Parameter Selection and Derivative Conditions for B-Splines Applied to Gas Turbine Blade Modeling

Andreas Söderlund

Bachelor's thesis
2015:K2

Lund University

Faculty of Science
Centre for Mathematical Sciences
Numerical Analysis

Centrum Scientiarum Mathematicarum

Abstract

In gas turbine blade modeling a stable but yet flexible method of describing the blade shape is crucial. Polynomials and Bézier curves have been previously used and in this paper B-splines are employed instead. This method removes the need for many of the interior derivative conditions on the curve but special care must be taken when selecting the parameter values and appropriate derivative conditions. This paper presents several parameter selection methods and shows how they affect the construction of the curve. In the end a satisfactory curve is produced where one of its conditions depends on a variable whose value can be set to optimize the shape of the curve in the modeling procedure.

Contents

1 Introduction

In the development of gas turbines several stages of computer simulation in both $\mathbb{R}$, $\mathbb{R}^2$ and $\mathbb{R}^3$ are employed [14]. Because a comprehensive model of the turbines in $\mathbb{R}^3$ can be difficult to create and analyse, several cross sections in two dimensions of the blade are stacked to give an approximate blade design.

Figure 1: The inlet guide vanes of an Atar turbojet. Air flows from the viewer towards the background of the photo [2].

There are several ways of describing these two-dimensional cross sections and Svensson [14] describes two of them, namely by combining three and four degree polynomials as proposed by Ye [16] and by composite Bézier curves. Because the blade surface requires several of its derivatives to be continuous, both of these methods will face problems at the points on the blade where the functions meet as not only the function values but also its derivatives must be matched. We will instead investigate if the curve can be described using the B-spline basis. With this approach we will immediately gain the desired smoothness along the curve and we can easily add additional conditions on the curve. Even though we have several conditions on the curve, the shape of the blade does not have to be unique. In Figure 2 we see three different blades satisfying the same conditions.

We will begin by explaining the two-dimensional geometry of the cross sections and the conditions that must be satisfied by the curve. In Section 3 we will introduce the B-spline basis and some of its properties. We will also look at how the basis can be used for interpolation. In Section 4 we introduce several methods of choosing the parameter values and how they are used to construct the knot sequence of the B-spline, and in Section 5 we will deal with the choice of

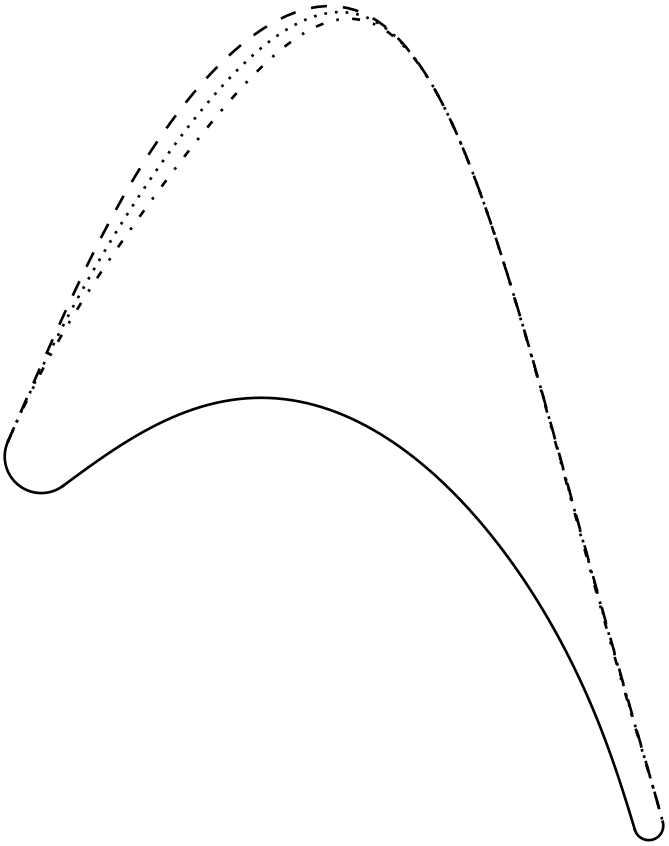

Figure 2: Three possible shapes of the blade fulfilling the same conditions.

derivative conditions for the blade. In the Results section we will look at different ways the blade can be constructed and what kind of blades these choices produce. In the end a satisfactory shape is obtained and we conclude the report by proposing possible improvements and extentions. In the Appendix the code used to produce the curves are presented.

I thank my supervisor Carmen Arévalo for her insightful comments and support throughout the development of this thesis, and Magnus Genrup at the Department of Energy Sciences who provided the topic.

2 Blade geometry

The profile of the turbine blade consists of two circular arcs connected by two curves as seen in Figure 3 where the upper curve is the suction side and the lower curve is the pressure side. The shape of the blade is given by 11 blade parameters chosen because of their strong influence on the performance of the turbine [14] and they are as follows:

R_1	Radius of leading edge circular arc
R_2	Radius of trailing edge circular arc
γ	Stagger angle
β_1	Inlet flow angle
β_2	Outlet flow angle
$\Delta\beta_1$	Inlet wedge angle
$\Delta\beta_2$	Outlet wedge angle
τ	Pitch
β_g	Gauging angle
Γ	Uncovered turning
B_x	Airfoil axial chord

These parameters are used to define the points in Figure 3 denoted by s_1, s_3, s_5, p_1 and

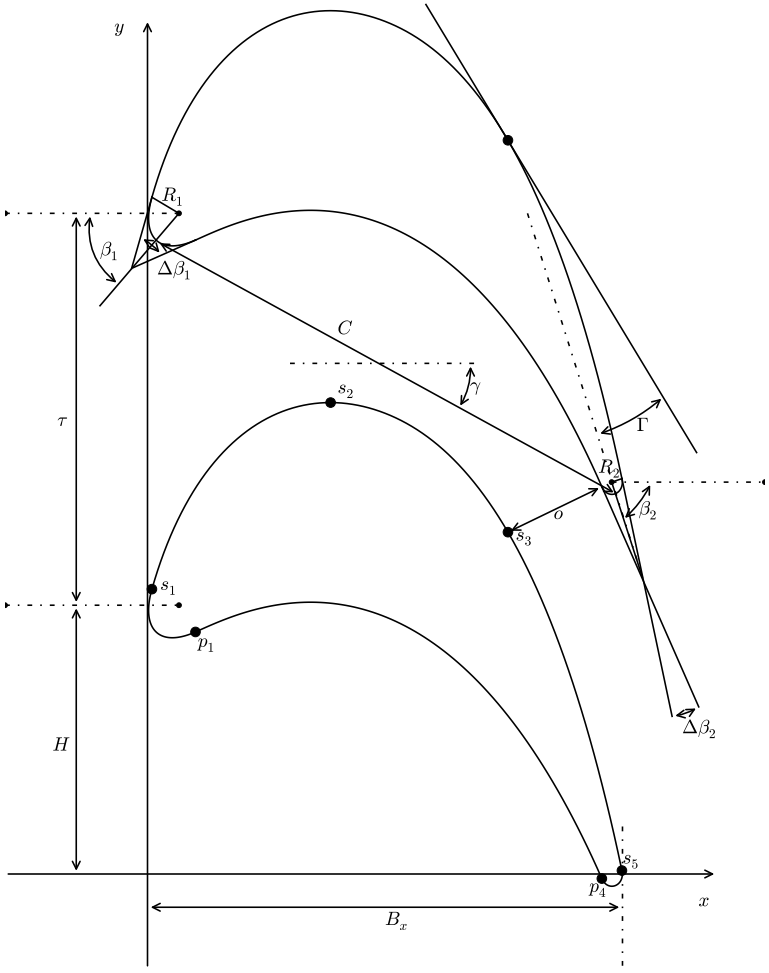

Figure 3: The geometry of the turbine blade and its defining points and angles.

p_4. The leading edge of the blade is described by a circular arc of radius R_1 and wedge angle $\Delta\beta_1$ while the trailing edge is described by a circular arc of radius R_2 and wedge angle $\Delta\beta_2$. Together with β_1 and $\beta 2$, $\Delta\beta_1$ and $\Delta\beta_2$ describe at which angles the suction side connects to the circular arcs. The height of the blade is calculated as

$$H = C\sin\gamma + R_1\sin\beta_1 - R_2\sin\beta_2$$

where C is the length of the blade defined by [14]

$$C = \frac{B_x - R_1(1-\cos\beta_1) - R_2(1-\cos\beta_2)}{\cos\gamma}.$$

The quantity β_g is the gauging angle that determines the width of the throat, o, and is defined as the ratio between o and the pitch τ [16],

$$\sin(\beta_g) = \frac{o}{\tau}. \tag{1}$$

2.1 Investigating s_1 and s_5

As seen in Figure 4, s_1 lies on the leading edge circular arc with center (R_1, H) and radius R_1 where it meets the suction side. Because the suction side meets the radius R_1 at a right angle at s_1, the angle θ_s between s_1 and the leading edge circular arc horizontal radius is

$$\theta_s = \frac{\pi}{2} - \frac{\Delta\beta_1}{2} - \beta_1$$

If we write s_1 as $s_1 = (x_{s_1}, y_{s_1})$ we see that

$$\begin{aligned} x_{s_1} &= R_1 - R_1\cos\theta_s \\ &= R_1\big(1 - \cos(\frac{\pi}{2} - \frac{\Delta\beta_1}{2} - \beta_1)\big) \\ &= R_1\big(1 - \sin(\frac{\Delta\beta_1}{2} + \beta_1)\big), \end{aligned}$$

and

$$y_{s_1} = H + R_1\cos(\frac{\Delta\beta_1}{2} + \beta_1).$$

The trailing edge circular arc has center $(B_x - R_2, 0)$ and radius R_2 and as before, we see that s_5 lies where the suction side meets the trailing edge circular arc. Following the previous reasoning, the angle ϕ_s between s_5 and the x-axis is

$$\phi_s = \frac{\pi}{2} - \frac{\Delta\beta_2}{2} - \beta_2.$$

If we write s_5 as $s_5 = (x_{s_5}, y_{s_5})$ we get

$$\begin{aligned} x_{s_5} &= B_x - R_2 + R_2\cos\phi_s \\ &= B_x - R_2\big(1 - \cos(\frac{\pi}{2} - \frac{\Delta\beta_2}{2} - \beta_2)\big) \\ &= B_x - R_2\big(1 - \sin(\frac{\Delta\beta_2}{2} + \beta_2)\big) \end{aligned}$$

and similarly

$$y_{s_5} = R_2 \cos(\frac{\Delta\beta_2}{2} + \beta_2).$$

This gives us the following two expressions for s_1 and s_5:

$$s_1 = \big(R_1\big(1 - \sin(\frac{\Delta\beta_1}{2} + \beta_1)\big), H + R_1 \cos(\frac{\Delta\beta_1}{2} + \beta_1)\big) \tag{2}$$

$$s_5 = \big(B_x - R_2\big(1 - \sin(\frac{\Delta\beta_2}{2} + \beta_2)\big), R_2 \cos(\frac{\Delta\beta_2}{2} + \beta_2)\big). \tag{3}$$

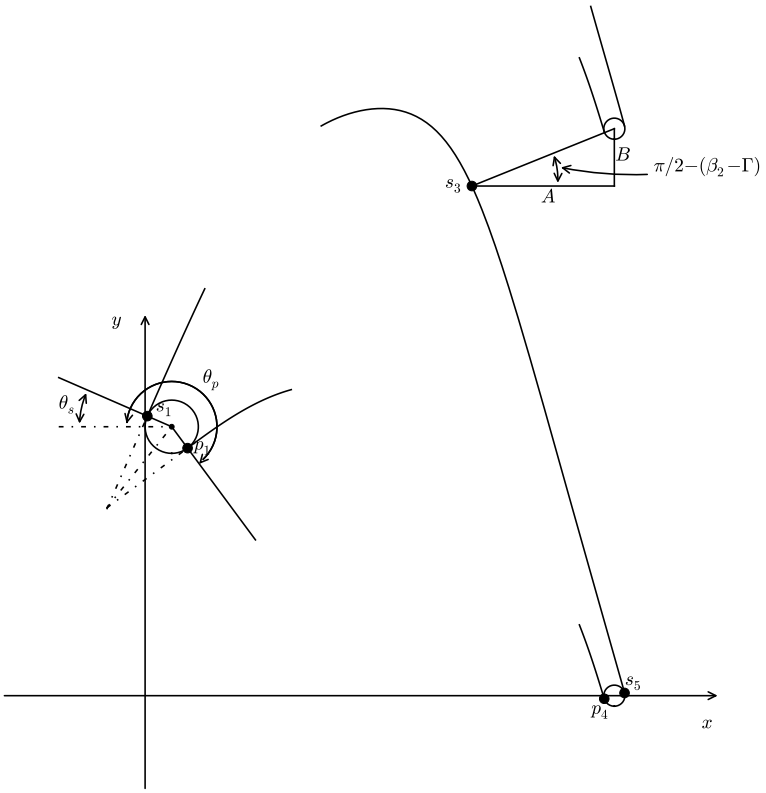

Figure 4: The defining points of the blade and their positions.

