# Geomodeller revisited

Brief introduction to GeoModeler

Kjetil A. Johannessen

February 12, 2014

**Why fix the geomodeller**

GoTools is lacking in several places

- Incomplete
- Unstable
- Inconsistent
- Unresponsive

Technical difficulties by writing C in python

**Other IGA and modelling libraries**

- GeoPDE - University of Pavia
- IGAkit - King Abdullah University of Science and Technology

All have very weak modelling capabilities

### Other CAD tools

- Rhinoceros 3D
- SolidWorks

are proprietaire software.

### Other CAD tools

- Rhinoceros 3D
- SolidWorks

are proprietaire software.

Open source alternatives include:

- FreeCAD - no splines, lots of crashbugs
- Blender - for artists
- openNURBS - C++ library, used by Rhino 3D

### Other CAD tools

- Rhinoceros 3D
- SolidWorks

are proprietaire software.

Open source alternatives include:

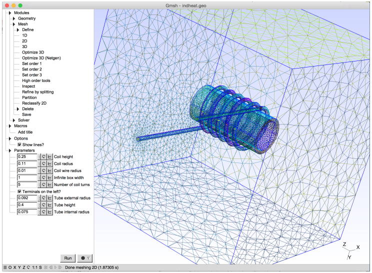
- FreeCAD - no splines, lots of crashbugs
- Blender - for artists
- openNURBS - C++ library, used by Rhino 3D

but still no volumetric modelling capabilities.

# Closest design cousin:

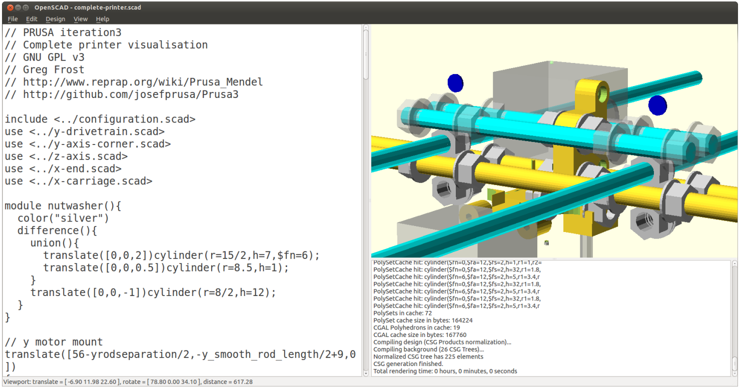
## GMSH:

```
cm = 1e-02;
e1 = 4.5 * cm; e2 = 6 * cm / 2; e3 =  5 * cm / 2;
// ...

Point(1) = {-e1-e2, 0    , 0, Lc1};
Point(2) = {-e1-e2, h1   , 0, Lc1};
// ...
Point(24)= { 0, h1+h3+h4+R2, 0, Lc2};
Point(25)= { 0, h1+h3-R2,    0, Lc2};

Line(1)  = {1 , 17};
Line(2)  = {17, 16};

Circle(3) = {14,15,16};

Line(4)  = {14,13};
// ...
Circle(8) = {8,9,10};
Line(9)  = {8,7};
// ...
Line(20) = {21,22};

Line Loop(21) = {17,-15,18,19,-20,16};
Plane Surface(22) = {21};
```



# Closest design cousin:

## OpenSCAD:



**We need**

- Analysis ready
  - watertight
  - non-overlapping
  - conforming
- Volumetric (trivariate) modelling support
- Scriptable
- Full discretization control

**We need**

- Analysis ready
  - watertight
  - non-overlapping
  - conforming
- Volumetric (trivariate) modelling support
- Scriptable
- Full discretization control

*Easy to learn, hard to master*

**Basics**

Based on python

- Fairly easy scripting language
- Full power of numpy (C fast linear algebra)

**Basics**

Based on python

- Fairly easy scripting language
- Full power of numpy (C fast linear algebra)

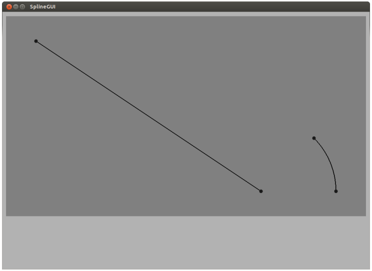
You make geometries by

- Creating curves from points
- Creating surfaces from curves
- Creating volumes from surfaces

