+' or '-' or 'positive'or 'negative' : a Formex of the same plexitude with all elements located completely at the positive/negative side of the plane(s) (p,n) retained, all elements lying completely at the negative/positive side removed and the elements intersecting the plane(s) replaced by new elements filling up the parts at the positive/negative side.
- with side='': two Formices of the same plexitude, one representing the positive side and one representing the negative side.

Let $dist$ be the signed distance of the vertices to a plane. The elements located completely at the positive or negative side of a plane have three vertices for which $|dist| > atol$. The elements intersecting a plane can have one or more vertices for which $|dist| < atol$. These vertices are projected on the plane so that their distance is zero.

If the Formex has a property set, the new elements will get the property numbers defined in newprops. This is a list of 7 property numbers flagging elements with following properties:

0. no vertices with $|dist| < atol$, triangle after cut
1. no vertices with $|dist| < atol$, triangle 1 from quad after cut
2. no vertices with $|dist| < atol$, triangle 2 from quad after cut
3. one vertex with $|dist| < atol$, two vertices at pos. or neg. side
4. one vertex with $|dist| < atol$, one vertex at pos. side, one at neg.
5. two vertices with $|dist| < atol$, one vertex at pos. or neg. side
6. three vertices with $|dist| < atol$

formex.**cutElements3AtPlane**(*F*, *p*, *n*, *newprops=None*, *side=''*, *atol=0.0*)
This function needs documentation.

Should it be called by the user? or only via cut3AtPlane? For now, lets suppose the last, so no need to check arguments here.

newprops should be a list of 7 values: each an integer or None side is either '+', '-' or ''

6.1.3 `arraytools` — A collection of numerical array utilities.

These are general utility functions that depend only on the `numpy` array model. All pyformex modules needing `numpy` should import everything from this module:

```
from arraytools import *
```

Classes defined in module arraytools

Functions defined in module arraytools

arraytools.**isInt**(*obj*)
Test if an object is an integer number

Returns True if the object is a single integer number, else False. The type of the object can be either a Python integer (int) or a numpy integer.

arraytools.**powers**(*x*, *n*)
Compute all the powers of x from zero up to n

Returns a list of arrays with same shape as x

arraytools.**sind**(*arg*, *angle_spec=0.017453292519943295*)
Return the sine of an angle in degrees.

For convenience, this can also be used with an angle in radians, by specifying *angle_spec=RAD*.

```
>>> print(sind(30), sind(pi/6,RAD))
0.5 0.5
```

arraytools.**cosd**(*arg*, *angle_spec=0.017453292519943295*)
Return the cosine of an angle in degrees.

For convenience, this can also be used with an angle in radians, by specifying `angle_spec=RAD`.

```
>>> 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.

For convenience, this can also be used with an angle in radians, by specifying `angle_spec=RAD`.

arraytools.**arcsind**(*arg*, *angle_spec=0.017453292519943295*)
Return the angle whose sine is equal to the argument.

By default, the angle is returned in Degrees. Specifying *angle_spec=RAD* will return the angle in radians.

```
>>> print(arcsind(0.5), arcsind(1.0,RAD))
30.0 1.57079632679
```

arraytools.**arccosd**(*arg*, *angle_spec=0.017453292519943295*)

Return the angle whose cosine is equal to the argument.

By default, the angle is returned in Degrees. Specifying *angle_spec=RAD* will return the angle in radians.

```
>>> print(arccosd(0.5), arccosd(-1.0,RAD))
60.0 3.14159265359
```

arraytools.**arctand**(*arg*, *angle_spec=0.017453292519943295*)

Return the angle whose tangens is equal to the argument.

By default, the angle is returned in Degrees. Specifying *angle_spec=RAD* will return the angle in radians.

```
>>> print(arctand(1.0), arctand(-1.0,RAD))
45.0 -0.785398163397
```

arraytools.**arctand2**(*sin*, *cos*, *angle_spec=0.017453292519943295*)

Return the angle whose sine and cosine values are given.

By default, the angle is returned in Degrees. Specifying *angle_spec=RAD* will return the angle in radians. This returns an angle in the range]-180,180].

```
>>> print(arctand2(0.0,-1.0), arctand2(-sqrt(0.5),-sqrt(0.5),RAD))
180.0 -2.35619449019
```

arraytools.**niceLogSize**(*f*)

Return the smallest integer e such that 10**e > abs(f).

This returns the number of digits before the decimal point.

```
>>> print([ niceLogSize(a) for a in [1.3, 35679.23, 0.4, 0.00045676] ])
[1, 5, 0, -3]
```

arraytools.**niceNumber**(*f*, *below=False*)

Return a nice number close to f.

f is a float number, whose sign is disregarded.

A number close to abs(f) but having only 1 significant digit is returned. By default, the value is above abs(f). Setting below=True returns a value above.

Example:

```
>>> numbers = [ 0.0837, 0.837, 8.37, 83.7, 93.7]
>>> [ str(niceNumber(f)) for f in numbers ]
['0.09', '0.9', '9.0', '90.0', '100.0']
>>> [ str(niceNumber(f,below=True)) for f in numbers ]
['0.08', '0.8', '8.0', '80.0', '90.0']
```

arraytools.**dotpr**(*A*, *B*, *axis=-1*)

Return the dot product of vectors of A and B in the direction of axis.

This multiplies the elements of the arrays A and B, and the sums the result in the direction of the specified axis. Default is the last axis. Thus, if A and B are sets of vectors in their last array

direction, the result is the dot product of vectors of A with vectors of B. A and B should be broadcast compatible.

```
>>> 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.]
```

`arraytools.`**`length`**`(`*A*, *axis=-1*`)`

Returns the length of the vectors of A in the direction of axis.

The components of the vectors are stored along the specified array axis (default axis is the last).

`arraytools.`**`normalize`**`(`*A*, *axis=-1*`)`

Normalize the vectors of A in the direction of axis.

The components of the vectors are stored along the specified array axis (default axis is the last).

`arraytools.`**`projection`**`(`*A*, *B*, *axis=-1*`)`

Return the (signed) length of the projection of vector of A on B.

The components of the vectors are stored along the specified array axis (default axis is the last).

`arraytools.`**`orthog`**`(`*A*, *B*, *axis=-1*`)`

Return the component of vector of A that is orthogonal to B.

The components of the vectors are stored along the specified array axis (default axis is the last).

`arraytools.`**`norm`**`(`*v*, *n=2*`)`

Return thr *n*-norm of the vector *v*.

Default is the quadratic norm (vector length). `n == 1` returns the sum. `n<=0` returns the max absolute value.

`arraytools.`**`horner`**`(`*a*, *u*`)`

Compute the value of a polynom using Horner's rule.

Parameters:

- *a*: float(n+1,nd), *nd*-dimensional coefficients of the polynom of degree *n*, starting from lowest degree.
- *u*: float(nu), parametric values where the polynom is evaluated

Returns float(nu,nd), nd-dimensional values of the polynom.

```
>>> 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.
- *b*: (ndof,nrhs,nsys) shaped float array.

Returns: a float array *x* with same shape as *b*, where `x[:,i,j]` solves the system of linear equations A[:,:,j].x[:,i,j] = b[:,i,j].

For ndof in [1,2,3], all solutions are by default computed directly and simultaneously. If `direct=False` is specified, a general linear equation solver is called for each system of equations. This is also the method used if `ndof>4`.

`arraytools.`**`inside`**`(`*p*, *mi*, *ma*`)`
Return true if point p is inside bbox defined by points mi and ma

`arraytools.`**`isClose`**`(`*values*, *target*, *rtol=1e-05*, *atol=1e-08*`)`
Returns an array flagging the elements close to target.

values is a float array, *target* is a float value. *values* and *target* should be broadcastable to the same shape.

The return value is a boolean array with shape of *values* flagging where the values are close to target. Two values *a* and *b* are considered close if $|a - b| < atol + rtol * |b|$

`arraytools.`**`anyVector`**`(`*v*`)`
Create a 3D vector.

v is some data compatible with a (3)-shaped float array. Returns v as such an array.

`arraytools.`**`unitVector`**`(`*v*`)`
Return a unit vector in the direction of v.

- *v* is either an integer specifying one of the global axes (0,1,2), or a 3-element array or compatible.

`arraytools.`**`rotationMatrix`**`(`*angle*, *axis=None*, *angle_spec=0.017453292519943295*`)`
Return a rotation matrix over angle, optionally around axis.

The angle is specified in degrees, unless angle_spec=RAD is specified. If axis==None (default), a 2x2 rotation matrix is returned. Else, axis should specifyi 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. In either case a 3x3 rotation matrix is returned. Note that:

- rotationMatrix(angle,[1,0,0]) == rotationMatrix(angle,0)
- rotationMatrix(angle,[0,1,0]) == rotationMatrix(angle,1)
- rotationMatrix(angle,[0,0,1]) == rotationMatrix(angle,2)

but the latter functions calls are more efficient. The result is returned as an array.

`arraytools.`**`rotmat`**`(`*x*`)`
Create a rotation matrix defined by 3 points in space.

x is an array of 3 points. After applying the resulting rotation matrix to the global axes, the 0 axis becomes // to the vectors x0-x1, the 1 axis lies in the plane x0,x1,x2 and is orthogonal to x0-x1, and the 3 axis is orthogonal to the plane x0,x1,x2.

`arraytools.`**`trfMatrix`**`(`*x*, *y*`)`
Find the transformation matrix from points x0 to x1.

x and y are each arrays of 3 non-colinear points. The return value is a tuple of a translation vector and a rotation matrix. The returned translation trl and rotationmatrix rot transform the points x thus 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

The rotation is to be applied first and should be around the first point x0. The full transformation of a Coords object is thus obtained by:

```
(coords-x0)*rot+trl+x0 = coords*rot+(trl+x0-x0*rot)
```

`arraytools.`**`rotMatrix`**(*u, w=[0.0, 0.0, 1.0], n=3*)

Create a rotation matrix that rotates axis 0 to the given vector.

u is a vector representing the Return either a 3x3(default) or 4x4(if n==4) rotation matrix.

`arraytools.`**`rotationAnglesFromMatrix`**(*mat*, *angle_spec=0.017453292519943295*)

Return rotation angles from rotation matrix mat.

This returns the three angles around the global axes 0, 1 and 2. The angles are returned in degrees, unless angle_spec=RAD.

`arraytools.`**`vectorRotation`**(*vec1*, *vec2*, *upvec=None*)

Return a rotation matrix for rotating vector vec1 to vec2.

If upvec is specified, the rotation matrix will be such that the plane of vec2 and the rotated upvec will be parallel to the original upvec.

This function is like `arraytools.rotMatrix()`, but allows the specification of vec1. The returned matrix should be used in postmultiplication to the Coords.

`arraytools.`**`growAxis`**(*a*, *add*, *axis=-1*, *fill=0*)

Increase the length of a single array axis.

The specified axis of the array *a* is increased with a value *add* and the new elements all get the value *fill*.

Parameters:

- *a*: array.
- *add*: int The value to add to the axis length. If<=0, the unchanged array is returned.
- *axis*: int The axis to change, default -1 (last).
- *fill*: int or float The value to set the new elements to.

Returns an array with same dimension and type as *a*, but with a length along *axis* equal to `a.shape[axis]+add`. The new elements all have the value *fill*.

Example:

```
>>> growAxis([[1,2,3],[4,5,6]],2)
array([[1, 2, 3, 0, 0],
       [4, 5, 6, 0, 0]])
```

`arraytools.`**`reorderAxis`**(*a*, *order*, *axis=-1*)

Reorder the planes of an array along the specified axis.

The elements of the array are reordered along the specified axis according to the specified order.

Parameters:

- *a*: array_like
- *order*: specifies how to reorder the elements. It is either one of the special string values defined below, or else it is an index holding a permutation of *arange(self.nelems()*. Each

value specifies the index of the old element that should be placed at its position. Thus, the order values are the old index numbers at the position of the new index number.

order can also take one of the following predefined 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

Returns an array with the same elements of self, where only the order along the specified axis has been changed.

Example:

```
>>> reorderAxis([[1,2,3],[4,5,6]],[2,0,1])
array([[3, 1, 2],
       [6, 4, 5]])
```

arraytools.**reverseAxis**(*a, axis=-1*)

Reverse the elements along a computed axis.

Example:

```
>>> reverseAxis([[1,2,3],[4,5,6]],0)
array([[4, 5, 6],
       [1, 2, 3]])
```

Note that if the axis is known in advance, it may be more efficient to use an indexing operation:

```
>>> A = array([[1,2,3],[4,5,6]])
>>> print(A[:,::-1])
[[3 2 1]
 [6 5 4]]
```

arraytools.**addAxis**(*a, axis=0*)

Add an additional axis with length 1 to an array.

The new axis is inserted before the specified one. Default is to add it at the front.

arraytools.**multiplex**(*a, n, axis=-1*)

Multiplex an array over a length n in direction of a new axis.

Inserts a new axis before the specified axis and repeats the data of the array n times in the direction of the new axis.

Returns an array with n times the original data in the direction of the specified axis (if positive) or the specified axis minus one (if negative).

Note that you can not use a negative number to multiplex if the new axis is the last one. To multiplex on the last dimension, use axis=a.ndim.

Example:

```
>>> a = arange(6).reshape(2,3)
>>> for i in range(-a.ndim,a.ndim+1):
...     c = multiplex(a,4,i)
...     print("%s: %s" % (i,c.shape))
-2: (4, 2, 3)
-1: (2, 4, 3)
0: (4, 2, 3)
```

```
1: (2, 4, 3)
2: (2, 3, 4)
>>> print(multiplex(a,4))
[[[0 1 2]
  [0 1 2]
  [0 1 2]
  [0 1 2]]

 [[3 4 5]
  [3 4 5]
  [3 4 5]
  [3 4 5]]]
```

`arraytools.`**`stack`**(*al*, *axis=0*)

Stack a list of arrays along a new axis.

al is a list of arrays all of the same shape. The return value is a new array with one extra axis, along which the input arrays are stacked. The position of the new axis can be specified, and is the first axis by default.

`arraytools.`**`concat`**(*al*, *axis=0*)

Smart array concatenation ignoring empty arrays

`arraytools.`**`minmax`**(*a*, *axis=-1*)

Compute the minimum and maximum along an axis.

a is an array. Returns an array of the same type as the input array, and with the same shape, except for the specified axis, which will have length 2. Along this axis are stored the minimum and maximum values along that axis in the input array.

Example:

```
>>> 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.]]

 [[ 0.  0.  0.]
  [ 2.  2.  0.]]]
```

`arraytools.`**`splitrange`**(*n*, *nblk*)

Split the range of integers 0..n in nblk almost equal sized slices.

This divides the range of integer numbers 0..n in nblk slices of (almost) equal size. If n > nblk, returns nblk+1 integers in the range 0..n. If n <= nblk, returns range(n+1).

Example:

```
>>> splitrange(7,3)
array([0, 2, 5, 7])
```

`arraytools.`**`splitar`**(*ar*, *nblk*, *close=False*)

Split an array in nblk subarrays along axis 0.

Splits the array ar along its first axis in nblk blocks of (almost) equal size.

Returns a list of nblk arrays, unless the size of the array is smaller than nblk, in which case a list with the original array is returned.

If close==True, the elements where the array is split occur in both blocks delimited by the element.

Example:

```
>>> 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])]
```

arraytools.**checkInt**(*value*, *min=None*, *max=None*)
Check that a value is an int in the range min..max

Range borders that are None are not checked upon. Returns an int in the specified range. Raises an exception if the value is invalid.

arraytools.**checkFloat**(*value*, *min=None*, *max=None*)
Check that a value is a float in the range min..max

Range borders that are None are not checked upon. Returns a float in the specified range. Raises an exception if the value is invalid.

arraytools.**checkArray**(*a*, *shape=None*, *kind=None*, *allow=None*, *size=None*, *ndim=None*)
Check that an array a has the correct shape, type and/or size.

The input *a* is anything that can be converted into a numpy array. Either *shape* and/or *kind* and/or *type* can be specified and will then be checked. The dimensions where *shape* contains a -1 value are not checked. The number of dimensions should match. If *kind* does not match, but the value is included in *allow*, conversion to the requested type is attempted. If *size* is specified, the size should exactly match. If 'ndim' is specified, the array should have precisely ndim dimensions.

Returns the array if valid; else, an error is raised.

arraytools.**checkArray1D**(*a*, *kind=None*, *allow=None*, *size=None*)
Check and force an array to be 1D.

Turns the first argument into a 1D array. Optionally checks the kind of data (int/float) and the size of the array.

Returns the array if valid; else, an error is raised.

This is equivalent to calling checkArray with shape=None and then ravel the result.

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 math:$nmin <= i < nmax$

Parameters:

- *nrs*: an integer array of any shape.
- *nmin*: minimum allowed value. If set to None, the test is skipped.
- *nmax*: maximum allowed value + 1! If set to None, the test is skipped.

Default range is [0,unlimited].

If the numbers are no unique or one of the limits is passed, an error is raised. Else, the sorted list of unique values is returned.

`arraytools.`**`readArray`**(*file*, *dtype*, *shape*, *sep=' '*)
Read an array from an open file.

This uses `numpy.fromfile()` to read an array with known shape and data type from an open file. The sep parameter can be specified as in *numpy.fromfile*. If an empty string is given as separator, the data is read in binary mode. In that case (only) an extra 'n' after the data will be stripped off.

`arraytools.`**`writeArray`**(*file*, *array*, *sep=' '*)
Write an array to an open file.

This uses `numpy.tofile()` to write an array to an open file. The sep parameter can be specified as in tofile.

`arraytools.`**`cubicEquation`**(*a*, *b*, *c*, *d*)
Solve a cubiq equation using a direct method.

a,b,c,d are the (floating point) coefficients of a third degree polynomial equation:

```
a*x**3+b*x**2+c*x+d=0
```

This function computes the three roots (real and complex) of this equation and returns full information about their kind, sorting order, occurrence of double roots. It uses scaling of the variables to enhance the accuracy.

The return value is a tuple (r1,r2,r3,kind), where r1,r2 and r3 are three float values and kind is an integer specifying the kind of roots.

Depending on the value of *kind*, the roots are defined as follows:

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=sqrt(-1).

If the coefficient a==0, a ValueError is raised.

Example:

```
>>> cubicEquation(1.,-3.,3.,-1.)
([1.0, 1.0, 1.0], 3)
```

`arraytools.`**`uniqueOrdered`**(*ar1*, *return_index=False*, *return_inverse=False*)
Find the unique elements of an array.

This works like numpy's unique, but uses a stable sorting algorithm. The returned index may therefore hold other entries for multiply occurring values. In such case, uniqueOrdered returns the first occurrence in the flattened array. The unique elements and the inverse index are always the same as those returned by numpy's unique.

Parameters:

- *ar1*: array_like This array will be flattened if it is not already 1-D.
- *return_index*: bool, optional If True, also return the indices against *ar1* that result in the unique array.

- *return_inverse*: bool, optional If True, also return the indices against the unique array that result in *ar1*.

Returns:

- *unique*: ndarray The unique values.
- *unique_indices*: ndarray, optional The indices of the unique values. Only provided if *return_index* is True.
- *unique_inverse*: ndarray, optional The indices to reconstruct the original array. Only provided if *return_inverse* is True.

Example:

```
>>> a = array([2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,7,8])
>>> uniq,ind,inv = unique(a,True,True)
>>> print(uniq)
[1 2 3 4 5 6 7 8]
>>> print(ind)
[7 0 1 2 3 4 5 6]
>>> print(inv)
[1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 6 7]
>>> uniq,ind,inv = uniqueOrdered(a,True,True)
>>> print(uniq)
[1 2 3 4 5 6 7 8]
>>> print(ind)
[7 0 1 2 3 4 5 6]
>>> print(inv)
[1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 6 7]
```

Notice the difference in the fourth element of the *ind* array.

`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*: array_like, 1-D, integer An array with non-negative integer values

Returns:

> A 1-D integer array with length equal to *nval*, where *nval* is the number of different values in *index*, and holding the original values corresponding to the new value *0..nval*.

Remark:

> Use `inverseUniqueIndex()` to find the inverse mapping needed to replace the values in the index by the new ones.

Example:

```
>>> 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.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*: array_like, 1-D, int or bool. If integer, it is a list with the non-negative numbers to be excluded from the range(0,n). If boolean, it normally has the length of the range and flags the elements to be returned with a False value.
- *n*: int: the 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:

If *index* is integer: a 1-D integer array with the numbers from range(0,n) that are not included in *index*. If *index* is boolean, the negated *index* padded to or cut at length *n*.

Example:

```
>>> 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.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*: array_like, 1-D, integer An array with non-negative values, which all have to be unique.

Returns:

A 1-D integer array with length *max+1*, with the positions in *index* of the values *0..max*, or -1 if the value does not occur in *index*.

Remark:

The inverse index translates the unique index numbers in a sequential index, so that `inverseUniqueIndex(index)[index] == arange(1+index.max())`.

Example:

```
>>> 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.`**`sortSubsets`**(*a*, *w=None*)

Sort subsets of an integer array a.

a is a 1-D integer array. Subsets of the array are the collections of equal values. w is a float array with same size of a, specifying a weight for each of the array elements in a. If no weight is specified, all elements have the same weight.

The subsets of a are sorted in order of decreasing total weight of the subsets (or number of elements if weight is None).

The return value is an integer array of the same size of a, specifying for each element the index of its subset in the sorted list of subsets.

Example:

```
>>> sortSubsets([0,1,2,3,1,2,3,2,3,3])
array([3, 2, 1, 0, 2, 1, 0, 1, 0, 0])

>>> sortSubsets([0,1,2,3,1,2,3,2,3,3],w=[9,8,7,6,5,4,3,2,1,0])
array([3, 1, 0, 2, 1, 0, 2, 0, 2, 2])
```

`arraytools.`**`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-D

Returns a 1-D integer array specifying the order in which the rows have to be taken to obtain an array sorted by columns.

Example:

```
>>> sortByColumns([[1,2],[2,3],[3,2],[1,3],[2,3]])
array([0, 3, 1, 4, 2])
```

`arraytools.`**`uniqueRows`**(*a*, *permutations=False*)

Find the unique rows of a 2-D array.

Parameters:

- *a*: array_like, 2-D
- *permutations*: bool If True, rows which are permutations of the same data are considered equal. The default is to consider permutations as different.

Returns:

- *uniq*: a 1-D integer array with the numbers of the unique rows from *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 axis -1 before calling this sorting function.
- *uniqid*: a 1-D integer array with length equal to *a.shape[0]* with the numbers of *uniq* corresponding to each of the rows of *a*.

Example:

```
>>> uniqueRows([[1,2],[2,3],[3,2],[1,3],[2,3]])
(array([0, 3, 1, 2]), array([0, 2, 3, 1, 2]))
>>> uniqueRows([[1,2],[2,3],[3,2],[1,3],[2,3]],permutations=True)
(array([0, 3, 1]), array([0, 2, 2, 1, 2]))
```

`arraytools.`**`argNearestValue`**`(`*values*, *target*`)`

Return the index of the item nearest to target.

Parameters:

- *values*: a list of float values
- *target*: a float value

Returns the position of the item in *values* that is nearest to *target*.

Example:

```
>>> argNearestValue([0.1,0.5,0.9],0.7)
1
```

`arraytools.`**`nearestValue`**`(`*values*, *target*`)`

Return the item nearest to target.

`values`: a list of float values

`target`: a single value

Returns the item in `values` values that is nearest to `target`.

`arraytools.`**`inverseIndex`**`(`*index*, *maxcon=4*`)`

Return an inverse index.

An index is an array pointing at other items by their position. The inverse index is a collection of the reverse pointers. Negative values in the input index are disregarded.

Parameters:

- *index*: an array of integers, where only non-negative values are meaningful, and negative values are silently ignored. A Connectivity is a suitable argument.
- *maxcon*: int: an initial estimate for the maximum number of rows a single element of index occurs at. The default will usually do well, because the procedure will automatically enlarge it when needed.

Returns:

An (mr,mc) shaped integer array where:

- *mr* will be equal to the highest positive value in index, +1.
- *mc* will be equal to the highest row-multiplicity of any number in *index*.

Row *i* of the inverse index contains all the row numbers of *index* that contain the number *i*. Because the number of rows containing the number *i* is usually not a constant, the resulting array will have a number of columns *mc* corresponding to the highest row-occurrence of any single number. Shorter rows are padded with -1 values to flag non-existing entries.

Example:

```
>>> inverseIndex([[0,1],[0,2],[1,2],[0,3]])
array([[ 0,  1,  3],
       [-1,  0,  2],
       [-1,  1,  2],
       [-1, -1,  3]])
```

arraytools.**matchIndex**(*target*, *values*)

Find position of values in target.

This function finds the position in the array *target* of the elements from the array *values*.

Parameters:

- *target*: an index array with all non-negative values. If not 1-D, it will be flattened.
- *values*: an index array with all non-negative values. If not 1-D, it will be flattened.

Returns:

An index array with the same size as *values*. For each number in *values*, the index 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.

Remark that after `m = matchIndex(target,values)` the equality `target[m] == values` holds in all the non-negative positions of *m*.

Example:

```
>>> A = array([1,3,4,5,7,8,9])
>>> B = array([0,6,7,1,2])
>>> matchIndex(A,B)
array([-1, -1,  4,  0, -1])
```

arraytools.**groupPositions**(*gid*, *values*)

Compute the group positions.

Computes the positions per group in a set of group identifiers.

Parameters:

- *gid*: (nid,) shaped int array of group identifiers
- *values*: (nval,) shaped int array with unique group identifiers for which to return the positions.

Returns:

- *pos*: list of int arrays giving the positions in *gid* of each group identifier in *values*.

```
>>> gid = array([ 2, 1, 1, 6, 6, 1 ])
>>> values = array([ 1, 2, 6 ])
>>> print(groupPositions(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*: (nval,) shaped array of values
- *gid*: (nval,) shaped int array of group identifiers

Returns:

- *ugid*: (ngrp,) shaped int array with unique group identifiers
- *minpos*: (ngrp,p) shape int array giving the position in *val* of the minimum of all values with the corresponding group identifier in *ugid*.

After return, the minimum values corresponding to the groups in *ugid* are given by `val[minpos]`.

```
>>> 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.`**`vectorLength`**`(`*vec*`)`

Return the lengths of a set of vectors.

vec is an (n,3) shaped array holding a collection of vectors. The result is an (n,) shaped array with the length of each vector.

`arraytools.`**`vectorNormalize`**`(`*vec*`)`

Normalize a set of vectors.

vec is a (n,3) shaped arrays holding a collection of vectors. The result is a tuple of two arrays:

- length (n): the length of the vectors vec
- normal (n,3): unit-length vectors along vec.

`arraytools.`**`vectorPairAreaNormals`**`(`*vec1*, *vec2*`)`

Compute area of and normals on parallellograms formed by vec1 and vec2.

vec1 and vec2 are (n,3) shaped arrays holding collections of vectors. As a convenience, single vectors may also be specified with shape (3,), and will be converted to (1,3).

The result is a tuple of two arrays:

- area (n) : the area of the parallellogram formed by vec1 and vec2.
- normal (n,3) : (normalized) vectors normal to each couple (vec1,2).

These are calculated from the cross product of vec1 and vec2, which indeed gives area * normal.

Note that where two vectors are parallel, an area zero results and an axis with components NaN.

`arraytools.`**`vectorPairArea`**`(`*vec1*, *vec2*`)`

Compute area of the parallellogram formed by a vector pair vec1,vec2.

vec1 and vec2 are (n,3) shaped arrays holding collections of vectors. The result is an (n) shaped array with the area of the parallellograms formed by each pair of vectors (vec1,vec2).

`arraytools.`**`vectorPairNormals`**`(`*vec1*, *vec2*`)`

Compute vectors normal to vec1 and vec2.

vec1 and vec2 are (n,3) shaped arrays holding collections of vectors. The result is an (n,3) shaped array of unit length vectors normal to each couple (edg1,edg2).

`arraytools.`**`vectorTripleProduct`**`(`*vec1*, *vec2*, *vec3*`)`

Compute triple product vec1 . (vec2 x vec3).

vec1, vec2, vec3 are (n,3) shaped arrays holding collections of vectors. The result is a (n,) shaped array with the triple product of each set of corresponding vectors from vec1,vec2,vec3. This is also the square of the volume of the parallellepid formex by the 3 vectors. If vec1 is a unit normal, the result is also the area of the parallellogram (vec2,vec3) projected in the direction vec1.

`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).

`arraytools.`**`percentile`**`(`*values, perc=[25.0, 50.0, 75.0], wts=None*`)`

Return the *perc* percentile(s) of *values*.

Parameters:

- *values*: a one-dimensional array of values for which to compute the percentile(s);
- *perc*: an integer, float or array specifying which percentile(s) to compute; by default, the quartiles are returned;
- *wts*: a one-dimensional array of weights assiged to *values*.

Returns the 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.

`arraytools.`**`multiplicity`**`(`*a*`)`

Return the multiplicity of the numbers in a

a is a 1-D integer array.

Returns a tuple of:

- 'mult': the multiplicity of the unique values in a
- 'uniq': the sorted list of unique values in a

Example:

```
>>> multiplicity([0,3,5,1,4,1,0,7,1])
(array([2, 3, 1, 1, 1, 1]), array([0, 1, 3, 4, 5, 7]))
```

`arraytools.`**`histogram2`**`(`*a*, *bins*, *range=None*`)`

Compute the histogram of a set of data.

This function is like numpy's 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. If bins is a sequence, it defines the bin edges, allowing for non-uniform bin widths. Both the leftmost and rightmost edges are included, thus the number of bins is len(bins)-1.
- *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*: integer array with length nbins, holding the number of elements in each bin,
- *ind*: a sequence of nbins integer arrays, each holding the indices of the elements fitting in the respective bins,
- *xbins*: array of same type as data and with length nbins+1: returns the bin edges.

Example:

```
>>> hist,ind,xbins = 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(xbins)
[1 2 3 4 5]
```

arraytools.**movingView**(*a*, *size*)

Create a moving view along the first axis of an array

Parameters:

- *a* : array_like: array for wihch to create a moving view
- *size* : int: size of the moving view

Returns an array that is a view of the original array with an extra first axis of length w.

Using swapaxes(0,axis) moving views over any axis can be created.

Example:

```
>>> x=arange(10).reshape((5,2))
>>> print(x)
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
>>> print(movingView(x, 3))
[[[0 1]
  [2 3]
  [4 5]]

 [[2 3]
  [4 5]
```

```
  [6 7]]

 [[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: array to be averaged
- *n* : int: moving sample size
- *m0* : optional, int: if specified, the first data set of a will be prepended this number of times
- *m1* : optional, int: if specified, the last data set of a will be appended this number of times

Returns:

An array with the moving average over n data sets along the first axis of a. The array has the same shape as a, except possibly for the length of the first axis. If neither m0 nor m1 are set, the first axis will have a length of a.shape[0] - (n-1). If both m0 and m1 are give, the first axis will have a length of a.shape[0] - (n-1) + 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].

Example:

```
>>> x=arange(10).reshape((5,2))
>>> print(movingAverage(x,3))
[[ 2.  3.]
 [ 4.  5.]
 [ 6.  7.]]
>>> 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 values between min and max

arraytools.**unitDivisor**(*div*, *start=0*)

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*: an integer, or a list of floating point values. If it is an integer, returns a list of floating point values dividing the interval 0.0 toi 1.0 in div equal parts.
- *start*: Set to 1 to skip the start value (0.0) of the interval.

Returns: 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.

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*, *umax*: float: start and end value of the interval. Default interval is [0.0..1.0].

Returns:

> A float array with 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.

Example:

```
>>> 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.**nodalSum**(*val*, *elems*, *avg=False*, *return_all=True*, *direction_treshold=None*)

Compute the nodal sum of values defined on elements.

val is a (nelems,nplex,nval) array of values defined at points of elements. elems is a (nelems,nplex) array with nodal ids of all points of elements.

The return value is a (nelems,nplex,nval) array where each value is replaced with the sum of its value at that node. If avg=True, the values are replaced with the average instead. If return_all==True(default), returns an array with shape (nelems,nplex,3), else, returns an array with shape (maxnodenr+1,3). In the latter case, nodes not occurring in elems will have all zero values.

If a direction_tolerance is specified and nval > 1, values will only be summed if their direction is close (projection of one onto the other is higher than the specified tolerance).

arraytools.**pprint**(*a*, *label=''*)

Pretty print an array with a label in front.

When printing a numpy array with a lable in font, the first row of the array is not aligned with the remainder. This function will solve that issue and prints the full array nicely aligned.

- *a*: a numpy array
- *label*: a sting to be printed in front of the array

Example:

```
>>> a = arange(12).reshape(-1,3)
>>> pprint(a,'Reshaped range = ')
Reshaped range = [[ 0  1  2]
                  [ 3  4  5]
                  [ 6  7  8]
                  [ 9 10 11]]
```

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 explicitely importing the `script` module.

`Classes defined in module script`

`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 dictionay that match criteria.

- dic: a dictionary object, defaults to pyformex.PF
- clas: a class name: if specified, only instances of this class 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.

The return value is a list of keys from dic.

`script.`**`named`**`(`*name*`)`
Returns the global object named name.

script.**getcfg**(*name*)
Return a value from the configuration.

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 acknowlegement.

script.**system**(*cmdline*, *result='output'*)
Run a command and return its output.

If result == 'status', the exit status of the command is returned. If result == 'output', the output of the command is returned. If result == 'both', a tuple of status and output is returned.

script.**playScript**(*scr*, *name=None*, *filename=None*, *argv=[ ]*, *pye=False*)
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 a name is specified, set the global variable pyformex.scriptName to it when the script is started. If a 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.`**`runAny`**(*appname=None*, *argv=[ ]*, *step=False*, *refresh=False*)

Run the current pyFormex application or script file.

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.`**`printall`**()

Print all Formices in globals()

`script.`**`isWritable`**(*path*)

Check that the specified path is writable.

BEWARE: this only works if the path exists!

`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*)
Create a new directory.

Create a new directory, including any needed parent directories.

- *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. If the path exists, the function returns True without doing anything.

Returns True if the pathname exists (before or after).

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.**checkRevision**(*rev*, *comp='>='*)
Check the pyFormex revision number.

- rev: a positive integer.
- comp: a string specifying a comparison operator.

By default, this function returns True if the pyFormex revision number is equal or larger than the specified number.

The comp argument may specify another comparison operator.

If pyFormex is unable to find its revision number (this is the case on very old versions) the test returns False.

script.**requireRevision**(*rev*, *comp='>='*)
Require a specified pyFormex revision number.

The arguments are like checkRevision. Ho9wever, this function will raise an error if the requirement fails.

script.**writeGeomFile**(*filename*, *objects*, *sep=' '*, *mode='w'*, *shortlines=False*)
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' the file will be compressed with gzip. If a file with the given name minus the trailing '.gz' exists, it will be destroyed.
- *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.

Returns the number of objects written to the file.

script.**readGeomFile**(*filename*)
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 `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.

Classes defined in module draw

Functions defined in module draw

draw.**closeGui**()
Close the GUI.

Calling this function from a script closes the GUI and terminates pyFormex.

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.

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 test. 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.

draw.**showInfo**(*text, actions=['OK'], modal=True*)
Show an informational message and wait for user acknowledgement.

draw.**warning**(*text, actions=['OK']*)
Show a warning message and wait for user acknowledgement.

draw.**error**(*text, actions=['OK']*)
Show an error message and wait for user acknowledgement.

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.

draw.**ack**(*question, **kargs*)
Show a Yes/No question and return True/False depending on answer.

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' of '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.

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.

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.

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 lok 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.

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, ot _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.

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.

draw.**dialogAccepted**()
Returns True if the last askItems() dialog was accepted.

draw.**dialogRejected**()
Returns True if the last askItems() dialog was rejected.

draw.**dialogTimedOut**()
Returns True if the last askItems() dialog timed out.

draw.**askFilename**(*cur=None*, *filter='All files (*.*)'*, *exist=True*, *multi=False*, *change=True*, *timeout=None*)
Ask for a file name or multiple file names using a file dialog.

cur is a directory or filename. All the files matching the filter in that directory (or that file's directory) will be shown. If cur is a file, it will be selected as the current filename.

Unless the user cancels the operation, or the change parameter was set to False, the parent directory of the selected file will become the new working directory.

draw.**askNewFilename**(*cur=None*, *filter='All files (*.*)'*, *timeout=None*)
Ask a single new filename.

This is a convenience function for calling askFilename with the arguments exist=False.

draw.**askDirname**(*path=None*, *change=True*, *byfile=False*)
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.

draw.**checkWorkdir**()
Ask the user to change the current workdir if it is not writable.

Returns True if the new workdir is writable.

draw.**printMessage**(*s*)
Print a message on the message board.

If a logfile was opened, the message is also written to the log file.

draw.**message**(*s*)
Print a message on the message board.

If a logfile was opened, the message is also written to the log file.

draw.**flatten**(*objects*, *recurse=True*)
Flatten a list of geometric 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.

draw.**drawable**(*objects*)
Filters the drawable objects from a list.

The input is a list, usually of drawable objects. For each item in the list, the following is done:

- if the item is drawable, it is kept as is,
- if the item is not drawable but can be converted to a Formex, it is converted,
- if it is neither drawable nor convertible to Formex, it is removed.

The result is a list of drawable objects (since a Formex is drawable).

draw.**draw**(*F*, *color='prop'*, *colormap=None*, *alpha=None*, *bkcolor=None*, *bkcolormap=None*, *bkalpha=None*, *mode=None*, *linewidth=None*, *linestipple=None*, *marksize=None*, *nolight=False*, *ontop=False*, *view=None*, *bbox=None*, *shrink=None*, *clear=None*, *wait=True*, *allviews=False*, *highlight=False*, *silent=True*, ***kargs*)
Draw object(s) with specified settings and options.

This is the main drawing function to get geometry rendered on the OpenGL canvas. It has a whole slew of arguments, but in most cases you will only need to use a few of them. We divide the arguments in three groups: geometry, settings, options.

Geometry: specifies what objects will be drawn.

- *F*: all geometry to be drawn is specified in this single argument. It can be one of the following:
  - a drawable object (a Geometry object like Formex, Mesh or TriSurface, or another object having a proper *actor* method),
  - the name of a global pyFormex variable refering to such an object,

–a list or nested list of any of the above items.

The possibility of a nested list means that any complex collections of geometry can be drawn in a single operations. The (nested) list is recursively flattened, replacing string values by the corresponding value from the pyFormex global variables dictionary, until a single list of drawable objects results. Next the undrawable items are removed from the list. The resulting list of drawable objects will then be drawn using the remaining settings and options arguments.

Settings: specify how the geometry will be drawn. 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*: specifies the color of the object (see below)
- *alpha*: float (0.0..1.0): alpha value to use in transparent mode
- *bkcolor*, *bkcolormap*: color for the backside of surfaces, if different from the front side. Specification as for front color.
- *bkalpha*: float (0.0..1.0): alpha value for 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.

Options: these arguments modify the working of the draw functions. If None, they are filled in from the current viewport drawing options. These can be changed with the `setDrawOptions()` function. The initial defaults are: view='last', bbox='auto', shrink=False, clear=False, shrinkfactor=0.8.

- *view*: is either the name of a defined view or 'last' or None. Predefined views are 'front', 'back', 'top', 'bottom', 'left', 'right', 'iso'. With view=None the camera settings remain unchanged (but might be changed interactively through the user interface). This may make the drawn object out of view! With view='last', the camera angles will be set to the same camera angles as in the last draw operation, undoing any interactive changes. On creation of a viewport, the initial default view is 'front' (looking in the -z direction).
- *bbox*: specifies the 3D volume at which the camera will be aimed (using the angles set by *view*). The camera position will be set so 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. The special value bbox=None will use the bounding box of the previous drawing operation, thus ensuring that the camera's target volume remains unchanged.

- *shrink*: bool: if specified, each object will be transformed by the `Coords.shrink()` transformation (with the current set shrinkfactor as a parameter), thus showing all the elements of the object separately. (Some other softwares call this an 'exploded' view).
- *clear*: bool. By default each new draw operation adds the newly drawn objects to the shown scene. Using *clear=True* will clear the scene before drawing and thus only show the objects of the current draw action.
- *wait*: bool. If True (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 specifyin *wait=False*. This means that the *next* draw operation does not have to wait.
- *allviews*: currently not used
- *highlight*: bool. If True, the object(s) will not be drawn as normal geometry, but as highlights (usually on top of other geometry), making them removeable by the remove highlight functions
- *silent*: bool. If True (default), non-drawable objects will be silently ignored. If set False, an error is raised if an object is not drawable.
- ***kargs*: any not-recognized keyword parameters are passed to the object's Actor constructor. This allows the user to create customized Actors with new parameters.

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.

`draw.`**`focus`**`(`*object*`)`
Move the camera thus that object comes fully into view.

object can be anything having a bbox() method or a list thereof. if no view is given, the default is used.

The camera is moved with fixed axis directions to a place where the whole object can be viewed using a 45. degrees lens opening. This technique may change in future!

`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.

`draw.`**`reset`**`()`
reset the canvas

`draw.`**`shrink`**`(`*onoff*, *factor=None*`)`
Set shrinking on or off, and optionally set shrink factor

`draw.`**`drawVectors`**`(`*P*, *v*, *size=None*, *nolight=True*, ***drawOptions*`)`
Draw a set of vectors.

If size==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.

`draw.`**`drawMarks`**`(`*X*, *M*, *color='black'*, *leader=''*, *ontop=True*`)`
Draw a list of marks at points X.

X is a Coords array. M is a list with the same length as X. The string representation of the marks are drawn at the corresponding 3D coordinate.

`draw.`**`drawFreeEdges`**`(`*M*, *color='black'*`)`
Draw the feature edges of a Mesh

`draw.`**`drawNumbers`**`(`*F*, *numbers=None*, *color='black'*, *trl=None*, *offset=0*, *leader=''*, *ontop=None*`)`
Draw numbers on all elements of F.

numbers is an array with F.nelems() integer numbers. If no numbers are given, the range from 0 to nelems()-1 is used. Normally, the numbers are drawn at the centroids of the elements. A translation 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. If an offset is specified, it is added to the shown numbers.

`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.

`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.

`draw.`**`drawBbox`**`(`*F*, *color=None*, *linewidth=None*`)`
Draw the bounding box of the geometric object F.

F is any object that has a *bbox* method. Returns the drawn Annotation.

`draw.`**`drawPrincipal`**`(`*F*, *weight=None*`)`
Draw the principal axes of the geometric object F.

F is any object that has a *coords* attribute. If specified, weight is an array of weights attributed to the points of F. It should have the same length as *F.coords*.

`draw.`**`drawText3D`**`(`*P*, *text*, *color=None*, *font='sans'*, *size=18*, *ontop=True*`)`
Draw a text at a 3D point P.

`draw.`**`drawAxes`**`(`*CS=None*, **args*, ***kargs*`)`
Draw the axes of a CoordinateSystem.

CS is a CoordinateSystem. If not specified, the global coordinate system is used. Other arguments can be added just like in the `AxesActor` class.

While you can draw a CoordinateSystem using the `draw()` function, this function gives a better result because it has specialized color and annotation settings and provides reasonable deafults.

`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()`.

`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.

`draw.`**`drawViewportAxes3D`**(*pos*, *color=None*)

Draw two viewport axes at a 3D position.

`draw.`**`drawActor`**(*A*)

Draw an actor and update the screen.

`draw.`**`drawAny`**(*A*)

Draw an Actor/Annotation/Decoration and update the screen.

`draw.`**`undraw`**(*itemlist*)

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.

`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 prefered 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 draing operation.

draw.**setTriade**(*on=None*, *pos='lb'*, *siz=100*)
Toggle the display of the global axes on or off.

If on is True, the axes triade is displayed, if False it is removed. The default (None) toggles between on and off.

draw.**drawText**(*text*, *x*, *y*, *gravity='E'*, *font='helvetica'*, *size=14*, *color=None*, *zoom=None*)
Show a text at position x,y using font.

draw.**annotate**(*annot*)
Draw an annotation.

draw.**decorate**(*decor*)
Draw a decoration.

draw.**createView**(*name*, *angles*, *addtogui=False*)
Create a new named view (or redefine an old).

The angles are (longitude, latitude, twist). By default, the view is local to the script's viewport. If gui is True, it is also added to the 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).

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 overlayed on the background colors. Specify a solid white background color to sea the image unaltered.

draw.**fgcolor**(*color*)
Set the default foreground color.

draw.**hicolor**(*color*)
Set the highlight color.

draw.**colormap**(*color=None*)
Gets/Sets the current canvas color map

draw.**colorindex**(*color*)
Return the index of a color in the current colormap

draw.**renderModes**()
Return a list of predefined render profiles.

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

draw.**wireMode**(*mode*)
Change the wire rendering mode.

Currently the following modes are defined: 'none', 'border', 'feature','all'

draw.**lights**(*state=True*)
Set the lights on or off

draw.**transparent**(*state=True*)
Set the transparency mode on or off.

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

draw.**linewidth**(*wid*)
Set the linewidth to be used in line drawings.

draw.**linestipple**(*factor*, *pattern*)
Set the linewidth to be used in line drawings.

draw.**pointsize**(*siz*)
Set the size to be used in point drawings.

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.

draw.**clear_canvas**()
Clear the canvas.

This is a low level function not intended for the user.

draw.**clear**()
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.

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 convertable to a float. The function still returns the old setting. This may be practical to save that value to restore it later.

`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 prefered 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*.

`draw.play(refresh=False)`

Start the current script or if already running, continue it.

`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().

`draw.fforward()`

Releases the drawing lock mechanism indefinely.

Releasing the drawing lock indefinely means that the lock will not be set again and your script will execute till the end.

`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

`draw.zoomRectangle()`

Zoom a rectangle selected by the user.

`draw.zoomBbox(bb)`

Zoom thus that the specified bbox becomes visible.

`draw.zoomAll()`

Zoom thus that all actors become visible.

`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 convertibel 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.

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

draw.**nViewports**()
Return the number of viewports.

draw.**layout**(*nvps=None*, *ncols=None*, *nrows=None*, *pos=None*, *rstretch=None*, *cstretch=None*)
Set the viewports layout.

draw.**addViewport**()
Add a new viewport.

draw.**removeViewport**()
Remove the last viewport.

draw.**linkViewport**(*vp*, *tovp*)
Link viewport vp to viewport tovp.

Both vp and tovp should be numbers of viewports.

draw.**updateGUI**()
Update the GUI.

draw.**highlightActor**(*actor*)
Highlight an actor in the scene.

draw.**removeHighlight**()
Remove the highlights from the current viewport

draw.**pick**(*mode='actor'*, *filter=None*, *oneshot=False*, *func=None*)
Enter interactive picking mode and return selection.

See viewport.py for more details. This function differs in that it provides default highlighting during the picking operation, a button to stop the selection operation

Parameters:

- *mode*: one of the pick modes
- *filter*: one of the *selection_filters*. The default picking filter activated on entering the pick mode. All available filters are presented in a combobox.

draw.**set_edit_mode**(*s*)
Set the drawing edit mode.

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.

`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.

`draw.`**`exportWebGL`**`(`*fn*, *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.

Returns the absolute pathname of the generated .html file.

`draw.`**`showURL`**`(`*url*`)`

Show an URL in the browser

- *url*: url to load

`draw.`**`showHTML`**`(`*fn*`)`

Show a local .html file in the browser

- *fn*: name of a local .html file

`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.

6.1.6 `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.iteritems():
...     print("%12s = %s -> %0.3f" % (k,v,luminance(v)))
       black = (0.0, 0.0, 0.0) -> 0.000
         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
    darkgrey = (0.5, 0.5, 0.5) -> 0.214
     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
    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
```

`Classes defined in module colors`

`Functions defined in module colors`

`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 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.

Examples: >>> GLcolor('indianred') (0.803921568627451, 0.3607843137254902, 0.3607843137254902) >>> 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)

`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 tuple.

`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 tuple.

`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.

`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).

Examples: >>> colorName('red') '#ff0000' >>> colorName('#ffddff') '#ffddff' >>> colorName([1.,0.,0.5]) '#ff0080'

`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']
```

colors.**closestColorName**(*color*)
Return the closest color name.

colors.**RGBA**(*rgb*, *alpha=1.0*)
Adds an alpha channel to an RGB color

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 `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.

`Classes defined in module geometry`

class geometry.**Geometry**
A generic geometry object allowing transformation of coords sets.

The Geometry class is a generic parent class for all geometric classes, intended to make the Coords transformations available without explicit declaration. This class is not intended to be used directly, only through derived classes. Examples of derived classes are `Formex`, `Mesh`, `Curve`.

There is no initialization to be done when constructing a new instance of this class. The class just defines a set of methods which operate on the attribute *coords*, which should be a Coords object. Most of the transformation methods of the Coords class are thus exported through the Geometry class to its derived classes, and when called, will get executed on the *coords* attribute. The derived class constructor should make sure that the *coords* attribute exists, has the proper type and contains the coordinates of all the points that should get transformed under a Coords transformation.

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. Most derived classes that are part of pyFormex however override this default and implement a more efficient copy method.

The following Geometry methods return the value of the same method applied on the *coords* attribute. Refer to the correponding coords.Coords method for their precise arguments.

x(), y(), z(), bbox(), center(), centroid(), sizes(), dsize(), bsphere(), inertia(), distanceFromPlane(), distanceFromLine(), distanceFromPoint(), directionalSize(), directionalWidth(), directionalExtremes(), __str__().

The following Coords transformation methods can be directly applied to a Geometry object or a derived class object. The return value is a new object identical to the original, except for the coordinates, which will have been transformed by the specified method. Refer to the correponding coords.Coords method in for the precise arguments of these methods:

scale(), translate(), centered(), rotate(), shear(), reflect(), affine(), position(), cylindrical(), hyperCylindrical(), toCylindrical(), spherical(), superSpherical(), toSpherical(), bump(), bump1(), bump2(), flare(), map(), map1(), mapd(), replace(), swapAxes(), rollAxes(), projectOnPlane(), projectOnSphere(), projectOnCylinder(), isopar(), transformCS(), addNoise(), rot(), trl().

nelems()

Return the number of elements in the Geometry.

This method should be re-implemented by the derived classes. For the (empty) Geometry class it always returns 0.

setProp(*prop=None*, *blocks=None*)

Create or destroy the property array for the Geometry.

A property array is a rank-1 integer array with dimension 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. They play an import role when creating new geometry: new elements inherit the property number of their parent element. Properties are also preserved on most geometrical 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*: a single integer value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

  The special value 'range' will set the property numbers equal to the element number.

  A value None (default) removes the properties from the Geometry.

- *blocks*: a single integer value or a list/array of integer values. If the number of passed values is less than the length of *prop*, they wil be repeated. If you give more, they will be ignored. Every prop will be repeated the corresponding number of times specified in blocks.

toProp(*prop*)

Converts the argument into a legal set of properties for the object.

The conversion involves resizing the argument to a 1D array of length self.nelems(), and converting the data type to integer.

getCoords()
Get the coords data.

Returns the full array of coordinates stored in the Geometry object. Note that subclasses may store more points in this array than are used to define the geometry.

level()
Return the dimensionality of the Geometry, or -1 if unknown

copy()
Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

splitProp(*prop=None*)
Partition a Geometry (Formex/Mesh) according to the values in prop.

Parameters:

- *prop*: an int array with length self.nelems(), or None. If None, the *prop* attribute of the Geometry is used.

Returns a list of Geometry objects of the same type as the input. Each object 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 their *prop* value.

It *prop* is None and the the object has no *prop* attribute, an empty list is returned.

scale(**args*, ***kargs*)
Apply '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.

translate(**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

centered(**args*, ***kargs*)
Apply 'centered' transformation to the Geometry object.

See `coords.Coords.centered()` for details.

align(**args*, ***kargs*)
Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

rotate(**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

shear (**args*, ***kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

reflect (**args*, ***kargs*)
Apply 'reflect' transformation to the Geometry object.

See `coords.Coords.reflect()` for details.

affine (**args*, ***kargs*)
Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

position (**args*, ***kargs*)
Apply 'position' transformation to the Geometry object.

See `coords.Coords.position()` for details.

cylindrical (**args*, ***kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See `coords.Coords.cylindrical()` for details.

hyperCylindrical (**args*, ***kargs*)
Apply 'hyperCylindrical' transformation to the Geometry object.

See `coords.Coords.hyperCylindrical()` for details.

toCylindrical (**args*, ***kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

spherical (**args*, ***kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args*, ***kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

egg (**args*, ***kargs*)
Apply 'egg' transformation to the Geometry object.

See `coords.Coords.egg()` for details.

toSpherical (**args*, ***kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

bump (**args*, ***kargs*)
Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

bump1 (**args*, ***kargs*)
Apply 'bump1' transformation to the Geometry object.

See `coords.Coords.bump1()` for details.

bump2 (**args*, ***kargs*)
Apply 'bump2' transformation to the Geometry object.

See `coords.Coords.bump2()` for details.

flare (**args*, ***kargs*)
Apply 'flare' transformation to the Geometry object.

See `coords.Coords.flare()` for details.

map (**args*, ***kargs*)
Apply 'map' transformation to the Geometry object.

See `coords.Coords.map()` for details.

map1 (**args*, ***kargs*)
Apply 'map1' transformation to the Geometry object.

See `coords.Coords.map1()` for details.

mapd (**args*, ***kargs*)
Apply 'mapd' transformation to the Geometry object.

See `coords.Coords.mapd()` for details.

replace (**args*, ***kargs*)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

swapAxes (**args*, ***kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

rollAxes (**args*, ***kargs*)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

projectOnPlane (**args*, ***kargs*)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (**args*, ***kargs*)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

projectOnCylinder (**args*, ***kargs*)
Apply 'projectOnCylinder' transformation to the Geometry object.

See `coords.Coords.projectOnCylinder()` for details.

isopar (**args*, ***kargs*)
Apply 'isopar' transformation to the Geometry object.

See `coords.Coords.isopar()` for details.

transformCS (**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

addNoise (**args*, ***kargs*)
Apply 'addNoise' transformation to the Geometry object.

See `coords.Coords.addNoise()` for details.

rot (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

trl (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

write (*fil*, *sep=' '*, *mode='w'*)
Write a Geometry to a .pgf file.

If fil is a string, a file with that name is opened. Else fil should be an open file. The Geometry is then written to that file in a native format, using sep as separator between the coordinates. If fil is a string, the file is closed prior to returning.

`Functions defined in module geometry`

6.2.2 `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 refering 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.

`Classes defined in module connectivity`

class `connectivity.`**`Connectivity`**
A class for handling element/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.

The current implementation limits a Connectivity object to numbers that are smaller than 2**31.

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.

A new Connectivity object is created with the following syntax

```
Connectivity(data=[],dtyp=None,copy=False,nplex=0)
```

Parameters:

- *data*: should be compatible with an integer array with shape *(nelems,nplex)*, where *nelems* is the number of elements and *nplex* is the plexitude of the elements.
- *dtype*: can be specified to force an integer type but is set by default from the passed *data*.

- *copy*: can be set True to force copying the data. By default, the specified data will be used without copying, if possible.
- *nplex*: can be specified to force a check on the plexitude of the data, or to set the plexitude for an empty Connectivity. An error will be raised if the specified data do not match the specified plexitude. If an *eltype* is specified, the plexitude of the element type will override this value.
- *eltype*: an Element type (a subclass of `Element`) or the name of an Element type, or None (default). If the Connectivity will be used to create a Mesh, the proper element type or name should be set: either here or at Mesh creation time. If the Connectivity will be used for other purposes, the element type may be not important.

Example:

```
>>> print(Connectivity([[0,1,2],[0,1,3],[0,3,2],[0,5,3]]))
[[0 1 2]
 [0 1 3]
 [0 3 2]
 [0 5 3]]
```

nelems()

Return the number of elements in the Connectivity table.

Example:

```
>>> 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.

This returns the highest node number plus one.

nnodes()

Return the actual number of nodes in the connectivity.

This returns the count of the unique node numbers.

nplex()

Return the plexitude of the elements in the Connectivity table.

Example:

```
>>> 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 a boolean array with shape (self.nelems(),). The True values flag the degenerate rows.

Example:

```
>>> Connectivity([[0,1,2],[0,1,1],[0,3,2]]).testDegenerate()
array([False,  True, False], dtype=bool)
```

listDegenerate()

Return a list with the numbers of the degenerate elements.