2.2 Investigating s_3

The point s_3 is defined as the position of the throat o on the suction side. This means that s_3 is the closest point on the lower blade to the upper blade and that the line through the throat is perpendicular to the slope at s_3. By drawing a right triangle with hypotenuse $o + R_2$, side A parallel to the x-axis and side B parallel to the y-axis as seen in Figure 4, we can calculate $s_3 = (x_{s_3}, y_{s_3})$. From the triangle we see that

$$x_{s_3} = B_x - R_2 - A$$
$$y_{s_3} = \tau - B$$

which means that we must find expressions for A and B. The angle between A and the slope of the suction side at s_3 is $\beta_2 - \Gamma$ which means that the angle between the hypotenuse and side A of the triangle is $\frac{\pi}{2} - (\beta_2 - \Gamma)$ using the perpendicularity of the hypotenuse and the slope. From this calculation and equation (1) we get that

$$\begin{aligned} A &= (o + R_2)\cos(\frac{\pi}{2} - (\beta_2 - \Gamma)) \\ &= (\tau \sin\beta_g + R_2)\sin(\beta_2 - \Gamma). \end{aligned}$$

In the same way, we see that

$$\begin{aligned} B &= (o + R_2)\sin(\frac{\pi}{2} - (\beta_2 - \Gamma)) \\ &= (\tau \sin\beta_g + R_2)\cos(\beta_2 - \Gamma) \end{aligned}$$

which means that we have an expression for s_3:

$$s_3 = \big(B_x - R_2(\tau \sin\beta_g + R_2)\sin(\beta_2 - \Gamma), \tau - (\tau \sin\beta_g + R_2)\cos(\beta_2 - \Gamma)\big) \tag{4}$$

2.3 Investigating p_1 and p_4

Just as s_1 and s_5 lie on the intersection of the suction side and the leading and trailing edge circular arcs, p_1 and p_4 lie on the intersection of the pressure side and the circular arcs, as seen in Figure 4. The angle θ_p in Figure 4 is

$$\theta_p = \pi + \beta_1 + \frac{\Delta\beta_1}{2}$$

and the angle ϕ_p between p_4 and the x-axis is

$$\phi_p = \frac{\pi}{2} - \beta_2 + \frac{\Delta\beta_2}{2}.$$

If $p_1 = (x_{p_1}, y_{p_1})$ we can calculate x_{p_1} as

$$\begin{aligned} x_{p_1} &= R_1 - R_1\cos\theta_p \\ &= R_1\big(1 - \cos(\pi + \beta_1 + \frac{\Delta\beta_1}{2})\big) \\ &= R_1\big(1 + \cos(\beta_1 + \frac{\Delta\beta_1}{2})\big) \end{aligned}$$

and similarly

$$y_{p_1} = H + R_1\sin(\beta_1 + \frac{\Delta\beta_1}{2})$$

which means that

$$p_1 = \big(R_1\big(1 + \cos(\beta_1 + \frac{\Delta\beta_1}{2})\big), H + R_1\sin(\beta_1 + \frac{\Delta\beta_1}{2})\big). \tag{5}$$

In the same way, we can find p_4 using ϕ_p:

$$p_4 = \big(B_x - R_2\big(1 - \sin(\beta_2 - \frac{\Delta\beta_2}{2})\big), -R_2\cos(\beta_2 - \frac{\Delta\beta_2}{2})\big). \tag{6}$$

2.4 Slopes at the points

Apart from the position of the points, we also know the slopes there. From Figure 4 we can find the slopes using the angle between the tangent line and the horizontal axis at each point. This means that the slopes at s_1 and p_1 are $\tan(\beta_1 + \frac{\Delta\beta_1}{2})$ and $\tan(\beta_1 - \frac{\Delta\beta_1}{2})$ and at s_5 and p_4 $\tan(-\beta_2 + \frac{\Delta\beta_2}{2})$ and $\tan(-\beta_2 - \frac{\Delta\beta_1}{2})$. At s_3 we previously saw that the angle between the tangent line and side A was $\beta_2 - \Gamma$, which means that the slope there is $\tan(-\beta_2 + \Gamma)$.

3 B-spline definition and properties

3.1 B-spline basics

A spline of degree n is a function piecewise defined by polynomials of degree $\leq n$ with sufficient smoothness at the points of the curve where the polynomials meet. These points are called the knots of the spline. A B-spline $s(x)$ (where B stands for Basis) is a spline defined by a non-decreasing knot sequence $u_0, \ldots, u_k$ of length $k+1$, $m+1$ control points $d_0, \ldots, d_m$ and a degree n satisfying [4]

$$\underbrace{k}_{\#\text{ of knots }-1} = \underbrace{n}_{\text{degree}} + \underbrace{m+1}_{\#\text{ of ctrl pts}} \tag{7}$$

and

$$s(x) = \sum_{i=0}^{m} d_i N_i^n(x)$$

where $(N_i^n)_{i=0}^m$ are the basis functions of the B-spline. These basis functions are defined by a recurrence relation,

$$N_i^0(x) = \begin{cases} 1 & \text{if } u_i \leq x < u_{i+1} \\ 0 & \text{else} \end{cases}$$

and

$$N_i^n(x) = \frac{x - u_i}{u_{i+n} - u_i} N_i^{n-1}(x) + \frac{u_{i+n+1} - x}{u_{i+n+1} - u_{i+1}} N_{i+1}^{n-1}(x).$$

Because each basis function N_i^n is nonzero on the interval $[u_i, u_{i+n}]$, called the support of the basis function, the B-spline is only defined for $x \in [u_n, u_{k-n}]$. Otherwise, x would lie on a part of the domain without a full basis. If x lies in the interval $[u_n, u_{k-n}]$, the curve is said to have full support. If r successive knots coincide, that is $u_i = u_{i+r-1}$, we say that the knot is of multiplicity r. At each knot, $s(x)$ is C^{n-r} where r is the multiplicity of the knot. A clamped B-spline is a B-spline where the first and last knot is of multiplicity $n+1$, and a common choice for u_0 and u_k is $u_0 = 0,\ u_k = 1$ which means that the domain for $s(x)$ is $[0, 1]$. Because $N_i^n \geq 0\ \forall i$, each point on the B-spline $s(x)$ lies in the convex hull of at most $n+1$ nearby control points [4]. The derivative of a basis function can be written as [4]

$$\frac{d}{dx} N_i^n(x) = \frac{n}{u_{n+i} - u_i} N_i^{n-1}(x) - \frac{n}{u_{n+i+1} - u_{i+1}} N_{i+1}^{n-1}(x) \tag{8}$$

which means that the derivative of the B-spline is

$$s'(x) = \sum_{i=0}^{m} d_i \frac{d}{dx} N_i^n(x). \tag{9}$$

Because B-splines have such convenient properties, as cited above, we use these splines to model the turbine blade. If the reader wishes to read more about B-splines, de Boor [3], Farin [4] and Piegl [11] to name a few are excellent sources.

3.2 B-spline interpolation

If we want the B-spline curve to satisfy $m+1$ conditions we will need to solve for $m+1$ unknowns. This can be done by letting the knot sequence, control points, degree or a combination of them act as the unknowns, but we are going to limit us to only having the control points $D = (d_0, \ldots, d_m)^\top$ as our unknowns. This has the advantage that we can reduce the interpolation to a simple matrix inversion problem for the equation $A_T D = C$ where A_T is

$$A_T = \begin{pmatrix} N_0^{n\,(q_0)}(t_0) & N_1^{n\,(q_0)}(t_0) & \ldots & N_m^{n\,(q_0)}(t_0) \\ N_0^{n\,(q_1)}(t_1) & N_1^{n\,(q_1)}(t_1) & \ldots & N_m^{n\,(q_1)}(t_1) \\ \vdots & \vdots & \ddots & \vdots \\ N_0^{n\,(q_m)}(t_m) & N_1^{n\,(q_m)}(t_m) & \ldots & N_m^{n\,(q_m)}(t_m) \end{pmatrix} \tag{10}$$

and

$$C = (\xi_0, \ldots, \xi_m)^\top$$

for the parameter vector $T = (t_0, \ldots, t_m)$, $u_n \leq t_i \leq u_{k-n}$ and where $0 \leq q_i < n$ indicates the derivative of N_i^n. Of course, the matrix A_T might not always be invertible. For $q_i = 0$ the following theorem is presented in de Boor [3]

Theorem 1 (Schoenberg-Whitney)**.** *The matrix $A_T = (A_j(t_i))$ defined in* (10) *with $q_i = 0$ is invertible if and only if*

$$A_i(t_i) \neq 0, \ i = 0, \ldots, m$$

i.e., if and only if $u_i < t_i < u_{i+n}\, \forall i$.

In which N_T does not contain any derivatives. A proof can be found in Powell [12]. From equation (8) we see that the derivatives of the basis functions are linear combinations of basis functions of lower degrees, which means that the theorem most likely can be extended to the case where we allow derivatives in A_T, though a proof of this is outside the scope of this paper. As we will see in the results, A_T will turn out to be invertible when we construct it. We saw that we could find a parameter vector that made A_T invertible for a given knot vector, but we can just as easily choose a knot vector that satisfies the theorem given the parameter vector. A method suggested by de Boor [3, p. 219] for a clamped B-spline defined on the interval $[a, b]$ is to average the knots according to

$$\begin{aligned} u_0 &= \cdots = u_n = a \\ u_{n+j} &= \frac{1}{n} \sum_{i=j}^{n+j-1} t_j \text{ for } j = 1, 2, \ldots, k-n \\ u_{k-n} &= \cdots = u_k = b \end{aligned} \tag{11}$$

which will create an invertible matrix A_T [3].

4 Parameter selection

If we want to solve the linear equation system $s^{(q_i)}(t_i) = \xi_i$, $0 \leq q_i < n$ we need a way to choose the parameter vector $T = (t_0, \ldots, t_m)$. In our case, the derivative conditions will only occur at the interpolation points P_j, that is where $q_i = 0$. This means that we can look at the simpler case $s(t_j) = P_j$ and then letting $t_i = t_j$ if $s^{(p)}(t_i)$ is the p-th derivative at $s(t_j)$. This will be enough as we only allow derivative conditions at the interpolation points P_i. We begin by introducing

the distance vector $H = (h_0, \ldots, h_{m-1})$ where $h_i = t_{i+1} - t_i$. Following Foley and Nielson [6] we can assume that $t_0 = 0$ without loss of generality which means that T is completely determined by H by the recurrence relation

$$\begin{aligned} t_0 &= 0 \\ t_{k+1} &= t_k + h_k. \end{aligned}$$

Three commonly used methods for choosing H for a given set of interpolation points $P = (P_0, \ldots, P_m)^\top$ are the uniform, chord length and centripetal methods. For the *uniform method* we simply let $h_i = 1$ which means that the distance between the t_i's are constant. For the *chord length method* we let

$$h_i = |P_{i+1} - P_i|$$

using the Euclidean distance as a metric. Both of these methods can produce poor results if, for example, the distance between the P_i's vary greatly, or if the data is badly scaled or changes direction abruptly [6]. The *centripetal method* is often used because it does not suffer from some of these problems and is defined as

$$h_i = |P_{i+1} - P_i|^\alpha$$

where a common choice for α is $1/2$. If we instead choose $\alpha = 0$ or 1 we see that we get the uniform and chord length methods respectively which means that the centripetal method is a generalization of these methods. The centripetal method is invariant under rotations, translations and equal scaling of the coordinates for $\alpha = 0, 1/2$ and 1 but, as shown by Foley and Nielson, the method is not scale invariant when the axes are scaled independently. Thus, the centripetal method is not affine invariant in general. Using an affine invariant metric introduced by Nielson [9], we can describe two affine invariant methods presented by Foley and Nielson as generally giving geometrically pleasing results [6].

Given $P = (P_0, \ldots, P_m)^\top$ where $P_i = (x_i, y_i)$, define the matrix G by

$$G = \begin{pmatrix} \frac{\sigma_Y}{g} & -\frac{\sigma_{XY}}{g} \\ -\frac{\sigma_{XY}}{g} & \frac{\sigma_X}{g} \end{pmatrix}$$

where

$$\begin{aligned} \sigma_X &= \frac{1}{m}\sum_{i=0}^{m}(x_i - \bar{x})^2, & \bar{x} &= \frac{1}{m}\sum_{i=0}^{m} x_i \\ \sigma_Y &= \frac{1}{m}\sum_{i=0}^{m}(y_i - \bar{y})^2, & \bar{y} &= \frac{1}{m}\sum_{i=0}^{m} y_i \end{aligned}$$

and

$$\begin{aligned} \sigma_{XY} &= \frac{1}{m}\sum_{i=0}^{m}(x_i - \bar{x})(y_i - \bar{y}) \\ g &= \sigma_X\sigma_Y - (\sigma_{XY})^2 \end{aligned}$$

If $U, V \in \mathbb{R}^2$, we define

$$M[P](U, V) = \sqrt{(U - V)Q(U - V)^\top}.$$

Then the metric $M[P](U,V)$ is affine invariant [9]. The first affine invariant method described by Foley and Nielson is the *affine invariant chord method* and defines the vector H as

$$h_i = M[P](P_i, P_{i+1})$$

though it satisfies the previous conditions on invariance, it does not take into account the local geometry of the curve. The second described affine invariant method is the *affine invariant angle method* and is defined by

$$h_i = d_i \left(1 + \frac{\delta\theta_i d_{i-1}}{d_{i-1}+d_i} + \frac{\delta\theta_{i+1} d_{i+1}}{d_i + d_{i+1}}\right)$$

where $d_i = M[P](P_i, P_{i+1})$, $\theta_i = \min(\alpha_i, \pi/2)$ and

$$\alpha_i = \pi - \arccos\left(\frac{d_{i-1}^2 + d_i^2 - M^2[P](P_{i-1}, P_{i+1})}{2d_i d_{i-1}}\right).$$

We are going to assume that $d_{-1} = d_{m+1} = 0$. Foley and Nielson shows that these two methods are affine invariant and suggests choosing $\delta = 1.5$ [6]. Because we have restricted our B-spline $s(x)$ to $x \in [0,1] \subset \mathbb{R}$ we need to scale our parameter vector T to the interval. This is easily done by calculating the total length of the distance vector as $L = \sum_i h_i$ and dividing each h_i by the length to get a new distance vector $\bar{H} = (\bar{h}_0, \ldots, \bar{h}_{m-1})$ where $\bar{h}_i = h_i/L$. The parameters then become

$$\begin{aligned} t_0 &= 0 \\ t_{i+1} &= t_i + \bar{h}_i \\ t_m &= 1. \end{aligned}$$

From now on, we are going to assume that T and H already have been scaled in the previous manner.

