# FEEG1001: Applied Computing for Aero and Mechanical Engineering

Alexander I J Forrester

## Contents

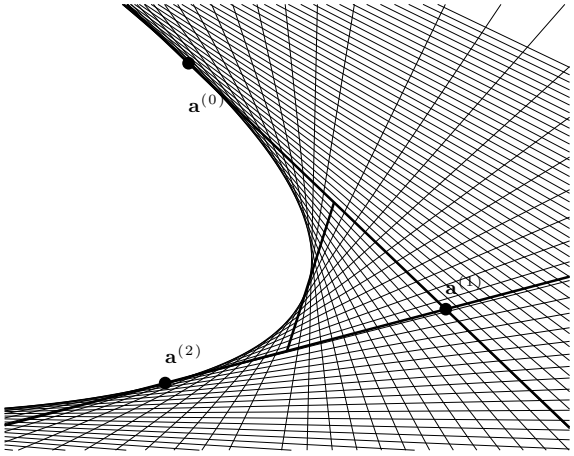


Figure 1: a conic as the intersections of projectivities between two lines.

# 1 Computational Definition of Curves

## 1.1 Lecture 1: Conics and Bézier curves

The classical, and initially intuitive, definition of conics is that of a plane intersecting a cone, with the angle of intersection determining the shape. Here we hope to give more insight and intuition, particularly when considering parametric conics and then on to NURBS, by using the projective geometry definition. We will only cover the essentials of projective geometry

Figure 1 shows a conic as the intersection of lines. The conic is defined by two tangent lines, here through $\mathbf{a}^{(0)}, \mathbf{a}^{(1)}$ and $\mathbf{a}^{(2)}, \mathbf{a}^{(1)}$ (but could easily be any two of the lines shown), and a third *shoulder tangent* (shown in bold). Described in this way, the conic is a mapping or *projectivity* between the two tangent lines, with this projectivity defining the shoulder tangent. The following sections cover this projective geometry construction of conics, before moving on to Bézier cures, splines, B-splines and NURBS. We will skip some of the intricacies of projective geometry and the reader wanting more on in this area may wish to refer to [Farin, 1999].

Studies on conics date back to Menaechmus (380-320 BC) [Thomas, 1939]. Conics are found in astronomy, palaeontology, aerodynamic design (see figure 2) where P-51 designer Edgar Schmued called second degree conics "the shape the air likes to touch" and, indeed, many areas of everyday life (see figure 3).

### 1.1.1 Parametric Bernstein Conic

To define a projectivity, and so a conic, we require three pre-image and image point pairs. Figure 4 is a reproduction of figure 1, showing these three pairs: $\mathbf{a}^{(0)} \rightarrow \mathbf{a}^{(1)}$, $\mathbf{a}^{(1)} \rightarrow \mathbf{a}^{(2)}$, and $\mathbf{s}^{(1)} \rightarrow \mathbf{s}^{(2)}$. Point $\mathbf{a}^{(1)}$ is the intersection of $\mathbf{A}$ and

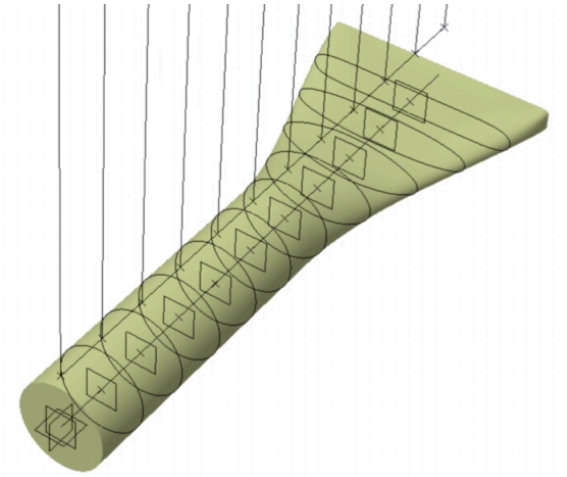

Figure 2: a conic-based jet nozzle design for an unmanned air vehicle (provided by András Sóbester).

Figure 3: a BMX ramp with a conic profile marked-out by hand using the same construction by projectivities as in figure 1.

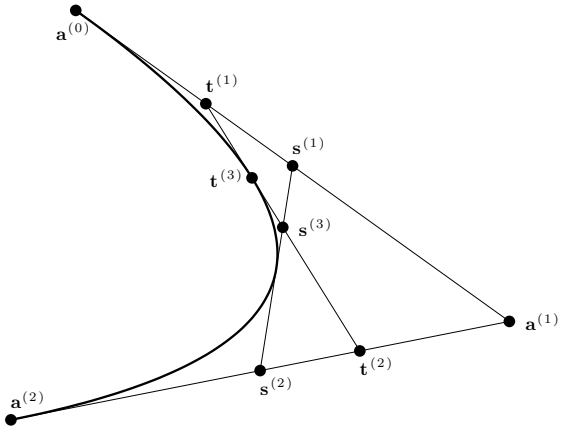


Figure 4: Construction of a Bernstein conic.

$\mathbf{B}$, and $\mathbf{S} = \mathbf{s}^{(1)} \times \mathbf{s}^{(2)}$ is the *shoulder tangent*. We will position this shoulder tangent at

$$\mathbf{s}^{(1)} = \frac{1}{2}\mathbf{a}^{(0)} + \frac{1}{2}\mathbf{a}^{(1)} \tag{1}$$

and

$$\mathbf{s}^{(2)} = \frac{1}{2}\mathbf{a}^{(1)} + \frac{1}{2}\mathbf{a}^{(2)}. \tag{2}$$

A point on the conic, which we will call $\mathcal{C}$, in figure 4 shall be found as the intersection of a tangent, $\mathbf{T}(t)$, with the conic ($t$ is a parameter that we vary so that $\mathbf{t}^{(3)}$ (the intersection of $\mathbf{T}$ with the conic) traces out, i.e. moves along the along the conic). The intersection of $\mathbf{T}$ with $\mathbf{A}$ is

$$\mathbf{t}^{(1)} = (1-t)\mathbf{a}^{(0)} + t\mathbf{a}^{(1)}, \tag{3}$$

and with $\mathbf{B}$ is

$$\mathbf{t}^{(2)} = (1-t)\mathbf{a}^{(1)} + t\mathbf{a}^{(2)}. \tag{4}$$

This intuitive result is derived in Farin [1999] using the *four tangent theorem* (which is also employed below). Thus as $t$ varies from 0 to 1, the intersection $\mathbf{t}^{(3)}$ traces out the conic from $\mathbf{a}^{(0)}$ to $\mathbf{a}^{(2)}$ (the whole conic is traced out by $-\infty < t < \infty$).

Looking at figure 4, by bringing $\mathbf{t}^{(1)}$ to $\mathbf{a}^{(1)}$, i.e. setting $t = 0$, we see that the intersection of $\mathbf{S}$ and $\mathbf{T}$,

$$\mathbf{s}^{(3)} = \frac{1}{2}\mathbf{t}^{(1)} + \frac{1}{2}\mathbf{t}^{(2)}. \tag{5}$$

We now have enough information to find any point on the conic $\mathbf{t}^{(3)}(t)$, but will need to invoke the four tangent theorem. Four tangents to a conic (e.g. **A**, **B**, **S** and **T** in figure 4) each have an intersection with the conic and three intersections with the other three tangents. The four tangent theorem states that the cross ratio of these four intersections equals the same constant for all four tangents.

In general, given four points on a line, such that

$$\mathbf{a}^{(3)} = \alpha_1 \mathbf{a}^{(1)} + \beta_1 \mathbf{a}^{(2)}$$

and

$$\mathbf{a}^{(4)} = \alpha_2 \mathbf{a}^{(1)} + \beta_2 \mathbf{a}^{(2)},$$

the cross ratio of these four points is here defined as

$$\mathrm{cr}(\mathbf{a}^{(1)}, \mathbf{a}^{(3)}, \mathbf{a}^{(4)}, \mathbf{a}^{(2)}) = \frac{\beta_1}{\alpha_1}\frac{\alpha_2}{\beta_2}. \tag{6}$$

Thus, from 6, 1 and 3, the cross ratio for the intersections of **A**

$$\mathrm{cr}(\mathbf{a}^{(0)}, \mathbf{t}^{(1)}, \mathbf{s}^{(1)}, \mathbf{a}^{(1)}) = \frac{1/2}{1/2}\frac{t}{1-t} = \frac{t}{1-t} \tag{7}$$

and from the four tangent theorem

$$\mathrm{cr}(\mathbf{a}^{(0)}, \mathbf{t}^{(1)}, \mathbf{s}^{(1)}, \mathbf{a}^{(1)}) = \mathrm{cr}(\mathbf{t}^{(1)}, \mathbf{t}^{(3)}, \mathbf{s}^{(3)}, \mathbf{t}^{(2)}). \tag{8}$$

Using 6 and 8,

$$\mathrm{cr}(\mathbf{t}^{(1)}, \mathbf{t}^{(3)}, \mathbf{s}^{(3)}, \mathbf{t}^{(2)}) = \frac{1/2}{1/2}\frac{t}{1-t}. \tag{9}$$

And so:

$$\mathbf{t}^{(3)}(t) = (1-t)\mathbf{t}^{(1)}(t) + t\mathbf{t}^{(2)}(t). \tag{10}$$

