pyFormex manual

pyFormex 0.8-a1

Benedict Verhegghe

May 12, 2009

CONTENTS

Preamble

This document is to become the pyFormex manual. As the pyFormex program itself is still under development, this document is by no means final and does not even pretend to be accurate for any version of pyFormex. However, since partial documentation is better than none, we decided to make this preliminary version available to the general public. This document may evolve fast, so check back regularly. For the most recent information and for topics that are not yet covered in this manual, we refer the user to the pydoc pages[1] that were automatically generated from the pyFormex source.

The contributors to this manual are Benedict Verhegghe, Tim Neels, Matthieu De Beule and Peter Mortier.

[1] http://pyformex.berlios.de/doc/index.html

CHAPTER

ONE

Introduction

This section 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 an easy task 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 manual and on the pyFormex website[1]. 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. While during development of the program we have concentrated mostly on wireframe type structures, surface and solid elements have been part of pyFormex right from the beginning. Still, many of the examples included with pyFormex are of frame type and most of the practical use of the program is in this area. There is also an extensive plugin for working with triangulated surfaces.

The stent[2] structure in the figure 1.1 is a good illustration of what pyFormex can do and what it was intended for. It is one of the many examples provided with pyFormex.

The structure is composed of 22032 line segments, each defined by 2 points. Nobody in his right mind would ever even try to input all the 132192 coordinates of all the points describing that structure. With pyFormex, one could define the structure by the following sequence of operations, illustrated in the figure 1.2:

- Create a nearly planar base module of two crossing wires. (The wires have a slight out-of-plane bend, to enable the crossing.)
- Extend the base module with a mirrored and translated copy.
- Replicate the base module in both directions of the base plane.

[1] http://pyformex.berlios.de

[2] A stent is a tube-shaped structure that is e.g. used to reopen (and keep open) obstructed blood vessels.

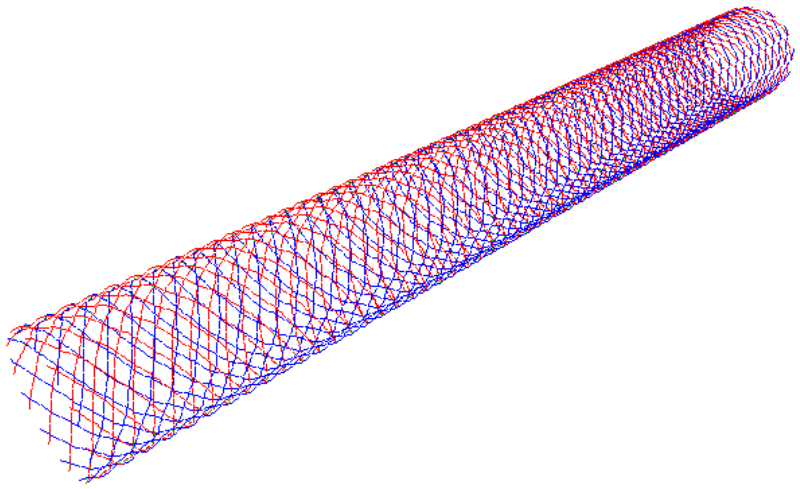


Figure 1.1: WireStent example.

- Roll the planar grid into a cylinder.

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

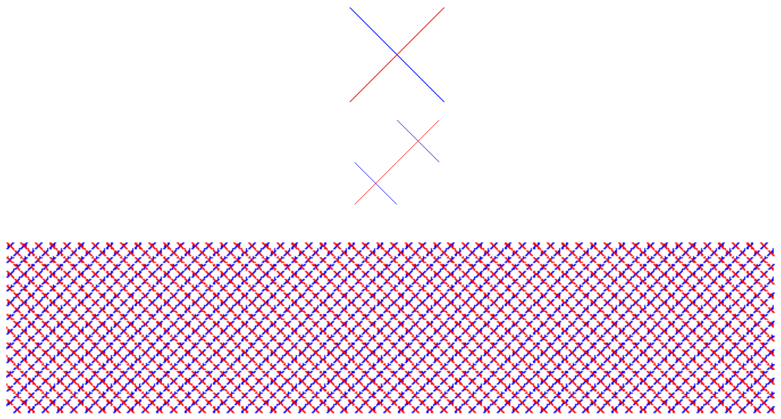


Figure 1.2: Steps in building the WireStent example.

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 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 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 behaviour 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 then 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 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 pyFormex scripting language and the elegance of Python have already attracted many on this path.

Finally, pyFormex is, and will always be, free software in both meanings of free: guaranteeing your freedom (see 1.2) and without charging a fee for it.

1.2 License and Disclaimer

pyFormex is ©2004-2008 Benedict Verhegghe

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

Full details of the license are available in appendix A, in the file COPYING included with the distribution and under the Help->License item of the pyFormex Graphical User Interface.

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

1.3.1 Prerequisites

In order to run pyFormex, you need to have the following installed (and working) on your computer.

- Python[4]: Version 2.4 or higher is recommended. Versions 2.3 and 2.2 might work with only a few minor changes. Nearly all Linux distributions come with Python installed, so this should not be no major obstacle.
- NumPy[5]: Version 1.0-rc1 or higher. Earlier versions can be made to work, but will require some changes to be made. NumPy is the package used for fast numerical array operations in Python and is essential for pyFormex.[6] On Linux systems, installing NumPy from source is usually straightforward. Debian users can install the package python-numpy.The extra packages python-numpy-ext and python-scipy give saom added functionality, but are not required for the basic operation of pyFormex. For Windows, binary packages are available on the Sourceforge download page[7].

If you only want to use the Formex data model and transformation methods, the above will suffice. But most probably you will also want to run the pyFormex Graphical User Interface (GUI) for visualizing your structures. Then you also need the following:

[3]http://www.gnu.org/licenses/gpl.html
[4]http://www.python.org
[5]http://numpy.scipy.org/
[6]The `Numarray` package, which was used up until pyFormex 0.3, is no longer supported
[7]http://www.numpy.org/

- Qt4[8]: The widget toolkit on which the GUI was built. For Debian users this comes in the packages python-qt4.
- PyQt4[9]: The Python bindings for Qt4. Debian users should install the packages python-qt4 and python-qt4-gl.
- PyOpenGL[10]: Python bindings for OpenGL, used for drawing and manipulating the 3D-structures. For debian users this is in the package python-opengl.

Since version 0.7, pyFormex includes an acceleration library that can increase the speed of some low level operations (e.g. drawing), especially when working on very large structures. If you want to compile and use this library, you also need to have a working C compiler and the Python and OpenGL header files. On a Debian based system, you should install tha packages python-dev, python-qt4-dev and libgl1-mesa-dev.

1.3.2 Downloading

The official releases of pyFormex can be downloaded from the pyFormex website[11]. As of the writing of this manual, the latest release is pyFormex 0.8-a1[12]. pyFormex is currently distributed in the form of a `.tar.gz` (tarball) archive. See 1.3.3 for how to proceed further.

Alternatively you can download the tarball releases from our local FTP server[13]. The server may be slower, but occasionally you can find there an interim release or release candidate not (yet) available on the official server.

Finally, you can also get the latest development code from the SVN repository on the pyFormex website[14]. If you have Subversion[15] installed on your system, you can just do

```
svn checkout svn://svn.berlios.de/pyformex/trunk pyformex
```

and the whole current pyFormex tree will be copied to a subdirectory `pyformex` on your current path.

Unless you want to help with the development or you absolutely need some of the latest features or bugfixes, the tarball releases are what you want to go for.

1.3.3 Installation on Linux platforms

Once you have downloaded the pyFormex tarball, unpack it with

```
tar xvzf pyformex-version.tar.gz
```

Then go to the created pyformex directory:

```
cd pyformex-version
```

and do (with root privileges)

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

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

[8] http://www.trolltech.com/products/qt
[9] http://www.riverbankcomputing.co.uk/pyqt/index.php
[10] http://pyopengl.sourceforge.net/
[11] http://pyformex.berlios.de
[12] http://prdownload.berlios.de/pyformex/pyformex.tar.gz
[13] ftp://bumps.ugent.be/pub/pyformex
[14] http://pyformex.berlios.de
[15] http://subversion.tigris.org/

1.3.4 Installation on Windows platforms

There is no installation procedure yet. All the pre-requisite packages are available for Windows, so in theory it is possible to run pyFormex on Windows. We know of some users who are running pyFormex succesfully using the `--nogui` option, i.e. without the Graphical User Interface (GUI). A few things may need to be changed for running the GUI on Windows. We might eventually have a look at this in the future, but it certainly is not our primary concern. Still, any feedback on (successful or not successful) installation attempts on Windows is welcome.

1.4 Running pyFormex

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

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

The pyFormex program takes some optional command line arguments, that modify the behaviour of the program. Appendix **??** 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, pyFormex 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 pyFormex GUI by starting pyformex with the command`pyformex -redirect`. The desktop starters installed by the pyFormex installation procedure use this option.

In some cases the user may want to use the mathematical power of pyFormex without the GUI. This is e.g. useful to run complex automated procedures from a script file. For convenience, pyFormex 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, pyFormex 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.

1.5 Getting started with the pyFormex GUI

While the pyFormex 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 first purpose of pyFormex will always remain to provide an framework for easily creating scripts to operate on geometries. Automization of otherwise tedious tasks is our main 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 pyFormex website were created that way.
- Changing pyFormex settings (though not everything can be changed through the GUI yet).
- Running pyFormex 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 pyFormex 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 pyFormex uses GNU Emacs[16], 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[17] is the language used in pyFormex 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 pyFormex 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 pyFormex 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 pyFormex 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.

1.6 Quick Python tutorial

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

There is ample documentation about Python freely available on the web[18]. If you are new to Python, but have already some programming experience, the Python tutorial[19] may be a good starting point. Or else, you can take a look at one of the beginners' guides[20].

Do not be 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.

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

1.7 Quick NumPy tutorial

Numerical Python (or NumPy for short) is an extension to the Python language providing efficient operations on large (numerical) arrays. pyFormex 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[21] already provides the basics.

[16] http://www.gnu.org/software/emacs
[17] http://www.python.org
[18] http://www.python.org
[19] http://docs.python.org/tut/
[20] http://wiki.python.org/moin/BeginnersGuide/
[21] http://www.scipy.org/Tentative_NumPy_Tutorial

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

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

1.8 How to proceed from here

By now, you should have enough basic knowledge about Python and NumPy to continue by studying the pyFormex tutorial in chapter 2.

CHAPTER

TWO

PyFormex tutorial

2.1 The pyFormex 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 let pyFormex re-execute it. You can easily change any geometrical parameter in any way you want: set directly, interactively ask the user, calculate from some formula, read from a file, ... angle, the radius of a dome, the ratio f/l of an arc. Using CAD, you would have often have to completely redo your drawing work. This idea of scripted geometry building is illustrated in figure 2.1: 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 pyFormex GUI gives access to many of its 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 restructed functionality.

2.2 Getting started

Warning: This section is outdated.

Note: This should include a short introduction to Python and Numpy

This section holds some basic information on how to use Python and pyFormex.

- Start the pyFormex GUI by entering the command *pyformex* in a terminal. Depending on your

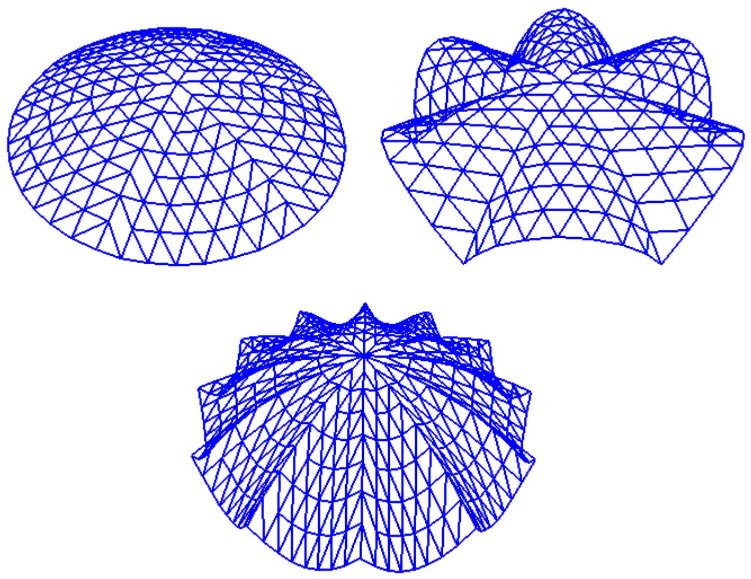


Figure 2.1: Same script, different domes

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.

- To create a new pyFormex-script, just open a new file with your favorite text editor and save it under a name with extension '.py'.
- A pyFormex script should start with a line `#!/usr/bin/env pyformex`
- To edit the script, you can
  - open it with your favorite text editor.
  - File > Open
    At this point, the script will be loaded but nothing will happen.
    File > Edit
    The script will now open in the default text editor. This default editor can be changed in the user configuration file (‘ /.pyformexrc’) or using the Settings > Commands menu option.
- To play a script, you can
  - File > Open
    File > Play
  - Type *pyformex myproject.py* in the terminal. This will start the pyFormex GUI and load your script at the same time.
    File > Play
  - To play a script without using the GUI (for example in finite element preprocessing, if you only want to write an output file, without drawing the structure), type *pyformex –nogui myproject.py*
- When writing a script in Python, there are some things you should keep in mind:
  - When using a function that requires arguments, an argument list must have any positional arguments followed by any keyword arguments, where the keywords must be chosen from the formal parameter names. It's not important whether a formal parameter has a default value or

not. No argument may receive a value more than once – formal parameter names corresponding to positional arguments cannot be used as keywords in the same calls.

Simply put: you can either set the arguments in the right order and only give their value, or you can give arguments by their name and value. This last option holds some advantages: not only is it easier to check what you did, but sometimes a function has many arguments with default values and you only want to change a few. If this isn't entirely clear yet, just look at the examples later in this tutorial or check the Python tutorial.

- Indentation is essential in Python. Indentation is Python's way of grouping statements. In straight-forward scripts, indentation is not needed (and forbidden!), but when using a for-statement for example, the body of the statement has to be indented. A small example might make this clear. Also notice the ':'

```
print 'properties'
for key, item in properties.iteritems():
    print key, item
```

- If you want to use functions from a seperate module (like `properties`), you add a line on top of the script

```
from properties import *
```

  All functions from that module are now available.

- The hash character, "#", is used to start a comment in Python.
- Python is case sensative.

• Python by default uses integer math on integer arguments!

2.3 Creating geometrical models

2.3.1 The Formex data model

The most important geometrical object in pyFormex is the `Formex` class. A `Formex` 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 in pyFormex will be numbered (0,1,2) for convenience. Higher order geometrical objects are defined by a collection of points. In pyFormex terms, we call the number of points of an object its *plexitude*.

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 numerical array[1] 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 2.2 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

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

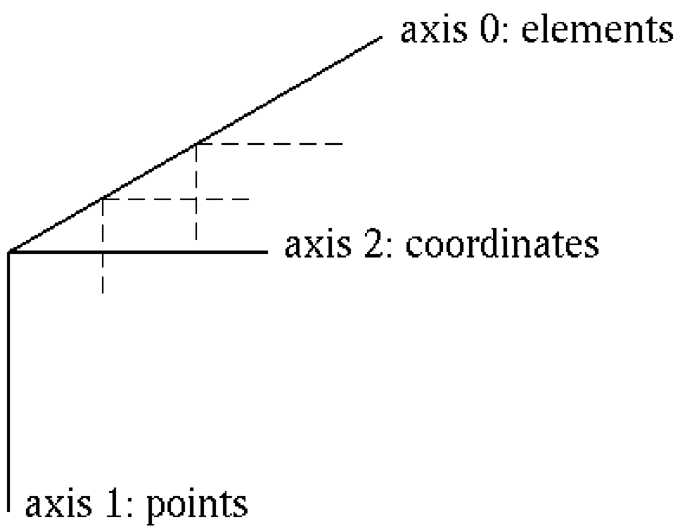


Figure 2.2: The structure of a Formex

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 $(n, 3)$, conttaining all the points of the element. Then, `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 floating point value representing a single coordinate of that point.

2.3.2 Creating a Formex

Creating a Formex using coordinates

The first and most useful way to create a Formex is by specifying it's nodes and elements in a 3D-list.

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

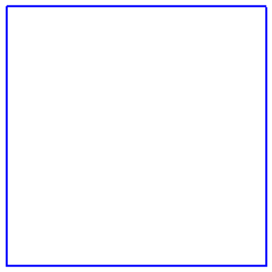


Figure 2.3: A very simple Formex

This creates a Formex F, which has the nodes (0,0), (1,0), (1,1) and (0,1). These nodes are all part of a single element, thus creating a square plane. This element is also the entire Formex. On the other hand, if you would change the position of the square brackets like in the following example, then you'd create a Formex F which is different from the previous. The nodes are the same, but the connection is different. The nodes (0,0) and (1,0) are linked together by an element, and so are the nodes (1,1) and (0,1). The Formex is now a set of 2 parallel bars, instead of a single square plane.

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

Figure 2.4: Same nodes, different Formex

If we want to define a Formex, similar to the square plane, but consisting of the 4 edges instead of the actual plane, we have to define four elements and combine them in a Formex. This is *not* the same Formex as fig 2.3, although it looks exactly the same.

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

The previous examples were limited to a 2-D environment for simplicity's sake. Of course, we could add a third dimension. For instance, it's no problem defining a pyramid consisting of 8 elements ('bars').

```
F=Formex([[[0,0,0],[0,1,0]], [[0,1,0],[1,1,0]], [[1,1,0],[1,0,0]], [[1,0,0],
[0,0,0]], [[0,0,0],[0,1,0]], [[0,0,0],[0.5,0.5,1]], [[1,0,0],[0.5,0.5,1]],
[[1,1,0], [0.5,0.5,1]], [[0,1,0],[0.5,0.5,1]]])
```

However, as you can see, even in this very small example the number of nodes, elements and coordinates you have to declare becomes rather large. Defining large Formices using this method would not be practical. This problem is easily overcome by copying, translating, rotating,... a smaller Formex — as will be explained in 2.3.7 — or by using patterns.

Creating a Formex using patterns

Another way of creating a Formex, is by using the coordinate generating functions `pattern` and mpattern. These functions create a series of coordinates from a simple string, by interpreting each of the characters of the string as a single unit step in one of the cordinate directions, or as some other simple action. These functions thus are very valuable in creating geometry where the points lie on a regular grid.

In this case, a line segment pattern is created from a string.

The function `pattern(s)` creates a list of line segments where all nodes lie on the gridpoints of a regular grid with unit step. The first point of the list is [0,0,0]. Each character from the given string *s* is interpreted

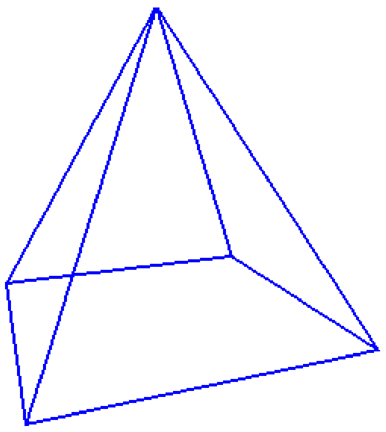


Figure 2.5: A pyramid

as a code specifying how to move to the next node. Currently defined are the following codes:
0 = goto origin [0,0,0]
1..8 move in the x,y plane
9 remains at the same place
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. The special character '\' can be put before any character to make the move without making a connection. The effect of any other character is undefined.