Example:

```
>>> 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.

Example:

```
>>> Connectivity([[0,1,2],[0,1,1],[0,3,2]]).listNonDegenerate()
array([0, 2])
```

removeDegenerate()

Remove the degenerate elements from a Connectivity table.

Degenerate elements are rows with repeating values.

Returns a Connectivity with the degenerate elements removed.

Example:

```
>>> Connectivity([[0,1,2],[0,1,1],[0,3,2]]).removeDegenerate()
Connectivity([[0, 1, 2],
       [0, 3, 2]])
```

reduceDegenerate(*target=None*)

Reduce degenerate elements to lower plexitude elements.

This will try to reduce the degenerate elements of the Connectivity to a lower plexitude. This is only possible if an element type was set in the Connectivity. This function uses the data of the Element database in `elements`.

If a target element type is given, only the reductions to that element type are performed. Else, all the target element types for which a reduction scheme is available, will be tried.

Returns:

> A list of Connectivities of which the first one contains the originally non-degenerate elements and the last one contains the elements that could not be reduced and may be empty. If the original Connectivity does not have an element type set, or the element type does not have any reduction schemes defined, a list with only the original is returned.

Note: If the Connectivity is part of a Mesh, you should use the Mesh.reduceDegenerate method instead, as that one will preserve the property numbers into the resulting Meshes.

Example:

```
>>> C = Connectivity([[0,1,2],[0,1,1],[0,3,2]],eltype='line3')
>>> print(C.reduceDegenerate())
```

```
[Connectivity([[0, 1]]), Connectivity([[0, 1, 2],
       [0, 3, 2]])]
```

testDuplicate (*permutations=True*, *return_multiplicity=False*)

Test the Connectivity list for duplicates.

By default, duplicates are elements that consist of the same set of nodes, in any particular order. Setting permutations to False will only find the duplicate rows that have matching values at every position.

Returns a tuple of two arrays and optionally a dictionary:

- an index used to sort the elements
- a flags array with the value True for indices of the unique elements and False for those of the duplicates.
- if return_multiplicity is True, returns also an extra dict with multiplicities as keys and a list of elements as value.

Example:

```
>>> conn = Connectivity([[0,1,3],[2,3,0],[0,2,3],[0,1,2],[0,2,1],[0,3,2]])
>>> print(conn)
[[0 1 3]
 [2 3 0]
 [0 2 3]
 [0 1 2]
 [0 2 1]
 [0 3 2]]
>>> ind,ok,D = conn.testDuplicate(return_multiplicity=True)
>>> print(ind,ok)
[3 4 0 1 2 5] [ True False  True  True False False]
>>> print(ok.cumsum())
[1 1 2 3 3 3]
>>> print(D)
{1: array([0]), 2: array([3, 4]), 3: array([1, 2, 5])}
```

listUnique (*permutations=True*)

Return a list with the numbers of the unique elements.

Example:

```
>>> Connectivity([[0,1,2],[0,2,1],[0,3,2]]).listUnique()
array([0, 2])
```

listDuplicate (*permutations=True*)

Return a list with the numbers of the duplicate elements.

Example:

```
>>> Connectivity([[0,1,2],[0,2,1],[0,3,2]]).listDuplicate()
array([1])
```

removeDuplicate (*permutations=True*)

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 False will only remove the duplicate rows that have matching values at matching positions.

Returns a new Connectivity with the duplicate elements removed.

Example:

```
>>> Connectivity([[0,1,2],[0,2,1],[0,3,2]]).removeDuplicate()
Connectivity([[0, 1, 2],
       [0, 3, 2]])
```

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 requirements.

Parameters:

- *order*: 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 of 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:

A 1-D integer array which is 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()`.

Example:

```
>>> A = Connectivity([[1,2],[2,3],[3,0],[0,1]])
>>> A[A.reorder('reverse')]
Connectivity([[0, 1],
       [3, 0],
       [2, 3],
       [1, 2]])
>>> A.reorder('nodes')
array([3, 2, 0, 1])
>>> 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.

Returns a tuple:

- *elems*: the renumbered connectivity
- *oldnrs*: 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*.

Example:

```
>>> e,n = Connectivity([[0,2],[1,4],[4,2]]).renumber(7)
>>> print(e,n)
[[ 7  9]
 [ 8 10]
 [10  9]] [0 1 2 4]
```

inverse()

Return the inverse index of a Connectivity table.

Returns the inverse index of the Connectivity, as computed by `arraytools.inverseIndex()`.

Example:

```
>>> 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]])
```

nParents()

Return the number of elements connected to each node.

Returns a 1-D int array with 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.

Example:

```
>>> 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.

- *nodes*: a single node number or a list/array thereof,
- *return_ncon*: if True, also return the number of connections for each element.

Returns an int array with the numbers of the elements that contain at least one of the specified nodes. If *return_ncon* is True, also returns an int array giving the number of connections for each connected element.

Example:

```
>>> A = Connectivity([[0,1,2],[0,1,3],[0,3,2],[1,2,3]])
>>> A.connectedTo(2)
array([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.

nodes: a single node number or a list/array thereof

Returns an (nelems,) shaped int array with the number of nodes from the list that are contained in each of the elements.

Note that this information can also be got from meth:*connectedTo*. This method however exapnds the results to the full element set, making it apt for use in selector expressions like:

```
self[self.hits(nodes) >= 2]
```

Example:

```
>>> 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*)

Return a table of adjacent items.

Create an element adjacency table (kind='e') or node adjacency table (kind='n').

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', requesting resp. element or node adjacency.
- *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. 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:

```
self[mask].adjacency('n')
```

Returns:

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.

Example:

```
>>> 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]])
```

selectNodes(*selector*)

Return a `Connectivity` containing subsets of the nodes.

Parameters:

- *selector*: 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.

Returns:

A `Connectivity` object with shape `(self.nelems*selector.nelems,selector.nplex)`. This function does not collapse the duplicate elements. The eltype of the result is equal to that of the selector, possibly None.

Example:

```
>>> 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=True*)

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. Multiple intermediate level entities may be created from each element.

Parameters:

- *selector*: an object that can be converted to a 1-dim or 2-dim integer 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*: *bool* . If True, rows which are permutations of the same data are considered equal.

  If the Connectivity has an element type, selector can also be a single integer specifying one of the hierarchical levels of element entities (See the Element class). In that case the selector is constructed automatically from self.eltype.getEntities(selector).

Returns:

- *hi*: a `Connectivity` defining the original elements in function of the intermediate level ones,
- *lo*: 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.

All intermediate level items that consist of the same set of nodes in any permutation order and with any multiplicity, are considered identical and are collapsed into single items if permutations is *True*. The resulting node numbering of the created intermediate entities (the *lo* return value) respects the numbering order of the original elements and applied the selector, but 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.getBorder()` for an application where an inverse operation is possible, because the border only contains unique rows. See also `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.

Example:

```
>>> Connectivity([[0,1,2],[0,2,1],[0,3,2]]).insertLevel([[0,1],[1,2],[2,0]])
(Connectivity([[0, 3, 1],
       [1, 3, 0],
       [2, 4, 1]]), Connectivity([[0, 1],
       [2, 0],
       [0, 3],
       [1, 2],
       [3, 2]]))
 >>> Connectivity([[0,1,2,3]]).insertLevel([[0,1,2],[1,2,3],[0,1,1],[0,0,1],[1,
 (Connectivity([[1, 2, 0, 0, 0]]), Connectivity([[0, 1, 1],
        [0, 1, 2],
        [1, 2, 3]]))
```

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 vertex numbers of an element are unique.

Example:

```
>>> hi,lo = Connectivity([[0,1,2],[0,2,1],[0,3,2]]).          insertLevel([[0,1
>>> 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.

Example:

```
>>> 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*)

Return the list of nodes shared by all elements in elist

Parameters:

- *elist*: an integer list-like with element numbers.

Returns a 1-D integer array with the list of nodes that are common to all elements in the specified list. This array may be empty.

`Functions defined in module connectivity`

connectivity.**findConnectedLineElems**(*elems*)

Find a single path of connected line elems.

This function is intended as a helper function for `connectedLineElems()`. It should probably not be used directly, because, as a side-effect, it changes the data in the *elems* argument. `connectedLineElems()` does not have this inconvenience.

The function searches a Connectivity table for a chain of elements in which the first node of all but the first element is equal to the last node of the previous element. To create such a chain, elements may be reordered and the node sequence of an element may be reversed.

Parameters:

- *elems*: Connectivity-like. Any plexitude is allowed, but only the first and the last column are relevant.

Returns:

- *con*: a Connectivity with the same shape as the input Connectivity *elems*, holding a single chain extracted from the input and filled with -1 for the remainder (if any). The chain will not necessarily be the longest path. It will however at least contain the first element of the input table.
- *inv*: an int array with two columns and number of rows equal to that of *con*. The first column holds the row number in *elems* of the entries in *con*. The second column holds a value +1 or -1, flagging whether the element is traversed in original direction (+1) in the chain or in the reverse direction (-1).

Example:

```
>>> con,inv = findConnectedLineElems([[0,1],[1,2],[0,4],[4,2]])
>>> print(con)
[[0 1]
 [1 2]
 [2 4]
 [4 0]]
>>> print(inv)
[[ 0  1]
 [ 1  1]
 [ 3 -1]
 [ 2 -1]]

>>> con,inv = findConnectedLineElems([[0,1],[1,2],[0,4]])
>>> print(con)
[[2 1]
 [1 0]
 [0 4]]
>>> print(inv)
[[ 1 -1]
 [ 0 -1]
 [ 2  1]]

>>> C = Connectivity([[0,1],[0,2],[0,3],[4,5]])
>>> con,inv = findConnectedLineElems(C)
>>> print(con)
[[ 1  0]
 [ 0  2]
 [-1 -1]
 [-1 -1]]
>>> print(inv)
[[ 0 -1]
 [ 1  1]
 [-1  0]
 [-1  0]]
>>> print(C)
[[-1 -1]
```

```
[-1 -1]
[ 0  3]
[ 4  5]]
```

connectivity.**connectedLineElems**(*elems*, *return_indices=False*)

Partition a collection of line segments into connected polylines.

The input argument is a (nelems,2) shaped array of integers. Each row holds the two vertex numbers of a single line segment.

The return value is a list of Connectivity tables of plexitude 2. The line elements of each Connectivity are ordered to form a continuous connected segment, i.e. the first vertex of each line element in a table is equal to the last vertex of the previous element. The connectivity tables are sorted in order of decreasing length.

If return_indices = True, a second list of tables is returned, with the same shape as those in the first list. The tables of the second list contain in the first column the original element number of the entries, and in 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).

Example:

```
>>> connectedLineElems([[0,1],[1,2],[0,4],[4,2]])
[Connectivity([[0, 1],
       [1, 2],
       [2, 4],
       [4, 0]])]

>>> connectedLineElems([[0,1],[1,2],[0,4]])
[Connectivity([[2, 1],
       [1, 0],
       [0, 4]])]

>>> connectedLineElems([[0,1],[0,2],[0,3],[4,5]])
[Connectivity([[1, 0],
       [0, 2]]), Connectivity([[4, 5]]), Connectivity([[0, 3]])]

>>> connectedLineElems([[0,1],[0,2],[0,3],[4,5]])
[Connectivity([[1, 0],
       [0, 2]]), Connectivity([[4, 5]]), Connectivity([[0, 3]])]

>>> connectedLineElems([[0,1],[0,2],[0,3],[4,5]],True)
([Connectivity([[1, 0],
       [0, 2]]), Connectivity([[4, 5]]), Connectivity([[0, 3]])], [array([[ 0, -1],
       [ 1,  1]], dtype=int32), array([[3, 1]], dtype=int32), array([[2, 1]], dtype
>>> connectedLineElems([[0,1,2],[2,0,3],[0,3,1],[4,5,2]])
[Connectivity([[3, 0, 2],
       [2, 1, 0],
       [0, 3, 1]]), Connectivity([[4, 5, 2]])]
```

Obviously, from the input elems table and the second return value, the first return value could be reconstructed:

```
first = [
    where(i[:,-1:] > 0, elems[i[:,0]], elems[i[:,0],::-1])
    for i in second
]
```

But since the construction of the first list is required by the algorithm, it is returned anyway.

`connectivity.`**`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.

Example:

```
>>> adjacencyArrays([[0,1],[1,2],[2,3],[3,4],[4,0]],3)
[array([[0],
       [1],
       [2],
       [3],
       [4]]), Adjacency([[1, 4],
       [0, 2],
       [1, 3],
       [2, 4],
       [0, 3]]), array([[2, 3],
       [3, 4],
       [0, 4],
       [0, 1],
       [1, 2]]), array([], shape=(5, 0), dtype=int64)]
```

6.2.3 `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.

`Classes defined in module adjacency`

class `adjacency.`**`Adjacency`**

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.

A new Adjacency table is created with the following syntax

```
Adjacency(data=[],dtyp=None,copy=False,ncon=0,normalize=True)
```

Parameters:

- *data*: should be compatible with an integer array with shape *(nelems,ncon)*, where *nelems* is the number of elements and *ncon* is the maximum number of connections per element.
- *dtyp*: can be specified to force an integer type but is set by default from the passed *data*.
- *copy*: can be set True to force copying the data. By default, the specified data will be used without copying, if possible.
- *ncon*: can be specified to force a check on the plexitude of the data, or to set the plexitude for an empty Connectivity. An error will be raised if the specified data do not match the specified plexitude.
- *normalize*: boolean: if True (default) the Adjacency will be normalized at creation time.
- *allow_self*: boolean: if True, connections of elements with itself are allowed. The default (False) will remove self-connections when the table is normalized.

Example:

```
>>> print Adjacency([[1,2,-1],
...                   [3,2,0],
...                   [1,-1,3],
...                   [1,2,-1],
...                   [-1,-1,-1]])
[[-1  1  2]
 [ 0  2  3]
 [-1  1  3]
 [-1  1  2]
 [-1 -1 -1]]
```

nelems()

Return the number of elements in the Adjacency table.

maxcon()

Return the maximum number of connections for any element.

This returns the row count of the Adjacency.

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 at least one row without -1 value.

By default, an Adjacency is 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: an integer array with shape (adj.shape[0],maxc), with `maxc <= adj.shape[1]`, where row *i* retains the unique non-negative numbers of the original array except the value *i*, and is possibly padded with -1 values.

Example:

```
>>> 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]])
>>> 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 an integer array with two columns, where each row contains a pair of adjacent elements. The element number in the first columne is always the smaller of the two element numbers.

symdiff(*adj*)

Return the symmetric difference of two adjacency tables.

Parameters:

- *adj*: Adjacency with the same number of rows as *self*.

Returns 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.

frontFactory(*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.

It returns an 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*.

Parameters: see `frontWalk()`.

Example:

```
>>> A = Adjacency([[1,2,-1],
...                 [3,2,0],
...                 [1,-1,3],
...                 [1,2,-1],
...                 [-1,-1,-1]])
>>> for p in A.frontFactory(): 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*: initial element numbers in the front. It can be a single element number or a list of numbers.
- *frontinc*: increment for the front number on each frontal step.
- *partinc*: increment for the front number when the front
- *maxval*: 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: an array of integers specifying for each element in which step the element was reached by the walker.

Example:

```
>>> 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]
```

Functions defined in module adjacency

adjacency.**sortAdjacency** (*adj*)
Sort an adjacency table.

An adjacency table is an integer array where each row lists the numbers of the items that are connected to the item with number equal to the row index. Rows are padded with -1 value to create rows of equal length.

This function sorts the rows of the adjacency table in ascending order and removes all columns containing only -1 values.

Paramaters:

- *adj*: an 2-D integer array with values >=0 or -1

Returns: an integer array with shape (adj.shape[0],maxc), with maxc <= adj.shape[1], where the rows are sorted in ascending order and where columns with only -1 values are removed.

Example:

```
>>> a = array([[ 0,  2,  1, -1],
...            [-1,  3,  1, -1],
...            [ 3, -1,  0,  1],
...            [-1, -1, -1, -1]])
>>> sortAdjacency(a)
array([[ 0,  1,  2],
       [-1,  1,  3],
       [ 0,  1,  3],
       [-1, -1, -1]])
```

adjacency.**reduceAdjacency**(*adj*)

Reduce an adjacency table.

An adjacency table is an integer array where each row lists the numbers of the items that are connected to the item with number equal to the row index. Rows are padded with -1 values to create rows of equal length.

A reduced 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 at least one row without -1 value.

Paramaters:

- *adj*: an 2-D integer array with value >=0 or -1

Returns: an integer array with shape (adj.shape[0],maxc), with maxc <= adj.shape[1], where row *i* retains the unique non-negative numbers of the original array except the value *i*, and is possibly padded with -1 values.

Example:

```
>>> a = array([[ 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]])
>>> reduceAdjacency(a)
array([[ 1,  2,  5],
       [-1,  0,  3],
       [-1, -1,  0],
       [-1, -1,  1],
       [-1, -1, -1],
       [-1, -1,  0]])
```

6.2.4 `elements` — Definition of elements.

This modules allows for a consistent local numbering scheme of element connectivities throughout pyFormex. When interfacing with other programs, one should be aware that conversions may be necessary. Conversions to/from external programs should be done by the interface modules.

`Classes defined in module elements`

class `elements.ElementType`

Base class for element type classes.

Element type data are stored in a class derived from ElementType. The derived element type classes contain only static data. No instances of these classes should be created. The base class defines the access methods, which are all class methods.

Derived classes should be created by calling the function `createElementType()`.

Each element is defined by the following attributes:

- *name*: a string. It is capitalized before use, thus all ElementType subclasses have a name starting with an uppercase letter. Usually the name has a numeric last part, equal to the plexitude of the element.
- *vertices*: the natural coordinates of its vertices,
- *edges*: a list of edges, each defined by 2 or 3 node numbers,
- *faces*: a list of faces, each defined by a list of minimum 3 node numbers,
- *element*: a list of all node numbers
- *drawfaces*: a list of faces to be drawn, if different from faces. This is an optional attribute. If defined, it will be used instead of the *faces* attribute to draw the element. This can e.g. be used to draw approximate representations for higher order elements for which there is no correct drawing function.

The vertices of the elements are defined in a unit space [0,1] in each axis direction.

The elements guarantee a fixed local numbering scheme of the vertices. One should however not rely on a specific numbering scheme of edges, faces or elements.

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.

The list of available element types can be found from:

```
>>> printElementTypes()
Available Element Types:
  0-dimensional elements: ['Point']
  1-dimensional elements: ['Line2', 'Line3', 'Line4']
  2-dimensional elements: ['Tri3', 'Tri6', 'Quad4', 'Quad6', 'Quad8', 'Quad9']
  3-dimensional elements: ['Tet4', 'Tet10', 'Tet14', 'Tet15', 'Wedge6', 'Hex8', 'He
```

Optional attributes:

- *conversions*: Defines possible strategies for conversion of the element to other element types. It is a dictionary with the target element name as key, and a list of actions as value. Each action in the list consists of a tuple (action, data), where action is one of the action identifier characters defined below, and data are the data needed for this action.

Conversion actions:

- 'm': add new nodes to the element by taking the mean values of existing nodes. data is a list of tuples containing the nodes numbers whose coorrdinates have to be averaged.
- 's': select nodes from the existing ones. data is a list of the node numbers to retain in the new element. This can be used to reduce the plexitude but also just to reorder the existing nodes.

- 'v': perform a conversion via an intermediate type. data is the name of the intermediate element type. The current element will first be converted to the intermediate type, and then conversion from that type to the target will be attempted.
- 'r': randomly choose one of the possible conversions. data is a list of element names. This can e.g. be used to select randomly between different but equivalent conversion paths.

classmethod nplex ()
Return the plexitude of the element

classmethod nvertices ()
Return the plexitude of the element

classmethod nnodes ()
Return the plexitude of the element

classmethod getEntities (*level*, *reduce=False*)
Return the type and connectivity table of some element entities.

The full list of entities with increasing dimensionality 0,1,2,3 is:

```
['points', 'edges', 'faces', 'cells' ]
```

If level is negative, the dimensionality returned is relative to the highest dimensionality (.i.e., that of the element). If it is positive, it is taken absolute.

Thus, for a 3D element type, getEntities(-1) returns the faces, while for a 2D element type, it returns the edges. For both types however, getLowerEntities(+1) returns the edges.

The return value is a dict where the keys are element types and the values are connectivity tables. If reduce == False: there will be only one connectivity table and it may include degenerate elements. If reduce == True, an attempt is made to reduce the degenerate elements. The returned dict may then have multiple entries.

If the requested entity level is outside the range 0..ndim, the return value is None.

classmethod getDrawFaces (*quadratic=False*)
Returns the local connectivity for drawing the element's faces

classmethod toMesh ()
Convert the element type to a Mesh.

Returns a Mesh with a single element of natural size.

classmethod toFormex ()
Convert the element type to a Formex.

Returns a Formex with a single element of natural size.

classmethod 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.

`Functions defined in module elements`

`elements.`**elementType** (*name=None*, *nplex=-1*)
Return the requested element type

Parameters:

- *name*: a string (case ignored) with the name of an element. If not specified, or the named element does not exist, the default element for the specified plexitude is returned.
- *nplex*: plexitude of the element. If specified and no element name was given, the default element type for this plexitude is returned.

Returns a subclass of `ElementType`.

Errors:

If neither *name* nor *nplex* can resolve into an element type, an error is raised.

Example:

```
>>> elementType('tri3').name()
'tri3'
>>> elementType(nplex=2).name()
'line2'
```

`elements.`**`elementTypes`**(*ndim=None*, *lower=True*)

Return the names of available elements.

If a value is specified for ndim, only the elements with the matching dimensionality are returned.

`elements.`**`printElementTypes`**(*lower=False*)

Print all available element types.

Prints a list of the names of all available element types, grouped by their dimensionality.

6.2.5 `mesh` — Finite element meshes in pyFormex.

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.

In the Mesh geometrical data model, 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 the coordinates array.

This model has some advantages over the Formex data model (which stores all the points of all the elements by their coordinates):

- a more compact storage, because coordinates of coinciding points are not repeated,
- faster connectivity related algorithms.

The downside is that geometry generating algorithms are far more complex and possibly slower.

In pyFormex we therefore mostly use the Formex data model when creating geometry, but when we come to the point of exporting the geometry to file (and to other programs), a Mesh data model may be more adequate.

The Mesh data model has at least the following attributes:

- *coords*: (ncoords,3) shaped Coords object, holding the coordinates of all points in the Mesh;

- *elems*: (nelems,nplex) shaped Connectivity object, defining the elements by indices into the Coords array. All values in elems should be in the range 0 <= value < ncoords.
- *prop*: an array of element property numbers, default None.
- *eltype*: an element type (a subclass of `Element`) or the name of an Element type, or None (default). If eltype is None, the eltype of the elems Connectivity table is used, and if that is missing, a default eltype is derived from the plexitude, by a call to `elements.elementType()`. In most cases the eltype can be set automatically. The user can override the default value, but an error will occur if the element type does not exist or does not match the plexitude.

A Mesh can be initialized by its attributes (coords,elems,prop,eltype) or by a single geometric object that provides a toMesh() method.

If only an element type is provided, a unit sized single element Mesh of that type is created. Without parameters, an empty Mesh is created.

setType(*eltype=None*)
Set the eltype from a character string.

This function allows the user to change the element type of the Mesh. The input is a character string with the name of one of the element defined in elements.py. The function will only allow to set a type matching the plexitude of the Mesh.

This method is seldom needed, because the applications should normally set the element type at creation time.

elType()
Return the element type of the Mesh.

elName()
Return the element name of the Mesh.

setNormals(*normals=None*)
Set/Remove the normals of the mesh.

getProp()
Return the properties as a numpy array (ndarray)

maxProp()
Return the highest property value used, or None

propSet()
Return a list with unique property values.

shallowCopy(*prop=None*)
Return a shallow copy.

A shallow copy of a Mesh is a Mesh object using the same data arrays as the original Mesh. The only thing that can be changed is the property array. This is a convenient method to use the same Mesh with different property attributes.

toFormex()
Convert a Mesh to a Formex.

The Formex inherits the element property numbers and eltype from the Mesh. Node property numbers however can not be translated to the Formex data model.

toMesh()
Convert to a Mesh.

This just returns the Mesh object itself. It is provided as a convenience for use in functions that want 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. The resulting TriSurface is only fully equivalent with the input Mesh if the latter has element type 'tri3'.

On success, returns a TriSurface corresponding with the input Mesh. If the Mesh can not be converted to a TriSurface, an error is raised.

toCurve()
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)

This is equivalent with

```
self.toFormex().toCurve()
```

Any other type will raise an exception.

nedges()
Return the number of edges.

This returns the number of rows that would be in getEdges(), without actually constructing the edges. The edges are not fused!

info()
Return short info about the Mesh.

This includes only the shape of the coords and elems arrays.

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. (Though numpy will limit the printed output).

TODO: We should add an option here to let numpy print the full tables.

centroids()
Return the centroids of all elements of the Mesh.

The centroid of an element is the point whose coordinates are the mean values of all points of the element. The return value is a Coords object with nelems points.

bboxes()
Returns the bboxes of all elements in the Mesh.

Returns a coords with shape (nelems,2,3). Along the axis 1 are stored the minimal and maximal values of the Coords in each of the elements of the Mesh.

getCoords()
Get the coords data.

Returns the full array of coordinates stored in the Mesh object. Note that this may contain points that are not used in the mesh. `compact()` will remove the unused points.

getElems()
Get the elems data.

Returns the element connectivity data as stored in the object.

getLowerEntities(*level=-1*, *unique=False*)
Get the entities of a lower dimensionality.

If the element type is defined in the `elements` module, this returns a Connectivity table with the entities of a lower dimensionality. The full list of entities with increasing dimensionality 0,1,2,3 is:

```
['points', 'edges', 'faces', 'cells' ]
```

If level is negative, the dimensionality returned is relative to that of the caller. If it is positive, it is taken absolute. 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.

By default, all entities for all elements are returned and common entities will appear multiple times. Specifying unique=True will return only the unique ones.

The return value may be an empty table, if the element type does not have the requested entities (e.g. the 'point' type). If the eltype is not defined, or the requested entity level is outside the range 0..3, the return value is None.

getNodes()
Return the set of unique node numbers in the Mesh.

This returns only the node numbers that are effectively used in the connectivity table. For a compacted Mesh, it is equivalent to `arange(self.nelems)`. This function also stores the result internally so that future requests can return it without the need for computing it again.

getPoints()
Return the nodal coordinates of the Mesh.

This returns only those points that are effectively 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.

This is a convenient function to create a table with the element edges. It is equivalent to `self.getLowerEntities(1,unique=True)`, but this also stores the result internally so that future requests can return it without the need for computing it again.

getFaces()

Return the unique faces of all the elements in the Mesh.

This is a convenient function to create a table with the element faces. It is equivalent to `self.getLowerEntities(2,unique=True)`, but this also stores the result internally so that future requests can return it without the need for computing it again.

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.

getElemEdges()

Defines the elements in function of its edges.

This returns a Connectivity table with the elements defined in function of the edges. It returns the equivalent of `self.elems.insertLevel(self.elType().getEntities(1))` but as a side effect it 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 border of the Mesh.

Returns a Connectivity table with the free entities of the specified level of the Mesh. Free entities are entities that are only connected with a single element.

If return_indices==True, also returns an (nentities,2) index for inverse lookup of the higher entity (column 0) and its local lower entity number (column 1).

getFreeEntitiesMesh(*level=-1*, *compact=True*)

Return a Mesh with lower entities.

Returns a Mesh representing 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.

By default, the resulting Mesh is compacted. Compaction can be switched off by setting *compact=False*.

getBorder(*return_indices=False*)

Return the border of the Mesh.

This returns a Connectivity table with the border of the Mesh. The border entities are of a lower hierarchical level than the mesh itself. These entities become part of the border if they are connected to only one element.

If return_indices==True, it returns also an (nborder,2) index for inverse lookup of the higher entity (column 0) and its local border part number (column 1).

This is a convenient shorthand for

```
self.getFreeEntities(level=-1,return_indices=return_indices)
```

getBorderMesh(*compact=True*)

Return a Mesh with the border elements.

The returned Mesh is of the next lower hierarchical level and contains all the free entitites of that level. If the Mesh has property numbers, the border elements inherit the property of the

element to which they belong.

By default, the resulting Mesh is compacted. Compaction can be switched off by setting *compact=False*.

This is a convenient shorthand for

```
self.getFreeEntitiesMesh(level=-1,compact=compact)
```

getBorderElems()
Return the elements that are on the border of the Mesh.

This returns a list with the numbers of the elements that are on the border of the Mesh. Elements are considered to be at the border if they contain at least one complete element of the border Mesh (i.e. an element of the first lower hierarchical level). Thus, in a volume Mesh, elements only touching the border by a vertex or an edge are not considered border elements.

getBorderNodes()
Return the nodes that are on the border of the Mesh.

This returns a list with the numbers of the nodes that are on the border of the Mesh.

peel(*nodal=False*)
Return a Mesh with the border elements removed.

If nodal is True all elements connected to a border node are removed. If nodal is False, it is a convenient shorthand for

```
self.cselect(self.getBorderElems())
```

getFreeEdgesMesh(*compact=True*)
Return a Mesh with the free edge elements.

The returned Mesh is of the hierarchical level 1 (no mather what the level of the parent Mesh is) and contains all the free entitites of that level. If the Mesh has property numbers, the border elements inherit the property of the element to which they belong.

By default, the resulting Mesh is compacted. Compaction can be switched off by setting *compact=False*.

This is a convenient shorthand for

```
self.getFreeEntitiesMesh(level=1,compact=compact)
```

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. The level is one of the lower entities of the mesh.

Parameters:

- *level*: 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.
- *diflevel*: if >= level, and smaller than the hierarchy of self.elems, 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 an Adjacency with integers 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 simultanuously 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*: 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.

The remainder of the parameters are like in `Adjacency.frontWalk()`.

Returns an array of integers specifying for each element in which step the element was reached by the walker.

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.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 (poly)line from a point in one element to a point in the other element without leaving the Mesh. The partitioning is returned as a integer array having a value for ech 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.

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(***arg*)

Partition a surface 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.partitionByAngle()`.

Currently this only works for 'tri3' and 'quad4' type Meshes. Also, the 'quad4' partitioning method currently only works correctly if the quads are nearly planar.

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 tgerms 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(***kargs*)

Fuse the nodes of a Meshes.

All nodes that are within the tolerance limits of each other are merged into a single node.

The merging operation can be tuned by specifying extra arguments that will be passed to `Coords:fuse()`.

matchCoords(*mesh*, ***kargs*)

Match nodes of Mesh with nodes of self.

This is a convenience function equivalent to:

```
self.coords.match(mesh.coords,**kargs)
```

See also `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()

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.

Example:

```
>>> x = Coords([[i] for i in arange(5)])
>>> M = Mesh(x,[[0,2],[1,4],[4,2]])
>>> M = M.compact()
>>> 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]]
```

select (*selected*, *compact=True*)

Return a Mesh only holding the selected elements.

Parameters:

- *selected*: an object that can be used as an index in the *elems* array, such as
  - a single element number
  - a list, or array, of element numbers
  - a bool array of length self.nelems(), where True values flag the elements to be selected
- *compact*: boolean. If True (default), the returned Mesh will be compacted, i.e. the unused nodes are removed and the nodes are renumbered from zero. If False, returns the node set and numbers unchanged.

Returns a Mesh (or subclass) with only the selected elements.

See `cselect()` for the complementary operation.

cselect (*selected*, *compact=True*)

Return a mesh without the selected elements.

Parameters:

- *selected*: an object that can be used as an index in the *elems* array, such as
  - a single element number
  - a list, or array, of element numbers
  - a bool array of length self.nelems(), where True values flag the elements to be selected
- *compact*: boolean. If True (default), the returned Mesh will be compacted, i.e. the unused nodes are removed and the nodes are renumbered from zero. If False, returns the node set and numbers unchanged.

Returns a Mesh with all but the selected elements.

This is the complimentary operation of `select()`.

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.

withProp (*val*)
Return a Mesh which holds only the elements with property val.

val is either a single integer, or a list/array of integers. The return value is a Mesh holding all the elements that have the property val, resp. one of the values in val. The returned Mesh inherits the matching properties.

If the Mesh has no properties, a copy with all elements is returned.

withoutProp (*val*)
Return a Mesh without the elements with property *val*.

This is the complementary method of Mesh.withProp(). val is either a single integer, or a list/array of integers. The return value is a Mesh holding all the elements that do not have the property val, resp. one of the values in val. The returned Mesh inherits the matching properties.

If the Mesh has no properties, a copy with all elements is returned.

connectedTo (*nodes*)
Return a Mesh with the elements connected to the specified node(s).

nodes: int or array_like, int.

Return a Mesh with all the elements from the original that contain at least one of the specified nodes.

notConnectedTo (*nodes*)
Return a Mesh with the elements not connected to the given node(s).

nodes: int or array_like, int.

Returns a Mesh with all the elements from the original that do not contain any of the specified nodes.

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 elementt'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 `Mesh.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*)
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.

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'.

reduceDegenerate(*eltype=None*)
Reduce degenerate elements to lower plexitude elements.

This will try to reduce the degenerate elements of the mesh to elements of a lower plexitude. If a target element type is given, only the matching reduce scheme is tried. Else, all the target element types for which a reduce scheme from the Mesh eltype is available, will be tried.

The result is a list of Meshes of which the last one contains the elements that could not be reduced and may be empty. Property numbers propagate to the children.

splitDegenerate (*autofix=True*)
Split a Mesh in degenerate and non-degenerate elements.

If autofix is 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.

The return value is a list of Meshes. 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. These may still contain degenerate elements.

removeDegenerate (*eltype=None*)
Remove the degenerate elements from a Mesh.

Returns a Mesh with all degenerate elements removed.

removeDuplicate (*permutations=True*)
Remove the duplicate elements from a Mesh.

Duplicate elements are elements that consist of the same nodes, by default in no particular order. Setting permutations=False will only consider elements with the same nodes in the same order as duplicates.

Returns a Mesh with all duplicate elements removed.

renumber (*order='elems'*)
Renumber the nodes of a Mesh in the specified order.

order is an index with length equal to the number of nodes. 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 value 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.

reorder (*order='nodes'*)
Reorder the elements of a Mesh.

Parameters:

- *order*: either a 1-D integer array with a permutation of `arange(self.nelems())`, specifying the requested order, or 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 a Mesh equivalent with self but with the elements ordered as specified.

See also: `Connectivity.reorder()`

renumberElems (*order='nodes'*)
Reorder the elements of a Mesh.

Parameters:

- *order*: either a 1-D integer array with a permutation of `arange(self.nelems())`, specifying the requested order, or 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 a Mesh equivalent with self but with the elements ordered as specified.

See also: `Connectivity.reorder()`

connect (*coordslist*, *div=1*, *degree=1*, *loop=False*, *eltype=None*)

Connect a sequence of toplogically 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 toplogically 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*: Either an integer, or a sequence of float numbers (usually in the range]0.0..1.0]) or a list of sequences of the same length of the connecting list of coordinates. In the latter case every sequence inside the list can either be a float sequence (usually in the

range]0.0..1.0]) or it contains one integer (e.g [[4],[0.3,1]]). This should only be used for degree==1.

With this parameter the generated elements can be further subdivided along the connection direction. If an int is given, the connected elements will be divided into this number of elements along the connection direction. If a sequence of float numbers is given, the numbers specify the relative distance along the connection direction where the elements should end. If the last value in the sequence is not equal to 1.0, there will be a gap between the consecutive connections. If a list of sequences is given, every consecutive element of the coordinate list is connected using the corresponding sequence in the list(1-length integer of float sequence specified as before).

- *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 (*n*, *step=1.0*, *dir=0*, *degree=1*, *eltype=None*)
Extrude a Mesh in one of the axes directions.

Returns a new Mesh obtained by extruding the given Mesh over *n* steps of length *step* in direction of axis *dir*.

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

Returns a new Mesh obtained by sweeping the given Mesh over a path. The returned Mesh has double plexitude of the original.

This method accepts all the parameters of `coords.sweepCoords()`, with the same meaning. Usually, you will need to at least set the *normal* parameter. The *eltype* parameter can be used to set the element type on the returned Meshes.

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 seperately 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 exluded from smoothing.

-weight [it is a string that can assume 2 values *inversedistance* and] *distance*. It allows to specify the weight of the adjancent points according to their distance to the point

classmethod **`concatenate`**(*clas*, *meshes*, ***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: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 compaitble 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.

`clip`(*t*, *compact=True*)

Return a Mesh with all the elements where t>0.

t should be a 1-D integer array with length equal to the number of elements of the Mesh. The resulting Mesh will contain all elements where t > 0.

cclip (*t*, *compact=True*)
This is the complement of clip, returning a Mesh where t<=0.

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)

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 succesful, 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 unsuccesful.

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 succesful, 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 * surface of base * 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.

addNoise(**args*, ***kargs*)
Apply 'addNoise' transformation to the Geometry object.

See `coords.Coords.addNoise()` for details.

affine(**args*, ***kargs*)
Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

align(**args*, ***kargs*)
Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

bump(**args*, ***kargs*)
Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

bump1(**args*, ***kargs*)
Apply 'bump1' transformation to the Geometry object.

See `coords.Coords.bump1()` for details.

bump2(**args*, ***kargs*)
Apply 'bump2' transformation to the Geometry object.

See `coords.Coords.bump2()` for details.

centered(**args*, ***kargs*)
Apply 'centered' transformation to the Geometry object.

See `coords.Coords.centered()` for details.

cylindrical (**args, **kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See `coords.Coords.cylindrical()` for details.

egg (**args, **kargs*)
Apply 'egg' transformation to the Geometry object.

See `coords.Coords.egg()` for details.

flare (**args, **kargs*)
Apply 'flare' transformation to the Geometry object.

See `coords.Coords.flare()` for details.

hyperCylindrical (**args, **kargs*)
Apply 'hyperCylindrical' transformation to the Geometry object.

See `coords.Coords.hyperCylindrical()` for details.

isopar (**args, **kargs*)
Apply 'isopar' transformation to the Geometry object.

See `coords.Coords.isopar()` for details.

map (**args, **kargs*)
Apply 'map' transformation to the Geometry object.

See `coords.Coords.map()` for details.

map1 (**args, **kargs*)
Apply 'map1' transformation to the Geometry object.

See `coords.Coords.map1()` for details.

mapd (**args, **kargs*)
Apply 'mapd' transformation to the Geometry object.

See `coords.Coords.mapd()` for details.

position (**args, **kargs*)
Apply 'position' transformation to the Geometry object.

See `coords.Coords.position()` for details.

projectOnCylinder (**args, **kargs*)
Apply 'projectOnCylinder' transformation to the Geometry object.

See `coords.Coords.projectOnCylinder()` for details.

projectOnPlane (**args, **kargs*)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (**args, **kargs*)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

replace (**args, **kargs*)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

rollAxes (**args*, ***kargs*)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

rot (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

rotate (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

scale (**args*, ***kargs*)
Apply 'scale' transformation to the Geometry object.

See `coords.Coords.scale()` for details.

shear (**args*, ***kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

spherical (**args*, ***kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args*, ***kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

swapAxes (**args*, ***kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

toCylindrical (**args*, ***kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

toSpherical (**args*, ***kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

transformCS (**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

translate (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

trl (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

setProp (*prop=None*, *blocks=None*)

Create or destroy the property array for the Geometry.

A property array is a rank-1 integer array with dimension 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. They play an import role when creating new geometry: new elements inherit the property number of their parent element. Properties are also preserved on most geometrical 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*: a single integer value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

  The special value 'range' will set the property numbers equal to the element number.

  A value None (default) removes the properties from the Geometry.

- *blocks*: a single integer value or a list/array of integer values. If the number of passed values is less than the length of *prop*, they wil be repeated. If you give more, they will be ignored. Every prop will be repeated the corresponding number of times specified in blocks.

toProp (*prop*)

Converts the argument into a legal set of properties for the object.

The conversion involves resizing the argument to a 1D array of length self.nelems(), and converting the data type to integer.

copy ()

Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

splitProp (*prop=None*)

Partition a Geometry (Formex/Mesh) according to the values in prop.

Parameters:

- *prop*: an int array with length self.nelems(), or None. If None, the *prop* attribute of the Geometry is used.

Returns a list of Geometry objects of the same type as the input. Each object 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 their *prop* value.

It *prop* is None and the the object has no *prop* attribute, an empty list is returned.

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.

write (*fil*, *sep=' '*, *mode='w'*)
Write a Geometry to a .pgf file.

If fil is a string, a file with that name is opened. Else fil should be an open file. The Geometry is then written to that file in a native format, using sep as separator between the coordinates. If fil is a string, the file is closed prior to returning.

`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.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:fuse()`. Setting fuse=False will merely concatenate all the mesh.coords, but not fuse them.

`mesh.`**unitAttractor** (*x*, *e0=0.0*, *e1=0.0*)
Moves values in the range 0..1 closer to or away from the limits.

- *x*: a list or array with values in the range 0.0 to 1.0.
- *e0*, *e1*: attractor parameters for the start, resp. the end of the range. A value larger than zero will attract the points closer to the corresponding endpoint, while a negative value will repulse them. If two positive values are given, the middle part of the interval will become sparsely populated.

Example:

```
>>> set_printoptions(precision=4)
>>> print(unitAttractor([0.,0.25,0.5,0.75,1.0],2.))
[ 0.      0.0039  0.0625  0.3164  1.    ]
```

mesh.**seed**(*n*, *e0=0.0*, *e1=0.0*)

Create one-dimensional element seeds in a unit interval

Returns parametric values along the unit interval in order to divide it in n elements, possibly of unequal length.

Parameters:

- *n*: number of elements (yielding n+1 parameter values).
- *e0*, *e1*: attractor parameters for the start, resp. the end of the range. A value larger than zero will attract the points closer to the corresponding endpoint, while a negative value will repulse them. If two positive values are given, the middle part of the interval will become sparsely populated.

Example:

```
>>> set_printoptions(precision=4)
>>> print(seed(5,2.,2.))
[ 0.      0.0639  0.3362  0.6638  0.9361  1.    ]
```

mesh.**gridpoints**(*seed0*, *seed1=None*, *seed2=None*)

Create weigths for linear lines, quadrilateral and hexahedral elements coordinates

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 parametes are integer values the divisions will be equally spaced between 0 and 1

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 parametes are integer values the divisions will be equally spaced between 0 and 1

mesh.**quadgrid**(*seed0*, *seed1*)

Create a quadrilateral mesh of unit size with the specified seeds.

The seeds are a monotonously increasing series of parametric values in the range 0..1. They define the positions of the nodes in the parametric directions 0, resp. 1. Normally, the first and last values of the seeds are 0., resp. 1., leading to a unit square grid.

The seeds are usually generated with the seed() function.

`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 parametes 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*, *W*, *nl*, *nw*)
Create a plane rectangular mesh of quad4 elements

Parameters:

- L,W: length,width of the rectangle
- nl,nw: seeds for the elements along the length, width of the rectangle. They should 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 float values in the range 0.0 to 1.0, specifying the relative position of the seeds. The values should be ordered and the first and last values should be 0.0 and 1.0.

`mesh.`**`rectangle_with_hole`**(*L*, *W*, *r*, *nr*, *nt*, *e0=0.0*, *eltype='quad4'*)
Create a quarter of rectangle with a central circular hole.

Parameters:

- L,W: length,width of the (quarter) rectangle
- r: radius of the hole
- nr,nt: number of elements over radial,tangential direction
- e0: concentration factor for elements in the radial direction

Returns a Mesh

6.2.6 `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.

`Classes defined in module simple`

`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.`**`regularGrid`**(*x0*, *x1*, *nx*)

Create a regular grid between points x0 and x1.

x0 and x1 are n-dimensional points (usually 1D, 2D or 3D). The space between x0 and x1 is divided in nx equal parts. nx should have the same dimension as x0 and x1. The result is a rectangular grid of coordinates in an array with shape (nx[0]+1, nx[1]+1, ..., n).

`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 a 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 rectangluar 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' of 'x', diagonals are added in /, resp. and both directions, to form 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*.
- *a3*: 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 $a3 < 360$, the result is an arc. Large values of *a1* and *a2* result in polygones. Thus *circle(120.)* is an equilateral triangle and *circle(60.)* is regular hexagon.

Remark that the default a2 == a1 produces a continuous line, while a2 > a1 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.`**`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 [x0,x1] and [x1,x2] will be divided in n segments.

`simple.`**`sphere`**`(`*ndiv=6*`)`

Create a triangulated spherical surface.

A high quality approximation of a spherical surface is constructed as follows. First an icosahedron is constructed. 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. *ndiv=1* results in an icosahedron.

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*, *top=90*`)`

Return a sphere consisting of surface triangles

A sphere with radius r is modeled by the triangles fromed 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.

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 rectangular 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 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 can be used to visualize the bboxes() of a geometry.

simple.**cuboid**(*xmin=[0.0, 0.0, 0.0], xmax=[1.0, 1.0, 1.0]*)

Create a rectangular prism

Creates a rectangular prism with faces parallel to the global axes through the points xmin and xmax.

Returns a single element Formex with eltype 'hex8'.

6.2.7 project — project.py

Functions for managing a project in pyFormex.

Classes defined in module project

class project.**Project**(*filename=None, access='wr', convert=True, signature='pyFormex 0.9.1 (0.9.1)', compression=5, binary=True, data={}, **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 (currently) not deal with other data types however.

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 with 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})
>>> import tempfile
>>> f = tempfile.mktemp('.pyf','w')
>>> P = Project(f)
>>> P.update(d)
>>> print dict.__str__(P)
{'a': 1, 'c': 3, 'b': 2, 'e': {'g': 5, 'f': 4}, 'd': [1, 2, 3]}
>>> P.save(quiet=True)
>>> P.clear()
>>> print dict.__str__(P)
{}
>>> P.load(quiet=True)
>>> print dict.__str__(P)
{'a': 1, 'c': 3, 'b': 2, 'e': {'g': 5, 'f': 4}, 'd': [1, 2, 3]}
```

header_data()

Construct the data to be saved in the header.

save (*quiet=False*)
Save the project to file.

readHeader (*quiet=False*)
Read the header from a project file.

Tries to read the header from different legacy formats, and if succesfull, adjusts the project attributes according to the values in the header. Returns the open file if succesfull.

load (*try_resolve=False, 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 successfull, 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 tha saved data can be read back immediately.

uncompress ()
Uncompress a compressed project file.

The project file is read, and if successfull, is written back in uncompressed format. This allows to make conversions of the data inside.

delete ()
Unrecoverably delete the project file.

pop (**args, **kw*)
Wrapper function for a class method.

popitem (**args, **kw*)
Wrapper function for a class method.

setdefault (**args, **kw*)
Wrapper function for a class method.

update (**args, **kw*)
Wrapper function for a class method.

`Functions defined in module project`

`project.`**find_global** (*module, name*)
Override the import path of some classes

`project.`**pickle_load** (*f, try_resolve=True*)
Load data from pickle file f.

6.2.8 `utils` — A collection of miscellaneous utility functions.

`Classes defined in module utils`

class `utils.`**NameSequence** (*name, ext=''*)
A class for autogenerating sequences of names.

The name is a string including a numeric part, which is incremented at each call of the 'next()' method.

The constructor takes name template and a possible extension as arguments. If the name starts with a non-numeric part, it is taken as a constant part. If the name ends with a numeric part, the next generated names will be obtained by incrementing this part. If not, a string '-000' will be appended and names will be generated by incrementing this part.

If an extension is given, it will be appended as is to the names. This makes it possible to put the numeric part anywhere inside the names.

Example:

```
>>> N = NameSequence('abc.98')
>>> [ N.next() for i in range(3) ]
['abc.98', 'abc.99', 'abc.100']
>>> N = NameSequence('abc-8x.png')
>>> [ N.next() for i in range(3) ]
['abc-8x.png', 'abc-9x.png', 'abc-10x.png']
>>> NameSequence('abc','.png').next()
'abc-000.png'
>>> N = NameSequence('/home/user/abc23','5.png')
>>> [ N.next() for i in range(2) ]
['/home/user/abc235.png', '/home/user/abc245.png']
```

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.

files(*sort=<function hsorted at 0x4f48d70>*)
Return a (sorted) list of files matching the name 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.

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.**splitFilename**(*filename*, *accept_ext=None*, *reject_ext=None*)
Split a file name in dir,base,ext tuple.

Parameters:

- *filename*: a filename, possibly including a directory path
- *accept_ext*: optional list of acceptable extension strings. If specified, only extensions appearing in this list will be recognized as such, while other ones will be made part of the base name.
- *reject_ext*: optional list of unacceptable extension strings. If specified, extensions appearing in this list will not be recognized as such, and be made part of the base name.

Returns a tuple dir,base,ext:

- *dir*: the directory path, not including the final path separator (unless it is the only one). If the filename starts with a single path separator, *dir* will consist of that single separator. If the filename does not contain a path separator, *dir* will be an empty string.
- *base*: filename without the extension. It can only be empty if the input is an empty string.
- *ext*: file extension: This is the part of the filename starting from the last '.' character that is not the first character of the filename. If the filename does not contain a '.' character or the only '.' character is also the first character of the filename (after the last path separator), the extension is an empty string. If not empty, it always starts with a '.'. A filename with

Examples:

```
>>> splitFilename("cc/dd/aa.bb")
('cc/dd', 'aa', '.bb')
>>> splitFilename("cc/dd/aa.")
('cc/dd', 'aa', '.')
>>> splitFilename("..//aa.bb")
('..', 'aa', '.bb')
>>> splitFilename("aa.bb")
('', 'aa', '.bb')
>>> splitFilename("aa/bb")
('aa', 'bb', '')
>>> splitFilename("aa/bb/")
('aa/bb', '', '')
>>> splitFilename("/aa/bb")
('/aa', 'bb', '')
>>> splitFilename(".bb")
('', '.bb', '')
>>> splitFilename("/")
('/', '', '')
>>> splitFilename(".")
('', '.', '')
>>> splitFilename("")
('', '', '')
>>> splitFilename("cc/dd/aa.bb",accept_ext=['.aa','.cc'])
```

```
('cc/dd', 'aa.bb', '')
>>> splitFilename("cc/dd/aa.bb",reject_ext=['.bb'])
('cc/dd', 'aa.bb', '')
```

utils.**buildFilename** (*dirname*, *basename*, *ext=''*)

Build a filename from a directory path, filename and optional extension.

The dirname and basename are joined using the system path separator, and the extension is added at the end. Note that no '.' is added between the basename and the extension. While the extension will normally start with a '.', this function can also be used to add another tail to the filename.

This is a convenience function equivalent with:

```
os.path.join(dirname,basename) + ext
```

utils.**changeExt** (*filename*, *ext*, *accept_ext=None*, *reject_ext=None*)

Change the extension of a file name.

This function splits the specified file name in a base name and an extension, replaces the extension with the specified one, and returns the reassembled file name. If the filename has no extension part, the specified extension is just appended.

Parameters:

- *fn*: file name, possibly including a directory path and extension
- *ext*: string: required extension of the output file name. The string should start with a '.'.
- *accept_ext*, *reject_ext*: lists of strings starting with a '.'. These have the same meaning as in `splitFilename()`.

Returns a file name with the specified extension.

Example:

```
>>> changeExt('image.png','.jpg')
'image.jpg'
>>> changeExt('image','.jpg')
'image.jpg'
>>> changeExt('image','jpg') # Deprecated
'image.jpg'
>>> changeExt('image.1','.jpg')
'image.jpg'
>>> changeExt('image.1','.jpg',reject_ext=['.1'])
'image.1.jpg'
```

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 splitFilename(fn)[1]

utils.**tildeExpand** (*fn*)

Perform tilde expansion on a filename.

Bash, the most used command shell in Linux, expands a '~' in arguments to the users home direction. This function can be used to do the same for strings that did not receive the bash tilde expansion, such as strings in the configuration file.

utils.**all_image_extensions** ()

Return a list with all known image extensions.

`utils.`**`fileDescription`**`(`*ftype*`)`
Return a description of the specified file type.

The description of known types are listed in a dict file_description. If the type is unknown, the returned string has the form `TYPE files (*.type)`

`utils.`**`fileType`**`(`*ftype*`)`
Normalize a filetype string.

The string is converted to lower case and a leading dot is removed. This makes it fit for use with a filename extension.

Example:

```
>>> fileType('pdf')
'pdf'
>>> fileType('.pdf')
'pdf'
>>> fileType('PDF')
'pdf'
>>> fileType('.PDF')
'pdf'
```

`utils.`**`fileTypeFromExt`**`(`*fname*`)`
Derive the file type from the file name.

The derived file type is the file extension part in lower case and without the leading dot.

Example:

```
>>> fileTypeFromExt('pyformex.pdf')
'pdf'
>>> fileTypeFromExt('pyformex')
''
>>> fileTypeFromExt('pyformex.pgf')
'pgf'
>>> fileTypeFromExt('pyformex.pgf.gz')
'pgf.gz'
>>> fileTypeFromExt('pyformex.gz')
'gz'
```

`utils.`**`fileSize`**`(`*fn*`)`
Return the size in bytes of the file fn

`utils.`**`findIcon`**`(`*name*`)`
Return the file name for an icon with given name.

If no icon file is found, returns the question mark icon.

`utils.`**`prefixFiles`**`(`*prefix, files*`)`
Prepend a prefix to a list of filenames.

`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 targes matches none of the regular expressions.

`utils.`**`matchAll`**(*regexps*, *target*)
Check whether targets matches all of the regular expressions.

`utils.`**`listTree`**(*path*, *listdirs=True*, *topdown=True*, *sorted=False*, *excludedirs=[ ]*, *excludefiles=[ ]*, *includedirs=[ ]*, *includefiles=[ ]*, *symlinks=True*)
List all files in path.

If `listdirs==False`, directories are not listed. By default the tree is listed top down and entries in the same directory are unsorted.

exludedirs and *excludefiles* are lists of regular expressions with dirnames, resp. filenames to exclude from the result.

includedirs and *includefiles* can be given to include only the directories, resp. files matching any of those patterns.

Note that 'excludedirs' and 'includedirs' force top down handling.

If *symlinks* is set False, symbolic links are removed from the list.

`utils.`**`removeFile`**(*filename*)
Remove a file, ignoring error when it does not exist.

`utils.`**`removeTree`**(*path*, *top=True*)
Remove all files below path. If top==True, also path is removed.

`utils.`**`sourceFiles`**(*relative=False*, *symlinks=True*, *extended=False*)
Return a list of the pyFormex source .py files.

- *symlinks*: if False, files that are symbolic links are retained in the list. The default is to remove them.
- *extended*: if True, the .py files in all the paths in the configured appdirs and scriptdirs are also added.

`utils.`**`grepSource`**(*pattern*, *options=''*, *relative=True*, *quiet=False*)
Finds pattern in the pyFormex source .py 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.`**`setSaneLocale`**(*localestring=''*)
Set a sane local configuration for LC_NUMERIC.

localestring is the locale string to be set, e.g. 'en_US.UTF-8'

This will change the `LC_ALL` setting to the specified string, and set the `LC_NUMBERIC` 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 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.

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.

utils.**gzip**(*filename*, *gzipped=None*, *remove=True*, *level=5*)
Compress a file in gzip format.

Parameters:

- *filename*: input file name
- *gzipped*: output file name. If not specified, it will be set to the input file name + '.gz'. 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 compression level. Higher values result in smaller files, but require longer compression times. The default of 5 gives already a fairly good compression ratio.

Returns the name of the compressed file.

utils.**gunzip**(*filename*, *unzipped=None*, *remove=True*)
Uncompress a file in gzip format.

Parameters:

- *filename*: compressed input file name (usually ending in '.gz')
- *unzipped*: output file name. If not specified and *filename* ends with '.gz', it will be set to the *filename* with the '.gz' removed. If an empty string is specified or it is not specified and the filename does not end in '.gz', 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.

Returns the name of the decompressed file.

utils.**mtime**(*fn*)
Return the (UNIX) time of last change of file fn.

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.**system1**(*cmd*)
Execute an external command.

utils.**system**(*cmd*, *timeout=None*, *gracetime=2.0*, *shell=True*)
Execute an external command.

Parameters:

- *cmd*: a string with the command to be executed
- *timeout*: float. If specified and > 0.0, the command will time out and be killed after the specified number of seconds.
- *gracetime*: float. The time to wait after the terminate signal was sent in case of a timeout, before a forced kill is done.
- *shell*: if True (default) the command is run in a new shell

Returns:

- *sta*: exit code of the command. In case of a timeout this will be *utils._TIMEOUT_EXITCODE*, or *utils._TIMEOUT_KILLCODE* if the command had to be forcedly killed. Otherwise, the exitcode of the command itself is returned.
- *out*: stdout produced by the command
- *err*: stderr produced by the command

utils.**runCommand**(*cmd*, *timeout=None*, *verbose=True*)
Run an external command in a user friendly way.

This uses the `system()` function to run an external command, adding some extra user notifications of what is happening. If no error occurs, the (sta,out) obtained form the `system()` function are returned. The value sta will be zero, unless a timeout condition has occurred, in which case sta will be -15 or -9. If the `system()` call returns with an error that is not a timeout,

Parameters:

- *cmd*: a string with the command to be executed
- *timeout*: float. If specified and > 0.0, the command will time out and be killed after the specified number of seconds.

- *verbose*: boolean. If True (default), a message including the command is printed before it is run and in case of a nonzero exit, the full stdout, exit status and stderr are printed (in that order).

If no error occurs in the execution of the command by the `system()` function, returns a tuple

- *sta*: 0, or a negative value in case of a timeout condition
- *out*: stdout produced by the command, with the last newline removed

Example: cmd = 'sleep 2' sta,out=runCommand3(cmd,quiet=False, timeout=5.) print (sta,out)

`utils.`**`spawn`**`(`*cmd*`)`
Spawn a child process.

`utils.`**`killProcesses`**`(`*pids*, *signal=15*`)`
Send the specified signal to the processes in list

- *pids*: a list of process ids.
- *signal*: the signal to send to the processes. The default (15) will try to terminate the process. See 'man kill' for more values.

`utils.`**`userName`**`()`
Find the name of the user.

`utils.`**`is_pyFormex`**`(`*appname*`)`
Checks whether an application name is rather a script name

`utils.`**`is_script`**`(`*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.`**`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. Joined together, the substrings restore the original. The even items only contain digits. 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.`**`hsorted`**`(`*l*`)`
Sort a list of strings in human order.

When human sort a list of strings, they tend to interprete 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.`**`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.`**`prefixDict`**(*d*, *prefix=''*)

Prefix all the keys of a dict with the given prefix.

- *d*: a dict where all the keys are strings.
- *prefix*: a string

The return value is a dict with all the items of d, but where the keys have been prefixed with the given string.

`utils.`**`subDict`**(*d*, *prefix=''*, *strip=True*)

Return a dict with the items whose key starts with prefix.

- *d*: a dict where all the keys are strings.
- *prefix*: a string
- *strip*: if True (default), the prefix is stripped from the keys.

The return value is 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.

`utils.`**`selectDict`**(*d*, *keys*)

Return a dict with the items whose key is in keys.

- *d*: a dict where all the keys are strings.
- *keys*: a set of key values, can be a list or another dict.

The return value is a dict with all the items from d whose key is in keys. See removeDict() for the complementary operation.

Example:

```
>>> d = dict([(c,c*c) for c in range(6)])
>>> selectDict(d,[4,0,1])
{0: 0, 1: 1, 4: 16}
```

utils.**removeDict**(*d*, *keys*)
Remove a set of keys from a dict.

- *d*: a dict
- *keys*: a set of key values

The return value is a dict with all the items from *d* whose key is not in *keys*. This is the complementary operation of selectDict.

Example:

```
>>> 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.**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.**sortedKeys**(*d*)
Returns the sorted keys of a dict.