Our method of choosing the knot vector of the B-spline relies on the fact that we have as many parameter values as we have control points of the spline. This will not be the case if we have several derivative conditions on the spline for some of the parameters, for example if $s(t_i) = P_i$ and $s'(t_i) = v_i$, because the number of parameter values will only depend on the number of interpolation points P_i. We are going to deal with this problem by calculating the knot vector using a parameter vector where the number of times each parameter value occurs in the vector is equal to how many times it occurs in the spline conditions. To illustrate, if we have the conditions $s(t_i) = P_i,\ s^{(k_0)}(t_i) = D_{k_0}, \ldots, s^{(k_n)}(t_i) = D_{k_n}$ then the knot vector will be calculated from the parameter vector

$$(t_0, \ldots, \underbrace{t_i, \ldots, t_i}_{n+1}, \ldots, t_m).$$

This ensures that the length of the parameter vector used for the knot vector generation is of the same length as the number of control points of the spline.

5 Derivative conditions

According to Svensson [14] we need the slope of the curvature of the blade to be smooth. The curvature $\kappa(t)$ of a parametric function $s(t)$ in $\mathbb{R}^2$ is defined as [13]

$$\kappa(t) = \frac{\det\left(s'(t), s''(t)\right)}{|s'(t)|^3} \tag{12}$$

which means that the restrictions given by Svensson affect the fourth derivative of the curve. If we only allow simple interior knots for our B-spline $s(x)$, the spline will be C^4 on its interior if it is of degree 5. The method by Ye described in Svensson [14] requires us to match the slopes of the leading and trailing circular arcs at s_1, s_5, p_1 and p_4 for the B-splines approximating the suction and pressure sides respectively. This means that we require the B-splines to be G^1 continuous at their endpoints. Here, G^r continuity means geometric continuity of degree r and is defined as:

Def 1. *Two parametric curves meet with G^r continuity if the corresponding arc-length parametrizations of the curves meet with C^r continuity.*

By a theorem shown in Barsky and DeRose [1], two curves meet with G^1 continuity if and only if they have a common unit tangent vector. This means that the first derivative end conditions for the B-splines only depend on the direction of the derivative for the circular arcs at the points s_1, s_5, p_1 and p_4. In Section 2 we saw that the slopes at s_1, s_3, s_5, p_1 and p_4 were defined which means that we know the directions of the derivative vectors. The lengths of the vectors depend on the parametrization of the curve which means that we have to look at suitable parametrizations.

5.1 Endpoint derivatives

Let us begin by parametrizing the leading and trailing edge circular arcs as

$$\Omega_1(\theta_1(t)) = \big(R_1(1-\cos\theta_1(t)), H + R_1\sin\theta_1(t)\big)$$
$$\Omega_2(\theta_2(t)) = \big(B_x - R_2(1-\cos\theta_2(t)), R_2\sin\theta_2(t)\big)$$

where $\theta_1(t)$, $\theta_2(t)$ parametrizations for the circles. For θ_s, ϕ_s, θ_p and ϕ_p from section 2 we see that

$$s_1 = \Omega_1(\theta_s),\ s_5 = \Omega_2(\phi_s)$$
$$p_1 = \Omega_1(\theta_p),\ p_4 = \Omega_2(\phi_p)$$

which shows that the parametrizations are correct. If we differentiate Ω_1 and Ω_2 we get

$$\Omega_1'(\theta_1(t)) = R_1\big(\theta_1'(t)\sin\theta_1(t), \theta_1'(t)\cos\theta_1(t)\big)$$
$$\Omega_2'(\theta_2(t)) = R_2\big(\theta_2'(t)\sin\theta_2(t), \theta_2'(t)\cos\theta_2(t)\big)$$

which means that we can write the first derivative end conditions as Ω_1' and Ω_2' depending only on their parametrizations. If we let v_{s_1}, v_{s_5}, v_{p_1} and v_{p_4} be the first derivatives at s_1, s_5, p_1 and p_4 respectively, we can write them as

$$v_{s_1} = \Omega_1'(\theta_1(t_{s_1})),\ t_{s_1} \text{ s.t. } \theta_1(t_{s_1}) = \theta_s \tag{13}$$
$$v_{s_5} = \Omega_2'(\theta_2(t_{s_5})),\ t_{s_5} \text{ s.t. } \theta_2(t_{s_5}) = \phi_s \tag{14}$$
$$v_{p_1} = \Omega_1'(\theta_1(t_{p_1})),\ t_{p_1} \text{ s.t. } \theta_1(t_{p_1}) = \theta_p \tag{15}$$
$$v_{p_4} = \Omega_2'(\theta_2(t_{p_4})),\ t_{p_4} \text{ s.t. } \theta_2(t_{p_4}) = \phi_p \tag{16}$$

for some parametrizations θ_1, θ_2.

5.2 Derivative at s_3

From the calculations in subsection 2.2 we see that the angle between the slope at s_3 and the horizontal line through s_3 is $\beta_2 - \Gamma$ which means that the slope as s_3 is defined. Using this we

can describe the slope at s_3 as a vector $\tilde{v}_{s_3}$ of unit length

$$\tilde{v}_{s_3} = \big(\cos(\beta_2 - \Gamma), -\sin(\beta_2 - \Gamma)\big).$$

As before, we do not have any information on the length of the derivative vector at s_3. This means that we can write the first derivative at s_3 as

$$v_{s_3} = \nu_{s_3}\tilde{v}_{s_3} = \nu_{s_3}\big(\cos(\beta_2 - \Gamma), -\sin(\beta_2 - \Gamma)\big) \tag{17}$$

where $\nu_{s_3} = |v_{s_3}|$ is a real number.

6 Results

Using the theory we built up in the previous sections we will now divide the turbine blade into four parts, the leading and trailing edge circular arcs, and the suction and pressure sides. The circular arcs are easily described as parts of parametrized circles which means that we will focus on the construction of the suction and pressure sides. We will also introduce a point

$$s_2 = \left(\frac{x_{s_1} + x_{s_3}}{2}, \frac{x_{s_1} + x_{s_3}}{2}\lambda\right) \tag{18}$$

on the suction side between s_1 and s_3. The variable λ will be used to optimize the shape of the blade. To construct the suction side

$$S(t) = \sum_{i=0}^{m} P_i N_i^n(t)$$

connecting s_1 to s_5 we will need to decide on a degree n, a knot sequence $(u_0, \ldots, u_k)$ and control points $P_s = (P_0, \ldots, P_m)$. From the continuity conditions in section 5 we see that $S(t)$ must be of degree 5 and without loss of generality we can assume that $0 \leq t \leq 1$ for $S(t)$. We will choose $S(t)$ to be a clamped B-spline which means that the first and last knots will be of multiplicity 6. We make this choice because B-splines do not represent circular arcs well which means that a closed B-spline describing the whole blade will be bad at the leading and trailing edges. With a clamped B-spline we will be able to describe the suction and pressure side independently in an easy way. Thus, to have full support, we need at least 12 knots for our B-spline. We have 7 conditions on $S(t)$,

$$C_s = (s_1, v_{s_1}, s_2, s_3, v_{s_3}, s_5, v_{s_5}),$$

which means that we are going to have 7 control points. From the equality (7) we see that we then need $k = 7 + 5 = 12$, that is 13 knots. Because we have 4 zero-derivative conditions we will have the parameter vector

$$T_s = (t_0, t_1, t_2, t_3)$$

which we will construct using the conditions C_s and one of the methods described in Section 4. We can now construct the matrix

$$A_{T_s} = \begin{pmatrix} N_0^5(t_0) & N_1^5(t_0) & \ldots & N_6^5(t_0) \\ N_0^{5\prime}(t_0) & N_1^{5\prime}(t_0) & \ldots & N_6^{5\prime}(t_0) \\ N_0^5(t_1) & N_1^5(t_1) & \ldots & N_6^5(t_1) \\ N_0^5(t_2) & N_1^5(t_2) & \ldots & N_6^5(t_2) \\ N_0^{5\prime}(t_2) & N_1^{5\prime}(t_2) & \ldots & N_6^{5\prime}(t_2) \\ N_0^5(t_3) & N_1^5(t_3) & \ldots & N_6^5(t_3) \\ N_0^{5\prime}(t_3) & N_1^{5\prime}(t_3) & \ldots & N_6^{5\prime}(t_3) \end{pmatrix}$$

if we use the averaging method suggested by de Boor for the interior knot. The last thing we need to do is to solve the equation system

$$A_{T_s} P_s = C_s$$

in order to get the control points. This process is described in the following algorithm:

1. Decide on a value λ
2. Construct s_1, s_2, s_3, s_5, p_1 and p_4 using equations (2), (18), (4), (3), (5) and (6).
3. Construct the parameter vector T_s and the knot vector.
4. Choose a parametrization for Ω_1 and Ω_2 and construct the end point derivatives v_{s_1} and v_{s_5}.
5. Choose a value ν_{s_3} and construct v_{s_3}.
6. Construct the basis functions $N_i^5(t)$ and their derivatives $N_i^{5(q_i)}(t)$.
7. Construct the matrix A_{T_s}, the condition vector C_s and solve $A_{T_s} P_s = C_s$
8. Return the B-spline
$$S(t) = \sum_{i=0}^{6} P_i N_i^5(t)$$

The method to construct the pressure side will be similar, but as we only have 4 conditions $p_1, v_{p_1}, p_4, v_{p_4}$ we will need 2 extra condition to satisfy the equality (7). To do this we will introduce the conditions a_{p_1}, a_{p_4} on the second derivatives at the endpoints. From this algorithm we see that we need to decide on the conditions described in Figure 3, a value λ for s_2, a parameter vector T_s for $S(t)$, ν_{s_3} for s_3, parametrizations for Ω_1 and Ω_2 and second derivatives for $P(t)$. Because these are not dependent on any previous calculations in the algorithm, they can be decided beforehand which simplifies the construction of the algorithm. By implementing the algorithm in Python we can look at the resulting parametrized blades we get from some different choices of the previous values. Throughout the examples we will use the following 11 conditions on the blade given by Svensson [14]:

$$\begin{aligned} R_1 &= .0555 \\ R_2 &= .0220 \\ \gamma &= 29.2 \\ \beta_1 &= 51.5 \\ \beta_2 &= 74.08^1 \\ \Delta\beta_1 &= 30 \\ \Delta\beta_2 &= 2 \\ \tau &= 1.1817 \\ \beta_g &= 75.4075 \\ \Gamma &= 8 \\ B_x &= 1 \end{aligned}$$

[1] Svensson states that it should be 15.92 but this is incorrect.

6.1 Basic choices

We will begin by making the simplest choices possible. We begin by choosing the parametrizations $\theta_1(t) = t$ and $\theta_2(t) = t$ for Ω_1 and Ω_2. This means that $\Omega_1'(t) = R_1\big(\sin(t), \cos(t)\big)$ and $\Omega_2'(t) = R_2\big(\sin(t), \cos(t)\big)$ and $|v_{s_1}| = |v_{p_1}| = R_1$ and $|v_{s_5}| = |v_{p_4}| = R_2$. We let $\nu_{s_3} = 1$ which means that $|v_{s_3}| = 1$ and let $a_{p_1} = a_{p_4} = (1, 1)$. We use the uniform method to get the parameter vector $T_s = (0, 1/3, 2/3, 1)$. Even though the leading and trailing edges can be generated using their parametrizations Ω_1 and Ω_2, use two NURBS[1] to describe them instead. We now have λ left which we will use to optimize the shape of the blade. As we see in Figure 5, the blade does not

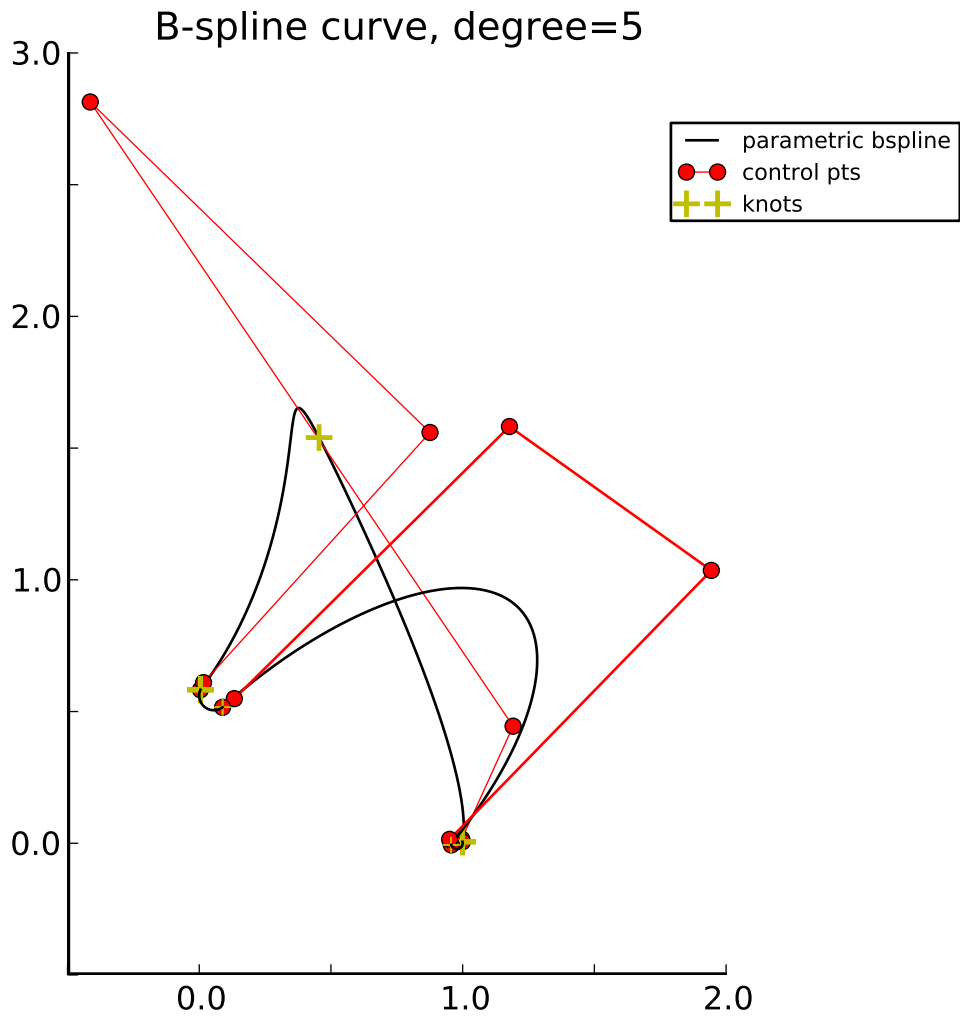

Figure 5: A B-spline curve satisfying the conditions on the suction side of the blade.

look good at all even though we chose $\lambda = 4.4$ which experimentally produced the best result. It is important to realize that this curve, even though it does not look nearly as good as it should, satisfies the conditions given on the blade. This means that even though we can easily meet the

[1]NURBS are a generalization of B-spline where each control point gets a weight value assigned to it. This means that, among other things, it can be used to describe conic sections [4, 5].

conditions on the curve, by making poor decisions when choosing the other values we can get a curve that is not good enough for our purposes.