This method has important restrictions, since it can only create lines on a regular grid. However, it can be a much easier and shorter way to define a simple Formex. This is illustrated by the difference in length between the previous creation of a square and the next one, although they define the same Formex (figure 2.3).

```
F=Formex(pattern('1234'))
```

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

```
#!/usr/bin/env pyformex
from simple import *
c=Formex(pattern(Pattern['cube']))
clear();draw(c)
```

Creating a Formex using coordinates from a file

Warning: This section is outdated.

In some cases, you might want to read coordinates from a file an combine them into a Formex. This is possible with the module `file2formex` and it's function `fileFormex()`. Each point is connected to the following, forming an element (bar).

The next file ('square.txt') would create the same square as before(figure 2.3).

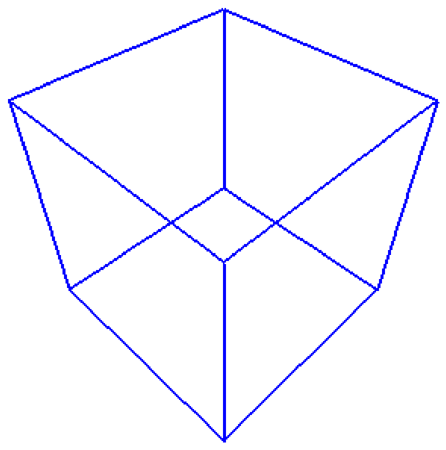


Figure 2.6: A cube

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

```
#!/usr/bin/env pyformex
from file2formex import *
F=fileFormex('square.text', closed='yes')
```

2.3.3 Drawing a Formex

Warning: This section is outdated.

Of course, you'd want to see what you have created. This is accomplished by the function `draw()`. The next example creates figure 2.5.

```
F=Formex([[[0,0,0],[0,1,0]], [[0,1,0],[1,1,0]], [[1,1,0],[1,0,0]], [[1,0,0],
[0,0,0]], [[0,0,0],[0,1,0]], [[0,0,0],[0.5,0.5,1]], [[1,0,0],[0.5,0.5,1]],
[[1,1,0], [0.5,0.5,1]], [[0,1,0],[0.5,0.5,1]]])
draw(F)
```

It also possible to draw multiple Formices at the same time.

```
from simple import *
F=Formex([[[0,0,0],[0,1,0]], [[0,1,0],[1,1,0]], [[1,1,0],[1,0,0]], [[1,0,0],
[0,0,0]], [[0,0,0],[0,1,0]], [[0,0,0],[0.5,0.5,1]], [[1,0,0],[0.5,0.5,1]],
[[1,1,0],[0.5,0.5,1]], [[0,1,0],[0.5,0.5,1]]]).setProp(1)
G=Formex(pattern(Pattern['cube'])).setProp(3)
draw(F+G)
```

It might be important to realize that even if you don't draw a particular Formex, that doesn't mean you didn't create it!

Now, when you are creating a large geometry, you might be interested in seeing the different steps in the

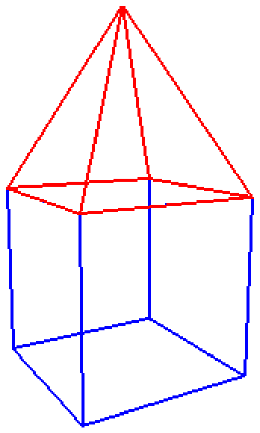


Figure 2.7: Drawing multiple Formices

creation. To remove all previously drawn Formices, you can use `clear()` what sweepes the screen clean. If you want to see a certain step in the creation longer than the default time, use `sleep(t)`, with *t* the delay (in seconds) before executing the next command.

```
F=Formex(pattern('164'))
draw(F)
G=F.replic(5,1,0)
clear()
draw(G)
```

2.3.4 Adding 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 `p` 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 modelling 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 Formex element it was generated from.

To create a Formex with property numbers, just specify them as a second argument in the constructor. The following example creates a Formex consisting of two triangles, one with property number 1, the second with property 3. The following lines show the creation of the four Formices displayed in figure 2.8, where elements with proprty value 1 are shown in red, those with property value 3 are shown in blue.

```
>>> F0 = Formex(mpattern('12-34'),[1,3])
>>> F1 = F0.replic2(4,2)
>>> F2 = F1 + F1.mirror(1)
>>> F3 = F2 + F2.rotate(180.,1)
```

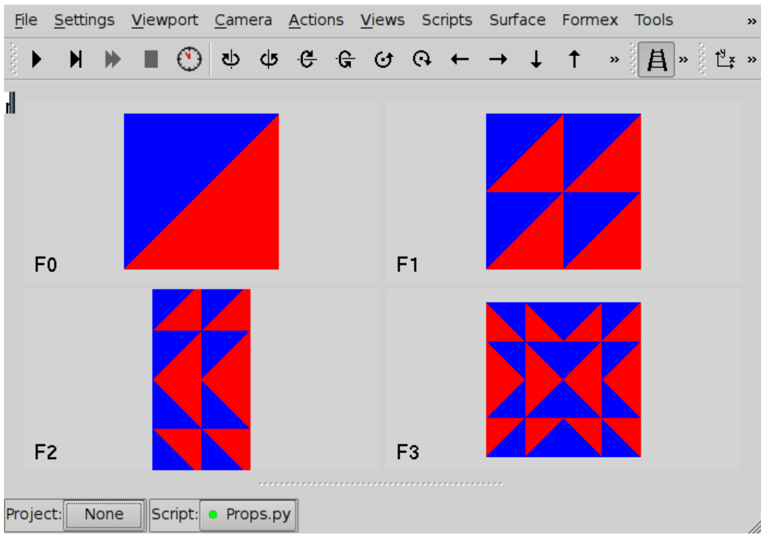


Figure 2.8: A Formex with property numbers drawn as colors

To create the properties on a Formex without, you should always use the `setProp` method. This ensures that the properties array is generated with the correct type and shape. If needed, the supplied values will be repeated to match the number of elements in the Formex. Once the `p` attribute is created, you can safely change the value of any of the property numbers.

```
>>> F = Formex(mpattern('12-34-32-14'))
>>> F.setProp([1,3])
>>> print F.p
    [1 3 1 3]
>>> F.p[2] = 5
>>> print F.p
    [1 3 5 3]
```

When drawing a Formex having property numbers with 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.

2.3.5 Saving images

Warning: This section is outdated.

After drawing the Formex, you might want to save the image. This is very easy to do:
File > Save Image
The filetype should be 'bmp', 'jpg', 'pbm', 'png', 'ppm', 'xbm', 'xpm', 'eps', 'ps', 'pdf' or 'tex'.

To create a better looking picture, several settings can be changed:

- Change the background color Settings > Background Color
- Use a different (bigger) linewidth Settings > Linewidth
- Change the canvas size. This prevents having to cut and rescale the figure with an image manipulation program (and loosing quality by doing so). Settings > Canvas Size

It is also possible to save a series of images. This can be especially useful when playing a script which creates several images, and you would like to save them all. For example, figure 1.2, which shows the different steps in the creation of the WireStent model, was created this way.
File > Toggle MultiSave

2.3.6 Information about a Formex

The Formex class has several methods related to abtaining information obout the object. We refer to the reference manual in chapter 8 for a full list. Some of the most interesting and often used ones are:

Function	Description
`F.nelems()`	Return the number of elements in the Formex.
`F.nplex()`	Return the plexitude (the number of point in each element) of the Formex.
`F.prop()`	Return the properties array (same as F.p).
`F.bbox()`	Return the bounding box of the Formex.
`F.center()`	Return the center of the Formex.

2.3.7 Changing the Formex

Until now, we've only created simple Formices. The strength of pyFormex however is that it is very easy to generate large geometrical models by a sequence of mathematical transformations. After initiating a basic Formex, it's possible to transform it by using copies, translations, rotations, projections,...

There are many transformations available, but this is not the right place to describe them all. This is what the reference manual in chapter 8 is for. A summary of all possible transformations and functions can be found there.

To illustrate some of these transformations and the recommended way of writing a script, we will analyse some of the examples. More of these interesting examples are found in 'installdir/examples'. Let's begin with the example 'Spiral.py'.

```
#!/usr/bin/env pyformex
# $Id$
##
## This file is part of pyFormex 0.3 Release Mon Feb 20 21:04:03 2006
## pyFormex is a python implementation of Formex algebra
## Homepage: http://pyformex.berlios.de/
## Distributed under the GNU General Public License, see file COPYING
## Copyright (C) Benedict Verhegghe except where stated otherwise
##
#
"""Spiral"""

m = 36 # number of cells along torus big circle
n = 10 # number of cells along torus small circle
```

```
def drawit(F,view='front'):
    clear()
    draw(F,view)

F = Formex(pattern("164"),[1,2,3]); drawit(F)
F = F.replic(m,1,0); drawit(F)
F = F.replic(n,1,1); drawit(F)
F = F.translate1(2,1); drawit(F,'iso')
F = F.cylindrical([2,1,0],[1.,360./n,1.]); drawit(F,'iso')
F = F.replic(5,m,2); drawit(F,'iso')
F = F.rotate(-10,0); drawit(F,'iso')
F = F.translate1(0,5); drawit(F,'iso')
F = F.cylindrical([0,2,1],[1.,360./m,1.]); drawit(F,'iso')
drawit(F,'right')
```

During this first read-through, you will have noticed that every step is drawn. Of course, this is not necessary, but it can be useful. And above all, it is very educational for use in a tutorial...

The next important thing is that parameters were used. It's recommended to always do this, especially when you want to do a parametric study of course, but it can also be very convenient if at some point you want to change the geometry (for example when you want to re-use the script for another application).

A simple function `drawit()` is defined for use in this script only. This function only provides a shorter way of drawing Formices, since it combines `clear()` and `draw`.

Now, let's dissect the script.

```
def drawit(F,view='front'):
    clear()
    draw(F,view)
```

This is a small function that is only defined in this script. It clears the screen and draws the Formex at the same time.

```
m = 36 # number of cells along torus big circle
n = 10 # number of cells along torus small circle
```

These are the parameters. They can easily be changed, and a whole new spiral will be created without any extra effort. The first step is to create a basic Formex. In this case, it's a triangle which has a different property number for every edge.

```
F = Formex(pattern("164"),[1,2,3]); drawit(F)
```

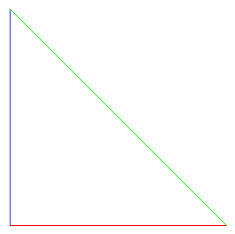


Figure 2.9: The basic Formex

This basic Formex is copied 'm' times in the 0-direction with a translation step of '1' (the length of an edge of the triangle). After that, the new Formex is copied 'n' times in the 1-direction with a translation step of '1'. Because of the recursive definition (F=F.replic), the original Formex F is overwritten by the transformed one.

```
F = F.replic(m,1,0); drawit(F)
F = F.replic(n,1,1); drawit(F)
```

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 cilinder. The result of all previous steps is a rectangular pattern with the desired dimensions, in a plane z=1.

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

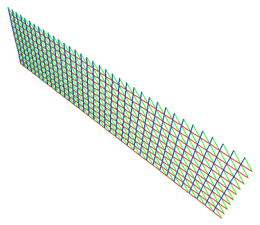


Figure 2.10: The rectangular pattern

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

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

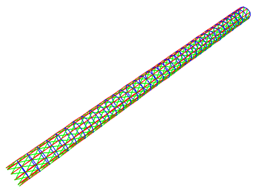


Figure 2.11: The cylinder

This cilinder is copied 5 times in the 2-direction with a translation step of 'm' (the lenght of the cilinder).

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

The next step is to rotate this cilinder -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'.

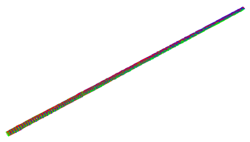


Figure 2.12: The new cylinder

```
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')
```

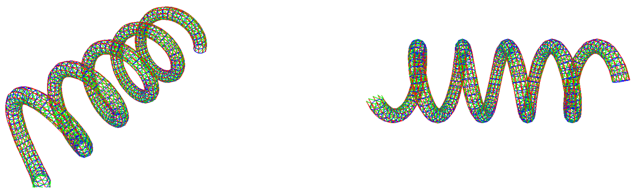


Figure 2.13: The spiral

2.3.8 Converting a Formex to a Finite Element model

The `feModel()` method 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, theses points have to be merged into a single nod, to express the continuity of the material. This is exactly what`feModel()` does. It returns a tuple of two numpy arrays (nodes,elems), where

- nodes is a float array with shape (?,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. The elements and their nodes are in the same order as in F.

```
>>> from simple import *
>>> F = Formex(pattern(Pattern['cube']))
>>> draw(F)
>>> nodes,elems = F.feModel()
>>> print 'Nodes',nodes
>>> print 'Elements',elems

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 reverse operation of transforming a finite element model back into a Formex is 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` funcion in the second line tests that all coordinates in bopth arrays are the same, within a small tolerance.

2.4 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 proprty 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()
```

2.4.1 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 2.4.4). 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 `ProeprtyDB` 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 2.4.2.

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.street
  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 away 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.

2.4.2 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.

2.4.3 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 pyFormex 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 2.4.4), which are attributed to a single point in space, and element properties (2.4.5), 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.

2.4.4 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.

2.4.5 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 generatr.

eltype This is the single most import 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, pyFormex 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.

2.4.6 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.

2.5 Exporting to finite element programs

CHAPTER

THREE

The Canvas

3.1 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 pyFormex 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 pyFormex 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 3.5. The remainder of this chapter will treat the canvas as if it was a single viewport.

pyFormex 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 pyFormex 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, pyFormex 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.

3.2 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 this 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, pyFormex will draw it with the current drawing color. If the Formex has properties, pyFormex 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.

3.3 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 4.5. 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. pyFormex 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.

3.4 Other canvas items

3.4.1 Actors

3.4.2 Marks

3.4.3 Decorations

3.5 Multiple viewports

Drawing in pyFormex 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.

CHAPTER

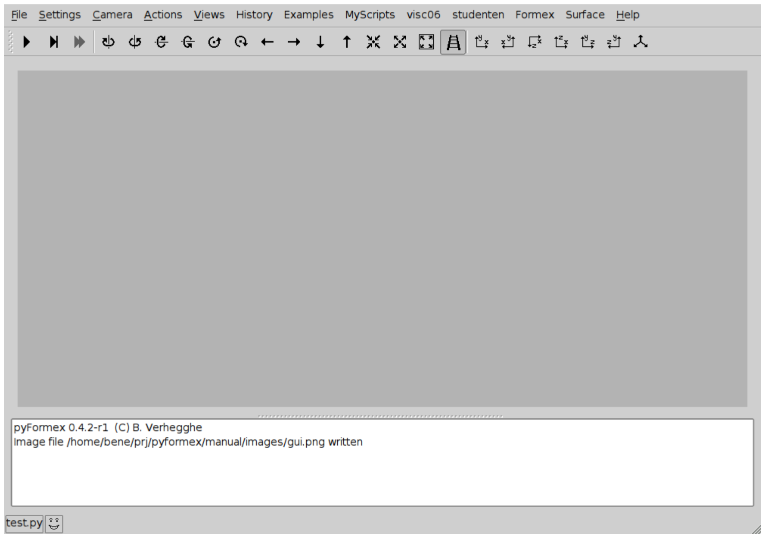
FOUR

The Graphical User Interface

4.1 Starting the GUI

You start the pyFormex GUI by entering the command `pyformex -gui`. Depending on your installation, you may also have a panel or menu button on your desktop from which you can start the pyFormex graphical interface by a simple mouse click.

When the main window appears, it will look like the one shown in the figure 4.1. 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.



```
pyFormex 0.4.2-r1  (C) B. Verhegghe
Image file /home/bene/prj/pyformex/manual/images/gui.png written
```

Figure 4.1: The pyFormex main window

4.2 Basic use of the GUI

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

The pyFormex window (figure 4.1) 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 pyFormex 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 pyFormex 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 pyFormex displays informative messages, requested results, possibly also errors and any text that your pyFormex 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 pyFormex 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 pyFormex main window can be resized in the usual ways.

4.3 The file menu

4.4 The viewport menu

4.5 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.

	LEFT	MIDDLE	RIGHT
NONE	rotate	pan	zoom
SHIFT			
CTRL			
ALT	rotate	pan	zoom

During picking operations, the mouse bindings are changed as follows:

	LEFT	MIDDLE	RIGHT
NONE	set		done
SHIFT	add		
CTRL	remove		
ALT	rotate	pan	zoom

4.5.1 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.

4.5.2 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).

4.6 Customizing the GUI

Some parts of the pyFormex 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 pyFormex GUI according to their wishes. If you send us your modifications, we might even include them in the official distribution.

4.6.1 Changing the appearance of the GUI

4.6.2 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 pyFormex 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 pyFormex 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!), pyFormex will itself try to detect the path to the installed examples.

4.6.3 Adding custom menus

When you start using pyFormex 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 pyFormex 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 pyFormex GUI with custom menus to execute some parts of your script.

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

CHAPTER

FIVE

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 pyFormex 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 pyFormex 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 pyFormex (still under development) does not allow such distinction yet. Still, since the topic is so important to the user and the configuration system in pyFormex is already quite complex, we tought it was necessary to provide already some information on how to configure pyFormex. Be aware though that important changes to this system will likely occur.

5.1 pyFormex configuration files

On startup, pyFormex 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 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 pyFormex to correctly start up.
- `/etc/pyformex`: this file can be used by the system administrator to make system-wide changes to the pyFormex installation. This could e.g. be used to give all users at a site access to a common set of scripts or extensions.
- `ṗyformexrc`: this is where the user normally stores his own default settings.
- `CURRENT-DIR/.pyformex`: if the current working directory from which pyFormex 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 pyFormex 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,pyFormex 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 pyFormex program. A user configuration file will be generated if none existed.

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!

5.2 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 pyFormex was installed (and which is also reported by the `pyformex --whereami` command).

5.3 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 pyFormex 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 pyFormex scripts and these will be added in the pyFormex menu under an item named name.
  - `autorun = '.pyformex.startup'`: name of a pyFormex 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 pyFormex website.
  - `uselib = False`: do not use the pyFormex 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 pyFormex installation. To autoload user extensions from a different place, the `autorun` script can be used.

CHAPTER

SIX

PyFormex example scripts

6.1 Creating geometry

To explain the modus operandi of pyFormex, the 'WireStent.py' script is parsed step by step. To start, all required modules to run the 'WireStent.py' script are imported (e.g. the `math` module to use the mathematical constant π). 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.

```
from formex import *

class DoubleHelixStent:
    """Constructs a double helix wire stent.

    A stent is a tubular shape such as used for opening obstructed
    blood vessels. This stent is made frome sets of wires spiraling
    in two directions.
    The geometry is defined by the following parameters:
      L  : length of the stent
      De : external diameter of the stent
      D  : average stent diameter
      d  : wire diameter
      be : pitch angle (degrees)
      p  : pitch
      nx : number of wires in one spiral set
      ny : number of modules in axial direction
      ds : extra distance between the wires (default is 0.0 for
           touching wires)
      dz : maximal distance of wire center to average cilinder
      nb : number of elements in a strut (a part of a wire between two
           crossings), default 4
    The stent is created around the z-axis.
    By default, there will be connectors between the wires at each
    crossing. They can be switched off in the constructor.
    The returned formex has one set of wires with property 1, the
    other with property 3. The connectors have property 2. The wire
    set with property 1 is winding positively around the z-axis.
    """
```

The constructor `__init__` of the `DoubleHelixStent` class requires 8 arguments:

- 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.