...roughly speaking

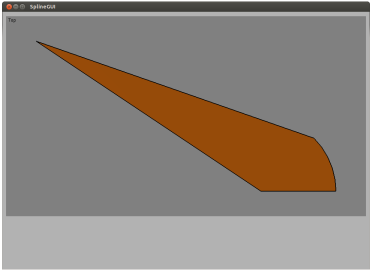
## Examples:

```
from math import *
import CurveFactory



myLine = CurveFactory.line([0,0],  [-3,2])
myArc  = CurveFactory.circle_segment(pi/4)









# write results to file
f = open('tutorial.g2', 'w')
myLine.write_g2(f)
myArc.write_g2(f)
```



## Examples:

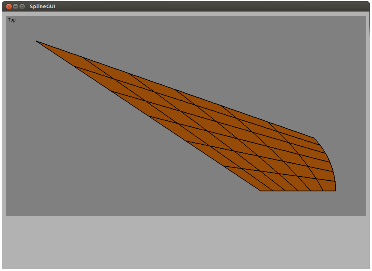
```
from math import *
import CurveFactory
import SurfaceFactory


myLine = CurveFactory.line([0,0],  [-3,2])
myArc  = CurveFactory.circle_segment(pi/4)

mySurface = SurfaceFactory.edge_curves([myLine, myArc])
```




```
# write results to file
f = open('tutorial.g2', 'w')
mySurface.write(f)
```

## Examples:

```
from math import *
import CurveFactory
import SurfaceFactory


myLine = CurveFactory.line([0,0],  [-3,2])
myArc  = CurveFactory.circle_segment(pi/4)

mySurface = SurfaceFactory.edge_curves([myLine, myArc])
mySurface.refine(5)
```

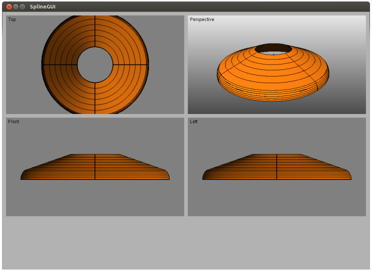



```
# write results to file
f = open('tutorial.g2', 'w')
mySurface.write(f)
```

# Examples:

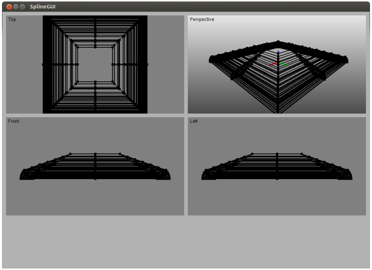
## Examples:

```
from math import *
import CurveFactory
import SurfaceFactory
import VolumeFactory

myLine = CurveFactory.line([0,0],  [-3,2])
myArc  = CurveFactory.circle_segment(pi/4)

mySurface = SurfaceFactory.edge_curves([myLine, myArc])
mySurface.refine(5)

mySurface.translate((2,0,0))     # move 2 in x-direction
mySurface = mySurface + (2,0,0) # move 2 in x-direction
mySurface += (1,0,0)            # move 1 in x-direction
mySurface.rotate(pi/2, (1,0,0)) # rotate into xz-plane

myVolume = VolumeFactory.revolve(mySurface)

# write results to file
f = open('tutorial.g2', 'w')
myVolume.write(f)
```



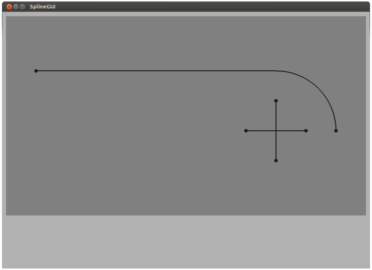
## Examples:

```
from math import *
import CurveFactory
import SurfaceFactory
import VolumeFactory

myLine = CurveFactory.line([0,0],  [-3,2])
myArc  = CurveFactory.circle_segment(pi/4)

mySurface = SurfaceFactory.edge_curves([myLine, myArc])
mySurface.refine(5)

mySurface.translate((2,0,0))     # move 2 in x-direction
mySurface = mySurface + (2,0,0) # move 2 in x-direction
mySurface += (1,0,0)             # move 1 in x-direction
mySurface.rotate(pi/2, (1,0,0)) # rotate into xz-plane

myVolume = VolumeFactory.revolve(mySurface)

# write results to file
f = open('tutorial.g2', 'w')
myVolume.write(f)
```