6.2 Modifying v_{s_3}

Clearly, the previous choices were not good enough. If we look at the derivatives of the curve we see that v_{s_3} is much longer than the other derivatives. Instead of letting $\nu_{s_3} = 1$, we will set it to a linear combination of $|v_{s_1}| = R_1$ and $|v_{s_5}| = R_2$ at the parameter value t_2 which corresponds to s_3.

$$\nu_{s_3} = (1 - t_2)R_1 + t_2 R_2.$$

We also need to change the values of a_{p_1} and a_{p_4}. We use Ω_1 and Ω_2 and let $a_{p_1} = \Omega_1''$ and $a_{p_4} = \Omega_2''$ Still using the uniform distribution for the suction and pressure sides, we get the blade seen in Figure 6 for $\lambda = 3.4$. This is not a good result, even if it is an improvement from the

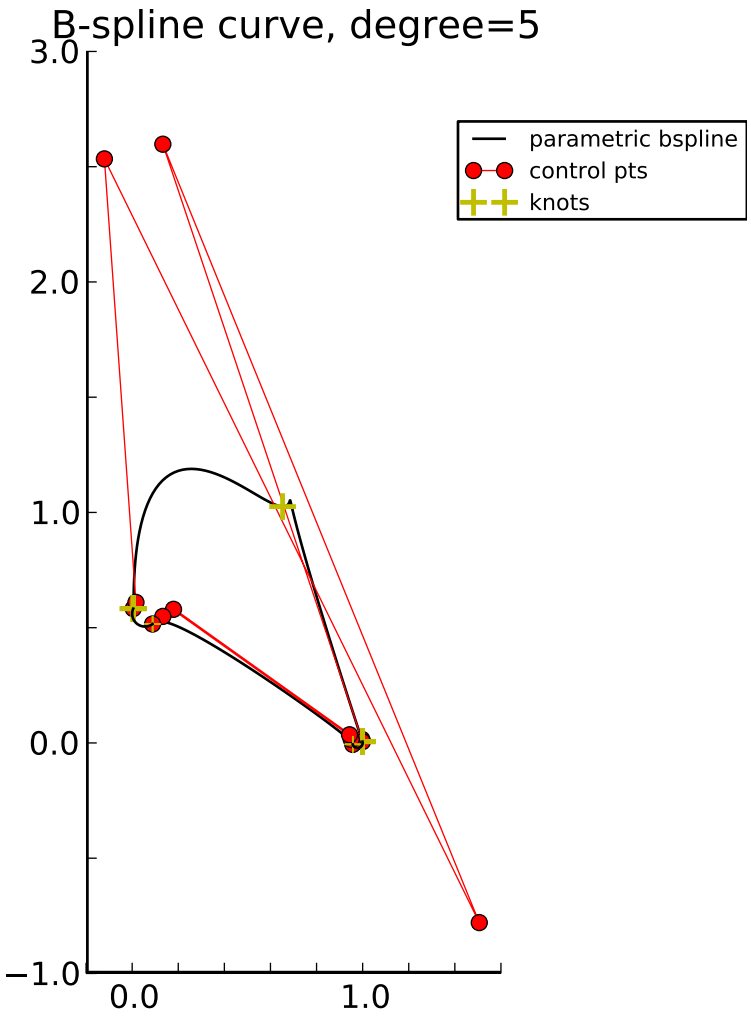

Figure 6: A B-spline curve satisfying the conditions on the suction side of the blade setting v_{s_3} as an average of R_1 and R_2.

last figure. We can clearly see that something really strange is going on around s_3. This does not disappear by using different methods for choosing the parameter values. For example, in Figure 7 we use the centripetal method to choose the parameter values.

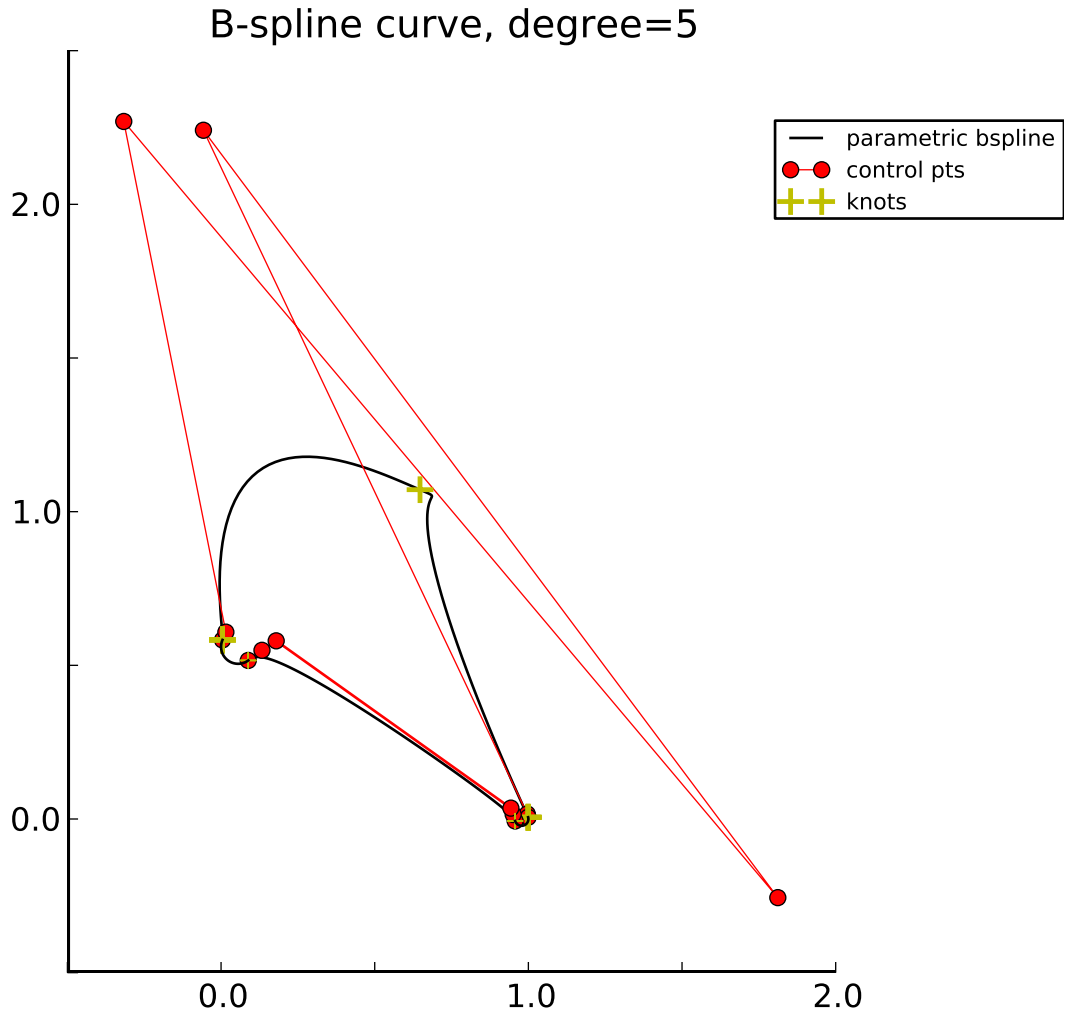

Figure 7: A B-spline curve satisfying the conditions on the suction side of the blade setting v_{s_3} as an average of R_1 and R_2 using the centripetal method to choose parameter values.

6.3 New parametrizations for Ω_1 and Ω_2

In Figures 5-7 it looks like there are only 5 control points of the curve even though there should be 7. In fact, if we look closely, there are 2 control points in the beginning of the curve and 2 in the end, but they are clustered together. This indicates that the derivatives v_{s_1} and v_{s_5} might be bad. If we look at the parametrizations of Ω_1 and Ω_2 we see that the circle does a full rotation as t goes from 0 to 2π. But $S(t)$ goes from s_1 to s_5 as t goes from 0 to 1. This could be the problem. If we instead think of the curve from p_1 to p_4 along the circular arcs and the suction side and that it does this as t goes from 0 to 1 then the interval for t when it is on

each circular arc is much shorter than the previous parametrization. This suggests again that we should change parametrization. If we introduce parameters $t_{-1}, t_0, t_1, t_2, t_3, t_4$ on the interval $(0, 1)$ where $t_{-1} = 0$, $t_4 = 1$, and the parameters correspond to the points along the curve, we can use the parameter selection methods to decide the new parameters. Let us also introduce $k_1 = t_0 - t_{-1}$ and $k_2 = t_4 - t_3$. Then we want a parametrization of Ω_1 that satisfies the following conditions:

$$\begin{aligned} \Omega_1\theta_1(a) &= p_1 \\ \Omega_1\theta_1(b) &= s_1 \\ b - a &= k_1 \end{aligned}$$

and similarly for Ω_2. Such parametrizations would be

$$\begin{aligned} \theta_1(t) &= \frac{2\pi t}{k_1} \\ \theta_2(t) &= \frac{2\pi t}{k_2} \end{aligned}$$

which gives us the following derivatives:

$$\begin{aligned} v_{s_1} &= R_1 \frac{2\pi}{k_1}\big(\sin\theta_s, \cos\theta_s\big) \\ v_{s_5} &= R_2 \frac{2\pi}{k_2}\big(\sin\phi_s, \cos\phi_s\big) \end{aligned}$$

if we use these together with $|v_{s_3}|$ as an average of $|v_{s_1}|$ and $|v_{s_5}|$ and the affine angle method of choosing the parameter values, and the uniform method to get k_1 and k_2 we get the curve in Figure 8. Notice that k_1 and k_2 are constructed two times, once for the suction side and once for the pressure side. For the pressure side we use k_1 and k_2 to construct v_{p_1} and v_{p_4} but a_{p_1} and a_{p_4} will be kept as before. This curve looks remarkably good but the control points in the middle looks askew. Let us instead choose the parametrization

$$\begin{aligned} \theta_1(t) &= \frac{2\pi t}{5k_1} \\ \theta_2(t) &= \frac{2\pi t}{5k_2} \end{aligned}$$

where we divide by the heuristic choice of the degree of the spline. Using this method, where we construct k_1 and k_2 using the affine chord method, and the parameter values using the affine angle method, for $\lambda = 3.4$ we obtain the curve in Figure 9 which is a remarkably good result. Using this method we can use λ to change the shape of the curve and still get good results which might make it possible to use λ as an optimization parameter.

6.4 Increasing the number of control points

Though the last shape was good, the control polygon did not follow the suction side very closely. Just as we introduced a_{p_1} and a_{p_4} for the pressure side, we can introduce a_{s_1} and a_{s_5} as conditions on the second derivatives at the endpoints. This means that we have the condition vector

$$C_s = (s_1, v_{s_1}, a_{s_1}, s_2, s_3, v_{s_3}, s_5, v_{s_5}, a_{s_5})$$

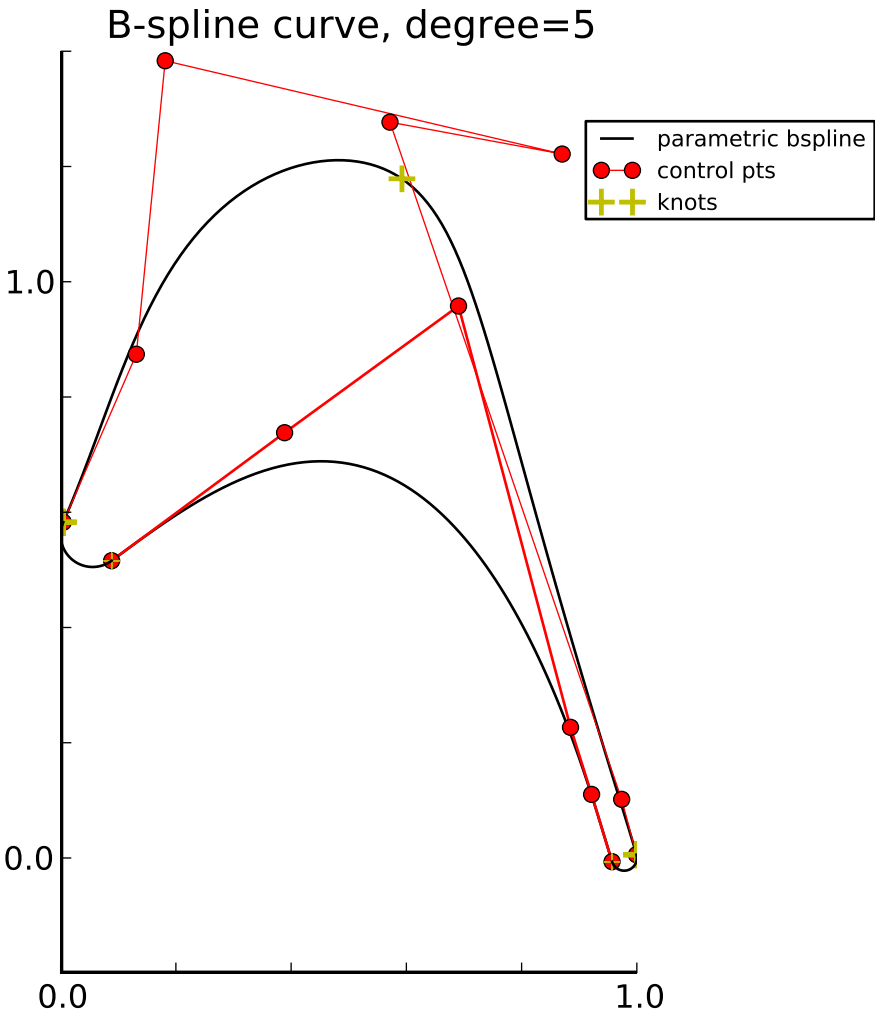

Figure 8: A B-spline curve satisfying the conditions on the suction side of the blade where v_{s_1} and v_{s_5} are chosen by finding k_1 and k_2.

and 9 control points. Letting

$$a_{s_1} = R_1 \frac{\Omega_1}{|\Omega_1|}$$
$$a_{s_5} = R_2 \frac{\Omega_2}{|\Omega_2|}$$

and using the previous procedure for this condition vector we get the blade in Figure 10 which shows that we can add more conditions and still get a nice curve.