6.1.1 Creating the base module

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.

```
    def __init__(self,De,L,d,nx,be,ds=0.0,nb=4,connectors=True):
        """Create the Wire Stent."""
        D = De - 2*d - ds
        r = 0.5*D
        dz = 0.5*(ds+d)
        p = math.pi*D*tand(be)
        nx = int(nx)
        ny = int(round(nx*L/p))  # The actual length may differ
a bit from L
```

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 Figure 6.1.

```
        # a single bumped strut, oriented along the x-axis
        bump_z=lambda x: 1.-(x/nb)**2
        base = Formex(pattern('1')).replic(nb,1.0).bump1
(2,[0.,0.,dz],bump_z,0)
```

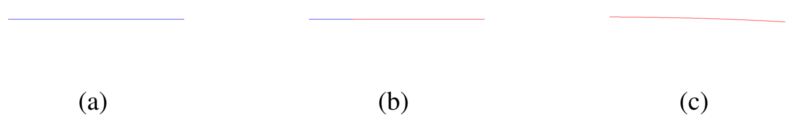


Figure 6.1: Creation of single bumped strut (c) from a straight (a) and replicated (b) line segment.

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: $y_2 = y_1 + skew * x_1$. 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 6.2 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 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.

```
        # scale back to size 1.
        base = base.scale([1./nb,1./nb,1.])
        # NE and SE directed struts
        NE = base.shear(1,0,1.)
        SE = base.reflect(2).shear(1,0,-1.)
        NE.setProp(1)
        SE.setProp(3)
        # a unit cell of crossing struts
        cell1 = (NE+SE).rosette(2,180)
        # add a connector between first points of NE and SE
        if connectors:
            cell1 += Formex([[NE[0][0],SE[0][0]]],2)
```

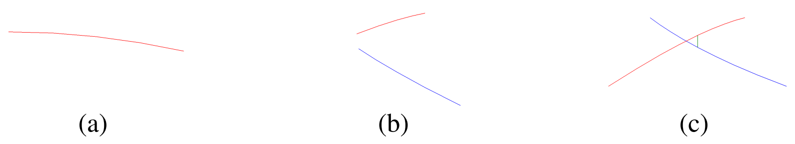


Figure 6.2: Creation of unit cell of crossing and connected struts (c) from a rescaled (a) and mirrored, skewed (b) bumped strut.

6.1.2 Extending the base module

Subsequently, a mirrored copy of the base cell is generated. 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 6.3. 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.

```
# and create its mirror
cell2 = cell1.reflect(2)
# 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]
```

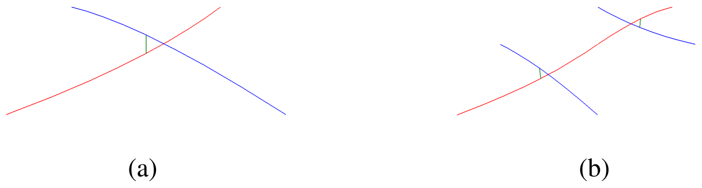


Figure 6.3: Creation of the complete extended base module (b) from the original and mirrored (a) unit cell.

6.1.3 Full nearly planar pattern

The fully nearly planar pattern is obtained by copying the base module in two directions and shown in Figure 6.4. `replic2()` generates this pattern with nx and ny replications with steps dx and dy in respectively, the default X- and Y-direction.

```
# Create the full pattern by replication
dx = 4.
dy = 4.
F = (self.cell1+self.cell2).replic2(nx,ny,dx,dy)
```

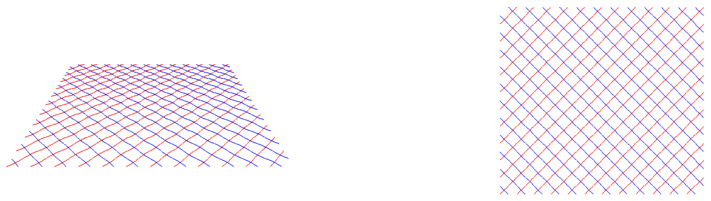


Figure 6.4: Creation of the fully nearly planar pattern.

6.1.4 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 6.5.

```
# 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
```

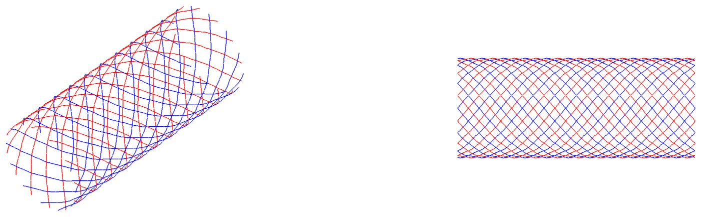

(a) (b)

Figure 6.5: Creation of the cylindrical stent structure ((a) iso and (b) right view).

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.

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

6.1.5 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 Figure 6.6: 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.

```
from examples.WireStent import DoubleHelixStent

for i in [16.,32]:
    for j in [6,10]:
        for k in [25,50]:
            stent = DoubleHelixStent(i,40.,0.22,j,k).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 pyFormex provides a multitude of features (such as parametric modeling, finite element pre- and postprocessing, optimization strategies, etcetera) in one sinlge consistent environment, it appearss to be the obvious way to go when studying the mechanical behavior of braided wire stents.

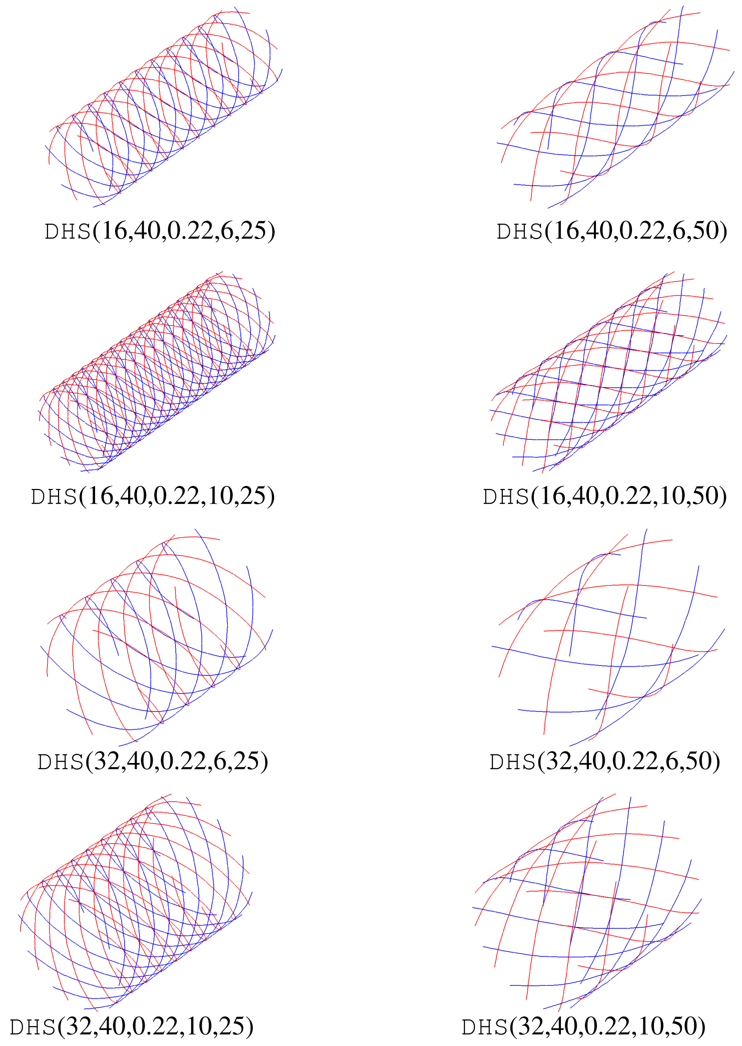


Figure 6.6: Variations on the wire stent geometry using the DoubleHelixStent(De, L, d, nx, β) (DHS() class.

6.2 Operating on surface meshes

Besides being used for creating geometries, pyFormex 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 pyFormex.

6.2.1 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 6.7 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.

The structure of such a device can be quite complex and difficult to analyse. The same functions pyFormex offers for creating geometries can also be employed to investigate triangulated meshes. A simple unroll

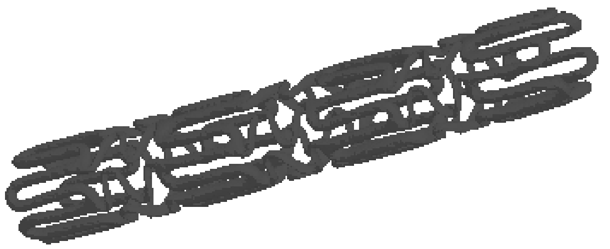


Figure 6.7: Triangulated mesh of a stent

operation of the stent gives a much better overview of the complete geometrical structure and allows easier analysis (see figure 6.8).

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

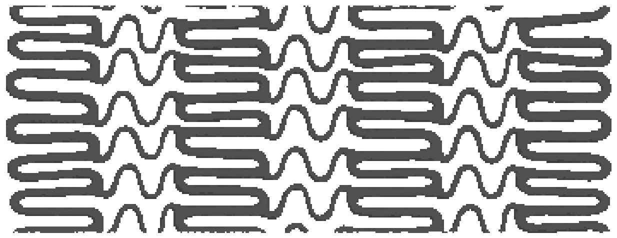


Figure 6.8: Result of unroll operation

This 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 figure 6.9.

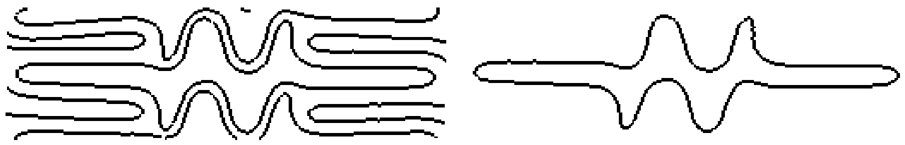


Figure 6.9: Intersection of stent cell with plane and inner line of stent cell

Finally, the `length()` function returns the circumference of the cell, which is 9.19 mm.

CHAPTER

SEVEN

PyFormex plugins

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

7.1 What are pyFormex plugins?

From its inception pyFormex was intended to be easily expandable. Its open architecture allows educated users to change the behavior of pyFormex 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 pyFormex, any set of functions that are not an essential part of pyFormex can be called a 'plugin', if its functionality can usefully be called from elsewhere and if the source can be placed inside the pyFormex distribution.

Thus, we distinct plugins from the vital parts of pyFormex 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) pyFormex 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 pyFormex 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 pyFormex 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 pyFormex.

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

7.2 How to create a pyFormex plugin.

CHAPTER

EIGHT

PyFormex — reference manual

This is the reference manual for pyFormex.

The definitions in the modules `coords`, `formex`, `script` and (for GUI applications) `draw` are available to your pyFormex scripts without explicitly importing them. Also available is the complete `numpy` namespace.

All the other modules need to be accessed using the normal Python import statements.

8.1 `formex` — General 3D geometry.

This module contains all the basic functionality for creating and transforming general 3D geometrical objects.

The largest and most important part of this module is the Formex class. Most algorithms in this and other modules of pyFormex are implemented to operate on data structures of this type. Some functions however operate on NumPy's ndarray type objects.

8.1.1 The Formex class

This class represents a structured set of 3D coordinates. Basically, a `Formex` is a three dimensional array[1] of float values. The array has a shape `(nelems,nplex,3)`. Each slice `[i,j]` of the array contains the three coordinates of a point in space. Each slice `[i]` of the array contains a connected set of *nplex* points: we will refer to it as an *element*. The number of points in an element is also called the *plexitude* of the element or the plexitude of the Formex (since all elements in a Formex have the same number of points).

The Formex class does not attribute any geometrical interpretation to the grouping of points in an element. This is left to the user, or to subclasses.

Thus, an element consisting of two points could (and usually will) represent a straight line segment between this two points. But the points could just as well be the two opposite corners of a rectangle. An element with plexitude 3 (in short: plex-3 element) could be interpreted as a triangle or as two straight line segments or as a curved line or even as a circle through these points. If it is a triangle, it could be either the circumference of the triangle or the part of the plane inside that circumference. As far as the Formex class concerns, each element is just a set of points.

All elements in a `Formex` must have the same number of points, but you can construct `Formex` instances with any (positive) number of nodes per element. A plex-1 `Formex` is just a collection of unconnected nodes (each element is a single point).

[1]Currently, this is implemented as a NumPy ndarray object. If you want to do complex things in pyFormex, some basic knowledge of NumPy may be required .

One way of attaching other data (than the coordinates of its points) to the elements of a `Formex`, is by using the 'property' attribute. The property is an array holding one integer value for each of the elements of the Formex. The interpretation of this property value is again completely left to the user. It could be a code for the type of element, or for the color to draw this element with. Often it will be used as an index into some other (possibly complex) data structure holding all the characteristics of that element. By including this property index into the Formex class, we make sure that when new elements are constructed from existing ones, the element properties are automatically propagated.

class `Formex` (*data=[[[]]],prop=None*)

Create a new Formex from the specified data and optional property values. The data can either be: another Formex instance, a string that can be interpreted by the pattern() method to create coordinates, or a 2D or 3D array of coordinates or a structure (e.g. compound list) that can be transformed to such an array.

If 2D coordinates are given, a 3-rd coordinate 0.0 will be added. Internally, a Forme always has 3D coordinates.

Example: `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.

If a prop argument is specified, the setProp() method will be called with this argument to assign the properties to the Formex.

If no data are specified, an empty Formex is created. The use of empty Formices is not very well tested yet, and therefore not not encouraged.

8.1.2 Formex class members

`f`

The attribute *f* holds the coordinates of all points in the Formex. It is a `Coords` type object with shape (nelems,nplex,3), where *nplex* is nonzero. An empty Formex has *nelems* == 0.

The class offers methods to access the coordinates in groups or individually. For performance reasons, the attribute can be changed directly, but you should make sure that *f* always stays a `Coords` type object with proper shape.

`p`

Either an integer array with shape (nelems,), or None. If not None, an integer value is attributed to each element of the Formex. There is no provision to attribute different values to the separate nodes of an element. If you need such functionality, use the *p* array as a pointer into a data structure that has different values per node.

The *p* is called property number or property for short. If it is not None, it will take part in the Formex transformations and its values will propagate to all copies created from the Formex elements.

8.1.3 Basic access methods

`__getitem__` (*i*)

This is equivalent to `self.f.__getitem__(i)`. It allows to access the data in the coordinate array *f* of the Formex with all the index methods of numpy. The result is an float array or a single float. Thus: `F[1]` returns the second element of *F*, `F[1,0]` the first point of that element and `F[1,0,2]` the z-coordinate of that point. `F[:,1]` is an array with the second point of all elements. `F[:,:,1]` is the y-coordinate of all points of all elements in the Formex.

`__setitem__` (*i,val*)

This is equivalent to `self.f.__getitem__(i)`. It allows to change individual elements, points or coordinates using the item selection syntax. Thus: `F[1:5,1,2] = 1.0` sets the z-coordinate of the second points of the elements 1, 2, 3 and 4 to the value 1.0.

element (*i*)
Returns the element *i*. `F.element[i]` is currently equivalent with `F[i]`.

point (*i,j*)
Returns the point *j* of the element *i*. `F.point[i,j]` is equivalent with `F[i,j]`.

coord (*i,j,k*)
Returns the coordinate *k* of the point *j* of the element *i*. `F.coord[i,j,k]` is equivalent with `F[i,j,k]`.

8.1.4 Methods returning information

nelems ()
Returns the number of elements in the Formex.

nplex ()
Returns the number of points in each element.

ndim (*self*)
Returns 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 *nelems()* and *nplex()*.

shape ()
Return the shape of the Formex.

The shape of a Formex is the shape of its data array, i.e. a tuple (nelems, nplex, ndim).

data ()
Return the Formex as a numpy array. 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.

x ()
Return the x-plane.

y ()
Return the y-plane.

z ()
Return the z-plane.

prop ()
Return the properties as a numpy array, or None if the Formex has no properties.

maxprop ()
Return the highest property used, or None if the Formex has no properties.

propSet ()
Return a list with the unique property values on this Formex, or None if the Formex has no properties.

bbox ()
Return the bounding box of the Formex.

The bounding box is the smallest rectangular volume in global coordinates, such that no points of the Formex are outside the box. It is returned as a [2,3] array: the first row holds the minimal coordinates and the second one the maximal.

center()
Return the center of the Formex. This is the center of its `bbox()`.

centroid()
Return the centroid of the Formex. This is the point whose coordinates are the mean values of those of all the pointsof the Formex.

sizes()
Returns an array with shape (3,) holding the length of the bbox along the 3 axes.

diagonal()
Return the length of the diagonal of the bbox().

bsphere()
Return the diameter of the bounding sphere of the Formex.

The bounding sphere is the smallest sphere with center in the `center()` of the Formex, and such that no points of the Formex are lying outside the sphere. It is not necessarily the smallest sphere surrounding all points of the 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 plex-1 Formex.

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

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 (nelems(),nplex()) shaped array with the distance of each point to the plane containing the point p and having normal n. Distance values are positive if the point is on the side of the plane indicated by the positive normal.

distanceFromLine(*p,q*)
Return the distance from the line (p,q) for all points of the Formex.

p and q are two points specified by 3 coordinates.

The return value is a (nelems(),nplex()) shaped array with the distance of each point to the line through p and q. All distance values are positive or zero.

distanceFromPoint(*p*)
Return the distance from the point p for all points of the Formex.

p is a point specified by 3 coordinates.

The return value is a (nelems(),nplex()) shaped array with the distance of each point to the line through p and q. All distance values are positive or zero.

feModel(*nodesperbox=1,repeat=True,rtol=1.e-5,atol=1.e-5*)
Return a tuple of nodal coordinates and element connectivity.

A tuple of two arrays is returned. The first is a 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 = feModel(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.

rtol and *atol* are the relative, resp. absolute tolerances used to decide whether any nodal coordinates are considered to be equal.

8.1.5 Methods returning string representations

point2str(*point*)
: Return a string representation of a point. The string holds the three coordinates, separated by a comma. *This is a class method, not an instance method.*

elem2str(*elem*)
: Return a string representation of an element. The string contains the string representations of all the element's nodes, separated by a semicolon. *This is a class method, not an instance method.*

asFormex()
: Return a string representation of a Formex.

  Coordinates are separated by commas, points are separated by semicolons and grouped between brackets, elements are separated by commas and grouped between braces. Thus a Formex `F = Formex([[[1,0],[0,1]],[[0,1],[1,2]]])` is formatted as '{[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 a string representation of the Formex as a numpy array.

setPrintFunction(*func*)
: This sets how a Formex will be formatted by the print statement or by a `"%s"` formatting string. *func* can be any of the above functions *asFormex*, *asFormexWithProp* or *asArray*, or a user-defined function.

  *This is a class method, not an instance method.*Use it as follows: `Formex.setPrintFunction(Formex.asArray)`.

fprint(*fmt="%10.3e %10.3e %10.3e"*)
: Prints all the points of the formex with the specified format. The format should hold three formatting codes, for the three coordinates of the point.

8.1.6 Methods changing the instance data

These are the only methods that change the data of the Formex object. All other transformation methodes return and operate on copies, leaving the original object unchanged.

setProp(*p*)
: Create a 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.

  Specifying a value `None` results in a Formex without properties.

append(*F*)
: Appends all the elements of a Formex F to the current one. Use the *__add__* function or the + operator to concatenate two Formices without changing either of the onjects.

  Only Formices having the same plexitude as the current one can be appended. If one of the Formices has properties and the other not, the elements with missing properties will be assigned property 0.

8.1.7 Methods returning copies

copy()
Return a deep copy of itself.

__add__(*other*)
Return a Formex with all elements of self and other. This allows the user to write simple expressions as F+G to concatenate the Formices F and G. As with the append() method, both Formices should have the same plexitude.

concatenate(*Flist*)
Return the concatenation of all formices in Flist. All formices should have the same plexitude. *This is a class method, not an instance method.*

select(*idx*)
Return a Formex which holds only elements with numbers in *idx*. *idx* can be a single element number or a list of numbers.

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 grade 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.

points()
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.

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.

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: two corners of the bbox.

unique(*self,rtol=1.e-4,atol=1.e-6*)
Return a Formex which holds only the unique elements.

Two elements are considered equal when all its nodal coordinates are close. Two values are close if they are both small compared to atol or their difference divided by the second value is small compared to rtol. Two elements are not considered equal if one's elements are a permutation of the other's.

Warning: this operation is slow when performed on large Formices. Its use is decouraged.

reverseElements()
Return a Formex where all elements have been reversed.

Reversing an element means reversing the order of its points.

8.1.8 Clipping methods

These methods can be use to make a selection of elements based on their nodal coordinates. The heart is the function

`test` (*nodes='all',dir=0,min=None,max=None*)

Flag elements having nodal coordinates between min and max.

This function is very convenient in clipping a Formex in one of the coordinate directions. It returns a 1D integer array flagging (with a value 1 or True) the elements having points with coordinates in the specified range. Use `where(result)` to get a list of element numbers passing the test. Or directly use the `clip()` or `cclip()` methods to create the clipped Formex.

The test plane can be defined in two ways, depending on the value of dir. If *dir* is a single integer (0, 1 or 2), it specifies a global axis and *min* and *max* are the minimum and maximum values for the coordinates along that axis. Default is the 0 (or x) direction. Else, *dir* should be compatible with a (3,) shaped array and specifies the direction of the normal on the planes. In this case, *min* and *max* are points and should also evaluate to (3,) shaped arrays.

xmin,xmax are there minimum and maximum values required for the coordinates in direction dir (default is the x or 0 direction).

nodes specifies which points are taken into account in the comparisons. It should be one of the following:

- a single (integer) node number (< the number of points in an element)
- 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 points 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.

If you want to have a list of the element numbers that satisfy the specified conditions, you can use numpy's where function on the result. Thus `where(F.where(min=1.0))` returns a list with all elements lying right of the plane `x=1.0`.

`clip` (*t*)

Returns 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.

8.1.9 Affine transformations

`scale` (*scale*)

Returns a copy scaled with `scale[i]` in direction `i`.

The *scale* should be a list of 3 numbers, or a single number. In the latter case, the scaling is homothetic.

`translate` (*dir,distance=None*)

Returns a copy translated over *distance* in direction *dir*.

dir is either an axis number (0,1,2) or a direction vector.

If a distance is given, the translation is over the specified distance in the specified direction. If no distance is given, and *dir* is specified as an axis number, translation is over a distance 1. If no distance is given, and *dir* is specified as a vector, translation is over the specified vector.

Thus, the following are all equivalent: `F.translate(1)`; `F.translate(1,1)`; `F.translate([0,1,0])`;`F.translate([0,2,0],1)`

`rotate`(*angle,axis=2*)

Return a copy rotated over *angle* around *axis*.

The angle is specified in degrees. The axis is either one of 0,1,2 designating one of the global axes, or a 3-component 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.

Positive angles rotate clockwise when looking in the positive direction of the axis.

As a convenience, the user may also specify a 3x3 rotation matrix as argument. In that case rotate(mat) is equivalent to affine(mat).

`shear`(*dir,dir1,skew*)

Returns a copy skewed in the direction *dir* of plane *(dir,dir1)*.

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

`reflect`(*dir,pos=0*)

Returns a Formex mirrored in direction *dir* against plane at *pos*.

Default position of the plane is through the origin.

`affine`(*mat,vec=None*)

Returns a general affine transform of the Formex.

The returned Formex has coordinates given by `mat * xorig + vec`, where *mat* is a 3x3 matrix and *vec* a length 3 list.

8.1.10 Non-affine transformations

`cylindrical`(*dir=[0,1,2],scale=[1.,1.,1.]*)

Converts from cylindrical to cartesian after scaling.

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]*)

Converts from cartesian to cylindrical coordinates.

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.,1.,1.],colat=False*)

Converts from spherical to cartesian after scaling.

dir specifies which coordinates are interpreted as longitude(theta), latitude(phi) and distance(r).
scale will scale the coordinate values prior to the transformation.
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. That choice may facilitate the creation of spherical domes.

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

Converts from cartesian to spherical coordinates.

dir specifies which coordinates axes are parallel to respectively the spherical axes distance(r), longitude(theta) and colatitude(phi). Colatitude is 90 degrees - latitude, i.e. the elevation angle measured from north pole(0) to south pole(180). 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*)
Return a Formex with a one-dimensional bump.

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. `func(0)` should be different from 0.

bump2 (*dir,a,func*)
Return a Formex with a two-dimensional bump.

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*)
Return a Formex 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.

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()`.

map (*func*)
Return a Formex mapped by a 3-D function.

This is one of the versatile mapping functions.
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: `E.map(lambda x,y,z:  [2*x,3*y,4*z])` is equivalent with `E.scale([2,3,4])`.

map1 (*dir,func*)
Return a Formex where coordinate *i* is mapped by a 1-D function.

func is a numerical function which takes one argument and produces one result. The coordinate *dir* will be replaced by func(coord[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,dist=None*)
Maps one coordinate by a function of the distance to a point.

func is a numerical function which takes one argument and produces one result. The coordinate *dir* will be replaced by `func(d)`, where `d` is calculated as the distance to *point*. The function must be applicable on arrays, so it should only include numerical operations and functions understood by the numpy module. By default, the distance `d` is calculated in 3-D, but one can specify a limited set of axes to calculate a 2-D or 1-D distance.

This method is one of several mapping methods. See also `map()` and `map1()`.
Example: `E.mapd(2,lambda d:sqrt(10**2-d**2),f.center(),[0,1])` maps E on a sphere with radius 10.

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

i and *j* are lists of 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 Formex 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, which swaps array axesof the ndarray object!

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

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

`projectOnSphere` (*radius,center=[0.,0.,0.]*)
Project the points of the Formex on a sphere with given center and radius.

`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.

`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.

8.1.11 Topology changing transformations

`replic` (*n,step,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,t2,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.]*)
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*)
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.

8.1.12 Transformations that are limited to specific plexitudes.

The following methods are only applicable for specific values of the plexitude.

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

Replaces each member of the Formex *F* 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].

intersectionWithPlane (*p,n*)
Return the intersection of a plex-2 Formex with the plane (p,n).

This is equivalent with the function intersectionWithPlane(F,p,n).

intersectionPointsWithPlane (*p,n*)
Return the intersection points of a plex-2 Formex with plane (p,n).

This is equivalent with the function intersectionWithPlane(F,p,n), but returns a Formex instead of an array of points.

intersectionLinesWithPlane (*p,n*)
Returns the intersection lines of a plex-3 Formex with plane (p,n).

This is equivalent with the function intersectionLinesWithPlane(F,p,n).

cutAtPlane (*p,n,newprops=None*)
Return all elements of a plex-2 or plex-3 Formex cut at plane.

This is equivalent with the function cutAtPlane(F,p,n) or cut3AtPlane(F,p,n,newprops).

split ()
Split a Formex in its elements.

Returns a list of Formices each comprising one element.

8.1.13 Write to file, read from file

write (*fil,sep=' '*)
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. It is advised, though not required, to use filenames ending in '.formex' for this purpose.

If *fil* is a string, the file is closed prior to returning. If an open file is specified, multiple Formices can be written to it before closing the file.

If *sep* is specified, it will be used as a separator between subsequent coordinates. If an empty string is specified, the formex will be stored in a binary format. The default is to use an ASCII format with a single space as separator.

`read`(*fil*)

Read a Formex from a file in native format.

This class method can be used to read back the data stored with the `write(fil,sep)` method. *fil* is either a filename, or an open file.

This method will always return a single Formex, event if the file contains more than one. Use it repeatibly with an open file as argument to read more Formices from the same file.

There is no need to specify the separator that was used in the write operation: it will be detected from the file header.

Also, note the existence of a *readfile* function that can be used to read Formex data from a file that is not in native format.

This is a class method, not an instance method.

8.1.14 Non-member functions

The following functions operate on or return Formex objects, but are not part of the Formex class.[2]

`connect`(*Flist,nodid=None,bias=None,loop=False*)

Return a Formex which connects the formices in *Flist*.

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 *Flist*. Each element of the Formex consist of a node from the corresponding element of each of the Formices in *Flist*. By default this will be the first node of that element, but a *nodid* list may be given to specify the node ids 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* is False, the length of the Formex will be the minimum length of the formices in *Flist*, each minus its respective bias. By setting *loop* True however, each Formex will loop around when its end is encountered, and the length of the resulting Formex is the maximum length in *Flist*.

`interpolate`(*F,G,div,swap=False*)

Create interpolations between two formices.

F and *G* are two Formices with the same shape. *v* 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 division for the interval [0..1]. Thus, `interpolate(F,G,n)` is equivalent with `interpolate(F,G,arange(0,n+1)/float(n))`

[2]They might be implemented as class methods in future releases.

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.

readfile (*file,sep=',',plexitude=1,dimension=3*)
Read a Formex from file.

This convenience function uses the numpy fromfile function to read the coordinates of a Formex from file.

file is either an open file object or a string with the name of the file to be read. *sep* is the separator string between subsequent coordinates. There can be extra blanks around the separator, and the separator can be omitted at the end of line. If an empty string is specified, the file is read in binary mode.

The file is read as a single stream of coordinates; the arguments *plexitude* and *dimension* determine how these are structured into a Formex. *plexitude* is the number of points that make up an element. The default is to return a plex-1 Formex (unconnected points). *dimension* is the number of coordinates that make up a point (2 or 3). As always, the resulting Formex will be 3D. The total number of coordinates on the file should be a multiple of `plexitude * dimension`.

vectorPairAreaNormals (*vec1,vec2*)
Compute area of and normals on parallellograms formed by two vectors.

vec1 and *vec2* are (n,3)-shaped arrays holding collections of vectors. The result is a tuple of two arrays:

- *area*, with shape (n): the area of the parallellogram formed by corresponding vectors of *vec1* and *vec2*.
- *normal*, with shape (n,3): unit-length normal vectors to each pair of vectors. The positive direction of the normals is thus that a rotation of *vec1* to *vec2* corresponds to a positive rotation around the normal.

Both values are calculated from the cross prduct of *vec1* and *vec2*, which indeed results in *area * normal*.

vectorPairArea (*vec1,vec2*)
Compute the area of the parallellogram formed by two vectors.

This returns the first part of `vectorPairAreaNormals(vec1,vec2)`.

vectorPairNormals (*vec1,vec2,normalized=True*)
Compute the normal vectors to pairs of two vectors.

With `normalized=True`, this returns the second part of `vectorPairAreaNormals(vec1,vec2)`.

With `normalized=False`, returns unnormalized normal vectors. This does not use the `vectorPairAreaNormals` function and is provided only to save computing time with very large arrays when normalization is not required. It is equivalent to `cross(vec1,vec2)`.

def polygonNormals (*x*)
Compute normals in all points of polygons in *x*.

x is an `(nel,nplex,3)` coordinate array representing a (possibly not plane) polygon.

The return value is an `(nel,nplex,3)` array with the unit normals on the two edges ending in each point.

pattern (*s*)
Return a line segment pattern created from a string.

This function creates a list of line segments where all nodes lie on the gridpoints of a regular grid with unit step. The first point of the list is [0,0,0]. Each character from the given string is interpreted

as a code specifying how to move to the next node.
Currently defined are the following codes:
0 = goto origin [0,0,0]
1..8 move in the x,y plane
9 remains at the same place
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:

```
         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 making a connection. The effect of any other character is undefined. The resulting list is directly suited to initialize a Formex.

translationVector (*dir,dist*)
Return a translation vector in direction dir over distance dist

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

The angle is specified in degrees. If axis==None (default), a 2x2 rotation matrix is returned. Else, axis should be one of [0,1,2] and specifies the rotation axis in a 3D world. A 3x3 rotation matrix is returned.

rotationAboutMatrix (*angle,axis*)
Return a rotation matrix over angle around an axis thru the origin.

The angle is specified in degrees. Axis is a list of three components specifying the axis. The result is a 3x3 rotation matrix in list format. Note that: rotationAboutMatrix(angle,[1,0,0]) == rotationMatrix(angle,0) rotationAboutMatrix(angle,[0,1,0]) == rotationMatrix(angle,1) rotationAboutMatrix(angle,[0,0,1]) == rotationMatrix(angle,2) but the latter functions are more efficient.

equivalence (*x,nodesperbox=1,shift=0.5,rtol=1.e-5,atol=1.e-5*)
Finds (almost) identical nodes and returns a compressed list.

The input x is an (nnod,3) array of nodal coordinates. This functions finds the nodes that are very close and replaces them with a single node. The return value is a tuple of two arrays: the remaining (nunique,3) nodal coordinates, and an integer (nnod) array holding an index in the unique coordinates array for each of the original nodes.

The procedure works by first dividing the 3D space in a number of equally sized boxes, with a mean population of nodesperbox. 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.

Currently the procedure 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. Nevertheless we intend to access this problem by repeating the procedure with the boxes shifted in space.

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

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

The {coords} module implements a data class for storing large sets of 3D coordinates and provides a extensive set of methods for transforming these coordinates. The {Coords} class is used by other classes, such as {Formex} and {Surface}, which thus inherit the same transformation capabilities. In future, other geometrical data models may (and should) also derive from the {Coords} 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.

8.2.1 Coords class: A structured collection of 3D coordinates.

The Coords class is the basic data structure used throughout pyFormex to store coordinates of points in a 3D space.

The {Coords} class 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 floating point numpy (Numerical Python) array whose last axis has a length equal to 3. Each set of 3 values along the last axis thus represents a single point in 3D cartesian space.

The datatype should be a float type; the default is Float, which is equivalent to numpy's float32. These restrictions are currently only checked at creation time. It is the responsibility of the user to keep consistency.

The Coords class has this constructor:

class `Coords` (*cls,data=None,dtyp=None,copy=False*)

Create a new instance of class Coords.

If no data are given, a single point (0.,0.,0.) will be created. If specified, data should evaluate to an (...,3) shaped array of floats. If copy==True, the data are copied. If no dtype is given that of data are used, or float32 by default.

Coords objects have the following methods:

`points`()

Return the data as a simple set of points.

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

`pshape`()

Return the shape of the points array.

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

`npoints`()

Return the total number of points.

`x`()

Return the x-plane

`y`()

Return the y-plane

z()
: Return the z-plane

bbox()
: Return the bounding box of a set of points.

  The bounding box is the smallest rectangular volume in global coordinates, such at no points are outside the box. It is returned as a Coords object with shape (2,3): the first row holds the minimal coordinates and the second row the maximal.

center()
: Return the center of the Coords.

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

centroid()
: Return 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.

sizes()
: Return the sizes of the Coords.

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

dsize()
: Return an estimate of the global size of the Coords.

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

bsphere()
: Return 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.

distanceFromPlane(*p,n*)
: Return the distance of points f from 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 [...] shaped array 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.

distanceFromLine(*p,n*)
: Return the distance of points f from the line (p,n).

  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.

distanceFromPoint(*p*)
: Return the distance of points f 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.

test(*dir=0,min=None,max=None,atol=0.*)
: Flag points having coordinates between min and max.

  This function is very convenient in clipping a Coords 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 Coords.

The test plane can be define in two ways depending on the value of dir. If dir == 0, 1 or 2, it specifies a global axis and min and max are the minimum and maximum values for the coordinates along that axis. Default is the 0 (or x) direction.

Else, dir should be compatible with a (3,) shaped array and specifies the direction of the normal on the planes. In this case, min and max are points and should also evaluate to (3,) shaped arrays.

One of the two clipping planes may be left unspecified.

fprint (*fmt="%10.3e %10.3e %10.3e"*)
Formatted printing of a Coords.

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,inplace=False*)
Return 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.

translate (*vector,distance=None,inplace=False*)
Translate a Coords object.

The translation vector can be specified in one of the following ways: - an axis number (0,1,2), - a single translation vector, - an array of translation vectors. If an axis number is given, a unit vector in the direction of the specified axis will be used. If an array of translation vectors is given, it should be broadcastable to the size of the Coords array. If a distance value is given, the translation vector is multiplied with this value before it is added to the coordinates.

Thus, the following are all equivalent: F.translate(1) F.translate(1,1) F.translate([0,1,0]) F.translate([0,2,0],0.5)

rotate (*angle,axis=2,around=None,inplace=False*)
Return 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*)
Return a copy skewed in the direction dir of plane (dir,dir1).

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

reflect (*dir=2,pos=0,inplace=False*)
Mirror the coordinates in direction dir against plane at pos.

Default position of the plane is through the origin. Default mirror direction is the z-direction.

affine (*mat,vec=None,inplace=False*)
Return a general affine transform of the Coords.

The returned Coords has coordinates given by xorig * mat + vec, where mat is a 3x3 matrix and vec a length 3 list.

cylindrical (*dir=[[0„,1„,2,]],scale=[[1.„,1.„,1.,]]*)

Converts from cylindrical to cartesian after scaling.

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,]]*)

Converts from cartesian to cylindrical coordinates.

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.„,1.„,1.,]],colat=False*)

Converts from spherical to cartesian after scaling.

<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 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.

superSpherical (*n=1.0,e=1.0,k=0.0,dir=[[0„,1„,2,]],scale=[[1.„,1.„,1.,]],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. <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.

The n and 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,]]*)

Converts from cartesian to spherical coordinates.

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*)

Return a Coords with a one-dimensional bump.

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 !! func(0) should be different from 0.

bump2 (*dir,a,func*)

Return a Coords with a two-dimensional bump.

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*)

Return 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.

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.

newmap (*func*)

Return a Coords mapped by a 3-D function.

This is one of the versatile mapping functions. 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: E.map(lambda x,y,z: [2*x,3*y,4*z]) is equivalent with E.scale([2,3,4])

map (*func*)

Return a Coords mapped by a 3-D function.

This is one of the versatile mapping functions. 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: E.map(lambda x,y,z: [2*x,3*y,4*z]) is equivalent with E.scale([2,3,4])

map1 (*dir,func,x=None*)

Return a Coords where coordinate i is mapped by a 1-D function.

<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,dist=None*)

Maps one coordinate by a function of the distance to a point.

<func> is a numerical function which takes one argument and produces one result. The coordinate dir will be replaced by func(d), where <d> is calculated as the distance to <point>. The function must be applicable on arrays, so it should only include numerical operations and functions understood by the numpy module. By default, the distance d is calculated in 3-D, but one can specify a limited set of axes to calculate a 2-D or 1-D distance. This method is one of several mapping methods. See also map3 and map1. Example: E.mapd(2,lambda d:sqrt(10**2-d**2),f.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.

projectOnSphere (*radius=1.,center=[[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.,dir=0,center=[[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..