## Examples:

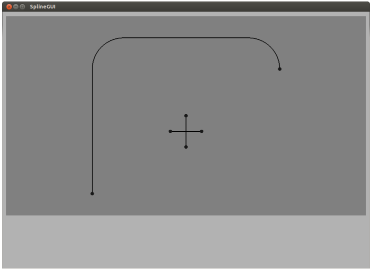
```
import CurveFactory   as cf
import SurfaceFactory as sf
from math import pi

x  = cf.line([-.5, 0], [.5, 0]) # just to show the origin
y  = cf.line([0, -.5], [0, .5]) # just to show the origin

c1 = cf.circle_segment(pi/2)
c2 = cf.line([0,1], [-4,1])
c1.append(c2)
```




```
f = open('ntnu.g2', 'w')
c1.write_g2(f)

x.write_g2(f)
y.write_g2(f)
```

## Examples:

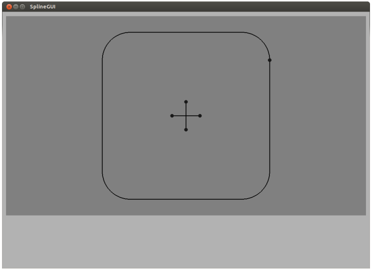
```
import CurveFactory   as cf
import SurfaceFactory as sf
from math import pi

x  = cf.line([-.5, 0], [.5, 0]) # just to show the origin
y  = cf.line([0, -.5], [0, .5]) # just to show the origin

c1 = cf.circle_segment(pi/2)
c2 = cf.line([0,1], [-4,1])
c1.append(c2)
c1 += (2, 2)
c2 = c1.clone().rotate(pi/2)
c1.append(c2)
```




```
f = open('ntnu.g2', 'w')
c1.write_g2(f)

x.write_g2(f)
y.write_g2(f)
```

## Examples:

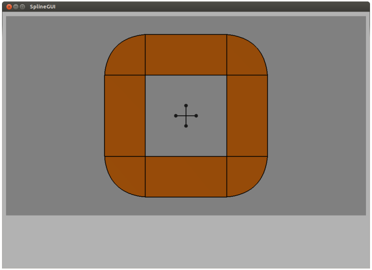
```
import CurveFactory   as cf
import SurfaceFactory as sf
from math import pi

x  = cf.line([-.5, 0], [.5, 0]) # just to show the origin
y  = cf.line([0, -.5], [0, .5]) # just to show the origin

c1 = cf.circle_segment(pi/2)
c2 = cf.line([0,1], [-4,1])
c1.append(c2)
c1 += (2, 2)
c2 = c1.clone().rotate(pi/2)
c1.append(c2)
c2 = c1.clone().rotate(pi)
c1.append(c2)
```




```
f = open('ntnu.g2', 'w')
c1.write_g2(f)

x.write_g2(f)
y.write_g2(f)
```

## Examples:

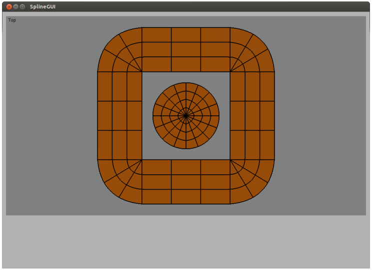
```
import CurveFactory   as cf
import SurfaceFactory as sf
from math import pi

x  = cf.line([-.5, 0], [.5, 0]) # just to show the origin
y  = cf.line([0, -.5], [0, .5]) # just to show the origin

c1 = cf.circle_segment(pi/2)
c2 = cf.line([0,1], [-4,1])
c1.append(c2)
c1 += (2, 2)
c2 = c1.clone().rotate(pi/2)
c1.append(c2)
c2 = c1.clone().rotate(pi)
c1.append(c2)

s1 = sf.thicken(c1, 1)
```




```
f = open('ntnu.g2', 'w')
s1.write_g2(f)

x.write_g2(f)
y.write_g2(f)
```