It is required that the keys of the dict be sortable, e.g. all strings or integers.

utils.**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.

`utils.`**`listFontFiles`**`()`
: List all fonts known to the system.

  Returns a list of path names to all the font files found on 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: OtherContainerClass.get_elements}

  Adapted from http://code.activestate.com/recipes/577504/

6.2.9 `geomtools` — Basic geometrical operations.

This module defines some basic operations on simple geometrical entities such as lines, triangles, circles, planes.

`Classes defined in module geomtools`

`Functions defined in module geomtools`

`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.`**`levelVolumes`**`(`*x*`)`
: Compute the level volumes of a collection of elements.

  x is an (nelems,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 (nelems,) shaped float array.

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.**distance**(*X*, *Y*)

Returns the distance of all points of X to those of Y.

Parameters:

- *X*: (nX,3) shaped array of points.
- *Y*: (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*, *return_dist=True*)

Find the point of Y closest to points of X.

Parameters:

- *X*: (nX,3) shaped array of points
- *Y*: (nY,3) shaped array of points
- *return_dist*: bool. If False, only the index of the closest point is returned. If True (default), the distance are also returned.

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.

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. It 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`**(*P1*, *D1*, *P2*, *D2*)

Finds the intersection of 2 coplanar lines.

The lines (P1,D1) and (P2,D2) are defined by a point and a direction vector. Let a and b be unit vectors along the lines, and c = P2-P1, let ld and d be the length and the unit vector of the cross product a*b, the intersection point X is then given by X = 0.5(P1+P2+sa*a+sb*b) where sa = det([c,b,d])/ld and sb = det([c,a,d])/ld

`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 foreward, 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. 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.`**`pointsAtLines`**`(`*q*, *m*, *t*`)`

Return the points of lines (q,m) at parameter values t.

Parameters:

- *q*,'m': (...,3) shaped arrays of points and vectors, defining a single line or a set of lines.
- *t*: array of parameter values, broadcast compatible with *q* and *m*.

Returns an 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*: (...,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.`**`intersectionTimesLWL`**`(`*q1*, *m1*, *q2*, *m2*, *mode='all'*`)`

Find the common perpendicular of lines (q1,m1) and lines (q2,m2)

For non-intersecting lines, the base points of the common perpendicular are found. For intersecting lines, the common point of intersection is found.

Lines are defined by a point (q) and a vector (m).

Parameters:

- *qi*,'mi' (i=1...2): (nqi,3) shaped arrays of points and vectors (*mode=all*) or broadcast compatible arrays (*mode=pair*), defining a single line or a set of lines.
- *mode*: *all* to calculate the intersection of each line (q1,m1) with all lines (q2,m2) or *pair* for pairwise intersections.

Returns a tuple of (nq1,nq2) shaped (*mode=all*) arrays of parameter values t1 and t2, such that the intersection points are given by `q1+t1*m1` and `q2+t2*m2`.

`geomtools.`**`intersectionPointsLWL`**`(`*q1*, *m1*, *q2*, *m2*, *mode='all'*`)`

Return the intersection points of lines (q1,m1) and lines (q2,m2)

with the perpendiculars between them.

This is like intersectionTimesLWL but returns a tuple of (nq1,nq2,3) shaped (*mode=all*) arrays of intersection points instead of the parameter values.

`geomtools.`**`intersectionTimesLWP`**`(`*q*, *m*, *p*, *n*, *mode='all'*`)`

Return the intersection of lines (q,m) with 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 intersections.

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.

geomtools.**intersectionPointsLWP**(*q*, *m*, *p*, *n*, *mode='all'*)
Return the intersection points of lines (q,m) with planes (p,n).

This is like intersectionTimesLWP but returns a (nq,np,3) shaped (*mode=all*) array of intersection points instead of the parameter values.

geomtools.**intersectionTimesSWP**(*S*, *p*, *n*, *mode='all'*)
Return the intersection of line segments S with planes (p,n).

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.
- *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[...,0,:] + t*S[...,1,:]*.

This function is comparable to intersectionTimesLWP, but ensures that parameter values 0<=t<=1 are points inside the line segments.

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 <= 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 <= 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[...,0,:] + t*S[...,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 <= 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.**intersectionTimesPOP**(*X*, *p*, *n*, *mode='all'*)
Return the intersection of perpendiculars from points X on planes (p,n).

Parameters:

- *X*: a (nX,3) shaped array of points (*mode=all*) or broadcast compatible array (*mode=pair*).
- *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 for each point X with all planes (p,n) or *pair* for pairwise intersections.

Returns a (nX,np) shaped (*mode=all*) array of parameter values t, such that the intersection points are given by X+t*n.

`geomtools.`**`intersectionPointsPOP`**(*X*, *p*, *n*, *mode='all'*)

Return the intersection points of perpendiculars from points X on planes (p,n).

This is like intersectionTimesPOP but returns a (nX,np,3) shaped (*mode=all*) array of intersection points instead of the parameter values.

`geomtools.`**`intersectionTimesPOL`**(*X*, *q*, *m*, *mode='all'*)

Return the intersection of perpendiculars from points X on lines (q,m).

Parameters:

- *X*: a (nX,3) shaped array of points (*mode=all*) or broadcast compatible array (*mode=pair*).
- *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.
- *mode*: *all* to calculate the intersection for each point X with all lines (q,m) or *pair* for pairwise intersections.

Returns a (nX,nq) shaped (*mode=all*) array of parameter values t, such that the intersection points are given by q+t*m.

`geomtools.`**`intersectionPointsPOL`**(*X*, *q*, *m*, *mode='all'*)

Return the intersection points of perpendiculars from points X on lines (q,m).

This is like intersectionTimesPOL but returns a (nX,nq,3) shaped (*mode=all*) array of intersection points instead of the parameter values.

`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 <= 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.`**`distancesPFL`**(*X*, *q*, *m*, *mode='all'*)

Return the distances of points X from lines (q,m).

Parameters:

- *X*: a (nX,3) shaped array of points (*mode=all*) or broadcast compatible array (*mode=pair*).
- *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.
- *mode*: *all* to calculate the distance of each point X from all lines (q,m) or *pair* for pairwise distances.

Returns a (nX,nq) shaped (*mode=all*) array of distances.

`geomtools.`**`distancesPFS`**(*X*, *S*, *mode='all'*)

Return the distances of points X from line segments S.

Parameters:

- *X*: a (nX,3) shaped array of points (*mode=all*) or broadcast compatible array (*mode=pair*).
- *S*: (nS,2,3) shaped array of line segments (*mode=all*) or broadcast compatible array (*mode=pair*), defining a single line segment or a set of line segments.

- *mode*: *all* to calculate the distance of each point X from all line segments S or *pair* for pairwise distances.

Returns a (nX,nS) shaped (*mode=all*) array of distances.

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.

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.**baryCoords**(*S*, *P*)
Compute the barycentric coordinates of points P wrt. simplexes S.

S is a (nel,nplex,3) shaped array of n-simplexes (n=nplex-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 (nplex,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.

6.2.10 `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.

`Classes defined in module fileread`

`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_stl_bin`**(*fn*)
Read a binary stl.

Returns a Coords with shape (ntri,4,3). The first item of each triangle is the normal, the other three are the vertices.

`fileread.`**`read_gambit_neutral`**(*fn*)
Read a triangular surface mesh in Gambit neutral format.

The .neu file nodes are numbered from 1! Returns a nodes,elems tuple.

`fileread.`**`read_gambit_neutral_hex`**(*fn*)
Read an hexahedral mesh in Gambit neutral format.

The .neu file nodes are numbered from 1! Returns a nodes,elems tuple.

6.2.11 `filewrite` — Write geometry to file in a whole number of formats.

This module defines bothe 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.

`Classes defined in module filewrite`

`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 force to type *float32* or *int32* before writing.

  The format string should contain a valid format converter for a 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*, *coords*, *elems*)
Write a mesh of polygons to a file in OFF format.

Parameters:

- *fn*: file name, by preference ending on '.off'
- *coords*: float array with shape (ncoords,3), with the coordinates of *ncoords* vertices.
- *elems*: int array with shape (nelems,nplex), with the definition of *nelems* polygon elements.

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 extermal program 'admesh'.
- *color*: a single color can be passed to a binary STL and will be stpored 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

6.3 pyFormex GUI modules

These modules are located under pyformex/gui.

6.3.1 `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 widgets`

class `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 refered 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 QtGui.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 QtGui.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 in inserted before the label. If the string contains an 'r', a spacer in inserted after the input field. If the string contains a 'c', a spacer in inserted between the label and the input filed.

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,**kargs)
```

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 widgets.**InputInfo**(*name*, *value*, **args*, ***kargs*)
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.

name()
Return the name of the InputItem.

text()
Return the displayed text of the InputItem.

setValue(*val*)
Change the widget's value.

class widgets.**InputLabel**(*name*, *value*, **args*, ***kargs*)
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 ot html. Specify plain=True to display in plain text.

setValue(*val*)
Change the widget's value.

name()
Return the name of the InputItem.

text()
Return the displayed text of the InputItem.

value()
Return the widget's value.

class widgets.**InputString**(*name*, *value*, *max=None*, **args*, ***kargs*)
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()
Select all text on first display.

value()
Return the widget's value.

name()
Return the name of the InputItem.

text()
Return the displayed text of the InputItem.

setValue(*val*)
Change the widget's value.

class 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 ot 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()
Select all text on first display.

value()
Return the widget's value.

setValue(*val*)
Change the widget's value.

name()
Return the name of the InputItem.

text()
Return the displayed text of the InputItem.

class 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.

name()
Return the name of the InputItem.

class 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.

name()
Return the name of the InputItem.

text()
Return the displayed text of the InputItem.

class 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.

name()
Return the name of the InputItem.

text()
Return the displayed text of the InputItem.

class 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.

name()
Return the name of the InputItem.

text()
Return the displayed text of the InputItem.

class widgets.**InputPush**(*name*, *value=None*, *choices=[ ]*, *direction='h'*, *icon=None*, *iconsonly=False*, **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 radio buttons, of which the default will initially be selected. If direction == 'v', the options are in a vbox.

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.

name()
Return the name of the InputItem.

text()
Return the displayed text of the InputItem.

class 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.

name()
Return the name of the InputItem.

text()
Return the displayed text of the InputItem.

class 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.

name()
Return the name of the InputItem.

text()
Return the displayed text of the InputItem.

class widgets.**InputTable**(*name*, *value*, *chead=None*, *rhead=None*, *celltype=None*, *rowtype=None*, *coltype=None*, *edit=True*, *resize=None*, *autowidth=True*, **args*, ***kargs*)
An input item for tabular data.

•*value*: a 2-D array of items, with *nrow* rows and *ncol* columns.

If *value* is an 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.

name()
Return the name of the InputItem.

text()
Return the displayed text of the InputItem.

setValue(*val*)
Change the widget's value.

class widgets.**InputSlider**(*name*, *value*, **args*, ***kargs*)
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 takes a float/integer argument.

name()
Return the name of the InputItem.

text()
Return the displayed text of the InputItem.

show()
Select all text on first display.

value()
Return the widget's value.

setValue(*val*)
Change the widget's value.

class widgets.**InputFSlider**(*name*, *value*, **args*, ***kargs*)
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.

name()
Return the name of the InputItem.

text()
Return the displayed text of the InputItem.

show()
Select all text on first display.

value()
Return the widget's value.

setValue(*val*)
Change the widget's value.

class widgets.**InputPoint**(*name*, *value*, **args*, ***kargs*)
A 3D point/vector input item.

value()
Return the widget's value.

setValue(*val*)
Change the widget's value.

name()
Return the name of the InputItem.

text()
Return the displayed text of the InputItem.

class widgets.**InputIVector**(*name*, *value*, **args*, ***kargs*)
A vector of int values.

value()
Return the widget's value.

setValue(*val*)
Change the widget's value.

name()
Return the name of the InputItem.

text()
Return the displayed text of the InputItem.

class 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 current input value is passed as an argument. 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.

value()
Return the widget's value.

doFunc()
Set the value by calling the button's func

name()
Return the name of the InputItem.

text()
Return the displayed text of the InputItem.

setValue (*val*)
Change the widget's value.

class 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.

value ()
Return the widget's value.

setValue (*value*)
Change the widget's value.

name ()
Return the name of the InputItem.

text ()
Return the displayed text of the InputItem.

class widgets.**InputFont** (*name*, *value*, **args*, ***kargs*)
An input item to select a font.

value ()
Return the widget's value.

name ()
Return the name of the InputItem.

text ()
Return the displayed text of the InputItem.

setValue (*val*)
Change the widget's value.

class widgets.**InputFile** (*name*, *value*, *pattern='*'*, *exist=False*, *multi=False*, *dir=False*, **args*, ***kargs*)
An input item to select a file.

The following arguments are passed to the FileSelection widget: path,pattern,exist,multi,dir.

value ()
Return the widget's value.

name ()
Return the name of the InputItem.

text ()
Return the displayed text of the InputItem.

class 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 ODict.

text()
Return the displayed text.

value()
Return the widget's value.

setValue(*val*)
Change the widget's value.

name()
Return the name of the InputItem.

class 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 widgets.**ScrollForm**
An scrolling 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 widgets.**InputGroup**(*name*, **args*, ***kargs*)
A boxed group of InputItems.

value()
Return the widget's value.

setValue(*val*)
Change the widget's value.

class widgets.**InputTab**(*name*, *tab*, **args*, ***kargs*)
A tab page in an input form.

class 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*, *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 '/'.
- 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. Currentley, 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_tab (*form*, *prefix*, *name*, *items*, ***extra*)
Add a Tab page of input items.

add_group (*form*, *prefix*, *name*, *items*, ***extra*)
Add a group 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 implicitely executed by getResult().

If a timeout is given, start the timeout timer.

acceptData (*result=1*)
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 fuction 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 ot TIMEOUT.

class widgets.**ListWidget** (*maxh=0*)
A customized QListWidget with ability to compute its required size.

class 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 colums. 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.
- *chead*: 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* ais 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=2*)
Set the value of an individual table element.

headerData (*col*, *orientation*, *role*)
Return the header data for the sepcified 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 widgets.**ArrayModel** (*data*, *chead=None*, *rhead=None*, *edit=True*)
A model representing a two-dimensional numpy array.

- *data*: a numpy array with two dimensions.
- *chead*, *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=2*)
Set the value of an individual table element.

headerData (*col*, *orientation*, *role*)
Return the header data for the sepcified row or column

flags (*index*)
Return the TableModel flags.

class 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 an 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 *chead* 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 widgets.**FileSelection**(*path='.'*, *pattern='*'*, *exist=False*, *multi=False*, *dir=False*, *button=None*, ***kargs*)
A file selection dialog.

The FileSelection 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 function `utils.fileDescription()` can be used to create some strings for common classes of files.

  As a convenience, if a string starts with a '.', the remainder of the string will be used as a lookup key in utils.fileDescription to get the actual string to be used. Thus, pattern='.png' will filter all '.png' files, and pattern='.img' will filter all image files in any of the supported formats.

  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.
- *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.

value()
Return the selected value

getFilename(*timeout=None*)
Ask for a filename by user interaction.

Return the filename selected by the user. If the user hits CANCEL or ESC, None is returned.

class widgets.**ProjectSelection**(*path=None*, *pattern=None*, *exist=False*, *compression=4*, *access=None*, *default=None*, *convert=True*)

A file selection dialog specialized for opening projects.

value()

Return the selected value

getFilename(*timeout=None*)

Ask for a filename by user interaction.

Return the filename selected by the user. If the user hits CANCEL or ESC, None is returned.

class 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:

- *Whole Window*: If checked, the whole pyFormex main window will be saved. If unchecked, only the current OpenGL viewport is saved.
- *Crop Root*: If checked, the window will be cropped from the root window. This mode is required if you want to include the window decorations.

value()

Return the selected value

getFilename(*timeout=None*)

Ask for a filename by user interaction.

Return the filename selected by the user. If the user hits CANCEL or ESC, None is returned.

class widgets.**ListSelection**(*choices*, *caption='ListSelection'*, *default=[]*, *single=False*, *check=False*, *sort=False*, **args*, ***kargs*)

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.

getResult()

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.

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_tab (*form*, *prefix*, *name*, *items*, ***extra*)
Add a Tab page of input items.

add_group (*form*, *prefix*, *name*, *items*, ***extra*)
Add a group 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 implicitely executed by getResult().

If a timeout is given, start the timeout timer.

acceptData (*result=1*)
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 fuction 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 ot TIMEOUT.

class 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 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 `getResult()` 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()` mehod to display it.

addCheck(*text*)

Add a check field at the bottom of the layout.

getResult()

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 widgets.**WarningBox**

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 `getResult()` 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()` mehod to display it.

class `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 `widgets.`**`ButtonBox`**(*name='', actions=None, default=None, parent=None, spacer=False, stretch=[-1, -1], cmargin=(2, 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 `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 `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 widgets

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).

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.

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).

widgets.**addTimeOut**(*widget*, *timeout=None*, *timeoutfunc=None*)
Add a timeout to a widget.

- *timeoutfunc* is a callable. If None it will be set to the widget's *timeout* method if one exists.
- *timeout* is a float value. If None, it will be set to to the global *input_timeout*.

If timeout is positive, a timer will be 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 is if the widget is destructed before the timer has finished.

widgets.**defaultItemType**(*item*)
Guess the InputItem type from the value

widgets.**simpleInputItem**(*name*, *value=None*, *itemtype=None*, ***kargs*)
A convenience function to create an InputItem dictionary

widgets.**groupInputItem**(*name*, *items=[]*, ***kargs*)
A convenience function to create an InputItem dictionary

widgets.**tabInputItem**(*name*, *items=[]*, ***kargs*)
A convenience function to create an InputItem dictionary

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.

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

`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!

`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*.

`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.

`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.

`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==None (default), it will be set to the default `[('Cancel',),('OK',)]`.

Specify actions=[] if you want an empty dialogDuttons. 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 horizontal box layout with the buttons.

6.3.2 `menu` — Menus for the pyFormex GUI.

This modules implements specialized classes and functions for building the pyFormex GUI menu system.

`Classes defined in module menu`

class `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)

QtGui.Menu and QtGui.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.

`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 icondir.
    - *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 menu.**Menu**(*title='UserMenu'*, *parent=None*, *before=None*, *tearoff=False*, *items=None*)
A popup/pulldown menu.

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.

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 icondir.

∗*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 `menu.`**`MenuBar`**(*title='TopMenuBar'*)
A menu bar allowing easy menu creation.

`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.

`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 icondir.
    - *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 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 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.

names ()
Return an ordered list of names of the action items.

toolbar (*name*)
Create a new toolbar corresponding to the menu.

`Functions defined in module menu`

menu.**resetWarnings** ()
Reset the warning filters to the default.

menu.**createMenuData** ()
Returns the default pyFormex GUI menu data.

6.3.3 `colorscale` — Color mapping of a range of values.

`Classes defined in module colorscale`

class colorscale.**ColorScale** (*palet='RAINBOW'*, *minval=0.0*, *maxval=1.0*, *midval=None*, *exp=1.0*, *exp2=None*)
Mapping floating point values into colors.

A colorscale maps floating point values within a certain range into colors and can be used to provide visual representation of numerical values. This is e.g. quite useful in Finite Element postprocessing (see the postproc plugin).

The ColorLegend class provides a way to make the ColorScale visible on the canvas.

scale (*val*)
Scale a value to the range -1...1.

If the ColorScale has only one exponent, values in the range mival..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 intevals -1..0 and 0..1.

color (*val*)
Return the color representing a value val.

The returned color is a tuple of three RGB values in the range 0-1. The color is obtained by first scaling the value to the -1..1 range using the 'scale' method, and then using that result to pick a color value from the palet. A palet specifies the three colors corresponding to the -1, 0 and 1 values.

class colorscale.**ColorLegend** (*colorscale*, *n*)
A colorlegend divides a 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.

overflow (*oflow=None*)
Raise a runtime error if oflow == 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.

`Functions defined in module colorscale`

6.3.4 `actors` — OpenGL actors for populating the 3D scene.

`Classes defined in module actors`

class actors.**Actor** (***kargs*)
An Actor is anything that can be drawn in an OpenGL 3D Scene.

The visualisation of the Scene Actors is dependent on camera position and angles, clipping planes, rendering mode and lighting.

An Actor subclass should minimally reimplement the following methods:

- *bbox()*: return the actors bounding box.
- *drawGL(mode)*: to draw the actor. Takes a mode argument so the drawing function can act differently depending on the mode. There are currently 5 modes: wireframe, flat, smooth, flatwire, smoothwire. drawGL should only contain OpenGL calls that are allowed inside a display list. This may include calling the display list of another actor but *not* creating a new display list.

The interactive picking functionality requires the following methods, for which we provide do-nothing defaults here:

- *npoints()*:
- *nelems()*:
- *pickGL()*:

bbox ()
Default implementation for bbox().

drawGL (***kargs*)
Perform the OpenGL drawing functions to display the actor.

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 actors.**TranslatedActor** (*A*, *trl=(0.0, 0.0, 0.0)*, ***kargs*)
An Actor translated to another position.

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 actors.**RotatedActor** (*A*, *rot=(1.0, 0.0, 0.0)*, *twist=0.0*, ***kargs*)
An Actor rotated to another position.

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 actors.**CubeActor** (*size=1.0, color=[(1.0, 0.0, 0.0), (0.0, 1.0, 1.0), (0.0, 1.0, 0.0), (1.0, 0.0, 1.0), (0.0, 0.0, 1.0), (1.0, 1.0, 0.0)], **kargs*)
An OpenGL actor with cubic shape and 6 colored sides.

drawGL (***kargs*)
Draw the cube.

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 actors.**SphereActor** (*size=1.0*, *color=None*, ***kargs*)
An OpenGL actor representing a sphere.

drawGL (***kargs*)
Draw the cube.

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 actors.**BboxActor** (*bbox*, *color=None*, *linewidth=None*, ***kargs*)
Draws a bbox.

drawGL (***kargs*)
Always draws a wireframe model of the bbox.

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 actors.**AxesActor** (*cs=None, size=1.0, psize=0.5, color=[(1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 1.0)], colored_axes=True, draw_planes=True, draw_reverse=True, linewidth=2, alpha=0.5, **kargs*)
An actor showing the three axes of a coordinate system.

If no coordinate system is specified, the global coordinate system is drawn.

The default actor consists of three colored lines of unit length along the unit vectors of the axes and three colored triangles representing the coordinate planes. This can be modified by the following parameters:

size: scale factor for the unit vectors. color: a set of three colors to use for x,y,z axes. colored_axes = False: draw black axes. draw_planes = False: do not draw the coordinate planes.

drawGL (***kargs*)
Draw the axes.

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 actors.**GridActor**(*nx=(1, 1, 1), ox=(0.0, 0.0, 0.0), dx=(1.0, 1.0, 1.0), linecolor=(0.0, 0.0, 0.0), linewidth=None, planecolor=(1.0, 1.0, 1.0), alpha=0.2, lines=True, planes=True, **kargs*)

Draws a (set of) grid(s) in one of the coordinate planes.

drawGL(***kargs*)
    Draw the grid.

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 actors.**CoordPlaneActor**(*nx=(1, 1, 1), ox=(0.0, 0.0, 0.0), dx=(1.0, 1.0, 1.0), linecolor=(0.0, 0.0, 0.0), linewidth=None, planecolor=(1.0, 1.0, 1.0), alpha=0.5, lines=True, planes=True, **kargs*)

Draws a set of 3 coordinate planes.

drawGL(***kargs*)
    Draw the grid.

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 actors.**PlaneActor**(*nx=(2, 2, 2), ox=(0.0, 0.0, 0.0), size=((0.0, 1.0, 1.0), (0.0, 1.0, 1.0)), linecolor=(0.0, 0.0, 0.0), linewidth=None, planecolor=(1.0, 1.0, 1.0), alpha=0.5, lines=True, planes=True, **kargs*)

A plane in a 3D scene.

drawGL(***kargs*)
    Draw the grid.

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 actors.**Text3DActor**(*text*, *font*, *facesize*, *color*, *trl*)
A text as a 3D object.

This class provides an Actor representing a text as an object in 3D space.

setLineWidth(*linewidth*)
Set the linewidth of the Drawable.

setLineStipple(*linestipple*)
Set the linewidth of the Drawable.

setTexture(*texture*)
Set the texture data of the Drawable.

class actors.**GeomActor**(*data*, *elems=None*, *eltype=None*, *mode=None*, *color=None*, *colormap=None*, *bkcolor=None*, *bkcolormap=None*, *alpha=1.0*, *bkalpha=None*, *linewidth=None*, *linestipple=None*, *marksize=None*, *texture=None*, *avgnormals=None*, ***kargs*)
An OpenGL actor representing a geometrical model.

The model can either be in Formex or Mesh format.

nplex()
Return the plexitude of the elements in the actor.

nelems()
Return the number of elements in the actor.

shape()
Return the number and plexitude of the elements in the actor.

points()
Return the vertices as a 2-dim array.

setColor(*color*, *colormap=None*)
Set the color of the Actor.

setBkColor(*color*, *colormap=None*)
Set the backside color of the Actor.

setAlpha(*alpha*, *bkalpha*)
Set the Actors alpha value.

drawGL(*canvas=None*, *mode=None*, *color=None*, ***kargs*)
Draw the geometry on the specified canvas.

The drawing parameters not provided by the Actor itself, are derived from the canvas defaults.

mode and color can be overridden for the sole purpose of allowing the recursive use for modes ending on ‘wire’ (‘smoothwire’ or ‘flatwire’). In these cases, two drawing operations are done: one with mode=’wireframe’ and color=black, and one with mode=mode[:-4].

pickGL(*mode*)
Allow picking of parts of the actor.

mode can be ‘element’, ‘face’, ‘edge’ or ‘point’

setLineWidth(*linewidth*)
Set the linewidth of the Drawable.

setLineStipple (*linestipple*)
Set the linewidth of the Drawable.

setTexture (*texture*)
Set the texture data of the Drawable.

select (*sel*)
Return a GeomActor with a selection of this actor's elements

Currently, the resulting Actor will not inherit the properties of its parent, but the eltype will be retained.

`Functions defined in module actors`

6.3.5 decors — 2D decorations for the OpenGL canvas.

2D decorations are objects that are drawn in 2D on the flat canvas instead of through the 3D OpenGL engine.

`Classes defined in module decors`

class decors.**Decoration** (*x, y, **kargs*)
A decoration is a 2-D drawing at canvas position x,y.

All decorations have at least the following attributes:

- x,y : (int) window coordinates of the insertion point
- **drawGL()** [function that draws the decoration at (x,y).] This should only use openGL function that are allowed in a display list.

drawGL (***kargs*)
Perform the OpenGL drawing functions to display the actor.

pickGL (***kargs*)
Mimick the OpenGL drawing functions to pick (from) the actor.

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 decors.**Mark** (*x, y, mark='dot', color=None, linewidth=None, **kargs*)
A mark at a fixed position on the canvas.

pickGL (***kargs*)
Mimick the OpenGL drawing functions to pick (from) the actor.

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 decors.**Line** (*x1*, *y1*, *x2*, *y2*, *color=None*, *linewidth=None*, ***kargs*)
A straight line on the canvas.

pickGL (***kargs*)
Mimick the OpenGL drawing functions to pick (from) the actor.

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 decors.**GlutText** (*text*, *x*, *y*, *font='9x15'*, *size=None*, *gravity=None*, *color=None*, *zoom=None*, ***kargs*)
A viewport decoration showing a text string.

- text: a simple string, a multiline string or a list of strings. If it is a string, it will be splitted on the occurrence of 'n' characters.
- x,y: insertion position on the canvas
- 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 positon, and 'W' or 'E' for the horizontal. The default(empty) string will center the text.

drawGL (***kargs*)
Draw the text.

pickGL (***kargs*)
Mimick the OpenGL drawing functions to pick (from) the actor.

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.

decors.**Text**
alias of GlutText

class `decors.`**`ColorLegend`**(*colorlegend*, *x*, *y*, *w*, *h*, *ngrid=0*, *linewidth=None*, *nlabel=-1*, *font=None*, *size=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.ColorLegend` class.

Parameters:

- *colorlegend*: a `colorscale.ColorLegend` instance providing conversion between numerical values and colors
- *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 (as set in the `colorlegend`) 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.
- *font*, *size*: font and size to be used for the labels
- *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:

- The number of colors is defined by the `colorlegend` argument.
- Large numbers of colors result inb 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 practices are to use either a low number of colors (<=20) and the default `ngrid` and `nlabel`, or 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.

pickGL(***kargs*)
Mimick the OpenGL drawing functions to pick (from) the actor.

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 decors.**Rectangle**(*x1*, *y1*, *x2*, *y2*, *color=None*, *texture=None*, ***kargs*)
A 2D-rectangle on the canvas.

pickGL(***kargs*)
Mimick the OpenGL drawing functions to pick (from) the actor.

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 decors.**Grid**(*x1*, *y1*, *x2*, *y2*, *nx=1*, *ny=1*, *color=None*, *linewidth=None*, ***kargs*)
A 2D-grid on the canvas.

pickGL(***kargs*)
Mimick the OpenGL drawing functions to pick (from) the actor.

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 decors.**LineDrawing**(*data*, *color=None*, *linewidth=None*, ***kargs*)
A collection of straight lines on the canvas.

pickGL(***kargs*)
Mimick the OpenGL drawing functions to pick (from) the actor.

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 decors.**Triade** (*pos='lb', siz=100, pat='3:012934', legend='xyz', color=[(1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 1.0), (0.0, 1.0, 1.0), (1.0, 0.0, 1.0), (1.0, 1.0, 0.0)], **kargs*)
An OpenGL actor representing a triade of global axes.

- *pos*: position on the canvas: two characters, of which first sets horizontal position ('l', 'c' or 'r') and second sets vertical position ('b', 'c' or 't').
- *size*: size in pixels of the zone displaying the triade.
- *pat*: shape to be drawn in the coordinate planes. Default is a square. '16' givec a triangle. '' disables the planes.
- *legend*: text symbols to plot at the end of the axes. A 3-character string or a tuple of 3 strings.

drawGL (***kargs*)
Perform the OpenGL drawing functions to display the actor.

pickGL (***kargs*)
Mimick the OpenGL drawing functions to pick (from) the actor.

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.

Functions defined in module decors

decors.**drawDot** (*x*, *y*)
Draw a dot at canvas coordinates (x,y).

decors.**drawLine** (*x1*, *y1*, *x2*, *y2*)
Draw a straight line from (x1,y1) to (x2,y2) in canvas coordinates.

decors.**drawGrid** (*x1*, *y1*, *x2*, *y2*, *nx*, *ny*)
Draw a rectangular grid of lines

The rectangle has (x1,y1) and 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.

decors.**drawRect** (*x1*, *y1*, *x2*, *y2*)
Draw the circumference of a rectangle.

decors.**drawRectangle**(*x1*, *y1*, *x2*, *y2*, *color*, *texture=None*)
Draw a single rectangular quad.

6.3.6 marks — OpenGL marks for annotating 3D actors.

Classes defined in module marks

class marks.**Mark**(*pos*, *nolight=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
- *draw()* : method to draw the mark

drawGL(***kargs*)
Perform the OpenGL drawing functions to display the actor.

pickGL(***kargs*)
Mimick the OpenGL drawing functions to pick (from) the actor.

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 marks.**AxesMark**(*pos*, *color=None*, ***kargs*)
Two viewport axes drawn at a 3D position.

pickGL(***kargs*)
Mimick the OpenGL drawing functions to pick (from) the actor.

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 marks.**TextMark**(*pos*, *text*, *color=None*, *font='sans'*, *size=18*, ***kargs*)
A text drawn at a 3D position.

pickGL(***kargs*)
Mimick the OpenGL drawing functions to pick (from) the actor.

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 marks.**MarkList** (*pos, val, color=(0.0, 0.0, 0.0), font='sans', size=18, leader='', gravity='', **kargs*)
A list of numbers drawn at 3D positions.

drawpick ()
This functions mimicks the drawing of a number list for picking.

pickGL (***kargs*)
Mimick the OpenGL drawing functions to pick (from) the actor.

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.

Functions defined in module marks

6.3.7 gluttext — 2D text decorations using GLUT fonts

This module provides the basic functions for using the GLUT library in the rendering of text on an OpenGL canvas.

Classes defined in module gluttext

Functions defined in module gluttext

gluttext.**glutSelectFont** (*font=None, size=None*)
Select one of the glut fonts using a font + size description.

- font: 'fixed', 'serif' or 'sans'
- size: an int that will be rounded to the nearest available size.

The return value is a 4-character string representing one of the GLUT fonts.

gluttext.**glutFont** (*font*)
Return GLUT font designation for the named font.

The recognized font names are:

- fixed: '9x15', '8x13',
- times-roman: 'tr10', 'tr24'
- helvetica: 'hv10', 'hv12', 'hv18'

If an unrecognized string is given, the default is 'hv18'.

`gluttext.`**`glutFontHeight`**(*font*)
Return the height of the named glut font.

This supposes that the last two characters of the name hold the font height.

`gluttext.`**`glutRenderText`**(*text*, *font*, *gravity=''*)
Draw a text in given font at the current rasterpoint.

font should be one of the legal fonts returned by glutFont(). If text is not a string, it will be formatted to a string before drawing. After drawing, the rasterpos will have been updated!

`gluttext.`**`glutBitmapLength`**(*font*, *text*)
Compute the length in pixels of a text string in given font.

We use our own function to calculate the length because the builtin has a bug.

`gluttext.`**`glutDrawText`**(*text*, *x*, *y*, *font='hv18'*, *gravity=''*, *spacing=1.0*)
Draw a text at given 2D position in window.

- text: a simple string, a multiline string or a list of strings. If it is a string, it will be splitted on the occurrence of 'n' characters.
- x,y: insertion position on the canvas
- 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 positon, and 'W' or 'E' for the horizontal. The default(empty) string will center the text.

6.3.8 `canvas` — This implements an OpenGL drawing widget for painting 3D scenes.

`Classes defined in module canvas`

class `canvas.`**`ActorList`**(*canvas*)
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.

`add`(*actor*)
Add an actor or a list thereof to a ActorList.

`delete`(*actor*)
Remove an actor or a list thereof from an ActorList.

`redraw`()
Redraw all actors in the list.

This redraws the specified actors (recreating their display list). This could e.g. be used after changing an actor's properties.

class `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
- bklormap: 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 (*clas*, *dict*, *strict=True*)
Transform a dict to acceptable settings.

setMode ()
Activate the mode canvas settings in the GL machine.

activate()
Activate the default canvas settings in the GL machine.

get(*key*, *default*)
Return the value for key or a default.

This is the equivalent of the dict get method, except that it returns only the default value if the key was not found in self, and there is no _default_ method or it raised a KeyError.

setdefault(*key*, *default*)
Replaces the setdefault function of a normal dictionary.

This is the same as the get method, except that it also sets the default value if get found a KeyError.

class canvas.**Canvas**(*settings={}*)
A canvas for OpenGL rendering.

The Canvas is a class holding all global data of an OpenGL scene rendering. This includes colors, line types, rendering mode. It also keeps lists of the actors and decorations in the scene. The canvas has a Camera object holding important viewing parameters. Finally, it stores the lighting information.

It does not however contain the viewport size and position.

enable_lighting(*state*)
Toggle lights on/off.

has_lighting()
Return the status of the lighting.

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

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 fdifferent 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. Value is an integer. If positive, edges are shown, else not.

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_lighting (*state*, *oldstate=None*)
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 (*on=None*, *pos='lb'*, *siz=100*)
Toggle the display of the global axes on or off.

If on is True, a triade of global axes is displayed, if False it is removed. The default (None) toggles between on and off.

clear ()
Clear the canvas to the background color.

setDefaults ()
Activate the canvas settings in the GL machine.

overrideMode (*mode*)
Override some settings

glinit ()
Initialize the rendering machine.

The rendering machine is initialized according to self.settings: - self.rendermode: one of - self.lighting

glupdate ()
Flush all OpenGL commands, making sure the display is updated.

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.

sceneBbox()
Return the bbox of all actors in the scene

setBbox(*bb=None*)
Set the bounding box of the scene you want to be visible.

bb is a (2,3) shaped array specifying a bounding box. If no bbox is given, the bounding box of all the actors in the scene is used, or if the scene is empty, a default unit bounding box.

addActor(*itemlist*)
Add a 3D actor or a list thereof to the 3D scene.

addHighlight(*itemlist*)
Add a highlight or a list thereof to the 3D scene.

addAnnotation(*itemlist*)
Add an annotation or a list thereof to the 3D scene.

addDecoration(*itemlist*)
Add a 2D decoration or a list thereof to the canvas.

addAny(*itemlist=None*)
Add any item or list.

This will add any actor/annotation/decoration item or a list of any such items to the canvas. This is the prefered method to add an item to the canvas, because it makes sure that each item is added to the proper list. It can however not be used to add highlights.

If you have a long list of a single type, it is more efficient to use one of the type specific add methods.

removeActor(*itemlist=None*)
Remove a 3D actor or a list thereof from the 3D scene.

Without argument, removes all actors from the scene. This also resets the bounding box for the canvas autozoom.

removeHighlight(*itemlist=None*)
Remove a highlight or a list thereof from the 3D scene.

Without argument, removes all highlights from the scene.

removeAnnotation(*itemlist=None*)
Remove an annotation or a list thereof from the 3D scene.

Without argument, removes all annotations from the scene.

removeDecoration (*itemlist=None*)
Remove a 2D decoration or a list thereof from the canvas.

Without argument, removes all decorations from the scene.

removeAny (*itemlist=None*)
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.

redrawAll ()
Redraw all actors in the scene.

setCamera (*bbox=None*, *angles=None*)
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 noet 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

draw_focus_rectangle(*ofs=0*, *color=(1.0, 0.2, 0.4)*)
Draw the focus rectangle.

The specified width is HALF of the line width

draw_cursor(*x*, *y*)
draw the cursor

pick_actors()
Set the list of actors inside the pick_window.

pick_parts(*obj_type*, *max_objects*, *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. max_objects specifies the maximum number of objects

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.

highlightActor(*actor*)
Highlight an actor in the scene.

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.

highlightElements (*K*)
Highlight a selection of actor elements on the canvas.

K is Collection of actor elements as returned by the pick() method. colormap is a list of two colors, for the elements not in, resp. in the Collection K.

highlightEdges (*K*)
Highlight a selection of actor edges on the canvas.

K is Collection of TriSurface actor edges as returned by the pick() method. colormap is a list of two colors, for the edges not in, resp. in the Collection K.

highlightPoints (*K*)
Highlight a selection of actor elements on the canvas.

K is Collection of actor elements as returned by the pick() method.

highlightPartitions (*K*)
Highlight a selection of partitions on the canvas.

K is a Collection of actor elements, where each actor element is connected to a collection of property numbers, as returned by the partitionCollection() method.

highlight (*K*, *mode*)
Highlight a Collection of actor/elements.

K is usually the return value of a pick operation, but might also be set by the user. mode is one of the pick modes.

Functions defined in module canvas

canvas.**gl_pickbuffer** ()
Return a list of the 2nd numbers in the openGL pick buffer.

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.

canvas.**glSmooth** (*smooth=True*)
Enable smooth shading

canvas.**glFlat** ()
Disable smooth shading

canvas.**onOff** (*onoff*)
Convert On/Off strings to a boolean

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.

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.

6.3.9 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 viewport

class 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 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 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 esier 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

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.

resetOptions()
Reset the Drawing options to some defaults

setOptions(*d*)
Set the Drawing options to some values

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_zoom(*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.

setPickable(*nrs=None*)
Set the list of pickable actors

start_selection(*mode*, *filter*)
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*)
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 partitionByConnection method.

When the picking operation is finished, the selection is returned. The return value is always a Collection object.

pickNumbers (**args*, ***kargs*)
Go into number picking mode and return the selection.

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 funcion 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.

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.

enable_lighting (*state*)
Toggle lights on/off.

has_lighting ()
Return the status of the lighting.

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

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 fdifferent 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. Value is an integer. If positive, edges are shown, else not.

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_lighting(*state*, *oldstate=None*)
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(*on=None*, *pos='lb'*, *siz=100*)
Toggle the display of the global axes on or off.

If on is True, a triade of global axes is displayed, if False it is removed. The default (None) toggles between on and off.

clear()
Clear the canvas to the background color.

setDefaults()
Activate the canvas settings in the GL machine.

overrideMode(*mode*)
Override some settings

glinit()
Initialize the rendering machine.

The rendering machine is initialized according to self.settings: - self.rendermode: one of - self.lighting

glupdate()
Flush all OpenGL commands, making sure the display is updated.

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.

sceneBbox()
Return the bbox of all actors in the scene

setBbox(*bb=None*)
Set the bounding box of the scene you want to be visible.

bb is a (2,3) shaped array specifying a bounding box. If no bbox is given, the bounding box of all the actors in the scene is used, or if the scene is empty, a default unit bounding box.

addActor(*itemlist*)
Add a 3D actor or a list thereof to the 3D scene.

addHighlight(*itemlist*)
Add a highlight or a list thereof to the 3D scene.

addAnnotation(*itemlist*)
Add an annotation or a list thereof to the 3D scene.

addDecoration(*itemlist*)
Add a 2D decoration or a list thereof to the canvas.

addAny(*itemlist=None*)
Add any item or list.

This will add any actor/annotation/decoration item or a list of any such items to the canvas. This is the prefered method to add an item to the canvas, because it makes sure that each item is added to the proper list. It can however not be used to add highlights.

If you have a long list of a single type, it is more efficient to use one of the type specific add methods.

removeActor(*itemlist=None*)
Remove a 3D actor or a list thereof from the 3D scene.

Without argument, removes all actors from the scene. This also resets the bounding box for the canvas autozoom.

removeHighlight(*itemlist=None*)
Remove a highlight or a list thereof from the 3D scene.

Without argument, removes all highlights from the scene.

removeAnnotation (*itemlist=None*)
Remove an annotation or a list thereof from the 3D scene.

Without argument, removes all annotations from the scene.

removeDecoration (*itemlist=None*)
Remove a 2D decoration or a list thereof from the canvas.

Without argument, removes all decorations from the scene.

removeAny (*itemlist=None*)
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.

redrawAll ()
Redraw all actors in the scene.

setCamera (*bbox=None*, *angles=None*)
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 noet 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

draw_focus_rectangle(*ofs=0*, *color=(1.0*, *0.2*, *0.4)*)
Draw the focus rectangle.

The specified width is HALF of the line width

draw_cursor(*x*, *y*)
draw the cursor

pick_actors()
Set the list of actors inside the pick_window.

pick_parts(*obj_type*, *max_objects*, *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. max_objects specifies the maximum number of objects

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.

highlightActor (*actor*)
Highlight an actor in the scene.

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.

highlightElements (*K*)
Highlight a selection of actor elements on the canvas.

K is Collection of actor elements as returned by the pick() method. colormap is a list of two colors, for the elements not in, resp. in the Collection K.

highlightEdges (*K*)
Highlight a selection of actor edges on the canvas.

K is Collection of TriSurface actor edges as returned by the pick() method. colormap is a list of two colors, for the edges not in, resp. in the Collection K.

highlightPoints (*K*)
Highlight a selection of actor elements on the canvas.

K is Collection of actor elements as returned by the pick() method.

highlightPartitions (*K*)
Highlight a selection of partitions on the canvas.

K is a Collection of actor elements, where each actor element is connected to a collection of property numbers, as returned by the partitionCollection() method.

highlight (*K*, *mode*)
Highlight a Collection of actor/elements.

K is usually the return value of a pick operation, but might also be set by the user. mode is one of the pick modes.

class viewport.**NewiMultiCanvas** (*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 pos is given, it specifies all viewports and nvps, nrows and ncols are disregarded.

Else:

If nvps is given, it specifies the number of viewports in the layout. Else, nvps will be set to the current number of viewports.

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.

If nvps is not equal to the current number of viewports, viewports will be added or removed to match the requested number.

By default they are laid out rowwise over two columns.

createView(*shared=None*)
Create a new viewport

If another QtCanvas instance is passed, both will share the same display lists and textures.

addView(*view*, *row*, *col*, *rowspan=1*, *colspan=1*)
Add a new viewport and make it visible

removeView(*view=None*)
Remove a view from the canvas

If view is None, the last one is removed. You can not remove a view when there is only one left.

setCurrent(*view*)
Make the specified viewport the current one.

view can be either a viewport or viewport number. The current viewport is the one that will be used for drawing operations. This may be different from the viewport having GUI focus (pf.canvas).

setStretch(*rowstretch*, *colstretch*)
Set the row and column stretch factors.

rowstretch and colstretch are lists of stretch factors to be applied on the subsequent rows/columns. If the lists are shorter than the number of rows/columns, the

link(*vp*, *to*)
Link viewport vp to to

class viewport.**FramedGridLayout**(*parent=None*)
A QtGui.QGridLayout where each added widget is framed.

class 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.

newView(*shared=None*, *settings=None*)
Create a new viewport

If another QtCanvas instance is passed, both will share the same display lists and textures.

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

showWidget (*w*)
Show the view w.

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.

link (*vp*, *to*)
Link viewport vp to to

`Functions defined in module viewport`

viewport.**dotpr** (*v*, *w*)
Return the dot product of vectors v and w

viewport.**length** (*v*)
Return the length of the vector v

viewport.**projection** (*v*, *w*)
Return the (signed) length of the projection of vector v on vector w.

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
```

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.

viewport.**OpenGLFormat** (*fmt=None*)
Some information about the OpenGL format.

6.3.10 camera — OpenGL camera handling

`Classes defined in module camera`

class `camera.Camera` (*center=[0.0, 0.0, 0.0], long=0.0, lat=0.0, twist=0.0, dist=1.0*)

A camera for OpenGL rendering.

The Camera class holds all the camera related settings related to the rendering of a scene in OpenGL. These include camera position, the viewing direction of the camera, and the lens parameters (opening angle, front and back clipping planes). This class also provides convenient methods to change the settings so as to get smooth camera manipulation.

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:

- *focus*: *[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 explicitely 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.

getRot()
Return the camera rotation matrix.

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.

setAngles(*angles*)
Set the rotation angles.

angles is either:

- a tuple of angles (long,lat,twist)
- a named view corresponding to one of the predefined viewing directions in views.py
- None

setRotation(*long*, *lat*, *twist=0*)
Set the rotation matrix of the camera from three 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.

move(*dx*, *dy*, *dz*)
Move the camera over translation (dx,dy,dz) in global coordinates.

The center of the camera is moved over the specified translation vector. This has the effect of moving the scene in opposite direction.

rotate(*val*, *vx*, *vy*, *vz*)
Rotate the camera around current axis (vx,vy,vz).

saveModelView()
Save the ModelView matrix.

setModelView()
Set the ModelView matrix from camera parameters.

loadModelView(*m=None*)
Load the ModelView matrix.

There are three uses of this function:

- Without argument and if the viewing parameters have not changed since the last save of the ModelView matrix, this will just reload the ModelView matrix from the saved value.

- If an argument is supplied, it should be a legal ModelView matrix and that matrix will be loaded (and saved) as the new ModelView matrix.
- Else, a new ModelView matrix is set up from the camera parameters, and it is loaded and saved.

In the latter two cases, the new ModelView matrix is saved, and if a camera attribute *modelview_callback* has been set, a call to this function is done, passing the camera instance as parameter.

loadCurrentRotation()
Load the current ModelView matrix with translations canceled out.

transform(*v*)
Transform a vertex using the currently saved Modelview matrix.

toWorld(*v*)
Transform a vertex from camera to world coordinates.

This multiplies The specified vector can have 3 or 4 (homogoneous) components. This uses the currently saved rotation matrix.

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*, *center=False*, *clip=True*)
Set the viewable area of the camera.

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 vamera area.

dx and dy are relative movements in fractions of the current area size.

setClip(*near*, *far*)
Set the near and far clipping planes

setPerspective(*on=True*)
Set perspective on or off

loadProjection(*force=False*, *pick=None*, *keepmode=False*)
Load the projection/perspective matrix.

The caller will have to setup the correct GL environment beforehand. No need to set matrix mode though. This function will switch to GL_PROJECTION mode before loading the matrix

If keepmode=True, does not switch back to GL_MODELVIEW mode.

A pick region can be defined to use the camera in picking mode. pick defines the picking region center and size (x,y,w,h).

This function does it best at autodetecting changes in the lens settings, and will only reload the matrix if such changes are detected. You can optionally force loading the matrix.

project (*x*, *y*, *z*)
Map the object coordinates (x,y,z) to window coordinates.

unProject (*x*, *y*, *z*)
Map the window coordinates (x,y,z) to object coordinates.

setTracking (*onoff=True*)
Enable/disable coordinate tracking using the camera

`Functions defined in module camera`

camera.**tand** (*arg*)
Return the tan of an angle in degrees.

6.3.11 `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.

`Classes defined in module image`

`Functions defined in module image`

image.**initialize** ()
Initialize the image module.

image.**imageFormats** ()
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.

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.

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.

image.**save_canvas** (*canvas*, *fn*, *fmt='png'*, *quality=-1*, *size=None*)
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

image.**save_window** (*filename*, *format*, *quality=-1*, *windowname=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.

image.**save_main_window** (*filename*, *format*, *quality=-1*, *border=False*)
Save the main pyFormex window as an image file.

This function needs a filename AND format. This is an alternative for save_window, by grabbin it from the root window, using save_rect. This allows us to grab the border as well.

image.**save_rect** (*x*, *y*, *w*, *h*, *filename*, *format*, *quality=-1*)
Save a rectangular part of the screen to a an image file.

image.**save** (*filename=None*, *window=False*, *multi=False*, *hotkey=True*, *autosave=False*, *border=False*, *rootcrop=False*, *format=None*, *quality=-1*, *size=None*, *verbose=False*)
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 implicitely be ended before starting the second.

In multisave mode, each call to saveNext() will save an image to the next generated file name. Filenames 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 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.

image.**saveImage** (*filename=None*, *window=False*, *multi=False*, *hotkey=True*, *autosave=False*, *border=False*, *rootcrop=False*, *format=None*, *quality=-1*, *size=None*, *verbose=False*)
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 implicitely be ended before starting the second.

In multisave mode, each call to saveNext() will save an image to the next generated file name. Filenames 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 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.

`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.

`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.

`image.`**`saveIcon`**(*fn*, *size=32*, *transparent=True*)

Save the current rendering as an icon.

`image.`**`autoSaveOn`**`()`

Returns True if autosave multisave mode is currently on.

Use this function instead of directly accessing the autosave variable.

`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 'ffmeg', 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'.

image.**saveMovie**(*filename*, *format*, *windowname=None*)
Create a movie from the pyFormex window.

6.3.12 imageViewer — A general image viewer

Part of this code was borrowed from the TrollTech Qt examples.

Classes defined in module imageViewer

class imageViewer.**ImageViewer**(*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.

Functions defined in module imageViewer

6.3.13 imagearray — Convert bitmap images into numpy arrays.

This module contains functions to convert bitmap images into numpy arrays and vice versa.

This code was based on ideas found on the PyQwt mailing list.

Classes defined in module imagearray

Functions defined in module imagearray

imagearray.**resizeImage**(*image*, *w=0*, *h=0*)
Load and optionally resize an image.

Parameters:

- *image*: a QImage, or any data that can be converted to a QImage, e.g. the name of a raster image file.
- *w*, *h*: requested size in pixels of the image. A value <= 0 will be replaced with the corresponding actual size of the image.

Returns a QImage with the requested size.

imagearray.**image2numpy**(*image*, *resize=(0, 0)*, *order='RGBA'*, *flip=True*, *indexed=None*, *expand=None*)
Transform an image to a Numpy array.

Parameters:

- *image*: 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*: a tuple of two integers (width,height). Positive value will force the image to be resized to this value.
- *order*: string with a permutation 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*: boolean: 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*: True, False 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 converted.
  - 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.
- *expand*: deprecated, retained for compatibility

Returns:

- if *indexed* is False: an int8 array with shape (height,width,4), holding the 4 components of the color of each pixel. Order of the components is as specified by the *order* argument. Indexed image formats will be expanded to a full color array.
- if *indexed* is True: a tuple (colors,colortable) where colors is an (height,width) shaped int array of indices into the colortable, which is an int8 array with shape (ncolors,4).
- if *indexed* is None (default), a tuple (colors,colortable) is returned, the type of which depend on the original image format:
  - for indexed formats, colors is an int (height,width) array of indices into the colortable, which is an int8 array with shape (ncolors,4).
  - for non-indexed formats, colors is a full (height,width,4) array and colortable is None.

imagearray.**gray2qimage**(*gray*)

Convert the 2D numpy array *gray* into a 8-bit QImage with a gray colormap. The first dimension represents the vertical image axis.

imagearray.**rgb2qimage**(*rgb*)

Convert the 3D numpy array into a 32-bit QImage.

Parameters:

- *rgb* : (height,width,nchannels) integer array specifying the pixels of an image. There can be 3 (RGB) or 4 (RGBA) channels.

Returns a QImage with size (height,width) in the format RGB32 (3channel) or ARGB32 (4channel).

`imagearray.image2glcolor(image, resize=(0, 0))`
Convert a bitmap image to corresponding OpenGL colors.

Parameters:

- *image*: 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*: a tuple of two integers (width,height). Positive value will force the image to be resized to this value.

Returns a (w,h,3) shaped 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.

`imagearray.loadImage_dicom(filename)`
Load a DICOM image into a numpy array.

This function uses the python-dicom module to load a DICOM image into a numpy array. See also `loadImage_gdcm()` for an equivalent using python-gdcm.

Parameters:

- *file*: the name of a DICOM image file

Returns a 3D array with the pixel data of all the images. The first axis is the *z* value, the last the *x*.

As a side effect, this function sets the global variable *_dicom_spacing* to a (3,) array with the pixel/slice spacing factors, in order (x,y,z).

`imagearray.loadImage_gdcm(filename)`
Load a DICOM image into a numpy array.

This function uses the python-gdcm module to load a DICOM image into a numpy array. See also `loadImage_dicom()` for an equivalent using python-dicom.

Parameters:

- *file*: the name of a DICOM image file

Returns a 3D array with the pixel data of all the images. The first axis is the *z* value, the last the *x*.

As a side effect, this function sets the global variable *_dicom_spacing* to a (3,) array with the pixel/slice spacing factors, in order (x,y,z).

`imagearray.readDicom(filename)`
Load a DICOM image into a numpy array.

This function uses the python-gdcm module to load a DICOM image into a numpy array. See also `loadImage_dicom()` for an equivalent using python-dicom.

Parameters:

- *file*: the name of a DICOM image file

Returns a 3D array with the pixel data of all the images. The first axis is the *z* value, the last the *x*.

As a side effect, this function sets the global variable *_dicom_spacing* to a (3,) array with the pixel/slice spacing factors, in order (x,y,z).

imagearray.**dicom2numpy**(*files*)

Read a set of DICOM image files.

Parameters:

- *files*: a list of file names of dicom images of the same size, or a directory containing such images. In the latter case, all the DICOM images in the directory will be read.

Returns a tuple of:

- *pixar*: a 3D array with the pixel data of all the images. The first axis is the *z* value, the last the *x*.
- *scale*: a (3,) array with the scaling factors, in order (x,y,z).

6.3.14 appMenu — Menu with pyFormex apps.

Classes defined in module appMenu

class 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/appsdir")
```

creates a menu with all pyFormex apps in the specified path and its subdirectories.