Substituting 3 and 4 into 10 , we find that the point on the conic, $\mathcal{C}(t)$ is a quadratic with coefficients based on our original points in figure 4:

$$\mathcal{C}(t) = \mathbf{t}^{(3)}(t) = (1-t)^2\mathbf{a}^{(0)} + 2t(1-t)\mathbf{a}^{(1)} + t^2\mathbf{a}^{(2)}. \tag{11}$$

With the quadratic Bernstein basis polynomials defined as

$$b_{i,2}(t) = \begin{pmatrix} 2 \\ i \end{pmatrix} t^i (1-t)^{2-i}, \quad i = 0, 1, 2, \tag{12}$$

where the binomial coefficient

$$\begin{pmatrix} 2 \\ i \end{pmatrix} = \frac{2!}{i!(2-i)!},$$

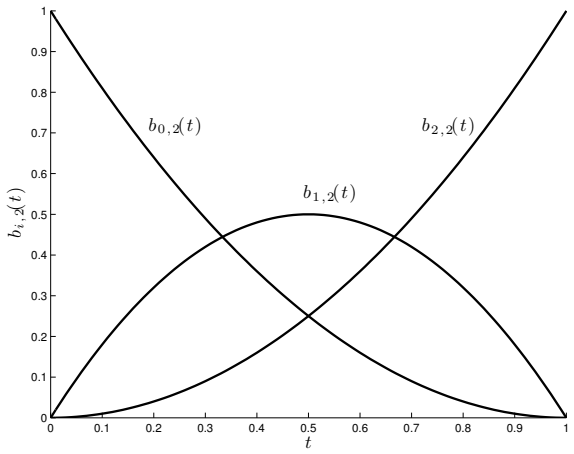


Figure 5: The Bernstein polynomials (equation 12), displaying the intuitive property of Bézier curve control point weighting falling away from control points.

equation 11 can be re-written as a projective quadratic Bézier curve:

$$\mathcal{B}(t) = \mathcal{C}(t) = \mathbf{t}^{(3)}(t) = \sum_{i=0}^{2} \mathbf{a}^{(i)} b_{i,2}(t). \tag{13}$$

We see that the Bézier curve is a weighted sum of the *control points*, $\mathbf{a}^{(i)}$. The influence of these control points, which is determined by their weighting $b_{i,2}(t)$, should be greatest when the curve passes closest to them and this influence should diminish as the curve moves away. For example, $b_{i,2}(t)$ should reach a maximum when $t = 0.5$ and $b_{i,2}(t) \to 0$ as $t \to \pm\infty$. Figure 5 shows the Bernstein polyniomial weightings (equation 12) used by the Bézier curve.

It is possible to add further control points to the sum in equation 13, weighted using higher degree Bernstein polynomials:

$$b_{i,n}(t) = \left( \begin{array}{c} n \\ i \end{array} \right) t^i (1-t)^{n-i}, \quad \text{where} \quad \left( \begin{array}{c} n \\ i \end{array} \right) = \frac{n!}{i!(n-i)!}, \tag{14}$$

which are shown in figure 6. Figure 7 demonstrates this by inserting a control point in to the definition of the quadratic Bézier curve in figure 4 to produce a cubic Bézier curve:

$$\begin{aligned} \mathcal{B}(t) &= \sum_{i=0}^{3} \mathbf{a}^{(i)} b_{i,3}(t) \\ &= \sum_{i=0}^{3} \mathbf{a}^{(i)} \left( \begin{array}{c} 3 \\ i \end{array} \right) t^i (1-t)^{3-i} \\ &= (1-t)^3 \mathbf{a}^{(0)} + 3t(1-t)^2 \mathbf{a}^{(1)} + 3t^2(1-t)\mathbf{a}^{(2)} + t^3 \mathbf{a}^{(3)}. \end{aligned} \tag{15}$$

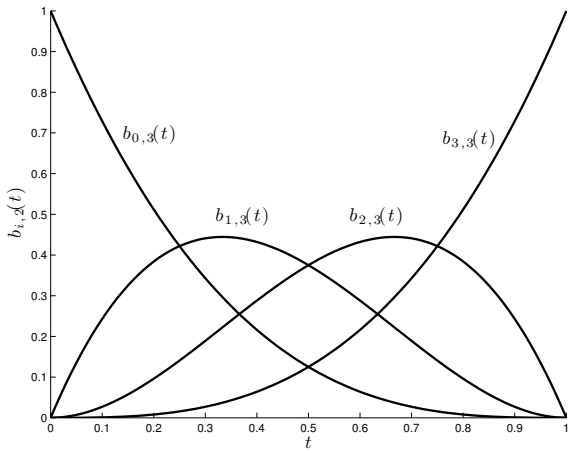


Figure 6: The Bernstein polynomials (equation 12), displaying the intuitive property of Bézier curve control point weighting falling away from control points.

The *Python* code to produce this curve is included in listing 1.

Listing 1: Sample Python code to calculate and plot the cubic Bézier curve in figure 4

```
import pylab as p
import numpy as np
# define control points
a=np.array([[1, 10], [2, 5], [7.6863, 3.9216], [0, 2]])
# calculate projective cubic bezier for t [0 ,1]
t=np.linspace(0, 1, 101)
cubicBezier=np.zeros([101, 2])
for i in range(0, 101):
    cubicBezier[i,:] = ((1 - t[i]) ** 3 * a[0, :] + \
       3 * t[i] * (1 - t[i]) ** 2 * a[1, :] + \
       3 * t[i] ** 2 * (1 - t[i]) * a[2, :] + \
       t[i] ** 3 * a[3, :])
# plot cubic Bezier
p.plot(cubicBezier[:, 0], cubicBezier[:, 1])
# plot control points
p.plot(a[:, 0], a[:, 1], 'ko')
# plot control polygon
p.plot(a[:, 0], a[:, 1], 'k')
p.show()
```

This is a rather naïve implementation, which only works with four control points. A generic Bézier curve could be produce by using the function for the binomial coefficient in listing 2.

Listing 2: *Python* code to calculate the binomial coefficient

```
def binom(n, i):
    if 0 <= i <= n:
```

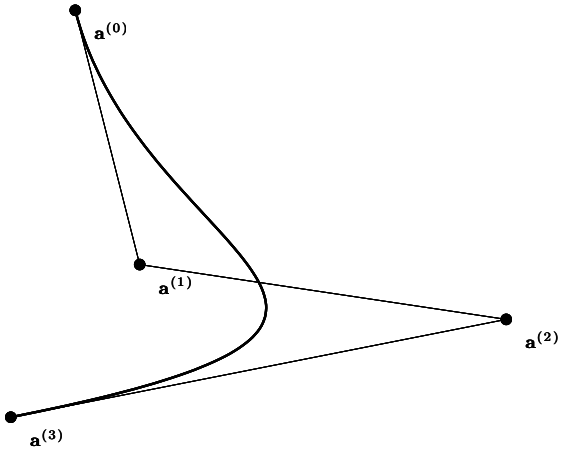


Figure 7: A cubic Bézier curve produced by inserting an additional control point to the quadratic Bézier curve in figure 4 (the *Python* code to calculate this curve is in listing 1).

```
        p = 1
        for t in xrange(min(i, n - i)):
            p = (p * (n - t)) // (t + 1)
        return p
    else:
        return 0
```

## 1.2 Lecture 2: rational conics, Bézier curves and splines

While the use of projective geometry to define conics is elegant and intuitive, it is unsuitable for design purposes. We can project onto the the plane $z = 1$ (an affine plane), simply by dividing the projective points through by the $z$-component (the figures in the previous sections are in this plane):

$$\mathbf{a} = \begin{pmatrix} a_x \\ a_y \end{pmatrix} = \frac{\mathbf{a}^{I\!P}}{\mathbf{a}_z^{I\!P}} = \begin{pmatrix} a_x^{I\!P}/a_z^{I\!P} \\ a_y^{I\!P}/a_z^{I\!P} \\ a_z^{I\!P}/a_z^{I\!P} \end{pmatrix},$$

where here we use $I\!P$ to denote a point in projective space. Equation 11 can now be written as a rational form as

$$\mathcal{C}(t) = \frac{(1-t)^2 z^{(0)}\mathbf{a}^{(0)} + 2t(1-t)z^{(1)}\mathbf{a}^{(1)} + t^2 z^{(2)}\mathbf{a}^{(2)}}{(1-t)^2 z^{(0)} + 2t(1-t)z^{(1)} + t^2 z^{(2)}}, \tag{16}$$

(and also for higher degree conics) where, e.g., $\mathbf{a}_z^{I\!P(0)}$ is written simply as $z^{(0)}$. These $z^{(i)}$s are known as *weights* and the $\mathbf{a}^{(i)}$s are the control points.

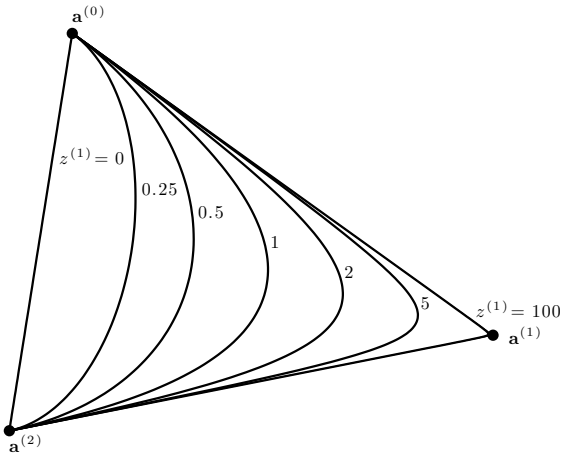


Figure 8: Varying weights for a rational quadratic conic.