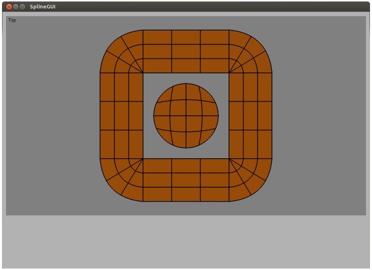
## Examples:

```
import CurveFactory   as cf
import SurfaceFactory as sf
from math import pi

x  = cf.line([-.5, 0], [.5, 0]) # just to show the origin
y  = cf.line([0, -.5], [0, .5]) # just to show the origin

c1 = cf.circle_segment(pi/2)
c2 = cf.line([0,1], [-4,1])
c1.append(c2)
c1 += (2, 2)
c2 = c1.clone().rotate(pi/2)
c1.append(c2)
c2 = c1.clone().rotate(pi)
c1.append(c2)

s1 = sf.thicken(c1, 1)
s2 = sf.disc(1.5)
s1.refine(2)
s2.refine(3)

f = open('ntnu.g2', 'w')
s1.write_g2(f)

# x.write_g2(f)
# y.write_g2(f)
```



## Examples:

```
import CurveFactory   as cf
import SurfaceFactory as sf
from math import pi

x  = cf.line([-.5, 0], [.5, 0]) # just to show the origin
y  = cf.line([0, -.5], [0, .5]) # just to show the origin

c1 = cf.circle_segment(pi/2)
c2 = cf.line([0,1], [-4,1])
c1.append(c2)
c1 += (2, 2)
c2 = c1.clone().rotate(pi/2)
c1.append(c2)
c2 = c1.clone().rotate(pi)
c1.append(c2)

s1 = sf.thicken(c1, 1)
s2 = sf.disc(1.5, 'square')
s1.refine(2)
s2.refine(3)

f = open('ntnu.g2', 'w')
s1.write_g2(f)

# x.write_g2(f)
# y.write_g2(f)
```



## Spline Evaluations:

```
import CurveFactory
from math import pi

c = CurveFactory.circle()
c.start()                    # parametric start of domain, here 0
c.end()                      # parametric end   of domain  here 2*pi

c.evaluate(pi/4)             # evaluate (x,y)-coordinate at t=pi/4
c(pi/4)                      # same as above
c(0)                         # evaluate curve at t=0
c[0]                         # return first control point of curve

c.evaluate_tangent(pi/2)     # evaluate tangent vector at t=pi/2
c.evaluate_derivative(pi/2) # same as above

t = [0,.1,.2,.3,.4,.5,.6,.7,.8,.9,1]
x = c(t) # evaluate at all t points, returns matrix x of size 11x2

c(3*pi)  # =c(pi), defined as a periodic spline. Well defined for all t
```

## Spline Evaluations:

```
from Surface import *
import VolumeFactory
from math import pi

basis = BSplineBasis(3, [0,0,0,1,1,1])
controlpoints = [[0,0], [1,0], [2,0],
                 [0,1], [1,1], [2,1],
                 [0,2], [1,2], [2,2]]
surf = Surface(basis, basis, controlpoints)

surf.start()                    # parametric start of domain, here (0,0)
surf.end()                      # parametric end   of domain  here (1,1)

surf(0.3, 0.4)                  # evaluate surface at (u,v)=(.3,.4)
surf(-0.2, 0.5)                 # evaluate outside domain, creates error
surf[0]                         # returns first control-point [0,0]
surf[3]                         # returns 4th control-point [0,1]

u = [0, .2, .4, .6, .8, 1]
v = [0, .2, .4, .6, .8, 1]
x = surf(u,v) # evaluates at all points, returns 3D tensor of size (6,6,2)
x[1,2,0]      # x-coordinate of surface evaluated at (.2, .4)
x[1,2,1]      # y-coordinate of surface evaluated at (.2, .4)

vol = VolumeFactory.extrude(surf,1) # create a volume from the surface
w = [0, .2, .4, .6, .8, 1]
x = vol(u,v,w) # returns a 4D-tensor of size (6,6,6,2)
x[3,4,5,:]     # (x,y) coordinate of volume evaluated at (.6, .8, 1.0)
```

## Spline Evaluations:

Uses efficient evaluations through numpy tensor products

```
# compute basis functions for all points t. Nu(i,j) is a matrix of all functions j for all points u[i]
Nu = self.basis1.evaluate(u)
Nv = self.basis2.evaluate(v)
Nw = self.basis3.evaluate(w)

# compute physical points [x,y,z] for all points (u[i],v[j],w[k]). For rational volumes, compute [X,Y,Z,W
result = np.tensordot(Nw, self.controlpoints, axes=(1,2))
result = np.tensordot(Nv, result,             axes=(1,2))
result = np.tensordot(Nu, result,             axes=(1,2))

# Project rational volumes down to geometry space: x = X/W, y=Y/W, z=Z/W
if self.rational:
    for i in range(self.dimension):
        result[:,:,:,i] /= result[:,:,:,-1]
```

