pyFormex manual

Release 0.3

Tim Neels and Benedict Verhegghe

May 30, 2006

CHAPTER

ONE

Introduction

1.1 What is pyFormex?

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

The short answer is that pyFormex is a program to *generate large structured sets of coordinates by means of subsequent mathematical transformations gathered in a script.* If you find this definition too dull, incomprehensible or just not descriptive enough, read on through this section and look at some of the examples in this manual and on the pyFormex website[1]. You will then probably have a better idea of what pyFormex is.

The initial intent of pyFormex was the rapid design of three-dimensional structures with a configuration 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.

The stent[2] structure in the figure below is a good illustration of what pyFormex can do and what it was intended for.

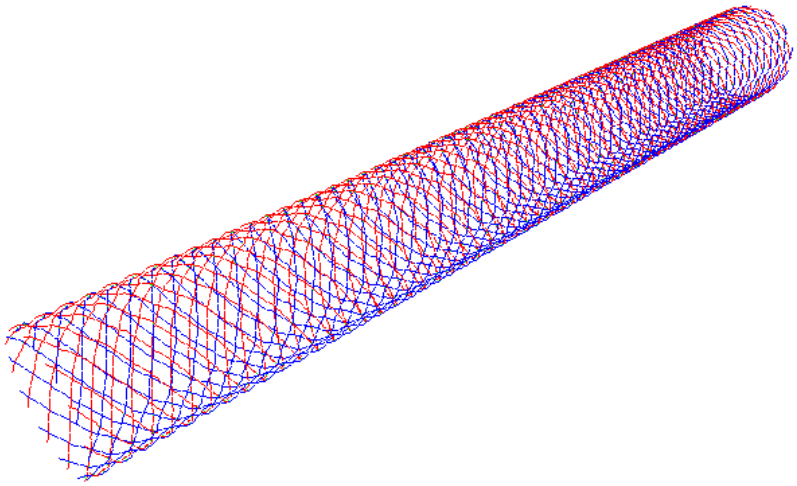


Figure 1.1: WireStent example.

This structure is composed of 22032 line segments, each with 2 nodes. Nobody in his right mind would ever even try

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

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

to input all the 132192 coordinates of all the points describing that structure. With pyFormex, one could define the structure by a sequence of operations like this:

- Create a planar base module of two crossing wires.
- Extend the base module with a mirrored and translated copy.
- Replicate the base module in both directions of the base plane.
- Roll the planar grid into a cylinder.

The procedure is illustrated by the subsequent images in the figure below.

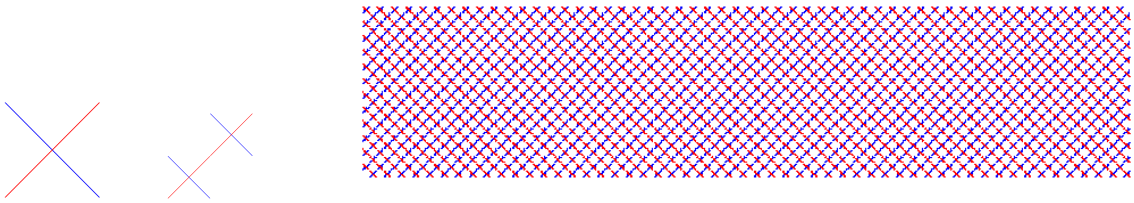


Figure 1.2: WireStent example.

1.2 Rationale

1.3 History

1.4 Installation

1.4.1 Installation on Linux platforms

1.5 Quick tutorial for the pyFormex GUI

In the current version () 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 there aren't many yet that can be changed through the GUI).
- Running pyFormex scripts, possibly starting other programs and display their results.

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 Python aware editor is preferable though, because that is the language used in pyFormex scripts. 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.