```
ApptMenu('My Scripts',dir="/path/to/my/sciptsdir",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=
```

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 (*first=0, last=None, random=False, count=-1, recursive=True*)
Run all apps in the range [first,last].

Runs the apps in the range first:last. If last is None, the length of the file list is used. If count is positive, at most count scripts per submenu are executed. Notice that the range is Python style. If random is True, the files are shuffled before running. If recursive is True, also the files in submenu are played. The first and last arguments do not apply to the submenus.

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).

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.

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 icondir.
    - *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.

Functions defined in module appMenu

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.

`appMenu.`**`classify`**(*appdir*, *pkg*, *nmax=0*)
Classify the files in submenus according to keywords.

`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

`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.

`appMenu.`**`reloadMenu`**(*mode='app'*)
Reload the named menu.

6.3.15 `toolbar` — Toolbars for the pyFormex GUI.

This module defines the functions for creating the pyFormex window toolbars.

`Classes defined in module toolbar`

`Functions defined in module toolbar`

`toolbar.`**`addActionButtons`**(*toolbar*)
Add the script action buttons to the toolbar.

`toolbar.`**`addButton`**(*toolbar*, *tooltip*, *icon*, *func*, *repeat=False*, *toggle=False*, *checked=False*, *icon0=None*)
Add a button to a toolbar.

- *toolbar*: the toolbar where the button will be added
- *tooltip*: the text to appears 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 buton.
- *icon1*: for a toggle button, icon to display when button is not checked.

toolbar.**removeButton**(*toolbar*, *button*)
Remove a button from a toolbar.

toolbar.**addCameraButtons**(*toolbar*)
Add the camera buttons to a toolbar.

toolbar.**toggleButton**(*attr*, *state=None*)
Update the corresponding viewport attribute.

This does not update the button state.

toolbar.**updateButton**(*button*, *attr*)
Update the button to correct state.

toolbar.**updateWireButton**()
Update the wire button to correct state.

toolbar.**updateTransparencyButton**()
Update the transparency button to correct state.

toolbar.**updateLightButton**()
Update the light button to correct state.

toolbar.**updateNormalsButton**(*state=True*)
Update the normals button to correct state.

toolbar.**updatePerspectiveButton**()
Update the normals button to correct state.

toolbar.**addTimeoutButton**(*toolbar*)
Add or remove the timeout button,depending on cfg.

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 calpy_itf — Calpy interface for pyFormex.

Currently this is only used to detect the installation of calpy and add the path of the calpy module to sys.path.

Importing this module will automatically check the availabilty of calpy and set the sys.path accordingly.

`Classes defined in module calpy_itf`

class `calpy_itf.`**`QuadInterpolator`**(*nelems*, *nplex*, *gprule*)
A class to interface with calpy's Quad class.

We want to use the calpy interpolation facilities without having to set up a full model for calpy processing. This class just sets the necessary data to make the interpolation methods (GP2NOdes, NodalAcc, NodalAvg) work.

Parameters:

- *nelems*: number of elements
- *nplex*: plexitude of the elements (supported is 4 to 9)
- *gprule*: gauss integration rule

`AddElements`(*nodes*, *matnr*, *emat*, *coord*, *dofid*)
Add elements to group and store interpolation matrices

`InterpolationMatrix`(*elnod*, *xyz*, *gpx*, *gpw*)
Form interpolation matrix for QUAD/4-9 element

The input gives a node set with either 4, 8 or 9 nodes. The 8 and 9 node versions may contain zeros to suppress higher order nodes. xyz[nnod,2] contains the coordinates of the actually existing nodes! gpx and gpw are the coordinates and weights of the Gauss integration points (gpx has 2 rows) The procedure returns two matrices: hall[3,nnod,ng] holds the values at all gauss points, of the interpolation function and its natural derivatives corresponding with each node. w[ng] holds the 2D weight of the gauss points.

`Assemble`(*s*, *emat*, *model*)
Assemble all elements of this group

`StressGP`(*v*, *emat*)
Compute stress at GP for all elements of this group

`GP2Nodes`(*data*)
Extrapolate a set of data at GP to 9 nodal points

data has shape (ndata,ngp,nelems) result has shape (ndata,nnodel,nelems)

`NodalAcc`(*nodes*, *endata*, *nnod=0*, *nodata=None*, *nodn=None*)
Store element data in nodes and compute averages

If matching nodal arrays nodata and nodn are specified, data are added to it. If not, nnod should be specified. By default, new nodata and nodn arrays are created and zeroed.

`NodalAvg`(**args*)
Return the nodal average from the accumulated data.

`AddBoundaryLoads`(*f*, *model*, *idloads*, *dloads*, *nodes*, *matnr*, *coord*, *dofid*, *emat*)
Assemble distributed boundary loads into the load vector

This procedures adds the distributed loads acting on the boundaries of the elements of this group to the global load vector f. The loads are specified by idloads and dloads, in the format as returned by fe_util.ReadBoundaryLoads() idloads contains : element number, side number, load case dloads contains : load components qx, qy Side number is the number of the first node of this side. We mod it with 4, to have an offset in the sidenode numbers array nodes,matnr is the node/mat numbers of the underlying element group coords contains

the coordinates of the nodes dofid contains the numbering of the nodal dofs emat is the properties matrix of the underlying group

BoundaryInterpolationMatrix (*elnod*, *xyz*, *gpx*, *gpw*, *localnodes*)
Form interpolation matrix for QUAD/4-9 boundary

The input gives a node set with either 2 or 3 nodes. The 3 node version may contain a zero to suppress the higher order node. xyz[nnod,2] contains the coordinates of the actually existing nodes! gpx, gpw are the (2D)-coordinates and weights of the GPs The procedure returns two matrices: hall[3,nnod,ng] holds the values at all gauss points, of the interpolation function and its natural derivatives corresponding with each node. w[ng] holds the 2D weight of the gauss points.

`Functions defined in module calpy_itf`

calpy_itf.**detect** (*trypaths=None*)
Check if we have calpy and if so, add its path to sys.path.

calpy_itf.**check** (*trypaths=None*)
Warn the user that calpy was not found.

6.4.2 cameratools — Camera tools

Some extra tools to handle the camera.

`Classes defined in module cameratools`

`Functions defined in module cameratools`

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 ccxdat —

`Classes defined in module ccxdat`

`Functions defined in module ccxdat`

ccxdat.**readDispl** (*fil*, *nnodes*, *nres*)
Read displacements from a Calculix .dat file

ccxdat.**readStress** (*fil*, *nelems*, *ngp*, *nres*)
Read stresses from a Calculix .dat file

ccxdat.**readResults** (*fn*, *DB*, *nnodes*, *nelems*, *ngp*)
Read Calculix results file for nnodes, nelems, ngp

Add results to the specified DB

ccxdat.**createResultDB** (*model*)
Create a results database for the given FE model

ccxdat.**addFeResult** (*DB*, *step*, *time*, *result*)
Add an FeResult for a time step to the result DB

This is currently 2D only

ccxdat.**computeAveragedNodalStresses**(*M*, *data*, *gprule*)
Compute averaged nodal stresses from GP stresses in 2D quad8 mesh

6.4.4 ccxinp —

Classes defined in module ccxinp

Functions defined in module ccxinp

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.

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.

ccxinp.**startPart**(*name*)
Start a new part.

ccxinp.**readCommand**(*line*)
Read a command line, return the command and a dict with options

ccxinp.**do_HEADING**(*opts*, *data*)
Read the nodal data

ccxinp.**do_PART**(*opts*, *data*)
Set the part name

ccxinp.**do_SYSTEM**(*opts*, *data*)
Read the system data

ccxinp.**do_NODE**(*opts*, *data*)
Read the nodal data

ccxinp.**do_ELEMENT**(*opts*, *data*)
Read element data

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 `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 curve`

class `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
- *parts*: 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 placeded off the curve (control points)
- *parts(j,k)*:
- *approx(ndiv,ntot)*:

Furthermore it may define, for efficiency reasons, the following methods:

- *sub_points_2*
- *sub_directions_2*

`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*, *normalized=False*, *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 normalized is True, the parameter values are give in a normalized space where 0 is the start of the curve and 1 is the end.

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 all 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.

approx (*ndiv=None*, *ntot=None*)
Return a PolyLine approximation of the curve

Parameters:

- *ndiv*: int: number of straight segments to use over each part of the curve. This is only used if
- *ntot*: int: number of straight segments to use over the total length of the curve.

Returns a PolyLine approximation for the curve. *C.approx(ndiv=n)* returns an approximation with *ndiv* segments over each part of the curve. This may results in segments with very different lengths. *C.approx(ntot=n)* returns an approximation with *ntot* segments over the total length of the curve. This produces more equally sized segments, but the internal end points of the curve parts may not be on the approximating Polyline.

frenet (*ndiv=None*, *ntot=None*, *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* and *ntot*. 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 binormial 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

approximate (*nseg*, *equidistant=True*, *npre=100*)

Approximate a Curve with a PolyLine of n segments

Parameters:

- *nseg*: number of straight segments of the resulting PolyLine
- *equidistant*: if True (default) the points are spaced almost equidistantly over the curve. If False, the points are spread equally over the parameter space.
- *npre*: 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.

Note: This is an alternative for Curve.approx, and may replace it in future.

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.

setProp (*p=None*)

Create or destroy the property array for the Formex.

A property array is a rank-1 integer array with dimension equal to the number of elements in the Formex (first dimension of data). You can specify a single value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

If a value None is given, the properties are removed from the Formex.

addNoise (**args*, ***kargs*)

Apply 'addNoise' transformation to the Geometry object.

See `coords.Coords.addNoise()` for details.

affine (**args*, ***kargs*)

Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

align (**args*, ***kargs*)

Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

bump (**args*, ***kargs*)

Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

bump1 (**args*, ***kargs*)
Apply 'bump1' transformation to the Geometry object.

See `coords.Coords.bump1()` for details.

bump2 (**args*, ***kargs*)
Apply 'bump2' transformation to the Geometry object.

See `coords.Coords.bump2()` for details.

centered (**args*, ***kargs*)
Apply 'centered' transformation to the Geometry object.

See `coords.Coords.centered()` for details.

cylindrical (**args*, ***kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See `coords.Coords.cylindrical()` for details.

egg (**args*, ***kargs*)
Apply 'egg' transformation to the Geometry object.

See `coords.Coords.egg()` for details.

flare (**args*, ***kargs*)
Apply 'flare' transformation to the Geometry object.

See `coords.Coords.flare()` for details.

hyperCylindrical (**args*, ***kargs*)
Apply 'hyperCylindrical' transformation to the Geometry object.

See `coords.Coords.hyperCylindrical()` for details.

isopar (**args*, ***kargs*)
Apply 'isopar' transformation to the Geometry object.

See `coords.Coords.isopar()` for details.

map (**args*, ***kargs*)
Apply 'map' transformation to the Geometry object.

See `coords.Coords.map()` for details.

map1 (**args*, ***kargs*)
Apply 'map1' transformation to the Geometry object.

See `coords.Coords.map1()` for details.

mapd (**args*, ***kargs*)
Apply 'mapd' transformation to the Geometry object.

See `coords.Coords.mapd()` for details.

position (**args*, ***kargs*)
Apply 'position' transformation to the Geometry object.

See `coords.Coords.position()` for details.

projectOnCylinder (**args*, ***kargs*)
Apply 'projectOnCylinder' transformation to the Geometry object.

See `coords.Coords.projectOnCylinder()` for details.

projectOnPlane (**args*, ***kargs*)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (**args*, ***kargs*)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

reflect (**args*, ***kargs*)
Apply 'reflect' transformation to the Geometry object.

See `coords.Coords.reflect()` for details.

replace (**args*, ***kargs*)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

rollAxes (**args*, ***kargs*)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

rot (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

rotate (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

scale (**args*, ***kargs*)
Apply 'scale' transformation to the Geometry object.

See `coords.Coords.scale()` for details.

shear (**args*, ***kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

spherical (**args*, ***kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args*, ***kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

swapAxes (**args*, ***kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

toCylindrical (**args*, ***kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

toSpherical (**args*, ***kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

transformCS (**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

translate (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

trl (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

nelems ()
Return the number of elements in the Geometry.

This method should be re-implemented by the derived classes. For the (empty) Geometry class it always returns 0.

toProp (*prop*)
Converts the argument into a legal set of properties for the object.

The conversion involves resizing the argument to a 1D array of length self.nelems(), and converting the data type to integer.

getCoords ()
Get the coords data.

Returns the full array of coordinates stored in the Geometry object. Note that subclasses may store more points in this array than are used to define the geometry.

level ()
Return the dimensionality of the Geometry, or -1 if unknown

copy ()
Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

splitProp (*prop=None*)
Partition a Geometry (Formex/Mesh) according to the values in prop.

Parameters:

- *prop*: an int array with length self.nelems(), or None. If None, the *prop* attribute of the Geometry is used.

Returns a list of Geometry objects of the same type as the input. Each object 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 their *prop* value.

It *prop* is None and the the object has no *prop* attribute, an empty list is returned.

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.

write (*fil*, *sep=' '*, *mode='w'*)
Write a Geometry to a .pgf file.

If fil is a string, a file with that name is opened. Else fil should be an open file. The Geometry is then written to that file in a native format, using sep as separator between the coordinates. If fil is a string, the file is closed prior to returning.

class 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 ()
Close a PolyLine.

If the PolyLine is already closed, it is returned unchanged. Else it is closed by adding a segment from the last to the first point (even if these are coincident).

> **Warning:** This method changes the PolyLine inplace.

open ()
Open a closed PolyLine.

If the PolyLine is closed, it is opened by removing the last segment. Else, it is returned unchanged.

> **Warning:** This method changes the PolyLine inplace.

Use split() if you want to open the PolyLine without losing a segment.

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.

approximate (*nseg, equidistant=True, npre=100*)
Approximate a PolyLine with a PolyLine of n segments

Parameters:

- *nseg*: number of straight segments of the resulting PolyLine
- *equidistant*: if True (default) the points are spaced almost equidistantly over the curve. If False, the points are spread equally over the parameter space.
- *npre*: 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.

Note: This is an alternative for Curve.approx, and may replace it in future.

---

roll (*n*)
Roll the points of a closed PolyLine.

lengths ()
Return the length of the parts of the curve.

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).

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*, ***kargs*)

Append another PolyLine to this one.

Returns the concatenation of two open PolyLines. Closed PolyLines cannot be concatenated.

insertPointsAt (*t*, *normalized=False*, *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*, *normalized=False*)

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.

endPoints ()

Return start and end points of the curve.

Returns a Coords with two points, or None if the curve is closed.

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).

addNoise (**args*, ***kargs*)
Apply 'addNoise' transformation to the Geometry object.

See `coords.Coords.addNoise()` for details.

affine (**args*, ***kargs*)
Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

align (**args*, ***kargs*)
Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

bump (**args*, ***kargs*)
Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

bump1 (**args*, ***kargs*)
Apply 'bump1' transformation to the Geometry object.

See `coords.Coords.bump1()` for details.

bump2 (**args*, ***kargs*)
Apply 'bump2' transformation to the Geometry object.

See `coords.Coords.bump2()` for details.

centered (**args*, ***kargs*)
Apply 'centered' transformation to the Geometry object.

See `coords.Coords.centered()` for details.

cylindrical (**args*, ***kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See `coords.Coords.cylindrical()` for details.

egg (**args*, ***kargs*)
Apply 'egg' transformation to the Geometry object.

See `coords.Coords.egg()` for details.

flare (**args*, ***kargs*)
Apply 'flare' transformation to the Geometry object.

See `coords.Coords.flare()` for details.

hyperCylindrical (**args*, ***kargs*)
Apply 'hyperCylindrical' transformation to the Geometry object.

See `coords.Coords.hyperCylindrical()` for details.

isopar (**args*, ***kargs*)
Apply 'isopar' transformation to the Geometry object.

See `coords.Coords.isopar()` for details.

map (**args*, ***kargs*)
Apply 'map' transformation to the Geometry object.

See `coords.Coords.map()` for details.

map1 (**args*, ***kargs*)
Apply 'map1' transformation to the Geometry object.

See `coords.Coords.map1()` for details.

mapd (**args*, ***kargs*)
Apply 'mapd' transformation to the Geometry object.

See `coords.Coords.mapd()` for details.

position (**args*, ***kargs*)
Apply 'position' transformation to the Geometry object.

See `coords.Coords.position()` for details.

projectOnCylinder (**args*, ***kargs*)
Apply 'projectOnCylinder' transformation to the Geometry object.

See `coords.Coords.projectOnCylinder()` for details.

projectOnPlane (**args*, ***kargs*)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (**args*, ***kargs*)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

reflect (**args*, ***kargs*)
Apply 'reflect' transformation to the Geometry object.

See `coords.Coords.reflect()` for details.

replace (**args*, ***kargs*)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

rollAxes (**args*, ***kargs*)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

rot (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

rotate (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

scale (**args*, ***kargs*)
Apply 'scale' transformation to the Geometry object.

See `coords.Coords.scale()` for details.

shear (**args*, ***kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

spherical (**args*, ***kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args*, ***kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

swapAxes (**args*, ***kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

toCylindrical (**args*, ***kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

toSpherical (**args*, ***kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

transformCS (**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

translate (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

trl (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

pointsAt (*t*, *normalized=False*, *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 normalized is True, the parameter values are give in a normalized space where 0 is the start of the curve and 1 is the end.

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.

toProp (*prop*)
Converts the argument into a legal set of properties for the object.

The conversion involves resizing the argument to a 1D array of length self.nelems(), and converting the data type to integer.

getCoords ()
Get the coords data.

Returns the full array of coordinates stored in the Geometry object. Note that subclasses may store more points in this array than are used to define the geometry.

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 all 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.

approx (*ndiv=None*, *ntot=None*)
Return a PolyLine approximation of the curve

Parameters:

- *ndiv*: int: number of straight segments to use over each part of the curve. This is only used if
- *ntot*: int: number of straight segments to use over the total length of the curve.

Returns a PolyLine approximation for the curve. *C.approx(ndiv=n)* returns an approximation with *ndiv* segments over each part of the curve. This may results in segments with very different lengths. *C.approx(ntot=n)* returns an approximation with *ntot* segments over the

total length of the curve. This produces more equally sized segments, but the internal end points of the curve parts may not be on the approximating Polyline.

level()
Return the dimensionality of the Geometry, or -1 if unknown

frenet(*ndiv=None*, *ntot=None*, *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* and *ntot*. 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 binormial 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

copy()
Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

splitProp(*prop=None*)
Partition a Geometry (Formex/Mesh) according to the values in prop.

Parameters:

- *prop*: an int array with length self.nelems(), or None. If None, the *prop* attribute of the Geometry is used.

Returns a list of Geometry objects of the same type as the input. Each object 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 their *prop* value.

It *prop* is None and the the object has no *prop* attribute, an empty list is returned.

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.

setProp (*p=None*)
Create or destroy the property array for the Formex.

A property array is a rank-1 integer array with dimension equal to the number of elements in the Formex (first dimension of data). You can specify a single value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

If a value None is given, the properties are removed from the Formex.

write (*fil*, *sep=' '*, *mode='w'*)
Write a Geometry to a .pgf file.

If fil is a string, a file with that name is opened. Else fil should be an open file. The Geometry is then written to that file in a native format, using sep as separator between the coordinates. If fil is a string, the file is closed prior to returning.

class `curve.`**Line** (*coords*)
A Line is a PolyLine with exactly two points.

Parameters:

- *coords*: compatible with (2,3) shaped float array

endPoints ()
Return start and end points of the curve.

Returns a Coords with two points, or None if the curve is closed.

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).

addNoise (**args*, ***kargs*)
Apply 'addNoise' transformation to the Geometry object.

See `coords.Coords.addNoise()` for details.

affine (**args*, ***kargs*)
Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

align (**args*, ***kargs*)
Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

bump (**args*, ***kargs*)
Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

bump1 (**args*, ***kargs*)
Apply 'bump1' transformation to the Geometry object.

See `coords.Coords.bump1()` for details.

bump2 (**args*, ***kargs*)
Apply 'bump2' transformation to the Geometry object.

See `coords.Coords.bump2()` for details.

centered (**args*, ***kargs*)
Apply 'centered' transformation to the Geometry object.

See `coords.Coords.centered()` for details.

cylindrical (**args*, ***kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See `coords.Coords.cylindrical()` for details.

egg (**args*, ***kargs*)
Apply 'egg' transformation to the Geometry object.

See `coords.Coords.egg()` for details.

flare (**args*, ***kargs*)
Apply 'flare' transformation to the Geometry object.

See `coords.Coords.flare()` for details.

hyperCylindrical (**args*, ***kargs*)
Apply 'hyperCylindrical' transformation to the Geometry object.

See `coords.Coords.hyperCylindrical()` for details.

isopar (**args*, ***kargs*)
Apply 'isopar' transformation to the Geometry object.

See `coords.Coords.isopar()` for details.

map (**args*, ***kargs*)
Apply 'map' transformation to the Geometry object.

See `coords.Coords.map()` for details.

map1 (**args*, ***kargs*)
Apply 'map1' transformation to the Geometry object.

See `coords.Coords.map1()` for details.

mapd (**args*, ***kargs*)
Apply 'mapd' transformation to the Geometry object.

See `coords.Coords.mapd()` for details.

position (**args*, ***kargs*)
Apply 'position' transformation to the Geometry object.

See `coords.Coords.position()` for details.

projectOnCylinder (**args*, ***kargs*)
Apply 'projectOnCylinder' transformation to the Geometry object.

See `coords.Coords.projectOnCylinder()` for details.

projectOnPlane (**args*, ***kargs*)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (**args*, ***kargs*)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

reflect (**args*, ***kargs*)
Apply 'reflect' transformation to the Geometry object.

See `coords.Coords.reflect()` for details.

replace (**args*, ***kargs*)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

rollAxes (**args*, ***kargs*)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

rot (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

rotate (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

scale (**args*, ***kargs*)
Apply 'scale' transformation to the Geometry object.

See `coords.Coords.scale()` for details.

shear (**args*, ***kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

spherical (**args*, ***kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args*, ***kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

swapAxes (**args*, ***kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

toCylindrical (**args*, ***kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

toSpherical (**args*, ***kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

transformCS (**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

translate (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

trl (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

pointsAt (*t*, *normalized=False*, *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 normalized is True, the parameter values are give in a normalized space where 0 is the start of the curve and 1 is the end.

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.

toProp (*prop*)
Converts the argument into a legal set of properties for the object.

The conversion involves resizing the argument to a 1D array of length self.nelems(), and converting the data type to integer.

getCoords ()
Get the coords data.

Returns the full array of coordinates stored in the Geometry object. Note that subclasses may store more points in this array than are used to define the geometry.

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 all 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.

approx (*ndiv=None*, *ntot=None*)
Return a PolyLine approximation of the curve

Parameters:

- *ndiv*: int: number of straight segments to use over each part of the curve. This is only used if
- *ntot*: int: number of straight segments to use over the total length of the curve.

Returns a PolyLine approximation for the curve. *C.approx(ndiv=n)* returns an approximation with *ndiv* segments over each part of the curve. This may results in segments with very different lengths. *C.approx(ntot=n)* returns an approximation with *ntot* segments over the total length of the curve. This produces more equally sized segments, but the internal end points of the curve parts may not be on the approximating Polyline.

level ()
Return the dimensionality of the Geometry, or -1 if unknown

frenet (*ndiv=None*, *ntot=None*, *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* and *ntot*. 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 binormial 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

copy()

Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

splitProp(*prop=None*)

Partition a Geometry (Formex/Mesh) according to the values in prop.

Parameters:

- *prop*: an int array with length self.nelems(), or None. If None, the *prop* attribute of the Geometry is used.

Returns a list of Geometry objects of the same type as the input. Each object 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 their *prop* value.

It *prop* is None and the the object has no *prop* attribute, an empty list is returned.

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.

setProp(*p=None*)

Create or destroy the property array for the Formex.

A property array is a rank-1 integer array with dimension equal to the number of elements in the Formex (first dimension of data). You can specify a single value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

If a value None is given, the properties are removed from the Formex.

close()
Close a PolyLine.

If the PolyLine is already closed, it is returned unchanged. Else it is closed by adding a segment from the last to the first point (even if these are coincident).

> **Warning:** This method changes the PolyLine inplace.

open()
Open a closed PolyLine.

If the PolyLine is closed, it is opened by removing the last segment. Else, it is returned unchanged.

> **Warning:** This method changes the PolyLine inplace.

Use `split()` if you want to open the PolyLine without losing a segment.

write(*fil*, *sep=' '*, *mode='w'*)
Write a Geometry to a .pgf file.

If fil is a string, a file with that name is opened. Else fil should be an open file. The Geometry is then written to that file in a native format, using sep as separator between the coordinates. If fil is a string, the file is closed prior to returning.

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.

approximate (*nseg*, *equidistant=True*, *npre=100*)
Approximate a PolyLine with a PolyLine of n segments

Parameters:

- *nseg*: number of straight segments of the resulting PolyLine
- *equidistant*: if True (default) the points are spaced almost equidistantly over the curve. If False, the points are spread equally over the parameter space.
- *npre*: 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.

Note: This is an alternative for Curve.approx, and may replace it in future.

roll (*n*)
Roll the points of a closed PolyLine.

lengths ()
Return the length of the parts of the curve.

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).

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*, ***kargs*)
Append another PolyLine to this one.

Returns the concatenation of two open PolyLines. Closed PolyLines cannot be concatenated.

insertPointsAt (*t*, *normalized=False*, *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*, *normalized=False*)
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.

class 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,2,3) or array(ncontrol,3) If *coords* was specified, this should be a (nparts,2,3) array with the intermediate control points, two for each part.

If *coords* was not specified, this should be the full array of *ncontrol* control points for the curve. The number of points should be a multiple of 3 plus 1. If the curve is closed, the last point is equal to the first and does not need to a multiple of 3 is also allowed, in which case the first point will be appended as last.

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) Specfies the degree of the curve. Default is 3.

•*endzerocurv* : 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*)
Returns the points defining part j of the curve.

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.

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'.

approx_by_subdivision (*tol=0.001*)
Return a PolyLine approximation of the curve.

tol is a tolerance value for the flatness of the curve. The flatness of each part is calculated as the maximum orthogonal distance of its intermediate control points from the straight segment through its end points.

Parts for which the distance is larger than tol are subdivided using de Casteljau's algorithm. The subdivision stops if all parts are sufficiently flat. The return value is a PolyLine connecting the end points of all parts.

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.

endPoints ()
Return start and end points of the curve.

Returns a Coords with two points, or None if the curve is closed.

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_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).

addNoise (**args*, ***kargs*)
Apply 'addNoise' transformation to the Geometry object.

See `coords.Coords.addNoise()` for details.

affine (**args*, ***kargs*)
Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

align (**args*, ***kargs*)
Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

bump (**args*, ***kargs*)
Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

bump1 (**args*, ***kargs*)
Apply 'bump1' transformation to the Geometry object.

See `coords.Coords.bump1()` for details.

bump2 (**args*, ***kargs*)
Apply 'bump2' transformation to the Geometry object.

See `coords.Coords.bump2()` for details.

centered (**args*, ***kargs*)
Apply 'centered' transformation to the Geometry object.

See `coords.Coords.centered()` for details.

cylindrical (**args*, ***kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See `coords.Coords.cylindrical()` for details.

egg (**args*, ***kargs*)
Apply 'egg' transformation to the Geometry object.

See `coords.Coords.egg()` for details.

flare (**args*, ***kargs*)
Apply 'flare' transformation to the Geometry object.

See `coords.Coords.flare()` for details.

hyperCylindrical (**args*, ***kargs*)
Apply 'hyperCylindrical' transformation to the Geometry object.

See `coords.Coords.hyperCylindrical()` for details.

isopar (**args*, ***kargs*)
Apply 'isopar' transformation to the Geometry object.

See `coords.Coords.isopar()` for details.

map (**args*, ***kargs*)
Apply 'map' transformation to the Geometry object.

See `coords.Coords.map()` for details.

map1 (**args*, ***kargs*)
Apply 'map1' transformation to the Geometry object.

See `coords.Coords.map1()` for details.

mapd (**args*, ***kargs*)
Apply 'mapd' transformation to the Geometry object.

See `coords.Coords.mapd()` for details.

position (**args*, ***kargs*)
Apply 'position' transformation to the Geometry object.

See `coords.Coords.position()` for details.

projectOnCylinder (**args*, ***kargs*)
Apply 'projectOnCylinder' transformation to the Geometry object.

See `coords.Coords.projectOnCylinder()` for details.

projectOnPlane (**args*, ***kargs*)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (**args*, ***kargs*)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

reflect (**args*, ***kargs*)
Apply 'reflect' transformation to the Geometry object.

See `coords.Coords.reflect()` for details.

replace (**args*, ***kargs*)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

rollAxes (**args*, ***kargs*)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

rot (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

rotate (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

scale (**args*, ***kargs*)
Apply 'scale' transformation to the Geometry object.

See `coords.Coords.scale()` for details.

shear (**args*, ***kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

spherical (**args*, ***kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args*, ***kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

swapAxes (**args*, ***kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

toCylindrical (**args*, ***kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

toSpherical (**args*, ***kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

transformCS (**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

translate (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

trl (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

pointsAt (*t*, *normalized=False*, *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 normalized is True, the parameter values are give in a normalized space where 0 is the start of the curve and 1 is the end.

If return_position is True, also returns the part numbers on which the point are lying and the local parameter values.

nelems ()
Return the number of elements in the Geometry.

This method should be re-implemented by the derived classes. For the (empty) Geometry class it always returns 0.

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.

toProp (*prop*)

Converts the argument into a legal set of properties for the object.

The conversion involves resizing the argument to a 1D array of length self.nelems(), and converting the data type to integer.

getCoords ()

Get the coords data.

Returns the full array of coordinates stored in the Geometry object. Note that subclasses may store more points in this array than are used to define the geometry.

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 all 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.

approx (*ndiv=None*, *ntot=None*)

Return a PolyLine approximation of the curve

Parameters:

- *ndiv*: int: number of straight segments to use over each part of the curve. This is only used if
- *ntot*: int: number of straight segments to use over the total length of the curve.

Returns a PolyLine approximation for the curve. *C.approx(ndiv=n)* returns an approximation with *ndiv* segments over each part of the curve. This may results in segments with very different lengths. *C.approx(ntot=n)* returns an approximation with *ntot* segments over the total length of the curve. This produces more equally sized segments, but the internal end points of the curve parts may not be on the approximating Polyline.

level ()

Return the dimensionality of the Geometry, or -1 if unknown

frenet (*ndiv=None*, *ntot=None*, *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* and *ntot*. 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 binormial 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

copy()

Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

splitProp(*prop=None*)

Partition a Geometry (Formex/Mesh) according to the values in prop.

Parameters:

- *prop*: an int array with length self.nelems(), or None. If None, the *prop* attribute of the Geometry is used.

Returns a list of Geometry objects of the same type as the input. Each object 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 their *prop* value.

It *prop* is None and the the object has no *prop* attribute, an empty list is returned.

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.

approximate (*nseg*, *equidistant=True*, *npre=100*)
Approximate a Curve with a PolyLine of n segments

Parameters:

- *nseg*: number of straight segments of the resulting PolyLine
- *equidistant*: if True (default) the points are spaced almost equidistantly over the curve. If False, the points are spread equally over the parameter space.
- *npre*: 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.

Note: This is an alternative for Curve.approx, and may replace it in future.

---

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.

setProp (*p=None*)
Create or destroy the property array for the Formex.

A property array is a rank-1 integer array with dimension equal to the number of elements in the Formex (first dimension of data). You can specify a single value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

If a value None is given, the properties are removed from the Formex.

write (*fil*, *sep=' '*, *mode='w'*)
Write a Geometry to a .pgf file.

If fil is a string, a file with that name is opened. Else fil should be an open file. The Geometry is then written to that file in a native format, using sep as separator between the coordinates. If fil is a string, the file is closed prior to returning.

class 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).

endPoints ()
Return start and end points of the curve.

Returns a Coords with two points, or None if the curve is closed.

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_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).

addNoise (**args, **kargs*)
Apply 'addNoise' transformation to the Geometry object.

See `coords.Coords.addNoise()` for details.

affine (**args, **kargs*)
Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

align (**args, **kargs*)
Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

bump (**args, **kargs*)
Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

bump1 (**args, **kargs*)
Apply 'bump1' transformation to the Geometry object.

See `coords.Coords.bump1()` for details.

bump2 (**args, **kargs*)
Apply 'bump2' transformation to the Geometry object.

See `coords.Coords.bump2()` for details.

centered (**args, **kargs*)
Apply 'centered' transformation to the Geometry object.

See `coords.Coords.centered()` for details.

cylindrical (**args, **kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See `coords.Coords.cylindrical()` for details.

egg (**args, **kargs*)
Apply 'egg' transformation to the Geometry object.

See `coords.Coords.egg()` for details.

flare (**args, **kargs*)
Apply 'flare' transformation to the Geometry object.

See `coords.Coords.flare()` for details.

hyperCylindrical (*args, **kargs)
Apply 'hyperCylindrical' transformation to the Geometry object.

See `coords.Coords.hyperCylindrical()` for details.

isopar (*args, **kargs)
Apply 'isopar' transformation to the Geometry object.

See `coords.Coords.isopar()` for details.

map (*args, **kargs)
Apply 'map' transformation to the Geometry object.

See `coords.Coords.map()` for details.

map1 (*args, **kargs)
Apply 'map1' transformation to the Geometry object.

See `coords.Coords.map1()` for details.

mapd (*args, **kargs)
Apply 'mapd' transformation to the Geometry object.

See `coords.Coords.mapd()` for details.

position (*args, **kargs)
Apply 'position' transformation to the Geometry object.

See `coords.Coords.position()` for details.

projectOnCylinder (*args, **kargs)
Apply 'projectOnCylinder' transformation to the Geometry object.

See `coords.Coords.projectOnCylinder()` for details.

projectOnPlane (*args, **kargs)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (*args, **kargs)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

reflect (*args, **kargs)
Apply 'reflect' transformation to the Geometry object.

See `coords.Coords.reflect()` for details.

replace (*args, **kargs)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

rollAxes (*args, **kargs)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

rot (*args, **kargs)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

rotate (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

scale (**args*, ***kargs*)
Apply 'scale' transformation to the Geometry object.

See `coords.Coords.scale()` for details.

shear (**args*, ***kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

spherical (**args*, ***kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args*, ***kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

swapAxes (**args*, ***kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

toCylindrical (**args*, ***kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

toSpherical (**args*, ***kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

transformCS (**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

translate (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

trl (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

pointsAt (*t*, *normalized=False*, *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 normalized is True, the parameter values are give in a normalized space where 0 is the start of the curve and 1 is the end.

If return_position is True, also returns the part numbers on which the point are lying and the local parameter values.

nelems()
Return the number of elements in the Geometry.

This method should be re-implemented by the derived classes. For the (empty) Geometry class it always returns 0.

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.

toProp(*prop*)
Converts the argument into a legal set of properties for the object.

The conversion involves resizing the argument to a 1D array of length self.nelems(), and converting the data type to integer.

getCoords()
Get the coords data.

Returns the full array of coordinates stored in the Geometry object. Note that subclasses may store more points in this array than are used to define the geometry.

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 all 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.

approx (*ndiv=None*, *ntot=None*)
Return a PolyLine approximation of the curve

Parameters:

- *ndiv*: int: number of straight segments to use over each part of the curve. This is only used if
- *ntot*: int: number of straight segments to use over the total length of the curve.

Returns a PolyLine approximation for the curve. *C.approx(ndiv=n)* returns an approximation with *ndiv* segments over each part of the curve. This may results in segments with very different lengths. *C.approx(ntot=n)* returns an approximation with *ntot* segments over the total length of the curve. This produces more equally sized segments, but the internal end points of the curve parts may not be on the approximating Polyline.

level ()
Return the dimensionality of the Geometry, or -1 if unknown

frenet (*ndiv=None*, *ntot=None*, *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* and *ntot*. 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 binormial 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

copy ()
Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

splitProp (*prop=None*)
Partition a Geometry (Formex/Mesh) according to the values in prop.

Parameters:

- *prop*: an int array with length self.nelems(), or None. If None, the *prop* attribute of the Geometry is used.

Returns a list of Geometry objects of the same type as the input. Each object 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 their *prop* value.

It *prop* is None and the the object has no *prop* attribute, an empty list is returned.

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.

approximate (*nseg*, *equidistant=True*, *npre=100*)
Approximate a Curve with a PolyLine of n segments

Parameters:

- *nseg*: number of straight segments of the resulting PolyLine
- *equidistant*: if True (default) the points are spaced almost equidistantly over the curve. If False, the points are spread equally over the parameter space.
- *npre*: 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.

Note: This is an alternative for Curve.approx, and may replace it in future.

---

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.

setProp (*p=None*)
Create or destroy the property array for the Formex.

A property array is a rank-1 integer array with dimension equal to the number of elements in the Formex (first dimension of data). You can specify a single value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

If a value None is given, the properties are removed from the Formex.

write (*fil*, *sep=' '*, *mode='w'*)
Write a Geometry to a .pgf file.

If fil is a string, a file with that name is opened. Else fil should be an open file. The Geometry is then written to that file in a native format, using sep as separator between the coordinates. If fil is a string, the file is closed prior to returning.

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*)
Returns the points defining part j of the curve.

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.

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'.

approx_by_subdivision (*tol=0.001*)
Return a PolyLine approximation of the curve.

tol is a tolerance value for the flatness of the curve. The flatness of each part is calculated as the maximum orthogonal distance of its intermediate control points from the straight segment through its end points.

Parts for which the distance is larger than tol are subdivided using de Casteljau's algorithm. The subdivision stops if all parts are sufficiently flat. The return value is a PolyLine connecting the end points of all parts.

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 curve.**CardinalSpline2** (*coords*, *tension=0.0*, *closed=False*)
A class representing a cardinal spline.

endPoints ()
Return start and end points of the curve.

Returns a Coords with two points, or None if the curve is closed.

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).

addNoise (**args*, ***kargs*)
Apply 'addNoise' transformation to the Geometry object.

See `coords.Coords.addNoise()` for details.

affine (**args*, ***kargs*)
Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

align (**args*, ***kargs*)
Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

bump (**args*, ***kargs*)
Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

bump1 (**args*, ***kargs*)
Apply 'bump1' transformation to the Geometry object.

See `coords.Coords.bump1()` for details.

bump2 (**args*, ***kargs*)
Apply 'bump2' transformation to the Geometry object.

See `coords.Coords.bump2()` for details.

centered (**args*, ***kargs*)
Apply 'centered' transformation to the Geometry object.

See `coords.Coords.centered()` for details.

cylindrical (**args*, ***kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See `coords.Coords.cylindrical()` for details.

egg (**args*, ***kargs*)
Apply 'egg' transformation to the Geometry object.

See `coords.Coords.egg()` for details.

flare (**args*, ***kargs*)
Apply 'flare' transformation to the Geometry object.

See `coords.Coords.flare()` for details.

hyperCylindrical (**args*, ***kargs*)
Apply 'hyperCylindrical' transformation to the Geometry object.

See `coords.Coords.hyperCylindrical()` for details.

isopar (**args*, ***kargs*)
Apply 'isopar' transformation to the Geometry object.

See `coords.Coords.isopar()` for details.

map (**args*, ***kargs*)
Apply 'map' transformation to the Geometry object.

See `coords.Coords.map()` for details.

map1 (**args*, ***kargs*)
Apply 'map1' transformation to the Geometry object.

See `coords.Coords.map1()` for details.

mapd (**args*, ***kargs*)
Apply 'mapd' transformation to the Geometry object.

See `coords.Coords.mapd()` for details.

position (**args*, ***kargs*)
Apply 'position' transformation to the Geometry object.

See `coords.Coords.position()` for details.

projectOnCylinder (**args*, ***kargs*)
Apply 'projectOnCylinder' transformation to the Geometry object.

See `coords.Coords.projectOnCylinder()` for details.

projectOnPlane (**args*, ***kargs*)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (**args, **kargs*)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

reflect (**args, **kargs*)
Apply 'reflect' transformation to the Geometry object.

See `coords.Coords.reflect()` for details.

replace (**args, **kargs*)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

rollAxes (**args, **kargs*)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

rot (**args, **kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

rotate (**args, **kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

scale (**args, **kargs*)
Apply 'scale' transformation to the Geometry object.

See `coords.Coords.scale()` for details.

shear (**args, **kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

spherical (**args, **kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args, **kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

swapAxes (**args, **kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

toCylindrical (**args, **kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

toSpherical (**args, **kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

transformCS (**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

translate (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

trl (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

pointsAt (*t*, *normalized=False*, *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 normalized is True, the parameter values are give in a normalized space where 0 is the start of the curve and 1 is the end.

If return_position is True, also returns the part numbers on which the point are lying and the local parameter values.

nelems ()
Return the number of elements in the Geometry.

This method should be re-implemented by the derived classes. For the (empty) Geometry class it always returns 0.

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.

toProp (*prop*)
Converts the argument into a legal set of properties for the object.

The conversion involves resizing the argument to a 1D array of length self.nelems(), and converting the data type to integer.

getCoords()
Get the coords data.

Returns the full array of coordinates stored in the Geometry object. Note that subclasses may store more points in this array than are used to define the geometry.

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 all 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.

approx(*ndiv=None*, *ntot=None*)
Return a PolyLine approximation of the curve

Parameters:

- *ndiv*: int: number of straight segments to use over each part of the curve. This is only used if
- *ntot*: int: number of straight segments to use over the total length of the curve.

Returns a PolyLine approximation for the curve. *C.approx(ndiv=n)* returns an approximation with *ndiv* segments over each part of the curve. This may results in segments with very different lengths. *C.approx(ntot=n)* returns an approximation with *ntot* segments over the total length of the curve. This produces more equally sized segments, but the internal end points of the curve parts may not be on the approximating Polyline.

level()
Return the dimensionality of the Geometry, or -1 if unknown

frenet(*ndiv=None*, *ntot=None*, *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* and *ntot*. 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 binormial 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

copy()

Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

splitProp(*prop=None*)

Partition a Geometry (Formex/Mesh) according to the values in prop.

Parameters:

- *prop*: an int array with length self.nelems(), or None. If None, the *prop* attribute of the Geometry is used.

Returns a list of Geometry objects of the same type as the input. Each object 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 their *prop* value.

It *prop* is None and the the object has no *prop* attribute, an empty list is returned.

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.

approximate(*nseg*, *equidistant=True*, *npre=100*)

Approximate a Curve with a PolyLine of n segments

Parameters:

- *nseg*: number of straight segments of the resulting PolyLine
- *equidistant*: if True (default) the points are spaced almost equidistantly over the curve. If False, the points are spread equally over the parameter space.
- *npre*: 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.

Note: This is an alternative for Curve.approx, and may replace it in future.

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.

setProp (*p=None*)
Create or destroy the property array for the Formex.

A property array is a rank-1 integer array with dimension equal to the number of elements in the Formex (first dimension of data). You can specify a single value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

If a value None is given, the properties are removed from the Formex.

write (*fil*, *sep=' '*, *mode='w'*)
Write a Geometry to a .pgf file.

If fil is a string, a file with that name is opened. Else fil should be an open file. The Geometry is then written to that file in a native format, using sep as separator between the coordinates. If fil is a string, the file is closed prior to returning.

class curve.**NaturalSpline** (*coords*, *closed=False*, *endzerocurv=False*)
A class representing a natural spline.

endPoints ()
Return start and end points of the curve.

Returns a Coords with two points, or None if the curve is closed.

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).

addNoise (**args*, ***kargs*)
Apply 'addNoise' transformation to the Geometry object.

See coords.Coords.addNoise() for details.

affine (**args*, ***kargs*)
Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

align (**args*, ***kargs*)
Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

bump (**args*, ***kargs*)
Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

bump1 (**args*, ***kargs*)
Apply 'bump1' transformation to the Geometry object.

See `coords.Coords.bump1()` for details.

bump2 (**args*, ***kargs*)
Apply 'bump2' transformation to the Geometry object.

See `coords.Coords.bump2()` for details.

centered (**args*, ***kargs*)
Apply 'centered' transformation to the Geometry object.

See `coords.Coords.centered()` for details.

cylindrical (**args*, ***kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See `coords.Coords.cylindrical()` for details.

egg (**args*, ***kargs*)
Apply 'egg' transformation to the Geometry object.

See `coords.Coords.egg()` for details.

flare (**args*, ***kargs*)
Apply 'flare' transformation to the Geometry object.

See `coords.Coords.flare()` for details.

hyperCylindrical (**args*, ***kargs*)
Apply 'hyperCylindrical' transformation to the Geometry object.

See `coords.Coords.hyperCylindrical()` for details.

isopar (**args*, ***kargs*)
Apply 'isopar' transformation to the Geometry object.

See `coords.Coords.isopar()` for details.

map (**args*, ***kargs*)
Apply 'map' transformation to the Geometry object.

See `coords.Coords.map()` for details.

map1 (**args*, ***kargs*)
Apply 'map1' transformation to the Geometry object.

See `coords.Coords.map1()` for details.

mapd (**args*, ***kargs*)
Apply 'mapd' transformation to the Geometry object.

See `coords.Coords.mapd()` for details.

position (**args*, ***kargs*)
Apply 'position' transformation to the Geometry object.

See `coords.Coords.position()` for details.

projectOnCylinder (**args*, ***kargs*)
Apply 'projectOnCylinder' transformation to the Geometry object.

See `coords.Coords.projectOnCylinder()` for details.

projectOnPlane (**args*, ***kargs*)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (**args*, ***kargs*)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

reflect (**args*, ***kargs*)
Apply 'reflect' transformation to the Geometry object.

See `coords.Coords.reflect()` for details.

replace (**args*, ***kargs*)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

rollAxes (**args*, ***kargs*)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

rot (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

rotate (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

scale (**args*, ***kargs*)
Apply 'scale' transformation to the Geometry object.

See `coords.Coords.scale()` for details.

shear (**args*, ***kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

spherical (**args*, ***kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args*, ***kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

swapAxes (**args*, ***kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

toCylindrical (**args*, ***kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

toSpherical (**args*, ***kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

transformCS (**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

translate (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

trl (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

pointsAt (*t*, *normalized=False*, *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 normalized is True, the parameter values are give in a normalized space where 0 is the start of the curve and 1 is the end.

If return_position is True, also returns the part numbers on which the point are lying and the local parameter values.

nelems ()
Return the number of elements in the Geometry.

This method should be re-implemented by the derived classes. For the (empty) Geometry class it always returns 0.

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.

toProp (*prop*)
Converts the argument into a legal set of properties for the object.

The conversion involves resizing the argument to a 1D array of length self.nelems(), and converting the data type to integer.

getCoords ()
Get the coords data.

Returns the full array of coordinates stored in the Geometry object. Note that subclasses may store more points in this array than are used to define the geometry.

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 all 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.

approx (*ndiv=None*, *ntot=None*)
Return a PolyLine approximation of the curve

Parameters:

- *ndiv*: int: number of straight segments to use over each part of the curve. This is only used if
- *ntot*: int: number of straight segments to use over the total length of the curve.

Returns a PolyLine approximation for the curve. *C.approx(ndiv=n)* returns an approximation with *ndiv* segments over each part of the curve. This may results in segments with very different lengths. *C.approx(ntot=n)* returns an approximation with *ntot* segments over the total length of the curve. This produces more equally sized segments, but the internal end points of the curve parts may not be on the approximating Polyline.

level ()
Return the dimensionality of the Geometry, or -1 if unknown

frenet (*ndiv=None*, *ntot=None*, *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* and *ntot*. 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 binormial 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

copy()

Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

splitProp(*prop=None*)

Partition a Geometry (Formex/Mesh) according to the values in prop.

Parameters:

- *prop*: an int array with length self.nelems(), or None. If None, the *prop* attribute of the Geometry is used.

Returns a list of Geometry objects of the same type as the input. Each object 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 their *prop* value.

It *prop* is None and the the object has no *prop* attribute, an empty list is returned.

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.

approximate (*nseg*, *equidistant=True*, *npre=100*)
Approximate a Curve with a PolyLine of n segments

Parameters:

- *nseg*: number of straight segments of the resulting PolyLine
- *equidistant*: if True (default) the points are spaced almost equidistantly over the curve. If False, the points are spread equally over the parameter space.
- *npre*: 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.

Note: This is an alternative for Curve.approx, and may replace it in future.

---

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.

setProp (*p=None*)
Create or destroy the property array for the Formex.

A property array is a rank-1 integer array with dimension equal to the number of elements in the Formex (first dimension of data). You can specify a single value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

If a value None is given, the properties are removed from the Formex.

write (*fil*, *sep=' '*, *mode='w'*)
Write a Geometry to a .pgf file.

If fil is a string, a file with that name is opened. Else fil should be an open file. The Geometry is then written to that file in a native format, using sep as separator between the coordinates. If fil is a string, the file is closed prior to returning.

class curve.**Arc3** (*coords*)
A class representing a circular arc.

endPoints ()
Return start and end points of the curve.

Returns a Coords with two points, or None if the curve is closed.

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).

addNoise (**args*, ***kargs*)
Apply 'addNoise' transformation to the Geometry object.

See `coords.Coords.addNoise()` for details.

affine (**args*, ***kargs*)
Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

align (**args*, ***kargs*)
Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

bump (**args*, ***kargs*)
Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

bump1 (**args*, ***kargs*)
Apply 'bump1' transformation to the Geometry object.

See `coords.Coords.bump1()` for details.

bump2 (**args*, ***kargs*)
Apply 'bump2' transformation to the Geometry object.

See `coords.Coords.bump2()` for details.

centered (**args*, ***kargs*)
Apply 'centered' transformation to the Geometry object.

See `coords.Coords.centered()` for details.

cylindrical (**args*, ***kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See `coords.Coords.cylindrical()` for details.

egg (**args*, ***kargs*)
Apply 'egg' transformation to the Geometry object.

See `coords.Coords.egg()` for details.

flare (**args*, ***kargs*)
Apply 'flare' transformation to the Geometry object.

See `coords.Coords.flare()` for details.

hyperCylindrical (**args*, ***kargs*)
Apply 'hyperCylindrical' transformation to the Geometry object.

See `coords.Coords.hyperCylindrical()` for details.

isopar (**args*, ***kargs*)
Apply 'isopar' transformation to the Geometry object.

See `coords.Coords.isopar()` for details.

map (**args*, ***kargs*)
Apply 'map' transformation to the Geometry object.

See `coords.Coords.map()` for details.

map1 (**args*, ***kargs*)
Apply 'map1' transformation to the Geometry object.

See `coords.Coords.map1()` for details.

mapd (**args*, ***kargs*)
Apply 'mapd' transformation to the Geometry object.

See `coords.Coords.mapd()` for details.

position (**args*, ***kargs*)
Apply 'position' transformation to the Geometry object.

See `coords.Coords.position()` for details.

projectOnCylinder (**args*, ***kargs*)
Apply 'projectOnCylinder' transformation to the Geometry object.

See `coords.Coords.projectOnCylinder()` for details.

projectOnPlane (**args*, ***kargs*)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (**args*, ***kargs*)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

reflect (**args*, ***kargs*)
Apply 'reflect' transformation to the Geometry object.

See `coords.Coords.reflect()` for details.

replace (**args*, ***kargs*)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

rollAxes (**args*, ***kargs*)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

rot (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

rotate (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

scale (**args*, ***kargs*)
Apply 'scale' transformation to the Geometry object.

See `coords.Coords.scale()` for details.

shear (**args*, ***kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

spherical (**args*, ***kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args*, ***kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

swapAxes (**args*, ***kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

toCylindrical (**args*, ***kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

toSpherical (**args*, ***kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

transformCS (**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

translate (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

trl (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

pointsAt (*t*, *normalized=False*, *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 normalized is True, the parameter values are give in a normalized space where 0 is the start of the curve and 1 is the end.

If return_position is True, also returns the part numbers on which the point are lying and the local parameter values.

nelems()
Return the number of elements in the Geometry.

This method should be re-implemented by the derived classes. For the (empty) Geometry class it always returns 0.

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.

toProp(*prop*)
Converts the argument into a legal set of properties for the object.

The conversion involves resizing the argument to a 1D array of length self.nelems(), and converting the data type to integer.

getCoords()
Get the coords data.

Returns the full array of coordinates stored in the Geometry object. Note that subclasses may store more points in this array than are used to define the geometry.

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 all 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.

approx (*ndiv=None*, *ntot=None*)
Return a PolyLine approximation of the curve

Parameters:

- *ndiv*: int: number of straight segments to use over each part of the curve. This is only used if
- *ntot*: int: number of straight segments to use over the total length of the curve.

Returns a PolyLine approximation for the curve. *C.approx(ndiv=n)* returns an approximation with *ndiv* segments over each part of the curve. This may results in segments with very different lengths. *C.approx(ntot=n)* returns an approximation with *ntot* segments over the total length of the curve. This produces more equally sized segments, but the internal end points of the curve parts may not be on the approximating Polyline.

level ()
Return the dimensionality of the Geometry, or -1 if unknown

frenet (*ndiv=None*, *ntot=None*, *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* and *ntot*. 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 binormial 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

copy ()
Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

splitProp (*prop=None*)
Partition a Geometry (Formex/Mesh) according to the values in prop.

Parameters:

- *prop*: an int array with length self.nelems(), or None. If None, the *prop* attribute of the Geometry is used.

Returns a list of Geometry objects of the same type as the input. Each object 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 their *prop* value.

It *prop* is None and the the object has no *prop* attribute, an empty list is returned.

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.

approximate (*nseg*, *equidistant=True*, *npre=100*)
Approximate a Curve with a PolyLine of n segments

Parameters:

- *nseg*: number of straight segments of the resulting PolyLine
- *equidistant*: if True (default) the points are spaced almost equidistantly over the curve. If False, the points are spread equally over the parameter space.
- *npre*: 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.

Note: This is an alternative for Curve.approx, and may replace it in future.

---

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.

setProp (*p=None*)
Create or destroy the property array for the Formex.

A property array is a rank-1 integer array with dimension equal to the number of elements in the Formex (first dimension of data). You can specify a single value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

If a value None is given, the properties are removed from the Formex.

write (*fil*, *sep=' '*, *mode='w'*)
Write a Geometry to a .pgf file.

If fil is a string, a file with that name is opened. Else fil should be an open file. The Geometry is then written to that file in a native format, using sep as separator between the coordinates. If fil is a string, the file is closed prior to returning.

class `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.

`endPoints`()

Return start and end points of the curve.

Returns a Coords with two points, or None if the curve is closed.

`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_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).

`addNoise`(**args*, ***kargs*)

Apply 'addNoise' transformation to the Geometry object.

See `coords.Coords.addNoise()` for details.

`affine`(**args*, ***kargs*)

Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

`align`(**args*, ***kargs*)

Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

`bump`(**args*, ***kargs*)

Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

`bump1`(**args*, ***kargs*)

Apply 'bump1' transformation to the Geometry object.

See `coords.Coords.bump1()` for details.

bump2 (**args*, ***kargs*)
Apply 'bump2' transformation to the Geometry object.

See `coords.Coords.bump2()` for details.

centered (**args*, ***kargs*)
Apply 'centered' transformation to the Geometry object.

See `coords.Coords.centered()` for details.

cylindrical (**args*, ***kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See `coords.Coords.cylindrical()` for details.

egg (**args*, ***kargs*)
Apply 'egg' transformation to the Geometry object.

See `coords.Coords.egg()` for details.

flare (**args*, ***kargs*)
Apply 'flare' transformation to the Geometry object.

See `coords.Coords.flare()` for details.

hyperCylindrical (**args*, ***kargs*)
Apply 'hyperCylindrical' transformation to the Geometry object.

See `coords.Coords.hyperCylindrical()` for details.

isopar (**args*, ***kargs*)
Apply 'isopar' transformation to the Geometry object.

See `coords.Coords.isopar()` for details.

map (**args*, ***kargs*)
Apply 'map' transformation to the Geometry object.

See `coords.Coords.map()` for details.

map1 (**args*, ***kargs*)
Apply 'map1' transformation to the Geometry object.

See `coords.Coords.map1()` for details.

mapd (**args*, ***kargs*)
Apply 'mapd' transformation to the Geometry object.

See `coords.Coords.mapd()` for details.

position (**args*, ***kargs*)
Apply 'position' transformation to the Geometry object.

See `coords.Coords.position()` for details.

projectOnCylinder (**args*, ***kargs*)
Apply 'projectOnCylinder' transformation to the Geometry object.

See `coords.Coords.projectOnCylinder()` for details.

projectOnPlane (**args*, ***kargs*)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (**args*, ***kargs*)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

reflect (**args*, ***kargs*)
Apply 'reflect' transformation to the Geometry object.

See `coords.Coords.reflect()` for details.

replace (**args*, ***kargs*)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

rollAxes (**args*, ***kargs*)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

rot (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

rotate (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

scale (**args*, ***kargs*)
Apply 'scale' transformation to the Geometry object.

See `coords.Coords.scale()` for details.

shear (**args*, ***kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

spherical (**args*, ***kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args*, ***kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

swapAxes (**args*, ***kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

toCylindrical (**args*, ***kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

toSpherical (**args*, ***kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

transformCS (**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

translate (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

trl (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

pointsAt (*t*, *normalized=False*, *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 normalized is True, the parameter values are give in a normalized space where 0 is the start of the curve and 1 is the end.

If return_position is True, also returns the part numbers on which the point are lying and the local parameter values.

nelems ()
Return the number of elements in the Geometry.

This method should be re-implemented by the derived classes. For the (empty) Geometry class it always returns 0.

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.

toProp (*prop*)
Converts the argument into a legal set of properties for the object.

The conversion involves resizing the argument to a 1D array of length self.nelems(), and converting the data type to integer.

getCoords()
Get the coords data.

Returns the full array of coordinates stored in the Geometry object. Note that subclasses may store more points in this array than are used to define the geometry.

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 all 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.

level()
Return the dimensionality of the Geometry, or -1 if unknown

frenet(*ndiv=None*, *ntot=None*, *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* and *ntot*. 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 binormial 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

copy()
Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

splitProp(*prop=None*)
Partition a Geometry (Formex/Mesh) according to the values in prop.

Parameters:

- *prop*: an int array with length self.nelems(), or None. If None, the *prop* attribute of the Geometry is used.

Returns a list of Geometry objects of the same type as the input. Each object 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 their *prop* value.

It *prop* is None and the the object has no *prop* attribute, an empty list is returned.

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.

approximate(*nseg*, *equidistant=True*, *npre=100*)
Approximate a Curve with a PolyLine of n segments

Parameters:

- *nseg*: number of straight segments of the resulting PolyLine
- *equidistant*: if True (default) the points are spaced almost equidistantly over the curve. If False, the points are spread equally over the parameter space.
- *npre*: 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.

Note: This is an alternative for Curve.approx, and may replace it in future.

---

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.

setProp(*p=None*)
Create or destroy the property array for the Formex.

A property array is a rank-1 integer array with dimension equal to the number of elements in the Formex (first dimension of data). You can specify a single value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

If a value None is given, the properties are removed from the Formex.

write (*fil*, *sep=' '*, *mode='w'*)
Write a Geometry to a .pgf file.

If fil is a string, a file with that name is opened. Else fil should be an open file. The Geometry is then written to that file in a native format, using sep as separator between the coordinates. If fil is a string, the file is closed prior to returning.

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 pecisely *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 curve.**Spiral** (*turns=2.0*, *nparts=100*, *rfunc=None*)
A class representing a spiral curve.

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).

addNoise (**args*, ***kargs*)
Apply 'addNoise' transformation to the Geometry object.

See `coords.Coords.addNoise()` for details.

affine (**args*, ***kargs*)
Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

align (**args*, ***kargs*)
Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

bump (**args*, ***kargs*)
Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

bump1 (**args*, ***kargs*)
Apply 'bump1' transformation to the Geometry object.

See `coords.Coords.bump1()` for details.

bump2 (**args*, ***kargs*)
Apply 'bump2' transformation to the Geometry object.

See `coords.Coords.bump2()` for details.

centered (**args*, ***kargs*)
Apply 'centered' transformation to the Geometry object.

See `coords.Coords.centered()` for details.

cylindrical (**args*, ***kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See `coords.Coords.cylindrical()` for details.

egg (**args*, ***kargs*)
Apply 'egg' transformation to the Geometry object.

See `coords.Coords.egg()` for details.

flare (**args*, ***kargs*)
Apply 'flare' transformation to the Geometry object.

See `coords.Coords.flare()` for details.

hyperCylindrical (**args*, ***kargs*)
Apply 'hyperCylindrical' transformation to the Geometry object.

See `coords.Coords.hyperCylindrical()` for details.

isopar (**args*, ***kargs*)
Apply 'isopar' transformation to the Geometry object.

See `coords.Coords.isopar()` for details.

map (**args*, ***kargs*)
Apply 'map' transformation to the Geometry object.

See `coords.Coords.map()` for details.

map1 (**args*, ***kargs*)
Apply 'map1' transformation to the Geometry object.

See `coords.Coords.map1()` for details.

mapd (**args*, ***kargs*)
Apply 'mapd' transformation to the Geometry object.

See `coords.Coords.mapd()` for details.

position (**args*, ***kargs*)
Apply 'position' transformation to the Geometry object.

See `coords.Coords.position()` for details.

projectOnCylinder (**args*, ***kargs*)
Apply 'projectOnCylinder' transformation to the Geometry object.

See `coords.Coords.projectOnCylinder()` for details.

projectOnPlane (**args*, ***kargs*)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (**args*, ***kargs*)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

reflect (**args*, ***kargs*)
Apply 'reflect' transformation to the Geometry object.

See `coords.Coords.reflect()` for details.

replace (**args*, ***kargs*)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

rollAxes (**args*, ***kargs*)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

rot (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

rotate (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

scale (**args*, ***kargs*)
Apply 'scale' transformation to the Geometry object.

See `coords.Coords.scale()` for details.

shear (**args*, ***kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

spherical (**args*, ***kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args*, ***kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

swapAxes (**args*, ***kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

toCylindrical (**args*, ***kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

toSpherical (**args*, ***kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

transformCS (**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

translate (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

trl (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

pointsAt (*t*, *normalized=False*, *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 normalized is True, the parameter values are give in a normalized space where 0 is the start of the curve and 1 is the end.

If return_position is True, also returns the part numbers on which the point are lying and the local parameter values.

nelems ()
Return the number of elements in the Geometry.

This method should be re-implemented by the derived classes. For the (empty) Geometry class it always returns 0.

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.

toProp (*prop*)

Converts the argument into a legal set of properties for the object.

The conversion involves resizing the argument to a 1D array of length self.nelems(), and converting the data type to integer.

getCoords ()

Get the coords data.

Returns the full array of coordinates stored in the Geometry object. Note that subclasses may store more points in this array than are used to define the geometry.

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 all 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.

approx (*ndiv=None*, *ntot=None*)

Return a PolyLine approximation of the curve

Parameters:

- *ndiv*: int: number of straight segments to use over each part of the curve. This is only used if
- *ntot*: int: number of straight segments to use over the total length of the curve.

Returns a PolyLine approximation for the curve. *C.approx(ndiv=n)* returns an approximation with *ndiv* segments over each part of the curve. This may results in segments with very different lengths. *C.approx(ntot=n)* returns an approximation with *ntot* segments over the total length of the curve. This produces more equally sized segments, but the internal end points of the curve parts may not be on the approximating Polyline.

level ()

Return the dimensionality of the Geometry, or -1 if unknown

frenet (*ndiv=None*, *ntot=None*, *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* and *ntot*. 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 binormial 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

copy()

Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

splitProp(*prop=None*)

Partition a Geometry (Formex/Mesh) according to the values in prop.

Parameters:

- *prop*: an int array with length self.nelems(), or None. If None, the *prop* attribute of the Geometry is used.

Returns a list of Geometry objects of the same type as the input. Each object 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 their *prop* value.

It *prop* is None and the the object has no *prop* attribute, an empty list is returned.

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.

approximate (*nseg*, *equidistant=True*, *npre=100*)
Approximate a Curve with a PolyLine of n segments

Parameters:

- *nseg*: number of straight segments of the resulting PolyLine
- *equidistant*: if True (default) the points are spaced almost equidistantly over the curve. If False, the points are spread equally over the parameter space.
- *npre*: 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.

Note: This is an alternative for Curve.approx, and may replace it in future.

---

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.

setProp (*p=None*)
Create or destroy the property array for the Formex.

A property array is a rank-1 integer array with dimension equal to the number of elements in the Formex (first dimension of data). You can specify a single value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

If a value None is given, the properties are removed from the Formex.

write (*fil*, *sep=' '*, *mode='w'*)
Write a Geometry to a .pgf file.

If fil is a string, a file with that name is opened. Else fil should be an open file. The Geometry is then written to that file in a native format, using sep as separator between the coordinates. If fil is a string, the file is closed prior to returning.

`Functions defined in module curve`

`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.

`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.

`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

6.4.6 `datareader` — Numerical data reader

`Classes defined in module datareader`

`Functions defined in module datareader`

`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']`

`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 `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 dxf`

class 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 dxf

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()`.

`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'.

`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	`plugins.curve.Arc`
Line	`plugins.curve.Line`
Polyline	`plugins.curve.PolyLine`

No object is returned for the *Vertex* function: they define the vertices of a PolyLine.

`dxf.`**`collectByType`**(*entities*)

Collect the dxf entities by type.

`dxf.`**`toLines`**(*coll*, *chordal=0.01*, *arcdiv=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 arcdiv parameters are passed to `Arc.approx()` to set the accuracy for the approximation of the Arc by line segments.

`dxf.`**`exportDXF`**(*filename*, *F*)

Export a Formex to a DXF file

Currently, only plex-2 Formices can be exported to DXF.

`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.

`dxf.`**`exportDxfText`**(*filename*, *parts*)
Export a set of dxf entities to a .dxftext file.

6.4.8 `export` — Classes and functions for exporting geometry in various formats.

`Classes defined in module export`

class `export.`**`ObjFile`**(*filename*)
Export a mesh in OBJ format.

This class exports a mesh in Wavefront OBJ format (see http://en.wikipedia.org/wiki/Wavefront_.obj_file).

Usage:

```
fil = ObjFile(PATH_TO_OBJFILE)
fil.write(MESH)
fil.close()
```

`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.

`Functions defined in module export`

6.4.9 `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 fe`

class `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 maximum plexitude of the model.

splitElems(*elems*)
Splits a set of element numbers over the element groups.

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 set in group

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.

It also provide the global and group element numbers, since they had to be calculated anyway.

renumber(*old=None*, *new=None*)
Renumber a set of nodes.

old and new are equally sized lists with unique node numbers, each smaller that 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 cooresponding with the old ones.

addNoise(**args*, ***kargs*)
Apply 'addNoise' transformation to the Geometry object.

See `coords.Coords.addNoise()` for details.

affine(**args*, ***kargs*)
Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

align(**args*, ***kargs*)
Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

bump(**args*, ***kargs*)
Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

bump1(**args*, ***kargs*)
Apply 'bump1' transformation to the Geometry object.

See coords.Coords.bump1() for details.

bump2 (**args*, ***kargs*)
Apply 'bump2' transformation to the Geometry object.

See coords.Coords.bump2() for details.

centered (**args*, ***kargs*)
Apply 'centered' transformation to the Geometry object.

See coords.Coords.centered() for details.

cylindrical (**args*, ***kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See coords.Coords.cylindrical() for details.

egg (**args*, ***kargs*)
Apply 'egg' transformation to the Geometry object.

See coords.Coords.egg() for details.

flare (**args*, ***kargs*)
Apply 'flare' transformation to the Geometry object.

See coords.Coords.flare() for details.

hyperCylindrical (**args*, ***kargs*)
Apply 'hyperCylindrical' transformation to the Geometry object.

See coords.Coords.hyperCylindrical() for details.

isopar (**args*, ***kargs*)
Apply 'isopar' transformation to the Geometry object.

See coords.Coords.isopar() for details.

map (**args*, ***kargs*)
Apply 'map' transformation to the Geometry object.

See coords.Coords.map() for details.

map1 (**args*, ***kargs*)
Apply 'map1' transformation to the Geometry object.

See coords.Coords.map1() for details.

mapd (**args*, ***kargs*)
Apply 'mapd' transformation to the Geometry object.

See coords.Coords.mapd() for details.

position (**args*, ***kargs*)
Apply 'position' transformation to the Geometry object.

See coords.Coords.position() for details.

projectOnCylinder (**args*, ***kargs*)
Apply 'projectOnCylinder' transformation to the Geometry object.

See coords.Coords.projectOnCylinder() for details.

projectOnPlane (**args*, ***kargs*)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (**args*, ***kargs*)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

reflect (**args*, ***kargs*)
Apply 'reflect' transformation to the Geometry object.

See `coords.Coords.reflect()` for details.

replace (**args*, ***kargs*)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

rollAxes (**args*, ***kargs*)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

rot (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

rotate (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

scale (**args*, ***kargs*)
Apply 'scale' transformation to the Geometry object.

See `coords.Coords.scale()` for details.

shear (**args*, ***kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

spherical (**args*, ***kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args*, ***kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

swapAxes (**args*, ***kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

toCylindrical (**args*, ***kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

toSpherical(**args*, ***kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

transformCS(**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

translate(**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

trl(**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

setProp(*prop=None*, *blocks=None*)
Create or destroy the property array for the Geometry.

A property array is a rank-1 integer array with dimension 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. They play an import role when creating new geometry: new elements inherit the property number of their parent element. Properties are also preserved on most geometrical 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*: a single integer value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

  The special value 'range' will set the property numbers equal to the element number.

  A value None (default) removes the properties from the Geometry.

- *blocks*: a single integer value or a list/array of integer values. If the number of passed values is less than the length of *prop*, they wil be repeated. If you give more, they will be ignored. Every prop will be repeated the corresponding number of times specified in blocks.

getCoords()
Get the coords data.

Returns the full array of coordinates stored in the Geometry object. Note that subclasses may store more points in this array than are used to define the geometry.

level()
Return the dimensionality of the Geometry, or -1 if unknown

copy()
Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

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.

write (*fil*, *sep=' '*, *mode='w'*)
Write a Geometry to a .pgf file.

If fil is a string, a file with that name is opened. Else fil should be an open file. The Geometry is then written to that file in a native format, using sep as separator between the coordinates. If fil is a string, the file is closed prior to returning.

class 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 used 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 accomodate better to versatile applications.

nelems ()
Return the number of elements in the Geometry.

This method should be re-implemented by the derived classes. For the (empty) Geometry class it always returns 0.

addNoise (**args*, ***kargs*)
Apply 'addNoise' transformation to the Geometry object.

See `coords.Coords.addNoise()` for details.

affine (**args*, ***kargs*)
Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

align (**args*, ***kargs*)
Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

bump (**args*, ***kargs*)
Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

bump1 (**args*, ***kargs*)
Apply 'bump1' transformation to the Geometry object.

See `coords.Coords.bump1()` for details.

bump2 (**args*, ***kargs*)
Apply 'bump2' transformation to the Geometry object.

See `coords.Coords.bump2()` for details.

centered (**args, **kargs*)
Apply 'centered' transformation to the Geometry object.

See `coords.Coords.centered()` for details.

cylindrical (**args, **kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See `coords.Coords.cylindrical()` for details.

egg (**args, **kargs*)
Apply 'egg' transformation to the Geometry object.

See `coords.Coords.egg()` for details.

flare (**args, **kargs*)
Apply 'flare' transformation to the Geometry object.

See `coords.Coords.flare()` for details.

hyperCylindrical (**args, **kargs*)
Apply 'hyperCylindrical' transformation to the Geometry object.

See `coords.Coords.hyperCylindrical()` for details.

isopar (**args, **kargs*)
Apply 'isopar' transformation to the Geometry object.

See `coords.Coords.isopar()` for details.

map (**args, **kargs*)
Apply 'map' transformation to the Geometry object.

See `coords.Coords.map()` for details.

map1 (**args, **kargs*)
Apply 'map1' transformation to the Geometry object.

See `coords.Coords.map1()` for details.

mapd (**args, **kargs*)
Apply 'mapd' transformation to the Geometry object.

See `coords.Coords.mapd()` for details.

position (**args, **kargs*)
Apply 'position' transformation to the Geometry object.

See `coords.Coords.position()` for details.

projectOnCylinder (**args, **kargs*)
Apply 'projectOnCylinder' transformation to the Geometry object.

See `coords.Coords.projectOnCylinder()` for details.

projectOnPlane (**args, **kargs*)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (**args*, ***kargs*)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

reflect (**args*, ***kargs*)
Apply 'reflect' transformation to the Geometry object.

See `coords.Coords.reflect()` for details.

replace (**args*, ***kargs*)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

rollAxes (**args*, ***kargs*)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

rot (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

rotate (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

scale (**args*, ***kargs*)
Apply 'scale' transformation to the Geometry object.

See `coords.Coords.scale()` for details.

shear (**args*, ***kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

spherical (**args*, ***kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args*, ***kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

swapAxes (**args*, ***kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

toCylindrical (**args*, ***kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

toSpherical (**args*, ***kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

transformCS(**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

translate(**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

trl(**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

setProp(*prop=None*, *blocks=None*)
Create or destroy the property array for the Geometry.

A property array is a rank-1 integer array with dimension 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. They play an import role when creating new geometry: new elements inherit the property number of their parent element. Properties are also preserved on most geometrical 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*: a single integer value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

  The special value 'range' will set the property numbers equal to the element number.

  A value None (default) removes the properties from the Geometry.

- *blocks*: a single integer value or a list/array of integer values. If the number of passed values is less than the length of *prop*, they wil be repeated. If you give more, they will be ignored. Every prop will be repeated the corresponding number of times specified in blocks.

getCoords()
Get the coords data.

Returns the full array of coordinates stored in the Geometry object. Note that subclasses may store more points in this array than are used to define the geometry.

level()
Return the dimensionality of the Geometry, or -1 if unknown

copy()
Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

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.

write (*fil*, *sep=' '*, *mode='w'*)
Write a Geometry to a .pgf file.

If fil is a string, a file with that name is opened. Else fil should be an open file. The Geometry is then written to that file in a native format, using sep as separator between the coordinates. If fil is a string, the file is closed prior to returning.

Functions defined in module fe

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.

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 `fe_abq` — Exporting 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 fe_abq

class fe_abq.**Output** (*kind=None*, *keys=None*, *set=None*, *type='FIELD'*, *variable='PRESELECT'*, *extra=''*, ***options*)
A request for output to .odb and history.

Parameters:

- *type*: 'FIELD' or 'HISTORY'
- *kind*: None, 'NODE', or 'ELEMENT' (first character suffices)

- *extra*: an extra string to be added to the command line. This allows to add Abaqus options not handled by this constructor. The string will be appended to the command line preceded by a comma.

For kind==' ':

- *variable*: 'ALL', 'PRESELECT' or ''

For kind=='NODE' or 'ELEMENT':

- *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'

fmt()

Format an output request.

Return a string with the formatted output command.

update(*data={}*, ***kargs*)

Add a dictionary to the Dict object.

The data can be a dict or Dict type object.

get(*key*, *default*)

Return the value for key or a default.

This is the equivalent of the dict get method, except that it returns only the default value if the key was not found in self, and there is no _default_ method or it raised a KeyError.

setdefault(*key*, *default*)

Replaces the setdefault function of a normal dictionary.

This is the same as the get method, except that it also sets the default value if get found a KeyError.

class 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.

update(*data={}*, ***kargs*)

Add a dictionary to the Dict object.

The data can be a dict or Dict type object.

get (*key*, *default*)
Return the value for key or a default.

This is the equivalent of the dict get method, except that it returns only the default value if the key was not found in self, and there is no _default_ method or it raised a KeyError.

setdefault (*key*, *default*)
Replaces the setdefault function of a normal dictionary.

This is the same as the get method, except that it also sets the default value if get found a KeyError.

class fe_abq.**Interaction** (*name=None*, *cross_section=1*, *friction=0.0*, *surfacebehavior=None*, *noseparation=False*, *pressureoverclosure=None*)
A Dict for setting surface interactions pressureoverclosure is an array = ['hard'/'soft','linear'/'nonlinear'/'exponential'/'tabular'/.., value1,value2,value3,...] Leave empty for default hard contact 'hard' will set penalty contact, either 'linear' or 'nonlinear' 'soft' will set soft pressure-overclosure, combine with 'linear'/'exponential'/'tabular'/'scale factor' for needed values on dataline: see abaqus keyword manual

update (*data={}*, ***kargs*)
Add a dictionary to the Dict object.

The data can be a dict or Dict type object.

get (*key*, *default*)
Return the value for key or a default.

This is the equivalent of the dict get method, except that it returns only the default value if the key was not found in self, and there is no _default_ method or it raised a KeyError.

setdefault (*key*, *default*)
Replaces the setdefault function of a normal dictionary.

This is the same as the get method, except that it also sets the default value if get found a KeyError.

class fe_abq.**AbqData** (*model*, *prop*, *nprop=None*, *eprop=None*, *steps*=[ ], *res*=[ ], *out*=[ ], *bound=None*)
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 (deprecated: set inside Step).
- *out* : a list of *Output* instances (deprecated: set inside Step).
- *bound* : a tag or alist of the initial boundary conditions. The default is to apply ALL boundary conditions initially. Specify a (possibly non-existing) tag to override the default.

write (*jobname=None*, *group_by_eset=True*, *group_by_group=False*, *header=''*, *create_part=False*)
Write an Abaqus input file.

- *jobname* : the name of the inputfile, with or without '.inp' extension. If None is specified, the output is written to sys.stdout An extra header text may be specified.
- *create_part* : if True, the model will be created as an Abaqus Part, followed by an assembly of that part.

`Functions defined in module fe_abq`

`fe_abq.`**`abqInputNames`**(*job*)
Returns corresponding Abq jobname and input filename.

job can be either a jobname or input file name, with or without directory part, with or without extension (.inp)

The Abq jobname is the basename without the extension. The abq filename is the abspath of the job with extension '.inp'

`fe_abq.`**`nsetName`**(*p*)
Determine the name for writing a node set property.

`fe_abq.`**`esetName`**(*p*)
Determine the name for writing an element set property.

`fe_abq.`**`fmtCmd`**(*cmd='*'*)
Format a command.

`fe_abq.`**`fmtData1D`**(*data*, *npl=8*, *sep=', '*, *linesep='\n'*)
Format numerical data in lines with maximum npl items.

data is a numeric array. The array is flattened and then the data are formatted in lines with maximum npl items, separated by sep. Lines are separated by linesep.

`fe_abq.`**`fmtData`**(*data*, *npl=8*, *sep=', '*, *linesep='\n'*)
Format numerical data in lines with maximum npl items.

data is a numeric array, which is coerced to be a 2D array, either by adding a first axis or by collapsing the first ndim-1 axies. Then the data are formatted in lines with maximum npl items, separated by sep. Lines are separated by linesep.

`fe_abq.`**`fmtOptions`**(*options*)
Format the options of an Abaqus command line.

- *options*: a dict with ABAQUS command keywords and values. If the keyword does not take any value, the value in the dict should be an empty string.

Returns a comma-separated string of 'keyword' or 'keyword=value' fields. The string includes an initial comma.

`fe_abq.`**`fmtHeading`**(*text=''*)
Format the heading of the Abaqus input file.

`fe_abq.`**`fmtPart`**(*name='Part-1'*)
Start a new Part.

`fe_abq.`**`fmtMaterial`**(*mat*)
Write a material section.

mat is the property dict of the material. The following keys are recognized and output accordingly:

- *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'. Default is 'LINEAR'. Defines the elastic behavior class of the material. The requirements for the other keys depend on this type. The fields labeled (opt) are optional.
- 'LINEAR':
  - young_modulus
  - shear_modulus
  - (opt) poisson_ratio: it is calculated if None
- 'HYERELASTIC':

  required:
  - model: one of 'ogden', 'polynomial' or 'reduced polynomial'
  - constants: list of all parameter required for the model (see Abaqus documentation)

  optional:
  - order: order of the model. If blank will be automatically calculated from the len of the constants list

    example:

```
intimaMat = {
  'name': 'intima',
  'density': 0.1, # Not Used, but Abaqus does not like a material without
  'elasticity':'hyperelastic',
  'type':'reduced polynomial',
  'constants': [6.79E-03, 5.40E-01, -1.11, 10.65, -7.27, 1.63, 0.0, 0.0, 0.0
}
```

- 'ANISOTROPIC HYPERELASTIC':
- 'USER':

fe_abq.**fmtTransform**(*setname*, *csys*)
Write transform command for the given set.

- *setname* is the name of a node set
- *csys* is a CoordSystem.

fe_abq.**fmtFrameSection**(*el*, *setname*)
Write a frame section for the named element set.

Recognized data fields in the property record:

- sectiontype GENERAL:
  - cross_section
  - moment_inertia_11
  - moment_inertia_12
  - moment_inertia_22
  - torsional_constant
- sectiontype CIRC:

- –radius

- •sectiontype RECT:
  - –width
  - –height
- •all sectiontypes:
  - –young_modulus
  - –shear_modulus
- •optional:
  - –density: density of the material
  - –yield_stress: yield stress of the material
  - –orientation: a vector specifying the direction cosines of the 1 axis

fe_abq.**fmtGeneralBeamSection**(*el*, *setname*)

Write a general beam section for the named element set.

To specify a beam section when numerical integration over the section is not required.

Recognized data fields in the property record:

- •sectiontype GENERAL:
  - –cross_section
  - –moment_inertia_11
  - –moment_inertia_12
  - –moment_inertia_22
  - –torsional_constant
- •sectiontype CIRC:
  - –radius
- •sectiontype RECT:
  - –width, height
- •all sectiontypes:
  - –young_modulus
  - –shear_modulus or poisson_ration
- •optional:
  - –density: density of the material (required in Abaqus/Explicit)

fe_abq.**fmtBeamSection**(*el*, *setname*)

Write a beam section for the named element set.

To specify a beam section when numerical integration over the section is required.

Recognized data fields in the property record:

- •all sectiontypes: material

- sectiontype GENERAL:
  - cross_section
  - moment_inertia_11
  - moment_inertia_12
  - moment_inertia_22
  - torsional_constant
- sectiontype CIRC:
  - radius
  - intpoints1 (number of integration points in the first direction) optional
  - intpoints2 (number of integration points in the second direction) optional
- sectiontype RECT:
  - width, height
  - intpoints1 (number of integration points in the first direction) optional
  - intpoints2 (number of integration points in the second direction) optional

fe_abq.**fmtConnectorSection**(*el*, *setname*)

Write a connector section.

Required:

- *sectiontype*: JOIN, HINGE, ...

Optional data:

- *behavior* : connector behavior name
- *orient* : connector orientation

fe_abq.**fmtConnectorBehavior**(*prop*)

Write a connector behavior. Implemented: Elasticity, Stop Examples: Elasticity P.Prop(name='connbehavior1',ConnectorBehavior='',Elasticity=dict(component=[1,2,3,4,5,6],value=[1,1,1,1,1,1])) Stop: P.Prop(name='connbehavior3',ConnectorBehavior='',Stop=dict(component=[1,2,3,4,5,6],lowerlimit=[1,1,1,1,1,1], upperlimit=[2, 2, 2, 2,2,2]))

fe_abq.**fmtSpring**(*el*, *setname*)

Write a spring of type spring.

Optional data:

- *springstiffness* : spring stiffness (force (S11) per relative displacement (E11))

fe_abq.**fmtDashpot**(*el*, *setname*)

Write a dashpot.

Optional data:

- *dashpotcoefficient* : dashpot coefficient (force (S11) per relative velocity (ER11, only produced in Standard))

fe_abq.**fmtSolidSection**(*el*, *setname*, *matname*)

Format the SOLID SECTION keyword.

Required:

- setname
- matname

Optional:

- orientation
- controls

controls is a dict with name, options and data keys. Options is a string which is added as is to the command. Data are added below the command. All other items besides name, options and data are formatted as extra command options.

Example:

```
P.elemProp(set='STENT',eltype='C3D8R',section=ElemSection(section=stentSec,material
```

fe_abq.**fmtShellSection**(*el*, *setname*, *matname*)
Format the shell SHELL SECTION keyword.

Required:

- setname
- matname

Optional:

- transverseshearstiffness
- offset (for contact surface SPOS or 0.5, SNEG or -0.5)

fe_abq.**fmtSurface**(*prop*)
Format the surface definitions.

Required:

- set: the elements/nodes in the surface, either numbers or a set name.
- name: the surface name
- surftype: 'ELEMENT' or 'NODE'
- label: face or edge identifier (only required for surftype = 'ELEMENT')

This label can be a string, or a list of strings. This allows to use different identifiers for the different elements in the surface. Thus:

```
Prop(name='mysurf',set=[0,1,2,6],surftype='element',label=['S1','S2','S1','S3')
```

will get exported to Abaqus as:

```
*SURFACE, NAME=mysurf, TYPE=element
1, S1
2, S2,
3, S1
7, S3
```

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='')
```

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

fe_abq.**fmtGeneralContact**(*prop*)
Format the general contact.

Only implemented on model level

Required:

- interaction: interaction properties: name or Dict

Optional:

- Exclusions (exl)
- Extra (extra). Example

```
extra = "*CONTACT CONTROLS ASSIGNMENT, TYPE=SCALE PENALTY\n, , 1.e3\n"
```

Example:

```
>>> P.Prop(generalinteraction=Interaction(name ='contactprop1'),exl =[
['surf11', 'surf12'],['surf21',surf22]])
```

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'
```

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')
```

fe_abq.**fmtInitialConditions**(*prop*)

Format initial conditions

Required:

-type -nodes -data

Example:

```
P.Prop(initialcondition='', nodes ='Nall', type = 'TEMPERATURE', data = 37.)
```

fe_abq.**fmtOrientation**(*prop*)

Format the orientation.

Optional:

- definition
- system: coordinate system
- a: a first point
- b: a second point

fe_abq.**fmtEquation**(*prop*)

Format multi-point constraint using an equation

Required:

- equation

Equation should be a list, which contains the different terms of the equation. Each term is again a list with three values:

- First value: node number
- Second value: degree of freedom
- Third value: coefficient

Example:

```
P.nodeProp(equation=[[209,1,1],[32,1,-1]])
```

This forces the displacement in Y-direction of nodes 209 and 32 to be equal.

fe_abq.**fmtMass** (*prop*)
Format mass

Required:

- mass : mass magnitude
- set : name of the element set on which mass is applied

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

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.

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.

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.

fe_abq.**writeSection** (*fil*, *prop*)
Write an element section.

prop is a an element property record with a section and eltype attribute

fe_abq.**writeDisplacements** (*fil*, *prop*, *dtype='DISPLACEMENT'*)
Write boundary conditions of type BOUNDARY, TYPE=DISPLACEMENT

prop is a list of node property records that should be scanned for displ attributes to write.

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 also remove initial conditions!

fe_abq.**writeCloads** (*fil*, *prop*)
Write cloads.

prop is a list of node property records that should be scanned for displ attributes to write.

By default, the loads are applied as new values in the current step. The user can set op='MOD' to add the loads to already existing ones.

fe_abq.**writeCommaList** (*fil*, **args*)
Write a list of values comma-separated to fil

fe_abq.**writeDloads** (*fil*, *prop*)
Write Dloads.

prop is a list of elem property records having an attribute dload.

By default, the loads are applied as new values in the current step. The user can set op='MOD' to add the loads to already existing ones.

fe_abq.**writeDsloads** (*fil*, *prop*)
Write Dsloads.

prop is a list property records having an attribute dsload

By default, the loads are applied as new values in the current step. The user can set op='MOD' to add the loads to already existing ones.

fe_abq.**writeNodeOutput** (*fil*, *kind*, *keys*, *set='Nall'*)
Write a request for nodal result output to the .odb 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

fe_abq.**writeNodeResult** (*fil*, *kind*, *keys*, *set='Nall'*, *output='FILE'*, *freq=1*, *globalaxes=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.'

fe_abq.**writeElemOutput** (*fil*, *kind*, *keys*, *set='Eall'*)
Write a request for element output to the .odb 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

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

fe_abq.**writeFileOutput** (*fil*, *resfreq=1*, *timemarks=False*)

Write the FILE OUTPUT command for Abaqus/Explicit

fe_abq.**exportMesh** (*filename*, *mesh*, *eltype=None*, *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. 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 oerride the value stored in the mesh. This should be used to set a correct Abaqus element type matchin the mesh.

6.4.11 `fe_post` — A class for holding results from Finite Element simulations.

`Classes defined in module fe_post`

class fe_post.**FeResult** (*name='__FePost__'*, *datasize={'S': 6, 'U': 3, 'COORD': 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 interprete 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(**arg*, ***kargs*)

A do nothing function to stand in for as yet undefined functions.

TotalEnergies(**arg*, ***kargs*)

A do nothing function to stand in for as yet undefined functions.

OutputRequest(**arg*, ***kargs*)

A do nothing function to stand in for as yet undefined functions.

Coordinates(**arg*, ***kargs*)

A do nothing function to stand in for as yet undefined functions.

Displacements(**arg*, ***kargs*)

A do nothing function to stand in for as yet undefined functions.

Unknown (**arg*, ***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.

`Functions defined in module fe_post`

6.4.12 flavia —

3. 2010 Benedict Verhegghe.

`Classes defined in module flavia`

`Functions defined in module flavia`

flavia.**readMesh** (*fn*)
Read a flavia mesh file.

Returns a list of Meshes if succesful.

flavia.**readCoords** (*fil*, *ndim*)
Read a set of coordinates from a flavia file

flavia.**readElems** (*fil*, *nplex*)
Read a set of coordinates from a flavia file

`flavia.`**`readResults`**(*fn*, *nnodes*, *ndim*)
Read a flavia results file for an ndim mesh.

`flavia.`**`readResult`**(*fil*, *nvalues*, *nres*)
Read a set of results from a flavia file

`flavia.`**`createFeResult`**(*model*, *results*)
Create an FeResult from meshes and results

`flavia.`**`readFlavia`**(*meshfile*, *resfile*)
Read flavia results files

Currently we only read matching pairs of meshfile,resfile files.

6.4.13 `inertia` — inertia.py

Compute inertia related quantities of a Formex. This comprises: center of gravity, inertia tensor, principal axes

Currently, these functions work on arrays of nodes, not on Formices! Use func(F,f) to operate on a Formex F.

`Classes defined in module inertia`

`Functions defined in module inertia`

`inertia.`**`centroids`**(*X*)
Compute the centroids of the points of a set of elements.

X (nelems,nplex,3)

`inertia.`**`center`**(*X*, *mass=None*)
Compute the center of gravity of an array of points.

mass is an optional array of masses to be atributed to the points. The default is to attribute a mass=1 to all points.

If you also need the inertia tensor, it is more efficient to use the inertia() function.

`inertia.`**`inertia`**(*X*, *mass=None*)
Compute the inertia tensor of an array of points.

mass is an optional array of masses to be atributed to the points. The default is to attribute a mass=1 to all points.

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, Ixy, Ixz, Iyz

`inertia.`**`principal`**(*inertia*, *sort=False*, *right_handed=False*)
Returns the principal values and axes of the inertia tensor.

If sort is True, they are sorted (maximum comes first). If right_handed is True, the axes define a right-handed coordinate system.

6.4.14 `isopar` — Isoparametric transformations

`Classes defined in module isopar`

class `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 isopar`

`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* colums.

`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 * sum(n) - 4. For ndim=3 we have npoints = 2 * sum(nx*ny+ny*nz+nz*nx) - 4 * 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.

`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 `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/

`Classes defined in module isosurface`

`Functions defined in module isosurface`

`isosurface.`**`isosurface`**(*data*, *level*, *nproc=-1*)

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).

6.4.16 `lima` — Lindenmayer Systems

`Classes defined in module lima`

class `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 lima`

`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 neu_exp — Gambit neutral file exporter.

This module contains some functions to export pyFormex mesh models to Gambit neutral files.

`Classes defined in module neu_exp`

`Functions defined in module neu_exp`

`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).

`neu_exp.`**writeNodes** (*fil*, *nodes*)
Write nodal coordinates.

`neu_exp.`**writeElems** (*fil*, *elems*)
Write element connectivity.

`neu_exp.`**writeGroup** (*fil*, *elems*)
Write group of elements.

`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: echo 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, BorderFace array for the surface S can be obtained from:

```
bf = M.matchFaces(S)[1]
```

See also http://combust.hit.edu.cn:8080/fluent/Gambit13_help/modeling_guide/mg0b.htm#mg0b01 for the description of the neu file syntax.

`neu_exp.`**`read_tetgen`**(*filename*)
Read a tetgen tetraeder model.

filename is the base of the path of the input files. For a filename 'proj', nodes are expected in 'proj.1.node' and elems are in file 'proj.1.ele'.

`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 `nurbs` — Using NURBS in pyFormex.

The `nurbs` module defines functions and classes to manipulate NURBS curves and surface in pyFormex.

`Classes defined in module nurbs`

class `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 ingle 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. It 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 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 knots = nctrl+degree+1 parameter values.

The knots vector should hold nknots values in ascending order. The values are only defined upon a multiplicative constant and will be normalized to set the last value to 1. Sensible default values are constructed automatically by calling `knotVector()`.

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.

bbox()
Return the bounding box of the NURBS curve.

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.

Returns (nu,3) shaped Coords with nu points at the specified parametric values.

derivs(*at*, *d=1*)
Returns the points and derivatives up to d at parameter values at

knotPoints()
Returns the points at the knot values.

The multiplicity of the knots is retained in the points set.

insertKnots(*u*)
Insert a set of knots in the 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.

decompose()
Decomposes a curve in subsequent Bezier curves.

Returns an equivalent unblended Nurbs.

removeKnots(*u*, *tol*)
Remove a knots in the 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.

approx (*ndiv=None*, *ntot=None*)
Return a PolyLine approximation of the Nurbs curve

If no *ntot* 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 *ntot* is given, a second approximation is computed with *ntot* straight segments of nearly equal length. The lengths are computed based on the first approximation with *ndiv* segments.

actor (***kargs*)
Graphical representation

class 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 knotVector() 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!

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.

actor (***kargs*)
Graphical representation

Functions defined in module nurbs

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 (slighly) different curves. Typical values are:

0.0: equally spaced (not recommended) 0.5: centripetal (default, recommended) 1.0: chord length (often used)

nurbs.**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*, *umax*: float: start and end value of the interval. Default interval is [0.0..1.0].

Returns:

A float array with 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.

Example:

```
>>> 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]
```

nurbs.**knotVector** (*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` sohuld be a multiple of `degree` plus 1.

Example:

```
>>> print knotVector(7,3)
[ 0.    0.    0.    0.    0.25  0.5   0.75  1.    1.    1.    1.  ]
>>> print knotVector(7,3,closed=True)
[ 0.   0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1. ]
>>> print knotVector(7,3,blended=False)
[ 0.  0.  0.  0.  1.  1.  1.  2.  2.  2.  2.]
```

`nurbs.`**`toCoords4`**(*x*)

Convert cartesian coordinates to homogeneous

***x*: `Coords`** Array with cartesian coordinates.

Returns a Coords4 object corresponding to the input cartesian coordinates.

`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, Casteljou

`nurbs.`**`deCasteljou`**(*P*, *u*)

Compute points on a Bezier curve using deCasteljou 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 deCasteljou 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.

nurbs.**curveToNurbs**(*B*)
Convert a BezierSpline to NurbsCurve

nurbs.**polylineToNurbs**(*B*)
Convert a PolyLine to NurbsCurve

nurbs.**frenet**(*d1*, *d2*, *d3=None*)
Returns the 3 Frenet vectors and the curvature.

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

6.4.19 `objects` — Selection of objects from the global dictionary.

This is a support module for other pyFormex plugins.

`Classes defined in module objects`

class 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 ODict 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 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 well reset the *prop* attribute of all selected objects to None.

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!

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 ODict with the currently selected objects in the order of the selection.names.

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.

Functions defined in module objects

objects.**draw_object_name** (*n*)
Draw the name of an object at its center.

objects.**draw_elem_numbers** (*n*)
Draw the numbers of an object's elements.

objects.**draw_nodes** (*n*)
Draw the nodes of an object.

objects.**draw_node_numbers** (*n*)
Draw the numbers of an object's nodes.

objects.**draw_free_edges** (*n*)
Draw the feature edges of an object.

objects.**draw_bbox** (*n*)
Draw the bbox of an object.

6.4.20 partition — Partitioning tools

Classes defined in module partition

Functions defined in module partition

partition.**prepare** (*V*)
Prepare the surface for slicing operation.

partition.**colorCut** (*F, P, N, prop*)
Color a Formex in two by a plane (P,N)

partition.**splitProp** (*F, name*)
Partition a Formex according to its prop values.

Returns a dict with the partitions, named like name-prop and exports these named Formex instances. It the Formex has no props, the whole Formex is given the name.

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 `plot2d` — plot2d.py

Generic 2D plotting functions for pyFormex.

`Classes defined in module plot2d`

`Functions defined in module plot2d`

`plot2d.`**`showStepPlot`**(*x*, *y*, *label=''*, *title=None*, *plot2d_system=None*)
Show a step plot of x,y data.

`plot2d.`**`showHistogram`**(*x*, *y*, *label*, *cumulative=False*, *plot2d_system=None*)
Show a histogram of x,y data.

`plot2d.`**`createHistogram`**(*data*, *cumulative=False*, ***kargs*)
Create a histogram from data

6.4.22 `polygon` — Polygonal facets.

`Classes defined in module polygon`

class `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.

`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.

addNoise(**args*, ***kargs*)
Apply 'addNoise' transformation to the Geometry object.

See `coords.Coords.addNoise()` for details.

affine(**args*, ***kargs*)
Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

align(**args*, ***kargs*)
Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

bump(**args*, ***kargs*)
Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

bump1(**args*, ***kargs*)
Apply 'bump1' transformation to the Geometry object.

See `coords.Coords.bump1()` for details.

bump2(**args*, ***kargs*)
Apply 'bump2' transformation to the Geometry object.

See `coords.Coords.bump2()` for details.

centered(**args*, ***kargs*)
Apply 'centered' transformation to the Geometry object.

See `coords.Coords.centered()` for details.

cylindrical(**args*, ***kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See `coords.Coords.cylindrical()` for details.

egg (**args*, ***kargs*)
Apply 'egg' transformation to the Geometry object.

See `coords.Coords.egg()` for details.

flare (**args*, ***kargs*)
Apply 'flare' transformation to the Geometry object.

See `coords.Coords.flare()` for details.

hyperCylindrical (**args*, ***kargs*)
Apply 'hyperCylindrical' transformation to the Geometry object.

See `coords.Coords.hyperCylindrical()` for details.

isopar (**args*, ***kargs*)
Apply 'isopar' transformation to the Geometry object.

See `coords.Coords.isopar()` for details.

map (**args*, ***kargs*)
Apply 'map' transformation to the Geometry object.

See `coords.Coords.map()` for details.

map1 (**args*, ***kargs*)
Apply 'map1' transformation to the Geometry object.

See `coords.Coords.map1()` for details.

mapd (**args*, ***kargs*)
Apply 'mapd' transformation to the Geometry object.

See `coords.Coords.mapd()` for details.

position (**args*, ***kargs*)
Apply 'position' transformation to the Geometry object.

See `coords.Coords.position()` for details.

projectOnCylinder (**args*, ***kargs*)
Apply 'projectOnCylinder' transformation to the Geometry object.

See `coords.Coords.projectOnCylinder()` for details.

projectOnPlane (**args*, ***kargs*)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (**args*, ***kargs*)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

reflect (**args*, ***kargs*)
Apply 'reflect' transformation to the Geometry object.

See `coords.Coords.reflect()` for details.

replace (**args*, ***kargs*)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

rollAxes (**args*, ***kargs*)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

rot (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

rotate (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

scale (**args*, ***kargs*)
Apply 'scale' transformation to the Geometry object.

See `coords.Coords.scale()` for details.

shear (**args*, ***kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

spherical (**args*, ***kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args*, ***kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

swapAxes (**args*, ***kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

toCylindrical (**args*, ***kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

toSpherical (**args*, ***kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

transformCS (**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

translate (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

trl (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

nelems()
Return the number of elements in the Geometry.

This method should be re-implemented by the derived classes. For the (empty) Geometry class it always returns 0.

setProp(*prop=None*, *blocks=None*)
Create or destroy the property array for the Geometry.

A property array is a rank-1 integer array with dimension 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. They play an import role when creating new geometry: new elements inherit the property number of their parent element. Properties are also preserved on most geometrical 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*: a single integer value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

  The special value 'range' will set the property numbers equal to the element number.

  A value None (default) removes the properties from the Geometry.

- *blocks*: a single integer value or a list/array of integer values. If the number of passed values is less than the length of *prop*, they wil be repeated. If you give more, they will be ignored. Every prop will be repeated the corresponding number of times specified in blocks.

getCoords()
Get the coords data.

Returns the full array of coordinates stored in the Geometry object. Note that subclasses may store more points in this array than are used to define the geometry.

level()
Return the dimensionality of the Geometry, or -1 if unknown

copy()
Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

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.

write(*fil*, *sep=' '*, *mode='w'*)
Write a Geometry to a .pgf file.

If fil is a string, a file with that name is opened. Else fil should be an open file. The Geometry is then written to that file in a native format, using sep as separator between the coordinates.

If fil is a string, the file is closed prior to returning.

`Functions defined in module polygon`

`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.

`polygon.`**`delaunay`**`(`*X*`)`

Return a Delaunay triangulation of the specified Coords.

While the Coords are 3d, only the first 2 components are used.

Returns a TriSurface with the Delaunay trinagulation in the x-y plane.

6.4.23 `polynomial` — Polynomials

This module defines the class Polynomial, representing a polynomial in n variables.

`Classes defined in module polynomial`

class `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 coeeficients 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 polynomial

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* colums.

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 postproc — Postprocessing functions

Postprocessing means collecting a geometrical model and computed values from a numerical simulation, and render the values on the domain.

Classes defined in module postproc

Functions defined in module postproc

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 `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 properties`

class `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.

`update`(*data={}*, ***kargs*)

Add a dictionary to the Dict object.

The data can be a dict or Dict type object.

`get`(*key*, *default*)

Return the value for key or a default.

This is the equivalent of the dict get method, except that it returns only the default value if the key was not found in self, and there is no _default_ method or it raised a KeyError.

`setdefault`(*key*, *default*)

Replaces the setdefault function of a normal dictionary.

This is the same as the get method, except that it also sets the default value if get found a KeyError.

class `properties.`**`MaterialDB`**(*data={}*)

A class for storing material properties.

`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.

update (*data={}*, ***kargs*)
Add a dictionary to the Dict object.

The data can be a dict or Dict type object.

get (*key*, *default*)
Return the value for key or a default.

This is the equivalent of the dict get method, except that it returns only the default value if the key was not found in self, and there is no _default_ method or it raised a KeyError.

setdefault (*key*, *default*)
Replaces the setdefault function of a normal dictionary.

This is the same as the get method, except that it also sets the default value if get found a KeyError.

class properties.**SectionDB** (*data={}*)
A class for storing section properties.

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.

update (*data={}*, ***kargs*)
Add a dictionary to the Dict object.

The data can be a dict or Dict type object.

get (*key*, *default*)
Return the value for key or a default.

This is the equivalent of the dict get method, except that it returns only the default value if the key was not found in self, and there is no _default_ method or it raised a KeyError.

setdefault (*key*, *default*)
Replaces the setdefault function of a normal dictionary.

This is the same as the get method, except that it also sets the default value if get found a KeyError.

class 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.

update (*data={}*, ***kargs*)
Add a dictionary to the Dict object.

The data can be a dict or Dict type object.

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.

get (*key*, *default*)
Return the value for key or a default.

This is the equivalent of the dict get method, except that it returns only the default value if the key was not found in self, and there is no _default_ method or it raised a KeyError.

setdefault (*key*, *default*)
Replaces the setdefault function of a normal dictionary.

This is the same as the get method, except that it also sets the default value if get found a KeyError.

class `properties.`**`ElemLoad`**(*label=None*, *value=None*, *dir=None*)
Distributed loading on an element.

`update`(*data={}*, ***kargs*)
Add a dictionary to the Dict object.

The data can be a dict or Dict type object.

`get`(*key*, *default*)
Return the value for key or a default.

This is the equivalent of the dict get method, except that it returns only the default value if the key was not found in self, and there is no _default_ method or it raised a KeyError.

`setdefault`(*key*, *default*)
Replaces the setdefault function of a normal dictionary.

This is the same as the get method, except that it also sets the default value if get found a KeyError.

class `properties.`**`EdgeLoad`**(*edge=-1*, *label=None*, *value=None*)
Distributed loading on an element edge.

`update`(*data={}*, ***kargs*)
Add a dictionary to the Dict object.

The data can be a dict or Dict type object.

`get`(*key*, *default*)
Return the value for key or a default.

This is the equivalent of the dict get method, except that it returns only the default value if the key was not found in self, and there is no _default_ method or it raised a KeyError.

`setdefault`(*key*, *default*)
Replaces the setdefault function of a normal dictionary.

This is the same as the get method, except that it also sets the default value if get found a KeyError.

class `properties.`**`CoordSystem`**(*csys*, *cdata*)
A class for storing coordinate systems.

class `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 `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.

update (*data={}*, ***kargs*)
Add a dictionary to the Dict object.

The data can be a dict or Dict type object.

get (*key*, *default*)
Return the value for key or a default.

This is the equivalent of the dict get method, except that it returns only the default value if the key was not found in self, and there is no _default_ method or it raised a KeyError.

setdefault (*key*, *default*)
Replaces the setdefault function of a normal dictionary.

This is the same as the get method, except that it also sets the default value if get found a KeyError.

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 '', '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*, *cload=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 str, a list of 6 codes (0/1), or a list of tuples (what??)
- 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 properties`

`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.

`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.

`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.

`properties.`**`checkArrayOrIdValue`**(*values*)
Check that a variable is an 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.

`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.

`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.

`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 `pyformex_gts` — Operations on triangulated surfaces using GTS functions.

This module provides access to GTS from insisde pyFormex.

`Classes defined in module pyformex_gts`

`Functions defined in module pyformex_gts`

`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*: a closed manifold surface
- *op*: boolean operation: one of '+', '-' or '*'.
- *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 closed manifold TriSurface

`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.

`pyformex_gts.`**`inside`**(*self*, *pts*)

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 `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 section2d`

class `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 decribed by a finite number of points and by straight segments connecting them.

`Functions defined in module section2d`

`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

`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.)

`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 `sectionize` — sectionize.py

Create, measure and approximate cross section of a Formex.

`Classes defined in module sectionize`

`Functions defined in module sectionize`

`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.

`sectionize.`**`centerline`**(*F*, *dir*, *nx=2*, *mode=2*, *th=0.2*)
Compute the centerline in the direction dir.

`sectionize.`**`createSegments`**(*F*, *ns=None*, *th=None*)
Create segments along 0 axis for sectionizing the Formex F.

`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.

`sectionize.`**`drawCircles`**(*sections*, *ctr*, *diam*)
Draw circles as approximation of Formices.

6.4.29 `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

`Classes defined in module tetgen`

`Functions defined in module tetgen`

`tetgen.`**`readNodeFile`**(*fn*)
Read a tetgen .node file.

Returns a tuple as described in readNodesBlock.

`tetgen.`**`readEleFile`**(*fn*)
Read a tetgen .ele file.

Returns a tuple as described in readElemsBlock.

`tetgen.`**`readFaceFile`**(*fn*)
Read a tetgen .face file.

Returns a tuple as described in readFacesBlock.

tetgen.**readSmeshFile**(*fn*)
Read a tetgen .smesh file.

Returns an array of triangle elements.

tetgen.**readPolyFile**(*fn*)
Read a tetgen .poly file.

Returns an array of triangle elements.

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).

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.

tetgen.**stripLine**(*line*)
Strip blanks, newline and comments from a line of text.

tetgen.**getInts**(*line*, *nint*)
Read a number of ints from a line, adding zero for omitted values.

line is a string with b;anks separated integer values. Returns a list of nint integers. The trailing ones are set to zero if the strings contains less values.

tetgen.**addElem**(*elems*, *nrs*, *e*, *n*, *nplex*)
Add an element to a collection.

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.

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.

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.

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

tetgen.**readNeigh** (*fn*)
Read a tetgen .neigh file.

Returns an arrays containing the tetrahedra neighbours:

tetgen.**writeNodes** (*fn*, *coords*, *offset=0*)
Write a tetgen .node file.

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.

tetgen.**writeTmesh** (*fn*, *elems*, *offset=0*)
Write a tetgen .ele file.

Writes elements of a tet4 mesh.

tetgen.**writeSurface** (*fn*, *coords*, *elems*)
Write a tetgen surface model to .node and .smesh files.

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.

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.

tetgen.**nextFilename** (*fn*)
Returns the next file name in a family of tetgen file names.

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.

`tetgen.`**`readTetgen`**(*fn*)
Read and draw a tetgen file.

This is an experimental function for the geometry import menu.

`tetgen.`**`tetgenConvexHull`**(*pts*)
Tetralize the convex hull of some points.

Finds the convex hull some points and returns a tet mesh of the convex hull and the convex hull (tri3 mesh).

If all points are on the same plane there is no convex hull.

This could be made an example:

from simple import regularGrid X = Coords(regularGrid([0., 0., 0.], [1., 1., 1.], [10, 10, 10]).reshape(-1, 3)).addNoise(rsize=0.05,asize=0.5) draw(X) from plugins.tetgen import tetgenConvexHull tch, ch =tetgenConvexHull(X) draw(ch, color='red', marksize=10)

`tetgen.`**`checkSelfIntersectionsWithTetgen`**(*self*, *verbose=False*)
check self intersections using tetgen

Returns couples of intersecting triangles

`tetgen.`**`tetMesh`**(*surfacefile*, *quality=False*, *volume=None*, *outputdir=None*)
Create a tetrahedral mesh inside a surface

- *surfacefile*: a file representing a surface. It can be an .off or .stl file (or other?)
- *quality*: if True, the output will be a quality mesh The circumradius-to-shortest-edge ratio can be constrained by specifying a float value for quality (default is 2.0)
- *volume*: float: applies a maximum tetrahedron volume constraint
- *outputdir*: if specified, the results will be placed in this directory. The default is to place the results in the same directory as the input file.

If the creation of the tetrahedral model is succesful, the results are read back using readTetgen and returned.

6.4.30 `tools` — tools.py

Graphic Tools for pyFormex.

`Classes defined in module tools`

`Functions defined in module tools`

`tools.`**`getObjectItems`**(*obj*, *items*, *mode*)
Get the specified items from object.

`tools.`**`getCollection`**(*K*)
Returns a collection.

`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.

`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.

`tools.`**`getPartition`**`(`*K*, *prop*`)`
Remove all partitions with property not in prop.

`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.

6.4.31 `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.

The surface contains *ntri* triangles, each having 3 vertices with 3 coordinates. The surface can be initialized from one of the following:

- a (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 (nedges,2) integer array of vertex numbers, an (ntri,3) integer array of edges numbers.

Additionally, a keyword argument prop= may be specified to set property values.

`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(*clas, 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:

- .stl (ASCII or BINARY)
- .gts
- .off
- .neu (Gambit Neutral)
- .smesh (Tetgen)

Gzipped .stl, .gts and .off files are also supported. Their names should be the normal filename with '.gz' appended. These files are uncompressed on the fly during the reading and the uncompressed versions are deleted after reading.

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.

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.

inertia()
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.

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 continously be deformed to an open disk.

nonManifoldEdges()
Finds edges and faces that are not Manifold.

Returns a tuple of:

- the edges that connect 3 or more faces,
- the faces connected to these edges.

isClosedManifold()
Check whether the TriSurface is a closed manifold.

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 usefull 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 surfaceInsideBorder function.

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.

perimeters()
Compute the perimeters of all triangles.

quality()
Compute a quality measure for the triangle schapes.

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 apect 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.

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 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 eacht 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*, ***kargs*)
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.

connectedElements (*target*, *elemlist=None*)
Return the elements from list connected with target

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

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 *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, wich can be checked with `isClosedManifold()`).

Returns a tuple of:

- an integer return code with the value:
  - 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.
- the intersecting triangles in the case of a return code 3, else None. If matched==True, intersecting triangles are returned as element indices of self, otherwise as a separate TriSurface object.

If verbose is True, prints the statistics reported by the gtscheck command.

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 the 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: stops the coarsening process if the number of edges was to fall below it
- *max_cost*: float: stops the coarsening process if the cost of collapsing an edge is larger
- *mid_vertex*: boolean: use midvertex as replacement vertex instead of the default, which is a volume optimized point
- *length_cost*: boolean: use length^2 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*: boolean: write progressive surface file
- *log*: boolean: log the evolution of the cost
- *verbose*: boolean: 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. 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

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*: boolean: print statistics about the surface

See also: `smoothLowPass()`, `smoothLaplaceHC()`

inside (*pts*, *method='gts'*, *tol=0.0*)
Test which of the points pts are inside the surface.

Parameters:

- *pts*: a Coords or compatible.
- *method*: string: method to be used for the detection. Depending on the software you have installed the following are possible:
  - 'gts': provided by pyformex-extra
  - 'vtk': provided by python-vtk
- *tol*: only available for method 'vtk'

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().

tetgen (*quality=True*, *volume=None*, *filename=None*, *format='.off'*)
Create a tetrahedral mesh inside the surface

- *surfacefile*: a file representing a surface. It can be an .off or .stl file (or other?)
- *quality*: if True, the output will be a quality mesh The circumradius-to-shortest-edge ratio can be constrained by specifying a float value for quality (default is 2.0) - *volume*: float: applies a maximum tetrahedron volume constraint
- *outputdir*: if specified, the results surface model and the tet model files will be placed in this directory. Else, a temporary directory will be used.

If the creation of the tetrahedral model is succesful, the resulting tetrahedral mesh is returned.

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*: a closed manifold surface

- *op*: boolean operation: one of '+', '-' or '*'.
- *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 closed manifold TriSurface

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.

scaledJacobian (*scaled=True*, *blksize=100000*)
Compute a quality measure for volume meshes.

Parameters:

- *scaled*: if False returns the Jacobian at the corners of each element. If True, returns a quality metrics, being the minimum value of the scaled Jacobian in each element (at one corner, the Jacobian divided by the volume of a perfect brick).
- *blksize*: int: to reduce the memory required for large meshes, the Mesh is split in blocks with this number of elements. If not positive, all elements are handled at once.

If *scaled* is True each tet or hex element gets a value between -1 and 1. Acceptable elements have a positive scaled Jacobian. However, good quality requires a minimum of 0.2. Quadratic meshes are first converted to linear. If the mesh contain mainly negative Jacobians, it probably has negative volumes and can be fixed with the correctNegativeVolumes.

addNoise (**args*, ***kargs*)
Apply 'addNoise' transformation to the Geometry object.

See `coords.Coords.addNoise()` for details.

affine (**args*, ***kargs*)
Apply 'affine' transformation to the Geometry object.

See `coords.Coords.affine()` for details.

align (**args*, ***kargs*)
Apply 'align' transformation to the Geometry object.

See `coords.Coords.align()` for details.

bump (**args*, ***kargs*)
Apply 'bump' transformation to the Geometry object.

See `coords.Coords.bump()` for details.

bump1 (**args*, ***kargs*)
Apply 'bump1' transformation to the Geometry object.

See `coords.Coords.bump1()` for details.

bump2 (**args*, ***kargs*)
Apply 'bump2' transformation to the Geometry object.

See `coords.Coords.bump2()` for details.

centered (**args*, ***kargs*)
Apply 'centered' transformation to the Geometry object.

See `coords.Coords.centered()` for details.

cylindrical (**args*, ***kargs*)
Apply 'cylindrical' transformation to the Geometry object.

See `coords.Coords.cylindrical()` for details.

egg (**args*, ***kargs*)
Apply 'egg' transformation to the Geometry object.

See `coords.Coords.egg()` for details.

flare (**args*, ***kargs*)
Apply 'flare' transformation to the Geometry object.

See `coords.Coords.flare()` for details.

hyperCylindrical (**args*, ***kargs*)
Apply 'hyperCylindrical' transformation to the Geometry object.

See `coords.Coords.hyperCylindrical()` for details.

isopar (**args*, ***kargs*)
Apply 'isopar' transformation to the Geometry object.

See `coords.Coords.isopar()` for details.

map (**args*, ***kargs*)
Apply 'map' transformation to the Geometry object.

See `coords.Coords.map()` for details.

map1 (**args*, ***kargs*)
Apply 'map1' transformation to the Geometry object.

See `coords.Coords.map1()` for details.

mapd (**args*, ***kargs*)
Apply 'mapd' transformation to the Geometry object.

See `coords.Coords.mapd()` for details.

position (**args*, ***kargs*)
Apply 'position' transformation to the Geometry object.

See `coords.Coords.position()` for details.

projectOnCylinder (**args*, ***kargs*)
Apply 'projectOnCylinder' transformation to the Geometry object.

See `coords.Coords.projectOnCylinder()` for details.

projectOnPlane (**args*, ***kargs*)
Apply 'projectOnPlane' transformation to the Geometry object.