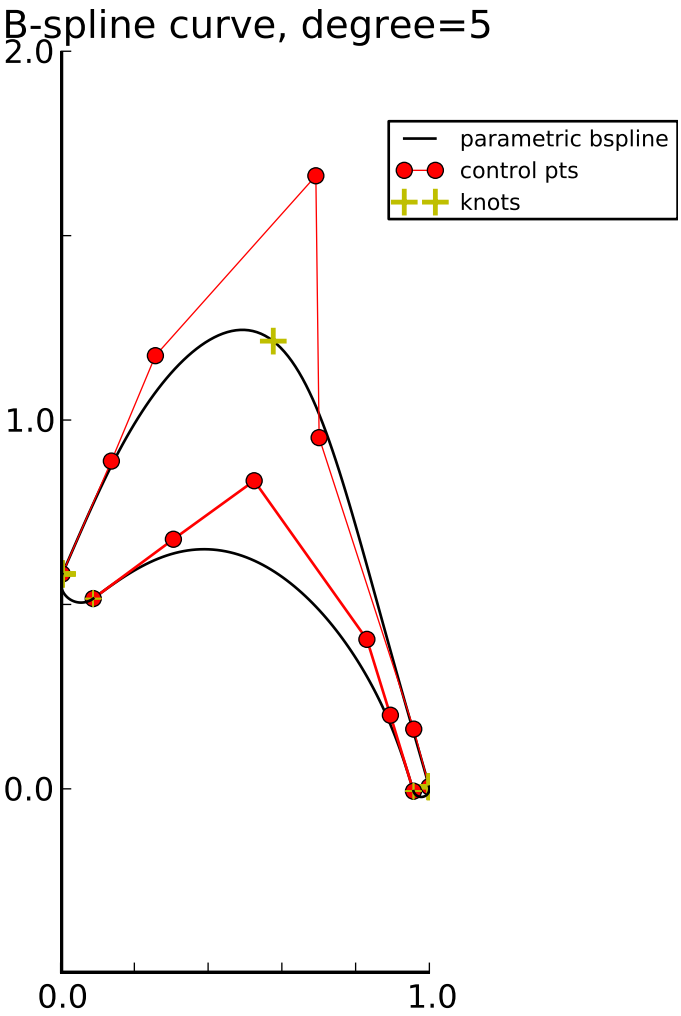

Figure 9: A B-spline curve satisfying the conditions on the suction side of the blade where v_{s_1} and v_{s_5} are chosen by finding k_1 and k_2 with really good results.

7 Conclusions

In this paper we have built up a theory describing the geometry of the blade and how to choose the correct B-spline to fit the geometric conditions. In the end we got a result that was satisfactory. Throughout the results in the previous section we saw that the matrices A_{T_s} and A_{T_p} were invertible in our code which suggests that the Schoenberg-Whitney theorem might hold true for $q > 0$. A proof of this is not in the scope of this paper though but an interested reader might find this a rewarding challenge to prove. We have only demanded G^1 continuity at the points s_1, s_5, p_1 and p_4 but increasing this to G^2 or higher might lead to curves that work better in the simulations. This too is outside the scope of this paper. We saw in the last part of the results that we got a nice looking curve depending only on the parameter λ controlling the height of s_2. By using this as a way of optimizing the curve, one could extend this theory as an alternative to the methods defined in Ye [16] and Svensson [14] in turbine analysis. Using a B-spline instead

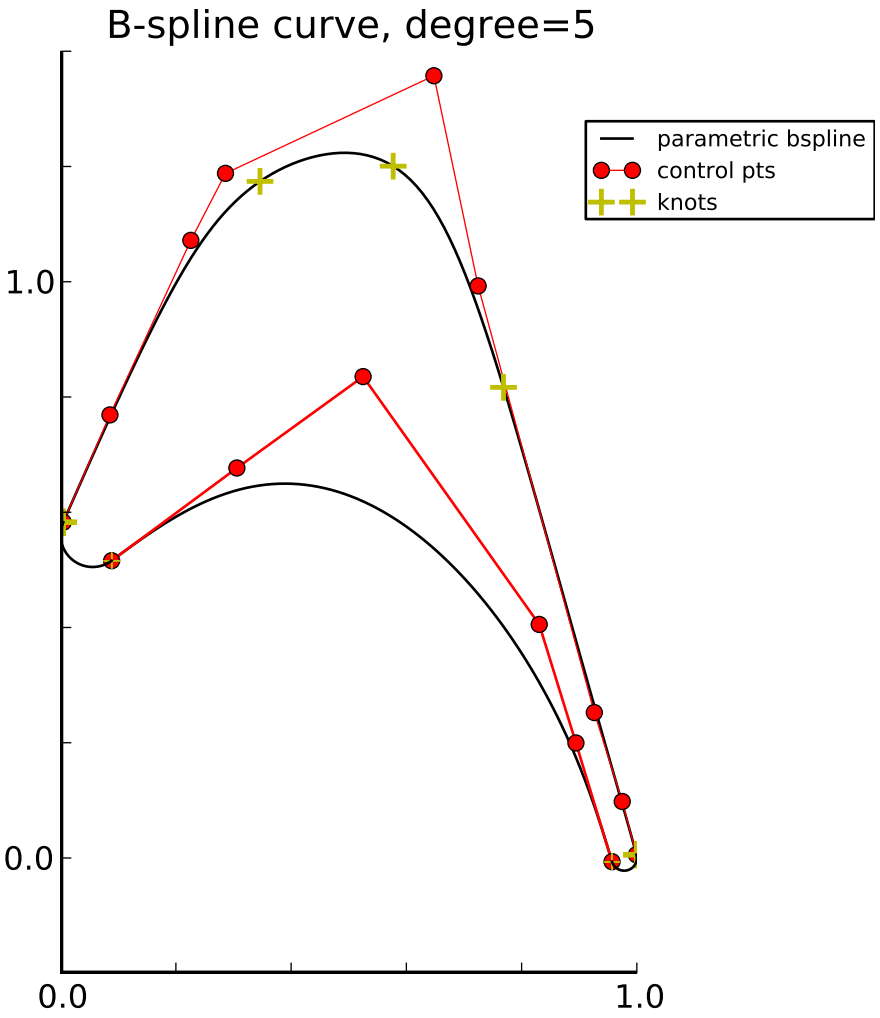

Figure 10: A B-spline curve satisfying the conditions on the suction side of the blade where v_{s_1} and v_{s_5} are chosen by finding k_1 and k_2 and with 2 more conditions a_{s_1} and a_{s_5}.

of polynomials [16] or Bezier curves [14] to describe the suction and pressure sides requires less interior conditions because we do not need to divide the sides into several parts and demanding the derivatives to meet continuously at these points. B-splines are not good at representing conic sections but as we saw in the results the leading and trailing edges could be represented as two NURBS, a generalization of B-splines. This means that one might be able to extend the results presented here to NURBS and describe the whole blade as a closed NURBS.

A Code

The code was written in Python and used NumPy [10], SciPy [8] and Matplotlib [7] extensively. For the B-splines a library developed by Oliver Borm and forked and extended to github by Sebastian Eiser called PyNURBS [15] was used. The code written for this paper relies on the use of a B-spline library but can easily be modified to fit your own B-spline code.