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.

fuse (*nodesperbox=1,shift=0.5,rtol=1.e-5,atol=1.e-5*)
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. The return value is a tuple of two arrays: - the unique points as a Coords object, - an integer (nnod) array holding an index in the unique coordinates array for each of the original nodes. This index will have the same shape as the pshape() of the coords array.

The procedure works by first dividing the 3D space in a number of equally sized boxes, with a mean population of nodesperbox. 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.

Currently the procedure 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. Nevertheless we intend to access this problem by repeating the procedure with the boxes shifted in space.

concatenate (*cls,L*)
Concatenate a list of Coords object.

All Coords object in the list L should have the same shape except for the length of the first axis. This function is equivalent to the numpy concatenate, but makes sure the result is a Cooords object.

This is a class method, not an instance method.

fromfile ()
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.

This is a class method, not an instance method.

interpolate (*clas,F,G,div*)
Create interpolations between two Coords.

F and G are two Coords with the same shape. v 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 Coords H where each coordinate Hijk is obtained from: Hijk = Fijk + v * (Gijk-Fijk). Thus, a Coords interpolate(F,G,[0.,0.5,1.0]) will contain all points of F and G and all points 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 resulting Coords array has an extra axis (the first). Its shape is (n,) + F.shape, where n is the number of divisions.

This is a class method, not an instance method.

8.2.2 Functions defined in the coords module

bbox (*objects*)
Compute the bounding box of a list of objects.

All the objects in list should have This is like the bbox() method of the Coords class, but the resulting box encloses all the Coords in the list. Objects returning a None bbox are ignored.

coordsmethod (*f*)
Decorator to apply a Coords method to a 'coords' attribute.

Many classes that model geometry use a 'coords' attribute to store the coordinates. This decorator can be used to apply the Coords method to that attribute, thus making the Coords transformations available to other classes.

The following lines show how to use the decorator. These lines make the 'scale' method of the Coords class available in your class, with the same arguments.

@coordsmethod def scale(self,*args,**kargs): pass

The coordinates are changed inplane, so if you want to save the original ones, you need to copy them before you use the transformation.

8.3 array — 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.

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

niceNumber (*f,approx=floor*)
Return a nice number close to but not smaller than f.

sind (*arg*)
Return the sin of an angle in degrees.

cosd (*arg*)
Return the cos of an angle in degrees.

tand (*arg*)
Return the tan of an angle in degrees.

dotpr (*A,B,axis=???*)
Return the dot product of vectors of A and B in the direction of axis.

The default axis is the last.

length (*A,axis=???*)
Returns the length of the vectors of A in the direction of axis.

The default axis is the last.

normalize (*A,axis=???*)
Normalize the vectors of A in the direction of axis.

The default axis is the last.

projection (*A,B,axis=???*)
Return the (signed) length of the projection of vector of A on B.

The default axis is the last.

norm (*v,n=2*)
Return a norm of the vector v.

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

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

isClose (*values,target,rtol=1.e-5,atol=1.e-8*)
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 |

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

unitVector (*axis*)
Return a unit vector in the direction of a global axis (0,1,2).

Use normalize() to get a unit vector in a general direction.

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

The angle is specified in degrees. If axis==None (default), a 2x2 rotation matrix is returned. Else, axis should specifying 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.

rotMatrix (*v,n=3*)
Create a rotation matrix that rotates axis 0 to the given vector.

Return either a 3x3(default) or 4x4(if n==4) rotation matrix.

growAxis (*a,size,axis=???,fill=0*)
Grow a single array axis to the given size and fill with given value.

reverseAxis (*a,axis=???*)
Reverse the elements along axis.

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

The input a is anything that can e converted into a numpy array. Either shape and or kind can be specified. The dimensions where shape contains a -1 value are not checked. The number of dimensions should match, though. If kind does not match, but is included in allow, conversion to the requested type is attempted. Returns the array if valid. Else, an error is raised.

checkArray1D (*a,size=None,kind=None,allow=None*)
Check that an array a has the correct size and type.

Either size and or kind can be specified. If kind does not match, but is included in allow, conversion to the requested type is attempted. Returns the array if valid. Else, an error is raised.

checkUniqueNumbers (*nrs,nmin=0,nmax=None,error=None*)
Check that an array contains a set of uniqe integers in range.

nrs is an integer array with any shape. All integers should be unique and in the range(nmin,nmax). Beware: this means that nmin <= i < nmax ! Default nmax is unlimited. Set nmin to None to error is the value to return if the tests are not passed. By default, a ValueError is raised. On success, None is returned

8.4 `script` — Basic pyFormex script functions

The {pyformex.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.

8.4.1 Exit class: Exception raised to exit from a running script.

Exit objects have the following methods:

8.4.2 ExitAll class: Exception raised to exit pyFormex from a script.

ExitAll objects have the following methods:

8.4.3 ExitSeq class: Exception raised to exit from a sequence of scripts.

ExitSeq objects have the following methods:

8.4.4 TimeOut class: Exception raised to timeout from a dialog widget.

TimeOut objects have the following methods:

8.4.5 Functions defined in the script module

Globals ()
Return the globals that are passed to the scripts on execution.

This basically contains the globals defined in draw.py, colors.py, and formex.py, as well as the globals from numpy. It also contains the definitions put into the pyformex.PF, by preference using the export() function. During execution of the script, the global variable __name__ will be set to either 'draw' or 'script' depending on whether the script was executed in the 'draw' module (–gui option) or the 'script' module (–nogui option).

export (*dic*)
Export the variables in the given dictionary.

export2 (*names,values*)
Export a list of names and values.

forget (*names*)
Remove the global variables specified in list.

rename (*oldnames,newnames*)
Rename the global variables in oldnames to newnames.

listAll (*clas=None,dic=None*)
Return a list of all objects in dic that are of given clas.

If no class is given, Formex objects are sought. If no dict is given, the objects from both GD.PF and locals() are returned.

named (*name*)
Returns the global object named name.

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'.

ack (*question*)
Show a Yes/No question and return True/False depending on answer.

error (*message*)
Show an error message and wait for user acknowlegement.

warning (*message*)

showInfo (*message*)

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.

playScript (*scr,name=None,filename=None,argv=[]*)
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 GD.scriptName to it when the script is started. If a filename is specified, set the global variable __file__ to it.

If step==True, an indefinite pause will be started after each line of the script that starts with 'draw'. Also (in this case), each line (including comments) is echoed to the message board.

force_finish ()

step_script (*s,glob,paus=True*)

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. In nongui mode the stopatbreakpt function can be called from another thread.

enableBreak (*mode=True*)

stopatbreakpt()
: Set the exitrequested flag.

playFile(*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.

play(*fn=None,argv=[],step=False*)
: Play a formex script from file fn or from the current file.

  This function does nothing if no file is passed or no current file was set.

exit(*all=False*)
: Exit from the current script or from pyformex if no script running.

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.

formatInfo(*F*)
: Return formatted information about a Formex.

printall()
: Print all Formices in globals()

printglobals()

printglobalnames()

printconfig()

chdir(*fn*)
: Change the current working directory.

  If fn is a directory name, the current directory is set to fn. If fn is a file name, the current directory is set to the directory holding fn. In either case, the current dirctory is stored in GD.cfg['workdir'] for persistence between pyFormex invocations.

  If fn does not exist, nothing is done.

workHere()
: Change the current working directory to the script's location.

  This function is deprecated: use chdir(_file__) instead.

runtime()
: Return the time elapsed since start of execution of the script.

startGui(*args=[]*)
: Start the gui

8.5 draw — Create 3D graphical representations.

The draw module provides the basic user interface to the OpenGL rendering capabilities of pyFOrmex.

*{*Warning: the {draw} module is still subject to regular changes. Therefore, the information given below may not be fully accurate. }

closeGui()

ask (*question,choices=None,default=None*)
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.

ack (*question*)
Show a Yes/No question and return True/False depending on answer.

error (*message,actions=[(303, (304, (305, (306, (307, (309, (310, (311, (312, (313, (314, (315, (316, (317, (3, "'OK'"))))))))))))))]*)
Show an error message and wait for user acknowledgement.

warning (*message,actions=[(303, (304, (305, (306, (307, (309, (310, (311, (312, (313, (314, (315, (316, (317, (3, "'OK'"))))))))))))))]*)
Show a warning message and wait for user acknowledgement.

showInfo (*message,actions=[(303, (304, (305, (306, (307, (309, (310, (311, (312, (313, (314, (315, (316, (317, (3, "'OK'"))))))))))))))]*)
Show a neutral message and wait for user acknowledgement.

showText (*text,type=None,actions=[(303, (304, (305, (306, (307, (309, (310, (311, (312, (313, (314, (315, (316, (317, (3, "'OK'"))))))))))))))]*)
Display a text and wait for user response.

This can display a large text and will add scrollbars when needed.

showFile (*filename*)

askItems (*items,caption=None*)
Ask the value of some items to the user.

Create an interactive widget to let the user set the value of some items. Input is a list of input items (basically [key,value] pairs). See the widgets.InputDialog class for complete description of the available input items.

The remaining arguments are keyword arguments that are passed to the InputDialog.getResult method. A timeout (in seconds) can be specified to have the input dialog interrupted automatically.

Return 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.

dialogTimedOut ()
Returns True if the last askItems() dialog timed out.

dialogAccepted ()
Returns True if the last askItems() dialog was accepted.

askFilename (*cur=None,filter="All files (*.*)",exist=False,multi=False*)
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.

askDirname (*cur=None*)
Ask for an existing directory name.

cur is a directory. All the directories in that directory will initially be shown.

log (*s*)
Display a message in the cmdlog window.

draw (*F,view=None,bbox=None,color='prop',colormap=None,alpha=0.5,mode=None,linewidth=None,shrink=None,marksize=N*
Draw object(s) with specified settings and direct camera to it.

The first argument is an object to be drawn. All other arguments are settings that influence how the object is being drawn.

F is either a Formex or a TriSurface object, or a name of such object (global or exported), or a list thereof. If F is a list, the draw() function is called repeatedly with each of ithe items of the list as first argument and with the remaining arguments unchanged.

The remaining arguments are drawing options. If None, they are filled in from the current viewport drawing options.

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. 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 wil 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.

color,colormap,linewidth,alpha,marksize are passed to the creation of the 3D actor.

shrink is a floating point shrink factor that will be applied to object before drawing it.

If the Formex has properties and a color list is specified, then the the properties will be used as an index in the color list and each member will be drawn with the resulting color. If color is one color value, the whole Formex will be drawn with that color. Finally, if color=None is specified, the whole Formex is drawn in black.

Each 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 images and is far more elegant that an explicit sleep() operation, because all script processing will continue up to the next drawing instruction. The value of drawdelay is set in the config, or 2 seconds by default. The user can disable the wait cycle for the next draw operation by specifying wait=False. Setting drawdelay=0 will disable the waiting mechanism for all subsequent draw statements (until set >0 again).

setDrawOptions (*d*)

showDrawOptions ()

reset ()

resetAll ()

shrink (*v*)

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).

_shrink (*F,factor*)

Return a shrinked object.

A shrinked object is one where each element is shrinked with a factor around its own center.

drawVectors (*P,v,d=1.0,color='red'*)

drawPlane (*P,N,size*)

drawMarks (*X,M,color=???*)

Draw a list of marks at pioints X.

X is an 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.

`drawNumbers` (*F,color=???,trl=None,offset=0*)
Draw numbers on all elements of F.

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.

`drawVertexNumbers` (*F,color=???,trl=None*)
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.

`drawText3D` (*P,text,color=???,font=None*)
Draw a text at a 3D point.

`drawViewportAxes3D` (*pos,color=None*)
Draw two viewport axes at a 3D position.

`drawBbox` (*A*)
Draw the bbox of the actor A.

`drawActor` (*A*)
Draw an actor and update the screen.

`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.

`focus` (*object*)
Move the camera thus that object comes fully into view.

object can be anything having a bbox() method.

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!

`view` (*v,wait=False*)
Show a named view, either a builtin or a user defined.

`setTriade` (*on=None,size=1.0,pos=[[0.0,,,0.0,,,0.0,]]*)
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.

`drawText` (*text,x,y,adjust='left',font='helvetica',size=10,color=None*)
Show a text at position x,y using font.

`annotate` (*annot*)
Draw an annotation.

`unannotate` (*annot*)

`decorate` (*decor*)
Draw a decoration.

`undecorate` (*decor*)

`frontView` ()

backView()

leftView()

rightView()

topView()

bottomView()

isoView()

createView(*name,angles*)
Create a new named view (or redefine an old).

The angles are (longitude, latitude, twist). If the view name is new, and there is a views toolbar, a view button will be added to it.

zoomBbox(*bb*)
Zoom thus that the specified bbox becomes visible.

zoomRectangle()
Zoom a rectangle selected by the user.

zoomAll()
Zoom thus that all actors become visible.

zoom(*f*)

bgcolor(*color*)
Change the background color (and redraw).

fgcolor(*color*)
Set the default foreground color.

renderMode(*mode*)

wireframe()

flat()

smooth()

smoothwire()

flatwire()

opacity(*alpha*)
Set the viewports transparency.

lights(*onoff*)
Set the lights on or off

set_light_value(*typ,val*)
Set the value of one of the lighting parameters for the currrent view

typ is one of 'ambient','specular','emission','shininess' val is a value between 0.0 and 1.0

set_ambient(*i*)

set_specular(*i*)

set_emission(*i*)

set_shininess(*i*)

linewidth(*wid*)
Set the linewidth to be used in line drawings.

clear_canvas()
Clear the canvas.

This is a low level function not intended for the user.

clear()
Clear the canvas

redraw()

pause(*msg="Use the Step or Continue button to proceed"*)
Pause the execution until an external event occurs.

When the pause statement is executed, execution of the pyformex script is suspended until some external event forces it to proceed again. Clicking the STEP or CONTINUE button will produce such an event.

step()
Perform one step of a script.

A step is a set of instructions until the next draw operation. If a script is running, this just releases the draw lock. Else, it starts the script in step mode.

fforward()

delay(*i*)
Set the draw delay in seconds.

sleep(*timeout=None*)
Sleep until key/mouse press in the canvas or until timeout

wakeup(*mode=0*)
Wake up from the sleep function.

This is the only way to exit the sleep() function. Default is to wake up from the current sleep. A mode > 0 forces wakeup for longer period.

printbbox()

printviewportsettings()

layout(*nvps=None,ncols=None,nrows=None*)
Set the viewports layout.

addViewport()
Add a new viewport.

removeViewport()
Remove a new viewport.

linkViewport(*vp,tovp*)
Link viewport vp to viewport tovp.

Both vp and tovp should be numbers of viewports.

viewport(*n*)
Select the current viewport

updateGUI()
Update the GUI.

flyAlong(*path='flypath',upvector=[[0.,,,1.,,,0.,]],sleeptime=None*)
Fly through the scene along the flypath.

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.

set_selection_filter(*i*)
Set the selection filter mode

pick(*mode='actor',single=False,func=None,filtr=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

If no filter is given, the available filters are presented in a combobox.

pickActors(*single=False,func=None,filtr=None*)

pickElements(*single=False,func=None,filtr=None*)

pickPoints(*single=False,func=None,filtr=None*)

pickEdges(*single=False,func=None,filtr=None*)

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.

pickNumbers(*marks=None*)

set_edit_mode(*i*)
Set the drawing edit mode.

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.

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.

setLocalAxes (*mode=True*)

setGlobalAxes (*mode=True*)

8.6 `colors` — Definition of some RGB colors and color conversion functions

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.

colorName (*color*)

Return a string designation for the color.

color can be anything that is accepted by GLColor. In most cases If color can not be converted, None is returned.

createColorDict ()

closestColorName (*color*)

Return the closest color name.

RGBA (*rgb,alpha=1.0*)

Adds an alpha channel to an RGB color

GREY (*val,alpha=1.0*)

Returns a grey OpenGL color of given intensity (0..1)

grey (*i*)

8.7 `connectivity` — connectivity.py

A pyFormex plugin for handling connectivity of nodes and elements.

8.7.1 Connectivity class: A class for handling element/node connectivity.

A connectivity object is a 2-dimensional integer array with all non-negative values. In this implementation, all values should be lower than 2**31.

Furthermore, all values in a row should be unique. This is not enforced at creation time, but a method is provided to check the uniqueness.

The Connectivity class has this constructor:

class Connectivity (*data,dtyp=None,copy=False*)

Create a new Connectivity object.

data should be integer type and evaluate to an 2-dim array. If copy==True, the data are copied. If no dtype is given, that of data are used, or int32 by default.

Connectivity objects have the following methods:

nelems ()

nplex()

Max()

unique()
Return a list of arrays with the unique values for each row.

checkUnique()
Flag the rows which have all unique entries.

Returns an array with the value True or Falsefor each row.

check()
Returns True if all rows have unique entries.

reverseIndex()
Return a reverse index for the connectivity table.

This is equivalent to the function reverseIndex()

expand()
Transform elems to edges and faces.

Return a tuple edges,faces where - edges is an (nedges,2) int array of edges connecting two node numbers. - faces is an (nelems,nplex) int array with the edge numbers connecting each pair os subsequent nodes in the elements of elems.

The order of the edges respects the node order, and starts with nodes 0-1. The node numbering in the edges is always lowest node number first.

The inverse operation can be obtained from function compactElems.

8.7.2 Functions defined in the connectivity module

magic_numbers(*elems,magic*)

demagic(*mag,magic*)

expandElems(*elems*)

compactElems(*edges,faces*)
Return compacted elems from edges and faces.

This is the inverse operation of expandElems. The algorithm only works if all vertex numbers of an element are unique.

reverseUniqueIndex(*index*)
Reverse an index.

index is a one-dimensional integer array with unique non-negative values.

The return value is the reverse index: each value shows the position of its index in the index array. The length of the reverse index is equal to maximum value in index plus one. Values not occurring in index get a value -1 in the reverse index.

Remark that reverseUniqueIndex(index)[index] == arange(1+index.max()). The reverse index thus translates the unique index numbers in a sequential index.

reverseIndex(*index,maxcon=3*)
Reverse an index.

index is a (nr,nc) shaped integer array.

The result is a (mr,mc) shaped integer array, where row i contains all the row numbers of index containing i.

Negative numbers in index are disregarded. mr will be equal to the highest positive value in index, +1. mc will be equal to the highest multiplicity of any number in index. On entry, maxcon is an estimate for this value. The procedure will automatically change it if needed.

Each row of the reverse index for a number that occurs less than mc times in index, will be filled up with -1 values.

mult is the highest possible multiplicity of any number in a single column of index.

`adjacencyList`(*elems*)
Create adjacency lists for 2-node elements.

`adjacencyArray`(*elems,maxcon=3*)
Create adjacency array for 2-node elements.

`connected`(*index,i*)
Return the list of elements connected to element i.

index is a (nr,nc) shaped integer array. An element j of index is said to be connected to element i, iff element j has at least one (non-negative) value in common with element i.

The result is a sorted list of unique element numbers, not containing the element number i itself.

`adjacent`(*index,rev=None*)
Return an index of connected elements.

index is a (nr,nc) shaped integer array. An element j of index is said to be connected to element i, iff element j has at least one (non-negative) value in common with element i.

The result is an integer array with shape (nr,mc), where row i holds a sorted list of the elements that are connected to element i, padded with -1 values to created an equal list length for all elements.

The result of this method provides the same information as repeated calls of connected(index,i), but may be more efficient if nr becomes large.

The reverse index may be specified, if it was already computed.

`closedLoop`(*elems*)
Check if a set of line elements form a closed curve.

elems is a connection table of line elements, such as obtained from the feModel() method on a plex-2 Formex.

