CAGD - Homework 5

Josefine Stål & Erik Ackzell

October 18, 2016

Task 1

In this task we convert between barycentric and homogeneous coordinates. Consider the points

$$p_0 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \quad p_1 = \begin{pmatrix} 3 \\ 3 \\ 3 \end{pmatrix}, \quad p_2 = \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix}$$

and let

$$q_1 = \begin{pmatrix} 0.25 \\ 0.25 \\ 0.5 \end{pmatrix}$$

in barycentric coordinates with respect to p_0, p_1, p_2. We want to express q_1 in homogeneous coordinates.
First, we express q_1 in Cartesian coordinates

$$q_1 = 0.25p_0 + 0.25p_1 + 0.5p_2 = \begin{pmatrix} 0.25 + 1.5 + 0.25 \\ 0.25 + 1.5 + 0.5 \\ 0.25 + 1.5 + 0.5 \end{pmatrix} = \begin{pmatrix} 2 \\ 2.25 \\ 2.25 \end{pmatrix}.$$

For any $\omega \in \mathbb{R}$, the homogeneous coordinates of q_1 are

$$q_1 = \begin{pmatrix} 2\omega \\ 2.25\omega \\ 2.25\omega \\ \omega \end{pmatrix},$$

which is what we wanted to determine.
Now let

$$q_2 = \begin{pmatrix} 5 \\ 4 \\ 4 \\ 3 \end{pmatrix}$$

in homogeneous coordinates. We wish to express q_2 in barycentric coordinates with respect to p_0, p_1, p_2.
First, we express q_2 in Cartesian coordinates

$$q_2 = \frac{1}{3}\begin{pmatrix} 5 \\ 4 \\ 4 \end{pmatrix}.$$

We now want to determine the coefficients a_0, a_1, a_2 such that

$$\sum_{i=0}^{2} a_i p_i$$

and

$$\sum_{i=0}^{2} a_i = 1.$$

This can be done by solving the linear equation system

$$\begin{pmatrix} 1 & 3 & 1 \\ 1 & 3 & 2 \\ 1 & 3 & 2 \\ 1 & 1 & 1 \end{pmatrix}\begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \frac{1}{3}\begin{pmatrix} 5 \\ 4 \\ 4 \\ 3 \end{pmatrix},$$

which reduces to

$$\begin{pmatrix} 1 & 3 & 1 \\ 1 & 3 & 2 \\ 1 & 1 & 1 \end{pmatrix}\begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \frac{1}{3}\begin{pmatrix} 5 \\ 4 \\ 3 \end{pmatrix}.$$

The solution is given by

$$a_0 = 1, \quad a_1 = \frac{1}{3}, \quad a_2 = -\frac{1}{3},$$

so in barycentric coordinates with respect to p_0, p_1, p_2,

$$q_2 = \frac{1}{3}\begin{pmatrix} 3 \\ 1 \\ -1 \end{pmatrix},$$

which is what we wanted to determine.