In a similar way, equation 13 can now be cast as a rational Bézier curve: 11 can be re-written as a projective quadratic Bézier curve:

$$\mathcal{B}(t) = \frac{\sum_{i=0}^{n} w_i \mathbf{a}^{(i)} b_{i,n}(t)}{\sum_{i=0}^{n} w_i b_{i,n}(t)}, \tag{17}$$

where $w_i$ are the weights. Figure 4 has all weights set at unity. Figure 8 shows the effect of varying $z^{(1)}$, with $z^{(0)}, z^{(2)} = 1$. The effect is intuitive; for $z^{(1)} = 0$, $\mathcal{C}$ becomes a straight line from $\mathbf{a}^{(0)}$ to $\mathbf{a}^{(2)}$ and as $z^{(1)} \to \infty$, $\mathcal{C}(t)$ tends to the line $\mathbf{a}^{(0)}, \mathbf{a}^{(1)}, \mathbf{a}^{(2)}$.

For the remainder of this chapter we will consider rational curves defined by $x, y$ coordinates of points and weightings.

### 1.2.1 Properties of Bézier curves

The above derivation of a conic and so Bézier curve has lead to an intuitive form of curve, which can be 'pushed and pulled' around by control points. Salomon [2006] lists a number of properties of Bézier curves (and discusses them in more detail than we shall do here), which help in understanding their usefulness in geometry definition and manipulation:

- the weights, $b_{i,n}(t)$ add up to 1 (they are *barycentric*),
- the curve passes through the two endpoints $\mathbf{a}^{(0)}$ and $\mathbf{a}^{(n)}$,
- the curve is symmetric with respect to the numbering of control points, i.e. the same result will be obtained by setting $\mathbf{a}^{(0)} = \mathbf{a}^{(n)}, \mathbf{a}^{(1)} = \mathbf{a}^{(n-1)}, ..., \mathbf{a}^{(n)} = \mathbf{a}^{(0)}$,

- the first derivative can be obtained as

$$\dot{\mathbf{a}}(t) = n \sum_{0}^{n-1} \Delta\mathbf{a}^{(i)} b_{i,n-1}(t), \quad \text{where} \quad \Delta\mathbf{a}^{(i)} = \mathbf{a}^{(i+1)} - \mathbf{a}^{(i)}, \tag{18}$$

- the weight functions $b_{i,n}$ have a maximum at $t = i/n$, that is the influence of the control points is spaced evenly with respect to $t$,
- the two end tangents, derived from 18 as $\dot{\mathbf{a}}(0) = n(\mathbf{a}^{(1)} - \mathbf{a}^{(0)})$ and $\dot{\mathbf{a}}(1) = n(\mathbf{a}^{(n)} - \mathbf{a}^{(n-1)})$, are easily controlled by moving $\mathbf{a}^{(1)}$ and $\mathbf{a}^{(n-1)}$ respectively,
- the Bézier curve is controlled globally by editing control point(s) and weighting(s), that is changing one control point will affect the entire curve (note how in figures 5 and 6 the weighting of each point has an influence over all values of $t$), with the affect greatest close to the control point,
- the curve is contained within the control polygon (within its *convex hull*), as seen in figure 7, where the curve lies within the triangle formed by $\mathbf{a}^{(0)}, \mathbf{a}^{(1)}, \mathbf{a}^{(2)}$ and then that formed by $\mathbf{a}^{(1)}, \mathbf{a}^{(2)}, \mathbf{a}^{(3)}$, and
- the curve is invariant to affine transformations, e.g. can be shifted up/-down, left/right, rotated, reflected (but is not invariant under projections).

### 1.2.2 Splines

A series of conics or Bézier curves can be concatenated to produce a more complex curve. The process is quite straightforward. For two successive curves of degree $n$ and $m$, defined by control points $\mathbf{p}^{(i)}, i = 0, 1, ..n$ and $\mathbf{q}^{(i)}, i = 0, 1, ..m$ to meet, the end and start points must coincide, i.e. $\mathbf{p}^{(n)} = \mathbf{q}^{(0)}$. For a smooth connection, $\dot{\mathbf{p}}^{(n)} = \dot{\mathbf{q}}^{(0)}$. Thus from (18) and the end tangent note above,

$$\mathbf{p}^{(n)} = \mathbf{q}^{(0)} = \frac{m}{m+n}\mathbf{q}^{(1)} + \frac{n}{m+n}\mathbf{p}^{(n-1)}. \tag{19}$$

Note that we do not need to specify the 1st and nth points, as these are found from the points either side of the connection. Figure 9 shows the concatenation of two quadratic Bézier curves to form a Bézier spline, while figure 10 shows two cubic Bézier curves joined into a Bézier spline representation of an aerofoil. The *Python* code to produce the spline in this figure is included below. By changing six parameters: $\mathbf{p}_y^{(2)}, \mathbf{q}_y^{(1)}, \mathbf{p}_{x\&y}^{(1)}, \mathbf{q}_{x\&y}^{(2)}$, a good degree of shape control is possible. Further control can be had by manipulating the weights, $z^{(i)}$, and we will consider this later.

## 1.3 Lab 1: create a Bézier spline using *Python*

Create a file `aclab1.py` that contains the functions listed below.

- The `binom` function in listing 2.

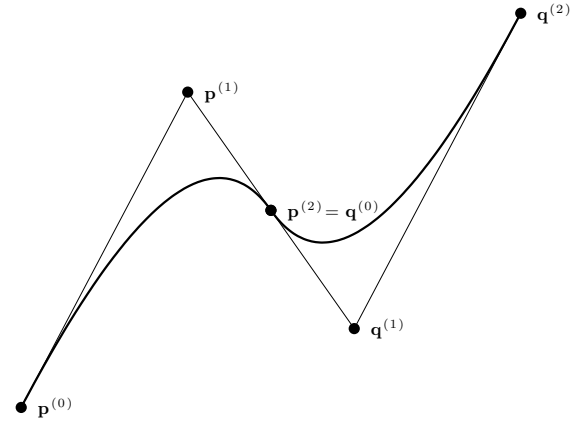


Figure 9: an example of a Bézier spline formed by joining two quadratic Bézier curves.

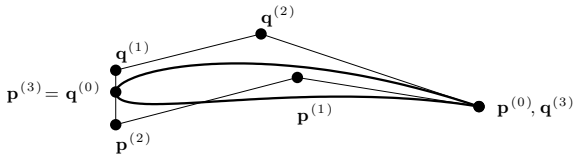


Figure 10: an example of a Bézier spline representation of an aerofoil, formed by joining two cubic Bézier curves.

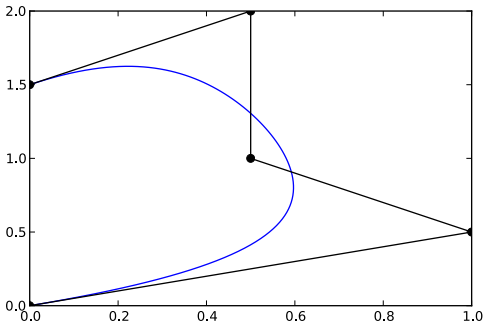


Figure 11: Example output from `bezier(points)`

- A function `bezier(points)` that creates a cubic Bézier curve with control points defined by an $n \times 2$ array `points`, returns a $101 \times 2$ array of $x, y$ points on the curve for $t[0, 1]$ and plots the curve, control points and control polygon.

  Example:

```
>>> points=array([[0, 0], [1, 0.5], [0.5, 1], [0.5, 2], [0, 1.5]])
>>> print bezier(points)

[[0. 0. ] [ 0.03910797 0.02000198]
...
[ 0.01970397 1.51910798] [0. 1.5 ]]
```

  You can tackle this code writing problem by modifying the code in listing 1, which implements the last line of equation 15. Instead of explicitly coding this cubic Bézier equation, you first need to code it as a sum of four terms using the `binom` function, i.e. as in the second line of equation 15. Use a `for` loop followed by `sum` to do this. Next make you code handle any number of control points by using `n+1` instead of four terms. Finally, make `points` an input to the function, rather than have the control points hard-coded as in listing1 (where the array `a` is the control points).

- A function `rational_bezier(points, weights)` that creates a rational cubic Bézier, with $n$ control points with $n$ weights in the input arrays `points` and `weights`, returns a $101 \times 2$ array of $x, y$ points on the curve for $t[0, 1]$ and plots the curve, control points and control polygon. Example:

```
>>> points = array([[0, 0], [1, 0.5], [0.5, 1], [0.5, 2], [0, 1.5]])
>>> weights = array([1, 10, 5, 15, 1])
>>> print rational_bezier(points, weights)
```

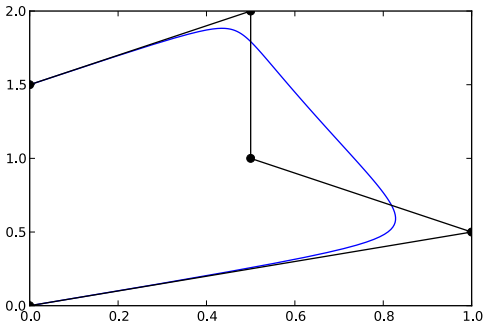


Figure 12: Example output from `rational_bezier(points, weights)`

```
[[ 0.          0.        ]
 [ 0.28824076  0.14582872]
...
 [ 0.18929222  1.68733879]
 [ 0.          1.5       ]]
```

This function only requires minor modifications to your `bezier(points)` code. Refer to equations 16 and 17.