## Spline Interpolation:

Uses efficient evaluations through numpy tensor products

```
# compute interpolations points
u = self.basis1.greville()
v = self.basis2.greville()
w = self.basis3.greville()

# compute basis function matrices
Nu = self.basis1.evaluate( u )
Nv = self.basis2.evaluate( v )
Nw = self.basis3.evaluate( w )

# solve the interpolation problem
Nu_inv = np.linalg.inv(Nu)
Nv_inv = np.linalg.inv(Nv)
Nw_inv = np.linalg.inv(Nw) # these are inverses of the 1D problems, and small compared to the total numbe
tmp = np.tensordot(Nw_inv, interpolation_pts_x, axes=(1,2))
tmp = np.tensordot(Nv_inv, tmp,                 axes=(1,2))
tmp = np.tensordot(Nu_inv, tmp,                 axes=(1,2))

self.controlpoints = tmp
```

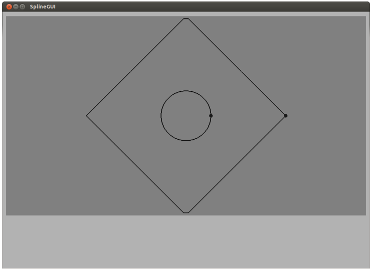
## Affine transformation:

Standard 4x4 matrices for move,rotate,etc

```
def translate(self, x):
# 3D rational example: create a 4x4 translation matrix
#
#  |xw|       |  1   0   0  x1 |    |xw|
#  |yw|   =   |  0   1   0  x2 | *  |yw|
#  |zw|       |  0   0   1  x3 |    |zw|
#  | w|_new   |  0   0   0   1 |    | w|_old
#
dim = self.dimension
rat = self.rational
n   = len(self)  # number of control points

# set up the translation matrix
translation_matrix = np.matrix(np.identity(dim+1))
for i in range(dim):
    translation_matrix[i,-1] = x[i]

# wrap out the controlpoints to a matrix (down from n-D tensor)
cp = np.matrix(np.reshape(self.controlpoints, (n, dim+rat)))

# do the actual scaling by matrix-matrix multiplication
cp  = cp * translation_matrix.T # right-mult, so we need transpose

# store results
self.controlpoints = np.reshape(np.array(cp), self.controlpoints.shape)

return self
```

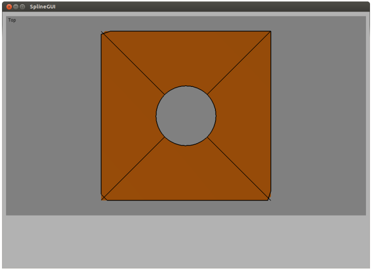
# Flow around a cylinder

```
from math import pi
import CurveFactory   as cf
import SurfaceFactory as sf


circle     = cf.circle(1.0)
boundary   = cf.n_gon(4)
boundary *= 4




f = open('flow-around-cylinder.g2', 'w')
circle.write_g2(f)
boundary.write_g2(f)
```



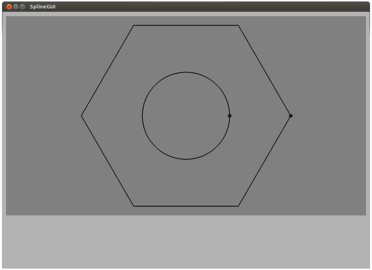
## Flow around a cylinder

```
from math import pi
import CurveFactory   as cf
import SurfaceFactory as sf


circle     = cf.circle(1.0)
boundary   = cf.n_gon(4)
boundary *= 4
surf       = sf.edge_curves([circle, boundary])

surf.rotate(pi/4)

f = open('flow-around-cylinder.g2', 'w')

surf.write_g2(f)
```