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' subdirecty located under the pyformex installation dir). Examples $>$ WireStent

1.6 Quick Python tutorial

This could be part of the tutorial in chapter 2

1.7 Quick NumPy tutorial

This could be part of the tutorial in chapter 2

CHAPTER

TWO

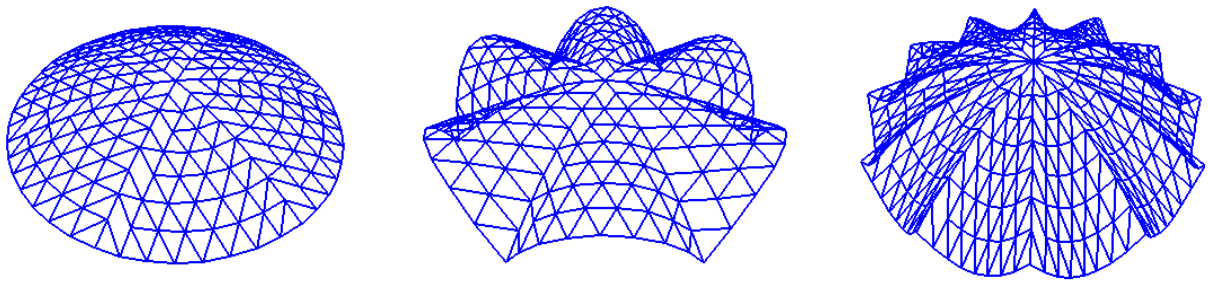
PYFORMEX— a tutorial

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, hypar,... 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.



(a) A basic Scallopdome (b) Another Scallopdome (c) Yet another Scallopdome

Figure 2.1: Same script, different domes

As mentioned, pyFormex is based on the programming language Python [1]. This implies that the scripts are also Python-based. It's a very easy language, but if you're interested in reading more, there is a very good tutorial available on `http://docs.python.org/tut/`. However, if you're only using Python to write pyFormex-scripts, the tutorial you're reading right now should be enough.

[1] `http://www.python.org`

2.2 Getting started

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
  - 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 '.pyformexrc' 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 seperate module (like `properties`), you add a line on top of the script

```
from properties import *
```

All functions from that module are now available.

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

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.

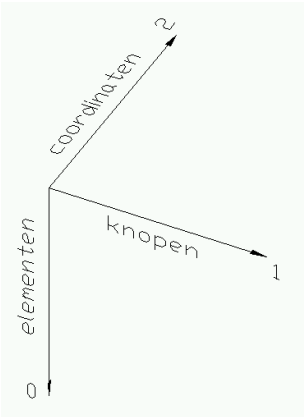


Figure 2.2: The scheme of a Formex

Creating a Formex using coordinates

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

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

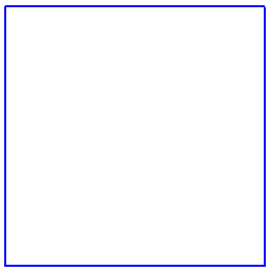


Figure 2.3: A very simple Formex

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

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

Figure 2.4: Same nodes, different Formex

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

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

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

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

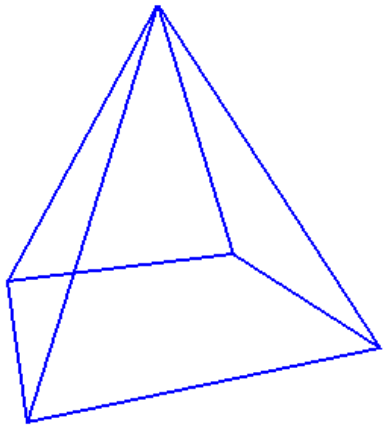


Figure 2.5: A pyramid

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 'ABCDEFGHI', resp. 'abcdefghi', correspond with '123456789' with an extra z +/-= 1. The special character '\' can be put before any character to make the move without making a connection. The effect of any other character is undefined.

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

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

Some simple patterns are defined in `simple.py` and are ready for use. These patterns are stacked in a dictionary

called 'Patterns'. Items of this dictionary can be accessed like `Patterns['cube']`.