The return value is a tuple of: - return code: - 0: the segments form a closed loop - 1: the segments form a single non-closed path - 2: the segments form multiple not connected paths - a new connection table which is equivalent to the input if it forms a closed loop. The new table has the elements in order of the loop.

`connectedLineElems`(*elems*)
Partition a segmented curve into connected segments.

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 ia a list of (nsegi,2) shaped array of integers.

is returned. Each part is a (nelems,2) shaped array of integers in which the element numbers are ordered.

8.8 `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.

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.

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).

point (*x=0.,y=0.,z=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.

line (*p1=[[0.,,,0.,,,0.,]],p2=[[1.,,,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.

rectangle (*nx,ny,b=None,h=None,bias=0.,diag=None*)
Return a Formex representing a rectangle of size(b,h) with (nx,ny) cells.

This is a convenience function to create a rectangle with given size. 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.

circle (*a1=2.,a2=0.,a3=360.*)
A polygonal approximation to 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 dash angle) is the angle enclosed between the start and end points of each line segment.
{a2} (the module angle) is the angle enclosed between the start points of two subsequent line segments. If {a2==0.0} is given, it will be taken equal to a1.
{a3} (the arc angle) is the total angle enclosed between the first point of the first segment and the end point of the last segment.

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 hexagone.

Remark that the default {a2 == a1} produces a continuous line, while a2 > a1 results in a dashed line.

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)}.

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.

quadraticCurve (*x=None,n=8*)

CreateDraw 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.

sphere2 (*nx,ny,r=1,bot=???,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.

sphere3 (*nx,ny,r=1,bot=???,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.

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.

sector (*r,t,nr,nt,h=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 drawing 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.

8.9 `utils` — A collection of miscellaneous utility functions.

8.9.1 NameSequence class: A class for autogenerating sequences of names.

The name includes a numeric part, whose number is incremented at each call of the 'next()' method.

The NameSequence class has this constructor:

class NameSequence (*name,ext=''*)

Create a new NameSequence from name,ext.

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.

Examples: NameSequence('hallo.98') will generate names hallo.98, hallo.99, hallo.100, ... NameSequence('hallo','.png') will generate names hallo-000.png, hallo-001.png, ... NameSequence('/home/user/hallo23','5.png') will generate names /home/user/hallo235.png, /home/user/hallo245.png, ...

NameSequence objects have the following methods:

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.

8.9.2 Functions defined in the utils module

congratulations(*name,version,typ='module',fatal=False*)
Report a detected module/program.

checkVersion(*name,version,external=False*)
Checks a version of a program/module.

name is either a module or an external program whose availability has been registered. Default is to treat name as a module. Add external=True for a program.

Return value is -1, 0 or 1, depending on a version found that is <, == or > than the requested values. This should normally understand version numbers in the format 2.10.1

checkModule(*name*)
Check if the named Python module is available, and record its version.

The version string is returned, empty if the module could not be loaded. The (name,version) pair is also inserted into the the_version dict.

hasModule(*name,check=False*)
Test if we have the named module available.

Returns a nonzero (version) string if the module is available, or an empty string if it is not.

By default, the module is only checked on the first call. The result is remembered in the the_version dict. The optional argument check==True forces a new detection.

checkExternal(*name=None,command=None,answer=None*)
Check if the named external command is available on the system.

name is the generic command name, command is the command as it will be executed to check its operation, answer is a regular expression to match positive answers from the command. answer should contain at least one group. In case of a match, the contents of the match will be stored in the the_external dict with name as the key. If the result does not match the specified answer, an empty value is inserted.

Usually, command will contain an option to display the version, and the answer re contains a group to select the version string from the result.

As a convenience, we provide a list of predeclared external commands, that can be checked by their name alone. If no name is given, all commands in that list are checked, and no value is returned.

hasExternal(*name*)
Test if we have the external command 'name' available.

Returns a nonzero string if the command is available, or an empty string if it is not.

The external command is only checked on the first call. The result is remembered in the the_external dict.

printDetected()

removeTree(*path,top=True*)
Remove all files below path. If top==True, also path is removed.

setSaneLocale()
Set a sane local configuration for LC_NUMERIC.

Some local settings change the LC_NUMERIC setting, so that floating point values are read or written with a comma instead of a the decimal point. Of course this makes your files completely incompatible. You will often not be able to process these files any further and create a lot of troubels for yourself and other people if you do so. The idiots that thought changing the LC_NUMERIC locale was a good thing should be hung.

Anyway, here's a function to set it back to a sane value. It is always called when pyFormex starts.

all_image_extensions()
Return a list with all known image extensions.

fileDescription(*type*)
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)'

findIcon(*name*)
Return the file name for an icon with given name.

If no icon file is found, returns the question mark icon.

projectName(*fn*)
Derive a project name from a file name.

The project name is the basename f the file without the extension.

splitme(*s*)

mergeme(*s1,s2*)

mtime(*fn*)
Return the (UNIX) time of last change of file fn.

countLines(*fn*)
Return the number of lines in a text file.

runCommand(*cmd,RaiseError=True,quiet=False*)
Run a command and raise error if exited with error.

spawn(*cmd*)
Spawn a child process.

changeExt(*fn,ext*)
Change the extension of a file name.

The extension is the minimal trailing part of the filename starting with a '.'. If the filename has no '.', the extension will be appended. If the given extension does not start with a dot, one is prepended.

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.

isPyFormex (*filename*)
Checks whether a file is a pyFormex script.

A script is considered to be a pyFormex script if its first line starts with '#!' and contains the substring 'pyformex' A file is considered to be a pyFormex script if its name ends in '.py' and the first line of the file contains the substring 'pyformex'. Typically, a pyFormex script starts with a line: #!/usr/bin/env pyformex

splitEndDigits (*s*)
Split a string in any prefix and a numerical end sequence.

A string like 'abc-0123' will be split in 'abc-' and '0123'. Any of both can be empty.

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.

interrogate (*item*)
Print useful information about item.

deprecated (*replacement*)

functionWasRenamed (*replacement,text=None*)

functionBecameMethod (*replacement*)

8.10 colorscale — Color mapping of a range of values.

8.10.1 ColorScale class: 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.

The ColorScale class has this constructor:

class ColorScale (*palet,minval=0.,maxval=1.,midval=None,exp=1.0,exp2=None*)
Create a colorscale to map a range of values into colors.

The values range from minval to maxval (default 0.0..1.0).

A midval may be specified to set the value corresponding to the midle of the color scale. It defaults to the middle value of the range. It is especially useful if the range extends over negative and positive values to set 0.0 as the middle value.

The palet is a list of 3 colors, corresponding to the minval, midval and maxval respectively. The middle color may be given as None, in which case it will be set to the middle color between the first and last.

The Palette variable provides some useful predefined palets. You will hardly ever need to define your own palets.

The mapping function between numerical and color values is by default linear. Nonlinear mappings can be obtained by specifying an exponent 'exp' different from 1.0. Mapping is done with the 'stuur' function from the 'utils' module. If 2 exponents are given, mapping is done independently e with exp in the range minval..midval and with exp2 in the range midval..maxval.

ColorScale objects have the following methods:

`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 wer specified, scaling is done independently in one of the intervals minval..midval or midval..maxval resulting into resp. the interval -1..0 or 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.

8.10.2 ColorLegend class: A colorlegend is a colorscale divided in a number of subranges.

The ColorLegend class has this constructor:

class `ColorLegend`(*colorscale,n*)

Create a color legend dividing a colorscale in n subranges.

The full value range of the colorscale is divided in n subranges, each half range being divided in n/2 subranges. This sets n+1 limits of the subranges. The n colors of the subranges correspond to the subrange middle value.

ColorLegend objects have the following methods:

`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 attirbute 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.

8.11 `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.

`initialize`()

Initialize the image module.

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.

checkImageFormat (*fmt,verbose=False*)
Checks image format; if verbose, warn if it is not.

Returns the image format, or None if it is not OK.

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.

save_canvas (*canvas,fn,fmt='png',options=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

save_window (*filename,format,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.

save_main_window (*filename,format,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.

save_rect (*x,y,w,h,filename,format*)
Save a rectangular part of the screen to a an image file.

save (*filename=None,window=False,multi=False,hotkey=True,autosave=False,border=False,rootcrop=False,format=None,verb*
Saves an image to file or Starts/stops multisave maode.

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.

`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.

`saveIcon`(*fn,size=32*)
Save the current rendering as an icon.

`autoSaveOn`()
Returns True if autosave multisave mode is currently on.

Use this function instead of directly accessing the autosave variable.

`createMovie`()
Create a movie from a saved sequence of images.

`saveMovie`(*filename,format,windowname=None*)
Create a movie from the pyFormex window.

8.12 `widgets` — A collection of custom widgets used in the pyFormex GUI

8.12.1 Options class:

Options objects have the following methods:

8.12.2 FileSelection class: A file selection dialog widget.

You can specify a default path/filename that will be suggested initially. If a pattern is specified, only matching files will be shown. A pattern can be something like 'Images (*.png *.jpg)' or a list of such strings. Default mode is to accept any filename. You can specify exist=True to accept only existing files. Or set exist=True and multi=True to accept multiple files. If dir==True, a single existing directory is asked.

The FileSelection class has this constructor:

class `FileSelection`(*path,pattern=None,exist=False,multi=False,dir=False*)
The constructor shows the widget.

FileSelection objects have the following methods:

`getFilename`()
Ask for a filename by user interaction.

Return the filename selected by the user. If the user hits CANCEL or ESC, None is returned.

8.12.3 SaveImageDialog class: A file selection dialog with extra fields.

The SaveImageDialog class has this constructor:

class `SaveImageDialog`(*path=None,pattern=None,exist=False,multi=False*)
Create the dialog.

SaveImageDialog objects have the following methods:

`getResult`()

8.12.4 ImageViewerDialog class:

The ImageViewerDialog class has this constructor:

class `ImageViewerDialog`(*path=None*)

ImageViewerDialog objects have the following methods:

`getFilename`()

Ask for a filename by user interaction.

Return the filename selected by the user. If the user hits CANCEL or ESC, None is returned.

8.12.5 AppearenceDialog class: A dialog for setting the GUI appearance.

The AppearenceDialog class has this constructor:

class `AppearenceDialog`()

Create the Appearance dialog.

AppearenceDialog objects have the following methods:

`setFont`()

`getResult`()

8.12.6 DockedSelection class: A widget that is docked in the main window and contains a modeless

The DockedSelection class has this constructor:

class `DockedSelection`(*slist=[],title='Selection Dialog',mode=None,sort=False,func=None*)

DockedSelection objects have the following methods:

`setSelected`(*selected,bool*)

`getResult`()

8.12.7 ModelessSelection class: A modeless dialog for selecting one or more items from a list.

The ModelessSelection class has this constructor:

class `ModelessSelection`(*slist=[],title='Selection Dialog',mode=None,sort=False,func=None*)

Create the SelectionList dialog.

ModelessSelection objects have the following methods:

`setSelected`(*selected,bool*)

Mark the specified items as selected.

`getResult`()

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.

8.12.8 Selection class: A dialog for selecting one or more items from a list.

The Selection class has this constructor:

class `Selection` (*slist=[],title='Selection Dialog',mode=None,sort=False,selected=[]*)

Create the SelectionList dialog.

selected is a list of items that are initially selected.

Selection objects have the following methods:

`setSelected` (*selected*)

Mark the specified items as selected.

`getResult` ()

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.

8.12.9 InputItem class: A single input item, usually with a label in front.

The created widget is a QHBoxLayout which can be embedded in the vertical layout of a dialog.

This is a super class, which just creates the label. The input field(s) should be added by a dedicated subclass.

This class also defines default values for the name() and value() methods.

Subclasses should override: - name(): if they called the superclass __init__() method without a name; - value(): if they did not create a self.input widget who's text() is the return value of the item. - setValue(): always, unless the field is readonly.

Subclases can set validators on the input, like input.setValidator(QtGui.QIntValidator(input)) Subclasses can define a show() method e.g. to select the data in the input field on display of the dialog.

The InputItem class has this constructor:

class `InputItem` (*name=None*)

Creates a new inputitem with a name label in front.

If a name is given, a label is created and added to the layout.

InputItem objects have the following methods:

`name` ()

Return the widget's name.

`value` ()

Return the widget's value.

`setValue` (*val*)

Change the widget's value.

8.12.10 InputInfo class: An unchangeable input item.

The InputInfo class has this constructor:

class `InputInfo` (*name,value*)

Creates a new info field with a label in front.

The info input field is an unchangeable text field.

InputInfo objects have the following methods:

value()
Return the widget's value.

setValue(*val*)
Change the widget's value.

8.12.11 InputString class: A string input item.

The InputString class has this constructor:

class InputString(*name,value*)
Creates a new 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.

InputString objects have the following methods:

show()
Select all text on first display.

value()
Return the widget's value.

setValue(*val*)
Change the widget's value.

8.12.12 InputBool class: A boolean input item.

The InputBool class has this constructor:

class InputBool(*name,value*)
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.

InputBool objects have the following methods:

name()
Return the widget's name.

value()
Return the widget's value.

setValue(*val*)
Change the widget's value.

8.12.13 InputCombo class: A combobox InputItem.

The InputCombo class has this constructor:

class InputCombo(*name,choices,default*)
Creates a new combobox for the selection of a value from a list.

choices is a list/tuple of possible values. default is the initial/default choice. If default is not in the choices list, it is prepended. If default is None, the first item of choices is taken as the default.

The choices are presented to the user as a combobox, which will initially be set to the default value.

InputCombo objects have the following methods:

value()
Return the widget's value.

setValue(*val*)
Change the widget's value.

8.12.14 InputRadio class: A radiobuttons InputItem.

The InputRadio class has this constructor:

class InputRadio(*name,choices,default,direction='h'*)
Creates radiobuttons for the selection of a value from a list.

choices is a list/tuple of possible values. default is the initial/default choice. If default is not in the choices list, it is prepended. If default 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.

InputRadio objects have the following methods:

value()
Return the widget's value.

setValue(*val*)
Change the widget's value.

8.12.15 InputPush class: A pushbuttons InputItem.

The InputPush class has this constructor:

class InputPush(*name,choices,default=None,direction='h'*)
Creates pushbuttons for the selection of a value from a list.

choices is a list/tuple of possible values. default is the initial/default choice. If default is not in the choices list, it is prepended. If default 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.

InputPush objects have the following methods:

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.

8.12.16 InputInteger class: An integer input item.

Options: 'min', 'max': range of the scale (integer)

The InputInteger class has this constructor:

class `InputInteger` (*name,value,None*)
Creates a new integer input field with a label in front.

InputInteger objects have the following methods:

`show`()
Select all text on first display.

`value`()
Return the widget's value.

`setValue`(*val*)
Change the widget's value.

8.12.17 InputFloat class: An float input item.

The InputFloat class has this constructor:

class `InputFloat` (*name,value*)
Creates a new float input field with a label in front.

InputFloat objects have the following methods:

`show`()
Select all text on first display.

`value`()
Return the widget's value.

`setValue`(*val*)
Change the widget's value.

8.12.18 InputSlider class: 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.

The InputSlider class has this constructor:

class `InputSlider` (*name,value,None*)
Creates a new integer input slider.

InputSlider objects have the following methods:

`set_value`(*val*)

8.12.19 InputFSlider class: A float 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.

The InputFSlider class has this constructor:

class `InputFSlider` (*name,value,None*)
Creates a new integer input slider.

InputFSlider objects have the following methods:

set_value(*val*)

8.12.20 InputColor class: A color input item.

The InputColor class has this constructor:

class InputColor(*name,value*)

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.

InputColor objects have the following methods:

setColor()

setValue(*value*)

Change the widget's value.

8.12.21 InputDialog class: A dialog widget to set the value of one or more items.

While general input dialogs can be constructed from all the underlying Qt classes, this widget provides a way to construct fairly complex input dialogs with a minimum of effort.

The InputDialog class has this constructor:

class InputDialog(*items,caption=None,parent=None,flags=None,actions=None,default=None,report_-pos=False*)

Creates a dialog which asks the user for the value of items.

Each item in the 'items' list is a tuple holding at least the name of the item, and optionally some more elements that limit the type of data that can be entered. The general format of an item is: name,value,type,range It should fit one of the following schemes: ('name',str) : type string, any string input allowed ('name',int) : type int, any integer value allowed ('name',int,'min','max') : type int, only min <= value <= max allowed For each item a label with the name and a LineEdit widget are created, with a validator function where appropriate.

Input items are defined by a list with the following structure: [name, value, type, range...] The fields have the following meaning: name: the name of the field, value: the initial or default value of the field, type: the type of values the field can accept, range: the range of values the field can accept, The first two fields are mandatory. In many cases the type can be determined from the value and no other fields are required. Thus: ['name', 'value'] will accept any string (initial string = 'value'), ['name', True] will show a checkbox with the item checked, ['name', 10] will accept any integer, ['name', 1.5] will accept any float.

Range settings for int and float types: ['name', 1, int, 0, 4] will accept an integer from 0 to 4, inclusive; ['name', 1, float, 0.0, 1.0, 2] will accept a float in the range from 0.0 to 1.0 with a maximum of two decimals.

Composed types: ['name', 'option1', 'select', ['option0','option1','option2']] will present a combobox to select between one of the options. The initial and default value is 'option1'.

['name', 'option1', 'radio', ['option0','option1','option2']] will present a group of radiobuttons to select between one of the options. The initial and default value is 'option1'. A variant 'vradio' aligns the options vertically.

['name', 'option1', 'push', ['option0','option1','option2']] will present a group of pushbuttons to select between one of the options. The initial and default value is 'option1'. A variant 'vpush' aligns the options vertically.

['name', 'red', 'color'] will present a color selection widget, with 'red' as the initial choice.

InputDialog objects have the following methods:

`__getitem__`(*name*)
: Return the input item with specified name.

`acceptData`()
: 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 t the dialog. The values will be set in the corresponding input items.

`timeout`()

`getResult`(*timeout=None,timeoutAccept=True*)
: 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.

  This function also sets the exit mode, so that the caller can test how the dialog was ended. self.accepted == TRUE/FALSE self.timedOut == TRUE/FALSE

8.12.22 NewInputDialog class: A dialog widget to set the value of one or more items.

While general input dialogs can be constructed from all the underlying Qt classes, this widget provides a way to construct fairly complex input dialogs with a minimum of effort.

The NewInputDialog class has this constructor:

class `NewInputDialog`(*items,caption=None,parent=None,flags=None,actions=None,default=None,report_-pos=False*)
: Creates a dialog which asks the user for the value of items.

  Each item in the 'items' list is a tuple holding at least the name of the item, and optionally some more elements that limit the type of data that can be entered. The general format of an item is: name,value,type,range It should fit one of the following schemes: ('name',str) : type string, any string input allowed ('name',int) : type int, any integer value allowed ('name',int,'min','max') : type int, only min <= value <= max allowed For each item a label with the name and a LineEdit widget are created, with a validator function where appropriate.

  Input items are defined by a list with the following structure: [name, value, type, range...] The fields have the following meaning: name: the name of the field, value: the initial or default value of the field, type: the type of values the field can accept, range: the range of values the field can accept, The first two fields are mandatory. In many cases the type can be determined from the value and no other fields are required. Thus: ['name', 'value'] will accept any string (initial string = 'value'), ['name', True] will show a checkbox with the item checked, ['name', 10] will accept any integer, ['name', 1.5] will accept any float.