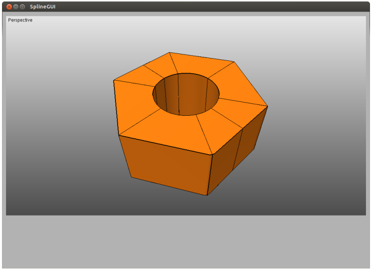
## Building a nut

```
from math import pi
import CurveFactory   as cf
import SurfaceFactory as sf


circle    = cf.circle(1.0)
boundary  = cf.n_gon(6)
boundary *= 2.4




f = open('nut.g2', 'w')
circle.write_g2(f)
boundary.write_g2(f)
```

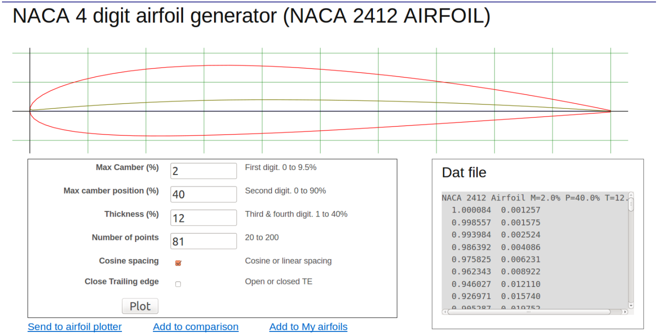


## Building a nut

```
from math import pi
import CurveFactory   as cf
import SurfaceFactory as sf
import VolumeFactory  as vf

circle     = cf.circle(1.0)
boundary   = cf.n_gon(6)
boundary  *= 2.4
surf       = sf.edge_curves([circle, boundary])

vol        = vf.extrude(surf, 2)

f = open('nut.g2', 'w')

vol.write_g2(f)
```



## NACA wing profile




http://airfoiltools.com/airfoil/naca4digit

## NACA wing profile

Center line (camber)

$$y(x) = \begin{cases} \frac{M}{P^2}(2Px - x^2) & 0 \leq x \leq P \\ \frac{M}{(1-P)^2}(1 - 2P + 2Px - x^2) & P \leq x \leq 1 \end{cases}$$

```
from Curve import *
import CurveFactory
import numpy as np

def camber(M,P):
    basis = BSplineBasis(3) # quadratic basis
    basis.insert_knot(P)    # create C1-knot at P

    t = basis.greville()    # interpolation points
    n = len(t)              # number of basis functions (=4)
    x = np.zeros((n,2))
    for i in range(n):
        if t[i] <= P:
            x[i,0] = t[i]
            x[i,1] = M/P/P*(2*P*t[i] - t[i]*t[i])
        else:
            x[i,0] = t[i]
            x[i,1] = M/(1-P)/(1-P)*(1-2*P + 2*P*t[i] - t[i]*t[i])

    return CurveFactory.interpolate(x, basis)
```

**NACA wing profile**

Previous parametrization used

$$
\begin{aligned}
x(t) &= t \\
y(t) &= \begin{cases} \frac{M}{P^2}(2Px - t^2) & 0 \leq t \leq P \\ \frac{M}{(1-P)^2}(1 - 2P + 2Px - t^2) & P \leq t \leq 1 \end{cases}
\end{aligned}
$$

## NACA wing profile

Previous parametrization used

$$
\begin{aligned}
x(t) &= t \\
y(t) &= \begin{cases} \frac{M}{P^2}(2Px - t^2) & 0 \leq t \leq P \\ \frac{M}{(1-P)^2}(1 - 2P + 2Px - t^2) & P \leq t \leq 1 \end{cases}
\end{aligned}
$$

but it is well known that $x(t) = t$ is a suboptimal parametrization. Webpage suggests

$$x(t) = \frac{1}{2}\left(1 - cos(t)\right), 0 \leq t \leq \pi$$

**NACA wing profile**

Previous parametrization used

$$\begin{aligned} x(t) &= t \\ y(t) &= \begin{cases} \frac{M}{P^2}(2Px - t^2) & 0 \le t \le P \\ \frac{M}{(1-P)^2}(1 - 2P + 2Px - t^2) & P \le t \le 1 \end{cases} \end{aligned}$$