See `coords.Coords.projectOnPlane()` for details.

projectOnSphere (**args*, ***kargs*)
Apply 'projectOnSphere' transformation to the Geometry object.

See `coords.Coords.projectOnSphere()` for details.

replace (**args*, ***kargs*)
Apply 'replace' transformation to the Geometry object.

See `coords.Coords.replace()` for details.

rollAxes (**args*, ***kargs*)
Apply 'rollAxes' transformation to the Geometry object.

See `coords.Coords.rollAxes()` for details.

rot (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

rotate (**args*, ***kargs*)
Apply 'rotate' transformation to the Geometry object.

See `coords.Coords.rotate()` for details.

scale (**args*, ***kargs*)
Apply 'scale' transformation to the Geometry object.

See `coords.Coords.scale()` for details.

shear (**args*, ***kargs*)
Apply 'shear' transformation to the Geometry object.

See `coords.Coords.shear()` for details.

spherical (**args*, ***kargs*)
Apply 'spherical' transformation to the Geometry object.

See `coords.Coords.spherical()` for details.

superSpherical (**args*, ***kargs*)
Apply 'superSpherical' transformation to the Geometry object.

See `coords.Coords.superSpherical()` for details.

swapAxes (**args*, ***kargs*)
Apply 'swapAxes' transformation to the Geometry object.

See `coords.Coords.swapAxes()` for details.

toCylindrical (**args*, ***kargs*)
Apply 'toCylindrical' transformation to the Geometry object.

See `coords.Coords.toCylindrical()` for details.

toSpherical (**args*, ***kargs*)
Apply 'toSpherical' transformation to the Geometry object.

See `coords.Coords.toSpherical()` for details.

transformCS (**args*, ***kargs*)
Apply 'transformCS' transformation to the Geometry object.

See `coords.Coords.transformCS()` for details.

translate (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

trl (**args*, ***kargs*)
Apply 'translate' transformation to the Geometry object.

See `coords.Coords.translate()` for details.

elementToNodal (*val*)
Compute nodal values from element values.

Given scalar values defined on elements, finds the average values at the nodes. Returns the average values at the (maxnodenr+1) nodes. Nodes not occurring in elems will have all zero values. NB. It now works with scalar. It could be extended to vectors.

nodalToElement (*val*)
Compute element values from nodal values.

Given scalar values defined on nodes, finds the average values at elements. NB. It now works with scalar. It could be extended to vectors.

setProp (*prop=None*, *blocks=None*)
Create or destroy the property array for the Geometry.

A property array is a rank-1 integer array with dimension 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. They play an import role when creating new geometry: new elements inherit the property number of their parent element. Properties are also preserved on most geometrical 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*: a single integer value or a list/array of integer values. If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored.

  The special value 'range' will set the property numbers equal to the element number.

  A value None (default) removes the properties from the Geometry.

- *blocks*: a single integer value or a list/array of integer values. If the number of passed values is less than the length of *prop*, they wil be repeated. If you give more, they will be ignored. Every prop will be repeated the corresponding number of times specified in blocks.

setType (*eltype=None*)
Set the eltype from a character string.

This function allows the user to change the element type of the Mesh. The input is a character string with the name of one of the element defined in elements.py. The function will only allow to set a type matching the plexitude of the Mesh.

This method is seldom needed, because the applications should normally set the element type at creation time.

elType()
Return the element type of the Mesh.

elName()
Return the element name of the Mesh.

toProp(*prop*)
Converts the argument into a legal set of properties for the object.

The conversion involves resizing the argument to a 1D array of length self.nelems(), and converting the data type to integer.

setNormals(*normals=None*)
Set/Remove the normals of the mesh.

copy()
Return a deep copy of the Geometry object.

The returned object is an exact copy of the input, but has all of its data independent of the former.

splitProp(*prop=None*)
Partition a Geometry (Formex/Mesh) according to the values in prop.

Parameters:

- *prop*: an int array with length self.nelems(), or None. If None, the *prop* attribute of the Geometry is used.

Returns a list of Geometry objects of the same type as the input. Each object 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 their *prop* value.

It *prop* is None and the the object has no *prop* attribute, an empty list is returned.

getProp()
Return the properties as a numpy array (ndarray)

maxProp()
Return the highest property value used, or None

propSet()
Return a list with unique property values.

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.

shallowCopy(*prop=None*)
Return a shallow copy.

A shallow copy of a Mesh is a Mesh object using the same data arrays as the original Mesh. The only thing that can be changed is the property array. This is a convenient method to use the same Mesh with different property attributes.

toFormex()
Convert a Mesh to a Formex.

The Formex inherits the element property numbers and eltype from the Mesh. Node property numbers however can not be translated to the Formex data model.

toMesh()
Convert to a Mesh.

This just returns the Mesh object itself. It is provided as a convenience for use in functions that want 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. The resulting TriSurface is only fully equivalent with the input Mesh if the latter has element type 'tri3'.

On success, returns a TriSurface corresponding with the input Mesh. If the Mesh can not be converted to a TriSurface, an error is raised.

toCurve()
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)

This is equivalent with

```
self.toFormex().toCurve()
```

Any other type will raise an exception.

info()
Return short info about the Mesh.

This includes only the shape of the coords and elems arrays.

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. (Though numpy will limit the printed output).

TODO: We should add an option here to let numpy print the full tables.

centroids()
Return the centroids of all elements of the Mesh.

The centroid of an element is the point whose coordinates are the mean values of all points of the element. The return value is a Coords object with nelems points.

bboxes()
Returns the bboxes of all elements in the Mesh.

Returns a coords with shape (nelems,2,3). Along the axis 1 are stored the minimal and maximal values of the Coords in each of the elements of the Mesh.

getCoords()
Get the coords data.

Returns the full array of coordinates stored in the Mesh object. Note that this may contain points that are not used in the mesh. `compact()` will remove the unused points.

webgl(*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

getElems()
Get the elems data.

Returns the element connectivity data as stored in the object.

getLowerEntities(*level=-1*, *unique=False*)
Get the entities of a lower dimensionality.

If the element type is defined in the `elements` module, this returns a Connectivity table with the entities of a lower dimensionality. The full list of entities with increasing dimensionality 0,1,2,3 is:

```
['points', 'edges', 'faces', 'cells' ]
```

If level is negative, the dimensionality returned is relative to that of the caller. If it is positive, it is taken absolute. 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.

By default, all entities for all elements are returned and common entities will appear multiple times. Specifying unique=True will return only the unique ones.

The return value may be an empty table, if the element type does not have the requested entities (e.g. the 'point' type). If the eltype is not defined, or the requested entity level is outside the range 0..3, the return value is None.

getNodes()
Return the set of unique node numbers in the Mesh.

This returns only the node numbers that are effectively used in the connectivity table. For a compacted Mesh, it is equivalent to `arange(self.nelems)`. This function also stores the result internally so that future requests can return it without the need for computing it again.

getPoints()
Return the nodal coordinates of the Mesh.

This returns only those points that are effectively 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.

This is a convenient function to create a table with the element edges. It is equivalent to `self.getLowerEntities(1,unique=True)`, but this also stores the result internally so that future requests can return it without the need for computing it again.

getFaces()
Return the unique faces of all the elements in the Mesh.

This is a convenient function to create a table with the element faces. It is equivalent to `self.getLowerEntities(2,unique=True)`, but this also stores the result internally so that future requests can return it without the need for computing it again.

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.

getFreeEntities(*level=-1*, *return_indices=False*)
Return the border of the Mesh.

Returns a Connectivity table with the free entities of the specified level of the Mesh. Free entities are entities that are only connected with a single element.

If return_indices==True, also returns an (nentities,2) index for inverse lookup of the higher entity (column 0) and its local lower entity number (column 1).

getFreeEntitiesMesh(*level=-1*, *compact=True*)
Return a Mesh with lower entities.

Returns a Mesh representing 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.

By default, the resulting Mesh is compacted. Compaction can be switched off by setting *compact=False*.

getBorder(*return_indices=False*)
Return the border of the Mesh.

This returns a Connectivity table with the border of the Mesh. The border entities are of a lower hierarchical level than the mesh itself. These entities become part of the border if they are connected to only one element.

If return_indices==True, it returns also an (nborder,2) index for inverse lookup of the higher entity (column 0) and its local border part number (column 1).

This is a convenient shorthand for

```
self.getFreeEntities(level=-1,return_indices=return_indices)
```

getBorderMesh(*compact=True*)
Return a Mesh with the border elements.

The returned Mesh is of the next lower hierarchical level and contains all the free entitites of that level. If the Mesh has property numbers, the border elements inherit the property of the element to which they belong.

By default, the resulting Mesh is compacted. Compaction can be switched off by setting *compact=False*.

This is a convenient shorthand for

```
self.getFreeEntitiesMesh(level=-1,compact=compact)
```

getBorderElems()

Return the elements that are on the border of the Mesh.

This returns a list with the numbers of the elements that are on the border of the Mesh. Elements are considered to be at the border if they contain at least one complete element of the border Mesh (i.e. an element of the first lower hierarchical level). Thus, in a volume Mesh, elements only touching the border by a vertex or an edge are not considered border elements.

getBorderNodes()

Return the nodes that are on the border of the Mesh.

This returns a list with the numbers of the nodes that are on the border of the Mesh.

peel(*nodal=False*)

Return a Mesh with the border elements removed.

If nodal is True all elements connected to a border node are removed. If nodal is False, it is a convenient shorthand for

```
self.cselect(self.getBorderElems())
```

getFreeEdgesMesh(*compact=True*)

Return a Mesh with the free edge elements.

The returned Mesh is of the hierarchical level 1 (no mather what the level of the parent Mesh is) and contains all the free entitites of that level. If the Mesh has property numbers, the border elements inherit the property of the element to which they belong.

By default, the resulting Mesh is compacted. Compaction can be switched off by setting *compact=False*.

This is a convenient shorthand for

```
self.getFreeEntitiesMesh(level=1,compact=compact)
```

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. The level is one of the lower entities of the mesh.

Parameters:

- *level*: 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.
- *diflevel*: if >= level, and smaller than the hierarchy of self.elems, 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 an Adjacency with integers 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 simultanuously 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*: 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.

The remainder of the parameters are like in `Adjacency.frontWalk()`.

Returns an array of integers specifying for each element in which step the element was reached by the walker.

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.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 (poly)line from a point in one element to a point in the other element without leaving the Mesh. The partitioning is returned as a integer array having a value for ech 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.

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.

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).

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(***kargs*)
Fuse the nodes of a Meshes.

All nodes that are within the tolerance limits of each other are merged into a single node.

The merging operation can be tuned by specifying extra arguments that will be passed to `Coords:fuse()`.

matchCoords(*mesh*, ***kargs*)
Match nodes of Mesh with nodes of self.

This is a convenience function equivalent to:

```
self.coords.match(mesh.coords,**kargs)
```

See also `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 ()
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.

Example:

```
>>> x = Coords([[i] for i in arange(5)])
>>> M = Mesh(x,[[0,2],[1,4],[4,2]])
>>> M = M.compact()
>>> 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]]
```

select (*selected*, *compact=True*)
Return a Mesh only holding the selected elements.

Parameters:

- *selected*: an object that can be used as an index in the *elems* array, such as
  - a single element number
  - a list, or array, of element numbers
  - a bool array of length self.nelems(), where True values flag the elements to be selected
- *compact*: boolean. If True (default), the returned Mesh will be compacted, i.e. the unused nodes are removed and the nodes are renumbered from zero. If False, returns the node set and numbers unchanged.

Returns a Mesh (or subclass) with only the selected elements.

See `cselect()` for the complementary operation.

cselect (*selected*, *compact=True*)
Return a mesh without the selected elements.

Parameters:

- *selected*: an object that can be used as an index in the *elems* array, such as
  - a single element number
  - a list, or array, of element numbers
  - a bool array of length self.nelems(), where True values flag the elements to be selected
- *compact*: boolean. If True (default), the returned Mesh will be compacted, i.e. the unused nodes are removed and the nodes are renumbered from zero. If False, returns the node set and numbers unchanged.

Returns a Mesh with all but the selected elements.

This is the complimentary operation of `select()`.

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.

withProp (*val*)

Return a Mesh which holds only the elements with property val.

val is either a single integer, or a list/array of integers. The return value is a Mesh holding all the elements that have the property val, resp. one of the values in val. The returned Mesh inherits the matching properties.

If the Mesh has no properties, a copy with all elements is returned.

withoutProp (*val*)

Return a Mesh without the elements with property *val*.

This is the complementary method of Mesh.withProp(). val is either a single integer, or a list/array of integers. The return value is a Mesh holding all the elements that do not have the property val, resp. one of the values in val. The returned Mesh inherits the matching properties.

If the Mesh has no properties, a copy with all elements is returned.

connectedTo (*nodes*)

Return a Mesh with the elements connected to the specified node(s).

nodes: int or array_like, int.

Return a Mesh with all the elements from the original that contain at least one of the specified nodes.

notConnectedTo (*nodes*)

Return a Mesh with the elements not connected to the given node(s).

nodes: int or array_like, int.

Returns a Mesh with all the elements from the original that do not contain any of the specified nodes.

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 elementt'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 `Mesh.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*)
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.

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'.

---

reduceDegenerate (*eltype=None*)
Reduce degenerate elements to lower plexitude elements.

This will try to reduce the degenerate elements of the mesh to elements of a lower plexitude. If a target element type is given, only the matching reduce scheme is tried. Else, all the target element types for which a reduce scheme from the Mesh eltype is available, will be tried.

The result is a list of Meshes of which the last one contains the elements that could not be reduced and may be empty. Property numbers propagate to the children.

splitDegenerate (*autofix=True*)
Split a Mesh in degenerate and non-degenerate elements.

If autofix is 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.

The return value is a list of Meshes. 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. These may still contain degenerate elements.

removeDuplicate (*permutations=True*)
Remove the duplicate elements from a Mesh.

Duplicate elements are elements that consist of the same nodes, by default in no particular order. Setting permutations=False will only consider elements with the same nodes in the same order as duplicates.

Returns a Mesh with all duplicate elements removed.

renumber (*order='elems'*)
Renumber the nodes of a Mesh in the specified order.

order is an index with length equal to the number of nodes. 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 value 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.

renumberElems (*order='nodes'*)
Reorder the elements of a Mesh.

Parameters:

- *order*: either a 1-D integer array with a permutation of `arange(self.nelems())`, specifying the requested order, or 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 a Mesh equivalent with self but with the elements ordered as specified.

See also: `Connectivity.reorder()`

reorder (*order='nodes'*)
Reorder the elements of a Mesh.

Parameters:

- *order*: either a 1-D integer array with a permutation of `arange(self.nelems())`, specifying the requested order, or 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 a Mesh equivalent with self but with the elements ordered as specified.

See also: `Connectivity.reorder()`

connect (*coordslist*, *div=1*, *degree=1*, *loop=False*, *eltype=None*)
Connect a sequence of toplogically 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 toplogically 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*: Either an integer, or a sequence of float numbers (usually in the range]0.0..1.0]) or a list of sequences of the same length of the connecting list of coordinates. In the latter case every sequence inside the list can either be a float sequence (usually in the range]0.0..1.0]) or it contains one integer (e.g [[4],[0.3,1]]). This should only be used for degree==1.

  With this parameter the generated elements can be further subdivided along the connection direction. If an int is given, the connected elements will be divided into this number of elements along the connection direction. If a sequence of float numbers is given, the numbers specify the relative distance along the connection direction where the elements should end. If the last value in the sequence is not equal to 1.0, there will be a gap between the consecutive connections. If a list of sequences is given, every consecutive element of the coordinate list is connected using the corresponding sequence in the list(1-length integer of float sequence specified as before).
- *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 (*n*, *step=1.0*, *dir=0*, *degree=1*, *eltype=None*)
Extrude a Mesh in one of the axes directions.

Returns a new Mesh obtained by extruding the given Mesh over *n* steps of length *step* in direction of axis *dir*.

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

Returns a new Mesh obtained by sweeping the given Mesh over a path. The returned Mesh has double plexitude of the original.

This method accepts all the parameters of `coords.sweepCoords()`, with the same meaning. Usually, you will need to at least set the *normal* parameter. The *eltype* parameter can be used to set the element type on the returned Meshes.

This operation is similar to the extrude() method, but the path can be any 3D curve.

classmethod concatenate (*clas*, *meshes*, ***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: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 compaitble 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.

clip (*t*, *compact=True*)
Return a Mesh with all the elements where t>0.

t should be a 1-D integer array with length equal to the number of elements of the Mesh. The resulting Mesh will contain all elements where t > 0.

cclip (*t*, *compact=True*)
This is the complement of clip, returning a Mesh where t<=0.

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)

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 succesful, 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 unsuccesful.

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 succesful, 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.

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 * surface of base * 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.

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 trisurface

trisurface.**stlConvert**(*stlname*, *outname=None*, *binary=False*, *options='-d'*)

Transform an .stl file to .off or .gts or binary .stl format.

Parameters:

- *stlname*: name of an existing .stl file (either ascii or binary).
- *outname*: name of the output file. The extension defines the format and should be one of '.off', '.gts', '.stl', '.stla', or .stlb'. As a convenience, if a file extension only is given (other than '.stl'), then the outname will be constructed by changing the extension of the input *stlname*.
- *binary*: if the extension of *outname* is '.stl', defines whether the output format is a binary or ascii STL format.

If the outname file exists and its mtime is more recent than the stlname, the outname file is considered uptodate and the conversion program will not be run.

The conversion program will be choosen depending on the extension. This uses the external commands 'admesh' or 'stl2gts'.

The return value is a tuple of the output file name, the conversion program exit code (0 if succesful) and the stdout of the conversion program (or a 'file is already uptodate' message).

trisurface.**read_gts**(*fn*)

Read a GTS surface mesh.

Return a coords,edges,faces tuple.

trisurface.**read_stl**(*fn*, *intermediate=None*)

Read a surface from .stl file.

This is done by first coverting 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.**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 center() of the coordinates *x*.

Returns an (ntri) shaped array with the volume of the tetraeders formed by the triangles and the point *pt*. If *x* represents a closed surface, the sum of this array will represent the volume inside the surface.

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 surface inside a given closed border line.

The border line is a closed polygonal line and can be specified as one of the following:

- a closed PolyLine,
- a 2-plex Mesh, with a Connectivity table such that the elements in order form a closed polyline,
- a simple Coords specifying the subsequent vertices of the polygonal border line.

The return value is a TriSurface filling the hole inside the border.

There are currently two fill methods available:

- '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.

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 change that the result contains intersecting triangles.

This '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 by the `check()` method.

Note: Because the 'border' does not create any new points, the returned surface will use the same point coordinate array as the input object.

trisurface.**read_error**(*cnt*, *line*)
Raise an error on reading the stl file.

trisurface.**read_stla**(*fn*, *dtype=<type 'numpy.float32'>*, *large=False*, *guess=True*)
Read an ascii .stl file into an [n,3,3] float array.

If the .stl is large, read_ascii_large() is recommended, as it is a lot faster.

trisurface.**read_ascii_large**(*fn*, *dtype=<type 'numpy.float32'>*)
Read an ascii .stl file into an [n,3,3] float array.

This is an alternative for read_ascii, which is a lot faster on large STL models. It requires the 'awk' command though, so is probably only useful on Linux/UNIX. It works by first transforming the input file to a .nodes file and then reading it through numpy's fromfile() function.

`trisurface.`**`off_to_tet`**(*fn*)
Transform an .off model to tetgen (.node/.smesh) format.

`trisurface.`**`find_row`**(*mat*, *row*, *nmatch=None*)
Find all rows in matrix matching given row.

`trisurface.`**`find_nodes`**(*nodes*, *coords*)
Find nodes with given coordinates in a node set.

nodes is a (nnodes,3) float array of coordinates. coords is a (npts,3) float array of coordinates.

Returns a (n,) integer array with ALL the node numbers matching EXACTLY ALL the coordinates of ANY of the given points.

`trisurface.`**`find_first_nodes`**(*nodes*, *coords*)
Find nodes with given coordinates in a node set.

nodes is a (nnodes,3) float array of coordinates. coords is a (npts,3) float array of coordinates.

Returns a (n,) integer array with THE FIRST node number matching EXACTLY ALL the coordinates of EACH of the given points.

`trisurface.`**`find_triangles`**(*elems*, *triangles*)
Find triangles with given node numbers in a surface mesh.

elems is a (nelems,3) integer array of triangles. triangles is a (ntri,3) integer array of triangles to find.

Returns a (ntri,) integer array with the triangles numbers.

`trisurface.`**`remove_triangles`**(*elems*, *remove*)
Remove triangles from a surface mesh.

elems is a (nelems,3) integer array of triangles. remove is a (nremove,3) integer array of triangles to remove.

Returns a (nelems-nremove,3) integer array with the triangles of nelems where the triangles of remove have been removed.

`trisurface.`**`Rectangle`**(*nx*, *ny*)
Create a plane rectangular surface consisting of a nx,ny grid.

`trisurface.`**`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.

6.4.32 `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()
```

`Classes defined in module turtle`

`Functions defined in module turtle`

`turtle.`**`sind`**`(`*arg*`)`
Return the sine of an angle in degrees.

`turtle.`**`cosd`**`(`*arg*`)`
Return the cosine of an angle in degrees.

`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.

`turtle.`**`push`**`()`
Save the current state of the turtle.

The turtle state includes its position, step and angle.

`turtle.`**`pop`**`()`
Restore the turtle state to the last saved state.

`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.

`turtle.`**`mv`**`(`*d=None*`)`
Move over step *d* without drawing.

`turtle.`**`ro`**`(`*a*`)`
Rotate over angle *a*. The new direction is incremented with *a*

`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.

`turtle.`**`st`**`(`*d*`)`
Set the step size.

`turtle.`**`an`**`(`*a*`)`
Set the angle

`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.

6.4.33 `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 units

class `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/mm^3'. If you want to use t/m^3, 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 everyting 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 entitities.

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 units`

`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 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.

6.4.34 `vascularsweepingmesher` — Vascular Sweeping Mesher

`Classes defined in module vascularsweepingmesher`

`Functions defined in module vascularsweepingmesher`

`vascularsweepingmesher.`**`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.

`vascularsweepingmesher.`**`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. from __future__ import print_function If isopHex is specified, a convenient set of control points for the isoparametric transformation hex64 is also returned. TODO: include other optons to get the control points for other isoparametric transformation for hex.

`vascularsweepingmesher.`**`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.

`vascularsweepingmesher.`**`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.

`vascularsweepingmesher.`**`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.

`vascularsweepingmesher.`**`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 crossectionally)

`vascularsweepingmesher.`**`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

`vascularsweepingmesher.`**`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.

`vascularsweepingmesher.`**`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.

`vascularsweepingmesher.`**`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.

`vascularsweepingmesher.`**`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.

6.4.35 `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 webgl

class webgl.**WebGL**(*name='Scene1'*)

A 3D geometry model for export to WebGL.

The WebGL class provides a limited model to be easily exported as a complete WebGL model, including the required HTML, Javascript and data files.

Currently the following features are included:

- create a new WebGL model
- add the current scene to the model
- add Geometry to the model (including color and transparency)
- set the camera position
- export the model

An example of its usage can be found in the WebGL example.

The created model uses the XTK toolkit from http://www.goXTK.com or the modified version of FEops.

objdict(*clas=None*)

Return a dict with the objects in this model.

Returns a dict with the name:object pairs in the model. Objects that have no name are disregarded.

addScene()

Add the current OpenGL scene to the WebGL model.

This method add all the geometry in the current viewport to the WebGL model.

add(***kargs*)

Add a geometry object to the model.

Currently, two types of objects can be added: pyFormex Geometry objects and file names. Geometry objects should be convertible to TriSurface (using their toSurface method). Geometry files should be in STL format.

The following keyword parameters are available and all optional:

- *obj*=: specify a pyFormex Geometry object
- *file*=: specify a geometry data file (STL). If no *obj* is specified, the file should exist. If an *obj* file is specified, this is the name that will be used to export the object.
- *name*=: specify a name for the object. The name will be used as a variable in the Javascript script and as filename for for export if an *obj* was specified but no *file* was given. It should only contain alphanumeric characters and not start with a digit.
- *caption*=: specify a caption to be used as a tooltip when the mouse hovers over the object.
- *color*=: specify a color to be sued for the object. The color should be a list of 3 values in the range 0..1 (OpenGL color).
- *opacity*=: specify a value for the opacity of the object (the 'alpha' value in pyFormex terms).

- *magicmode=*: specify True or False. If magicmode is True, colors will be set from the normals of the object. This is incompatible with *color=*.
- *control=*: a list of attributes that get a gui controller

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.

camera (***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 (*obj*)
Export an object in XTK Javascript format

format_gui_controller (*name*, *attr*)
Format a single controller

format_gui ()
Create the controller GUI script

exportPGF (*fn*, *sep=''*)
Export the current scene to a pgf file

export (*name=None*, *title=None*, *description=None*, *keywords=None*, *author=None*, *createdby=False*)
Export the WebGL scene.

Parameters:

- *name*: a string that will be used for the filenames of the HTML, JS and STL files.
- *title*: an optional title to be set in the .html file. If not specified, the *name* is used.

You can also set the meta tags 'description', 'keywords' and 'author' to be included in the .html file. The first two have defaults if not specified.

Returns the name of the exported htmlfile.

Functions defined in module webgl

webgl.**saneSettings** (*k*)
Sanitize sloppy settings for JavaScript output

webgl.**properties** (*o*)
Return properties of an object

properties are public attributes (not starting with an '_') that are not callable.

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

6.5 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 avaliable:

- geometry_menu
- formex_menu
- surface_menu
- tools_menu
- draw2d_menu
- nurbs_menu
- dxf_tools
- jobs menu
- postproc_menu
- bifmesh_menu

6.6 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.

6.6.1 `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`

`Functions defined in module olist`

`olist.`**`roll`**`(`*a*, *n=1*`)`
Roll the elements of a list n positions forward (backward if n < 0)

`olist.`**`union`**`(`*a*, *b*`)`
Return a list with all items in a or in b, in the order of a,b.

`olist.difference(a, b)`
Return a list with all items in a but not in b, in the order of a.

`olist.symdifference(a, b)`
Return a list with all items in a or b but not in both.

`olist.intersection(a, b)`
Return a list with all items in a and in b, in the order of a.

`olist.concatenate(a)`
Concatenate a list of lists

`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.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.

`olist.remove(a, b)`
Returns the complement of select(a,b).

`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.

`olist.collectOnLength(items, return_indices=False)`
Collect items of a list in separate bins according to the item length.

items is a list of items of any type having the len() method. The items are put in separate lists according to their length.

The return value is a dict where the keys are item lengths and the values are lists of items with this length.

If return_indices is True, a second dict is returned, with the same keys, holding the original indices of the items in the lists.

6.6.2 mydict —

CDict is a Dict with lookup cascading into the next level Dict's if the key is not found in the CDict itself.

(C) 2005,2008 Benedict Verhegghe Distributed under the GNU GPL version 3 or later

Classes defined in module mydict

class `mydict.Dict(data={}, default=None)`
A Python dictionary with default values and attribute syntax.

Dict is functionally nearly equivalent with the builtin Python dict, but provides the following extras:

- Items can be accessed with attribute syntax as well as dictionary syntax. Thus, if C is a Dict, C['foo'] and C.foo are equivalent. This works as well for accessing values as for setting values. In the following, the terms *key* or *attribute* therefore have the same meaning.
- Lookup of a nonexisting key/attribute does not automatically raise an error, but calls a _default_ lookup method which can be set by the user. The default is to raise a KeyError, but an alternative is to return None or some other default value.

There are a few caveats though:

- Keys that are also attributes of the builtin dict type, can not be used with the attribute syntax to get values from the Dict. You should use the dictionary syntax to access these items. It is possible to set such keys as attributes. Thus the following will work:

  ```
  C['get'] = 'foo'
  C.get = 'foo'
  print(C['get'])
  ```

  but this will not:

  ```
  print(C.get)
  ```

  This is done so because we want all the dict attributes to be available with their normal binding. Thus,

  ```
  print(C.get('get'))
  ```

  will print foo

To avoid name clashes with user defines, many Python internal names start and end with '__'. The user should avoid such names. The Python dict has the following attributes not enclosed between '__', so these are the ones to watch out for: 'clear', 'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values'.

update (*data={}*, ***kargs*)
Add a dictionary to the Dict object.

The data can be a dict or Dict type object.

get (*key*, *default*)
Return the value for key or a default.

This is the equivalent of the dict get method, except that it returns only the default value if the key was not found in self, and there is no _default_ method or it raised a KeyError.

setdefault (*key*, *default*)
Replaces the setdefault function of a normal dictionary.

This is the same as the get method, except that it also sets the default value if get found a KeyError.

class mydict.**CDict** (*data={}*, *default=<function returnNone at 0x4eaecf8>*)
A cascading Dict: properties not in Dict are searched in all Dicts.

This is equivalent to the Dict class, except that if a key is not found and the CDict has items with values that are themselves instances of Dict or CDict, the key will be looked up in those Dicts as

well.

As you expect, this will make the lookup cascade into all lower levels of CDict's. The cascade will stop if you use a Dict. There is no way to guarantee in which order the (Cascading)Dict's are visited, so if multiple Dicts on the same level hold the same key, you should know yourself what you are doing.

update (*data={}*, ***kargs*)
Add a dictionary to the Dict object.

The data can be a dict or Dict type object.

get (*key*, *default*)
Return the value for key or a default.

This is the equivalent of the dict get method, except that it returns only the default value if the key was not found in self, and there is no _default_ method or it raised a KeyError.

setdefault (*key*, *default*)
Replaces the setdefault function of a normal dictionary.

This is the same as the get method, except that it also sets the default value if get found a KeyError.

`Functions defined in module mydict`

mydict.**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.

mydict.**cascade** (*d*, *key*)
Cascading lookup in a dictionary.

This is equivalent to the dict lookup, except that when the key is not found, a cascading lookup through lower level dict's is started and the first matching key found is returned.

mydict.**returnNone** (*key*)
Always returns None.

mydict.**raiseKeyError** (*key*)
Raise a KeyError.

6.6.3 `odict` — Specialized dictionary type structures.

`Classes defined in module odict`

class odict.**ODict** (*data={}*)
An ordered dictionary.

This is a dictionary that keeps the keys in order. The default order is the insertion order. The current order can be changed at any time.

The `ODict` can be initialized with a Python dict, a list of (key,value) tuples, or another `ODict` object. If a plain Python dict is used, the resulting order is undefined.

update (*data={}*)
Add a dictionary to the ODict object.

The new keys will be appended to the existing, but the order of the added keys is undetemined if data is a dict object. If data is an ODict its order will be respected..

sort (*keys*)
Set the order of the keys.

keys should be a list containing exactly all the keys from self.

keys ()
Return the keys in order.

values ()
Return the values in order of the keys.

items ()
Return the key,value pairs in order of the keys.

iteritems ()
Return the key,value pairs in order of the keys.

pos (*key*)
Return the position of the specified key.

If the key is not in the ODict, None is returned

class odict.**KeyedList** (*alist=*[ ])
A named item list.

A KeyedList is a list of lists or tuples. Each item (sublist or tuple) should at least have 2 elements: the first one is used as a key to identify the item, but is also part of the information (value) of the item.

items ()
Return the key+value lists in order of the keys.

update (*data={}*)
Add a dictionary to the ODict object.

The new keys will be appended to the existing, but the order of the added keys is undetemined if data is a dict object. If data is an ODict its order will be respected..

sort (*keys*)
Set the order of the keys.

keys should be a list containing exactly all the keys from self.

keys ()
Return the keys in order.

values ()
Return the values in order of the keys.

iteritems ()
Return the key,value pairs in order of the keys.

pos (*key*)
Return the position of the specified key.

If the key is not in the ODict, None is returned

`Functions defined in module odict`

6.6.4 `collection` — Tools for handling collections of elements belonging to multiple parts.

This module defines the Collection class.

`Classes defined in module collection`

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.

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 typ.

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.

has_key (*key*)
Check whether the collection has an entry for the key.

get (*key*, *default=[ ]*)
Return item with given key or default.

keys ()
Return a sorted array with the keys

items ()
Return a zipped list of keys and values.

`Functions defined in module collection`

6.6.5 `config` — A general yet simple configuration class.

(C) 2005 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=None*)

A configuration class allowing Python expressions in the input.

The configuration settings are stored in the __dict__ of a Python object. An item 'foo' in the configuration 'config' can be accessed either as dictionary lookup (`config['foo']`) or as object attribute (`config.foo`).

The configuration object can be initialized from a multiline string or a text file (or any other object that allows iterating over strings).

The format of the config file/text is described hereafter.

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 access as `self['name']`.

As an example, if your config file looks like:

```
aa = 'bb'
bb = aa
[cc]
aa = 'aa'
_n = 3
rng = range(_n)
```

the resulting configuration dictionary is `{'aa': 'bb', 'bb': 'bb', 'cc': {'aa': 'aa', 'rng': [0, 1, 2]}}`

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.

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.

read (*fil*, *debug=False*)
Read a configuration from a file or text

fil is a sequence of strings. Any type that allows a loop like `for line in fil:` to iterate over its text lines will do. This could be a file type, 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 -*-nn'*, *trailer='n# End of confign'*)
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.

get (*key*, *default*)
Return the value for key or a default.

This is the equivalent of the dict get method, except that it returns only the default value if the key was not found in self, and there is no _default_ method or it raised a KeyError.

setdefault (*key*, *default*)
Replaces the setdefault function of a normal dictionary.

This is the same as the get method, except that it also sets the default value if get found a KeyError.

`Functions defined in module config`

6.6.6 `flatkeydb` — Flat Text File Database.

A simple database stored as a flat text file.

(C) 2005 Benedict Verhegghe.
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 dunplicat key is encountered.

The default for both is to raise an error. Overriding is done by changing the instance attibute.

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 overriden 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 overriden 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 overriden 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 overriden 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.

6.6.7 `sendmail` — sendmail.py: a simple program to send an email message

(C) 2008 Benedict Verhegghe (benedict.verhegghe@ugent.be) I wrote this software in my free time, for my joy, not as a commissioned task. Any copyright claims made by my employer should therefore be considered void.

Distributed under the GNU General Public License, version 3 or later

`Classes defined in module sendmail`

`Functions defined in module sendmail`

sendmail.**message**(*sender=''*, *to=''*, *cc=''*, *subject=''*, *text=''*)
Create an email message

'to' and 'cc' can be lists of email addresses.

sendmail.**sendmail**(*message*, *sender*, *to*, *serverURL='localhost'*)
Send an email message

'message' is an email message (e.g. returned by message()) 'sender' is a single mail address 'to' can be a list of addresses

6.6.8 `timer` — A timer class.

`Classes defined in module timer`

class timer.**Timer**(*start=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.

Parameters:

- *start*: a datetime object. If not specified, the time of the creation of the Timer is used.

reset(*start=None*)
(Re)Start the timer.

Sets the start time of the timer to the specified value, or to the current time by default.

Parameters:

- *start*: a datetime object. If not specified, the current time as returned by datetime.now() is used.

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=True*)
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.

`Functions defined in module timer`

CHAPTER

SEVEN

PYFORMEX FAQ ‘N TRICKS

Date October 15, 2013

Version 0.9.1

Author Benedict Verhegghe <benedict.verhegghe@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 toi 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
     ```

     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 prefered 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
  ```

CHAPTER

EIGHT

PYFORMEX FILE FORMATS

Date October 15, 2013

Version 0.9.1

Author Benedict Verhegghe <benedict.verhegghe@ugent.be>

Abstract

This document describes the native file formats used by pyFormex. The 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 prefered 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 evend edited by humans. You may also wish to save data in this format to make them accessible 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 used 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 prefered 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 1x4x3 = 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*3 = 6 entries in the connectivity table, and finally 2 property numbers.

CHAPTER

NINE

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 avaliable 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=1
  ```

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 occaional 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 `xfwm4`. 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 ne 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 teminal. 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
```

CHAPTER

TEN

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>.

CHAPTER

ELEVEN

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 Verhegghe, 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.

PYTHON MODULE INDEX

a

actors, 264
adjacency, 175
appMenu, 305
arraytools, 120

c

calpy_itf, 310
camera, 295
cameratools, 312
canvas, 276
ccxdat, 312
ccxinp, 313
collection, 491
colors, 156
colorscale, 263
config, 491
connectivity, 163
coords, 83
curve, 314

d

datareader, 385
decors, 269
draw, 143
dxf, 385

e

elements, 179
export, 388

f

fe, 388
fe_abq, 397
fe_post, 409
fileread, 233
filewrite, 234
flatkeydb, 494
flavia, 411
formex, 104

g

geometry, 158
geomtools, 223
gluttext, 275

i

image, 299
imagearray, 302
imageViewer, 302
inertia, 412
isopar, 413
isosurface, 414

l

lima, 414

m

marks, 274
menu, 258
mesh, 182
mydict, 487

n

neu_exp, 415
nurbs, 416

o

objects, 422
odict, 489
olist, 486

p

partition, 426
plot2d, 427
polygon, 427
polynomial, 432
postproc, 433
project, 210
properties, 434
pyformex_gts, 440

s

script, 139
section2d, 441
sectionize, 443
sendmail, 497
simple, 206

t

tetgen, 443
timer, 497
toolbar, 309
tools, 446
trisurface, 447
turtle, 479

u

units, 481
utils, 212

v

vascularsweepingmesher, 482
viewport, 284

w

webgl, 483
widgets, 236

INDEX

A

abq_eltype() (in module ccxinp), 313
AbqData (class in fe_abq), 399
abqInputNames() (in module fe_abq), 400
accept_draw() (viewport.QtCanvas method), 286
accept_drawing() (viewport.QtCanvas method), 287
accept_selection() (viewport.QtCanvas method), 285
acceptData() (widgets.InputDialog method), 248
acceptData() (widgets.ListSelection method), 253
ack() (in module draw), 143
ack() (in module script), 140
action() (appMenu.AppMenu method), 307
action() (menu.BaseMenu method), 258
action() (menu.Menu method), 260
action() (menu.MenuBar method), 261
ActionList (class in menu), 262
actionList() (appMenu.AppMenu method), 307
actionList() (menu.BaseMenu method), 258
actionList() (menu.Menu method), 259
actionList() (menu.MenuBar method), 261
actionsLike() (appMenu.AppMenu method), 307
actionsLike() (menu.BaseMenu method), 258
actionsLike() (menu.Menu method), 260
actionsLike() (menu.MenuBar method), 261
activate() (canvas.CanvasSettings method), 277
Actor (class in actors), 264
actor() (nurbs.Coords4 method), 418
actor() (nurbs.NurbsCurve method), 419
actor() (nurbs.NurbsSurface method), 420
ActorList (class in canvas), 276
actors (module), 264
add() (appMenu.AppMenu method), 307
add() (canvas.ActorList method), 276
add() (collection.Collection method), 491
add() (menu.ActionList method), 262
Add() (units.UnitsSystem method), 481
add() (webgl.WebGL method), 484
add_group() (widgets.InputDialog method), 248
add_group() (widgets.ListSelection method), 253
add_input() (widgets.InputDialog method), 248
add_input() (widgets.ListSelection method), 253
add_items() (widgets.InputDialog method), 247
add_items() (widgets.ListSelection method), 252
add_tab() (widgets.InputDialog method), 247
add_tab() (widgets.ListSelection method), 252
addActionButtons() (in module toolbar), 309
addActionButtons() (in module widgets), 257
addActor() (canvas.Canvas method), 280
addActor() (viewport.QtCanvas method), 290
addActor() (webgl.WebGL method), 485
addAnnotation() (canvas.Canvas method), 280
addAnnotation() (viewport.QtCanvas method), 290
addAny() (canvas.Canvas method), 280
addAny() (viewport.QtCanvas method), 290
addAxis() (in module arraytools), 125
AddBoundaryLoads() (calpy_itf.QuadInterpolator method), 311
addButton() (in module toolbar), 309
addCameraButtons() (in module toolbar), 310
addCheck() (widgets.MessageBox method), 254
addDecoration() (canvas.Canvas method), 280
addDecoration() (viewport.QtCanvas method), 290
added() (utils.DictDiff method), 213
addElem() (in module tetgen), 444
AddElements() (calpy_itf.QuadInterpolator method), 311
addFeResult() (in module ccxdat), 312
addHighlight() (canvas.Canvas method), 280
addHighlight() (viewport.QtCanvas method), 290
addMaterial() (properties.ElemSection method), 436
addMeanNodes() (mesh.Mesh method), 192
addMeanNodes() (trisurface.TriSurface method), 469
addNodes() (mesh.Mesh method), 192
addNodes() (trisurface.TriSurface method), 469
addNoise() (coords.Coords method), 96

addNoise() (curve.Arc method), 372
addNoise() (curve.Arc3 method), 366
addNoise() (curve.BezierSpline method), 339
addNoise() (curve.CardinalSpline method), 346
addNoise() (curve.CardinalSpline2 method), 353
addNoise() (curve.Curve method), 317
addNoise() (curve.Line method), 329
addNoise() (curve.NaturalSpline method), 359
addNoise() (curve.PolyLine method), 324
addNoise() (curve.Spiral method), 378
addNoise() (fe.FEModel method), 393
addNoise() (fe.Model method), 389
addNoise() (formex.Formex method), 113
addNoise() (geometry.Geometry method), 162
addNoise() (mesh.Mesh method), 200
addNoise() (polygon.Polygon method), 428
addNoise() (trisurface.TriSurface method), 457
addRule() (lima.Lima method), 414
addScene() (webgl.WebGL method), 484
addSection() (properties.ElemSection method), 436
addTimeOut() (in module widgets), 256
addTimeoutButton() (in module toolbar), 310
addView() (viewport.MultiCanvas method), 294
addView() (viewport.NewiMultiCanvas method), 294
addViewport() (built-in function), 52
addViewport() (in module draw), 155
Adjacency (class in adjacency), 175
adjacency (module), 175
adjacency() (connectivity.Connectivity method), 169
adjacency() (mesh.Mesh method), 187
adjacency() (trisurface.TriSurface method), 465
adjacencyArrays() (in module connectivity), 175
affine() (coords.Coords method), 91
affine() (curve.Arc method), 372
affine() (curve.Arc3 method), 366
affine() (curve.BezierSpline method), 339
affine() (curve.CardinalSpline method), 346
affine() (curve.CardinalSpline2 method), 353
affine() (curve.Curve method), 317
affine() (curve.Line method), 329
affine() (curve.NaturalSpline method), 359
affine() (curve.PolyLine method), 324
affine() (curve.Spiral method), 378
affine() (fe.FEModel method), 393
affine() (fe.Model method), 389
affine() (formex.Formex method), 114
affine() (geometry.Geometry method), 161
affine() (mesh.Mesh method), 200
affine() (polygon.Polygon method), 428
affine() (trisurface.TriSurface method), 457
align() (coords.Coords method), 90
align() (curve.Arc method), 372
align() (curve.Arc3 method), 366
align() (curve.BezierSpline method), 340
align() (curve.CardinalSpline method), 346
align() (curve.CardinalSpline2 method), 353
align() (curve.Curve method), 317
align() (curve.Line method), 329
align() (curve.NaturalSpline method), 360
align() (curve.PolyLine method), 324
align() (curve.Spiral method), 379
align() (fe.FEModel method), 393
align() (fe.Model method), 389
align() (formex.Formex method), 114
align() (geometry.Geometry method), 160
align() (in module coords), 103
align() (mesh.Mesh method), 200
align() (polygon.Polygon method), 428
align() (trisurface.TriSurface method), 457
all_image_extensions() (in module utils), 215
Amplitude (class in properties), 437
an() (in module turtle), 480
angles() (polygon.Polygon method), 427
annotate() (in module draw), 152
anyPerpendicularVector() (in module geomtools), 226
anyVector() (in module arraytools), 123
append() (coords.Coords method), 98
append() (curve.Line method), 337
append() (curve.PolyLine method), 323
append() (flatkeydb.FlatDB method), 495
append() (formex.Formex method), 107
append() (objects.DrawableObjects method), 425
append() (objects.Objects method), 422
append() (trisurface.TriSurface method), 448
AppMenu (class in appMenu), 305
appMenu (module), 305
approx() (curve.Arc method), 378
approx() (curve.Arc3 method), 369
approx() (curve.BezierSpline method), 343
approx() (curve.CardinalSpline method), 349
approx() (curve.CardinalSpline2 method), 357
approx() (curve.Curve method), 316
approx() (curve.Line method), 333
approx() (curve.NaturalSpline method), 363
approx() (curve.PolyLine method), 327
approx() (curve.Spiral method), 382
approx() (nurbs.NurbsCurve method), 419
approx_by_subdivision() (curve.BezierSpline

method), 339
approx_by_subdivision() (curve.CardinalSpline method), 352
approximate() (curve.Arc method), 377
approximate() (curve.Arc3 method), 371
approximate() (curve.BezierSpline method), 344
approximate() (curve.CardinalSpline method), 351
approximate() (curve.CardinalSpline2 method), 358
approximate() (curve.Curve method), 317
approximate() (curve.Line method), 336
approximate() (curve.NaturalSpline method), 364
approximate() (curve.PolyLine method), 322
approximate() (curve.Spiral method), 383
apt() (coords.Coords method), 85
Arc (class in curve), 372
arc() (dxf.DxfExporter method), 386
arc2points() (in module curve), 384
Arc3 (class in curve), 365
arccosd() (in module arraytools), 121
arcsind() (in module arraytools), 120
arctand() (in module arraytools), 121
arctand2() (in module arraytools), 121
area() (mesh.Mesh method), 200
area() (polygon.Polygon method), 428
area() (trisurface.TriSurface method), 477
areaNormals() (in module geomtools), 223
areaNormals() (trisurface.TriSurface method), 448
areas() (formex.Formex method), 113
areas() (mesh.Mesh method), 199
areas() (trisurface.TriSurface method), 448
argNearestValue() (in module arraytools), 132
ArrayModel (class in widgets), 250
arraytools (module), 120
asArray() (formex.Formex method), 107
asFormex() (formex.Formex method), 107
asFormexWithProp() (formex.Formex method), 107
ask() (in module draw), 143
ask() (in module script), 140
ask() (objects.DrawableObjects method), 424
ask() (objects.Objects method), 423
ask1() (objects.DrawableObjects method), 425
ask1() (objects.Objects method), 423
askDirname() (in module draw), 145
askFilename() (in module draw), 145
askItems() (in module draw), 144
askNewFilename() (in module draw), 145
aspectRatio() (trisurface.TriSurface method), 450
asPoints() (formex.Formex method), 108
Assemble() (calpy_itf.QuadInterpolator method), 311
atLength() (curve.Line method), 336
atLength() (curve.PolyLine method), 322
atoms() (polynomial.Polynomial method), 433
autorun, 63
autoSaveOn() (in module image), 301
average() (coords.Coords method), 86
averageNormals() (in module geomtools), 225
avgDirections() (curve.Line method), 336
avgDirections() (curve.PolyLine method), 322
avgNodes() (mesh.Mesh method), 191
avgNodes() (trisurface.TriSurface method), 469
avgVertexNormals() (trisurface.TriSurface method), 448
AxesActor (class in actors), 266
AxesMark (class in marks), 274

B

baryCoords() (in module geomtools), 232
BaseMenu (class in menu), 258
bbox() (actors.Actor method), 264
bbox() (coords.Coords method), 85
bbox() (in module coords), 101
bbox() (nurbs.Coords4 method), 417
bbox() (nurbs.NurbsCurve method), 418
bbox() (nurbs.NurbsSurface method), 419
BboxActor (class in actors), 266
bboxes() (coords.Coords method), 87
bboxes() (mesh.Mesh method), 184
bboxes() (trisurface.TriSurface method), 462
bboxIntersection() (in module coords), 101
begin_2D_drawing() (canvas.Canvas method), 280
begin_2D_drawing() (viewport.QtCanvas method), 290
BezierSpline (class in curve), 337
bgcolor() (in module draw), 152
boolean() (in module pyformex_gts), 441
boolean() (trisurface.TriSurface method), 456
border() (trisurface.TriSurface method), 449
borderEdgeNrs() (trisurface.TriSurface method), 449
borderEdges() (trisurface.TriSurface method), 449
borderNodeNrs() (trisurface.TriSurface method), 449
BoundaryInterpolationMatrix() (calpy_itf.QuadInterpolator method), 312
boxes() (coords.Coords method), 97
boxes() (in module simple), 209
breakpt() (in module script), 140
bsphere() (coords.Coords method), 87

buildFilename() (in module utils), 215
bump() (coords.Coords method), 93
bump() (curve.Arc method), 372
bump() (curve.Arc3 method), 366
bump() (curve.BezierSpline method), 340
bump() (curve.CardinalSpline method), 346
bump() (curve.CardinalSpline2 method), 353
bump() (curve.Curve method), 317
bump() (curve.Line method), 330
bump() (curve.NaturalSpline method), 360
bump() (curve.PolyLine method), 324
bump() (curve.Spiral method), 379
bump() (fe.FEModel method), 393
bump() (fe.Model method), 389
bump() (formex.Formex method), 114
bump() (geometry.Geometry method), 161
bump() (mesh.Mesh method), 200
bump() (polygon.Polygon method), 428
bump() (trisurface.TriSurface method), 457
bump1() (coords.Coords method), 93
bump1() (curve.Arc method), 372
bump1() (curve.Arc3 method), 366
bump1() (curve.BezierSpline method), 340
bump1() (curve.CardinalSpline method), 346
bump1() (curve.CardinalSpline2 method), 353
bump1() (curve.Curve method), 318
bump1() (curve.Line method), 330
bump1() (curve.NaturalSpline method), 360
bump1() (curve.PolyLine method), 324
bump1() (curve.Spiral method), 379
bump1() (fe.FEModel method), 393
bump1() (fe.Model method), 389
bump1() (formex.Formex method), 114
bump1() (geometry.Geometry method), 161
bump1() (mesh.Mesh method), 200
bump1() (polygon.Polygon method), 428
bump1() (trisurface.TriSurface method), 457
bump2() (coords.Coords method), 93
bump2() (curve.Arc method), 372
bump2() (curve.Arc3 method), 366
bump2() (curve.BezierSpline method), 340
bump2() (curve.CardinalSpline method), 346
bump2() (curve.CardinalSpline2 method), 353
bump2() (curve.Curve method), 318
bump2() (curve.Line method), 330
bump2() (curve.NaturalSpline method), 360
bump2() (curve.PolyLine method), 324
bump2() (curve.Spiral method), 379
bump2() (fe.FEModel method), 393
bump2() (fe.Model method), 390
bump2() (formex.Formex method), 114
bump2() (geometry.Geometry method), 161
bump2() (mesh.Mesh method), 200
bump2() (polygon.Polygon method), 428
bump2() (trisurface.TriSurface method), 458
ButtonBox (class in widgets), 255

C

calpy_itf (module), 310
Camera (class in camera), 295
camera (module), 295
camera() (webgl.WebGL method), 485
cameratools (module), 312
cancel_draw() (viewport.QtCanvas method), 286
cancel_drawing() (viewport.QtCanvas method), 287
cancel_selection() (viewport.QtCanvas method), 285
Canvas (class in canvas), 278
canvas (module), 276
CanvasMouseHandler (class in viewport), 284
CanvasSettings (class in canvas), 276
canvasSize() (in module draw), 153
CardinalSpline (class in curve), 345
CardinalSpline2 (class in curve), 353
cascade() (in module mydict), 489
cclip() (formex.Formex method), 110
cclip() (mesh.Mesh method), 198
cclip() (trisurface.TriSurface method), 476
ccxdat (module), 312
ccxinp (module), 313
CDict (class in mydict), 488
cellType() (widgets.ArrayModel method), 250
cellType() (widgets.TableModel method), 249
center() (coords.Coords method), 86
center() (in module inertia), 412
centered() (coords.Coords method), 90
centered() (curve.Arc method), 373
centered() (curve.Arc3 method), 366
centered() (curve.BezierSpline method), 340
centered() (curve.CardinalSpline method), 346
centered() (curve.CardinalSpline2 method), 354
centered() (curve.Curve method), 318
centered() (curve.Line method), 330
centered() (curve.NaturalSpline method), 360
centered() (curve.PolyLine method), 324
centered() (curve.Spiral method), 379
centered() (fe.FEModel method), 394
centered() (fe.Model method), 390
centered() (formex.Formex method), 114
centered() (geometry.Geometry method), 160
centered() (mesh.Mesh method), 200

centered() (polygon.Polygon method), 428
centered() (trisurface.TriSurface method), 458
centerline() (in module sectionize), 443
centroid() (coords.Coords method), 86
centroids() (formex.Formex method), 106
centroids() (in module inertia), 412
centroids() (mesh.Mesh method), 184
centroids() (trisurface.TriSurface method), 462
changeBackgroundColorXPM() (in module image), 301
changed() (utils.DictDiff method), 213
changeExt() (in module utils), 215
changeLayout() (viewport.MultiCanvas method), 295
changeLayout() (viewport.NewiMultiCanvas method), 293
changeSize() (viewport.QtCanvas method), 284
changeValues() (objects.DrawableObjects method), 425
changeValues() (objects.Objects method), 423
chdir() (in module script), 141
check() (in module calpy_itf), 312
check() (objects.DrawableObjects method), 425
check() (objects.Objects method), 423
check() (trisurface.TriSurface method), 454
checkArray() (in module arraytools), 127
checkArray1D() (in module arraytools), 127
checkArrayOrIdValue() (in module properties), 440
checkBorder() (trisurface.TriSurface method), 449
checkDict() (canvas.CanvasSettings class method), 277
checkFloat() (in module arraytools), 127
checkIdValue() (in module properties), 440
checkImageFormat() (in module image), 299
checkInt() (in module arraytools), 127
checkKeys() (flatkeydb.FlatDB method), 495
checkPrintSyntax() (in module script), 140
checkRecord() (flatkeydb.FlatDB method), 495
checkRevision() (in module script), 142
checkSelfIntersectionsWithTetgen() (in module tetgen), 446
checkString() (in module properties), 440
checkUniqueNumbers() (in module arraytools), 127
checkWorkdir() (in module draw), 145
circle() (in module curve), 384
circle() (in module simple), 207
circulize() (formex.Formex method), 110
circulize1() (formex.Formex method), 110
classify() (in module appMenu), 309
clear() (canvas.Canvas method), 279
clear() (in module draw), 153
clear() (objects.DrawableObjects method), 425
clear() (objects.Objects method), 423
clear() (viewport.QtCanvas method), 289
clear_canvas() (in module draw), 153
clip() (formex.Formex method), 110
clip() (mesh.Mesh method), 198
clip() (trisurface.TriSurface method), 476
clipAtPlane() (mesh.Mesh method), 199
clipAtPlane() (trisurface.TriSurface method), 476
close() (curve.Line method), 334
close() (curve.PolyLine method), 321
close() (dxf.DxfExporter method), 386
closeDialog() (in module draw), 143
closeGui() (in module draw), 143
closest() (in module geomtools), 224
closestColorName() (in module colors), 158
closestPair() (in module geomtools), 224
closestToPoint() (coords.Coords method), 88
coarsen() (trisurface.TriSurface method), 455
collectByType() (in module dxf), 387
Collection (class in collection), 491
collection (module), 491
collectOnLength() (in module olist), 487
color() (colorscale.ColorLegend method), 264
color() (colorscale.ColorScale method), 263
colorCut() (in module partition), 426
colorindex() (in module draw), 152
ColorLegend (class in colorscale), 263
ColorLegend (class in decors), 270
colormap() (in module draw), 152
colorName() (in module colors), 157
colors (module), 156
ColorScale (class in colorscale), 263
colorscale (module), 263
columnCount() (widgets.ArrayModel method), 250
columnCount() (widgets.TableModel method), 249
colWidths() (widgets.Table method), 250
combine() (connectivity.Connectivity method), 171
compact() (mesh.Mesh method), 190
compact() (trisurface.TriSurface method), 468
complement() (in module arraytools), 130
computeAveragedNodalStresses() (in module ccxdat), 313
computeSection() (properties.ElemSection method), 436
concat() (in module arraytools), 126

concatenate() (coords.Coords class method), 99
concatenate() (formex.Formex class method), 108
concatenate() (in module olist), 487
concatenate() (mesh.Mesh class method), 198
concatenate() (trisurface.TriSurface class method), 475
Config (class in config), 492
config (module), 491
connect() (in module formex), 117
connect() (mesh.Mesh method), 196
connect() (trisurface.TriSurface method), 473
connectCurves() (in module simple), 209
connectedElements() (trisurface.TriSurface method), 452
connectedLineElems() (in module connectivity), 174
connectedTo() (connectivity.Connectivity method), 168
connectedTo() (mesh.Mesh method), 192
connectedTo() (trisurface.TriSurface method), 470
Connectivity (class in connectivity), 163
connectivity (module), 163
connectPoints() (in module sectionize), 443
convert() (mesh.Mesh method), 193
convert() (project.Project method), 212
convert() (trisurface.TriSurface method), 471
convertDXF() (in module dxf), 387
convertFormexToCurve() (in module curve), 385
convertInp() (in module fileread), 233
convertInputItem() (in module widgets), 256
convertPrintSyntax() (in module script), 140
convertRandom() (mesh.Mesh method), 194
convertRandom() (trisurface.TriSurface method), 471
convertUnits() (in module units), 482
coord() (formex.Formex method), 105
Coordinates() (fe_post.FeResult method), 410
CoordPlaneActor (class in actors), 267
Coords (class in coords), 84
coords (module), 83
Coords4 (class in nurbs), 416
CoordsBox (class in widgets), 255
CoordSystem (class in properties), 437
copy() (curve.Arc method), 376
copy() (curve.Arc3 method), 370
copy() (curve.BezierSpline method), 344
copy() (curve.CardinalSpline method), 350
copy() (curve.CardinalSpline2 method), 358
copy() (curve.Curve method), 320
copy() (curve.Line method), 334
copy() (curve.NaturalSpline method), 364
copy() (curve.PolyLine method), 328
copy() (curve.Spiral method), 383
copy() (fe.FEModel method), 396
copy() (fe.Model method), 392
copy() (formex.Formex method), 117
copy() (geometry.Geometry method), 160
copy() (mesh.Mesh method), 203
copy() (polygon.Polygon method), 431
copy() (trisurface.TriSurface method), 461
cosd() (in module arraytools), 120
cosd() (in module turtle), 480
countLines() (in module utils), 219
cpAllSections() (in module vascularsweepingmesher), 483
cpBoundaryLayer() (in module vascularsweepingmesher), 482
cpOneSection() (in module vascularsweepingmesher), 483
cpQuarterLumen() (in module vascularsweepingmesher), 482
cpStackQ16toH64() (in module vascularsweepingmesher), 483
create_insert_action() (appMenu.AppMenu method), 308
create_insert_action() (menu.BaseMenu method), 259
create_insert_action() (menu.Menu method), 260
create_insert_action() (menu.MenuBar method), 262
createAppMenu() (in module appMenu), 309
createBackground() (canvas.Canvas method), 279
createBackground() (viewport.QtCanvas method), 289
createFeResult() (in module flavia), 412
createHistogram() (in module plot2d), 427
createMenuData() (in module menu), 263
createMovie() (in module image), 301
createResultDB() (in module ccxdat), 312
createSegments() (in module sectionize), 443
createView() (in module draw), 152
createView() (viewport.NewiMultiCanvas method), 294
cselect() (formex.Formex method), 108
cselect() (mesh.Mesh method), 191
cselect() (trisurface.TriSurface method), 469
Cube() (in module trisurface), 479
CubeActor (class in actors), 265
cubicEquation() (in module arraytools), 128
cuboid() (in module simple), 210
currentDialog() (in module draw), 145
CursorShapeHandler (class in viewport), 284

curvature() (in module trisurface), 478
curvature() (trisurface.TriSurface method), 448
Curve (class in curve), 314
curve (module), 314
curveToNurbs() (in module nurbs), 422
cut2AtPlane() (in module formex), 119
cut3AtPlane() (in module formex), 119
cutElements3AtPlane() (in module formex), 119
cutWithPlane() (curve.Line method), 336
cutWithPlane() (curve.PolyLine method), 323
cutWithPlane() (formex.Formex method), 112
cutWithPlane() (trisurface.TriSurface method), 452
cutWithPlane1() (trisurface.TriSurface method), 452
cylinder() (in module simple), 209
cylindrical() (coords.Coords method), 92
cylindrical() (curve.Arc method), 373
cylindrical() (curve.Arc3 method), 366
cylindrical() (curve.BezierSpline method), 340
cylindrical() (curve.CardinalSpline method), 346
cylindrical() (curve.CardinalSpline2 method), 354
cylindrical() (curve.Curve method), 318
cylindrical() (curve.Line method), 330
cylindrical() (curve.NaturalSpline method), 360
cylindrical() (curve.PolyLine method), 324
cylindrical() (curve.Spiral method), 379
cylindrical() (fe.FEModel method), 394
cylindrical() (fe.Model method), 390
cylindrical() (formex.Formex method), 114
cylindrical() (geometry.Geometry method), 161
cylindrical() (mesh.Mesh method), 200
cylindrical() (polygon.Polygon method), 428
cylindrical() (trisurface.TriSurface method), 458