Range settings for int and float types: ['name', 1, int, 0, 4] will accept an integer from 0 to 4, inclusive; ['name', 1, float, 0.0, 1.0, 2] will accept a float in the range from 0.0 to 1.0 with a maximum of two decimals.

Composed types: ['name', 'option1', 'select', ['option0','option1','option2']] will present a combobox to select between one of the options. The initial and default value is 'option1'.

['name', 'option1', 'radio', ['option0','option1','option2']] will present a group of radiobuttons to select between one of the options. The initial and default value is 'option1'. A variant 'vradio' aligns the options vertically.

['name', 'option1', 'push', ['option0','option1','option2']] will present a group of pushbuttons to select between one of the options. The initial and default value is 'option1'. A variant 'vpush' aligns the options vertically.

['name', 'red', 'color'] will present a color selection widget, with 'red' as the initial choice.

NewInputDialog objects have the following methods:

`__getitem__` (*name*)
Return the input item with specified name.

`acceptData` ()
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 t the dialog. The values will be set in the corresponding input items.

`timeout` ()

`getResult` (*timeout=None,timeoutAccept=True*)
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.

This function also sets the exit mode, so that the caller can test how the dialog was ended. self.accepted == TRUE/FALSE self.timedOut == TRUE/FALSE

8.12.23 TableModel class: A table model that represent data as a two-dimensional array of items.

data is 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]. Optional lists of column and row headers can be specified.

The TableModel class has this constructor:

class `TableModel` (*data,chead=None,rhead=None,parent=None*)

TableModel objects have the following methods:

rowCount (*parent=None*)

columnCount (*parent=None*)

data (*index,role*)

headerData (*col,orientation,role*)

insertRows (*row=None,count=None*)

removeRows (*row=None,count=None*)

8.12.24 Table class: A dialog widget to show two-dimensional arrays of items.

The Table class has this constructor:

class Table (*data,chead=None,rhead=None,caption=None,parent=None,actions=[(303, (304, (305, (306, (307, (309, (310, (311, (312, (313, (314, (315, (316, (317, (7, '('), (319, (303, (304, (305, (306, (307, (309, (310, (311, (312, (313, (314, (315, (316, (317, (3, "'OK'"))))))))))))))), (12, ','), (8, ')')))))))))))))))],default='OK'*)

Create the Table dialog.

data is a 2-D array of items, mith nrow rows and ncol columns. chead is an optional list of ncol column headers. rhead is an optional list of nrow row headers.

Table objects have the following methods:

8.12.25 TableDialog class: A dialog widget to show two-dimensional arrays of items.

The TableDialog class has this constructor:

class TableDialog (*items,caption=None,parent=None,tab=False*)

Create the Table dialog.

If tab = False, a dialog with one table is created and items should be a list [table_header,table_-data]. If tab = True, a dialog with multiple pages is created and items should be a list of pages [page_header,table_header,table_data].

TableDialog objects have the following methods:

8.12.26 ButtonBox class:

The ButtonBox class has this constructor:

class ButtonBox (*name,choices,funcs*)

ButtonBox objects have the following methods:

setText (*text,index=0*)

setIcon (*icon,index=0*)

__str__ ()

8.12.27 ComboBox class:

The ComboBox class has this constructor:

class **`ComboBox`** (*name,choices,func*)

ComboBox objects have the following methods:

`setIndex` (*i*)

8.12.28 BaseMenu class: A general menu class.

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.

The BaseMenu class has this constructor:

class **`BaseMenu`** (*title='AMenu',parent=None,before=None,items=None*)

Create a menu.

This is a hierarchical menu that keeps a list of its item names and actions.

BaseMenu objects have the following methods:

`item` (*text*)

Get the menu item with given normalized text.

Text normalization removes all '&' characters and converts to lower case.

`itemAction` (*item*)

Return the action corresponding to item.

item is either one of the menu's item texts, or one of its values. This method guarantees that the return value is either the corresponding Action, or None.

`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` (*str,val,before=None*)

Create and insert an action.

`insertItems` (*items,before=None*)

Insert a list of items in the menu.

Each item is a tuple of two to five elements: Text, Action, [Icon, ShortCut, ToolTip].

Item text is the text that will be displayed in the menu. It will be 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, - None : this will create a separator item with no action.

Icon is the name of one of the icons in the installed icondir. ShortCut is an optional key combination to select the item. Tooltip is a popup help string.

If before is given, it specifies the text OR the action of one of the items in the menu: the new items will be inserted before that one.

8.12.29 Menu class: A popup/pulldown menu.

The Menu class has this constructor:

class **`Menu`**(*title='UserMenu',parent=None,before=None,items=None*)

Create a popup/pulldown menu.

If parent==None, the menu is a standalone popup menu. If parent is given, the menu will be inserted in the parent menu. If parent==GD.GUI, the menu is inserted in the main menu bar.

If insert == True, the menu will be inserted in the main menubar before the item specified by before. If before is None or not the normalized text of an item of the main menu, the new menu will be inserted at the end. Calling the close() function of an inserted menu will remove it from the main menu.

If insert == False, the created menu will be an independent dialog and the user will have to process it explicitely.

Menu objects have the following methods:

`process`()

`remove`()

Remove this menu from its parent.

8.12.30 MenuBar class: A menu bar allowing easy menu creation.

The MenuBar class has this constructor:

class **`MenuBar`**()

Create the menubar.

MenuBar objects have the following methods:

8.12.31 DAction class: 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.

The DAction class has this constructor:

class **`DAction`**(*name,icon=None,data=None,signal=None*)

Create a new DAction with name, icon and string data.

If the DAction is used in a menu, a name is sufficient. For use in a toolbar, you will probably want to specify an icon. When the action is triggered, the data is sent as a parameter to the SLOT function connected with the 'Clicked' signal. If no data is specified, the name is used as data.

See the views.py module for an example.

DAction objects have the following methods:

`activated`()

8.12.32 ActionList class: 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.

The ActionList class has this constructor:

class `ActionList` (*actions=[],function=None,menu=None,toolbar=None,icons=None*)

Create an new action list, empty by default.

A list of strings can be passed to initialize the actions. If a menu and/or toolbar are passed, a button is added to them for each string in the action list. If a function is passed, it will be called with the string as parameter when the item is triggered.

If no icon names are specified, they are taken equal to the action names. Icons will be taken from the installed icon directory. If you want to specify other icons, use the add() method.

ActionList objects have the following methods:

`add` (*name,icon=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.

`names` ()

Return an ordered list of names of the action items.

8.12.33 Functions defined in the widgets module

`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.

`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.

`dialogButtons` (*dialog,actions,default*)

Create a set of dialog buttons

dia is a dialog widget 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 does not matter), the button will be bound to the dialog’s ’accept’ or ’reject’ slot. default is the name of the action to set as the default.

`messageBox` (*message,level=’info’,choices=[(303, (304, (305, (306, (307, (309, (310, (311, (312, (313, (314, (315, (316, (317, (3, "’OK’")))))))))))))))],timeout=None*)

Display a message box and wait for user response.

The message box displays a text, an icon depending on the level (either ’about’, ’info’, ’warning’ or ’error’) and 1-3 buttons with the specified action text. The ’about’ level has no buttons.

The function returns the text of the button that was clicked or an empty string is ESC was hit.

`textBox` (*text,type=None,choices=[(303, (304, (305, (306, (307, (309, (310, (311, (312, (313, (314, (315, (316, (317, (3, "’OK’")))))))))))))))]*)

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.

`normalize` (*s*)

Normalize a string.

Text normalization removes all ’&’ characters and converts to lower case.

8.13 `curve` — Definition of curves in pyFormex.

(C) 2008 Benedict Verhegghe (bverheg at users.berlios.de) 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. Acknowledgements: Gianluca De Santis

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.

8.13.1 Curve class: 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: sub_points(t,j)

Curve objects have the following methods:

`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.

`lengths`()

`pointsAt`(*t*)
Returns 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.

`subPoints`(*div=10,extend=[[0.,,,0.,]]*)
Return a series of points on the PolyLine.

The parameter space of each segment is divided into ndiv parts. The coordinates of the points at these parameter values are returned as a Coords object. 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.

`length`()
Return the total length of the curve.

This is only available for curves that implement the 'lengths' method.

8.13.2 PolyLine class: A class representing a series of straight line segments.

The PolyLine class has this constructor:

class `PolyLine`(*coords=[],closed=False*)
Initialize a PolyLine from a coordinate array.

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.

PolyLine objects have the following methods:

`asFormex`()
: Return the polyline as a Formex.

`sub_points`(*t,j*)
: Return the points at values t in part j

`sub_points2`(*t,j*)
: Return the points at value,part pairs (t,j)

`vectors`()
: Return the vectors of the points to the next one.

  The vectors are returned as a Coords object. If not closed, this returns one less vectors than the number of points.

`directions`()
: Returns unit vectors in the direction of the next point.

`avgDirections`(*normalized=True*)
: Returns average directions at the inner nodes.

  If open, the number of directions returned is 2 less than the number of points.

`lengths`()
: Return the length of the parts of the curve.

`atLength`(*div*)
: Returns the parameter values for relative curve lengths div.

  'div' is a list of relative curve lengths (from 0.0 to 1.0). As a convenience, an 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`()

8.13.3 Polygon class: A Polygon is a closed PolyLine.

The Polygon class has this constructor:

class `Polygon`(*coords=[]*)

Polygon objects have the following methods:

8.13.4 BezierSpline class: A class representing a Bezier spline curve.

The BezierSpline class has this constructor:

class `BezierSpline`(*pts,deriv=None,curl=0.5,control=None,closed=False*)
: Create a cubic spline curve through the given points.

  The curve is defined by the points and the directions at these points. If no directions are specified, the average of the segments ending in that point is used, and in the end points of an open curve, the direction of the end segment. The curl parameter can be set to influence the curliness of the curve. curl=0.0 results in straight segment.

BezierSpline objects have the following methods:

`sub_points`(*t,j*)

8.13.5 CardinalSpline class: A class representing a cardinal spline.

The CardinalSpline class has this constructor:

class **`CardinalSpline`** (*pts,tension=0.0,closed=False*)

Create a natural spline through the given points.

The Cardinal Spline with given tension is a Bezier Spline with curl: 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).

CardinalSpline objects have the following methods:

8.13.6 CardinalSpline2 class: A class representing a cardinal spline.

The CardinalSpline2 class has this constructor:

class **`CardinalSpline2`** (*pts,tension=0.0,closed=False*)

Create a natural spline through the given points.

This is a direct implementation of the Cardinal Spline. For open curves, it misses the interpolation in the two end intervals of the point set. It is retained here because the implementation may some day replace the BezierSpline implementation.

CardinalSpline2 objects have the following methods:

`compute_coefficients`()

`sub_points`(*t,j*)

8.13.7 NaturalSpline class: A class representing a natural spline.

The NaturalSpline class has this constructor:

class **`NaturalSpline`** (*pts,endcond=[['notaknot',,'notaknot',]],closed=False*)

Create a natural spline through the given points.

pts specifies the coordinates of a set of points. A natural spline is constructed through this points. endcond specifies the end conditions in the first, resp. last point. It can be 'notaknot' or 'secder'. With 'notaknot', maximal continuity (up to the third derivative) is obtained between the first two splines. With 'secder', the spline ends with a zero second derivative. If closed is True, the spline is closed, and endcond is ignored.

NaturalSpline objects have the following methods:

`compute_coefficients`()

`sub_points`(*t,j*)

8.14 `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.

8.14.1 Model class: Contains all FE model data.

The Model class has this constructor:

class **`Model`** (*coords,elems*)

Create new model data.

coords is an array with nodal coordinates elems is either a single element connectivity array, or a list of such. In a simple case, coords and elems can be the arrays obtained by coords, elems = F.feModel() This is however limited to a model where all elements have the same number of nodes. Then you can use the list of elems arrays. The 'fe' plugin has a helper function to create this list. E.g., if FL is a list of Formices (possibly with different plexitude), then fe.mergeModels([Fi.feModel() for Fi in FL]) will return the (coords,elems) tuple to create the Model.

The model can have node and element property numbers.

Model objects have the following methods:

`nnodes` ()

`nelems` ()

`mplex` ()

Return the maximum plexitude of the model.

`splitElems` (*set*)

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,set*)

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.

8.14.2 Functions defined in the fe module

`mergeNodes` (*nodes*)

Merge all the nodes of a list of node sets.

Each item in nodes is a Coords array. The return value is a tuple with: - the coordinates of all unique nodes, - a list of indices translating the old node numbers to the new.

`mergeModels` (*femodels*)

Merge all the nodes of a list of FE models.

Each item in femodels is a (coords,elems) tuple. 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.

checkUniqueNumbers(*nrs,nmin=0,nmax=None,error=None*)
Check that an array contains a set of unique integers in range.

nrs is an integer array with any shape. All integers should be unique and in the range(nmin,nmax). Beware: this means that nmin <= i < nmax ! Default nmax is unlimited. Set nmin to None to error is the value to return if the tests are not passed. By default, a ValueError is raised. On success, None is returned

mergedModel()
Returns the fe Model obtained from merging individual models.

The input arguments are (coords,elems) tuples. The return value is a merged fe Model.

8.15 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.

centroids(*X*)
Compute the centroids of the points of a set of elements.

X (nelems,nplex,3)

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(*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

principal(*inertia*)
Returns the principal values and axes of the inertia tensor.

8.16 isopar — Isoparametric transformations

8.16.1 Isopar class: A class representing an isoparametric transformation

The Isopar class has this constructor:

class Isopar(*eltype,coords,oldcoords*)
Create an isoparametric transformation.

type 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

Isopar objects have the following methods:

transform (*X*)
Apply isoparametric transform to a set of coordinates.

Returns a Coords array with same shape as X

transformFormex (*F*)
Apply an isoparametric transform to a Formex.

The result is a topologically equivalent Formex.

8.16.2 Functions defined in the isopar module

build_matrix (*atoms,x,y=0,z=0*)
Build a matrix of functions of coords.

Atoms is a list of text strings representing some function of x(,y)(,z). x is a list of x-coordinats of the nodes, y and z can be set to lists of y,z coordinates of the nodes. Each line of the returned matrix contains the atoms evaluated at a node.

transformFormex (*F,trf*)

isopar (*F,eltype,coords,oldcoords*)

8.17 `lima` — Lindenmayer Systems

8.17.1 Lima class: A class for operations on Lindenmayer Systems.

The Lima class has this constructor:

class Lima (*axiom="",rules={}*)

Lima objects have the following methods:

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.

grow (*ngen=1*)

8.17.2 Functions defined in the lima module

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.

8.18 `mesh` — mesh.py

A plugin providing some useful meshing functions.

`createWedgeElements`(*S1,S2,div=1*)

Create wedge elements between to triangulated surfaces.

6-node wedge elements are created between two input surfaces (S1 and S2). The keyword div determines the number of created wedge element layers. Layers with equal thickness are created when an integer value is used for div. div can also be specified using a list, that defines the interpolation between the two surfaces. Consequently, this can be used to create layers with unequal thickness. For example, div=2 gives the same result as [0.,0.5,1.]

`sweepGrid`(*nodes,elems,path,scale=1.,angle=0.,a1=None,a2=None*)

Sweep a quadrilateral mesh along a path

The path should be specified as a (n,2,3) Formex. The input grid (quadrilaterals) has to be specified with the nodes and elems and can for example be created with the functions gridRectangle or gridBetween2Curves. This quadrilateral grid should be within the YZ-plane. The quadrilateral grid can be scaled and/or rotated along the path.

There are three options for the first (a1) / last (a2) element of the path: 1) None: No corresponding hexahedral elements 2) 'last': The direction of the first/last element of the path is used to direct the input grid at the start/end of the path 3) specify a vector: This vector is used to direct the input grid at the start/end of the path

The resulting hexahedral mesh is returned in terms of nodes and elems.

`connectMesh`(*coords1,coords2,elems,n=1,n1=None,n2=None*)

Connect two meshes to form a hypermesh.

coords1,elems and coords2,elems are 2 meshes with same topology. The coordinates are given in corresponding order. The two meshes are connected by a higher order mesh with n elements in the direction between the two meshes. n1 and n2 are node selection indices permitting a permutation of the nodes of the base sets in their appearance in the hypermesh. This can e.g. be used to achieve circular numbering of the hypermesh.

`extrudeMesh`(*coords,elems,n,step=1.,dir=0,autofix=True*)

Extrude a mesh in one of the axes directions.

Returns a hypermesh obtained by extruding the given mesh (coords,elems) over n steps of length step in direction of axis dir. The returned mesh has double plexitude of the original.

This function is usually used to points into lines, lines into surfaces and surfaces into volumes. By default it will try to fix the connectivity ordering where appropriate. It autofix is switched off, the connectivities are merely stacked, and the user may have to fix it himself.

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

8.19 `project` — project.py

Functions for managing a project in pyFormex.

8.19.1 Project class: A pyFormex project is a persistent storage of pyFormex objects.

The Project class has this constructor:

class `Project` (*filename,create=False*)
Create a new project with the given filename.

If the filename exists and create is False, the file is opened and the contents is read into the project dictionary. If not, a new empty file and project are created.

Project objects have the following methods:

`save` (*filename=None*)
Save the project to file.

`load` (*filename=None*)
Load a project from file.

The loaded definition will update the current project.

`delete` ()
Unrecoverably delete the project file.

8.20 `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.

8.20.1 Database class: A class for storing properties in a database.

The Database class has this constructor:

class `Database` (*data={}*)
Initialize a database.

The database can be initialized with a dict.

Database objects have the following methods:

`readDatabase` (*filename,None*)
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.

8.20.2 MaterialDB class: A class for storing material properties.

The MaterialDB class has this constructor:

class `MaterialDB` (*data={}*)

Initialize a materials database.

If data is a dict, it contains the database. If data is a string, it specifies a filename where the database can be read.

MaterialDB objects have the following methods:

8.20.3 SectionDB class: A class for storing section properties.

The SectionDB class has this constructor:

class `SectionDB` (*data={}*)

Initialize a section database.

If data is a dict, it contains the database. If data is a string, it specifies a filename where the database can be read.

SectionDB objects have the following methods:

8.20.4 ElemSection class: Properties related to the section of an element.

The ElemSection class has this constructor:

class `ElemSection` (*section=None,material=None,orientation=None,behavior=None*)

Create a new element section property. Empty by default.

An element section property can hold the following sub-properties: - section: the section properties of the element. This can be a dict or a string. The required data in this dict depend on the sectiontype. Currently the following keys are used by fe_abq.py: - sectiontype: the type of section: one of following: '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). !! Currently only 'solid' and 'general' are allowed. - the cross section characteristics : cross_section, moment_inertia_11, moment_inertia_12, moment_inertia_22, torsional_rigidity - for sectiontype 'circ': radius - 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 - 'orientation' is a list of 3 direction cosines of the first beam section axis. - behavior: the behavior of the connector

ElemSection objects have the following methods:

`addSection` (*section*)

Create or replace the section properties of the element.

If 'section' is a dict, it will be added to 'self.secDB'. If 'section' is a string, this string will be used as a key to search in 'self.secDB'.

`computeSection` (*section*)

Compute the section characteristics of specific sections.

`addMaterial` (*material*)

Create or replace the material properties of the element.

If the argument is a dict, it will be added to 'self.matDB'. If the argument is a string, this string will be used as a key to search in 'self.matDB'.

8.20.5 ElemLoad class: Distributed loading on an element.

The ElemLoad class has this constructor:

class **`ElemLoad`** (*label=None,value=None*)
Create a new element load. Empty by default.

