

---

# pyFormex manual

*pyFormex 0.5a2*

Tim Neels & Benedict Verhegghe

August 9, 2007

# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	What is pyFormex? . . . . .	3
1.2	License and Disclaimer . . . . .	5
1.3	Installation . . . . .	5
1.4	Running pyFormex . . . . .	7
1.5	Quick tutorial for the pyFormex GUI . . . . .	7
1.6	Quick Python tutorial . . . . .	8
1.7	Quick NumPy tutorial . . . . .	8
<b>2</b>	<b>pyFormex tutorial</b>	<b>9</b>
2.1	Introduction . . . . .	9
2.2	Getting started . . . . .	9
2.3	The geometrical model . . . . .	11
2.4	Adding properties . . . . .	21
2.5	Exporting to finite element programs . . . . .	25
<b>3</b>	<b>The Canvas</b>	<b>27</b>
3.1	Introduction . . . . .	27
3.2	Drawing a Formex . . . . .	28
3.3	Viewing the scene . . . . .	28
3.4	Other canvas items . . . . .	29
<b>4</b>	<b>The Graphical User Interface</b>	<b>31</b>
4.1	Starting the GUI . . . . .	31
4.2	Basic use of the GUI . . . . .	31
4.3	The file menu . . . . .	32
4.4	The viewport menu . . . . .	32
4.5	Customizing the GUI . . . . .	32
<b>5</b>	<b>Configuring pyFormex</b>	<b>35</b>
<b>6</b>	<b>pyFormex example scripts</b>	<b>37</b>
6.1	Creating geometry . . . . .	37
6.2	Operating on surface meshes . . . . .	37
<b>7</b>	<b>pyFormex plugins</b>	<b>39</b>
<b>8</b>	<b>pyFormex — reference manual</b>	<b>41</b>
8.1	formex — the base module . . . . .	41
8.2	simple — simple geometries . . . . .	54

8.3 Drawing . . . . .	54
<b>A The GNU General Public License</b>	<b>57</b>
<b>Index</b>	<b>63</b>

## **Abstract**

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<sup>1</sup> that were automatically generated from the pyFormex source.

---

<sup>1</sup><http://pyformex.berlios.de/doc/index.html>

## CONTENTS

---

# 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 prerequisites software parts that you need to have installed in order to be able to run this program. We explain how to download and install pyFormex. Finally, you'll find out what basic knowledge you should have in order to understand the tutorial and successfully use pyFormex.

## 1.1 What is pyFormex?

You probably expect to find here a short definition of what pyFormex is and what it can do for you. I may have to disappoint you: describing the essence of pyFormex in a few lines is not 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<sup>1</sup>. 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, most 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<sup>2</sup> 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 ??:

- 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.

<sup>1</sup><http://pyformex.berlios.de>

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

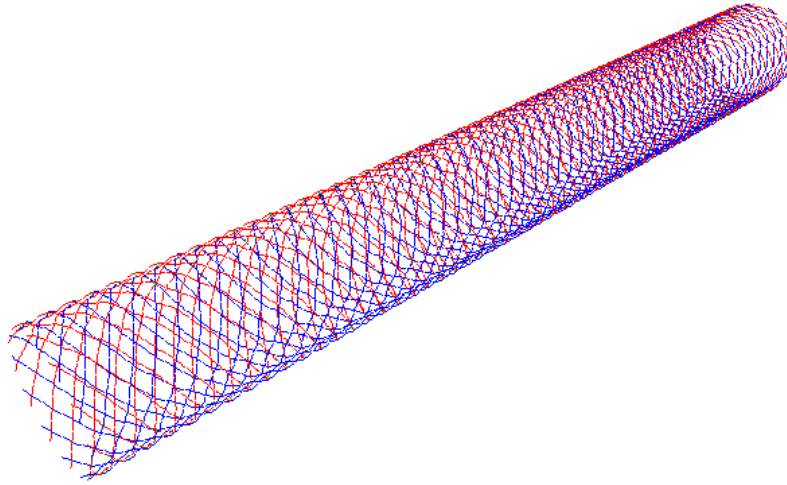


Figure 1.1: WireStent example.

- Roll the planar grid into a cylinder.

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

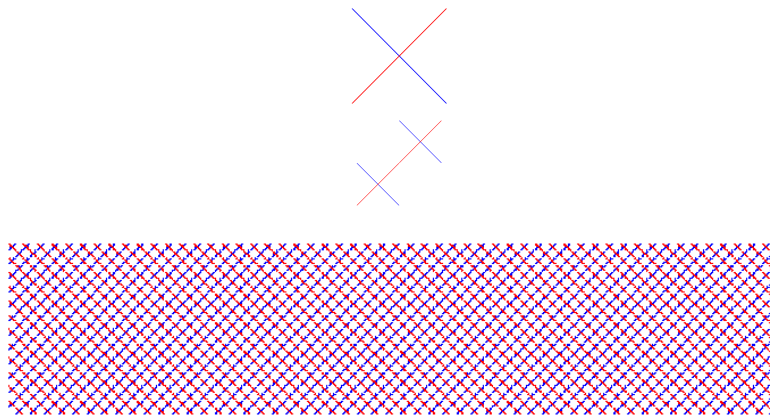


Figure 1.2: Steps in building the WireStent example.

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 preprocessor. 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.

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.

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 ©2004-2007 Benedict Verhegghe

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License<sup>3</sup> as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

Full details of the license are available in appendix A and in the file COPYING included with the distribution.