A.1 `turbine.py`

This code constructs the curves and uses the code in `util.py`

```
# -*- coding: utf-8 -*-
from __future__ import division
import scipy as sp
import numpy as np
import matplotlib.pyplot as plt
import util
from math import radians, pi
from spline import Spline, SplineError, Arc

class SuctionSide(Spline):
    """
    Construct a B-spline satisfying the constraints put on the turbine
        blade

 Input:

    parameters
    ----------
    11 turbine parameters beta1, beta2, GAMMA, tau, betag, gamma, Bx, R1,
         R2, dbeta1, dbeta2
    chosen for their strong influence on the blade.

    tMethod
    -------
    The method for choosing the interpolation parameters. Can be
    ('uniform',)
    ('chordlength',)
    ('centripetal', alpha) #where alpha is a number. If not chosen then
        0.5 is used.
    ('affinechord',)
    ('affineangle', alpha) #where alpha is a number. If not chosen then
        1.5 is used.

    derMethod
    ---------
    Which method to use to create the derivative conditions. Can be
    'radius'
    'fastparam'
    'param'

    dertMethod
    ----------
```

```
    If h1 and h2 are needed for the derMethod, choose a method to produce
        h1 and h2. Can be
    ('uniform',)
    ('chordlength',)
    ('centripetal', alpha) #where alpha is a number. If not chosen then
       0.5 is used.
    ('affinechord',)
    ('affineangle', alpha) #where alpha is a number. If not chosen then
       1.5 is used.

    vs3Method
    ---------
    The method to choose vs3. Can be
    alpha #where alpha is a number, either a float or an integer. Will be
        the length of the vector.
    'average'

    der2
    ----
    If the second derivative should be included. Can be
    True
    False

    lambd
    -----
    The value that constructs s2. Can be used as an optimization
       parameter.

Output:

    A B-spline of degree 5

    """
    def __init__(self, parameters, tMethod, derMethod, dertMethod,
       vs3Method, der2, lambd):

        beta1, beta2, GAMMA, tau, betag, gamma, Bx, R1, R2, dbeta1,
            dbeta2 = parameters

        # Convert the angles to radians
        beta1 = radians(beta1); beta2 = radians(beta2);dbeta1 = radians(
            dbeta1)
        dbeta2 = radians(dbeta2); betag = radians(betag); gamma = radians
            (gamma)
        GAMMA = radians(GAMMA)

        chord = (Bx - R1*(1 - sp.cos(beta1)) - R2*(1 - sp.cos(beta2)))/sp
            .cos(gamma)
        height = chord*sp.sin(gamma) + R1*sp.sin(beta1) - R2*sp.sin(beta2
            )

        s1 = np.array((R1*(1 - sp.sin(dbeta1/2 + beta1)), height + R1*sp.
            cos(dbeta1/2 + beta1)))
```

```
s3 = np.array((Bx - (tau*sp.cos(betag) + R2)*sp.sin(beta2-GAMMA)
    - R2, tau - (tau*sp.cos(betag) + R2)*sp.cos(beta2-GAMMA)))
s5 = np.array((Bx - R2*(1 - sp.sin(beta2 + dbeta2/2)), R2*sp.cos(
    beta2 + dbeta2/2)))

s2 = np.array(((s1[0]+s3[0])/2,lambd*(s1[0]+s3[0])/2))

p1 = (R1*(1 + sp.sin(beta1 - dbeta1/2.)), height - R1*sp.cos(
    beta1 - dbeta1/2.))
p4 = (Bx - R2*(1 + sp.sin(beta2 - dbeta2/2.)), -R2*sp.cos(beta2 -
     dbeta2/2.))

newparam = util.spacing(np.array([p1, s1, s2, s3, s5, p4]), (0,
    1), *dertMethod)
tParam = util.spacing(np.array([s1, s2, s3, s5]), (0, 1), *
    tMethod)

if der2 == True:
    tParam9 = [0,0,0,tParam[1],tParam[2],tParam[2],1,1,1]
else:
    tParam9 = [0,0,tParam[1],tParam[2],tParam[2],1,1]
knots = util.knotAveraging(tParam9, 5, (0,1))

if derMethod == 'radius':
    vs1 = R1*np.array((sp.cos(dbeta1/2 + beta1), sp.sin(dbeta1/2
        + beta1)))
    vs5 = R2*np.array((sp.cos(dbeta2/2 + beta2), -sp.sin(dbeta2/2
         + beta2)))

elif derMethod == 'fastparam':
    h1, h2 = newparam[1]-newparam[0], newparam[-1]-newparam[-2]
    vs1 = R1*(np.pi-beta1)/h1*np.array((sp.cos(dbeta1/2 + beta1),
         sp.sin(dbeta1/2 + beta1)))
    vs5 = R2*(np.pi-beta2)/h2*np.array((sp.cos(dbeta2/2 + beta2),
         -sp.sin(dbeta2/2 + beta2)))

elif derMethod == 'param':
    h1, h2 = newparam[1]-newparam[0], newparam[-1]-newparam[-2]
    vs1 = R1*(np.pi-beta1)/h1/5.*np.array((sp.cos(dbeta1/2 +
        beta1), sp.sin(dbeta1/2 + beta1)))
    vs5 = R2*(np.pi-beta2)/h2/5.*np.array((sp.cos(dbeta2/2 +
        beta2), -sp.sin(dbeta2/2 + beta2)))
else:
    raise SplineError, "derMethod must be one of 'radius', '
        fastparam' or 'param'."

if type(vs3Method) in (int, float):
    vs3 = vs3Method*np.array([sp.cos(beta2 - GAMMA),-sp.sin(beta2
         - GAMMA)])
elif vs3Method == 'average':
    leng = (1 - tParam[2])*np.linalg.norm(vs1) + tParam[2]*np.
        linalg.norm(vs5)
```

```
        vs3 = leng*np.array([sp.cos(beta2 - GAMMA),-sp.sin(beta2 -
            GAMMA)])
    else:
        raise SplineError, "vs3Method must be either a number or '
            average'."

    if der2 == True:
        as1 = np.array((sp.sin(dbeta1/2 + beta1), -sp.cos(dbeta1/2 +
            beta1)))/20.
        as5 = np.array((-sp.sin(dbeta2/2 + beta2), -sp.cos(dbeta2/2 +
             beta2)))/20.

    if der2 == True:
        conditions = np.array([s1,vs1,as1,s2,s3,vs3,s5,vs5,as5])
    else:
        conditions = np.array([s1,vs1,s2,s3,vs3,s5,vs5])

    basislist = util.basisarray(5, knots)
    der0 = basislist[-1]
    der1 = util.basisderivative(1, basislist, knots)

    if der2 == True:
        der2 = util.basisderivative(2, basislist, knots)
        rang = 9
        probmatrix = np.array([
        [der0[i](tParam[0]) for i in range(rang)],
        [der1[i](tParam[0]) for i in range(rang)],
        [der2[i](tParam[0]) for i in range(rang)],
        [der0[i](tParam[1]) for i in range(rang)],
        [der0[i](tParam[2]) for i in range(rang)],
        [der1[i](tParam[2]) for i in range(rang)],
        [der0[i](tParam[3]) for i in range(rang)],
        [der1[i](tParam[3]) for i in range(rang)],
        [der2[i](tParam[3]) for i in range(rang)],
        ])
    else:
        rang = 7
        probmatrix = np.array([
        [der0[i](tParam[0]) for i in range(rang)],
        [der1[i](tParam[0]) for i in range(rang)],
        [der0[i](tParam[1]) for i in range(rang)],
        [der0[i](tParam[2]) for i in range(rang)],
        [der1[i](tParam[2]) for i in range(rang)],
        [der0[i](tParam[3]) for i in range(rang)],
        [der1[i](tParam[3]) for i in range(rang)],
        ])

    xpts = np.linalg.solve(probmatrix,conditions[:,0])
    ypts = np.linalg.solve(probmatrix,conditions[:,1])

    self.conditions = conditions
    self.parameters = (beta1, beta2, GAMMA, tau, betag, gamma, Bx, R1
        , R2, dbeta1, dbeta2)
```

```
        super(SuctionSide, self).__init__(np.array((xpts,ypts)), knots)

    @classmethod
    def functionOfLambda(cls, parameters, tMethod, derMethod, dertMethod,
         vs3Method, der2):
        """
        Constructs a function that depends on lambda. Can be used to
            optimize the shape of the B-spline
        """
        def suctionside(l):
            return cls(parameters, tMethod, derMethod, dertMethod,
                vs3Method, der2, l)
        return suctionside

class PressureSide(Spline):
    """
    Construct a B-spline satisfying the constraints put on the turbine
        blade

 Input:

    parameters
    ----------
    11 turbine parameters beta1, beta2, GAMMA, tau, betag, gamma, Bx, R1,
         R2, dbeta1, dbeta2
    chosen for their strong influence on the blade.

    derMethod
    ---------
    Which method to use to create the derivative conditions. Can be
    'radius'
    'fastparam'
    'param'

    dertMethod
    ----------
    If h1 and h2 are needed for the derMethod, choose a method to produce
         h1 and h2. Can be
    ('uniform',)
    ('chordlength',)
    ('centripetal', alpha) #where alpha is a number. If not chosen then
        0.5 is used.
    ('affinechord',)
    ('affineangle', alpha) #where alpha is a number. If not chosen then
        1.5 is used.

    der2Method
    ----------
    The method for choosing the second derivatives. Can be
    'constant'
    'radius'
```

```
Output:

    A B-spline of degree 5

    """

    def __init__(self, parameters, derMethod, dertMethod, der2Method):

        beta1, beta2, GAMMA, tau, betag, gamma, Bx, R1, R2, dbeta1,
            dbeta2 = parameters

        beta1 = radians(beta1); beta2 = radians(beta2);dbeta1 = radians(
            dbeta1)
        dbeta2 = radians(dbeta2); betag = radians(betag); gamma = radians
            (gamma)
        GAMMA = radians(GAMMA)

        chord = (Bx - R1*(1 - sp.cos(beta1)) - R2*(1 - sp.cos(beta2)))/sp
            .cos(gamma)
        height = chord*sp.sin(gamma) + R1*sp.sin(beta1) - R2*sp.sin(beta2
            )

        s1 = np.array((R1*(1 - sp.sin(dbeta1/2 + beta1)), height + R1*sp.
            cos(dbeta1/2 + beta1)))
        s5 = np.array((Bx - R2*(1 - sp.sin(beta2 + dbeta2/2)), R2*sp.cos(
            beta2 + dbeta2/2)))

        p1 = (R1*(1 + sp.sin(beta1 - dbeta1/2.)), height - R1*sp.cos(
            beta1 - dbeta1/2.))
        p4 = (Bx - R2*(1 + sp.sin(beta2 - dbeta2/2.)), -R2*sp.cos(beta2 -
             dbeta2/2.))

        newparam = util.spacing(np.array((s5,p4,p1,s1)), (0,1), *
            dertMethod)
        tParam = (0,1)
        knots = np.array([0,0,0,0,0,0,1,1,1,1,1,1])

        if derMethod == 'radius':
            vp1 = R1*np.array([-sp.cos(beta1 - dbeta1/2.), -sp.sin(beta1
                - dbeta1/2.)])
            vp4 = R2*np.array([-sp.cos(beta2 - dbeta2/2.), sp.sin(beta2 -
                 dbeta2/2.)])
        elif derMethod == 'fastparam':
            h2, h1 = newparam[1]-newparam[0], newparam[-1]-newparam[-2]
            vp1 = R1*(np.pi-beta1)/h1*np.array([-sp.cos(beta1 - dbeta1
                /2.), -sp.sin(beta1 - dbeta1/2.)])
            vp4 = R2*(np.pi-beta2)/h2*np.array([-sp.cos(beta2 - dbeta2
                /2.), sp.sin(beta2 - dbeta2/2.)])
        elif derMethod == 'param':
            h2, h1 = newparam[1]-newparam[0], newparam[-1]-newparam[-2]
            vp1 = R1*(np.pi-beta1)/h1/5.*np.array([-sp.cos(beta1 - dbeta1
                /2.), -sp.sin(beta1 - dbeta1/2.)])
```

```
            vp4 = R2*(np.pi-beta2)/h2/5.*np.array([-sp.cos(beta2 - dbeta2
                /2.), sp.sin(beta2 - dbeta2/2.)])
        else:
            raise SplineError, "derMethod must be one of 'radius', '
                fastparam' or 'param'."

        if der2Method == 'constant':
            ap1 = np.array([1.,1.])
            ap4 = np.array([1.,1.])
        elif der2Method == 'radius':
            ap1 = R1/20.*np.array((sp.sin(beta1 - dbeta1/2.), -sp.cos(
                beta1 - dbeta1/2.)))
            ap4 = R2/20.*np.array((-sp.sin(beta2 - dbeta2/2.), -sp.cos(
                beta2 - dbeta2/2.)))
        else:
            raise SplineError, "der2Method must be 'constant' or 'radius
                '."

        conditions = np.array([p4, vp4, ap4, p1, vp1, ap1])

        basislist = util.basisarray(5, knots)
        der0 = basislist[-1]
        der1 = util.basisderivative(1, basislist, knots)
        der2 = util.basisderivative(2, basislist, knots)

        rang = 6
        probmatrix = np.array([
        [der0[i](tParam[0]) for i in range(rang)],
        [der1[i](tParam[0]) for i in range(rang)],
        [der2[i](tParam[0]) for i in range(rang)],
        [der0[i](tParam[1]) for i in range(rang)],
        [der1[i](tParam[1]) for i in range(rang)],
        [der2[i](tParam[1]) for i in range(rang)]
         ])

        xpts = np.linalg.solve(probmatrix,conditions[:,0])
        ypts = np.linalg.solve(probmatrix,conditions[:,1])

        self.conditions = conditions
        self.parameters = (beta1, beta2, GAMMA, tau, betag, gamma, Bx, R1
            , R2, dbeta1, dbeta2)

        super(PressureSide, self).__init__(np.array((xpts,ypts)), knots)

def leadingEdge(parameters):
    """
    Constructs the leading edge circular arc using a NURBS.

 Input:

    parameters
    ----------
```

```
    11 turbine parameters beta1, beta2, GAMMA, tau, betag, gamma, Bx, R1,
        R2, dbeta1, dbeta2
    chosen for their strong influence on the blade.

 Output:

    A NURBS of degree 3
    """
    beta1, beta2, GAMMA, tau, betag, gamma, Bx, R1, R2, dbeta1, dbeta2 =
        parameters

    beta1 = radians(beta1); beta2 = radians(beta2);dbeta1 = radians(
        dbeta1)
    dbeta2 = radians(dbeta2); betag = radians(betag); gamma = radians(
        gamma)
    GAMMA = radians(GAMMA)

    chord = (Bx - R1*(1 - sp.cos(beta1)) - R2*(1 - sp.cos(beta2)))/sp.cos
        (gamma)
    height = chord*sp.sin(gamma) + R1*sp.sin(beta1) - R2*sp.sin(beta2)

    leadingedge = Arc(R1, (R1, height), pi/2. + beta1 + dbeta1/2., beta1
        - pi/2. - dbeta1/2.)
    return leadingedge

def trailingEdge(parameters):
    """
    Constructs the trailing edge circular arc using a NURBS.

 Input:

    parameters
    ----------
    11 turbine parameters beta1, beta2, GAMMA, tau, betag, gamma, Bx, R1,
        R2, dbeta1, dbeta2
    chosen for their strong influence on the blade.

 Output:

    A NURBS of degree 3
    """
    beta1, beta2, GAMMA, tau, betag, gamma, Bx, R1, R2, dbeta1, dbeta2 =
        parameters

    beta1 = radians(beta1); beta2 = radians(beta2);dbeta1 = radians(
        dbeta1)
    dbeta2 = radians(dbeta2); betag = radians(betag); gamma = radians(
        gamma)
    GAMMA = radians(GAMMA)

    trailingedge = Arc(R2, (Bx - R2,0), dbeta2/2. - pi/2. - beta2, pi/2.
        - beta2 - dbeta2/2.)
    return trailingedge
```

```
if __name__ == "__main__":
    inputData = [51.5, 90-15.92, 8., 1.1817, 75.4075, 29.2, 1., .0555,
        .0220, 30., 2.0]
    suctionside = SuctionSide.functionOfLambda(inputData, ('affineangle'
        ,1.5), 'param', ('affinechord',), 'average', False)
    pressureside = PressureSide(inputData, 'param', ('centripetal',0.9),
        'radius')
    pressureside.plot2D(100, False)
    leadingedge = leadingEdge(inputData)
    trailingedge = trailingEdge(inputData)
    leadingedge.plot2D(100, False)
    trailingedge.plot2D(100, False)
    print(suctionside)
    suctionside(3.4).plot2D(100, False)
    ax = plt.gca()
    ax.set_aspect('equal')
    plt.show()
```