D

DAction (class in menu), 262
data() (widgets.TableModel method), 249
Database (class in properties), 434
datareader (module), 385
deCasteljou() (in module nurbs), 421
decompose() (nurbs.NurbsCurve method), 418
decorate() (in module draw), 152
Decoration (class in decors), 269
decors (module), 269
defaultItemType() (in module widgets), 256
degenerate() (in module geomtools), 223
degenerate() (trisurface.TriSurface method), 451
degree() (polynomial.Polynomial method), 432
degrees() (polynomial.Polynomial method), 432
delaunay() (in module polygon), 432
delay() (in module draw), 153
delete() (canvas.ActorList method), 276
delete() (project.Project method), 212
delProp() (objects.DrawableObjects method), 425
delProp() (properties.PropertyDB method), 439
deNormalize() (nurbs.Coords4 method), 417
derivs() (nurbs.NurbsCurve method), 418
derivs() (nurbs.NurbsSurface method), 419
detect() (in module calpy_itf), 312
dialogAccepted() (in module draw), 145
dialogRejected() (in module draw), 145
dialogTimedOut() (in module draw), 145
dicom2numpy() (in module imagearray), 305
Dict (class in mydict), 487
DictDiff (class in utils), 213
difference() (in module olist), 486
directionalExtremes() (coords.Coords method), 88
directionalSize() (coords.Coords method), 88
directionalWidth() (coords.Coords method), 89
directions() (curve.Line method), 335
directions() (curve.PolyLine method), 322
directionsAt() (curve.Arc method), 375
directionsAt() (curve.Arc3 method), 369
directionsAt() (curve.BezierSpline method), 342
directionsAt() (curve.CardinalSpline method), 349
directionsAt() (curve.CardinalSpline2 method), 356
directionsAt() (curve.Curve method), 315
directionsAt() (curve.Line method), 332
directionsAt() (curve.NaturalSpline method), 362
directionsAt() (curve.PolyLine method), 326
directionsAt() (curve.Spiral method), 381
displaceLines() (in module geomtools), 226
Displacements() (fe_post.FeResult method), 410
display() (canvas.Canvas method), 279
display() (viewport.QtCanvas method), 290
distance() (in module geomtools), 224
distanceFromLine() (coords.Coords method), 88
distanceFromPlane() (coords.Coords method), 87
distanceFromPoint() (coords.Coords method), 88
distanceOfPoints() (trisurface.TriSurface method), 450
distancesPFL() (in module geomtools), 231
distancesPFS() (in module geomtools), 231
divide() (formex.Formex method), 112
do_ELEMENT() (in module ccxinp), 313
do_HEADING() (in module ccxinp), 313
do_lighting() (canvas.Canvas method), 279
do_lighting() (viewport.QtCanvas method), 289
do_NODE() (in module ccxinp), 313
do_nothing() (fe_post.FeResult method), 410

do_PART() (in module ccxinp), 313
do_SYSTEM() (in module ccxinp), 313
doFunc() (widgets.InputButton method), 244
dolly() (camera.Camera method), 297
dotpr() (in module arraytools), 121
dotpr() (in module viewport), 295
draw (module), 143
draw() (in module draw), 146
draw_bbox() (in module objects), 426
draw_cursor() (canvas.Canvas method), 282
draw_cursor() (viewport.QtCanvas method), 292
draw_elem_numbers() (in module objects), 426
draw_focus_rectangle() (canvas.Canvas method), 282
draw_focus_rectangle() (viewport.QtCanvas method), 292
draw_free_edges() (in module objects), 426
draw_node_numbers() (in module objects), 426
draw_nodes() (in module objects), 426
draw_object_name() (in module objects), 426
draw_state_line() (viewport.QtCanvas method), 288
draw_state_rect() (viewport.QtCanvas method), 287
drawable() (in module draw), 146
DrawableObjects (class in objects), 424
drawActor() (in module draw), 151
drawAnnotation() (objects.DrawableObjects method), 424
drawAny() (in module draw), 151
drawAxes() (in module draw), 150
drawBbox() (in module draw), 150
drawChanges() (objects.DrawableObjects method), 424
drawCircles() (in module sectionize), 443
drawDot() (in module decors), 273
drawFreeEdges() (in module draw), 149
drawGL() (actors.Actor method), 264
drawGL() (actors.AxesActor method), 266
drawGL() (actors.BboxActor method), 266
drawGL() (actors.CoordPlaneActor method), 267
drawGL() (actors.CubeActor method), 265
drawGL() (actors.GeomActor method), 268
drawGL() (actors.GridActor method), 267
drawGL() (actors.PlaneActor method), 267
drawGL() (actors.SphereActor method), 265
drawGL() (decors.Decoration method), 269
drawGL() (decors.GlutText method), 270
drawGL() (decors.Triade method), 273
drawGL() (marks.Mark method), 274
drawGrid() (in module decors), 273
drawImage() (in module draw), 151
drawImage3D() (in module draw), 150
drawLine() (in module decors), 273
drawLinesInter() (in module draw), 155
drawLinesInter() (viewport.QtCanvas method), 286
drawMarks() (in module draw), 149
drawNumbers() (in module draw), 149
drawpick() (marks.MarkList method), 275
drawPrincipal() (in module draw), 150
drawPropNumbers() (in module draw), 150
drawRect() (in module decors), 273
drawRectangle() (in module decors), 273
drawText() (in module draw), 152
drawText3D() (in module draw), 150
drawVectors() (in module draw), 149
drawVertexNumbers() (in module draw), 150
drawViewportAxes3D() (in module draw), 151
dsize() (coords.Coords method), 87
dualMesh() (trisurface.TriSurface method), 451
dxf (module), 385
DxfExporter (class in dxf), 385
dynapan() (viewport.QtCanvas method), 287
dynarot() (viewport.QtCanvas method), 287
dynazoom() (viewport.QtCanvas method), 287

E

edgeAdjacency() (mesh.Mesh method), 189
edgeAdjacency() (trisurface.TriSurface method), 467
edgeAngles() (trisurface.TriSurface method), 450
edgeConnections() (mesh.Mesh method), 189
edgeConnections() (trisurface.TriSurface method), 467
edgeCosAngles() (trisurface.TriSurface method), 450
edgeDistance() (in module geomtools), 232
EdgeLoad (class in properties), 437
edit_drawing() (viewport.QtCanvas method), 287
editAnnotations() (objects.DrawableObjects method), 424
editFile() (in module draw), 144
egg() (coords.Coords method), 95
egg() (curve.Arc method), 373
egg() (curve.Arc3 method), 366
egg() (curve.BezierSpline method), 340
egg() (curve.CardinalSpline method), 346
egg() (curve.CardinalSpline2 method), 354
egg() (curve.Curve method), 318
egg() (curve.Line method), 330
egg() (curve.NaturalSpline method), 360

egg() (curve.PolyLine method), 324
egg() (curve.Spiral method), 379
egg() (fe.FEModel method), 394
egg() (fe.Model method), 390
egg() (formex.Formex method), 114
egg() (geometry.Geometry method), 161
egg() (mesh.Mesh method), 201
egg() (polygon.Polygon method), 428
egg() (trisurface.TriSurface method), 458
elbbox() (formex.Formex method), 109
element, 21
element() (formex.Formex method), 105
element2str() (formex.Formex class method), 107
elements (module), 179
elementToNodal() (trisurface.TriSurface method), 460
ElementType (class in elements), 179
elementType() (in module elements), 181
elementTypes() (in module elements), 182
ElemLoad (class in properties), 436
elemNrs() (fe.Model method), 389
elemProp() (properties.PropertyDB method), 439
ElemSection (class in properties), 435
elName() (mesh.Mesh method), 183
elName() (trisurface.TriSurface method), 461
elType() (mesh.Mesh method), 183
elType() (trisurface.TriSurface method), 461
emit_cancel() (viewport.QtCanvas method), 287
emit_done() (viewport.QtCanvas method), 287
enable_lighting() (canvas.Canvas method), 278
enable_lighting() (viewport.QtCanvas method), 288
end_2D_drawing() (canvas.Canvas method), 280
end_2D_drawing() (viewport.QtCanvas method), 290
endPoints() (curve.Arc method), 372
endPoints() (curve.Arc3 method), 365
endPoints() (curve.BezierSpline method), 339
endPoints() (curve.CardinalSpline method), 345
endPoints() (curve.CardinalSpline2 method), 353
endPoints() (curve.Curve method), 314
endPoints() (curve.Line method), 329
endPoints() (curve.NaturalSpline method), 359
endPoints() (curve.PolyLine method), 323
endPoints() (curve.Spiral method), 378
endSection() (dxf.DxfExporter method), 386
Engineering() (units.UnitsSystem method), 481
entities() (dxf.DxfExporter method), 386
equal() (utils.DictDiff method), 214
error() (in module draw), 143
error() (in module script), 140
esetName() (in module fe_abq), 400
eval() (polynomial.Polynomial method), 433
evalAtoms() (polynomial.Polynomial method), 433
evalAtoms1() (polynomial.Polynomial method), 432
evaluate() (in module isopar), 413
exit() (in module script), 141
exponents() (in module isopar), 413
export (module), 388
Export() (fe_post.FeResult method), 410
export() (in module script), 139
export() (webgl.WebGL method), 485
export2() (in module script), 139
exportDXF() (in module dxf), 387
exportDxf() (in module dxf), 388
exportDxfText() (in module dxf), 388
exportMesh() (in module fe_abq), 409
exportObjects() (in module tools), 447
exportPGF() (webgl.WebGL method), 485
exportWebGL() (in module draw), 156
extend() (curve.BezierSpline method), 339
extend() (curve.CardinalSpline method), 352
extendedSectionChar() (in module section2d), 442
externalAngles() (polygon.Polygon method), 427
extractCanvasSettings() (in module canvas), 283
extractMeshes() (in module fileread), 233
extrude() (formex.Formex method), 111
extrude() (mesh.Mesh method), 197
extrude() (trisurface.TriSurface method), 474

F

faceDistance() (in module geomtools), 232
fd() (in module turtle), 480
fe (module), 388
fe_abq (module), 397
fe_post (module), 409
featureEdges() (trisurface.TriSurface method), 451
FEModel (class in fe), 393
FeResult (class in fe_post), 409
fforward() (in module draw), 154
fgcolor() (in module draw), 152
fileDescription() (in module utils), 215
fileName() (appMenu.AppMenu method), 306
fileread (module), 233
files() (utils.NameSequence method), 213
FileSelection (class in widgets), 251
fileSize() (in module utils), 216
fileType() (in module utils), 216
fileTypeFromExt() (in module utils), 216
filewrite (module), 234

fill() (polygon.Polygon method), 428
fillBorder() (in module trisurface), 478
fillBorder() (trisurface.TriSurface method), 450
filterFiles() (appMenu.AppMenu method), 306
find_first_nodes() (in module trisurface), 479
find_global() (in module project), 212
find_nodes() (in module trisurface), 479
find_row() (in module trisurface), 479
find_triangles() (in module trisurface), 479
findBisectrixUsingPlanes() (in module vascularsweepingmesher), 482
findConnectedLineElems() (in module connectivity), 172
findIcon() (in module utils), 216
FindListItem() (in module properties), 440
finish_draw() (viewport.QtCanvas method), 286
finish_drawing() (viewport.QtCanvas method), 287
finish_selection() (viewport.QtCanvas method), 285
firstWord() (in module flatkeydb), 496
fixNormals() (trisurface.TriSurface method), 454
fixVolumes() (mesh.Mesh method), 200
fixVolumes() (trisurface.TriSurface method), 477
flags() (widgets.ArrayModel method), 250
flags() (widgets.TableModel method), 250
flare() (coords.Coords method), 94
flare() (curve.Arc method), 373
flare() (curve.Arc3 method), 366
flare() (curve.BezierSpline method), 340
flare() (curve.CardinalSpline method), 346
flare() (curve.CardinalSpline2 method), 354
flare() (curve.Curve method), 318
flare() (curve.Line method), 330
flare() (curve.NaturalSpline method), 360
flare() (curve.PolyLine method), 324
flare() (curve.Spiral method), 379
flare() (fe.FEModel method), 394
flare() (fe.Model method), 390
flare() (formex.Formex method), 114
flare() (geometry.Geometry method), 162
flare() (mesh.Mesh method), 201
flare() (polygon.Polygon method), 429
flare() (trisurface.TriSurface method), 458
FlatDB (class in flatkeydb), 494
flatkeydb (module), 494
flatten() (in module draw), 146
flatten() (in module olist), 487
flavia (module), 411
flyAlong() (in module draw), 154
fmt() (fe_abq.Output method), 398
fmtAnalyticalSurface() (in module fe_abq), 404
fmtBeamSection() (in module fe_abq), 402
fmtCmd() (in module fe_abq), 400
fmtConnectorBehavior() (in module fe_abq), 403
fmtConnectorSection() (in module fe_abq), 403
fmtConstraint() (in module fe_abq), 406
fmtContactPair() (in module fe_abq), 405
fmtDashpot() (in module fe_abq), 403
fmtData() (in module fe_abq), 400
fmtData1D() (in module fe_abq), 400
fmtEquation() (in module fe_abq), 406
fmtFrameSection() (in module fe_abq), 401
fmtGeneralBeamSection() (in module fe_abq), 402
fmtGeneralContact() (in module fe_abq), 405
fmtHeading() (in module fe_abq), 400
fmtInertia() (in module fe_abq), 407
fmtInitialConditions() (in module fe_abq), 406
fmtMass() (in module fe_abq), 407
fmtMaterial() (in module fe_abq), 400
fmtOptions() (in module fe_abq), 400
fmtOrientation() (in module fe_abq), 406
fmtPart() (in module fe_abq), 400
fmtShellSection() (in module fe_abq), 404
fmtSolidSection() (in module fe_abq), 403
fmtSpring() (in module fe_abq), 403
fmtSurface() (in module fe_abq), 404
fmtSurfaceInteraction() (in module fe_abq), 405
fmtTransform() (in module fe_abq), 401
focus() (in module draw), 149
forceReST() (in module utils), 218
forget() (in module script), 139
forget() (objects.DrawableObjects method), 426
forget() (objects.Objects method), 423
forgetAll() (in module script), 139
format_actor() (webgl.WebGL method), 485
format_gui() (webgl.WebGL method), 485
format_gui_controller() (webgl.WebGL method), 485
formatDict() (in module mydict), 489
Formex, 21
Formex (class in formex), 104
formex (module), 104
fprint() (coords.Coords method), 89
FramedGridLayout (class in viewport), 294
frameScale() (in module postproc), 433
frenet() (curve.Arc method), 376
frenet() (curve.Arc3 method), 370
frenet() (curve.BezierSpline method), 343
frenet() (curve.CardinalSpline method), 350
frenet() (curve.CardinalSpline2 method), 357

frenet() (curve.Curve method), 316
frenet() (curve.Line method), 333
frenet() (curve.NaturalSpline method), 363
frenet() (curve.PolyLine method), 328
frenet() (curve.Spiral method), 382
frenet() (in module nurbs), 422
fromfile() (coords.Coords class method), 99
fromfile() (formex.Formex class method), 113
fromstring() (coords.Coords class method), 99
fromstring() (formex.Formex class method), 113
frontFactory() (adjacency.Adjacency method), 177
frontWalk() (adjacency.Adjacency method), 178
frontWalk() (mesh.Mesh method), 188
frontWalk() (trisurface.TriSurface method), 466
fullAppName() (appMenu.AppMenu method), 306
fuse() (coords.Coords method), 97
fuse() (formex.Formex method), 106
fuse() (mesh.Mesh method), 190
fuse() (trisurface.TriSurface method), 467

G

GenericDialog (class in widgets), 253
GeomActor (class in actors), 268
Geometry (class in geometry), 158
geometry (module), 158
geomtools (module), 223
get() (canvas.CanvasSettings method), 278
get() (collection.Collection method), 491
get() (config.Config method), 493
get() (fe_abq.Interaction method), 399
get() (fe_abq.Output method), 398
get() (fe_abq.Result method), 398
get() (mydict.CDict method), 489
get() (mydict.Dict method), 488
get() (properties.Database method), 434
get() (properties.EdgeLoad method), 437
get() (properties.ElemLoad method), 437
get() (properties.ElemSection method), 436
get() (properties.MaterialDB method), 435
get() (properties.PropertyDB method), 438
get() (properties.SectionDB method), 435
Get() (units.UnitsSystem method), 481
getBorder() (mesh.Mesh method), 186
getBorder() (trisurface.TriSurface method), 464
getBorderElems() (mesh.Mesh method), 187
getBorderElems() (trisurface.TriSurface method), 465
getBorderMesh() (mesh.Mesh method), 186
getBorderMesh() (trisurface.TriSurface method), 464
getBorderNodes() (mesh.Mesh method), 187
getBorderNodes() (trisurface.TriSurface method), 465
getCells() (mesh.Mesh method), 186
getCells() (trisurface.TriSurface method), 464
getcfg() (in module script), 140
getCollection() (in module tools), 446
getColor() (in module widgets), 257
getCoords() (curve.Arc method), 375
getCoords() (curve.Arc3 method), 369
getCoords() (curve.BezierSpline method), 343
getCoords() (curve.CardinalSpline method), 349
getCoords() (curve.CardinalSpline2 method), 356
getCoords() (curve.Curve method), 320
getCoords() (curve.Line method), 333
getCoords() (curve.NaturalSpline method), 363
getCoords() (curve.PolyLine method), 327
getCoords() (curve.Spiral method), 382
getCoords() (fe.FEModel method), 396
getCoords() (fe.Model method), 392
getCoords() (formex.Formex method), 117
getCoords() (geometry.Geometry method), 159
getCoords() (mesh.Mesh method), 185
getCoords() (polygon.Polygon method), 431
getCoords() (trisurface.TriSurface method), 463
getDocString() (in module utils), 220
getDrawFaces() (elements.ElementType class method), 181
getEdges() (mesh.Mesh method), 185
getEdges() (trisurface.TriSurface method), 463
getElemEdges() (mesh.Mesh method), 186
getElemEdges() (trisurface.TriSurface method), 447
getElems() (fe.Model method), 389
getElems() (mesh.Mesh method), 185
getElems() (trisurface.TriSurface method), 463
getEntities() (elements.ElementType class method), 181
getFaces() (mesh.Mesh method), 185
getFaces() (trisurface.TriSurface method), 464
getFilename() (widgets.FileSelection method), 251
getFilename() (widgets.ProjectSelection method), 252
getFilename() (widgets.SaveImageDialog method), 252
getFiles() (appMenu.AppMenu method), 306
getFreeEdgesMesh() (mesh.Mesh method), 187
getFreeEdgesMesh() (trisurface.TriSurface method), 465
getFreeEntities() (mesh.Mesh method), 186

getFreeEntities() (trisurface.TriSurface method), 464
getFreeEntitiesMesh() (mesh.Mesh method), 186
getFreeEntitiesMesh() (trisurface.TriSurface method), 464
getIncs() (fe_post.FeResult method), 411
getInts() (in module tetgen), 444
getLowerEntities() (mesh.Mesh method), 185
getLowerEntities() (trisurface.TriSurface method), 463
getMouseFunc() (viewport.CanvasMouseHandler method), 284
getMouseFunc() (viewport.QtCanvas method), 285
getNodes() (mesh.Mesh method), 185
getNodes() (trisurface.TriSurface method), 463
getObjectItems() (in module tools), 446
getParams() (in module fileread), 233
getPartition() (in module tools), 447
getPoints() (mesh.Mesh method), 185
getPoints() (trisurface.TriSurface method), 463
getProp() (formex.Formex method), 106
getProp() (mesh.Mesh method), 183
getProp() (properties.PropertyDB method), 439
getProp() (trisurface.TriSurface method), 461
getres() (fe_post.FeResult method), 411
getResult() (widgets.ListSelection method), 252
getResult() (widgets.MessageBox method), 254
getResults() (widgets.InputDialog method), 248
getResults() (widgets.ListSelection method), 253
getRot() (camera.Camera method), 297
getSize() (viewport.QtCanvas method), 284
getSteps() (fe_post.FeResult method), 411
getValues() (widgets.CoordsBox method), 255
gl_pickbuffer() (in module canvas), 283
GLcolor() (in module colors), 157
glEnable() (in module canvas), 283
glFlat() (in module canvas), 283
glinit() (canvas.Canvas method), 279
glinit() (viewport.QtCanvas method), 289
glLineStipple() (in module canvas), 283
glob() (utils.NameSequence method), 213
globalInterpolationCurve() (in module nurbs), 420
Globals() (in module script), 139
glSmooth() (in module canvas), 283
glupdate() (canvas.Canvas method), 279
glupdate() (viewport.QtCanvas method), 289
glutBitmapLength() (in module gluttext), 276
glutDrawText() (in module gluttext), 276
glutFont() (in module gluttext), 275
glutFontHeight() (in module gluttext), 276
glutRenderText() (in module gluttext), 276
glutSelectFont() (in module gluttext), 275
GlutText (class in decors), 270
gluttext (module), 275
go() (in module turtle), 480
GP2Nodes() (calpy_itf.QuadInterpolator method), 311
gray2qimage() (in module imagearray), 303
grepSource() (in module utils), 217
GREY() (in module colors), 158
Grid (class in decors), 272
GridActor (class in actors), 266
gridpoints() (in module mesh), 205
groupArgmin() (in module arraytools), 133
groupInputItem() (in module widgets), 256
groupPositions() (in module arraytools), 133
growAxis() (in module arraytools), 124
growCollection() (in module tools), 446
growSelection() (mesh.Mesh method), 188
growSelection() (trisurface.TriSurface method), 466
gts_refine() (trisurface.TriSurface method), 455
gts_smooth() (trisurface.TriSurface method), 455
gunzip() (in module utils), 218
gzip() (in module utils), 218

H

has_key() (collection.Collection method), 491
has_lighting() (canvas.Canvas method), 278
has_lighting() (viewport.QtCanvas method), 288
hasAnnotation() (objects.DrawableObjects method), 424
header_data() (project.Project method), 211
headerData() (widgets.ArrayModel method), 250
headerData() (widgets.TableModel method), 249
hex8_els() (in module mesh), 206
hex8_wts() (in module mesh), 206
hicolor() (in module draw), 152
highlight() (canvas.Canvas method), 283
highlight() (viewport.QtCanvas method), 293
highlightActor() (canvas.Canvas method), 282
highlightActor() (in module draw), 155
highlightActor() (viewport.QtCanvas method), 292
highlightActors() (canvas.Canvas method), 282
highlightActors() (viewport.QtCanvas method), 293
highlightEdges() (canvas.Canvas method), 283
highlightEdges() (viewport.QtCanvas method), 293
highlightElements() (canvas.Canvas method), 283

highlightElements() (viewport.QtCanvas method), 293
highlightPartitions() (canvas.Canvas method), 283
highlightPartitions() (viewport.QtCanvas method), 293
highlightPoints() (canvas.Canvas method), 283
highlightPoints() (viewport.QtCanvas method), 293
histogram2() (in module arraytools), 135
hits() (connectivity.Connectivity method), 169
hits() (mesh.Mesh method), 192
hits() (trisurface.TriSurface method), 470
horner() (in module arraytools), 122
hsorted() (in module utils), 220
human() (polynomial.Polynomial method), 433
hyperCylindrical() (curve.Arc method), 373
hyperCylindrical() (curve.Arc3 method), 366
hyperCylindrical() (curve.BezierSpline method), 340
hyperCylindrical() (curve.CardinalSpline method), 346
hyperCylindrical() (curve.CardinalSpline2 method), 354
hyperCylindrical() (curve.Curve method), 318
hyperCylindrical() (curve.Line method), 330
hyperCylindrical() (curve.NaturalSpline method), 360
hyperCylindrical() (curve.PolyLine method), 324
hyperCylindrical() (curve.Spiral method), 379
hyperCylindrical() (fe.FEModel method), 394
hyperCylindrical() (fe.Model method), 390
hyperCylindrical() (formex.Formex method), 114
hyperCylindrical() (geometry.Geometry method), 161
hyperCylindrical() (mesh.Mesh method), 201
hyperCylindrical() (polygon.Polygon method), 429
hyperCylindrical() (trisurface.TriSurface method), 458

I

idraw() (viewport.QtCanvas method), 286
ignore_error() (in module flatkeydb), 497
image (module), 299
image2glcolor() (in module imagearray), 304
image2numpy() (in module imagearray), 302
imagearray (module), 302
imageFormatFromExt() (in module image), 299
imageFormats() (in module image), 299
ImageView (class in widgets), 255
ImageViewer (class in imageViewer), 302
imageViewer (module), 302
importDXF() (in module dxf), 386
Increment() (fe_post.FeResult method), 410
index() (appMenu.AppMenu method), 307
index() (menu.BaseMenu method), 258
index() (menu.Menu method), 260
index() (menu.MenuBar method), 261
inertia (module), 412
inertia() (coords.Coords method), 87
inertia() (in module inertia), 412
inertia() (trisurface.TriSurface method), 449
info() (formex.Formex method), 107
info() (mesh.Mesh method), 184
info() (trisurface.TriSurface method), 462
initialize() (in module image), 299
inputAny() (in module widgets), 256
InputBool (class in widgets), 239
InputButton (class in widgets), 244
InputColor (class in widgets), 245
InputCombo (class in widgets), 240
InputDialog (class in widgets), 246
InputFile (class in widgets), 245
InputFloat (class in widgets), 242
InputFont (class in widgets), 245
InputForm (class in widgets), 246
InputFSlider (class in widgets), 243
InputGroup (class in widgets), 246
InputInfo (class in widgets), 238
InputInteger (class in widgets), 242
InputItem (class in widgets), 236
InputIVector (class in widgets), 244
InputLabel (class in widgets), 238
InputList (class in widgets), 239
InputPoint (class in widgets), 244
InputPush (class in widgets), 241
InputRadio (class in widgets), 241
InputSlider (class in widgets), 243
InputString (class in widgets), 238
InputTab (class in widgets), 246
InputTable (class in widgets), 242
InputText (class in widgets), 239
InputWidget (class in widgets), 245
insert() (flatkeydb.FlatDB method), 495
insert_action() (appMenu.AppMenu method), 308
insert_action() (menu.BaseMenu method), 259
insert_action() (menu.Menu method), 260
insert_action() (menu.MenuBar method), 261
insert_menu() (appMenu.AppMenu method), 308
insert_menu() (menu.BaseMenu method), 259
insert_menu() (menu.Menu method), 260
insert_menu() (menu.MenuBar method), 261

insert_sep() (appMenu.AppMenu method), 308
insert_sep() (menu.BaseMenu method), 259
insert_sep() (menu.Menu method), 260
insert_sep() (menu.MenuBar method), 261
insertItems() (appMenu.AppMenu method), 308
insertItems() (menu.BaseMenu method), 259
insertItems() (menu.Menu method), 260
insertItems() (menu.MenuBar method), 262
insertKnots() (nurbs.NurbsCurve method), 418
insertLevel() (connectivity.Connectivity method), 170
insertPointsAt() (curve.Line method), 337
insertPointsAt() (curve.PolyLine method), 323
insertRows() (widgets.TableModel method), 249
inside() (in module arraytools), 123
inside() (in module pyformex_gts), 441
inside() (trisurface.TriSurface method), 456
insideSimplex() (in module geomtools), 233
insideTriangle() (in module geomtools), 232
Interaction (class in fe_abq), 399
internalAngles() (polygon.Polygon method), 428
International() (units.UnitsSystem method), 481
interpolate() (coords.Coords method), 99
interpolate() (in module formex), 118
InterpolationMatrix() (calpy_itf.QuadInterpolator method), 311
interpoly() (in module isopar), 414
interrogate() (in module utils), 223
intersection() (in module olist), 487
intersection() (in module pyformex_gts), 441
intersection() (trisurface.TriSurface method), 457
intersectionLinesPWP() (in module geomtools), 230
intersectionLinesWithPlane() (in module formex), 118
intersectionPointsLWL() (in module geomtools), 227
intersectionPointsLWP() (in module geomtools), 228
intersectionPointsLWT() (in module geomtools), 229
intersectionPointsPOL() (in module geomtools), 231
intersectionPointsPOP() (in module geomtools), 230
intersectionPointsPWP() (in module geomtools), 230
intersectionPointsSWP() (in module geomtools), 228
intersectionPointsSWT() (in module geomtools), 229
intersectionSphereSphere() (in module geomtools), 231
intersectionSWP() (in module geomtools), 228
intersectionTimesLWL() (in module geomtools), 227
intersectionTimesLWP() (in module geomtools), 227
intersectionTimesLWT() (in module geomtools), 229
intersectionTimesPOL() (in module geomtools), 231
intersectionTimesPOP() (in module geomtools), 230
intersectionTimesSWP() (in module geomtools), 228
intersectionTimesSWT() (in module geomtools), 229
intersectionWithPlane() (formex.Formex method), 112
intersectionWithPlane() (trisurface.TriSurface method), 452
inverse() (connectivity.Connectivity method), 168
inverseIndex() (in module arraytools), 132
inverseUniqueIndex() (in module arraytools), 130
is_pyFormex() (in module utils), 220
is_script() (in module utils), 220
isClose() (in module arraytools), 123
isClosedManifold() (trisurface.TriSurface method), 449
isConvex() (polygon.Polygon method), 427
isInt() (in module arraytools), 120
isManifold() (trisurface.TriSurface method), 449
Isopar (class in isopar), 413
isopar (module), 413
isopar() (coords.Coords method), 96
isopar() (curve.Arc method), 373
isopar() (curve.Arc3 method), 367
isopar() (curve.BezierSpline method), 340
isopar() (curve.CardinalSpline method), 347
isopar() (curve.CardinalSpline2 method), 354
isopar() (curve.Curve method), 318
isopar() (curve.Line method), 330
isopar() (curve.NaturalSpline method), 360
isopar() (curve.PolyLine method), 324
isopar() (curve.Spiral method), 379
isopar() (fe.FEModel method), 394
isopar() (fe.Model method), 390
isopar() (formex.Formex method), 114
isopar() (geometry.Geometry method), 162
isopar() (mesh.Mesh method), 201
isopar() (polygon.Polygon method), 429

isopar() (trisurface.TriSurface method), 458
isosurface (module), 414
isosurface() (in module isosurface), 414
isWritable() (in module script), 141
item() (appMenu.AppMenu method), 307
item() (menu.BaseMenu method), 258
item() (menu.Menu method), 260
item() (menu.MenuBar method), 261
items() (collection.Collection method), 491
items() (odict.KeyedList method), 490
items() (odict.ODict method), 490
iteritems() (odict.KeyedList method), 490
iteritems() (odict.ODict method), 490

K

keep() (objects.DrawableObjects method), 426
keep() (objects.Objects method), 423
key_error_handler() (flatkeydb.FlatDB method), 495
KeyedList (class in odict), 490
keys() (collection.Collection method), 491
keys() (config.Config method), 493
keys() (odict.KeyedList method), 490
keys() (odict.ODict method), 490
killProcesses() (in module utils), 220
knotPoints() (nurbs.NurbsCurve method), 418
knotVector() (in module nurbs), 420

L

largestByConnection() (mesh.Mesh method), 188
largestByConnection() (trisurface.TriSurface method), 466
layer() (dxf.DxfExporter method), 386
layout() (built-in function), 52
layout() (in module draw), 155
length() (curve.Arc method), 376
length() (curve.Arc3 method), 369
length() (curve.BezierSpline method), 343
length() (curve.CardinalSpline method), 349
length() (curve.CardinalSpline2 method), 357
length() (curve.Curve method), 316
length() (curve.Line method), 333
length() (curve.NaturalSpline method), 363
length() (curve.PolyLine method), 327
length() (curve.Spiral method), 382
length() (in module arraytools), 122
length() (in module viewport), 295
length() (mesh.Mesh method), 199
length() (trisurface.TriSurface method), 476
length_intgrnd() (curve.BezierSpline method), 338
length_intgrnd() (curve.CardinalSpline method), 352
lengths() (curve.BezierSpline method), 338
lengths() (curve.CardinalSpline method), 352
lengths() (curve.Line method), 336
lengths() (curve.PolyLine method), 322
lengths() (formex.Formex method), 113
lengths() (mesh.Mesh method), 199
lengths() (trisurface.TriSurface method), 476
level() (curve.Arc method), 376
level() (curve.Arc3 method), 370
level() (curve.BezierSpline method), 343
level() (curve.CardinalSpline method), 350
level() (curve.CardinalSpline2 method), 357
level() (curve.Curve method), 320
level() (curve.Line method), 333
level() (curve.NaturalSpline method), 363
level() (curve.PolyLine method), 328
level() (curve.Spiral method), 382
level() (fe.FEModel method), 396
level() (fe.Model method), 392
level() (formex.Formex method), 106
level() (geometry.Geometry method), 160
level() (polygon.Polygon method), 431
levelVolumes() (in module geomtools), 223
levelVolumes() (mesh.Mesh method), 199
levelVolumes() (trisurface.TriSurface method), 476
lights() (in module draw), 153
Lima (class in lima), 414
lima (module), 414
lima() (in module lima), 415
Line (class in curve), 329
Line (class in decors), 270
line() (dxf.DxfExporter method), 386
line() (in module simple), 207
LineDrawing (class in decors), 272
lineIntersection() (in module geomtools), 226
linestipple() (in module draw), 153
linewidth() (in module draw), 153
link() (viewport.MultiCanvas method), 295
link() (viewport.NewiMultiCanvas method), 294
linkViewport() (built-in function), 52
linkViewport() (in module draw), 155
listAll() (in module script), 139
listAll() (objects.DrawableObjects method), 425
listAll() (objects.Objects method), 423
listDegenerate() (connectivity.Connectivity method), 165
listDuplicate() (connectivity.Connectivity method), 166
listFontFiles() (in module utils), 222
listNonDegenerate() (connectivity.Connectivity

method), 165
ListSelection (class in widgets), 252
listTree() (in module utils), 217
listUnique() (connectivity.Connectivity method), 166
ListWidget (class in widgets), 248
load() (project.Project method), 212
loadCurrentRotation() (camera.Camera method), 298
loadFiles() (appMenu.AppMenu method), 306
loadImage_dicom() (in module imagearray), 304
loadImage_gdcm() (in module imagearray), 304
loadModelView() (camera.Camera method), 297
loadProjection() (camera.Camera method), 298
localParam() (curve.Arc method), 372
localParam() (curve.Arc3 method), 366
localParam() (curve.BezierSpline method), 339
localParam() (curve.CardinalSpline method), 346
localParam() (curve.CardinalSpline2 method), 353
localParam() (curve.Curve method), 315
localParam() (curve.Line method), 329
localParam() (curve.NaturalSpline method), 359
localParam() (curve.PolyLine method), 323
localParam() (curve.Spiral method), 378
lock() (camera.Camera method), 297
longestEdge() (trisurface.TriSurface method), 450
lpattern() (in module formex), 118
luminance() (in module colors), 157

M

makeEditable() (widgets.ArrayModel method), 250
makeEditable() (widgets.TableModel method), 249
map() (coords.Coords method), 94
map() (curve.Arc method), 373
map() (curve.Arc3 method), 367
map() (curve.BezierSpline method), 340
map() (curve.CardinalSpline method), 347
map() (curve.CardinalSpline2 method), 354
map() (curve.Curve method), 318
map() (curve.Line method), 330
map() (curve.NaturalSpline method), 360
map() (curve.PolyLine method), 325
map() (curve.Spiral method), 379
map() (fe.FEModel method), 394
map() (fe.Model method), 390
map() (formex.Formex method), 114
map() (geometry.Geometry method), 162
map() (mesh.Mesh method), 201
map() (polygon.Polygon method), 429
map() (trisurface.TriSurface method), 458
map1() (coords.Coords method), 94
map1() (curve.Arc method), 373
map1() (curve.Arc3 method), 367
map1() (curve.BezierSpline method), 340
map1() (curve.CardinalSpline method), 347
map1() (curve.CardinalSpline2 method), 354
map1() (curve.Curve method), 318
map1() (curve.Line method), 330
map1() (curve.NaturalSpline method), 360
map1() (curve.PolyLine method), 325
map1() (curve.Spiral method), 379
map1() (fe.FEModel method), 394
map1() (fe.Model method), 390
map1() (formex.Formex method), 114
map1() (geometry.Geometry method), 162
map1() (mesh.Mesh method), 201
map1() (polygon.Polygon method), 429
map1() (trisurface.TriSurface method), 458
mapd() (coords.Coords method), 94
mapd() (curve.Arc method), 373
mapd() (curve.Arc3 method), 367
mapd() (curve.BezierSpline method), 340
mapd() (curve.CardinalSpline method), 347
mapd() (curve.CardinalSpline2 method), 354
mapd() (curve.Curve method), 318
mapd() (curve.Line method), 330
mapd() (curve.NaturalSpline method), 360
mapd() (curve.PolyLine method), 325
mapd() (curve.Spiral method), 379
mapd() (fe.FEModel method), 394
mapd() (fe.Model method), 390
mapd() (formex.Formex method), 115
mapd() (geometry.Geometry method), 162
mapd() (mesh.Mesh method), 201
mapd() (polygon.Polygon method), 429
mapd() (trisurface.TriSurface method), 458
mapHexLong() (in module vascularsweepingmesher), 483
mapQuadLong() (in module vascularsweepingmesher), 483
Mark (class in decors), 269
Mark (class in marks), 274
MarkList (class in marks), 275
marks (module), 274
maskedEdgeFrontWalk() (mesh.Mesh method), 188
maskedEdgeFrontWalk() (trisurface.TriSurface method), 466
match() (coords.Coords method), 98
match() (flatkeydb.FlatDB method), 496

matchAll() (in module utils), 217
matchAny() (in module utils), 216
matchCentroids() (mesh.Mesh method), 190
matchCentroids() (trisurface.TriSurface method), 468
matchCoords() (mesh.Mesh method), 190
matchCoords() (trisurface.TriSurface method), 467
matchCount() (in module utils), 216
matchIndex() (in module arraytools), 133
matchMany() (in module utils), 216
matchNone() (in module utils), 216
MaterialDB (class in properties), 434
maxcon() (adjacency.Adjacency method), 176
maxnodes() (connectivity.Connectivity method), 164
maxProp() (formex.Formex method), 106
maxProp() (mesh.Mesh method), 183
maxProp() (trisurface.TriSurface method), 461
maxWinSize() (in module widgets), 256
meanNodes() (mesh.Mesh method), 191
meanNodes() (trisurface.TriSurface method), 469
memory_report() (in module utils), 223
Menu (class in menu), 259
menu (module), 258
MenuBar (class in menu), 261
mergedModel() (in module fe), 397
mergeMeshes() (in module mesh), 204
mergeNodes() (in module mesh), 204
Mesh (class in mesh), 182
mesh (module), 182
meshes() (fe.Model method), 388
message() (in module draw), 146
message() (in module sendmail), 497
MessageBox (class in widgets), 254
minmax() (in module arraytools), 126
mirror() (formex.Formex method), 111
mkdir() (in module script), 141
mkpdir() (in module script), 142
Model (class in fe), 388
monomial() (in module polynomial), 433
mouse_draw() (viewport.QtCanvas method), 286
mouse_draw_line() (viewport.QtCanvas method), 288
mouse_pick() (viewport.QtCanvas method), 288
mouse_rectangle_zoom() (viewport.QtCanvas method), 285
mouseMoveEvent() (viewport.QtCanvas method), 288
mousePressEvent() (viewport.QtCanvas method), 288
mouseReleaseEvent() (viewport.QtCanvas method), 288
move() (camera.Camera method), 297
movingAverage() (in module arraytools), 137
movingView() (in module arraytools), 136
mplex() (fe.Model method), 388
mtime() (in module utils), 218
MultiCanvas (class in viewport), 294
multiplex() (in module arraytools), 125
multiplicity() (in module arraytools), 135
mv() (in module turtle), 480
mydict (module), 487

N

name() (elements.ElementType class method), 181
name() (widgets.InputBool method), 239
name() (widgets.InputButton method), 244
name() (widgets.InputColor method), 245
name() (widgets.InputCombo method), 241
name() (widgets.InputFile method), 245
name() (widgets.InputFloat method), 242
name() (widgets.InputFont method), 245
name() (widgets.InputFSlider method), 243
name() (widgets.InputInfo method), 238
name() (widgets.InputInteger method), 242
name() (widgets.InputItem method), 237
name() (widgets.InputIVector method), 244
name() (widgets.InputLabel method), 238
name() (widgets.InputList method), 240
name() (widgets.InputPoint method), 244
name() (widgets.InputPush method), 241
name() (widgets.InputRadio method), 241
name() (widgets.InputSlider method), 243
name() (widgets.InputString method), 238
name() (widgets.InputTable method), 243
name() (widgets.InputText method), 239
name() (widgets.InputWidget method), 246
named() (in module script), 139
names() (menu.ActionList method), 263
NameSequence (class in utils), 212
NaturalSpline (class in curve), 359
ncoords() (coords.Coords method), 84
ncoords() (nurbs.Coords4 method), 417
ndarray, 21
ndim() (formex.Formex method), 105
nearestValue() (in module arraytools), 132
nEdgeAdjacent() (mesh.Mesh method), 189
nEdgeAdjacent() (trisurface.TriSurface method), 467
nEdgeConnected() (mesh.Mesh method), 189

nEdgeConnected() (trisurface.TriSurface method), 467
nedges() (mesh.Mesh method), 184
nedges() (trisurface.TriSurface method), 447
nelems() (actors.GeomActor method), 268
nelems() (adjacency.Adjacency method), 176
nelems() (connectivity.Connectivity method), 164
nelems() (curve.Arc method), 375
nelems() (curve.Arc3 method), 369
nelems() (curve.BezierSpline method), 342
nelems() (curve.CardinalSpline method), 349
nelems() (curve.CardinalSpline2 method), 356
nelems() (curve.Curve method), 320
nelems() (curve.NaturalSpline method), 362
nelems() (curve.Spiral method), 381
nelems() (fe.FEModel method), 393
nelems() (fe.Model method), 388
nelems() (formex.Formex method), 105
nelems() (geometry.Geometry method), 159
nelems() (polygon.Polygon method), 430
neu_exp (module), 415
NewiMultiCanvas (class in viewport), 293
newRecord() (flatkeydb.FlatDB method), 495
newView() (viewport.MultiCanvas method), 294
next() (utils.NameSequence method), 213
nextFilename() (in module tetgen), 445
nextInc() (fe_post.FeResult method), 411
nextitem() (appMenu.AppMenu method), 307
nextitem() (menu.BaseMenu method), 259
nextitem() (menu.Menu method), 260
nextitem() (menu.MenuBar method), 261
nextStep() (fe_post.FeResult method), 411
nfaces() (trisurface.TriSurface method), 447
ngroups() (fe.Model method), 388
niceLogSize() (in module arraytools), 121
niceNumber() (in module arraytools), 121
nNodeAdjacent() (mesh.Mesh method), 189
nNodeAdjacent() (trisurface.TriSurface method), 467
nNodeConnected() (mesh.Mesh method), 189
nNodeConnected() (trisurface.TriSurface method), 467
nnodes() (connectivity.Connectivity method), 164
nnodes() (elements.ElementType class method), 181
nnodes() (fe.Model method), 388
nnodes() (formex.Formex method), 117
NodalAcc() (calpy_itf.QuadInterpolator method), 311
NodalAvg() (calpy_itf.QuadInterpolator method), 311
nodalSum() (in module arraytools), 138
nodalToElement() (trisurface.TriSurface method), 460
nodeAdjacency() (mesh.Mesh method), 189
nodeAdjacency() (trisurface.TriSurface method), 467
nodeConnections() (mesh.Mesh method), 189
nodeConnections() (trisurface.TriSurface method), 467
nodeProp() (properties.PropertyDB method), 439
nonManifoldEdgeNodes() (mesh.Mesh method), 190
nonManifoldEdgeNodes() (trisurface.TriSurface method), 467
nonManifoldEdges() (mesh.Mesh method), 189
nonManifoldEdges() (trisurface.TriSurface method), 449
nonManifoldNodes() (mesh.Mesh method), 189
nonManifoldNodes() (trisurface.TriSurface method), 467
norm() (in module arraytools), 122
normalize() (adjacency.Adjacency method), 176
normalize() (in module arraytools), 122
normalize() (nurbs.Coords4 method), 417
notConnectedTo() (mesh.Mesh method), 192
notConnectedTo() (trisurface.TriSurface method), 470
nParents() (connectivity.Connectivity method), 168
nplex() (actors.GeomActor method), 268
nplex() (connectivity.Connectivity method), 164
nplex() (elements.ElementType class method), 181
nplex() (formex.Formex method), 105
npoints() (coords.Coords method), 84
npoints() (formex.Formex method), 106
npoints() (nurbs.Coords4 method), 417
npoints() (polygon.Polygon method), 427
nsetName() (in module fe_abq), 400
numsplit() (in module utils), 220
nurbs (module), 416
NurbsCurve (class in nurbs), 418
NurbsSurface (class in nurbs), 419
nvertices() (elements.ElementType class method), 181
nViewports() (in module draw), 155

O

objdict() (webgl.WebGL method), 484
object_type() (objects.DrawableObjects method), 425
object_type() (objects.Objects method), 422

Objects (class in objects), 422
objects (module), 422
ObjFile (class in export), 388
objSize() (in module widgets), 256
ODict (class in odict), 489
odict (module), 489
odict() (objects.DrawableObjects method), 425
odict() (objects.Objects method), 423
off_to_tet() (in module trisurface), 479
offset() (trisurface.TriSurface method), 451
olist (module), 486
onOff() (in module canvas), 283
open() (curve.Line method), 335
open() (curve.PolyLine method), 321
OpenGLFormat() (in module viewport), 295
OpenGLSupportedVersions() (in module viewport), 295
origin() (in module coords), 101
orthog() (in module arraytools), 122
out() (dxf.DxfExporter method), 386
Output (class in fe_abq), 397
OutputRequest() (fe_post.FeResult method), 410
overflow() (colorscale.ColorLegend method), 264
overrideMode() (canvas.Canvas method), 279
overrideMode() (viewport.QtCanvas method), 289

P

pairs() (adjacency.Adjacency method), 177
parse() (flatkeydb.FlatDB method), 496
parseLine() (flatkeydb.FlatDB method), 496
part() (curve.BezierSpline method), 338
part() (curve.CardinalSpline method), 352
partition (module), 426
partition() (in module partition), 426
partitionByAngle() (mesh.Mesh method), 189
partitionByAngle() (trisurface.TriSurface method), 452
partitionByConnection() (mesh.Mesh method), 188
partitionByConnection() (trisurface.TriSurface method), 466
partitionCollection() (in module tools), 446
parts() (curve.BezierSpline method), 338
parts() (curve.CardinalSpline method), 352
parts() (curve.Line method), 336
parts() (curve.PolyLine method), 323
pattern() (in module coords), 101
pause() (in module draw), 154
peek() (utils.NameSequence method), 213
peel() (mesh.Mesh method), 187
peel() (trisurface.TriSurface method), 465
percentile() (in module arraytools), 135
perimeters() (trisurface.TriSurface method), 450
perpendicularVector() (in module geomtools), 226
pick() (in module draw), 155
pick() (viewport.QtCanvas method), 285
pick_actors() (canvas.Canvas method), 282
pick_actors() (viewport.QtCanvas method), 292
pick_edges() (canvas.Canvas method), 282
pick_edges() (viewport.QtCanvas method), 292
pick_elements() (canvas.Canvas method), 282
pick_elements() (viewport.QtCanvas method), 292
pick_faces() (canvas.Canvas method), 282
pick_faces() (viewport.QtCanvas method), 292
pick_numbers() (canvas.Canvas method), 282
pick_numbers() (viewport.QtCanvas method), 292
pick_parts() (canvas.Canvas method), 282
pick_parts() (viewport.QtCanvas method), 292
pick_points() (canvas.Canvas method), 282
pick_points() (viewport.QtCanvas method), 292
pickGL() (actors.GeomActor method), 268
pickGL() (decors.ColorLegend method), 272
pickGL() (decors.Decoration method), 269
pickGL() (decors.GlutText method), 270
pickGL() (decors.Grid method), 272
pickGL() (decors.Line method), 270
pickGL() (decors.LineDrawing method), 272
pickGL() (decors.Mark method), 269
pickGL() (decors.Rectangle method), 272
pickGL() (decors.Triade method), 273
pickGL() (marks.AxesMark method), 274
pickGL() (marks.Mark method), 274
pickGL() (marks.MarkList method), 275
pickGL() (marks.TextMark method), 274
pickle_load() (in module project), 212
pickNumbers() (viewport.QtCanvas method), 286
PlaneActor (class in actors), 267
PlaneSection (class in section2d), 441
play() (in module draw), 154
play() (in module turtle), 480
playScript() (in module script), 140
plexitude, 21
plot2d (module), 427
point() (formex.Formex method), 105
point() (in module simple), 207
point2str() (formex.Formex class method), 107
points() (actors.GeomActor method), 268
points() (coords.Coords method), 84
pointsAt() (curve.Arc method), 375
pointsAt() (curve.Arc3 method), 368
pointsAt() (curve.BezierSpline method), 342
pointsAt() (curve.CardinalSpline method), 348

pointsAt() (curve.CardinalSpline2 method), 356
pointsAt() (curve.Curve method), 315
pointsAt() (curve.Line method), 332
pointsAt() (curve.NaturalSpline method), 362
pointsAt() (curve.PolyLine method), 326
pointsAt() (curve.Spiral method), 381
pointsAt() (in module formex), 118
pointsAt() (nurbs.NurbsCurve method), 418
pointsAt() (nurbs.NurbsSurface method), 419
pointsAtLines() (in module geomtools), 226
pointsAtSegments() (in module geomtools), 227
pointsize() (in module draw), 153
pointsOff() (curve.BezierSpline method), 338
pointsOff() (curve.CardinalSpline method), 352
pointsOn() (curve.BezierSpline method), 338
pointsOn() (curve.CardinalSpline method), 352
pointsOnBezierCurve() (in module nurbs), 421
Polygon (class in polygon), 427
polygon (module), 427
polygon() (in module simple), 208
polygonNormals() (in module geomtools), 225
PolyLine (class in curve), 321
polyline() (dxf.DxfExporter method), 386
polylineToNurbs() (in module nurbs), 422
Polynomial (class in polynomial), 432
polynomial (module), 432
polynomial() (in module polynomial), 433
pop() (in module turtle), 480
pop() (project.Project method), 212
popitem() (project.Project method), 212
pos() (odict.KeyedList method), 490
pos() (odict.ODict method), 490
position() (coords.Coords method), 91
position() (curve.Arc method), 373
position() (curve.Arc3 method), 367
position() (curve.BezierSpline method), 341
position() (curve.CardinalSpline method), 347
position() (curve.CardinalSpline2 method), 354
position() (curve.Curve method), 318
position() (curve.Line method), 330
position() (curve.NaturalSpline method), 361
position() (curve.PolyLine method), 325
position() (curve.Spiral method), 380
position() (fe.FEModel method), 394
position() (fe.Model method), 390
position() (formex.Formex method), 115
position() (geometry.Geometry method), 161
position() (mesh.Mesh method), 201
position() (polygon.Polygon method), 429
position() (trisurface.TriSurface method), 458
postproc (module), 433
powers() (in module arraytools), 120
pprint() (in module arraytools), 138
Predefined() (units.UnitsSystem method), 481
prefixDict() (in module utils), 221
prefixFiles() (in module utils), 216
prepare() (in module partition), 426
prevInc() (fe_post.FeResult method), 411
prevStep() (fe_post.FeResult method), 411
principal() (in module inertia), 412
princTensor2D() (in module section2d), 442
print() (properties.PropertyDB method), 438
printall() (in module script), 141
printbbox() (objects.DrawableObjects method), 426
printbbox() (objects.Objects method), 423
printElementTypes() (in module elements), 182
printMessage() (in module draw), 146
printSteps() (fe_post.FeResult method), 411
printval() (objects.DrawableObjects method), 426
printval() (objects.Objects method), 423
processArgs() (in module script), 141
Project (class in project), 210
project (module), 210
project() (camera.Camera method), 299
project() (canvas.Canvas method), 281
project() (viewport.QtCanvas method), 291
projected() (in module polygon), 432
projectedArea() (in module geomtools), 224
projection() (in module arraytools), 122
projection() (in module viewport), 295
projectionVOP() (in module geomtools), 226
projectionVOV() (in module geomtools), 226
projectName() (in module utils), 215
projectOnCylinder() (coords.Coords method), 95
projectOnCylinder() (curve.Arc method), 373
projectOnCylinder() (curve.Arc3 method), 367
projectOnCylinder() (curve.BezierSpline method), 341
projectOnCylinder() (curve.CardinalSpline method), 347
projectOnCylinder() (curve.CardinalSpline2 method), 354
projectOnCylinder() (curve.Curve method), 318
projectOnCylinder() (curve.Line method), 331
projectOnCylinder() (curve.NaturalSpline method), 361
projectOnCylinder() (curve.PolyLine method), 325
projectOnCylinder() (curve.Spiral method), 380
projectOnCylinder() (fe.FEModel method), 394
projectOnCylinder() (fe.Model method), 390

projectOnCylinder() (formex.Formex method), 115
projectOnCylinder() (geometry.Geometry method), 162
projectOnCylinder() (mesh.Mesh method), 201
projectOnCylinder() (polygon.Polygon method), 429
projectOnCylinder() (trisurface.TriSurface method), 458
projectOnPlane() (coords.Coords method), 95
projectOnPlane() (curve.Arc method), 373
projectOnPlane() (curve.Arc3 method), 367
projectOnPlane() (curve.BezierSpline method), 341
projectOnPlane() (curve.CardinalSpline method), 347
projectOnPlane() (curve.CardinalSpline2 method), 354
projectOnPlane() (curve.Curve method), 319
projectOnPlane() (curve.Line method), 331
projectOnPlane() (curve.NaturalSpline method), 361
projectOnPlane() (curve.PolyLine method), 325
projectOnPlane() (curve.Spiral method), 380
projectOnPlane() (fe.FEModel method), 394
projectOnPlane() (fe.Model method), 390
projectOnPlane() (formex.Formex method), 115
projectOnPlane() (geometry.Geometry method), 162
projectOnPlane() (mesh.Mesh method), 201
projectOnPlane() (polygon.Polygon method), 429
projectOnPlane() (trisurface.TriSurface method), 458
projectOnSphere() (coords.Coords method), 95
projectOnSphere() (curve.Arc method), 373
projectOnSphere() (curve.Arc3 method), 367
projectOnSphere() (curve.BezierSpline method), 341
projectOnSphere() (curve.CardinalSpline method), 347
projectOnSphere() (curve.CardinalSpline2 method), 354
projectOnSphere() (curve.Curve method), 319
projectOnSphere() (curve.Line method), 331
projectOnSphere() (curve.NaturalSpline method), 361
projectOnSphere() (curve.PolyLine method), 325
projectOnSphere() (curve.Spiral method), 380
projectOnSphere() (fe.FEModel method), 394
projectOnSphere() (fe.Model method), 391
projectOnSphere() (formex.Formex method), 115
projectOnSphere() (geometry.Geometry method), 162
projectOnSphere() (mesh.Mesh method), 201
projectOnSphere() (polygon.Polygon method), 429
projectOnSphere() (trisurface.TriSurface method), 459
projectOnSurface() (coords.Coords method), 95
ProjectSelection (class in widgets), 252
Prop() (properties.PropertyDB method), 438
properties (module), 434
properties() (in module webgl), 485
PropertyDB (class in properties), 437
propSet() (formex.Formex method), 106
propSet() (mesh.Mesh method), 183
propSet() (trisurface.TriSurface method), 461
pshape() (coords.Coords method), 84
push() (in module turtle), 480
pwdir() (in module script), 141
pyf_eltype() (in module ccxinp), 313
pyformex_gts (module), 440
pyformexIcon() (in module widgets), 256