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<sup>4</sup>: 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<sup>5</sup>: 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.<sup>6</sup> On Linux systems, installing NumPy from source is usually straightforward. Debian users can install the packages python-numpy, python-numpy-dev and python-numpy-ext. There are binary packages for Windows on the Sourceforge download page<sup>7</sup>.

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.

- Qt<sup>8</sup>: The widget toolkit on which the GUI was built. For Debian users this comes in the packages python-qt4 and python-qt4-dev.

---

<sup>3</sup><http://www.gnu.org/licenses/gpl.html>

<sup>4</sup><http://www.python.org>

<sup>5</sup><http://numpy.scipy.org/>

<sup>6</sup>The Numarray package, which was used up until pyFormex 0.3, is no longer supported

<sup>7</sup><http://www.numpy.org/>

<sup>8</sup><http://www.trolltech.com/products/qt>



- PyQt4<sup>9</sup>: The Python bindings for Qt4. Debian users should install the packages `python-qt4`, `python-qt4-dev` and `python-qt4-gl`.
- PyOpenGL<sup>10</sup>: Python bindings for OpenGL, used for drawing and manipulating the 3D-structures. For debian users this is in the package `python-opengl`.

### 1.3.2 Downloading

The official releases of pyFormex can be downloaded from the pyFormex website<sup>11</sup>. As of the writing of this manual, the latest release is pyFormex 0.5a2<sup>12</sup>. 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<sup>13</sup>. 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<sup>14</sup>. If you have Subversion<sup>15</sup> 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

```
make install
```

This will install pyFormex in `/usr/local/lib/pyformex-version`. If you want to install somewhere else, you need to change the Makefile first.

The installation procedure installs everything into this single directory, but also makes a couple of symlinks: `/usr/local/share/doc/pyformex-VERSION` is a link to the `doc` subdirectory of `pyformex`, and `/usr/local/bin/pyformex` is a link to the main executable `pyformex.py`.

### 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.

---

<sup>9</sup><http://www.riverbankcomputing.co.uk/pyqt/index.php>

<sup>10</sup><http://pyopengl.sourceforge.net/>

<sup>11</sup><http://pyformex.berlios.de>

<sup>12</sup><http://prdownload.berlios.de/pyformex/pyformex.tar.gz>

<sup>13</sup><ftp://bumps.ugent.be/pub/pyformex>

<sup>14</sup><http://pyformex.berlios.de>

<sup>15</sup><http://subversion.tigris.org/>

## 1.4 Running pyFormex

To start pyFormex, enter the command `pyformex --gui`. 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.

The pyFormex program takes some optional command line arguments, that modify the behaviour of the program.

### 1.4.1 Command line options

The whole list of possible command line options can be seen by executing the command `pyformex --help`. This will produce the following output.

```
Could not import pyformex.  
This probably means that pyFormex was not properly installed.
```

*As of version 0.4.2, the `--nogui` option has become the default. This is because the command line will mostly be used to start non-interactive execution of pyFormex, while the GUI will often be launched from a start button or menu, which can easily incorporate the `--gui` option.*

### 1.4.2 Running pyFormex without the GUI

If you start pyFormex *without* the `--gui` option, no Graphical User Interface is created. This is extremely useful to run automated scripts in batch mode. In this operating mode, pyFormex will interpret all arguments remaining after interpreting the options, as filenames of scripts to be run (and possibly arguments to be interpreted by these scripts). Thus, if you want to run a pyFormex script `myscript.py` in batch mode, just give the command `pyformex myscript.py`.

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

## 1.5 Quick tutorial for the pyFormex GUI

In the current version (0.5a2) 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, change, import or export Formex structures.

The GUI does not (yet) provide a means to interactively design a structure, select parts of a structure or set/show information about (parts of) the structure. Designing a structure is done by writing a small script

with the mathematical expressions needed to generate it. Any text editor will be suitable for this purpose. The author uses XEmacs, but this is just a personal preference. A pyFormex editor integrated into the GUI remains on our TODO list, but it certainly is not our top priority, because general purpose editors are adequate for most of our purposes.

Since Python<sup>16</sup> is the language used in pyFormex scripts, a Python aware editor is preferable. It will highlight the syntax and help you with proper alignment (which is very important in Python).

The best way to learn to use pyFormex is by studying and changing some of the examples. I suggest that you first take a look at the examples included in the pyFormex GUI and select those that display structures that look interesting to you. Then you can study the source code of those examples and see how the structures got built. When starting up, pyFormex reads through the Examples directory (this is normally the 'examples' subdirectory located under the pyformex installation dir). Examples > WireStent

## 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<sup>17</sup>. If you are new to Python, but have already some programming experience, the Python tutorial<sup>18</sup> may be a good starting point. Or else, you can take a look at one of the beginners' guides<sup>19</sup>.

## 1.7 Quick NumPy tutorial

Numerical Python (or NumPy for short) is an extension to the Python language that allows for 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<sup>20</sup> already provides the basics.

---

<sup>16</sup><http://www.python.org>

<sup>17</sup><http://www.python.org>

<sup>18</sup><http://docs.python.org/tut/>

<sup>19</sup><http://wiki.python.org/moin/BeginnersGuide/>

<sup>20</sup>[http://www.scipy.org/Tentative\\_NumPy\\_Tutorial](http://www.scipy.org/Tentative_NumPy_Tutorial)

# PyFormex tutorial

## 2.1 Introduction

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 frame structures. But it can also be used for other tasks, like finite element preprocessing, or just for creating some nice pictures.