# 2 Computational analysis of geometry

## 2.1 Lecture 3: *XFOIL* (an example of computational analysis)

*XFOIL* is an interactive program for the design and analysis of subsonic isolated airfoils [Drela, 1989]. It combines the speed and accuracy of high-order panel methods with a fully-coupled viscous boundary layer method [Drela and Giles, 1987]. Here we will consider *XFOIL* as a black-box analysis method and use it to understand geometry manipulation, linking computer codes, and optimisation, i.e. we will *use* it, but not try to *understand* it (that would be part of a aerodynamics and computational fluid dynamics module).

*XFOIL* can be found under "Your School Software". Selecting `XfoilP4` opens the DOS command prompt and the program can be run interactively using a set of commands detailed in the user guide. A sample *XFOIL* session can be found at http://web.mit.edu/drela/Public/web/xfoil/. Listing 3 is a sample session that runs a viscous analysis of an aerofoil defined in `aerofoil.dat`. The inputs work, line-by-line as follows:

- load aerofoil coordinates form the `aerofoil.dat file`,
- name this aerofoil `CubicBez`,

- smooth the aerofoil using the `panel` command,
- enter the analysis routine using the `oper` command,
- define a viscous analysis at a Reynold's number of $1.4 \times 10^6$,
- set the Mach number to 0.1,
- `type 1` means that the Mach number, Reynolds number, aerofoil chord and velocity will be fixed and the lift will vary (other types of analysis are described in the *XFOIL* user guide, section 5.7),
- store data in a file using the `pacc` command (short for polar accumulate),
- set `polar.dat` as the file in which to store data,
- return to ignore a prompt at this point,
- iterate the analysis 1000 times,
- set a target lift coefficient for the analysis of 1.2 (i.e. the angle of attack will be varied automatically to obtain this lift coefficient),
- return to come out of `opera` menu, and
- quit XFOIL.

Instead of entering these commands directly into the *XFOIL* prompt, the program can be run in batch by saving the commands in listing 3 in a file called, say, `commands.in` and typing `XFOIL_PATH\XfoilP4 < YOUR_PATH\commands.in` at the DOS prompt (replace '\' with '/' for LINUX/Mac operating systems). Depending on the operating syste,, *XFOIL* requires a slightly different input file format: listing 3 is for Windows and listing 4 is for OSX.

Listing 3: Sample Xfoil input file

```
load C:\Users\aijf197\Documents\python\aerofoil.dat
CubicBez
panel
oper
visc 1397535
M 0.1
type 1
pacc
C:\Users\aijf197\Documents\python\polar.dat

iter
5000
cl 1.2


quit
```

Listing 4: Sample Xfoil input file for *OSX*

```
load /Users/alex/Dropbox/Teaching/PiNotebook/aerofoil.dat
CubicBez
panel
oper
visc 1397535
M 0.1
type 1
pacc
/Users/alex/Dropbox/Teaching/PiNotebook/polar.dat

iter 5000
cl 1.2


quit
```

Listing 5 is an example of the `polar.dat` output file. The format of this file is always the same, which is useful to know if you are going to interrogate it for data. If the output file entered in *XFOIL* already exists, the `pacc` command in listing 3 will fail. It is therefore best to use a new output file name or delete the existing output file before *XFOIL* is run again.

Listing 5: sample Xfoil output file.

```
       XFOIL           Version 6.97

 Calculated polar for: CubicBez

 1 1 Reynolds number fixed          Mach number fixed

 xtrf =   1.000 (top)        1.000 (bottom)
 Mach =   0.100     Re =     1.398 e 6     Ncrit =   9.000

   alpha     CL        CD        CDp       CM      Top_Xtr   Bot_Xtr
  ------  --------  --------  --------  --------  --------  --------
   3.699    1.2000   0.00745   0.00290  -0.1629    0.4208    1.0000
```

### 2.1.1 Creating the `.dat` and `.in` files using *Python*

The `aerofoil.dat` file contains the $x$,$y$ coordinates of the aerofoil in two columns ($x$ and $y$), with the coordinates starting at the trailing edge, running along the lower surface to the leading edge, and back along the upper surface to the trailing edge. Assuming the coordinates are stored in the (101×1) arrays `lower` and `upper`, we can write these aerofoil coordinates in two ways: line-by-line using the `write` function (listing 6) or, more succinctly, using the `savetxt` function (listing 7). Note that in both cases the last element of the `lower` array is omitted so that the leading edge point is not duplicated.

Listing 6: line-by-line writing of aerofoil coordinates.

```
data_file = open(file_path + 'aerofoil.dat', 'w')
```

```
for i in range(0, 100):
    data_file.write("%f %f\r" %(lower[i, 0], lower[i, 1]))
#LINUX/OSX: data_file.write("%f\t%f\n" %(lower[i, 0], lower[i, 1]))
for i in range(0,101):
    data_file.write("%f %f\r" %(upper[i, 0], upper[i, 1]))
#LINUX/OSX: data_file.write("%f\t%f\n" %(upper[i, 0], upper[i, 1]))
data_file.close()
```

Listing 7: writing aerofoil coordinates with `savetxt`.

```
savetxt(file_path + 'aerofoil.dat', vstack([lower[0:-1], upper]))
```

While with the `aerofoil.dat` file we are writing columns of numbers from which the shape of the aerofoil is not immediately apparent, the input file is 'readable' (at least to the trained eye) and we wish to preserve this in the *Python* code via which we write it. In listing 8 the `write` function is used in a format where the structure of the resulting file is apparent and the variable names, e.g. `Re` for Reynolds number, are logical. Writing code in this way is not particularly succinct, but is often preferable from a development perspective.

Listing 8: writing the `commands.in` file in *Python* in an easily identifiable format.

```
command_file=open(file_path + 'commands.in','w')
command_file.write('load ' + file_path + 'aerofoil.dat\n\
CubicBez\n\
panel\n\
oper\n\
visc ' + str(Re) + '\n\
M ' + str(M) + '\n\
type 1\n\
pacc\n'\
+ file_path + 'polar.dat\n\
\n\
iter\n 1000\n\
cl 1.2\n\
\n\
\n\
quit\n')
command_file.close()
```

### 2.1.2 Running *XFOIL* from *Python*

System calls can be made from *Python* after importing `os` using `os.system()`. For example, assuming `XfoilP4` is in `C:\apps\XFOIL\` and `commands.in` and `aerofoil.dat` are in `C:\Users\aijf197\Documents\`, an output similar to listing 5 could be produced with the code in listing 9. Note the use of double backslashes in the file paths. This is because \ is a special character in *Python* and needs to be escaped by adding the extra backslash. *XFOIL* needs to be able to write to the directory defined by your file path. This might not be possible for remote/shared drives, so it may be necessary to work in a local directory, e.g. **use the 'Documents' folder and not 'My Documents'**. However,

**make sure you copy your work across to 'My Documents' or back-up in some other way before you logout**. Note that to run *XFOIL* using `os.system()` on one of the University workstations, you need to first open and close *XFOIL* from the start menu (you only need to do this once each time you login to a new machine).

Listing 9: sample *Python* code to run *XFOIL*

```
import os
file_path = 'C:\\Users\\aijf197\\Documents\\'
xfoil_path = 'C:\\apps\\XFOIL\\'
run_xfoil_command = xfoil_path + 'XFOIL < ' + file_path + \
 'commands.in'
os.system(run_xfoil_command)
```

To avoid worrying about operating systems and entering \ or /, the `os.path.sep` function can be used to re-write listing 9 as shown in listing 10.

Listing 10: sample *Python* code to run *XFOIL* with `os.path.sep` command

```
import os
sep=os.path.sep
file_path = 'C:' + sep + 'Users' + sep + 'aijf197' + sep + \
 'Documents' + sep
xfoil_path ='C:' + sep + 'apps' + sep + 'XFOIL' + sep
run_xfoil_command = xfoil_path + 'xfoil < ' + file_path + \
 'commands.in'
os.system(run_xfoil_command)
```

Now we need to retrieve the results from `polar.dat` (listing 5), which can be read using, for example, the *Python* `readlines()` function as shown in listing 11. Here we read all the lines of the file and then assign specific elements of the last line to `cl` and `cd`. A command to delete the `polar.dat` file is included to allow *XFOIL* to be run again in the same way if needed.

Listing 11: sample *Python* code to read results from `polar.dat`

```
aero_data_file = open(file_path + 'polar.dat', 'r')
lines = aero_data_file.readlines()
aero_data_file.close()
#delete Xfoil output file ready for next Xfoil run
os.system('del ' + file_path + 'polar.dat')
#Linux/OSX: os.system('rm -f ' + file_path + 'polar.dat')
cl = float(lines[-1][11: 17])
cd = float(lines[-1][20: 27])
```

## 2.2 Lab 2: run an *XFOIL* analysis of a Bézier spline aerofoil using *Python*

Create a file `aclab2.py` that contains the functions listed below.