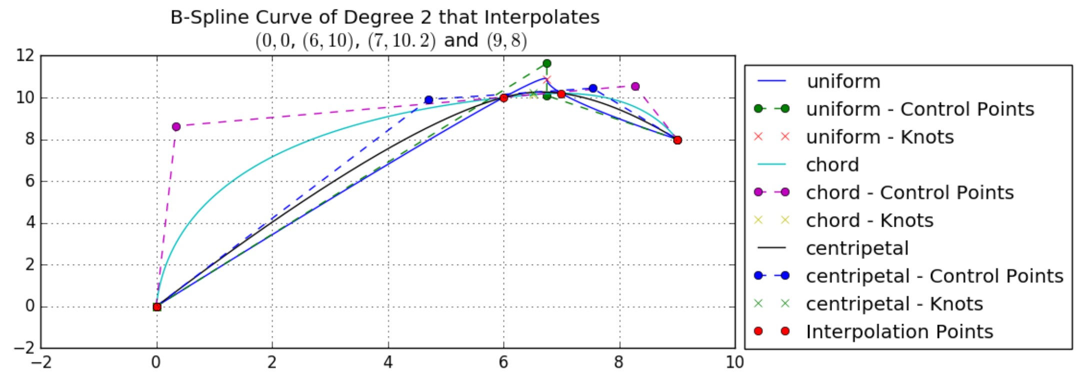
Task 2

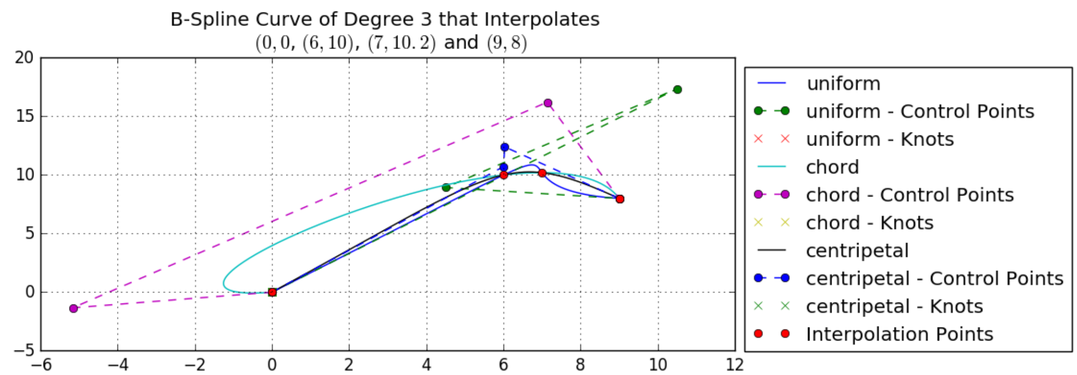
In this task we interpolate the points $(0, 0)$, $(6, 10)$, $(7, 10.2)$ and $(9, 8)$ using B-Spline curves of degree 2 and 3. First we needed to set the parameter values $t_0, ..., t_s$ using one of the three methods *Uniform Parametrization*, *Chord Length*

Parametrization or *Centripetal Parametrization*. Then we calculated the knots followed by solving the system below to find the control points.

$$\begin{pmatrix} N_0^p(t_0) & N_1^p(t_0) & \cdots & N_s^p(t_0) \\ N_0^p(t_1) & N_1^p(t_1) & \cdots & N_s^p(t_1) \\ \vdots & \vdots & & \vdots \\ N_0^p(t_s) & N_1^p(t_s) & \cdots & N_s^p(t_s) \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_s \end{pmatrix} = \begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_s \end{pmatrix}.$$

where N_j^p is the j:th basis function of degree p, b_j the control points and d_j the interpolation points. The code written for this task can be found in Appendix II.





Task 3

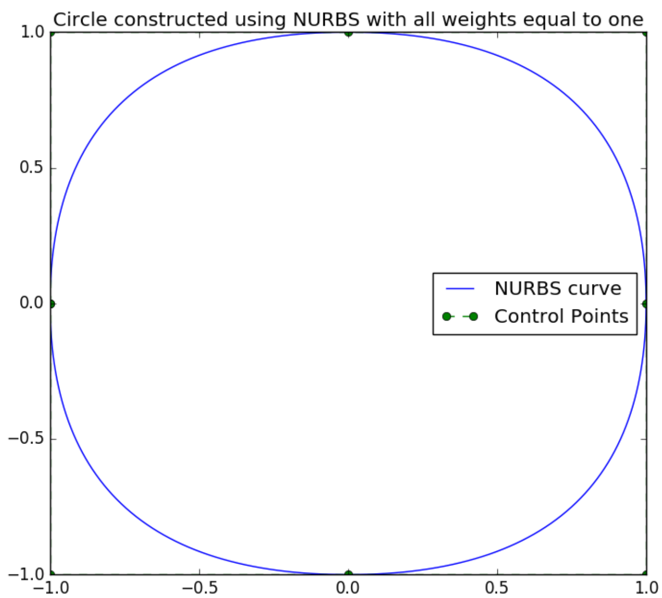
In this task we construct a circle using second degree NURBS and the de Boor algorithm. The code can be seen in Appendix I.
The following nine control points are used

$$\{(-1,0),(-1,1),(0,1),(1,1),(1,0),(1,-1),(0,-1),(-1,-1),(-1,0)\},$$

consisting of the vertices and midpoints of the unit square centered at $(0, 0)$ and its sides, respectively. As we want the curve to pass through the midpoints of the vertices and as these, apart from the starting point, will be situated at $\frac{1}{4}$, $\frac{2}{4}$ and $\frac{3}{4}$ of the whole curve, we need to include these points in the knot sequence with multiplicity two.
As we also want the curve to be clamped, this results in the knot sequence