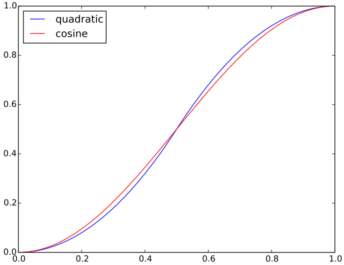
but it is well known that $x(t) = t$ is a suboptimal parametrization. Webpage suggests

$$x(t) = \frac{1}{2}(1 - cos(t)), 0 \le t \le \pi$$

but might as well choose a piecewise polynomial

$$x(t) = \begin{cases} 2t^2 & 0 \le t \le \frac{1}{2} \\ -2(t^2 - 2t + \frac{1}{2}) & \frac{1}{2} \le t \le 1 \end{cases}$$

## NACA wing profile




$$x(t) = \frac{1}{2}\left(1 - cos(t)\right), 0 \leq t \leq \pi$$

$$x(t) = \begin{cases} 2t^2 & 0 \leq t \leq \frac{1}{2} \\ -2(t^2 - 2t + \frac{1}{2}) & \frac{1}{2} \leq t \leq 1 \end{cases}$$

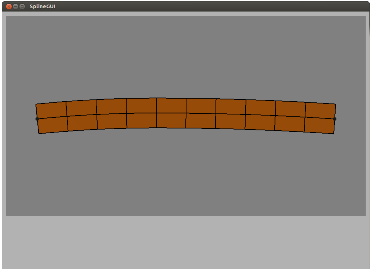
## NACA wing profile

```
from Curve import *
import CurveFactory
import numpy as np

def camber(M,P):
    basis = BSplineBasis(5)        # p=4 basis
    basis.insert_knot([P,P,P])     # create C1-knot at P for y-parametrization
    basis.insert_knot([.5,.5,.5]) # create C1-knot at 0.5 for x-parametrization

    t = basis.greville()     # interpolation points
    n = len(t)               # number of basis functions
    x = np.zeros((n,2))
    for i in range(n):
        if t[i] <= 0.5:
            x[i,0] = t[i]**2 / P
        else:
            x[i,0] = -2*(t[i]**2-2*t[i]+.5)
        if t[i] <= P:
            x[i,1] = M/P/P*(2*P*x[i,0] - x[i,0]*x[i,0])
        else:
            x[i,1] = M/(1-P)/(1-P)*(1-2*P + 2*P*x[i,0] - x[i,0]*x[i,0])

    return CurveFactory.interpolate(x, basis)
```

## NACA wing profile

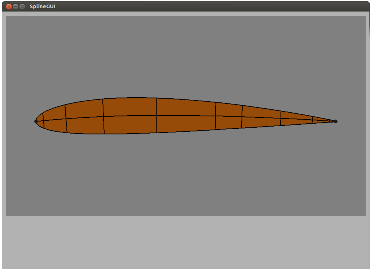
```
from SurfaceFactory import *
from math import *

c = camber(2,4)
```

```
c.insert_knot([.1, .2, .3, .6, .7, .8, .9])
s = thicken(c, .05)

f = open('naca.g2', 'w')
s.write_g2(f)
c.write_g2(f)
```



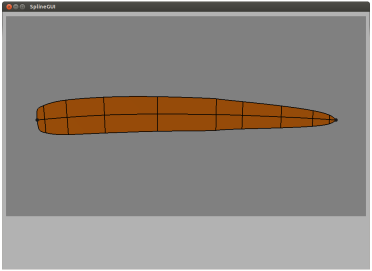
## NACA wing profile

```
from SurfaceFactory import *
from math import *

c = camber(2,4)
def thickness(x):
    T  = 0.12
    a0 = 0.2969
    a1 = -0.126
    a2 = -0.3516
    a3 = 0.2843
    a4 = -0.1015
    return T/0.2*(a0*sqrt(x) + a1*x + a2*x**2 + a3*x**3

c.insert_knot([.1, .2, .3, .6, .7, .8, .9])
s = thicken(c, thickness)

f = open('naca.g2', 'w')
s.write_g2(f)
c.write_g2(f)
```



## NACA wing profile

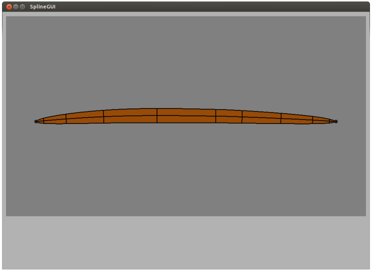
```
from SurfaceFactory import *
from math import *

c = camber(2,4)
def thickness(t):
    T  = 0.12
    a0 = 0.2969
    a1 = -0.126
    a2 = -0.3516
    a3 = 0.2843
    a4 = -0.1015
    return T/0.2*(a0*sqrt(t) + a1*t + a2*t**2 + a3*t**3

c.insert_knot([.1, .2, .3, .6, .7, .8, .9])
s = thicken(c, thickness)

f = open('naca.g2', 'w')
s.write_g2(f)
c.write_g2(f)
```