- A function `bezier_spline_aerofoil(file_path)` that creates a Bézier spline aerofoil with lower curve control points:

$$\begin{aligned} \mathbf{l}^{(0)} &= (1.0, 0.0) \\ \mathbf{l}^{(1)} &= (0.5, 0.08) \\ \mathbf{l}^{(2)} &= (0.0, -0.05), \end{aligned} \tag{20}$$

  and upper curve control points:

$$\begin{aligned} \mathbf{u}^{(1)} &= (0.0, 0.1) \\ \mathbf{u}^{(2)} &= (0.4, 0.2) \\ \mathbf{u}^{(3)} &= (1.0, 0.0). \end{aligned} \tag{21}$$

  The weighting of all control points is 1, i.e. $z_l^{(0,1,2,3)} = 1$ and $z_u^{(0,1,2,3)} = 1$. The function should print aerofoil coordinates to the file `aerofoil.dat` in two columns. The first column is the x-coordinates, starting from the trailing edge ($x = 1$) with 101 points along the lower surface to the leading edge ($x = 0$) and then 100 points alone the upper surface to the trailing edge ($x = 1$). The second column contains 201 y-coordinates from the trailing edge ($y = 0$) to the leading edge ($y = 0$) and back to the trailing edge ($y = 0$).

  Example:

```
1.000000        0.000000
0.985001        0.002337
...
0.000150        0.022811
0.000000        0.025000
0.000120        0.027257
...
0.982060        0.005910
1.000000        0.000000
```

  You already have the function `rational_bezier` from `aclab1.py` to create the upper and lower Bézier curves, but you need to find the leading edge control point $\mathbf{l}^{(3)} = \mathbf{u}^{(0)}$ by coding equation 19. With this point calculated, `rational_bezier` can be called to calculate the points on the upper and lower curves and then these can be written to `aerofoil.dat` using, e.g. listing 6.

- A function `run_xfoil(file_path, xfoil_path)` that runs *XFOIL*, reading in `aerofoil.dat` created by `bezier_spline_aerofoil()` and running the viscous analysis using the commands in listing 3, reads and returns $C_D$ and $C_L$ from polar.dat.

  Example:

```
>>> print run_xfoil(file_path, xfoil_path)
(0.00802, 1.2)
```

Listings 8, 10 and 11, which will form the basis of your code, are provided in a text file on Blackboard for your convenience.

If your function is not working, first try opening *XFOIL* from the start menu and copy and paste the contents of *commands.in* (opened in notepad) into the *XFOIL* command prompt line-by-line to check that your input file is formatted correctly. Next check that `polar.dat` has been created and contains $C_l$ and $C_D$ values.

# 3 Optimization

## 3.1 Lecture 4: shape optimization

All areas of design are concerned with producing the best product in terms of physical performance, cost, aesthetics, etc. Engineering design is, more formally, the optimisation of performance criteria based on analyses. The performance criteria will include profit margin and various fitness criteria such as weight, stiffness, cost to consumer, etc. The analyses could include cost modelling, finite element analysis, wind-tunnel testing, consumer trials, etc. Indeed, a major part of engineering design is identifying appropriate criteria and analyses on which to base decisions.

We are going to design an aerofoil section for SUHPA (Southampton University Human Powered Aircraft, figure 13). We will conisider the section for the six metre station (six metres out from the wing root). Here the chord is 0.698 m and the design speed for this aerofoil will be 12.5 m/s. Thus the constants defined at the top of our functions that run *XFOIL* will look like:

```
a=340.3 #speed of sound
v=12.5 #velocity
M=v/a #Mach number
nu=0.00001461 #kinematic viscosity
L=0.698 #chord length of aircraft wing
Re=v*L/nu #Reynold's number
```

We will also need to define the lift coefficient, which at this location along the wing is 0.843.

We will minimise the drag (known as the cost function, or objective function) of an aerofoil for a fixed lift (our constraint function). We have one form of analysis: *XFOIL*. The optimum aerofoil is found through methodical changes to its geometry based results from *XFOIL* according to some optimisation algorithm. There are many optimisation algorithms. Indeed whole scientific communities are devoted to their development. Algorithms include *local* methods such as the Newton-Raphson method (which steps towards a local minimum, or saddle point, based on first and second derivatives of the objective function) and quasi-Newton methods (e.g. Broyden-Fletcher-Goldfarb-Shanno, BFGS, where second derivatives are not required). *Global* methods include those inspired by nature like genetic algorithms and particle swarm. Each of these algorithms is


Figure 13: SUHPA in flight during the 2013 Icarus Cup at Sywell Aerodrome (piloted by Guy Martin, photo by Fred To).

suited to different types of problems. We will examine the use of the *Python* implementations of BFGS.

### 3.1.1 BFGS

*SciPy*'s `scipy.optimize.fmin_bfgs` minimises a function using the BFGS algorithm. Information from the *SciPy* reference guide on the required input parameters and outputs is given below.

The algorithm is called with:

```
scipy.optimize.fmin_bfgs(f, x0, fprime=None, args=(), gtol=1e-05, norm=inf,
epsilon=1.4901161193847656e-08, maxiter=None, full_output=0, disp=1,
retall=0, callback=None),
```

where the input parameters are:

`f` : callable f(`x`,*`args`). Objective function to be minimized.
`x0` : ndarray. Initial guess.
`fprime` : callable f(`x`,*`args`), optional. Gradient of f.
`args` : tuple, optional. Extra arguments passed to `f` and `fprime`.
`gtol` : float, optional. Gradient norm must be less than `gtol` before successful termination.
`norm` : float, optional. Order of `norm` (Inf is max, -Inf is min)
`epsilon` : int or ndarray, optional. If `fprime` is approximated, use this value for the step size.

`callback` : callable, optional An optional user-supplied function to call after each iteration. Called as `callback`(xk), where xk is the current parameter vector.

`maxiter` : int. Maximum number of iterations to perform.

`full_output` : bool. If True,return `fopt`, `func_calls`, `grad_calls`, and `warnflag` in addition to `xopt`.

`disp` : bool Print convergence message if True.

`retall` : bool. Return a list of results at each iteration if True.

and returns:

`xopt` : ndarray. Parameters which minimize f, i.e. f(`xopt`) == `fopt`.

`fopt` : float. Minimum value.

`gopt` : ndarray. Value of gradient at minimum, f(`xopt`), which should be near 0.

`Bopt` : ndarray. Value of 1/f(`xopt`), i.e. the inverse hessian matrix.

`func_calls` : int. Number of function calls made.

`grad_calls` : int. Number of gradient calls made.

`warnflag` : integer 1 : Maximum number of iterations exceeded. 2 : Gradient and/or function calls not changing.

`allvecs` : list. Results at each iteration. Only returned if `retall` is True.

The call statement above contains the default values for the parameters. We only need to specify parameters that require different values. Naturally `f` (the objective function) must be specified, along with a starting point for the optimiser, `x0`. *XFOIL* does not provide gradient information, so `fprime` is left at `None`. We can pass arguments such as `cl` (the lift coefficient), `file_path` and `xfoil_path` to the objective function using the `args` parameter. The algorithm will stop when either `gtol` or `maxiter` is reached. By default `gtol=1e-05` is used, i.e. the algorithm iterates until the gradient of the objective function with respect to the variables is below $1 \times 10^{-5}$. It computational resources and/or time are limited, `maxiter` can be specified too.

The other parameter we need to worry about is `epsilon`. With `fprime=None`, the BFGS method approximates the gradient of the function with respect to the design variables using finite differencing. The step size used for this finite differencing (`epsilon`) is critical. For analytical functions, a smaller step size gives better precision (but not so small that machine precision is an issue). For iterative solutions using computational meshes (e.g. *XFOIL*), the 'noise' in the function means that small step sizes will be inaccurate and may even yield gradients in the wrong direction. A balance between a large enough step to avoid noise issues and a small enough step size to yield a useful gradient is required.

### 3.1.2 L-BFGS-B

Here the 'L' stands for 'limited memory', which means that this algorithm is more efficient for large numbers of design variables and, more important to us,

the 'B' stands for 'bounded', which means that bounds can be set on the design variables. This is important as often geometry becomes physically unrealisable or unsolvable outside of certain bounds on the design variables, resulting in simulation failures and so optimiser failure. *SciPy*'s `scipy.optimize.fmin_l_bfgs_b` is called in a similar way to `scipy.optimize.fmin_bfgs` (or a maddeningly slightly different way, depending on how you see things):

```
scipy.optimize.fmin_l_bfgs_b(f, x0, fprime=None, args=(), approx_grad=0,
bounds=None, m=10, factr=10000000.0, gtol=1e-05, epsilon=1e-08, iprint=-1,
maxfun=15000, disp=None)
```

The new parameters that concern us are:

`approx_grad` : bool. Whether to approximate the gradient numerically (in which case `func` returns only the function value).

`bounds` : list. (min, max) pairs for each element in x, defining the bounds on that parameter. Use `None` for one of min or max when there is no bound in that direction.

### 3.1.3 Further considerations

As alluded to above, 'noise' in the objective function may cause difficulties when calculating gradients. As a subsonic aerofoil geometry variable changes, one would expect a smooth, continuous change in the lift and drag. However, as the flow is being solved over a finite computational mesh, changes in the geometry and subsequent re-meshing will result in small step-changes in the solution. Figure 14 shows a set of aerofoils and the *XFOIL* predicted $C_D$ for these airfoils. Note the scatter around an assumed smooth trend. The BFGS `epsilon` will need to be selected carefully for the optimiser to succeed here. Sometimes it is preferable to fit a curve (or surface for multiple variables) to the data and optimise using the curve-fit in lieu of the true objective function. This is quicker and removes problems with noise. Listing 12 shows the *Python* code to produce the data and 'moving least-squares' curve fit in figure 14.