By writing a very simple script, a large geometry can be created by copying, translating, rotating,... Formices. pyFormex will interpret this script and draw what you have created. This is clearly very different than the traditional way of creating a model, like CAD. There are two huge advantages about using pyFormex:

- It is especially suited for the automated design of spatial frame structures. A dome, arc, hyper,... can be rather difficult to draw with CAD, but when using mathematical transformations, it becomes a piece of cake!
- Using a script makes it very easy to apply changes in the geometry: you simply modify the script and let pyFormex play it again. For instance, you can easily change an angle, the radius of a dome, the ratio  $f/l$  of an arc,... Using CAD, you would have to make an entirely new drawing! This is also illustrated in fig 2.1: these domes were all created with the same script, but with other values of the parameters.

## 2.2 Getting started

This is the first paragraph.

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

- Each script should begin with `#!/usr/bin/env pyformex`
- To start the pyFormex GUI, double click on the file 'pyformex' in the installation directory, or type *pyformex* in the terminal. Using the terminal can be very useful, because errors that are created while running the script will appear in the terminal. This can provide useful information when something goes wrong with your script.
- To create a new pyFormex-script, just open a new file with your favorite text editor and save it as 'myproject.py'.
- To edit a script, you can

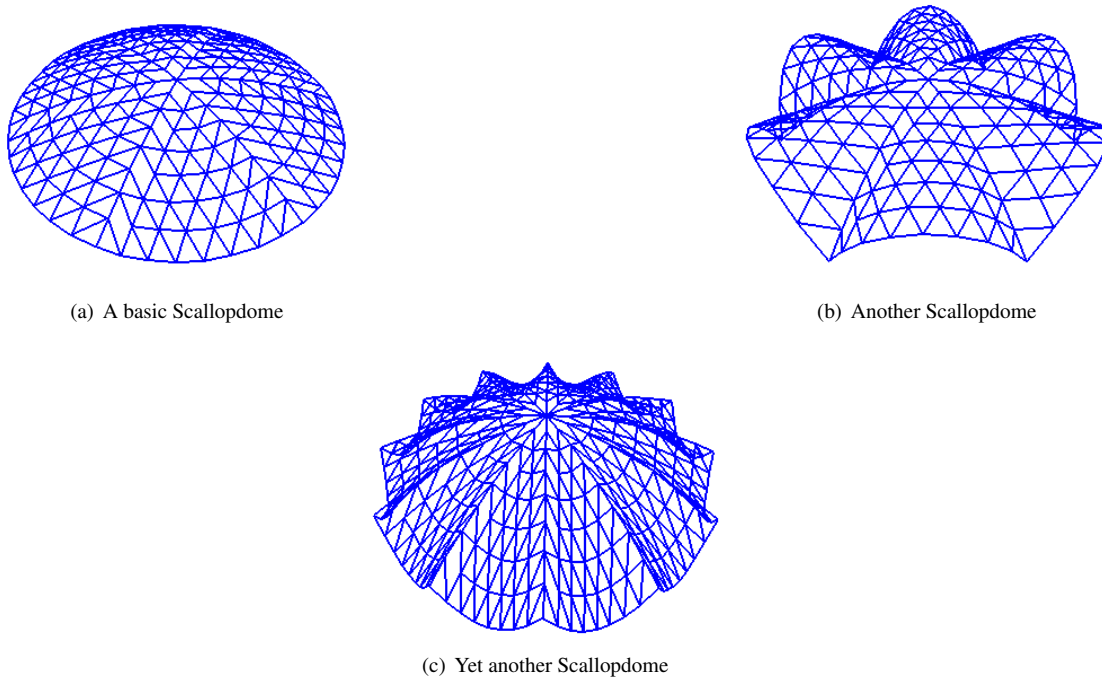


Figure 2.1: Same script, different domes

- 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 file `‘.pyformexcrc’` in the installation directory.
- 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 separate 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 sensitive.

## 2.3 The geometrical model

### 2.3.1 Creating a Formex

What is a Formex?

A Formex is a Numarray of order 3 (axes 0,1,2) and type Float. A scalar element represents a coordinate (F:uniple).

A row along the axis 2 is a set of coordinates and represents a point (node, vertex, F: signet). For simplicity's sake, the current implementation only deals with points in a 3-dimensional space. This means that the length of axis 2 is always 3. The user can create Formices (plural of Formex) in a 2-D space, but internally these will be stored with 3 coordinates, by adding a third value 0. All operations work with 3-D coordinate sets. However, a method exists to extract only a limited set of coordinates from the results, permitting to return to a 2-D environment.

A plane along the axes 2 and 1 is a set of points (F: cantle). This can be thought of as a geometrical shape (2 points form a line segment, 3 points make a triangle, ...) or as an element in FE terms. But it really is up to the user as to how this set of points is to be interpreted.

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

Additionally, a Formex may have a property set, which is an 1-D array of integers. The length of the array is equal to the length of axis 0 of the Formex data (i.e. the number of elements in the Formex). Thus, a single integer value may be attributed to each element. It is up to the user to define the use of this integer (e.g. it could be an index in a table of element property records). If a property set is defined, it will be copied together with the Formex data whenever copies of the Formex (or parts thereof) are made. Properties can be specified at creation time, and they can be set, modified or deleted at any time. Of course, the properties that are copied in an operation are those that exist at the time of performing the operation.

Simply put: a Formex is a set of elements, and every element can have a property number.