A.2 util.py

```
# -*- coding: utf-8 -*-
from __future__ import division
import numpy as np
import spline

def basisarray(degree, knots):

    def knotfraction(u0, u1, u2, u3):
        if u2 == u3: return 0
        return (u0 - u1)/(u2 - u3)

    def basisfunc(x, i, deg):
        if deg == 0:
            if knots[i + 1] == knots[-1]:
                return float(knots[i] <= x <= knots[i + 1])
            if knots[i] == knots[i + 1]: return 0.
            return float(knots[i] <= x < knots[i + 1])
        return knotfraction(x,knots[i], knots[i + deg], knots[i])*
            basisfunc(
        x, i, deg - 1) + knotfraction(knots[i + deg + 1], x, knots[i +
            deg + 1], knots[i + 1]
        )*basisfunc(x, i + 1, deg - 1)
    basislist = [[(lambda x, i = j, deg = degr: basisfunc(x, i, deg)) for
         j in range(len(knots) - 1 - degr)] for degr in range(degree + 1)]
    return basislist

def basisderivative(deriv, basislist, knots):
    degree = len(basislist) - 1
    if type(deriv) != int:
        raise TypeError, "Deriv has to be an integer"
    if deriv < 1:
```

```
            raise TypeError, " Deriv has to be 1 or larger"
        basisfuncs = basislist[-1 - deriv]

        def knotfraction(deg, u1, u2):
            if u1 == u2: return 0
            return 1/float(u1 - u2)

        def derivative(x, i, deg, der):

            if der == 1:
                return  knotfraction(deg, knots[i + deg], knots[i])*
                    basisfuncs[i](x
                ) - knotfraction(deg, knots[i + deg + 1], knots[i + 1])*
                    basisfuncs[i + 1](x)
            return knotfraction(deg, knots[deg + i], knots[i])*derivative(x,
                i, deg - 1, der - 1
            ) - knotfraction(deg, knots[deg + i + 1], knots[i + 1])*
                derivative(
            x, i + 1, deg - 1, der - 1)

        derivativelist = [(lambda x, i = j, deg = degree, der = deriv:
            derivative(x, i, deg, deriv)) for j in range(len(basislist[-1]))]
        return derivativelist

    def spacing(points,  domain = (0,1), method = 'uniform', *args):

        if type(points) not in [tuple, list, np.ndarray]:
            raise spline.SplineError, "The points must be represented as a
                tuple, list or numpy array"
        try:
            points = np.array(points)
        except:
            raise spline.SplineError, "The points cannot be converted into a
                numpy array"
        if len(points) < 2:
            method = 'uniform'

        if type(domain) not in [tuple, list, np.ndarray]:
            raise spline.SplineError, "The domain must be represented as a
                tuple, list or numpy array"
        if len(domain) != 2:
            raise spline.SplineError, "The domain must be a starting point
                and endingpoint only, for example (0.,1.)"
        if type(domain[0]) not in [int, float] or type(domain[1]) not in [int
            , float]:
            raise spline.SplineError, "The domain must be an array like
                object of length 2 and containing only two integer or floats"

        if not type(method) == str:
            raise TypeError, "The method name must be a string"
        if not method in ['uniform', 'chordlength', 'centripetal', '
            affinechord', 'affineangle']:
```

```
            raise spline.SplineError, """method name must be one of the
                allowed values
            'uniform', 'chordlength', 'centripetal', 'affinechord' or '
                affineangle'"""

        if method == 'uniform':
            h_list = uniform_h(points, domain)
        elif method == 'chordlength':
            h_list = centripetal_h(points, domain, 1)
        elif method == 'centripetal':
            if len(args) == 0:
                h_list = centripetal_h(points, domain, 0.5)
            elif len(args) == 1:
                h_list = centripetal_h(points, domain, args[0])
            else:
                raise spline.SplineError, "centripetal method only accepts
                    one alpha value"
        elif method == 'affinechord':
            h_list = affineInvariant_h(points, domain)
        elif method == 'affineangle':
            if len(args) == 0:
                h_list = affineInvariantAngle_h(points, domain, 1.5)
            elif len(args) == 1:
                h_list = affineInvariantAngle_h(points, domain, args[0])
            else:
                raise spline.SplineError, "affineangle method only accepts
                    one delta value"
        else:
            raise TypeError, "method: {} not implemented".format(method)

        domainlength = domain[1] - domain[0]
        t_values = np.zeros(len(h_list) + 1)
        t_values[0], t_values[-1] = domain
        if len(t_values) == 2:
            return t_values
        for i in range(1,len(t_values) - 1):
            t_values[i] = t_values[i - 1] + h_list[i - 1]*domainlength
        return t_values

    def uniform_h(points, domain):

        return np.array([(domain[1] - domain[0])/(len(points) - 1.)]*(len(
            points) - 1))

    def centripetal_h(points, domain, alpha = 0.5):

        if not type(alpha) in [int, float]:
            raise TypeError, "alpha value has to be an integer or a float"

        lengths = []
        if not len(np.shape(points[0])) in [0,1]:
            raise spline.SplineError, "The individual points has to be
                vectors or scalars"
```

```
    for i in range(1, len(points)):
        if not len(np.shape(points[i])) in [0,1]:
            raise spline.SplineError, "The individual points has to be
                vectors or scalars"
        lengths.append(np.linalg.norm(points[i] - points[i - 1])**alpha)
    totallength = sum(lengths)
    return lengths/totallength

def affineInvariantMetric_h(points):
    try:
        points = np.array(points)
    except:
        raise TypeError, "points not of a recognized type. Must be array-
            like"
    if not np.shape(points)[1] == 2:
        raise TypeError, "This method only woks for two-dim vectors
            arranged as ((x0,y0),(x1,y1),...)"

    pointlen = len(points)
    x_bar = sum(points[:,0])/pointlen; y_bar = sum(points[:,1])/pointlen
    sigma_x = sum([(x - x_bar)**2 for x in points[:,0]])/pointlen
    sigma_y = sum([(y - y_bar)**2 for y in points[:,1]])/pointlen
    sigma_xy = sum([(x - x_bar)*(y - y_bar) for (x,y) in points[:,]])/
        pointlen
    g = sigma_x*sigma_y - sigma_xy**2

    metricmatrix = np.array([[sigma_y/g, -sigma_xy/g],[-sigma_xy/g,
        sigma_x/g]])

    h_list = [np.sqrt(np.dot(np.dot((Xi - Yi),metricmatrix),(Xi - Yi).T))
         for (Xi, Yi) in zip(points[:-1,:],points[1:,:])]
    if not len(h_list) + 1 == pointlen:
        raise spline.SplineError, "h_list has not been calculated
            correctly"

    return h_list

def affineInvariantMetric(points, Pi, Pj):
    try:
        points = np.array(points)
    except:
        raise TypeError, "points not of a recognized type. Must be array-
            like"
    if not np.shape(points)[1] == 2:
        raise TypeError, "This method only woks for two-dim vectors
            arranged as ((x0,y0),(x1,y1),...)"

    pointlen = len(points)
    x_bar = sum(points[:,0])/pointlen; y_bar = sum(points[:,1])/pointlen
    sigma_x = sum([(x - x_bar)**2 for x in points[:,0]])/pointlen
    sigma_y = sum([(y - y_bar)**2 for y in points[:,1]])/pointlen
    sigma_xy = sum([(x - x_bar)*(y - y_bar) for (x,y) in points[:,]])/
        pointlen
```

```
    g = sigma_x*sigma_y - sigma_xy**2

    metricmatrix = np.array([[sigma_y/g, -sigma_xy/g],[-sigma_xy/g,
        sigma_x/g]])

    h = np.sqrt(np.dot(np.dot((Pi - Pj),metricmatrix),(Pi - Pj).T))

    return h

def affineInvariant_h(points, domain):
    try:
        points = np.array(points)
    except:
        raise TypeError, "points not of a recognized type. Must be array-
            like"
    if not np.shape(points)[1] == 2:
        raise TypeError, "This method only works for two-dim vectors
            arranged as ((x0,y0),(x1,y1),...)"

    lengths = affineInvariantMetric_h(points)
    totallength = sum(lengths)
    return lengths/totallength

def affineInvariantAngle_h(points, domain, delta = 1.5):
    try:
        points = np.array(points)
    except:
        raise TypeError, "points not of a recognized type. Must be array-
            like"
    if not np.shape(points)[1] == 2:
        raise TypeError, "This method only works for two-dim vectors
            arranged as ((x0,y0),(x1,y1),...)"

    points = np.vstack([points[0],points,points[-1]]) #This is done so
        that d_{-1} and d_{n} will equal 0
    dlengths = affineInvariantMetric_h(points)

    def theta_i(di1, di2, Pi, Pj):
        if di1 == 0. or di2 == 0.:
            alpha_i = 0
        else:
            alpha_i = np.pi - np.arccos((di1**2 + di2**2 - (
                affineInvariantMetric(points, Pi, Pj))**2)/(2.*di1*di2))
        return min(np.pi/2, alpha_i)

    lengths = []
    for i in range(1, len(dlengths) - 1):

        lengths.append(dlengths[i]*(1 + (delta*theta_i(dlengths[i-1],
            dlengths[i],points[i-1],points[i+1])*dlengths[i-1])/(dlengths[
            i-1] + dlengths[i])
        + (delta*theta_i(dlengths[i],dlengths[i+1],points[i],points[i+2])
            *dlengths[i+1])/(dlengths[i] + dlengths[i + 1])))
```

```
    if not len(lengths) + 3 == len(points):
        raise spline.SplineError, "h_list has not been calculated
            correctly"

    totallength = sum(lengths)
    return lengths/totallength

def knotAveraging(tparam, deg, domain):
    """
    Create the knot vector by averaging the parameter values.
    """
    tlen = len(tparam)
    if tlen <= deg:
        raise TypeError, "Not enough parameter values. Has to be strictly
             more than the degree"

    start = np.ones(deg + 1)*domain[0]
    stop = np.ones(deg + 1)*domain[1]
    length = domain[1] - domain[0]

    center = np.empty(tlen - deg - 1)
    for i in range(len(center)):
        s, e = int(i + 1), int(i + 1 + deg)
        center[i] = domain[0] + length/deg*sum(tparam[s:e])

    return np.hstack([start, center, stop])
```

A.3 spline.py

The code extends the spline classes from PyNURBS, further documentation on them can be found there.

```
# -*- coding: utf-8 -*-
from __future__ import division
import numpy as np
import spline

def basisarray(degree, knots):

    def knotfraction(u0, u1, u2, u3):
        if u2 == u3: return 0
        return (u0 - u1)/(u2 - u3)

    def basisfunc(x, i, deg):
        if deg == 0:
            if knots[i + 1] == knots[-1]:
                return float(knots[i] <= x <= knots[i + 1])
            if knots[i] == knots[i + 1]: return 0.
            return float(knots[i] <= x < knots[i + 1])
        return knotfraction(x,knots[i], knots[i + deg], knots[i])*
            basisfunc(
        x, i, deg - 1) + knotfraction(knots[i + deg + 1], x, knots[i +
            deg + 1], knots[i + 1]
        )*basisfunc(x, i + 1, deg - 1)
```

```
    basislist = [[(lambda x, i = j, deg = degr: basisfunc(x, i, deg)) for
        j in range(len(knots) - 1 - degr)] for degr in range(degree + 1)]
    return basislist

def basisderivative(deriv, basislist, knots):
    degree = len(basislist) - 1
    if type(deriv) != int:
        raise TypeError, "Deriv has to be an integer"
    if deriv < 1:
        raise TypeError, " Deriv has to be 1 or larger"
    basisfuncs = basislist[-1 - deriv]

    def knotfraction(deg, u1, u2):
        if u1 == u2: return 0
        return 1/float(u1 - u2)

    def derivative(x, i, deg, der):

        if der == 1:
            return  knotfraction(deg, knots[i + deg], knots[i])*
                basisfuncs[i](x
            ) - knotfraction(deg, knots[i + deg + 1], knots[i + 1])*
                basisfuncs[i + 1](x)
        return knotfraction(deg, knots[deg + i], knots[i])*derivative(x,
            i, deg - 1, der - 1
        ) - knotfraction(deg, knots[deg + i + 1], knots[i + 1])*
            derivative(
        x, i + 1, deg - 1, der - 1)

    derivativelist = [(lambda x, i = j, deg = degree, der = deriv:
        derivative(x, i, deg, deriv)) for j in range(len(basislist[-1]))]
    return derivativelist

def spacing(points,  domain = (0,1), method = 'uniform', *args):

    if type(points) not in [tuple, list, np.ndarray]:
        raise spline.SplineError, "The points must be represented as a
            tuple, list or numpy array"
    try:
        points = np.array(points)
    except:
        raise spline.SplineError, "The points cannot be converted into a
            numpy array"
    if len(points) < 2:
        method = 'uniform'

    if type(domain) not in [tuple, list, np.ndarray]:
        raise spline.SplineError, "The domain must be represented as a
            tuple, list or numpy array"
    if len(domain) != 2:
        raise spline.SplineError, "The domain must be a starting point
            and endingpoint only, for example (0.,1.)"
```

```
    if type(domain[0]) not in [int, float] or type(domain[1]) not in [int
        , float]:
        raise spline.SplineError, "The domain must be an array like
            object of length 2 and containing only two integer or floats"

    if not type(method) == str:
        raise TypeError, "The method name must be a string"
    if not method in ['uniform', 'chordlength', 'centripetal', '
        affinechord', 'affineangle']:
        raise spline.SplineError, """method name must be one of the
            allowed values
        'uniform', 'chordlength', 'centripetal', 'affinechord' or '
            affineangle'"""

    if method == 'uniform':
        h_list = uniform_h(points, domain)
    elif method == 'chordlength':
        h_list = centripetal_h(points, domain, 1)
    elif method == 'centripetal':
        if len(args) == 0:
            h_list = centripetal_h(points, domain, 0.5)
        elif len(args) == 1:
            h_list = centripetal_h(points, domain, args[0])
        else:
            raise spline.SplineError, "centripetal method only accepts
                one alpha value"
    elif method == 'affinechord':
        h_list = affineInvariant_h(points, domain)
    elif method == 'affineangle':
        if len(args) == 0:
            h_list = affineInvariantAngle_h(points, domain, 1.5)
        elif len(args) == 1:
            h_list = affineInvariantAngle_h(points, domain, args[0])
        else:
            raise spline.SplineError, "affineangle method only accepts
                one delta value"
    else:
        raise TypeError, "method: {} not implemented".format(method)

    domainlength = domain[1] - domain[0]
    t_values = np.zeros(len(h_list) + 1)
    t_values[0], t_values[-1] = domain
    if len(t_values) == 2:
        return t_values
    for i in range(1,len(t_values) - 1):
        t_values[i] = t_values[i - 1] + h_list[i - 1]*domainlength
    return t_values

def uniform_h(points, domain):

    return np.array([(domain[1] - domain[0])/(len(points) - 1.)]*(len(
        points) - 1))
```

```
def centripetal_h(points, domain, alpha = 0.5):

    if not type(alpha) in [int, float]:
        raise TypeError, "alpha value has to be an integer or a float"

    lengths = []
    if not len(np.shape(points[0])) in [0,1]:
        raise spline.SplineError, "The individual points has to be
            vectors or scalars"
    for i in range(1, len(points)):
        if not len(np.shape(points[i])) in [0,1]:
            raise spline.SplineError, "The individual points has to be
                vectors or scalars"
        lengths.append(np.linalg.norm(points[i] - points[i - 1])**alpha)
    totallength = sum(lengths)
    return lengths/totallength

def affineInvariantMetric_h(points):
    try:
        points = np.array(points)
    except:
        raise TypeError, "points not of a recognized type. Must be array-
            like"
    if not np.shape(points)[1] == 2:
        raise TypeError, "This method only woks for two-dim vectors
            arranged as ((x0,y0),(x1,y1),...)"

    pointlen = len(points)
    x_bar = sum(points[:,0])/pointlen; y_bar = sum(points[:,1])/pointlen
    sigma_x = sum([(x - x_bar)**2 for x in points[:,0]])/pointlen
    sigma_y = sum([(y - y_bar)**2 for y in points[:,1]])/pointlen
    sigma_xy = sum([(x - x_bar)*(y - y_bar) for (x,y) in points[:,]])/
        pointlen
    g = sigma_x*sigma_y - sigma_xy**2

    metricmatrix = np.array([[sigma_y/g, -sigma_xy/g],[-sigma_xy/g,
        sigma_x/g]])

    h_list = [np.sqrt(np.dot(np.dot((Xi - Yi),metricmatrix),(Xi - Yi).T))
         for (Xi, Yi) in zip(points[:-1,:],points[1:,:])]
    if not len(h_list) + 1 == pointlen:
        raise spline.SplineError, "h_list has not been calculated
            correctly"

    return h_list

def affineInvariantMetric(points, Pi, Pj):
    try:
        points = np.array(points)
    except:
        raise TypeError, "points not of a recognized type. Must be array-
            like"
    if not np.shape(points)[1] == 2:
```

```
            raise TypeError, "This method only woks for two-dim vectors
                arranged as ((x0,y0),(x1,y1),...)"

        pointlen = len(points)
        x_bar = sum(points[:,0])/pointlen; y_bar = sum(points[:,1])/pointlen
        sigma_x = sum([(x - x_bar)**2 for x in points[:,0]])/pointlen
        sigma_y = sum([(y - y_bar)**2 for y in points[:,1]])/pointlen
        sigma_xy = sum([(x - x_bar)*(y - y_bar) for (x,y) in points[:,]])/
            pointlen
        g = sigma_x*sigma_y - sigma_xy**2

        metricmatrix = np.array([[sigma_y/g, -sigma_xy/g],[-sigma_xy/g,
            sigma_x/g]])

        h = np.sqrt(np.dot(np.dot((Pi - Pj),metricmatrix),(Pi - Pj).T))

        return h

    def affineInvariant_h(points, domain):
        try:
            points = np.array(points)
        except:
            raise TypeError, "points not of a recognized type. Must be array-
                like"
        if not np.shape(points)[1] == 2:
            raise TypeError, "This method only works for two-dim vectors
                arranged as ((x0,y0),(x1,y1),...)"

        lengths = affineInvariantMetric_h(points)
        totallength = sum(lengths)
        return lengths/totallength

    def affineInvariantAngle_h(points, domain, delta = 1.5):
        try:
            points = np.array(points)
        except:
            raise TypeError, "points not of a recognized type. Must be array-
                like"
        if not np.shape(points)[1] == 2:
            raise TypeError, "This method only works for two-dim vectors
                arranged as ((x0,y0),(x1,y1),...)"

        points = np.vstack([points[0],points,points[-1]]) #This is done so
            that d_{-1} and d_{n} will equal 0
        dlengths = affineInvariantMetric_h(points)

        def theta_i(di1, di2, Pi, Pj):
            if di1 == 0. or di2 == 0.:
                alpha_i = 0
            else:
                alpha_i = np.pi - np.arccos((di1**2 + di2**2 - (
                    affineInvariantMetric(points, Pi, Pj))**2)/(2.*di1*di2))
            return min(np.pi/2, alpha_i)
```

```
    lengths = []
    for i in range(1, len(dlengths) - 1):

        lengths.append(dlengths[i]*(1 + (delta*theta_i(dlengths[i-1],
            dlengths[i],points[i-1],points[i+1])*dlengths[i-1])/(dlengths[
            i-1] + dlengths[i])
        + (delta*theta_i(dlengths[i],dlengths[i+1],points[i],points[i+2])
            *dlengths[i+1])/(dlengths[i] + dlengths[i + 1])))

    if not len(lengths) + 3 == len(points):
        raise spline.SplineError, "h_list has not been calculated
            correctly"

    totallength = sum(lengths)
    return lengths/totallength

def knotAveraging(tparam, deg, domain):
    """
    Create the knot vector by averaging the parameter values.
    """
    tlen = len(tparam)
    if tlen <= deg:
        raise TypeError, "Not enough parameter values. Has to be strictly
             more than the degree"

    start = np.ones(deg + 1)*domain[0]
    stop = np.ones(deg + 1)*domain[1]
    length = domain[1] - domain[0]

    center = np.empty(tlen - deg - 1)
    for i in range(len(center)):
        s, e = int(i + 1), int(i + 1 + deg)
        center[i] = domain[0] + length/deg*sum(tparam[s:e])

    return np.hstack([start, center, stop])
```