Listing 12: *Python* code to produce a moving least squares fit to the *XFOIL* data.

```
from scipy.optimize import fsolve

#1D quadratic moving least squares
def mls(x,X,y,sigma):
    N = max(shape(y))
    weights = zeros(N)
    A = zeros([N, 3])
    A[:, 0] = X**2
    A[:, 1] = X
    A[:, 2] = ones([1, N])
    for i in range(1, N):
        weights[i] = exp(-sum((x - X[i]) ** 2) / (2 * sigma))
    W = diag(weights)
```

```
    a = linalg.lstsq(dot(dot(A.conj().T, W), A),\
dot(dot(A.conj().T, W), y) )
    f = a[0][0] * x ** 2 + a[0][1] * x + a[0][2]
    return f

#1D quadratic moving least squares cross-validation
def mls_error(sigma, X, y):
    y_test = zeros(11)
    error = zeros(11)
    for i in range(0, 11):
        y_test[i] = mls(X[i], append(X[0: i], X[i + 1: -1]),\
append(y[0: i], y[i + 1: -1]), sigma)
        error[i] = (cd[i] - y_test[i]) ** 2
    sum_error = sum(error)
    return sum_error

file_path = '...'
xfoil_path = '...'

w_array = linspace(0.6, 1.2, 11)
figure(1)
cd = parameter_sweep(w_array, 0.843, file_path, xfoil_path)
savefig('aerofoils.pdf')
#fit moving least squares
sigma_best = fsolve(mls_error, 0.2, args = (w_array, cd))
w_fine = linspace(1.15, 1.85, 101)
y_pred = zeros(101)
for i in range(0,101):
    y_pred[i] = mls(w_fine[i], w_array, cd, sigma_best)

figure(2)
pylab.plot(w_array, cd, 'o', label = 'XFOIL data')
plot(w_fine, y_pred, label = 'MLS fit')
legend()
xlabel('w')
ylabel('c_d')
savefig('parameter_sweep.pdf')
```

## 3.2 Lab 3: conduct an aerofoil shape parameter study with *Python* and *XFOIL*

Create a file `aclab3.py` that contains the functions listed below.

- A function `parametric_aerofoil(w,file_path)` that creates a Bézier spline aerofoil with the same control points as Lab 2, but with $z_u^{(2)} = w$ ($w$ being defined by `w`; the first input argument of the function. As in Lab 2, this function should write coordinates to the file `aerofoil.dat` and in the same format as for Lab 2.

- A function `run_xfoil_wcl(w,cl,file_path,xfoil_path)` that runs *XFOIL*, reading in `aerofoil.dat` created by `parametric_aerofoil()` and running the viscous analysis using the commands in listing 3 (except that the lift coefficient in `commands.in` should now be defined by `cl`), reads and

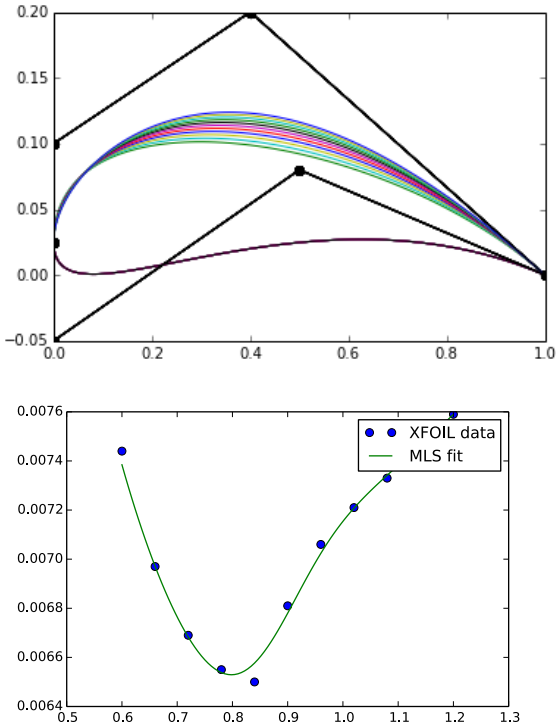


Figure 14: aerofoil shapes (note scales are not equal) and a moving least squares for to the *XFOIL* data, calculated in listing 12.

returns CD and CL from polar.dat. Use the SUHPA velocity and aerofoil chord values.

Example:

```
>>> print run_xfoil_wcl(1.1, 0.8433, file_path, xfoil_path)
(0.00738, 0.843)
```

- A function `parameter_sweep(w_array,cl,file_path,xfoil_path)` which is the same as `run_xfoil_wcl()`, but where `w_array` is an array of weights for the control point at `[0.4, 0.2]`. The function should return the corresponding array of $C_D$ values.

  Example:

```
>>> w_array = linspace(0.6, 1.2, 11)
>>> print parameter_sweep(w_array, 0.843, file_path, xfoil_path)

[ 0.00744   0.00697   0.00669   0.00655   0.0065    0.00681   0.00706
0.00721
  0.00733   0.00746   0.00759]
```

  Add the code in lising 12 to plot a 'moving least squares' fit to your *XFOIL* data:

## 3.3 Lab4: optimise an aerofoil using *Python* and *XFOIL*

Create a file aclab4.py that contains the functions listed below.

- A function `one_dim_opt(x0,cl,file_path,xfoil_path)` which optimises the Bézier spline aerofoil weight $z_u^{(2)}$ for SUHPA using the `scipy.optimize.fmin_bfgs` optimiser.

  Use lower curve control points:

$$\begin{aligned} \mathbf{l}^{(0)} &= (1.0, 0.0) \\ \mathbf{l}^{(1)} &= (0.5, 0.08) \\ \mathbf{l}^{(2)} &= (0.0, -0.05), \end{aligned} \tag{22}$$

  and upper curve control points:

$$\begin{aligned} \mathbf{u}^{(1)} &= (0.0, 0.11) \\ \mathbf{u}^{(2)} &= (0.4, 0.2) \\ \mathbf{u}^{(3)} &= (1.0, 0.0). \end{aligned} \tag{23}$$

  Example:

```
>>> print one_dim_opt(x0,cl,file_path,xfoil_path)

Optimization terminated successfully.
```

```
         Current function value: 0.006490
         Iterations: 2
         Function evaluations: 24
         Gradient evaluations: 8
[ 0.85011615]
```

The `scipy.optimize.fmin_bfgs` optimiser is called in the format:

```
>>> opt_out=fmin_bfgs(run_xfoil_wcl,x0,args=(cl,file_path,xfoil_path))
```

where `x0` is the starting point for the optimiser, i.e. your best guess at the optimum $z_u^{(2)}$.

- A function `four_dim_opt(x0,weight_limits,cl,file_path,xfoil_path)` which optimises the Bézier spline aerofoil weights $z_u^{(2)}, z_u^{(3,)}, z_l^{(2)}, z_l^{(3)}$ at Mach number $M = 0.05$ using the `scipy.optimize.fmin_l_bfgs_b` optimiser. Use the above control points.

  Example:

```
>>> print four_dim_opt(x0,weight_limits,cl,file_path,xfoil_path)

(array([ 1.3,  0.5,  0.5,  0.5]), 10, 0)
```

  The `scipy.optimize.fmin_l_bfgs_b` optimiser is called in the format:

```
>>> opt_out=fmin_l_bfgs_b(run_xfoil_wcl, x0, args=(cl,file_path,xfoil_path),
bounds=weight_limits, epsilon=step_size, approx_grad=True)
```

  where `x0` is the starting point for the optimiser, e.g. `x0=[1, 1, 1, 1]` , `weight_limits` are upper and lower bounds on the the weights to be optimised,[1] e.g. `weight_limits=((0.5, 1.3),(0.5, 1.3),(0.5, 1.3),(0.5, 1.3))` and `step_size` is the finite differencing step size (the same as for `scipy.optimize.fmin_bfgs`). The input `approx_grad=True` is required so that `step_size` can be defined.

**This lab will be assessed during next week's lab session.**

# 4 Further computational geometry

The material in this section will not be assessed, but is included here to show how your *Python* implemented geometry sits in context with the NURBS based geometry of modern CAD engines.

[1] If no upper and lower bounds are set, the optimiser may try to evaluate geometries that cannot be solved in *XFOIL*. Wide bounds increase the search-space and so better designs could be found, but failures are more likely. If *XFOIL* fails, shrink your bounds. Look at the optimised values (stored in `opt`) to see if upper or lower bounds are being hit and expand your bounds accordingly

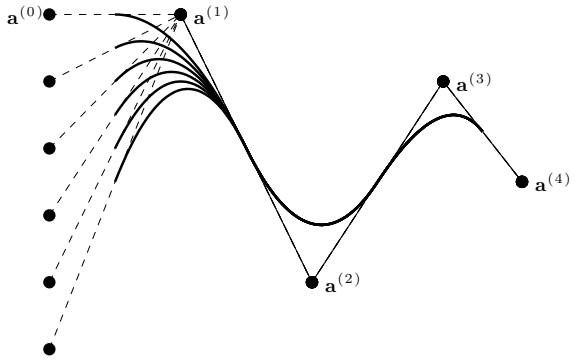


Figure 15: a quadratic rational B-spline showing the vicinity of local control as control point $\mathbf{a}^{(0)}$ is shifted.

## 4.1 Lecture 5: B-splines and NURBS

B-splines, where the 'B' stands for 'basis', have a number of key characteristics which may make them preferable to Bézier splines in many applications.

- The degree of a B-spline is not determined by the number of control points.
- Local control of the spline is possible (unlike Bézier splines where moving one control point affects the whole spline).
- The degree of continuity between segments can be specified.