#### Creating a Formex using coordinates

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

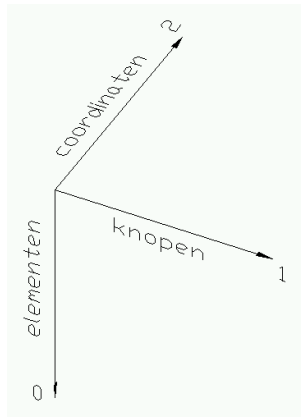


Figure 2.2: The scheme of a Formex

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

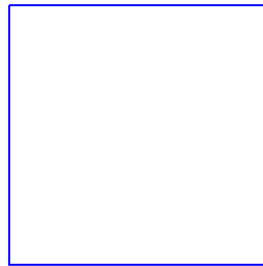


Figure 2.3: A very simple Formex

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]]])
```

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').



Figure 2.4: Same nodes, different Formex

```
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]]])
```

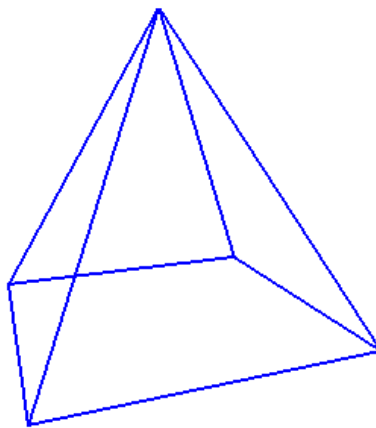


Figure 2.5: A pyramid

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 2.3.6 — or by using patterns.

### Creating a Formex using patterns

The second way of creating a new Formex, is by defining patterns. 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 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 'ABCDEFGH', 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)
```

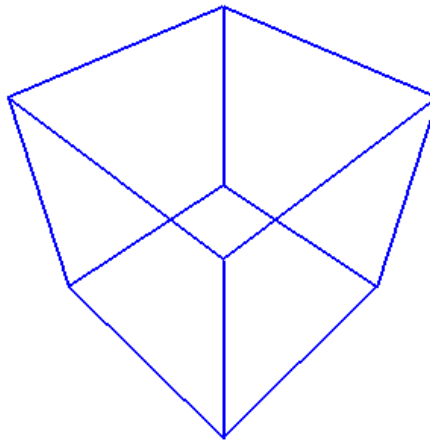


Figure 2.6: A cube

### Creating a Formex using coordinates from a file

In some cases, you might want to read coordinates from a file and combine them into a Formex. This is possible with the module `file2formex` and its 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).

```
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')
```

### 2.3.2 Adding property numbers

Each Formex element can have a property number. Each property number is represented by a different color when the Formex is drawn. This is the first reason why you could use property numbers: to make your drawing more transparent or just more beautiful. However, these numbers can also be used as an entry in a dictionary of properties - thus linking the element with a property. This property can be about anything, but in finite element processing this would be the element section, material, loads,... The use of properties in this way will be further explained in 2.4. Property numbers can be specified at creation time, and they can be set, modified or deleted at any time.

```
>>> #!/usr/bin/env pyformex
>>> F=Formex(pattern('1234'), [5])
>>> print F.prop()
>>> G=Formex(pattern('1234'), [6,8,2,4])
>>> print G.prop()
>>> F.setProp([6,7])
>>> print F.prop()
>>> G.setProp([6,7,8,9])
>>> print G.prop()

[5 5 5 5]
[6 8 2 4]
[6 7 6 7]
[6 7 8 9]
```

### 2.3.3 Drawing a Formex

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 is 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!

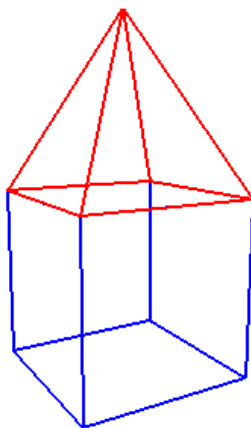


Figure 2.7: Drawing multiple Formices

Now, when you are creating a large geometry, you might be interested in seeing the different steps in the creation. To remove all previously drawn Formices, you can use `clear()` what sweeps 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 Saving images

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 ??, which shows the different steps in the creation of the WireStent model, was created this way.

File > Toggle MultiSave

### 2.3.5 Information about a Formex

There are a number of functions available that return information about a Formex. Especially when using pyFormex as finite element preprocessor, the most useful functions are:

Function	Description
<code>F.nelems()</code>	Return the number of elements in the Formex.
<code>F.nnodes()</code>	Return the number of nodes in the Formex.
<code>F.prop()</code>	Return the properties as a numpy array.
<code>F.bbox()</code>	Return the bounding box of the Formex.
<code>F.center()</code>	Return the center of the Formex.
<code>F.feModel()</code>	Return a tuple of nodal coordinates and element connectivity.