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

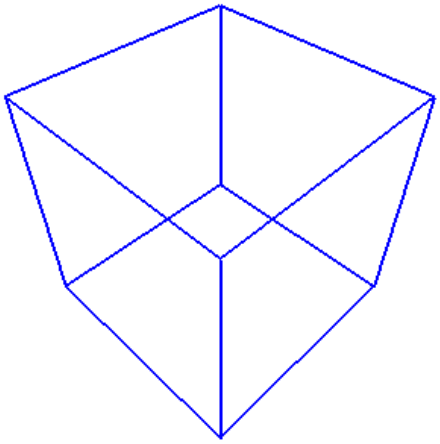


Figure 2.6: A cube

Creating a Formex using coordinates from a file

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

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

```
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 futher 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 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)
```

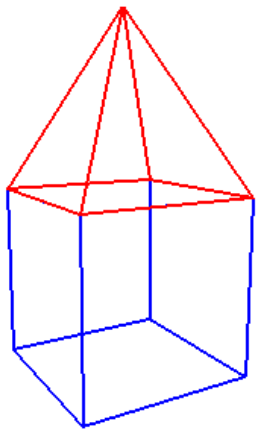


Figure 2.7: Drawing multiple Formices

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

Now, when you are creating a large geometry, you might be interested in seeing the different steps in the creation. To remove all previously drawn Formices, you can use `clear()` what sweepes the screen clean. If you want to see a certain step in the creation longer than the default time, use `sleep(t)`, with *t* the delay (in seconds) before executing the next command.

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

2.3.4 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 1.1, 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**
`F.nelems()`	Return the number of elements in the Formex.
`F.nnodes()`	Return the number of nodes in the Formex.
`F.prop()`	Return the properties as a numpy array.
`F.bbox()`	Return the bounding box of the Formex.
`F.center()`	Return the center of the Formex.
`F.nodesAndElements()`	Return a tuple of nodal coordinates and element connectivity.

`nodesAndElements()` 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.nodesAndElements()
>>> 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 3 is for. A summary of all possible transformations and functions can be found there.

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

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

```
##
#
"""Spiral"""

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

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

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

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

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

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

Now, let's dissect the script.

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

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

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

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

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

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

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

Now a copy of this last Formex is translated in direction '2' with a translation step of '1'. This necessary for the

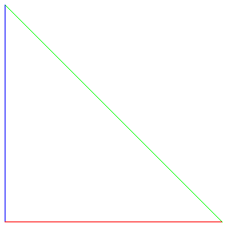


Figure 2.8: The basic Formex

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.translate1(2,1); drawit(F,'iso')
```

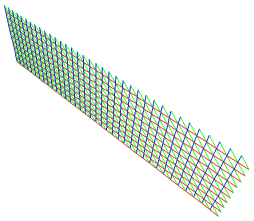


Figure 2.9: The rectangular pattern

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

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

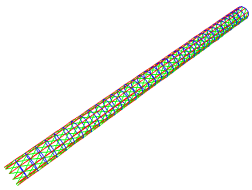


Figure 2.10: The cylinder

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

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

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

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

Figure 2.11: The new cylinder

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

```
F = F.translate1(0,5); drawit(F,'iso')
```

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

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

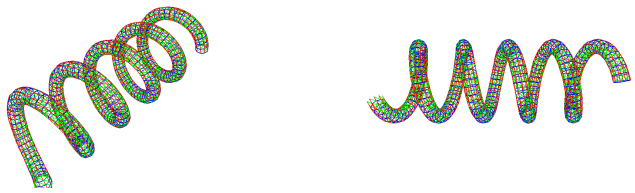


Figure 2.12: The spiral

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 seperate 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 $[F0, F1, F2, M0, M1, M2]$.

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

- as a list of 6 items $[R0, R1, R2, M0, M1, M2]$. 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: the 2 points that specify the transformation.
$[x1, y1, z1, x2, y2, z2]$

```
>>> 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 distibuted 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
    orientation = None
    sectiontype = general
```

2.5 Export to finite element programs

CHAPTER

THREE

PYFORMEX— reference manual

3.1 formex — the base module

This module contains all the basic functionality for creating, structuring and transforming sets of coordinates.

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

A class to hold a structured set of coordinates. A `Formex` is a three dimensional array of float values. The array has a shape `(nelems,nnodel,3)`. Each slice `[i,j]` of the array contains the three coordinates of a point in space. We will also call this a *node*. Each slice `[i]` of the array contains a connected set of nnodel points: we will refer to it as an *element*.

It is up to the user on how to interprete this connection: two connected nodes will usually represent a line segment between these two points. An element with three nodes however could just as well be interpreted as a triangle or as a (possibly curved) line. And 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. When `nnodel==1`, the `Formex` contains only unconnected nodes (each element is just one point).