Local control of the spline means that moving or changing the weight of a control point will only affect the spline in the vicinity of this point. The extent of this vicinity is determined by the degree of the B-spline. In figure 15 we see that moving control point $\mathbf{a}^{(0)}$ only affects the a degree two spline up until midway between $\mathbf{a}^{(1)}$ and $\mathbf{a}^{(3)}$ (up until the end of the first segment).

A B-spline is made of $n-d$ segments, which we shall call $\mathcal{B}_0, \mathcal{B}_1, ..., \mathcal{B}_{(n-d)}$, where $n+1$ is the number of control points (remember these are numbered from zero) and $d$ is the degree of each segment (the degree of the B-spline). The segments start and end at joints $\mathbf{k}^{(0)}, \mathbf{k}^{(1)}, ..., \mathbf{k}^{(n-d+1)}$. Note that, unlike a Bézier spline, the ends of the B-spline $\mathbf{k}^{(0)}$ and $\mathbf{k}^{(n-d+1)}$ do not coincide with the first and last control points. The exception to this is if there are multiple control points: a degree $d$ B-spline passes through a control point repeated $d$ times. For example if $\mathbf{a}^{(0)} = \mathbf{a}^{(1)}$, a degree 2 B-spline will start at the first control point (the first segment will be a straight line towards $\mathbf{a}^{(2)}$).

We will start by defining a uniform B-spline using control points and then go on to interpolating B-splines, i.e. defined by the joint points, which may be more practical in some situations.

Expressed in the same way as (**??**), the $i$th segment of a $d$-degree uniform B-spline can be calculated as

$$\mathcal{B}_i(t) = (t^d, t^{d-1}, ..., t, 1)\mathbf{M}^{(d)} \begin{pmatrix} \mathbf{a}^{(i-1)} \\ \mathbf{a}^{(i)} \\ \vdots \\ \mathbf{a}^{(i+d-1)} \end{pmatrix} \tag{24}$$

where elements of the basis matrix $\mathbf{M}$ are given by

$$m_{i,j}^{(d)} = \frac{1}{d!} \begin{pmatrix} n \\ i \end{pmatrix} \sum_{k=j}^{d} (d-k)^i (-1)^{d-j} \begin{pmatrix} d+1 \\ k-j \end{pmatrix}. \tag{25}$$

The first four $\mathbf{M}$ matrices are:

$$\begin{aligned}
\mathbf{M}^{(d=1)} &= \frac{1}{1!} \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix}, \\
\mathbf{M}^{(d=2)} &= \frac{1}{2!} \begin{pmatrix} 1 & -2 & 1 \\ -2 & 20 & \\ 1 & 1 & 0 \end{pmatrix}, \\
\mathbf{M}^{(d=3)} &= \frac{1}{3!} \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix}, \\
\mathbf{M}^{(d=4)} &= \frac{1}{4!} \begin{pmatrix} 1 & -4 & 6 & -4 & 1 \\ -4 & 12 & -12 & 4 & 0 \\ 6 & -6 & -6 & 6 & 0 \\ -4 & 12 & 12 & 4 & 0 \\ 1 & 11 & 11 & 1 & 0 \end{pmatrix}.
\end{aligned}$$

Varying $t$ from zero to one, equation 24 traces out the $i$th segment of the B-spline between joints $\mathbf{k}^{(i)}$ and $\mathbf{k}^{(i+1)}$. These joints can be calculated as

$$\mathbf{k}^{(i)} = \frac{1}{d!}(\mathbf{a}^{(i)}\mathbf{a}^{(1)} \dots \mathbf{a}^{(i+d-1)}) \begin{pmatrix} c_0^{(d)} \\ c_1^{(d)} \\ \vdots \\ c_{d-1}^{(d)} \end{pmatrix} \tag{26}$$

where

$$c_j^{(d)} = \sum_{k=j}^{d} (d-k)^d (-1)^{k-j} \begin{pmatrix} d+1 \\ k-j \end{pmatrix} \tag{27}$$

($c_{0,1,...,d}^{(d)}$ is also the first $d$ elements of the last row of the $\mathbf{M}^{(d)}$ matrix).

### 4.1.1 interpolating B-spline

Having found the joint points, given the control points, we can now find the control points (and so the B-spline) given the joint points. We actually need a little more information. A B-spline with $n+1$ control points has $n-d+1$ joints and so along the $n-d+1$ equations for these joints, we need $d-1$ further equations. Specifying tangents gives us the remaining expressions. For a quadratic B-spline we only need to specify one tangent, e.g. the start tangent, resulting the the following system of equations:

$$\frac{1}{2!}\begin{pmatrix} -2 & 2 & 0 & \dots & 0 & 0 \\ c_0^{(2)} & c_1^{(2)} & 0 & \dots & 0 & 0 \\ 0 & c_0^{(2)} & c_1^{(2)} & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & c_0^{(2)} & c_1^{(2)} \end{pmatrix} \begin{pmatrix} \mathbf{a}^{(0)} \\ \mathbf{a}^{(1)} \\ \mathbf{a}^{(2)} \\ \vdots \\ \mathbf{a}^{(n)} \end{pmatrix} = \begin{pmatrix} \mathbf{t}^{(0)} \\ \mathbf{k}^{(0)} \\ \mathbf{k}^{(1)} \\ \vdots \\ \mathbf{k}^{(n-d+1)} \end{pmatrix}, \quad (28)$$

which can easily be solve to find $\mathbf{a}^{(0,\dots,n)}$.

We can use the above to define a B-spline through a set of joints; yielding a versatile curve with intuitive,local control via the calculated control points. However, specifying a mix of joints and tangents may be undesirable when compared with the convenient method of defining a Bézier curve with start, end and intermediate control points. An answer lies in the use of a *knot vector*.

### 4.1.2 Knots

Each segment of a B-Spline is defined over an interval where $t$ varies $\in [0,1]$ and the curve is defined, in this interval by $d+1$ control points. For example, from equation 24 we see the first segment of a degree 3 B-spline is defined by $\mathbf{a}^{(0)}, \mathbf{a}^{(1)}, ..., \mathbf{a}^{(3)}$ multiplied by some weightings (basis functions; the 'B' in B-spline). These basis functions overlap from segment to segment. Figure 16 shows how four basis functions are 'in play' in each interval for a degree three B-spline. Referring back to equation 24, in each interval $\mathbf{a}^{(i-1)}$ is multiplied by the basis function decreasing from $\frac{1}{6}$, $\mathbf{a}^{(i)}$ by the one decreasing from $\frac{2}{3}$, $\mathbf{a}^{(i+1)}$ by the one increasing from $\frac{1}{6}$, and $\mathbf{a}^{(i+2)}$ by the one increasing from 0. Note how similar these basis functions are to Bézier curve weightings, except that, since they overlap, the curve does not pass through any control points.

The start and end of the intervals can be defined as a sequence of knots in a non-decreasing vector with unit, uniform spacing (e.g $\mathbf{k} = [-10123]$, as in figure 16). The parameter $t$ varies $\in [0,1]$ between each of these knots. In practice, we work with the knot-vector when defining the B-spline and convert to $t \in [0,1]$ 'in the maths'. The power of this knot-vector definition is when we make it non-uniform, with repeated values and non-unit intervals. This produces a non-uniform B-spline and we only need to make it rational to create a NURBS. All we need to do is multiply by control point weights and divide through by the

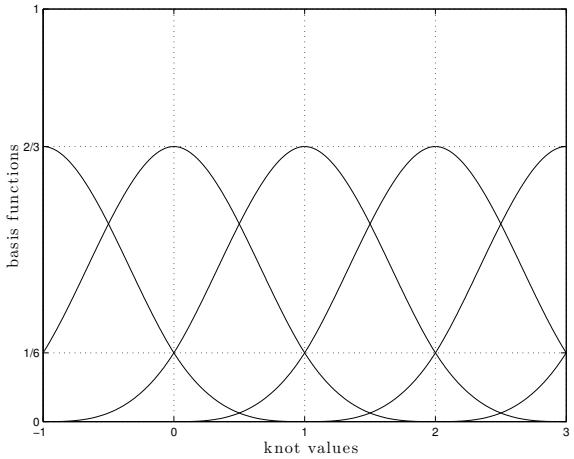


Figure 16: Overlapping degree three B-spline basis functions.

sum of the product of these weights and the basis function (in the same way as equation 17) and we have a non-uniform rational basis spline (NURBS).

### 4.1.3 NURBS

By varying the knot vector in a non-uniform manner, we can achieve an increased level of versatility and recoup some of the attractive features of B/'ezier curves. The first feature to note is that repeating a knot value $d+1$ times results in the NURBS passing through a control points. The knot vector is $n + d + 1$ long so for a degree three NURBS with four control points there are eight knots. Repeating the first and last knot values four times will create a NURBS with one cubic segment that starts and ends at the first and last control point. The basis functions for this NURBS are shown in figure 17. We see that they are the same as those for a Bézier curve (figure 6). A Bézier curve is indeed a special case of a NURBS from which, using the knot vector and the ability to specify the degree of the NURBS, we can add more and more complexity.

We can take advantage of NURBS local control to, for example, refine the shape of the B/'ezier curve aerofoil definition in figure 10. With the influence of a control point extending over $d + 1$ knot values, adding an extra control point will not give local control in our degree three definition. Reducing to degree two *and* adding a control point gives local control, as shown in figure 18 where moving a control point on the lower surface gives control over the aft portion of the aerofoil without affecting the leading edge region.