`feModel()` is very important in finite element processing. It returns all nodes and all elements of the Formex in a format useful for FE processing. A tuple of two arrays is returned. The first is float array with the coordinates of the unique nodes of the Formex. The second is an integer array with the node numbers connected by each element.

```
>>> #!/usr/bin/env pyformex
>>> from simple import *

>>> c = Formex(pattern(Pattern['cube']))
>>> draw(c)
>>> nodes,elems = c.feModel()
>>> print 'Nodes'
>>> print nodes
>>> print 'Elements'
>>> print elems

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

### 2.3.6 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.translatel(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.translatel(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)
```

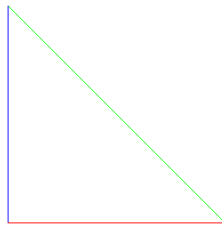


Figure 2.8: The basic Formex

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 ( $F=F.\text{replic}$ ), 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')
```

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')
```

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

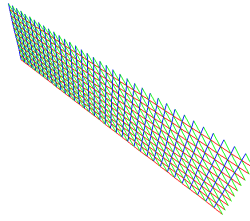


Figure 2.9: The rectangular pattern

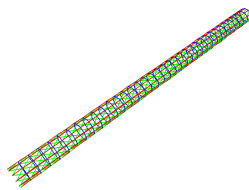


Figure 2.10: The cylinder

the spiral.

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

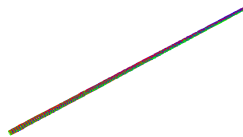


Figure 2.11: The new cylinder

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

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

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

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



Figure 2.12: The spiral

## 2.4 Adding properties

Each element of a Formex can hold a property number. This number can be used as an entry in a database, which holds some sort of property. The module `properties` delivers such databases. The Formex and the database are separate entities, only linked by the property numbers.

### 2.4.1 The class `Property`

The first kind of database is called `Property`. This is the base class, and can hold any kind of property.

```
>>> Stick = Property(1, {'colour':'green', 'name':'Stick', 'weight': 25,
'comment':'This could be anything: a gum, a frog, a usb-stick,...'})
>>> author = Property(5, {'Name':'Tim Neels', 'Address': CascadingDict
{'street':'Krijgslaan', 'city':'Gent', 'country':'Belgium'}})
```

Data can be accessed in two ways: through the `Property`-instance itself, or through a dict `'properties'`.

```
>>> print Stick
>>> print properties[1]

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

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

Adding and changing properties is very easy.

```
>>> Stick.weight=30
>>> Stick.length=10
>>> print properties[1]

comment = This could be anything: a gum, a frog, a usb-stick,...
length = 10
colour = green
name = Stick
weight = 30
```



### 2.4.2 The class NodeProperty

The class NodeProperty can hold properties of nodes in finite element processing. The data is stored in a Dict called 'nodeproperties'. A NodeProperty can hold following sub-properties:

**cload** A concentrated load. This is a list of 6 items [F\_0, F\_1, F\_2, M\_0, M\_1, M\_2].

**bound** A boundary condition. This can be defined in 2 ways:

- as a list of 6 items [ R\_0, R\_1, R\_2, M\_0, M\_1, M\_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. In Abaqus several types are available:
  - PINNED
  - ENCASTRE
  - XSYMM
  - YSYMM
  - ZSYMM
  - XASYMM
  - YASYMM
  - ZASYMM

**displacement** The prescribed displacement.

**coords** The coordinate system which is used for the definition of cload and bound. There are three options: cartesian, spherical and cylindrical.

**coordset** A list of 6 coordinates of the 2 points that specify the transformation: [x\_1, y\_1, z\_1, x\_2, y\_2, z\_2].

```
>>> top = NodeProperty(1,cload=[5,0,-75,0,0,0])
>>> foot = NodeProperty(2,bound='pinned')
>>> print 'nodeproperties'
>>> for key, item in nodeproperties.iteritems():
>>>     print key, item
```

```
nodeproperties
```

```
1
  cload = [5, 0, -75, 0, 0, 0]
  coords = cartesian
  displacement = None
  bound = None
  coordset = []
2
  cload = None
  coords = cartesian
  displacement = None
  bound = pinned
  coordset = []
```

### 2.4.3 The class ElemProperty

The class ElemProperty holds properties related to a single element. The data is stored in a Dict called 'elemproperties'. An element property can hold the following sub-properties:

**elemsection** The section properties of the element. This is an ElemSection instance.

**elemload** The loading of the element. This is a list of ElemLoad instances.

**elemtype** The type of element that is to be used in the analysis.

An elemsection can hold the following sub-properties:

**section** The section properties of the element. This can be a dictionary or a string. The required data in this dict depends on the sectiontype.

**material** The element material properties. This can be a dictionary which holds these properties or a string which can be used to search a material database.

**sectiontype** The sectiontype of the element.

**orientation** A list [First direction cosine, second direction cosine, third direction cosine] of the first beam section axis. This allows to change the orientation of the cross-section.

An element load can hold the following sub-properties:

**magnitude** The magnitude of the distributed load.

**loadlabel** The distributed load type label.

The general structure of the element properties database looks like this:

- Property
- NodeProperty
  - cload
  - bound
  - displacement
  - coords
  - coordset
- ElemProperty
  - elemsection
    - section
    - material
    - sectiontype
    - orientation
  - elemload
    - magnitude
    - loadlabel
  - elemtype

```
>>> vert = ElemSection('IPEA100', 'steel')
>>> hor = ElemSection({'name': 'IPEM800', 'A': 951247, 'I': CascadingDict(
{'Ix': 1542, 'Iy': 6251, 'Ixy': 352})}, {'name': 'S400', 'E': 210, 'fy': 400})
>>> q = ElemLoad(magnitude=2.5, loadlabel='PZ')
>>> top = ElemProperty(1, hor, [q], 'B22')
>>> column = ElemProperty(2, vert, elemtype='B22')
>>> diagonal = ElemProperty(4, hor, elemtype='B22')
```

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