Q

QtCanvas (class in viewport), 284
quad4_wts() (in module mesh), 205
quadgrid() (in module mesh), 205
QuadInterpolator (class in calpy_itf), 311
quadraticCurve() (in module simple), 208
quality() (trisurface.TriSurface method), 450
quit() (in module script), 141

R

raiseKeyError() (in module mydict), 489
randomNoise() (in module arraytools), 137
read() (config.Config method), 493
read() (formex.Formex class method), 113
read() (timer.Timer method), 498
read() (trisurface.TriSurface class method), 448
Read() (units.UnitsSystem method), 481
read_ascii_large() (in module trisurface), 478
read_error() (in module trisurface), 478
read_gambit_neutral() (in module fileread), 234
read_gambit_neutral_hex() (in module fileread), 234
read_gts() (in module trisurface), 477
read_off() (in module fileread), 233
read_stl() (in module trisurface), 477
read_stl_bin() (in module fileread), 234
read_stla() (in module trisurface), 478
read_tetgen() (in module neu_exp), 416
readArray() (in module arraytools), 127

readCommand() (in module ccxinp), 313
readCoords() (in module flavia), 411
readData() (in module datareader), 385
readDatabase() (properties.Database method), 434
readDatabase() (properties.MaterialDB method), 434
readDatabase() (properties.SectionDB method), 435
readDicom() (in module imagearray), 304
readDispl() (in module ccxdat), 312
readDXF() (in module dxf), 387
readEleFile() (in module tetgen), 443
readElems() (in module fileread), 233
readElems() (in module flavia), 411
readElemsBlock() (in module tetgen), 444
readEsets() (in module fileread), 233
readFaceFile() (in module tetgen), 443
readFacesBlock() (in module tetgen), 444
readFile() (flatkeydb.FlatDB method), 496
readFlavia() (in module flavia), 412
readFromFile() (objects.DrawableObjects method), 426
readFromFile() (objects.Objects method), 424
readGeomFile() (in module script), 142
readHeader() (project.Project method), 212
readInpFile() (in module fileread), 233
readInput() (in module ccxinp), 313
readMesh() (in module flavia), 411
readMeshFile() (in module fileread), 233
readNeigh() (in module tetgen), 445
readNodeFile() (in module tetgen), 443
readNodes() (in module fileread), 233
readNodesBlock() (in module tetgen), 444
readPolyFile() (in module tetgen), 444
readResult() (in module flavia), 412
readResults() (in module ccxdat), 312
readResults() (in module flavia), 411
readSmeshFacetsBlock() (in module tetgen), 445
readSmeshFile() (in module tetgen), 444
readStress() (in module ccxdat), 312
readSurface() (in module tetgen), 444
readTetgen() (in module tetgen), 445
record_error_handler() (flatkeydb.FlatDB method), 495
rect() (in module simple), 207
Rectangle (class in decors), 272
rectangle() (in module mesh), 206
rectangle() (in module simple), 207
Rectangle() (in module trisurface), 479
rectangle_with_hole() (in module mesh), 206
redraw() (canvas.ActorList method), 276
redrawAll() (canvas.Canvas method), 281
redrawAll() (viewport.QtCanvas method), 291
reduceAdjacency() (in module adjacency), 179
reduceDegenerate() (connectivity.Connectivity method), 165
reduceDegenerate() (mesh.Mesh method), 194
reduceDegenerate() (trisurface.TriSurface method), 472
refine() (trisurface.TriSurface method), 453
reflect() (coords.Coords method), 91
reflect() (curve.Arc method), 374
reflect() (curve.Arc3 method), 367
reflect() (curve.BezierSpline method), 341
reflect() (curve.CardinalSpline method), 347
reflect() (curve.CardinalSpline2 method), 355
reflect() (curve.Curve method), 319
reflect() (curve.Line method), 331
reflect() (curve.NaturalSpline method), 361
reflect() (curve.PolyLine method), 325
reflect() (curve.Spiral method), 380
reflect() (fe.FEModel method), 395
reflect() (fe.Model method), 391
reflect() (formex.Formex method), 115
reflect() (geometry.Geometry method), 161
reflect() (mesh.Mesh method), 193
reflect() (polygon.Polygon method), 429
reflect() (trisurface.TriSurface method), 471
refreshDict() (in module utils), 222
regularGrid() (in module simple), 207
reload() (appMenu.AppMenu method), 307
reloadMenu() (in module appMenu), 309
remember() (objects.DrawableObjects method), 425
remember() (objects.Objects method), 423
remove() (collection.Collection method), 491
remove() (formex.Formex method), 109
remove() (in module olist), 487
remove_triangles() (in module trisurface), 479
removeActor() (canvas.Canvas method), 280
removeActor() (viewport.QtCanvas method), 290
removeAnnotation() (canvas.Canvas method), 280
removeAnnotation() (objects.DrawableObjects method), 424
removeAnnotation() (viewport.QtCanvas method), 290
removeAny() (canvas.Canvas method), 281
removeAny() (viewport.QtCanvas method), 291
removeButton() (in module toolbar), 310
removed() (utils.DictDiff method), 213
removeDecoration() (canvas.Canvas method), 280
removeDecoration() (viewport.QtCanvas method),

291
removeDegenerate() (connectivity.Connectivity method), 165
removeDegenerate() (mesh.Mesh method), 195
removeDegenerate() (trisurface.TriSurface method), 451
removeDict() (in module utils), 222
removeDuplicate() (connectivity.Connectivity method), 166
removeDuplicate() (formex.Formex method), 109
removeDuplicate() (mesh.Mesh method), 195
removeDuplicate() (trisurface.TriSurface method), 472
removeFile() (in module utils), 217
removeHighlight() (canvas.Canvas method), 280
removeHighlight() (in module draw), 155
removeHighlight() (viewport.QtCanvas method), 290
removeItem() (appMenu.AppMenu method), 308
removeItem() (menu.BaseMenu method), 259
removeItem() (menu.Menu method), 260
removeItem() (menu.MenuBar method), 261
removeKnots() (nurbs.NurbsCurve method), 418
RemoveListItem() (in module properties), 440
removeRows() (widgets.TableModel method), 249
removeTree() (in module utils), 217
removeView() (viewport.NewiMultiCanvas method), 294
removeViewport() (built-in function), 52
removeViewport() (in module draw), 155
rename() (in module script), 139
renderMode() (in module draw), 152
renderModes() (in module draw), 152
renumber() (connectivity.Connectivity method), 167
renumber() (fe.Model method), 389
renumber() (mesh.Mesh method), 195
renumber() (trisurface.TriSurface method), 472
renumberElems() (mesh.Mesh method), 195
renumberElems() (trisurface.TriSurface method), 473
renumberIndex() (in module arraytools), 129
reorder() (connectivity.Connectivity method), 167
reorder() (mesh.Mesh method), 195
reorder() (trisurface.TriSurface method), 473
reorderAxis() (in module arraytools), 124
rep() (coords.Coords method), 100
rep() (formex.Formex method), 111
replace() (coords.Coords method), 95
replace() (curve.Arc method), 374
replace() (curve.Arc3 method), 367
replace() (curve.BezierSpline method), 341
replace() (curve.CardinalSpline method), 347
replace() (curve.CardinalSpline2 method), 355
replace() (curve.Curve method), 319
replace() (curve.Line method), 331
replace() (curve.NaturalSpline method), 361
replace() (curve.PolyLine method), 325
replace() (curve.Spiral method), 380
replace() (fe.FEModel method), 395
replace() (fe.Model method), 391
replace() (formex.Formex method), 115
replace() (geometry.Geometry method), 162
replace() (mesh.Mesh method), 201
replace() (polygon.Polygon method), 429
replace() (trisurface.TriSurface method), 459
replay() (in module draw), 154
replic() (formex.Formex method), 111
replic2() (formex.Formex method), 111
replicate() (coords.Coords method), 97
replicate() (formex.Formex method), 111
report() (camera.Camera method), 297
report() (connectivity.Connectivity method), 164
report() (mesh.Mesh method), 184
report() (trisurface.TriSurface method), 462
requireRevision() (in module script), 142
reset() (canvas.CanvasSettings method), 277
reset() (in module draw), 149
reset() (in module turtle), 480
reset() (timer.Timer method), 497
resetArea() (camera.Camera method), 298
resetDefaults() (canvas.Canvas method), 278
resetDefaults() (viewport.QtCanvas method), 288
resetGUI() (in module draw), 156
resetLighting() (canvas.Canvas method), 278
resetLighting() (viewport.QtCanvas method), 288
resetOptions() (viewport.QtCanvas method), 284
resetWarnings() (in module menu), 263
resized() (curve.Arc method), 377
resized() (curve.Arc3 method), 371
resized() (curve.BezierSpline method), 344
resized() (curve.CardinalSpline method), 351
resized() (curve.CardinalSpline2 method), 358
resized() (curve.Curve method), 321
resized() (curve.Line method), 334
resized() (curve.NaturalSpline method), 364
resized() (curve.PolyLine method), 329
resized() (curve.Spiral method), 383
resized() (fe.FEModel method), 396
resized() (fe.Model method), 392
resized() (formex.Formex method), 117
resized() (geometry.Geometry method), 160

resized() (mesh.Mesh method), 203
resized() (polygon.Polygon method), 431
resized() (trisurface.TriSurface method), 461
resizeImage() (in module imagearray), 302
resolve() (connectivity.Connectivity method), 172
Result (class in fe_abq), 398
returnNone() (in module mydict), 489
reverse() (curve.BezierSpline method), 339
reverse() (curve.CardinalSpline method), 353
reverse() (curve.Line method), 336
reverse() (curve.PolyLine method), 323
reverse() (formex.Formex method), 111
reverse() (mesh.Mesh method), 193
reverse() (polygon.Polygon method), 428
reverse() (trisurface.TriSurface method), 470
reverseAxis() (in module arraytools), 125
revolve() (mesh.Mesh method), 197
revolve() (trisurface.TriSurface method), 474
rgb2qimage() (in module imagearray), 303
RGBA() (in module colors), 158
RGBAcolor() (in module colors), 157
RGBcolor() (in module colors), 157
ro() (in module turtle), 480
roll() (curve.Line method), 336
roll() (curve.PolyLine method), 322
roll() (in module olist), 486
rollAxes() (coords.Coords method), 95
rollAxes() (curve.Arc method), 374
rollAxes() (curve.Arc3 method), 367
rollAxes() (curve.BezierSpline method), 341
rollAxes() (curve.CardinalSpline method), 347
rollAxes() (curve.CardinalSpline2 method), 355
rollAxes() (curve.Curve method), 319
rollAxes() (curve.Line method), 331
rollAxes() (curve.NaturalSpline method), 361
rollAxes() (curve.PolyLine method), 325
rollAxes() (curve.Spiral method), 380
rollAxes() (fe.FEModel method), 395
rollAxes() (fe.Model method), 391
rollAxes() (formex.Formex method), 115
rollAxes() (geometry.Geometry method), 162
rollAxes() (mesh.Mesh method), 201
rollAxes() (polygon.Polygon method), 429
rollAxes() (trisurface.TriSurface method), 459
ros() (formex.Formex method), 111
rosette() (formex.Formex method), 111
rot() (coords.Coords method), 100
rot() (curve.Arc method), 374
rot() (curve.Arc3 method), 367
rot() (curve.BezierSpline method), 341
rot() (curve.CardinalSpline method), 347
rot() (curve.CardinalSpline2 method), 355
rot() (curve.Curve method), 319
rot() (curve.Line method), 331
rot() (curve.NaturalSpline method), 361
rot() (curve.PolyLine method), 325
rot() (curve.Spiral method), 380
rot() (fe.FEModel method), 395
rot() (fe.Model method), 391
rot() (formex.Formex method), 115
rot() (geometry.Geometry method), 163
rot() (mesh.Mesh method), 202
rot() (polygon.Polygon method), 430
rot() (trisurface.TriSurface method), 459
rotate() (camera.Camera method), 297
rotate() (coords.Coords method), 91
rotate() (curve.Arc method), 374
rotate() (curve.Arc3 method), 367
rotate() (curve.BezierSpline method), 341
rotate() (curve.CardinalSpline method), 347
rotate() (curve.CardinalSpline2 method), 355
rotate() (curve.Curve method), 319
rotate() (curve.Line method), 331
rotate() (curve.NaturalSpline method), 361
rotate() (curve.PolyLine method), 325
rotate() (curve.Spiral method), 380
rotate() (fe.FEModel method), 395
rotate() (fe.Model method), 391
rotate() (formex.Formex method), 115
rotate() (geometry.Geometry method), 160
rotate() (mesh.Mesh method), 202
rotate() (polygon.Polygon method), 430
rotate() (trisurface.TriSurface method), 459
RotatedActor (class in actors), 265
rotationAngle() (in module geomtools), 226
rotationAnglesFromMatrix() (in module arraytools), 124
rotationMatrix() (in module arraytools), 123
rotmat() (in module arraytools), 123
rotMatrix() (in module arraytools), 124
rowCount() (widgets.ArrayModel method), 250
rowCount() (widgets.TableModel method), 249
rowHeights() (widgets.Table method), 251
run() (appMenu.AppMenu method), 306
runAll() (appMenu.AppMenu method), 307
runAllNext() (appMenu.AppMenu method), 307
runAny() (in module script), 140
runApp() (appMenu.AppMenu method), 306
runCommand() (in module utils), 219
runCurrent() (appMenu.AppMenu method), 307
runNextApp() (appMenu.AppMenu method), 307
runRandom() (appMenu.AppMenu method), 307

runScript() (in module script), 140
runTetgen() (in module tetgen), 445
runtime() (in module script), 142

S

saneSettings() (in module webgl), 485
save() (in module image), 300
save() (project.Project method), 211
save_canvas() (in module image), 299
save_main_window() (in module image), 300
save_rect() (in module image), 300
save_window() (in module image), 299
saveBuffer() (canvas.Canvas method), 282
saveBuffer() (viewport.QtCanvas method), 292
saveIcon() (in module image), 301
saveImage() (in module image), 300
SaveImageDialog (class in widgets), 252
saveModelView() (camera.Camera method), 297
saveMovie() (in module image), 302
saveNext() (in module image), 301
scale() (colorscale.ColorScale method), 263
scale() (coords.Coords method), 90
scale() (curve.Arc method), 374
scale() (curve.Arc3 method), 368
scale() (curve.BezierSpline method), 341
scale() (curve.CardinalSpline method), 348
scale() (curve.CardinalSpline2 method), 355
scale() (curve.Curve method), 319
scale() (curve.Line method), 331
scale() (curve.NaturalSpline method), 361
scale() (curve.PolyLine method), 325
scale() (curve.Spiral method), 380
scale() (fe.FEModel method), 395
scale() (fe.Model method), 391
scale() (formex.Formex method), 115
scale() (geometry.Geometry method), 160
scale() (mesh.Mesh method), 202
scale() (polygon.Polygon method), 430
scale() (trisurface.TriSurface method), 459
scaledJacobian() (trisurface.TriSurface method), 457
sceneBbox() (canvas.Canvas method), 280
sceneBbox() (viewport.QtCanvas method), 290
script (module), 139
ScrollForm (class in widgets), 246
seconds() (timer.Timer method), 498
section() (dxf.DxfExporter method), 386
section2d (module), 441
sectionChar() (in module section2d), 442
SectionDB (class in properties), 435
sectionize (module), 443
sectionize() (in module sectionize), 443
sector() (in module simple), 209
seed() (in module mesh), 205
segmentOrientation() (in module geomtools), 226
select() (actors.GeomActor method), 269
select() (formex.Formex method), 108
select() (in module olist), 487
select() (mesh.Mesh method), 191
select() (trisurface.TriSurface method), 468
selectDict() (in module utils), 221
selectDictValues() (in module utils), 222
selectFont() (in module widgets), 257
selectNodes() (connectivity.Connectivity method), 170
selectNodes() (formex.Formex method), 108
selectNodes() (mesh.Mesh method), 192
selectNodes() (trisurface.TriSurface method), 469
sendmail (module), 497
sendmail() (in module sendmail), 497
set() (collection.Collection method), 491
set() (coords.Coords method), 89
set() (objects.DrawableObjects method), 425
set() (objects.Objects method), 422
set_edit_mode() (in module draw), 155
set_material_value() (in module draw), 153
setAll() (widgets.InputList method), 240
setAlpha() (actors.GeomActor method), 268
setAmbient() (canvas.Canvas method), 278
setAmbient() (viewport.QtCanvas method), 288
setAngles() (camera.Camera method), 297
setArea() (camera.Camera method), 298
setBackground() (canvas.Canvas method), 279
setBackground() (viewport.QtCanvas method), 289
setBbox() (canvas.Canvas method), 280
setBbox() (viewport.QtCanvas method), 290
setBkColor() (actors.GeomActor method), 268
setCamera() (canvas.Canvas method), 281
setCamera() (viewport.QtCanvas method), 291
setCellData() (widgets.TableModel method), 249
setChecked() (widgets.InputList method), 240
setChoices() (widgets.InputCombo method), 240
setClip() (camera.Camera method), 298
setColor() (actors.Actor method), 264
setColor() (actors.AxesActor method), 266
setColor() (actors.BboxActor method), 266
setColor() (actors.CoordPlaneActor method), 267
setColor() (actors.CubeActor method), 265
setColor() (actors.GeomActor method), 268
setColor() (actors.GridActor method), 267
setColor() (actors.PlaneActor method), 267

setColor() (actors.RotatedActor method), 265
setColor() (actors.SphereActor method), 266
setColor() (actors.TranslatedActor method), 265
setColor() (decors.ColorLegend method), 272
setColor() (decors.Decoration method), 269
setColor() (decors.GlutText method), 270
setColor() (decors.Grid method), 272
setColor() (decors.Line method), 270
setColor() (decors.LineDrawing method), 273
setColor() (decors.Mark method), 269
setColor() (decors.Rectangle method), 272
setColor() (decors.Triade method), 273
setColor() (marks.AxesMark method), 274
setColor() (marks.Mark method), 274
setColor() (marks.MarkList method), 275
setColor() (marks.TextMark method), 275
setCoords() (trisurface.TriSurface method), 447
setCurrent() (viewport.MultiCanvas method), 294
setCurrent() (viewport.NewiMultiCanvas method), 294
setCursorShape() (viewport.CursorShapeHandler method), 284
setCursorShape() (viewport.QtCanvas method), 284
setCursorShapeFromFunc() (viewport.CursorShapeHandler method), 284
setCursorShapeFromFunc() (viewport.QtCanvas method), 285
setData() (widgets.ArrayModel method), 250
setData() (widgets.TableModel method), 249
setdefault() (canvas.CanvasSettings method), 278
setdefault() (config.Config method), 493
setdefault() (fe_abq.Interaction method), 399
setdefault() (fe_abq.Output method), 398
setdefault() (fe_abq.Result method), 399
setdefault() (mydict.CDict method), 489
setdefault() (mydict.Dict method), 488
setdefault() (project.Project method), 212
setdefault() (properties.Database method), 434
setdefault() (properties.EdgeLoad method), 437
setdefault() (properties.ElemLoad method), 437
setdefault() (properties.ElemSection method), 436
setdefault() (properties.MaterialDB method), 435
setdefault() (properties.PropertyDB method), 438
setdefault() (properties.SectionDB method), 435
setDefaults() (canvas.Canvas method), 279
setDefaults() (viewport.QtCanvas method), 289
setDrawOptions() (in module draw), 149
setEdgesAndFaces() (trisurface.TriSurface method), 448
setElems() (trisurface.TriSurface method), 447
setFgColor() (canvas.Canvas method), 279
setFgColor() (viewport.QtCanvas method), 289
setIcon() (widgets.ButtonBox method), 255
setIcon() (widgets.InputPush method), 241
setLens() (camera.Camera method), 298
setLineStipple() (actors.Actor method), 264
setLineStipple() (actors.AxesActor method), 266
setLineStipple() (actors.BboxActor method), 266
setLineStipple() (actors.CoordPlaneActor method), 267
setLineStipple() (actors.CubeActor method), 265
setLineStipple() (actors.GeomActor method), 268
setLineStipple() (actors.GridActor method), 267
setLineStipple() (actors.PlaneActor method), 267
setLineStipple() (actors.RotatedActor method), 265
setLineStipple() (actors.SphereActor method), 266
setLineStipple() (actors.Text3DActor method), 268
setLineStipple() (actors.TranslatedActor method), 265
setLineStipple() (canvas.Canvas method), 279
setLineStipple() (decors.ColorLegend method), 272
setLineStipple() (decors.Decoration method), 269
setLineStipple() (decors.GlutText method), 270
setLineStipple() (decors.Grid method), 272
setLineStipple() (decors.Line method), 270
setLineStipple() (decors.LineDrawing method), 272
setLineStipple() (decors.Mark method), 269
setLineStipple() (decors.Rectangle method), 272
setLineStipple() (decors.Triade method), 273
setLineStipple() (marks.AxesMark method), 274
setLineStipple() (marks.Mark method), 274
setLineStipple() (marks.MarkList method), 275
setLineStipple() (marks.TextMark method), 274
setLineStipple() (viewport.QtCanvas method), 289
setLineWidth() (actors.Actor method), 264
setLineWidth() (actors.AxesActor method), 266
setLineWidth() (actors.BboxActor method), 266
setLineWidth() (actors.CoordPlaneActor method), 267
setLineWidth() (actors.CubeActor method), 265
setLineWidth() (actors.GeomActor method), 268
setLineWidth() (actors.GridActor method), 267
setLineWidth() (actors.PlaneActor method), 267
setLineWidth() (actors.RotatedActor method), 265
setLineWidth() (actors.SphereActor method), 265
setLineWidth() (actors.Text3DActor method), 268

setLineWidth() (actors.TranslatedActor method), 265
setLineWidth() (canvas.Canvas method), 279
setLineWidth() (decors.ColorLegend method), 272
setLineWidth() (decors.Decoration method), 269
setLineWidth() (decors.GlutText method), 270
setLineWidth() (decors.Grid method), 272
setLineWidth() (decors.Line method), 270
setLineWidth() (decors.LineDrawing method), 272
setLineWidth() (decors.Mark method), 269
setLineWidth() (decors.Rectangle method), 272
setLineWidth() (decors.Triade method), 273
setLineWidth() (marks.AxesMark method), 274
setLineWidth() (marks.Mark method), 274
setLineWidth() (marks.MarkList method), 275
setLineWidth() (marks.TextMark method), 274
setLineWidth() (viewport.QtCanvas method), 289
setMaterial() (canvas.Canvas method), 278
setMaterial() (viewport.QtCanvas method), 288
setMaterialDB() (in module properties), 440
setMaterialDB() (properties.PropertyDB method), 438
setMode() (canvas.CanvasSettings method), 277
setModelView() (camera.Camera method), 297
setNone() (widgets.InputList method), 240
setNormals() (mesh.Mesh method), 183
setNormals() (trisurface.TriSurface method), 461
setOpenGLFormat() (in module viewport), 295
setOptions() (viewport.QtCanvas method), 284
setPerspective() (camera.Camera method), 298
setPickable() (viewport.QtCanvas method), 285
setPointSize() (canvas.Canvas method), 279
setPointSize() (viewport.QtCanvas method), 289
setPrefs() (in module script), 141
setPrintFunction() (formex.Formex class method), 107
setProp() (curve.Arc method), 377
setProp() (curve.Arc3 method), 371
setProp() (curve.BezierSpline method), 345
setProp() (curve.CardinalSpline method), 351
setProp() (curve.CardinalSpline2 method), 359
setProp() (curve.Curve method), 317
setProp() (curve.Line method), 334
setProp() (curve.NaturalSpline method), 365
setProp() (curve.PolyLine method), 329
setProp() (curve.Spiral method), 384
setProp() (fe.FEModel method), 396
setProp() (fe.Model method), 392
setProp() (formex.Formex method), 116
setProp() (geometry.Geometry method), 159
setProp() (mesh.Mesh method), 202
setProp() (objects.DrawableObjects method), 424
setProp() (polygon.Polygon method), 431
setProp() (trisurface.TriSurface method), 460
setRenderMode() (canvas.Canvas method), 278
setRenderMode() (viewport.QtCanvas method), 288
setRotation() (camera.Camera method), 297
setSaneLocale() (in module utils), 217
setSectionDB() (in module properties), 440
setSectionDB() (properties.PropertyDB method), 438
setSelected() (widgets.InputList method), 240
setSlColor() (canvas.Canvas method), 279
setSlColor() (viewport.QtCanvas method), 289
setStepInc() (fe_post.FeResult method), 411
setStretch() (viewport.NewiMultiCanvas method), 294
setText() (widgets.ButtonBox method), 255
setText() (widgets.InputPush method), 241
setTexture() (actors.Actor method), 265
setTexture() (actors.AxesActor method), 266
setTexture() (actors.BboxActor method), 266
setTexture() (actors.CoordPlaneActor method), 267
setTexture() (actors.CubeActor method), 265
setTexture() (actors.GeomActor method), 269
setTexture() (actors.GridActor method), 267
setTexture() (actors.PlaneActor method), 267
setTexture() (actors.RotatedActor method), 265
setTexture() (actors.SphereActor method), 266
setTexture() (actors.Text3DActor method), 268
setTexture() (actors.TranslatedActor method), 265
setTexture() (decors.ColorLegend method), 272
setTexture() (decors.Decoration method), 269
setTexture() (decors.GlutText method), 270
setTexture() (decors.Grid method), 272
setTexture() (decors.Line method), 270
setTexture() (decors.LineDrawing method), 273
setTexture() (decors.Mark method), 270
setTexture() (decors.Rectangle method), 272
setTexture() (decors.Triade method), 273
setTexture() (marks.AxesMark method), 274
setTexture() (marks.Mark method), 274
setTexture() (marks.MarkList method), 275
setTexture() (marks.TextMark method), 275
setToggle() (canvas.Canvas method), 278
setToggle() (viewport.QtCanvas method), 289
setTracking() (camera.Camera method), 299
setTriade() (canvas.Canvas method), 279
setTriade() (in module draw), 151

setTriade() (viewport.QtCanvas method), 289
setType() (mesh.Mesh method), 183
setType() (trisurface.TriSurface method), 460
setValue() (widgets.InputBool method), 239
setValue() (widgets.InputButton method), 244
setValue() (widgets.InputColor method), 245
setValue() (widgets.InputCombo method), 240
setValue() (widgets.InputFloat method), 242
setValue() (widgets.InputFont method), 245
setValue() (widgets.InputFSlider method), 244
setValue() (widgets.InputGroup method), 246
setValue() (widgets.InputInfo method), 238
setValue() (widgets.InputInteger method), 242
setValue() (widgets.InputItem method), 237
setValue() (widgets.InputIVector method), 244
setValue() (widgets.InputLabel method), 238
setValue() (widgets.InputList method), 240
setValue() (widgets.InputPoint method), 244
setValue() (widgets.InputPush method), 241
setValue() (widgets.InputRadio method), 241
setValue() (widgets.InputSlider method), 243
setValue() (widgets.InputString method), 239
setValue() (widgets.InputTable method), 243
setValue() (widgets.InputText method), 239
setValue() (widgets.InputWidget method), 246
setValue() (widgets.ListSelection method), 252
setValues() (widgets.CoordsBox method), 255
setView() (in module draw), 152
setWireMode() (canvas.Canvas method), 278
setWireMode() (viewport.QtCanvas method), 288
shallowCopy() (mesh.Mesh method), 183
shallowCopy() (trisurface.TriSurface method), 461
shape() (actors.GeomActor method), 268
shape() (in module simple), 206
shape() (trisurface.TriSurface method), 447
sharedNodes() (connectivity.Connectivity method), 172
shear() (coords.Coords method), 91
shear() (curve.Arc method), 374
shear() (curve.Arc3 method), 368
shear() (curve.BezierSpline method), 341
shear() (curve.CardinalSpline method), 348
shear() (curve.CardinalSpline2 method), 355
shear() (curve.Curve method), 319
shear() (curve.Line method), 331
shear() (curve.NaturalSpline method), 361
shear() (curve.PolyLine method), 326
shear() (curve.Spiral method), 380
shear() (fe.FEModel method), 395
shear() (fe.Model method), 391
shear() (formex.Formex method), 115
shear() (geometry.Geometry method), 160
shear() (mesh.Mesh method), 202
shear() (polygon.Polygon method), 430
shear() (trisurface.TriSurface method), 459
shortestEdge() (trisurface.TriSurface method), 450
show() (widgets.InputDialog method), 248
show() (widgets.InputFloat method), 242
show() (widgets.InputFSlider method), 243
show() (widgets.InputInteger method), 242
show() (widgets.InputSlider method), 243
show() (widgets.InputString method), 238
show() (widgets.InputText method), 239
show() (widgets.ListSelection method), 253
showBuffer() (canvas.Canvas method), 282
showBuffer() (viewport.QtCanvas method), 292
showCameraTool() (in module cameratools), 312
showDoc() (in module draw), 144
showFile() (in module draw), 144
showHistogram() (in module plot2d), 427
showHTML() (in module draw), 156
showImage() (widgets.ImageView method), 255
showInfo() (in module draw), 143
showLineDrawing() (in module draw), 155
showMessage() (in module draw), 143
showStepPlot() (in module plot2d), 427
showText() (in module draw), 144
showURL() (in module draw), 156
showWidget() (viewport.MultiCanvas method), 294
shrink() (formex.Formex method), 110
shrink() (in module draw), 149
simple (module), 206
simpleInputItem() (in module widgets), 256
sind() (in module arraytools), 120
sind() (in module turtle), 480
sizes() (coords.Coords method), 86
skipComments() (in module tetgen), 444
slice() (trisurface.TriSurface method), 453
smallestAltitude() (trisurface.TriSurface method), 450
smallestDirection() (in module geomtools), 224
smooth() (mesh.Mesh method), 197
smooth() (trisurface.TriSurface method), 453
smoothLaplaceHC() (trisurface.TriSurface method), 453
smoothLowPass() (trisurface.TriSurface method), 453
solveMany() (in module arraytools), 122
sort() (coords.Coords method), 97
sort() (odict.KeyedList method), 490
sort() (odict.ODict method), 490

sortAdjacency() (in module adjacency), 178
sortByColumns() (in module arraytools), 131
sortedKeys() (in module utils), 222
sortElemsByLoadedFace() (in module fe), 397
sortSets() (in module appMenu), 308
sortSubsets() (in module arraytools), 130
sourceFiles() (in module utils), 217
spawn() (in module utils), 220
sphere() (in module simple), 208
sphere2() (in module simple), 208
sphere3() (in module simple), 208
SphereActor (class in actors), 265
spherical() (coords.Coords method), 92
spherical() (curve.Arc method), 374
spherical() (curve.Arc3 method), 368
spherical() (curve.BezierSpline method), 341
spherical() (curve.CardinalSpline method), 348
spherical() (curve.CardinalSpline2 method), 355
spherical() (curve.Curve method), 319
spherical() (curve.Line method), 331
spherical() (curve.NaturalSpline method), 361
spherical() (curve.PolyLine method), 326
spherical() (curve.Spiral method), 380
spherical() (fe.FEModel method), 395
spherical() (fe.Model method), 391
spherical() (formex.Formex method), 115
spherical() (geometry.Geometry method), 161
spherical() (mesh.Mesh method), 202
spherical() (polygon.Polygon method), 430
spherical() (trisurface.TriSurface method), 459
Spiral (class in curve), 378
splash image, 63
split() (coords.Coords method), 97
split() (curve.Arc method), 376
split() (curve.Arc3 method), 369
split() (curve.BezierSpline method), 343
split() (curve.CardinalSpline method), 349
split() (curve.CardinalSpline2 method), 357
split() (curve.Curve method), 316
split() (curve.Line method), 333
split() (curve.NaturalSpline method), 363
split() (curve.PolyLine method), 327
split() (curve.Spiral method), 382
split() (formex.Formex method), 113
split() (trisurface.TriSurface method), 454
splitAlpha() (in module appMenu), 309
splitar() (in module arraytools), 126
splitAt() (curve.Line method), 337
splitAt() (curve.PolyLine method), 323
splitByConnection() (mesh.Mesh method), 188
splitByConnection() (trisurface.TriSurface method), 466
splitDegenerate() (mesh.Mesh method), 194
splitDegenerate() (trisurface.TriSurface method), 472
splitDigits() (in module utils), 221
splitElems() (fe.Model method), 389
splitFilename() (in module utils), 214
splitFloat() (in module datareader), 385
splitKeyValue() (flatkeydb.FlatDB method), 496
splitKeyValue() (in module flatkeydb), 497
splitProp() (curve.Arc method), 377
splitProp() (curve.Arc3 method), 371
splitProp() (curve.BezierSpline method), 344
splitProp() (curve.CardinalSpline method), 351
splitProp() (curve.CardinalSpline2 method), 358
splitProp() (curve.Curve method), 320
splitProp() (curve.Line method), 334
splitProp() (curve.NaturalSpline method), 364
splitProp() (curve.PolyLine method), 328
splitProp() (curve.Spiral method), 383
splitProp() (formex.Formex method), 117
splitProp() (geometry.Geometry method), 160
splitProp() (in module partition), 426
splitProp() (mesh.Mesh method), 203
splitProp() (trisurface.TriSurface method), 461
splitRandom() (mesh.Mesh method), 193
splitRandom() (trisurface.TriSurface method), 470
splitrange() (in module arraytools), 126
st() (in module turtle), 480
stack() (in module arraytools), 126
start_draw() (viewport.QtCanvas method), 286
start_drawing() (viewport.QtCanvas method), 287
start_selection() (viewport.QtCanvas method), 285
startGui() (in module script), 142
startPart() (in module ccxinp), 313
stats() (trisurface.TriSurface method), 450
status() (lima.Lima method), 414
stlConvert() (in module trisurface), 477
stopatbreakpt() (in module script), 140
StressGP() (calpy_itf.QuadInterpolator method), 311
stripLine() (in module tetgen), 444
strNorm() (in module utils), 218
structuredHexMeshGrid() (in module vascularsweepingmesher), 482
structuredQuadMeshGrid() (in module vascularsweepingmesher), 482
stuur() (in module utils), 222
sub_curvature() (curve.BezierSpline method), 338
sub_curvature() (curve.CardinalSpline method), 352

sub_directions() (curve.Arc3 method), 365
sub_directions() (curve.BezierSpline method), 338
sub_directions() (curve.CardinalSpline method), 352
sub_directions() (curve.CardinalSpline2 method), 353
sub_directions() (curve.Curve method), 315
sub_directions() (curve.Line method), 335
sub_directions() (curve.NaturalSpline method), 359
sub_directions() (curve.PolyLine method), 322
sub_directions() (curve.Spiral method), 378
sub_directions_2() (curve.Arc method), 372
sub_directions_2() (curve.Arc3 method), 365
sub_directions_2() (curve.BezierSpline method), 339
sub_directions_2() (curve.CardinalSpline method), 345
sub_directions_2() (curve.CardinalSpline2 method), 353
sub_directions_2() (curve.Curve method), 315
sub_directions_2() (curve.Line method), 329
sub_directions_2() (curve.NaturalSpline method), 359
sub_directions_2() (curve.PolyLine method), 323
sub_directions_2() (curve.Spiral method), 378
sub_points() (curve.BezierSpline method), 338
sub_points() (curve.CardinalSpline method), 352
sub_points() (curve.Curve method), 314
sub_points() (curve.Line method), 335
sub_points() (curve.PolyLine method), 321
sub_points() (curve.Spiral method), 378
sub_points_2() (curve.Arc method), 372
sub_points_2() (curve.Arc3 method), 365
sub_points_2() (curve.BezierSpline method), 339
sub_points_2() (curve.CardinalSpline method), 345
sub_points_2() (curve.CardinalSpline2 method), 353
sub_points_2() (curve.Curve method), 315
sub_points_2() (curve.Line method), 335
sub_points_2() (curve.NaturalSpline method), 359
sub_points_2() (curve.PolyLine method), 321
sub_points_2() (curve.Spiral method), 378
subDict() (in module utils), 221
subdivide() (mesh.Mesh method), 194
subdivide() (trisurface.TriSurface method), 471
subMenus() (appMenu.AppMenu method), 307
subMenus() (menu.BaseMenu method), 258
subMenus() (menu.Menu method), 260
subMenus() (menu.MenuBar method), 261
subPoints() (curve.Arc method), 375
subPoints() (curve.Arc3 method), 369
subPoints() (curve.BezierSpline method), 342
subPoints() (curve.CardinalSpline method), 349
subPoints() (curve.CardinalSpline2 method), 356
subPoints() (curve.Curve method), 315
subPoints() (curve.Line method), 332
subPoints() (curve.NaturalSpline method), 362
subPoints() (curve.PolyLine method), 327
subPoints() (curve.Spiral method), 381
superSpherical() (coords.Coords method), 92
superSpherical() (curve.Arc method), 374
superSpherical() (curve.Arc3 method), 368
superSpherical() (curve.BezierSpline method), 341
superSpherical() (curve.CardinalSpline method), 348
superSpherical() (curve.CardinalSpline2 method), 355
superSpherical() (curve.Curve method), 319
superSpherical() (curve.Line method), 331
superSpherical() (curve.NaturalSpline method), 361
superSpherical() (curve.PolyLine method), 326
superSpherical() (curve.Spiral method), 380
superSpherical() (fe.FEModel method), 395
superSpherical() (fe.Model method), 391
superSpherical() (formex.Formex method), 116
superSpherical() (geometry.Geometry method), 161
superSpherical() (mesh.Mesh method), 202
superSpherical() (polygon.Polygon method), 430
superSpherical() (trisurface.TriSurface method), 459
surface2webgl() (in module webgl), 485
surface_volume() (in module trisurface), 477
surfaceType() (trisurface.TriSurface method), 449
swapAxes() (coords.Coords method), 95
swapAxes() (curve.Arc method), 374
swapAxes() (curve.Arc3 method), 368
swapAxes() (curve.BezierSpline method), 342
swapAxes() (curve.CardinalSpline method), 348
swapAxes() (curve.CardinalSpline2 method), 355
swapAxes() (curve.Curve method), 319
swapAxes() (curve.Line method), 331
swapAxes() (curve.NaturalSpline method), 362
swapAxes() (curve.PolyLine method), 326
swapAxes() (curve.Spiral method), 381
swapAxes() (fe.FEModel method), 395
swapAxes() (fe.Model method), 391
swapAxes() (formex.Formex method), 116

swapAxes() (geometry.Geometry method), 162
swapAxes() (mesh.Mesh method), 202
swapAxes() (polygon.Polygon method), 430
swapAxes() (trisurface.TriSurface method), 459
sweep() (mesh.Mesh method), 197
sweep() (trisurface.TriSurface method), 475
sweepCoords() (in module coords), 103
symdiff() (adjacency.Adjacency method), 177
symdifference() (in module olist), 487
system() (in module script), 140
system() (in module utils), 219
system1() (in module utils), 219

T

tabInputItem() (in module widgets), 256
Table (class in widgets), 250
TableModel (class in widgets), 248
tand() (in module arraytools), 120
tand() (in module camera), 299
test() (coords.Coords method), 89
test() (formex.Formex method), 110
test() (mesh.Mesh method), 198
test() (trisurface.TriSurface method), 475
testBbox() (in module coords), 101
testDegenerate() (connectivity.Connectivity method), 164
testDuplicate() (connectivity.Connectivity method), 166
tetgen (module), 443
tetgen() (trisurface.TriSurface method), 456
tetgenConvexHull() (in module tetgen), 446
tetMesh() (in module tetgen), 446
Text (in module decors), 270
text() (widgets.InputBool method), 239
text() (widgets.InputButton method), 244
text() (widgets.InputColor method), 245
text() (widgets.InputCombo method), 241
text() (widgets.InputFile method), 245
text() (widgets.InputFloat method), 242
text() (widgets.InputFont method), 245
text() (widgets.InputFSlider method), 243
text() (widgets.InputInfo method), 238
text() (widgets.InputInteger method), 242
text() (widgets.InputItem method), 237
text() (widgets.InputIVector method), 244
text() (widgets.InputLabel method), 238
text() (widgets.InputList method), 240
text() (widgets.InputPoint method), 244
text() (widgets.InputPush method), 241
text() (widgets.InputRadio method), 241
text() (widgets.InputSlider method), 243
text() (widgets.InputString method), 238
text() (widgets.InputTable method), 243
text() (widgets.InputText method), 239
text() (widgets.InputWidget method), 246
Text3DActor (class in actors), 267
TextBox (class in widgets), 255
TextMark (class in marks), 274
tildeExpand() (in module utils), 215
timedOut() (widgets.InputDialog method), 248
timedOut() (widgets.ListSelection method), 253
timeEval() (in module utils), 219
timeout() (in module toolbar), 310
timeout() (widgets.InputDialog method), 248
timeout() (widgets.ListSelection method), 253
Timer (class in timer), 497
timer (module), 497
toCoords() (nurbs.Coords4 method), 417
toCoords4() (in module nurbs), 421
toCurve() (mesh.Mesh method), 184
toCurve() (trisurface.TriSurface method), 462
toCylindrical() (coords.Coords method), 92
toCylindrical() (curve.Arc method), 374
toCylindrical() (curve.Arc3 method), 368
toCylindrical() (curve.BezierSpline method), 342
toCylindrical() (curve.CardinalSpline method), 348
toCylindrical() (curve.CardinalSpline2 method), 355
toCylindrical() (curve.Curve method), 319
toCylindrical() (curve.Line method), 332
toCylindrical() (curve.NaturalSpline method), 362
toCylindrical() (curve.PolyLine method), 326
toCylindrical() (curve.Spiral method), 381
toCylindrical() (fe.FEModel method), 395
toCylindrical() (fe.Model method), 391
toCylindrical() (formex.Formex method), 116
toCylindrical() (geometry.Geometry method), 161
toCylindrical() (mesh.Mesh method), 202
toCylindrical() (polygon.Polygon method), 430
toCylindrical() (trisurface.TriSurface method), 459
toFormex() (curve.Arc method), 377
toFormex() (curve.Arc3 method), 371
toFormex() (curve.BezierSpline method), 345
toFormex() (curve.CardinalSpline method), 351
toFormex() (curve.CardinalSpline2 method), 358
toFormex() (curve.Curve method), 317
toFormex() (curve.Line method), 335
toFormex() (curve.NaturalSpline method), 365
toFormex() (curve.PolyLine method), 321
toFormex() (curve.Spiral method), 384
toFormex() (elements.ElementType class method),

181
toFormex() (mesh.Mesh method), 183
toFormex() (trisurface.TriSurface method), 461
toFront() (in module olist), 487
toggleAnnotation() (objects.DrawableObjects method), 424
toggleButton() (in module toolbar), 310
toLines() (in module dxf), 387
toMesh() (curve.BezierSpline method), 339
toMesh() (curve.CardinalSpline method), 352
toMesh() (curve.Line method), 335
toMesh() (curve.PolyLine method), 321
toMesh() (elements.ElementType class method), 181
toMesh() (formex.Formex method), 107
toMesh() (mesh.Mesh method), 183
toMesh() (trisurface.TriSurface method), 462
toolbar (module), 309
toolbar() (menu.ActionList method), 263
tools (module), 446
toProp() (curve.Arc method), 375
toProp() (curve.Arc3 method), 369
toProp() (curve.BezierSpline method), 343
toProp() (curve.CardinalSpline method), 349
toProp() (curve.CardinalSpline2 method), 356
toProp() (curve.Curve method), 320
toProp() (curve.Line method), 333
toProp() (curve.NaturalSpline method), 363
toProp() (curve.PolyLine method), 327
toProp() (curve.Spiral method), 382
toProp() (formex.Formex method), 117
toProp() (geometry.Geometry method), 159
toProp() (mesh.Mesh method), 203
toProp() (trisurface.TriSurface method), 461
toSpherical() (coords.Coords method), 93
toSpherical() (curve.Arc method), 374
toSpherical() (curve.Arc3 method), 368
toSpherical() (curve.BezierSpline method), 342
toSpherical() (curve.CardinalSpline method), 348
toSpherical() (curve.CardinalSpline2 method), 355
toSpherical() (curve.Curve method), 320
toSpherical() (curve.Line method), 332
toSpherical() (curve.NaturalSpline method), 362
toSpherical() (curve.PolyLine method), 326
toSpherical() (curve.Spiral method), 381
toSpherical() (fe.FEModel method), 395
toSpherical() (fe.Model method), 391
toSpherical() (formex.Formex method), 116
toSpherical() (geometry.Geometry method), 161
toSpherical() (mesh.Mesh method), 202
toSpherical() (polygon.Polygon method), 430
toSpherical() (trisurface.TriSurface method), 459
toSurface() (formex.Formex method), 107
toSurface() (mesh.Mesh method), 184
toSurface() (trisurface.TriSurface method), 462
TotalEnergies() (fe_post.FeResult method), 410
totalMemSize() (in module utils), 223
toWorld() (camera.Camera method), 298
transArea() (camera.Camera method), 298
transform() (camera.Camera method), 298
transform() (isopar.Isopar method), 413
transformCS() (coords.Coords method), 96
transformCS() (curve.Arc method), 374
transformCS() (curve.Arc3 method), 368
transformCS() (curve.BezierSpline method), 342
transformCS() (curve.CardinalSpline method), 348
transformCS() (curve.CardinalSpline2 method), 355
transformCS() (curve.Curve method), 320
transformCS() (curve.Line method), 332
transformCS() (curve.NaturalSpline method), 362
transformCS() (curve.PolyLine method), 326
transformCS() (curve.Spiral method), 381
transformCS() (fe.FEModel method), 395
transformCS() (fe.Model method), 392
transformCS() (formex.Formex method), 116
transformCS() (geometry.Geometry method), 162
transformCS() (mesh.Mesh method), 202
transformCS() (polygon.Polygon method), 430
transformCS() (trisurface.TriSurface method), 459
translate() (coords.Coords method), 90
translate() (curve.Arc method), 375
translate() (curve.Arc3 method), 368
translate() (curve.BezierSpline method), 342
translate() (curve.CardinalSpline method), 348
translate() (curve.CardinalSpline2 method), 356
translate() (curve.Curve method), 320
translate() (curve.Line method), 332
translate() (curve.NaturalSpline method), 362
translate() (curve.PolyLine method), 326
translate() (curve.Spiral method), 381
translate() (fe.FEModel method), 396
translate() (fe.Model method), 392
translate() (formex.Formex method), 116
translate() (geometry.Geometry method), 160
translate() (lima.Lima method), 414
translate() (mesh.Mesh method), 202
translate() (polygon.Polygon method), 430
translate() (trisurface.TriSurface method), 460
TranslatedActor (class in actors), 265
translatem() (formex.Formex method), 111

transparent() (in module draw), 153
trfMatrix() (in module arraytools), 123
Triade (class in decors), 273
triangle() (in module simple), 208
triangleBoundingCircle() (in module geomtools), 225
triangleCircumCircle() (in module geomtools), 225
triangleInCircle() (in module geomtools), 225
triangleObtuse() (in module geomtools), 225
TriSurface (class in trisurface), 447
trisurface (module), 447
trl() (coords.Coords method), 100
trl() (curve.Arc method), 375
trl() (curve.Arc3 method), 368
trl() (curve.BezierSpline method), 342
trl() (curve.CardinalSpline method), 348
trl() (curve.CardinalSpline2 method), 356
trl() (curve.Curve method), 320
trl() (curve.Line method), 332
trl() (curve.NaturalSpline method), 362
trl() (curve.PolyLine method), 326
trl() (curve.Spiral method), 381
trl() (fe.FEModel method), 396
trl() (fe.Model method), 392
trl() (formex.Formex method), 116
trl() (geometry.Geometry method), 163
trl() (mesh.Mesh method), 202
trl() (polygon.Polygon method), 430
trl() (trisurface.TriSurface method), 460
turtle (module), 479

U

unchanged() (utils.DictDiff method), 214
uncompress() (project.Project method), 212
underlineHeader() (in module utils), 218
undoChanges() (objects.DrawableObjects method), 424
undoChanges() (objects.Objects method), 423
undraw() (in module draw), 151
uniformParamValues() (in module arraytools), 138
uniformParamValues() (in module nurbs), 420
union() (in module olist), 486
unique() (formex.Formex method), 109
uniqueOrdered() (in module arraytools), 128
uniqueRows() (in module arraytools), 131
unitAttractor() (in module mesh), 204
unitDivisor() (in module arraytools), 137
units (module), 481
UnitsSystem (class in units), 481
unitVector() (in module arraytools), 123
Unknown() (fe_post.FeResult method), 410
unProject() (camera.Camera method), 299
unProject() (canvas.Canvas method), 281
unProject() (viewport.QtCanvas method), 291
unQuote() (in module flatkeydb), 496
update() (canvas.CanvasSettings method), 277
update() (config.Config method), 493
update() (fe_abq.Interaction method), 399
update() (fe_abq.Output method), 398
update() (fe_abq.Result method), 398
update() (mydict.CDict method), 489
update() (mydict.Dict method), 488
update() (odict.KeyedList method), 490
update() (odict.ODict method), 490
update() (project.Project method), 212
update() (properties.Database method), 434
update() (properties.EdgeLoad method), 437
update() (properties.ElemLoad method), 437
update() (properties.ElemSection method), 436
update() (properties.MaterialDB method), 434
update() (properties.PropertyDB method), 438
update() (properties.SectionDB method), 435
update() (widgets.Table method), 251
updateButton() (in module toolbar), 310
updateData() (widgets.InputDialog method), 248
updateData() (widgets.ListSelection method), 253
updateDialogItems() (in module widgets), 257
updateGUI() (in module draw), 155
updateLightButton() (in module toolbar), 310
updateNormalsButton() (in module toolbar), 310
updatePerspectiveButton() (in module toolbar), 310
updateText() (in module widgets), 257
updateTransparencyButton() (in module toolbar), 310
updateWireButton() (in module toolbar), 310
userName() (in module utils), 220
utils (module), 212

V

value() (widgets.FileSelection method), 251
value() (widgets.InputBool method), 239
value() (widgets.InputButton method), 244
value() (widgets.InputColor method), 245
value() (widgets.InputCombo method), 240
value() (widgets.InputFile method), 245
value() (widgets.InputFloat method), 242
value() (widgets.InputFont method), 245
value() (widgets.InputFSlider method), 243
value() (widgets.InputGroup method), 246
value() (widgets.InputInfo method), 238

value() (widgets.InputInteger method), 242
value() (widgets.InputItem method), 237
value() (widgets.InputIVector method), 244
value() (widgets.InputLabel method), 238
value() (widgets.InputList method), 240
value() (widgets.InputPoint method), 244
value() (widgets.InputPush method), 241
value() (widgets.InputRadio method), 241
value() (widgets.InputSlider method), 243
value() (widgets.InputString method), 238
value() (widgets.InputTable method), 242
value() (widgets.InputText method), 239
value() (widgets.InputWidget method), 246
value() (widgets.ListSelection method), 252
value() (widgets.ProjectSelection method), 252
value() (widgets.SaveImageDialog method), 252
value() (widgets.Table method), 251
values() (odict.KeyedList method), 490
values() (odict.ODict method), 490
vascularsweepingmesher (module), 482
vectorLength() (in module arraytools), 134
vectorNormalize() (in module arraytools), 134
vectorPairAngle() (in module arraytools), 135
vectorPairArea() (in module arraytools), 134
vectorPairAreaNormals() (in module arraytools), 134
vectorPairCosAngle() (in module arraytools), 135
vectorPairNormals() (in module arraytools), 134
vectorRotation() (in module arraytools), 124
vectors() (curve.Line method), 335
vectors() (curve.PolyLine method), 322
vectors() (polygon.Polygon method), 427
vectorTripleProduct() (in module arraytools), 134
vertex() (dxf.DxfExporter method), 386
vertexDistance() (in module geomtools), 232
vertices() (trisurface.TriSurface method), 447
view() (formex.Formex method), 106
view() (in module draw), 151
viewIndex() (viewport.MultiCanvas method), 294
viewport (module), 284
viewport() (in module draw), 155
visualizeSubmappingQuadRegion() (in module vascularsweepingmesher), 483
volume() (mesh.Mesh method), 200
volume() (trisurface.TriSurface method), 448
volumes() (formex.Formex method), 113
volumes() (mesh.Mesh method), 199
volumes() (trisurface.TriSurface method), 476

W

w() (nurbs.Coords4 method), 417
wait() (in module draw), 153
wait_drawing() (viewport.QtCanvas method), 287
wait_selection() (viewport.QtCanvas method), 285
warning() (in module draw), 143
WarningBox (class in widgets), 254
WEBcolor() (in module colors), 157
WebGL (class in webgl), 484
webgl (module), 483
webgl() (trisurface.TriSurface method), 463
wheel_zoom() (viewport.QtCanvas method), 287
wheelEvent() (viewport.QtCanvas method), 288
whereProp() (formex.Formex method), 109
widgets (module), 236
wireMode() (in module draw), 153
withoutProp() (mesh.Mesh method), 192
withoutProp() (trisurface.TriSurface method), 470
withProp() (formex.Formex method), 109
withProp() (mesh.Mesh method), 192
withProp() (trisurface.TriSurface method), 470
write() (config.Config method), 493
write() (curve.Arc method), 378
write() (curve.Arc3 method), 371
write() (curve.BezierSpline method), 345
write() (curve.CardinalSpline method), 351
write() (curve.CardinalSpline2 method), 359
write() (curve.Curve method), 321
write() (curve.Line method), 335
write() (curve.NaturalSpline method), 365
write() (curve.PolyLine method), 329
write() (curve.Spiral method), 384
write() (dxf.DxfExporter method), 386
write() (export.ObjFile method), 388
write() (fe.FEModel method), 397
write() (fe.Model method), 393
write() (fe_abq.AbqData method), 399
write() (formex.Formex method), 113
write() (geometry.Geometry method), 163
write() (mesh.Mesh method), 203
write() (polygon.Polygon method), 431
write() (trisurface.TriSurface method), 448
write_neu() (in module neu_exp), 416
write_stl_asc() (in module filewrite), 235
write_stl_bin() (in module filewrite), 235
writeArray() (in module arraytools), 128
writeBCsets() (in module neu_exp), 415
writeCloads() (in module fe_abq), 407
writeCommaList() (in module fe_abq), 408
writeData() (in module filewrite), 234
writeDisplacements() (in module fe_abq), 407
writeDloads() (in module fe_abq), 408
writeDsloads() (in module fe_abq), 408

writeElemOutput() (in module fe_abq), 408
writeElemResult() (in module fe_abq), 409
writeElems() (in module fe_abq), 407
writeElems() (in module neu_exp), 415
writeFile() (flatkeydb.FlatDB method), 496
writeFileOutput() (in module fe_abq), 409
writeGeomFile() (in module script), 142
writeGroup() (in module neu_exp), 415
writeGTS() (in module filewrite), 235
writeHeading() (in module neu_exp), 415
writeIData() (in module filewrite), 235
writeNodeOutput() (in module fe_abq), 408
writeNodeResult() (in module fe_abq), 408
writeNodes() (in module fe_abq), 407
writeNodes() (in module neu_exp), 415
writeNodes() (in module tetgen), 445
writeOFF() (in module filewrite), 235
writeSection() (in module fe_abq), 407
writeSet() (in module fe_abq), 407
writeSmesh() (in module tetgen), 445
writeSTL() (in module filewrite), 235
writeSurface() (in module tetgen), 445
writeTetMesh() (in module tetgen), 445
writeTmesh() (in module tetgen), 445
writeToFile() (objects.DrawableObjects method), 426
writeToFile() (objects.Objects method), 423

X

x() (coords.Coords method), 84
x() (nurbs.Coords4 method), 417
xpattern() (in module coords), 102

Y

y() (coords.Coords method), 85
y() (nurbs.Coords4 method), 417

Z

z() (coords.Coords method), 85
z() (nurbs.Coords4 method), 417
zoom() (canvas.Canvas method), 281
zoom() (viewport.QtCanvas method), 291
zoomAll() (canvas.Canvas method), 282
zoomAll() (in module draw), 154
zoomAll() (viewport.QtCanvas method), 292
zoomArea() (camera.Camera method), 298
zoomBbox() (in module draw), 154
zoomCentered() (canvas.Canvas method), 282
zoomCentered() (viewport.QtCanvas method), 292
zoomRectangle() (canvas.Canvas method), 281
zoomRectangle() (in module draw), 154
zoomRectangle() (viewport.QtCanvas method), 291