The knot vector for the lower surface NURBS is $\mathbf{k} = [0, 0, 0, 1, 2, 3, 3, 3]$ and the resulting basis functions are shown in figure 19. Note how the second basis function is non-zero only for the first and second segment, i.e. it has no influence on the last segment (the leading edge region of the aerofoil in figure 18.

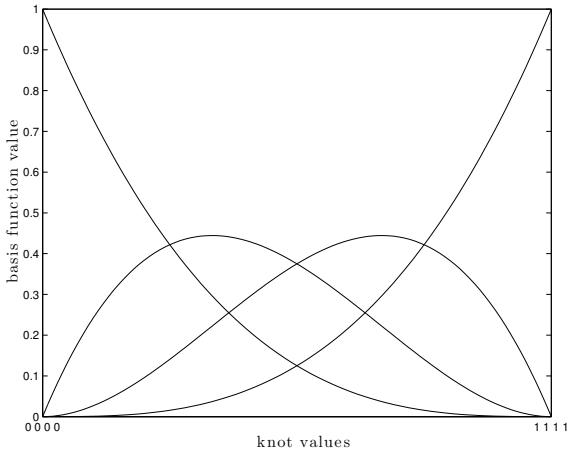


Figure 17: Repeated knot values in a third degree NUBRS yield the cubic Bernstein polynomials in figure 6 (i.e. the NURBS is equivalent to a B/'ezier curve).

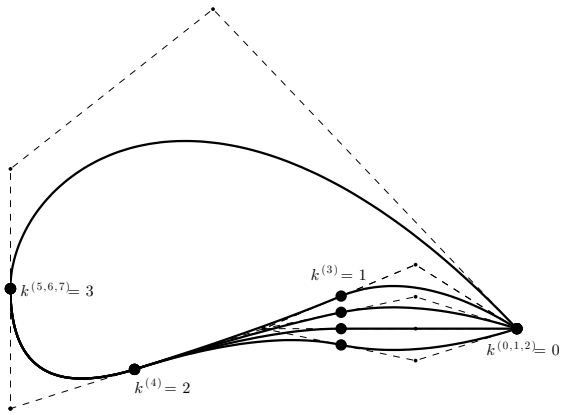


Figure 18: Reducing the degree of the lower surface NURBS to three and adding a control point gives local control towards the trailing edge, potentially giving more design capability than the B/'ezier curve definition in figure 10 (shown here by moving the additional control point). Note that the plot has been stretched to highlight the geometry changes.

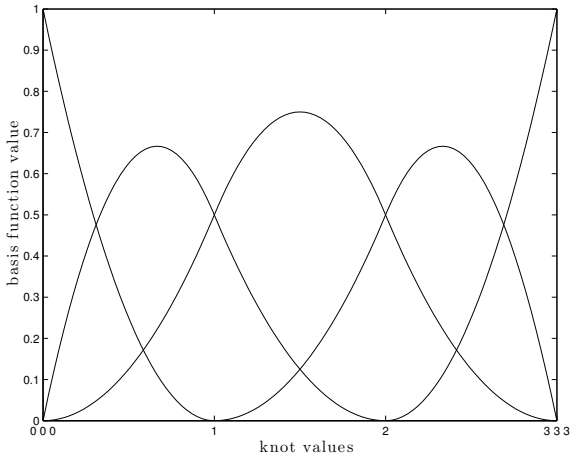


Figure 19: Degree two NURBS basis functions for knot vector $\mathbf{k} = [0, 0, 0, 1, 2, 3, 3, 3]$, resulting in a curve starting and ending at the first and last control points. The central segment is 'uniform', but the first and last segments have basis functions distorted by the repeated knot values.

A further level of detail can be obtained by varying the spacing of the knot vector. Bringing knots together pulls the NURBS towards the control polygon: for a degree three NURBS two repeated knot values matches the gradient of the NURBS at the knot to the control polygon (i.e. alligns it with a line drawn between the knots either side), three repeated knot values causes the NURBS to pass through a control point (with a gradient discontinuity), four repeated knots causes a break in the NURBS between two control points.

Figure 20 shows the range of 'manoeuvres' possible with a degree three NURBS (omitting the cause of changing control point weights, which is similar to changin rational B/'ezier spline weights, as shown in figure 12). Figure 20(a) is a uniform degree threevB-spline and the uniform weightings from the uniform knot-vector are shown in the right hand plot. Figure 20(b) has the degree increased to seven and the first and last knot values repeated to yield a Bézier spline equivalent. In figure 20(c) the degree is reduced back to three and the first and last knots repeated such that the NURBS starts and ends at the first and last control points. Figure 20(c) to (f) show the effect of repeated knot values: causing tangent matching, control point interpolation, and breaks in the NURBS.

As a weighted sum of control points, the equation for a NURBS can be expressed in a similar way as a Bézier curve:

$$\mathcal{N}(t) = sum_{i=0}^{n} \mathbf{a}^{(i)} B_{i,d}(t). \tag{29}$$

However, since the basis functions $B_{i,d}(t)$ depend on the knot vector, the calculation of $B_{i,d}(t)$ is rather tedious and usually performed *recursively*. That

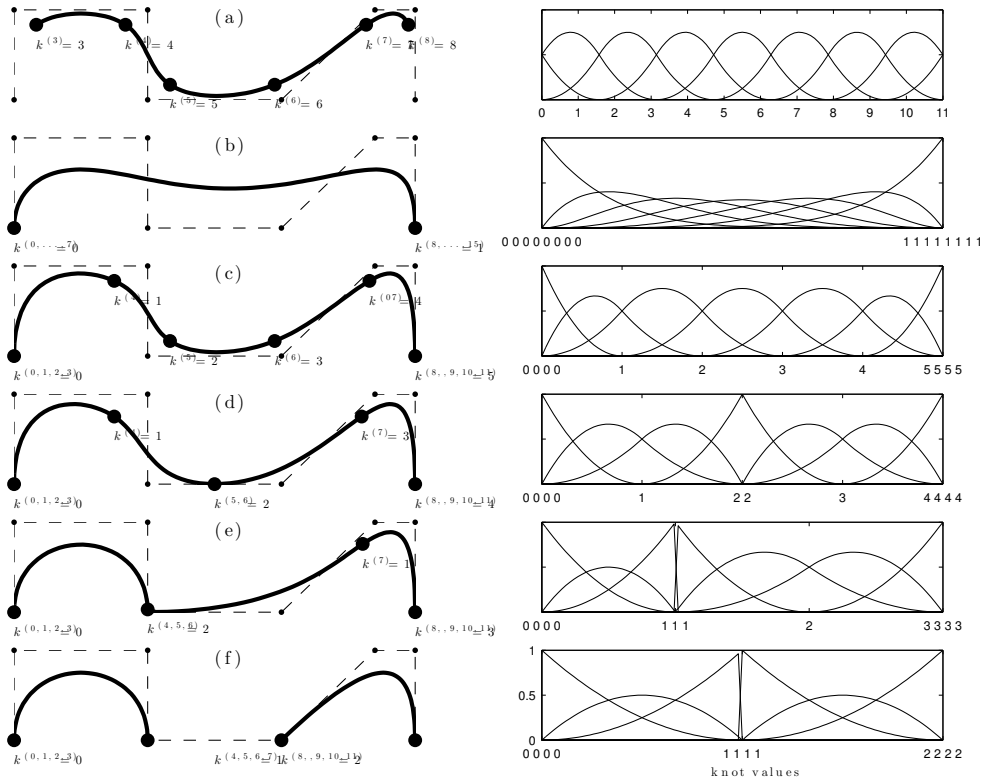


Figure 20: Various NURBS defined by eight control points (left) and knot vectors (right): (a) degree three with uniform knot vector; (b) degree seven; (c-f) degree three. The control point locations are chosen to be similar to Figure 7.19 in Salomon [2006].

is, we start by calculating $B_{0,1}(t)$ and $B_{1,1}(t)$ from which $B_{0,2}(t)$ can be calculated, and so on through to $B_{n,d}(t)$. The implementation of NURBS in *Python* is beyond the scope of this module, but the necessary equations are included below.

The basis functions for $d = 1$ are defined as:

$$B_{i,1}(t) = \begin{cases} 1, & \text{if } \ t \in [i, i+1] \\ 0, & \text{otherwise} \end{cases} \tag{30}$$

and the remaining basis functions can be calculated recursively from

$$B_{i,d}(t) = \frac{t - k^{(i)}}{k^{(i+d-1)} - k^{(i)}} B_{i,d-1}(t) + \frac{k^{(i+d)} - t}{k^{(i+d)} - k^{(i+1)}} B_{i+1,d-1}(t), \tag{31}$$

noting that where a zero denominator occurs, the term should be evaluated as zero.

# References


M. Drela. XFOIL: An analysis and design system for low reynolds number airfoils. *Conference on Low Reynolds Number Airfoil Aerodynamics, University of Notre Dame*, 1989.

M. Drela and M.B. Giles. Viscous-inviscid analysis of transonic and low reynolds number airfoils. *AIAA Journal*, 25(10):1347–1355, October 1987.

G. Farin. *NURBS: from projective geometry to practical use.* Aerospace Science Series. A K Peters, Natick, Massachusetts, second edition edition, 1999.

D. Salomon. *Curves and Surfaces for Computer Graphics.* Springer, New York, 2006.

I. Thomas. *Selections Illustrating the History of Greek Mathematics (with English translation).* Harvard University Press, Cambridge, Massachusetts, 1939.