```
elemproperties
```

```
1
  elemtype = B22
  elemload = [CascadingDict({'magnitude': 2.5, 'loadlabel': 'PZ'})]
  elemsection =
    section =
      A = 951247
      I =
        Iy = 6251
        Ix = 1542
        Ixy = 352
      name = IPEM800
  material =
    fy = 400
    E = 210
    name = S400
  orientation = None
  sectiontype = general
```

```
2
  elemtype = B22
  elemload = None
  elemsection =
    section =
      torsional_rigidity = 1542
      name = IPEA100
      moment_inertia_22 = 1140000
      cross_section = 878
      moment_inertia_11 = 1412000
      moment_inertia_12 = 1254
  material =
    shear_modulus = 25
    young_modulus = 210
    name = steel
  orientation = None
  sectiontype = general
```

```
4
  elemtype = B22
  elemload = None
  elemsection =
    section =
      A = 951247
      I =
        Iy = 6251
        Ix = 1542
        Ixy = 352
      name = IPEM800
  material =
    fy = 400
    E = 210
    name = S400
```

## 2.5 Exporting to finite element programs



# The Canvas

## 3.1 Introduction

When you have create 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 one to four viewports, which can each individually be used for your drawing purposes. See the viewport menu of the GUI for details about using multiple viewports. In the remainder of this section we 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 displaying 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 8.3, the number of arguments looks frightening. However, most of these arguments have sensible default values, making the access to drawing functionality easy even for beginners. To display your created Formex `F` on the screen, a simple `draw(F)` will suffice in many cases.

If you draw several Formices with subsequent `draw()` commands, they will clutter the view. You can use the `clear()` instruction to wipe out the screen before drawing the next one. If you want to see them together in the same view, you can use different colors to differentiate. Color drawing is as easy as `draw(F, color='red')`. The color specification can take various forms. It can be a single color or an array of colors or even an array of indices in a color table. In the latter case you use `draw(F, color=indices, colormap=table)` to draw the Formex. If multiple colors are specified, each element of the Formex will be drawn with the corresponding color, and if the color array (or the color indices array) has less entries than the number of elements, it is wrapped around.

A single color entry can be specified by a string ('red') or by a triplet 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 properties as a color index into the specified color map or a (configurable) default color map.

*There should be some examples here.*

## 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 described in 4. 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, you will notice that moving the scene will not just translation the picture: its shape will change too, because of changing perspective.

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

The easiest way to set all camera parameters for properly viewing a scene is by just telling the direction from which you want to look, and let the program determine the rest of the settings itself. `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





# 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.

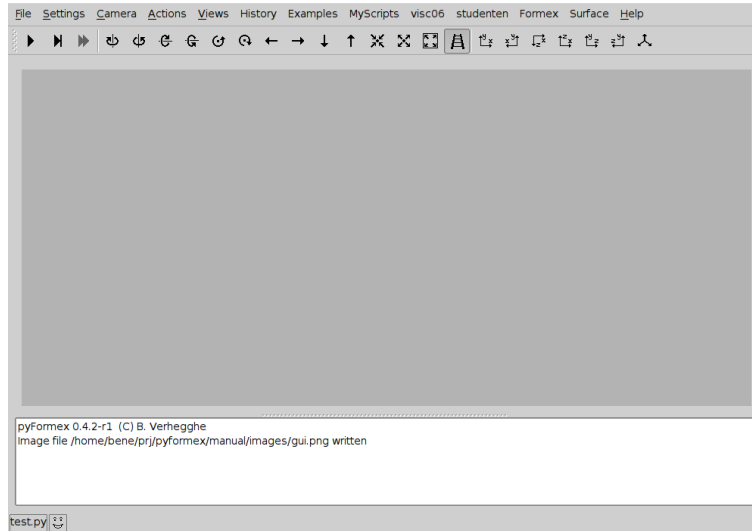


Figure 4.1: The pyFormex main window

## 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 important functions are described in following sections.

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

The canvas is a drawing board where your 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 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.

### 4.5.1 Changing the appearance of the GUI

### 4.5.2 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 not start with a `'.'` or `'_'`, will be included in the menu. If your `scriptdirs` setting has only one item, the menu item will be created directly in the menubar. If there are multiple items, a top menu named `'Scripts'` will be created with submenus for each entry.

Notice the special entry for the examples supplied with the distribution. You do not specify the directory where the examples are: you would probably not even know the correct path, and it could change when a new version of 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.

### 4.5.3 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`.



# Configuring pyFormex

This chapter has not been written yet.



## PyFormex example scripts

### 6.1 Creating geometry

This section has not been written yet.

### 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 6.1 shows an example of such a stent prior to expansion (balloon not shown). The 3D surface is obtained by micro CT and consists of triangles.

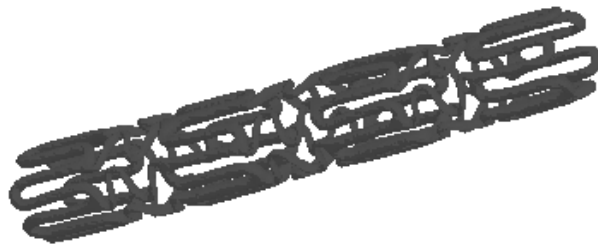


Figure 6.1: Triangulated mesh of a stent

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 operation of the stent gives a much better overview of the complete geometrical structure and allows easier analysis (see figure 6.2).

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

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



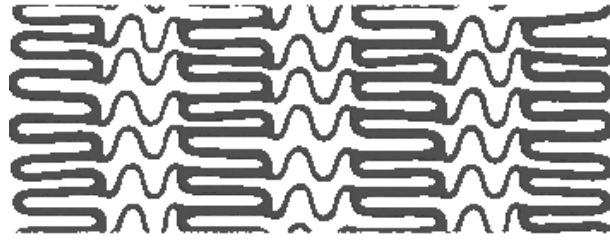


Figure 6.2: Result of unroll operation

order to obtain a line describing the stent cell, the function `intersectionLinesWithPlane()` has been used. The result can be seen in figure 6.4.



Figure 6.3: Intersection of stent cell with plane



Figure 6.4: Inner line of stent cell

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

# PyFormex plugins

This chapter has not been written yet.



# PyFormex — reference manual

## 8.1 formex — the base module

This module contains all the basic functionality for creating, structuring and transforming sets of coordinates. All the definitions in this module are available in your scripts without the need to import the module.

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<sup>1</sup> 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).

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.

---

<sup>1</sup>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.