One way of attaching other data to the `Formex`, is by the use of the 'property' attribute. The property is an array holding one integer value for each of the elements of the Formex. The use of this property value is completely defined by the user. It could be a code for the type of element, or for the color to draw this element with. Most 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.

f

A three dimensional array of float values. The array has a shape (nelems,nnodel,3). Each slice [i,j] of the array contains the three coordinates of a point in space. We will also call this a *node*. Each slice [i] of the array contains a connected set of nnodel points: we will refer to it as an *element*. It is up to the user on how to interprete this connection: two connected nodes will usually represent a line segment between these two points. An element with three nodes however could just as well be interpreted as a triangle or as a (possibly curved) line. And if it is a triangle, it could be either the circumference of the triangle or the part of the plane inside that circumference.

nelems()

Return the number of elements in the Formex.

nnodel()

Return the number of nodes per element.

Examples:

1: unconnected nodes,
2: straight line elements,
3: triangles or quadratic line elements,
4: tetraeders or quadrilaterals or cubic line elements.

nnodes()
Return the number of nodes in the Formex.

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

prop()
Return the properties as a numpy array.

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.

The center of the Formex is the center of its `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.

nodesAndElements(*nodesperbox=1,repeat=True*)
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 = nodesAndElements(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.

setProp(*p=0*)
Create a property array for the Formex.

A property array is a rank-1 integer array with dimension equal to the number of elements in the Formex (first dimension of data). You can specify a single value or a list/array of integer values.

If the number of passed values is less than the number of elements, they wil be repeated. If you give more, they will be ignored. The default argument will give all elements a property value 0.

removeProp()
Remove the properties from a Formex.

copy()
Returns a deep copy of itself.

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.

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(*vector,distance=None*)

Returns a copy translated over *distance* in the direction of *vector*.

If no distance is given, translation is over the specified vector. If a distance is given, translation is over the specified distance in the direction of the vector.

translate1(*dir,distance*)

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

The direction is specified by the axis number (0,1,2).

rotate(*angle,axis=2*)

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

The angle is specified in degrees. Default rotation is around z-axis.

rotateAround(*vector,angle*)

Returns a copy rotated over *angle* around *vector*.

The angle is specified in degrees. The rotation axis is specified by a vector of three values. It is an axis through the center.

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.

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.]*)

Converts from spherical to cartesian after scaling.

dir specifies which coordinates are interpreted as resp. distance(r), longitude(theta) and colatitude(phi).
scale will scale the coordinate values prior to the transformation.
Angles are then interpreted in degrees.
Colatitude is 90 degrees - latitude, i.e. the elevation angle measured from north pole(0) to south pole(180). This choice facilitates 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()`.

newmap (*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*.

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

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

The original Formex is the first of the n replicas.

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

Replicate in two directions.

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

`rosette`(*n,angle,axis=2,point=[0.,0.,0.]*)

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

`translatem`(**args*)

Multiple subsequent translations in axis directions.

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

`circulize`(*angle*)

Transform a linear sector into a circular one.

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

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

Returns a general affine transform of the Formex.

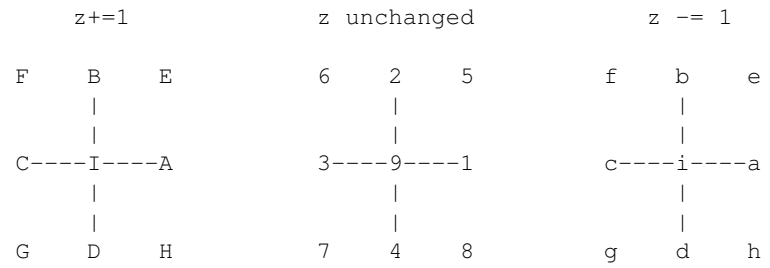
The returned Formex has coordinates given by `mat * xorig + vec`, where *mat* is a 3x3 matrix and *vec* a length 3 list.

`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.

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 id to be used for each of the formices. Finally, a list of bias values may be given to specify an offset in element number for the subsequent formices.

If `loop==False`, the order of the Formex will be the minimum order of the formices in *Flist*, each minus its respective bias. By setting `loop=True` however, each Formex will loop around if its end is encountered, and the order of the result is the maximum order in *Flist*.