A.4 plots.py

`simp_plot` produces Figure 5, `av_v2_plot` produces Figure 6, `av_v2_diff_plot` produces Figure 7 and `good_der_plot` produces Figure 9

```
# -*- coding: utf-8 -*-
from __future__ import division
import matplotlib.pyplot as plt
from turbine import SuctionSide, PressureSide, leadingEdge, trailingEdge
from matplotlib.lines import Line2D

def simp_plot(shw = False):
    inputData = [51.5, 90-15.92, 8., 1.1817, 75.4075, 29.2, 1., .0555,
        .0220, 30., 2.0]
    lambd = 4.4#;SuctionSide.functionOfLambda(inputData, ('affineangle
        ',1.5), 'param', ('affinechord',), 'average', False)
```

```
    turbine1 = SuctionSide.functionOfLambda(inputData,('uniform',), '
        radius', ('uniform',), 1, False)
    turbine2 = PressureSide(inputData, 'radius', ('uniform',), 'constant'
        )
    leading, trailing = leadingEdge(inputData), trailingEdge(inputData)
    fig = plt.figure(facecolor = 'white')
    turbine1(lambd).plot2D(100, True, fig)
    turbine2.plot2D(100, True, fig, False)
    leading.plot2D(100, False)
    trailing.plot2D(100, False)

    ax = plt.gca()
    ax.set_frame_on(False)
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()
    ax.set_aspect('equal')

    xticks = ax.xaxis.get_major_ticks()
    xticks[0].label1.set_visible(False)
    xticks[2].label1.set_visible(False)
    xticks[4].label1.set_visible(False)
    yticks = ax.yaxis.get_major_ticks()
    yticks[0].label1.set_visible(False)
    yticks[2].label1.set_visible(False)
    yticks[4].label1.set_visible(False)
    yticks[6].label1.set_visible(False)

    xmin, xmax = ax.get_xaxis().get_view_interval()
    ymin, ymax = ax.get_yaxis().get_view_interval()
    ax.add_artist(Line2D((xmin, xmax), (ymin, ymin), color='black',
        linewidth=2))
    ax.add_artist(Line2D( (ymin, ymin), (xmin, ymax), color='black',
        linewidth=2))

    if not shw:
        plt.savefig('../../latex/img/simplistic_curve.pdf', format='pdf',
            dpi = 1200, bbox_inches='tight', pad_inches=0)
    if shw:
        plt.show()

def av_v2_plot(shw = False):
    inputData = [51.5, 90-15.92, 8., 1.1817, 75.4075, 29.2, 1., .0555,
        .0220, 30., 2.0]
    lambd = 3.4
    turbine1 = SuctionSide.functionOfLambda(inputData,('uniform',), '
        radius', ('uniform',), 'average', False)
    turbine2 = PressureSide(inputData, 'radius', ('uniform',), 'radius')
    leading, trailing = leadingEdge(inputData), trailingEdge(inputData)
    fig = plt.figure(facecolor = 'white')
    turbine1(lambd).plot2D(100, True, fig)
    turbine2.plot2D(100, True, fig, False)
    leading.plot2D(100, False)
    trailing.plot2D(100, False)
```

```
    ax = plt.gca()
    ax.set_frame_on(False)
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()
    ax.set_aspect('equal')

    xticks = ax.xaxis.get_major_ticks()
    xticks[0].label1.set_visible(False)
    xticks[2].label1.set_visible(False)
    xticks[3].label1.set_visible(False)
    xticks[4].label1.set_visible(False)
    xticks[5].label1.set_visible(False)
    xticks[7].label1.set_visible(False)
    xticks[8].label1.set_visible(False)
    xticks[9].label1.set_visible(False)
    yticks = ax.yaxis.get_major_ticks()
    yticks[1].label1.set_visible(False)
    yticks[3].label1.set_visible(False)
    yticks[5].label1.set_visible(False)
    yticks[7].label1.set_visible(False)

    xmin, xmax = ax.get_xaxis().get_view_interval()
    ymin, ymax = ax.get_yaxis().get_view_interval()
    ax.add_artist(Line2D((xmin, xmax), (ymin, ymin), color='black',
        linewidth=2))
    ax.add_artist(Line2D( (xmin, xmin), (ymin, ymax), color='black',
        linewidth=2))

    if not shw:
        plt.savefig('../../latex/img/average_v2_curve.pdf', format='pdf',
            dpi = 1200, bbox_inches='tight', pad_inches=0)
    if shw:
        plt.show()

def av_v2_diff_plot(shw = False):
    inputData = [51.5, 90-15.92, 8., 1.1817, 75.4075, 29.2, 1., .0555,
        .0220, 30., 2.0]
    lambd = 3.4
    turbine1 = SuctionSide.functionOfLambda(inputData,('centripetal',), '
        radius', ('uniform',), 'average', False)
    turbine2 = PressureSide(inputData, 'radius', ('chordlength',), '
        radius')
    leading, trailing = leadingEdge(inputData), trailingEdge(inputData)
    fig = plt.figure(facecolor = 'white')
    turbine1(lambd).plot2D(100, True, fig)
    turbine2.plot2D(100, True, fig, False)
    leading.plot2D(100, False)
    trailing.plot2D(100, False)

    ax = plt.gca()
    ax.set_frame_on(False)
    ax.get_xaxis().tick_bottom()
```

```
    ax.get_yaxis().tick_left()
    ax.set_aspect('equal')

    xticks = ax.xaxis.get_major_ticks()
    xticks[0].label1.set_visible(False)
    xticks[2].label1.set_visible(False)
    xticks[4].label1.set_visible(False)
    yticks = ax.yaxis.get_major_ticks()
    yticks[0].label1.set_visible(False)
    yticks[2].label1.set_visible(False)
    yticks[4].label1.set_visible(False)
    yticks[6].label1.set_visible(False)

    xmin, xmax = ax.get_xaxis().get_view_interval()
    ymin, ymax = ax.get_yaxis().get_view_interval()
    ax.add_artist(Line2D((xmin, xmax), (ymin, ymin), color='black',
        linewidth=2))
    ax.add_artist(Line2D( (xmin, xmin), (ymin, ymax), color='black',
        linewidth=2))

    if not shw:
        plt.savefig('../../latex/img/average_v2_diff_curve.pdf', format='
            pdf', dpi = 1200, bbox_inches='tight', pad_inches=0)
    if shw:
        plt.show()

def good_der_plot(shw = False):
    inputData = [51.5, 90-15.92, 8., 1.1817, 75.4075, 29.2, 1., .0555,
        .0220, 30., 2.0]
    lambd = 3.4
    turbine1 = SuctionSide.functionOfLambda(inputData,('affineangle',), '
        param', ('affinechord',), 'average', False)
    turbine2 = PressureSide(inputData, 'param', ('chordlength',), 'radius
        ')
    leading, trailing = leadingEdge(inputData), trailingEdge(inputData)
    fig = plt.figure(facecolor = 'white')
    turbine1(lambd).plot2D(100, True, fig)
    turbine2.plot2D(100, True, fig, False)
    leading.plot2D(100, False)
    trailing.plot2D(100, False)

    ax = plt.gca()
    ax.set_frame_on(False)
    ax.get_xaxis().tick_bottom()
    ax.get_yaxis().tick_left()
    ax.set_aspect('equal')

    xticks = ax.xaxis.get_major_ticks()
    xticks[1].label1.set_visible(False)
    xticks[2].label1.set_visible(False)
    xticks[3].label1.set_visible(False)
    xticks[4].label1.set_visible(False)
    yticks = ax.yaxis.get_major_ticks()
```

```
yticks[0].label1.set_visible(False)
yticks[2].label1.set_visible(False)
yticks[4].label1.set_visible(False)

xmin, xmax = ax.get_xaxis().get_view_interval()
ymin, ymax = ax.get_yaxis().get_view_interval()
ax.add_artist(Line2D((xmin, xmax), (ymin, ymin), color='black',
    linewidth=2))
ax.add_artist(Line2D( (xmin, xmin), (ymin, ymax), color='black',
    linewidth=2))

if not shw:
    plt.savefig('../../latex/img/good_der_curve.pdf', format='pdf',
        dpi = 1200, bbox_inches='tight', pad_inches=0)
if shw:
    plt.show()
```

References

[1] Brian A. Barsky and Tony D. DeRose. Geometric continuity of parametric curves: Three equivalent characterizations. *IEEE Computer Graphics and Applications*, 9(6):60–68, 1989.

[2] Olivier Cleynen. Turbine inlet guide vanes of atar turbojet. `https://en.wikipedia.org/wiki/File:Turbine_inlet_guide_vanes_of_Atar_turbojet.jpg`, March 2012. [Online; accessed 2015-01-17].

[3] Carl de Boor. *A practical guide to splines*, volume 27 of *Applied Mathematical Sciences*. Springer-Verlag, New York-Berlin, 1978.

[4] Gerald Farin. *Curves and surfaces for computer-aided geometric design*. Computer Science and Scientific Computing. Academic Press, Inc., San Diego, CA, fourth edition, 1997. A practical guide, Chapter 1 by P. Bézier; Chapters 11 and 22 by W. Boehm, With 1 IBM-PC floppy disk (3.5 inch; HD).

[5] Gerald E. Farin. *NURBS*. A K Peters, Ltd., Natick, MA, second edition, 1999. From projective geometry to practical use.

[6] Thomas A. Foley and Gregory M. Nielson. Knot selection for parametric spline interpolation. In *Mathematical methods in computer aided geometric design (Oslo, 1988)*, pages 261–271. Academic Press, Boston, MA, 1989.

[7] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[8] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. `http://www.scipy.org/`, 2001–. [Online; accessed 2015-01-17].

[9] Gregory M. Nielson. Coordinate free scattered data interpolation. In C. K. Chui, L. L. Shumaker, and F. I. Utreras, editors, *Topics in multivariate approximation (Santiago, 1986)*, pages 175–184. Academic Press, Boston, MA, 1987.

[10] Travis Oliphant et al. NumPy. `http://www.numpy.org/`, 2006–. [Online; accessed 2015-01-17].

[11] L. Piegl and W. Tiller. *The NURBS Book*. Monographs in Visual Communication. Springer, 1997.

[12] M. J. D. Powell. *Approximation theory and methods*. Cambridge University Press, Cambridge-New York, 1981.

[13] Andrew Pressley. *Elementary differential geometry*. Springer Undergraduate Mathematics Series. Springer-Verlag London, Ltd., London, second edition, 2010.

[14] Jonas Svensson. Parametric blade profiling methods and improvements on LUAX-T with emphasis on oxy-fuel cycles. Master's Thesis. Faculty of Engineering, Lund University, 2011.

[15] Runar Tenfjord, Alex Wiltschko, Sebastian Eiser, et al. PyNURBS - non uniform rational B-splines in Python. `https://github.com/bashseb/PyNURBS`, 2013.

[16] Z. Q. Ye and P. Kavanagh. Axial-flow turbine cascade design procedure and sample design cases. Technical Report ISU-ERI-Ames-84159, Department of Mechanical Engineering, Iowa State University, January 1984.

Bachelor's Theses in Mathematical Sciences 2015:K2
ISSN 1654-6229

LUNFNA-4003-2015

Numerical Analysis
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/