# NACA wing profile

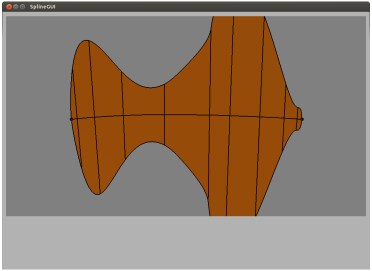
## NACA wing profile

```
from SurfaceFactory import *
from math import *

c = camber(2,4)
def thickness(y):
    T  = 0.12
    a0 = 0.2969
    a1 = -0.126
    a2 = -0.3516
    a3 = 0.2843
    a4 = -0.1015
    return T/0.2*(a0*sqrt(y) + a1*y + a2*y**2 + a3*y**3

c.insert_knot([.1, .2, .3, .6, .7, .8, .9])
s = thicken(c, thickness)

f = open('naca.g2', 'w')
s.write_g2(f)
c.write_g2(f)
```



## NACA wing profile

```
from SurfaceFactory import *
from math import *

c = camber(2,4)
def thickness(t):
    return t*(1-t)*(sin(4*pi*t)+1.5)
```




```
c.insert_knot([.1, .2, .3, .6, .7, .8, .9])
s = thicken(c, thickness)

f = open('naca.g2', 'w')
s.write_g2(f)
c.write_g2(f)
```

**What it doesn't do**

- No point & click user interface
- No automatic topology solutions for multi-patch problems
- No boolean operations (union $A \bigcup B$ or intersection $A \bigcap B$)
- No trimming

### What it does

- Spline evaluations (and derivatives, tangents, normals)

## What it does

- Spline evaluations (and derivatives, tangents, normals)
- Elementary operations (move, scale, rotate, etc)

## What it does

- Spline evaluations (and derivatives, tangents, normals)
- Elementary operations (move, scale, rotate, etc)
- Spline operations (extrude, loft, revolve, sweep)

## What it does

- Spline evaluations (and derivatives, tangents, normals)
- Elementary operations (move, scale, rotate, etc)
- Spline operations (extrude, loft, revolve, sweep)
- Volumetric meshing

## What it does

- Spline evaluations (and derivatives, tangents, normals)
- Elementary operations (move, scale, rotate, etc)
- Spline operations (extrude, loft, revolve, sweep)
- Volumetric meshing
- Interpolations and approximations

## What it does

- Spline evaluations (and derivatives, tangents, normals)
- Elementary operations (move, scale, rotate, etc)
- Spline operations (extrude, loft, revolve, sweep)
- Volumetric meshing
- Interpolations and approximations

and all of the above works for

- Rational splines
- Periodic splines
- 2D splines

## Design choices

- Stable!
  - lots of testing
  - state assumptions where used

## Design choices

- Stable!
  - lots of testing
  - state assumptions where used
- Readable
  - lots of documentation
  - lots of code comments

## Design choices

- Stable!
  - lots of testing
  - state assumptions where used
- Readable
  - lots of documentation
  - lots of code comments
- Don't have to be perfect
  - make it working first
  - optimize *if needed*

## Design choices

- Stable!
  - lots of testing
  - state assumptions where used
- Readable
  - lots of documentation
  - lots of code comments
- Don't have to be perfect
  - make it working first
  - optimize *if needed*
  - **working** trumphs **optimized**

## Design choices

- Stable!
  - lots of testing
  - state assumptions where used
- Readable
  - lots of documentation
  - lots of code comments
- Don't have to be perfect
  - make it working first
  - optimize *if needed*
  - **working** trumphs **optimized**
- Easy to learn...
  - Simple interface to create simple geometries

## Design choices

- Stable!
  - lots of testing
  - state assumptions where used
- Readable
  - lots of documentation
  - lots of code comments
- Don't have to be perfect
  - make it working first
  - optimize *if needed*
  - **working** trumphs **optimized**
- Easy to learn...
  - Simple interface to create simple geometries
- ...hard to master
  - Visibility and control to create complex geometries