An element load can hold the following sub-properties: - label: the distributed load type label. - value: the magnitude of the distibuted load.

ElemLoad objects have the following methods:

8.20.6 CoordSystem class: A class for storing coordinate systems.

The CoordSystem class has this constructor:

class **`CoordSystem`** (*csys,cdata*)
Create a new coordinate system.

csys is one of 'Rectangular', 'Spherical', 'Cylindrical'. Case is ignored and the first letter suffices. cdata is a list of 6 coordinates specifying the two points that determine the coordinate transformation

CoordSystem objects have the following methods:

8.20.7 Amplitude class: A class for storing an amplitude.

The amplitude is a list of tuples (time,value).

The Amplitude class has this constructor:

class **`Amplitude`** (*data,definition='TABULAR'*)
Create a new amplitude.

Amplitude objects have the following methods:

8.20.8 PropertyDB class: 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)

The PropertyDB class has this constructor:

class **`PropertyDB`** ()
Create a new properties database.

PropertyDB objects have the following methods:

`autoName` (*clas,kind*)
This is a class method, not an instance method.

`setMaterialDB` (*aDict*)
Set the materials database to an external source

`setSectionDB` (*aDict*)
Set the sections database to an external source

`Prop` (*kind=",tag=None,set=None,setname=None*)
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 - set: a single number or a list of numbers identifying the geometrical elements for wich the property is set, or the name of a previously defined set. - setname: the name to be used for this set. Default is to use an automatically generated name. If setname is specified without a set, this is interpreted as a set= field. Besides these, any other fields may be defined and will be added without checking.

getProp (*kind=",rec=None,tag=None,attr=[],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. If a list of attibutes is given, only the properties having those attributes are returned.

If delete==True, the returned properties are removed from the database.

delete (*plist,kind="*)

Delete properties in list pdel from list plist.

sanitize (*kind*)

Sanitize the record numbers after deletion

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,setname=None,tag=None,cload=None,bound=None,displ=None,csys=None,ampl=None*)

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 values - bound: a boundary condition: a list of 6 codes (0/1) - displ: a prescribed displacement: a list of tuples (dofid,value) - csys: a CoordSystem - ampl: the name of an Amplitude

elemProp (*prop=None,grp=None,set=None,setname=None,tag=None,section=None,eltype=None,dload=None,ampl=None*)

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

8.20.9 Functions defined in the properties module

checkIdValue (*values*)

Check that a variable is a list of values or (id,value) tuples

If ok, return the values as a list of tuples.

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.

autoName (*base*)

Nset ()

Eset()

8.21 `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.

8.21.1 planeSection class: 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.

The planeSection class has this constructor:

class planeSection(*F*)

Initialize a plane section.

Initialization can be done either by a list of points or a set of line segments.

1. By Points Each point is connected to the following one, and (unless they are very close) the last one back to the first. Traversing the resulting path should rotate positively around the z axis to yield a positive surface.

2. By Segments It is the responsibilty of the user to ensure that the segments form a closed curve. If not, the calculated section data will be rather meaningless.

planeSection objects have the following methods:

sectionChar()

8.21.2 Functions defined in the section2d module

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

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 reutrns 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 glabla 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.)

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

8.22 `surface` — Import/Export Formex structures to/from stl format.

An stl is stored as a numerical array with shape [n,3,3]. This is compatible with the pyFormex data model.

8.22.1 TriSurface class: A class for handling triangulated 3D surfaces.

The TriSurface class has this constructor:

class TriSurface ()
Create a new 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 3-plex Formex with ntri elements ; - 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.

Internally, the surface is stored in a (coords,edges,faces) tuple.

TriSurface objects have the following methods:

getElems ()
Get the elems data.

setElems (*elems*)
Change the elems data.

refresh ()
Make the internal information consistent and complete.

This function should be called after one of the data fields have been changed.

compress ()
Remove all nodes which are not used.

Normally, the surface definition can hold nodes that are not used in the edge/facet tables. They do however influence the bounding box of the surface. This method will remove all the unconnected nodes.

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.

ncoords ()

nedges ()

nfaces ()

nplex ()

ndim ()

vertices()

shape()
Return the number of ;points, edges, faces of the TriSurface.

copy()
Return a (deep) copy of the surface.

If an index is given, only the specified faces are retained.

select(*idx,compress=True*)
Return a TriSurface which holds only elements with numbers in ids.

idx can be a single element number or a list of numbers or any other index mechanism accepted by numpy's ndarray By default, the vertex list will be compressed to hold only those used in the selected elements. Setting compress==False will keep all original nodes in the surface.

setProp(*p=None*)
Create or delete the property array for the TriSurface.

A property array is a rank-1 integer array with dimension equal to the number of elements in the TriSurface. 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 TriSurface.

prop()
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.

x()

y()

z()

bbox()

center()

centroid()

sizes()

dsize()

bsphere()

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 an (nfaces,3) shaped Coords array.

distanceFromPlane(*None*)

distanceFromLine(*None*)

distanceFromPoint(*None*)

test(*nodes='all',dir=0,min=None,max=None*)
Flag elements having nodal coordinates between min and max.

This function is very convenient in clipping a TriSurface 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 TriSurface

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 TriSurface 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 TriSurface. The resulting TriSurface will contain all elements where t > 0.

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

pointNormals()
Compute the normal vectors in each point of a collection of triangles.

The normal vector in a point is the average of the normal vectors of the neighbouring triangles. The normal vectors are normalized.

offset(*distance=1.*)
Offset a surface with a certain distance.

All the nodes of the surface are translated over a specified distance along their normal vector. This creates a new congruent surface.

feModel()
Return a tuple of nodal coordinates and element connectivity.

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)

This is a class method, not an instance method.

write(*fname,ftype=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 'gts' file type is used.

toFormex()
Convert the surface to a Formex.

scale(*None*)
This method is equivalent to the corresponding `Coords` *method executed on the objects coordinates.*

translate(*None*)
This method is equivalent to the corresponding `Coords` *method executed on the objects coordinates.*

rotate(*None*)
: *This method is equivalent to the corresponding* `Coords` *method executed on the objects coordinates.*

shear(*None*)
: *This method is equivalent to the corresponding* `Coords` *method executed on the objects coordinates.*

reflect(*None*)
: *This method is equivalent to the corresponding* `Coords` *method executed on the objects coordinates.*

affine(*None*)
: *This method is equivalent to the corresponding* `Coords` *method executed on the objects coordinates.*

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.

facetArea()

area()
: Return the area of the surface

volume()
: Return the enclosed volume of the surface.

  This will only be correct if the surface is a closed manifold.

edgeConnections()
: Find the elems connected to edges.

nodeConnections()
: Find the elems connected to nodes.

nEdgeConnected()
: Find the number of elems connected to edges.

nNodeConnected()
: Find the number of elems connected to nodes.

edgeAdjacency()
: Find the elems adjacent to elems via an edge.

nEdgeAdjacent()
: Find the number of adjacent elems.

nodeAdjacency()
: Find the elems adjacent to elems via one or two nodes.

nNodeAdjacent()
: Find the number of adjacent elems.

surfaceType()

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()

isClosedManifold()

checkBorder()
Return the border of TriSurface as a set of segments.

fillBorder(*method=0*)
If the surface has a single closed border, fill it.

Filling the border is done by adding a single point inside the border and connectin it with all border segments. This works well if the border is smooth and nearly planar.

border()
Return the border of TriSurface as a Plex-2 Formex.

edgeCosAngles()
Return the cos of the angles over all edges.

The surface should be a manifold (max. 2 elements per edge). Edges with only one element get angles = 1.0.

edgeAngles()
Return the angles over all edges (in degrees).

data()
Compute data for all edges and faces.

aspectRatio()

smallestAltitude()

longestEdge()

shortestEdge()

stats()
Return a text with full statistics.

edgeFront(*startat=0,okedges=None,front_increment=1*)
Generator function returning the frontal elements.

startat is an element number or list of numbers of the starting front. On first call, this function returns the starting front. Each next() call returns the next front. front_increment determines haw the property increases at each frontal step. There is an extra increment +1 at each start of a new part. Thus, the start of a new part can always be detected by a front not having the property of the previous plus front_increment.

nodeFront(*startat=0,front_increment=1*)
Generator function returning the frontal elements.

startat is an element number or list of numbers of the starting front. On first call, this function returns the starting front. Each next() call returns the next front.

walkEdgeFront(*startat=0,nsteps=???,okedges=None,front_increment=1*)

walkNodeFront(*startat=0,nsteps=???,front_increment=1*)

growSelection(*sel,mode='node',nsteps=1*)
Grows 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.

partitionByEdgeFront (*okedges,firstprop=0,startat=0*)
Detects different parts of the surface using a frontal method.

okedges flags the edges where the two adjacent triangles are to be in the same part of the surface. startat is a list of elements that are in the first part. The partitioning is returned as a property type array having a value corresponding to the part number. The lowest property number will be firstprop

partitionByNodeFront (*firstprop=0,startat=0*)
Detects different parts of the surface using a frontal method.

okedges flags the edges where the two adjacent triangles are to be in the same part of the surface. startat is a list of elements that are in the first part. The partitioning is returned as a property type array having a value corresponding to the part number. The lowest property number will be firstprop

partitionByConnection ()

partitionByAngle (*angle=180.,firstprop=0,startat=0*)

cutAtPlane (*None*)
Cut a surface with a plane.

connectedElements (*target,elemlist=None*)
Return the elements from list connected with target

smoothLowPass (*n_iterations=2,lambda_value=0.5*)
Smooth the surface using a low-pass filter.

smoothLaplaceHC (*n_iterations=2,lambda_value=0.5,alpha=0.,beta=0.2*)
Smooth the surface using a Laplace filter and HC algorithm.

check (*verbose=False*)
Check the surface using gtscheck.

split (*base,verbose=False*)
Check the surface using gtscheck.

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.

refine (*max_edges=None,min_cost=None,log=False,verbose=False*)
Refine the surface using gtsrefine.

smooth (*lambda_value=0.5,n_iterations=2,fold_smoothing=None,verbose=False*)
Smooth the surface using gtssmooth.

boolean (*surf,op,inter=False,check=False,verbose=False*)
Perform a boolean operation with surface surf.

8.22.2 Functions defined in the surface module

areaNormals (*x*)
Compute the area and normal vectors of a collection of triangles.

x is an (ntri,3,3) array of coordinates.

Returns a tuple of areas,normals. The normal vectors are normalized. The area is always positive.

stlConvert (*stlname,outname=None,options='-d'*)
Transform an .stl file to .off or .gts format.

If outname is given, it is either '.off' or '.gts' or a filename ending on one of these extensions. If it is only an extension, the stlname will be used with extension changed.

If the outname file exists and its mtime is more recent than the stlname, the outname file is considered uptodate and the conversion programwill 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).

read_gts (*fn*)
Read a GTS surface mesh.

Return a coords,edges,faces tuple.

read_off (*fn*)
Read an OFF surface mesh.

The mesh should consist of only triangles! Returns a nodes,elems tuple.

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.

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.

write_gts (*fn,nodes,edges,faces*)

write_stla (*f,x*)
Export an x[n,3,3] float array as an ascii .stl file.

write_stlb (*f,x*)
Export an x[n,3,3] float array as an binary .stl file.

write_gambit_neutral (*fn,nodes,elems*)

write_off (*fn,nodes,elems*)

write_smesh (*fn,nodes,elems*)

surface_volume (*x,pt=None*)
Return the volume inside a 3-plex Formex.

For each element of Formex, return the volume of the tetrahedron formed by the point pt (default the center of x) and the 3 points of the element.

surfaceInsideLoop (*coords,elems*)
Create a surface inside a closed curve defined by coords and elems.

coords is a set of coordinates. elems is an (nsegments,2) shaped connectivity array defining a set of line segments forming a closed loop.

The return value is coords,elems tuple where coords has one more point: the center of th original coords elems is (nsegment,3) and defines triangles describing a surface inside the original curve.

fillHole (*coords,elems*)
Fill a hole surrounded by the border defined by coords and elems.

Coords is a (npoints,3) shaped array of floats. Elems is a (nelems,2) shaped array of integers representing the border element numbers and must be ordered.

create_border_triangle (*coords,elems*)
Create a triangle within a border.

The triangle is created from the two border elements with the sharpest angle. Coords is a (npoints,3) shaped array of floats. Elems is a (nelems,2) shaped array of integers representing the border element numbers and must be ordered. A list of two objects is returned: the new border elements and the triangle.

coordsmethod (*f*)
Define a TriSurface method as the equivalent Coords method.

This decorator replaces the TriSurface's vertex coordinates with the ones resulting from applying the transform f.

The coordinates are changed inplane, so copy them before if you do not want them to be lost.

read_error (*cnt,line*)
Raise an error on reading the stl file.

degenerate (*area,norm*)
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.

read_stla (*fn,dtype=Float,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.

read_ascii_large (*fn,dtype=Float*)
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.

off_to_tet (*fn*)
Transform an .off model to tetgen (.node/.smesh) format.

find_row (*mat,row,nmatch=None*)
Find all rows in matrix matching given row.

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.

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.

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.

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.

Rectangle(*nx,ny*)
Create a plane rectangular surface consisting of a nx,ny grid.

Cube()
Create a surface in the form of a cube

Sphere(*level=4,verbose=False,filename=None*)
Create a spherical surface by caling the gtssphere command.

If a filename is given, it is stored under that name, else a temporary file is created. Beware: this may take a lot of time if level is 8 or higher.

8.23 `turtle` —

sind(*arg*)
Return the sin of an angle in degrees.

cosd(*arg*)
Return the sin of an angle in degrees.

reset()

push()

pop()

fd(*d=None,connect=True*)

mv(*d=None*)

ro(*a*)

go(*p*)

st(*d*)

an(*a*)

play(*scr,glob=None*)

8.24 `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.

roll(*a,n=1*)
Roll the elements of a list n positions forward (backward if n < 0)

`union` (*a,b*)
Return a list with all items in a or in b, in the order of a,b.

`difference` (*a,b*)
Return a list with all items in a but not in b, in the order of a.

`symdifference` (*a,b*)
Return a list with all items in a or b but not in both.

`intersection` (*a,b*)
Return a list with all items in a and in b, in the order of a.

`concatenate` (*a*)
Concatenate a list of lists

`flatten` (*a,recurse=False*)
Flatten a nested list.

By default, lists are flattened one level deep. If recurse=True, flattening recurses through all sublists.

`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.

`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.

8.25 `mydict` —

Dict is a dictionary with default values and alternate attribute syntax. CDict is a Dict with lookup cascading into the next level Dict's if the key is not found in the CDict itself.

(C) 2005,2008 Benedict Verhegghe Distributed under the GNU GPL version 3 or later

8.25.1 Dict class: 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, the following are equivalent: C['foo'] or C.foo This works as well for accessing values as for setting values. In the following, the words 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 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 one to watch out for: 'clear', 'copy', 'fromkeys', 'get', 'has_key', 'items', 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values'.

The Dict class has this constructor:

class `Dict` (*data={},default=None*)

Create a new Dict instance.

The Dict can be initialized with a Python dict or a Dict. If defined, default is a function that is used for alternate key lookup if the key was not found in the dict.

Dict objects have the following methods:

`__repr__` ()

Format the Dict as a string.

We use the format Dict({}), so that the string is a valid Python representation of the Dict.

`__getitem__` (*key*)

Allows items to be addressed as self[key].

This is equivalent to the dict lookup, except that we provide a default value if the key does not exist.

`__delitem__` (*key*)

Allow items to be deleted using del self[key].

Silently ignore if key is nonexistant.

`__getattr__` (*key*)

Allows items to be addressed as self.key.

This makes self.key equivalent to self['key'], except if key is an attribute of the builtin type 'dict': then we return that attribute instead, so that the 'dict' methods keep their binding.

`__setattr__` (*key,value=None*)

Allows items to be set as self.key=value.

This works even if the key is an existing attribute of the builtin dict class: the key,value pair is stored in the dict, leaving the dict's attributes unchanged.

`__delattr__` (*key*)

Allow items to be deleted using del self.key.

This works even if the key is an existing attribute of the builtin dict class: the item is deleted from the dict, leaving the dict's attributes unchanged.

`update` (*data={}*)

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.

`__deepcopy__` (*memo*)

Create a deep copy of ourself.

`__reduce__` ()

`__setstate__` (*state*)

8.25.2 CDict class: 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.

The CDict class has this constructor:

class `CDict` (*data={},default=returnNone*)

CDict objects have the following methods:

`__repr__` ()
: Format the CDict as a string.

  We use the format Dict({}), so that the string is a valid Python representation of the Dict.

`__str__` ()
: Format a CDict into a string.

`__getitem__` (*key*)
: Allows items to be addressed as self[key].

  This is equivalent to the dict lookup, except that we cascade through lower level dict's.

8.25.3 Functions defined in the mydict module

`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.

`returnNone` (*key*)
: Always returns None.

`raiseKeyError` (*key*)
: Raise a KeyError.

`__newobj__` (*cls*)

8.26 `odict` — A dictionary that keeps the keys in order of insertion.

8.26.1 ODict class: 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 class has this constructor:

class `ODict` (*data={}*)

Create a new ODict instance.

The ODict can be initialized with a Python dict or an ODict. The order after insertion is indeterminate if a plain dict is used.

ODict objects have the following methods:

`__repr__`()

Format the Dict as a string.

We use the format Dict({}), so that the string is a valid Python representation of the Dict.

`__setitem__`(*key,value*)

Allows items to be set using self[key] = value.

`__delitem__`(*key*)

Allow items to be deleted using del self[key].

Raises an error if key does not exist.

`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..

`__add__`(*data*)

Add two ODicts's together, returning the result.

`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.

8.26.2 KeyList class: A named item list.

A KeyList 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.

The KeyList class has this constructor:

class `KeyList` (*alist=[]*)

Create a new KeyList, possibly filling it with data.

data should be a list of tuples/lists each having at least 2 elements. The (string value of the) first is used as the key.

KeyList objects have the following methods:

`items`()

Return the key+value lists in order of the keys.

CHAPTER

NINE

PyFormex FAQ 'n TRICKS

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[1], and we'll probably include it in the next version.

9.1 FAQ

1. **How was the pyFormex logo created?**

   With the Gimp, using the following command sequence:

   ```
   Xtra -> Script-Fu -> Logos -> Alien-neon
   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
   ```

   Then switch off the background layer and save the image in PNG format. Export the image with 'Save Background Color' switched off!

2. **Why is pyFormex written in Python?**

   Because

   - it is very easy to learn (See www.python.org)
   - it is extremely powerful (More on www.python.org)

   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.

3. **Is an interpreted language like Python fast enough with large data models?**

   See the question above

[1] By preference via the forums on the pyFormex web site

9.2 TRICKS

1. **Set the directory where a script is found 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.[2] 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. **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.

[2]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!

- 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 m
  ```

  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
  ```

- **Install the gl2ps extension**

  Saving images in EPS format is done through the gl2ps library, which can be accessed from Python using the wrappers from python-gl2ps-1.1.2.tar.gz.

  You need to have the OpenGL header files installed in order to do this. (On Debian: apt-get install libgl1-mesa-dev.)

  ```
  tar xvzf python-gl2ps-1.1.2.tar.gz
  cd python-gl2ps-1.1.2
  su root
  python setup.py install
  ```

APPENDIX

A

GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

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

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.

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.

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.

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.

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.

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.

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.

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.

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. 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.

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.

INDEX