**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 Formex 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 encouraged.

### 8.1.2 Formex class members

**f**

A three dimensional array of float values. The array always has a shape with three components (`nelems, nplex, 3`), even if the Formex is empty. An empty Formex has `shape[0] == 0`.

**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.__setitem__(i, val)`. 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.

**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* () .

**sizes** ()

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

**size** ()

Return the size of the Formex. This is defined as 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.

**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 will 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 objects.

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.

#### **nodes** ()

Return a Formex containing only the nodes.

This is obviously a Formex with plexitude 1. It holds the same data as the original Formex, but in another shape: the number of nodes per element is 1, and the number of elements is equal to the total number of nodes. 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*.

If the Formex has no properties, a copy is returned. The returned Formex is always without properties.

#### **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.

**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) the elements having nodal 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.

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

- a single (integer) node number (< the number of nodes)
- a list of node 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.

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*.

**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.

**translate** (\*args)

Multiple subsequent translations in axis directions.

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

### 8.1.12 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, even if the file contains more than one. Use it repeatedly 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.*

### 8.1.13 Non-member functions

The following functions operate on or return Formex objects, but are not part of the Formex class.<sup>2</sup>

**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.

---

<sup>2</sup>They might be implemented as class methods in future releases.

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*)

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  $H_{ijk} = F_{ijk} + v * (G_{ijk} - F_{ijk})$ . 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))`

#### **divide** (*F,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].

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

#### **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`.

#### **bbox** (*formexlist*)

Computes the overall bounding box of a list of Formices.

The result has the same format as Formex class `bbox()` method, but the resulting box encloses all the Formices in the list.

### 8.1.14 Other functions

The following functions are defined in the file `formex.py`, but do not depend on the Formex class. They are defined here because they are mainly supporting functions for the Formex class itself.

#### **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.

**length** (*arg*)

Return the quadratic norm of a vector with all elements of *arg*.

**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*)

Return an array flagging the elements in *values* that are 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 the values that are close to *target*.

Two values *a* and *b* are considered close if  $\text{abs}(a-b) < \text{atol} + \text{rtol} * \text{abs}(b)$

**vectorPairAreaNormals** (*vec1,vec2*)

Compute area of and normals on parallelograms 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 parallelogram 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 product of *vec1* and *vec2*, which indeed results in *area* \* *normal*.

**vectorPairArea** (*vec1,vec2*)

Compute the area of the parallelogram 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)`.

**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]

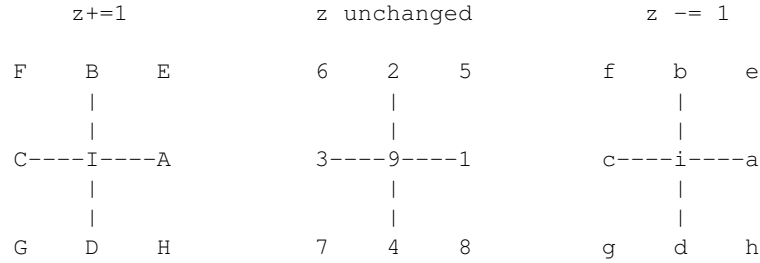
1..8 move in the x,y plane

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:



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.

**distanceFromPlane** (*f,p,n*)

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

*f* is an [...,3] array of coordinates. *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** (*f,p,q*)

Return the distance of points *f* from the line (*p,q*).

*f* is an [...,3] array of coordinates. *p* and *q* are two points specified by 3 coordinates.

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

**distanceFromPoint** (*f,p*)

Return the distance of points *f* from the point *p*.

*f* is an [...,3] array of coordinates. *p* is a point specified by 3 coordinates.

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

## 8.2 simple — simple geometries

This module contains some functions, data and classes for generating Formices with simple geometries.

**circle** (*a1=1.,a2=2.,a3=360.*)

Creates a Formex which is 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 by the begin and end point of each line segment.

*a2* (the module angle) is the angle enclosed between the start points of two subsequent line segments.

*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.

Remark that *a1* == *a2* produces a continues line, *a1* < *a2* gives a dashed line. The default *a3*=360. produces a full circle; for *a3* < 360, the result is an arc. Large angle values result in polygons: circle(120,120) is an equilateral triangle and circle(60,60) is regular hexagone. The default values produces a dashed (near-)circle.

## 8.3 Drawing

The following function provide drawing capabilities to your scripts.

**draw** (*F, view=None, bbox='auto', color='prop', colormap=None, wait=True, eltype=None, al-views=False, marksize=None, linewidth=None, alpha=0.5*)

Draw a Formex or a list of Formices on the canvas.

If *F* is a list, all its items are drawn with the same settings.

If a setting is unspecified, its current values are used.

Draws an actor on the canvas, and directs the camera to it from the specified view. Named views are either predefined or can be added by the user. If *view=None* is specified, the camera settings remain unchanged. This may make the drawn object out of view! A special name '\_\_last\_\_' may be used to keep the same camera angles as in the last draw operation. The camera will be zoomed on the newly drawn object. The initial default view is 'front' (looking in the -z direction).

If *bbox* == 'auto', the camera will zoom automatically on the shown object. A *bbox* may be specified to have other zoom settings, e.g. to keep the previous settings. If *bbox* == None, the previous *bbox* will be kept.

If other actors are on the scene, they may or may not be visible with the new camera settings. Clear the canvas before drawing if you only want a single actor!

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 than 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).



# The GNU General Public License

Version 2, June 1991

Copyright © 1989, 1991 Free Software Foundation, Inc.

51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

## **Preamble**

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) 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 this service 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 make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. 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.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author’s protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors’ reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program

proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

### GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute 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 and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
  - (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
  - (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
  - (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

---

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
  - (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
  - (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
  - (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

### NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

- 
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

## END OF TERMS AND CONDITIONS

### Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>  
Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

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

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) <year> <name of author>  
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.  
This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program  
‘Gnomovision’ (which makes passes at compilers) written by James Hacker.



## APPENDIX A. THE GNU GENERAL PUBLIC LICENSE

---

<signature of Ty Coon>, 1 April 1989  
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

# INDEX

## Symbols

`__add__()` (Formex method), 45  
`__getitem__()` (Formex method), 42  
`__setitem__()` (Formex method), 42

## A

`affine()` (Formex method), 47  
`append()` (Formex method), 45  
`asArray()` (Formex method), 44  
`asFormex()` (Formex method), 44  
`asFormexWithProp()` (Formex method), 44

## B

`bbox()`  
    Formex method, 43  
    in module , 51  
`bsphere()` (Formex method), 43  
`bump()` (Formex method), 48  
`bump1()` (Formex method), 48  
`bump2()` (Formex method), 48

## C

`cclip()` (Formex method), 46  
`center()` (Formex method), 43  
`circle()` (in module ), 54  
`circulize()` (Formex method), 49  
`circulize1()` (Formex method), 49  
`clip()` (Formex method), 46  
`concatenate()` (Formex method), 45  
`connect()` (in module ), 50  
`coord()` (Formex method), 42  
`copy()` (Formex method), 45  
`cosd()` (in module ), 51  
`cylindrical()` (Formex method), 47

## D

`data()` (Formex method), 43  
`distanceFromLine()` (in module ), 54  
`distanceFromPlane()` (in module ), 53  
`distanceFromPoint()` (in module ), 54  
`divide()` (in module ), 51

`draw()` (in module ), 54

## E

`elbbox()` (Formex method), 45  
`elem2str()` (Formex method), 44  
`element()` (Formex method), 42  
`equivalence()` (in module ), 53

## F

`f` (array attribute), 42  
`feModel()` (Formex method), 44  
Formex, 11  
Formex (class in ), 42  
`fprint()` (Formex method), 44

## I

`inside()` (in module ), 52  
`interpolate()` (in module ), 51  
`isClose()` (in module ), 52

## L

`length()` (in module ), 52

## M

`map()` (Formex method), 48  
`map1()` (Formex method), 48  
`mapd()` (Formex method), 48  
`maxprop()` (Formex method), 43

## N

`ndim()` (Formex method), 43  
`nelems()` (Formex method), 43  
`nodes()` (Formex method), 45  
`nplex()` (Formex method), 43  
`npoints()` (Formex method), 43

## P

`p` (array attribute), 42  
`pattern()` (in module ), 52  
`point()` (Formex method), 42

point2str() (Formex method), 44  
prop() (Formex method), 43  
propSet() (Formex method), 43

## R

read() (Formex method), 50  
readfile() (in module ), 51  
reflect() (Formex method), 47  
remove() (Formex method), 45  
replace() (Formex method), 49  
replic() (Formex method), 49  
replic2() (Formex method), 49  
reverseElements() (Formex method), 46  
rosette() (Formex method), 49  
rotate() (Formex method), 47  
rotationAboutMatrix() (in module ), 53  
rotationMatrix() (in module ), 53

## S

scale() (Formex method), 46  
select() (Formex method), 45  
selectNodes() (Formex method), 45  
setPrintFunction() (Formex method), 44  
setProp() (Formex method), 44  
shape() (Formex method), 43  
shear() (Formex method), 47  
shrink() (Formex method), 49  
sind() (in module ), 51  
size() (Formex method), 43  
sizes() (Formex method), 43  
spherical() (Formex method), 47  
swapaxes() (Formex method), 49

## T

tand() (in module ), 52  
test() (Formex method), 46  
toCylindrical() (Formex method), 47  
toSpherical() (Formex method), 48  
translate() (Formex method), 47  
translatem() (Formex method), 50  
translationVector() (in module ), 53

## U

unique() (Formex method), 46

## V

vectorPairArea() (in module ), 52  
vectorPairAreaNormals() (in module ), 52  
vectorPairNormals() (in module ), 52

## W

withProp() (Formex method), 45  
write() (Formex method), 50

## X

x() (Formex method), 43

## Y

y() (Formex method), 43

## Z

z() (Formex method), 43