$$\{0, 0, 0, 0.25, 0.25, 0.5, 0.5, 0.75, 0.75, 1, 1, 1\}.$$

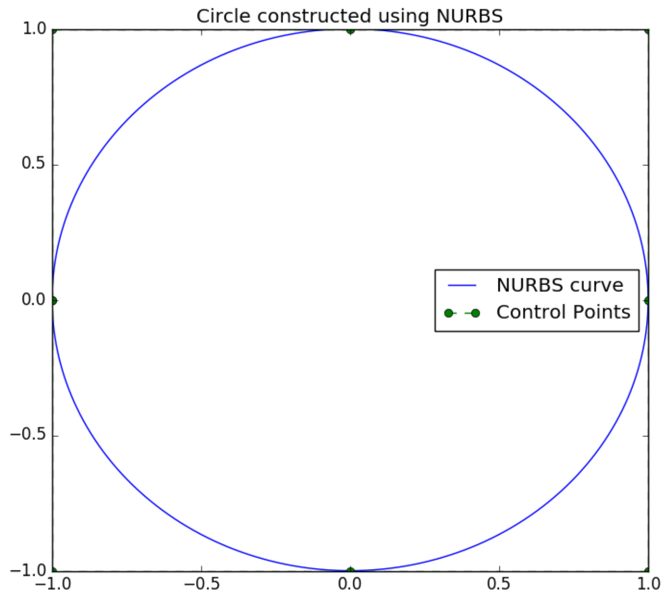
If all weights are set to one, we end up with a B-spline curve with the following appearance



In order to obtain an actual circle, we change the weights corresponding to the control points located in the vertices of the square. By trial and error, we find the correct weights ω to be

$$\omega = \{1, 0.707, 1, 0.707, 1, 0.707, 1, 0.707, 1\},$$

so the weights corresponding to the vertices of the square are approximately $\frac{\sqrt{2}}{2}$. The circle can be seen below.



Task 4

In this task we contructed a rational Bzier curve, with the Rational de Casteljau Algorithm, with varying weights. The control points that we used were $(0, 0)$, $(4, 3)$, $(3, 1)$ and $(5, 1)$ and the weights were originally $[1, 2, 3, 4]$.

To evaluate a rational Bzier curve in $\mathbb{R}^2$ we must first project the points to $\mathbb{R}^3$ using homogeneous coordinates. Thereafter the algorithm proceeds as usual. Before we plot, we traverse the coordinates to be cartesian again to be able to plot the curve on the plane.

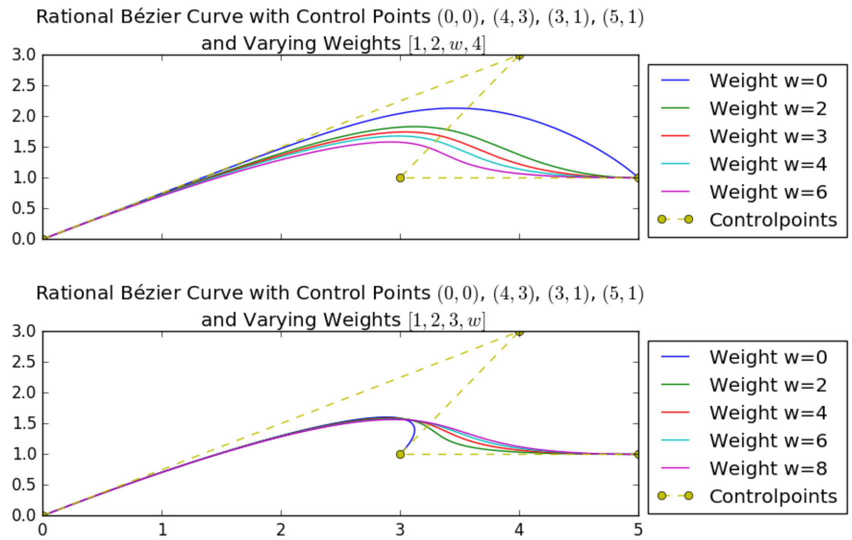
The first plot demonstrate how the curve changes when the third weight varies between 0 and 6. The second plot demonstrates the changes when the last weight varies between 0 and 8. What is clear is that if the weight is 0 we neglect the relevant control point. The code written for this task can be found in Appendix III.

Do-over Homework 2 Task 5

<u>Claim 1:</u> The Lagrange form is invariant under all affine maps.

<u>Proof:</u> Let $\varphi : \mathbb{E}^n \to \mathbb{E}^n$ be any affine map. Then φ can be expressed as

$$\varphi(u) = Au + v,$$



with $A \in \mathbb{R}^{n\times n}$ and $v \in \mathbb{R}^n$. Let p be a polynomial of degree n. Then p can be expressed as a linear combination of Lagrange basis polynomials L_i^n of degree n, i.e.

$$p(x) = \sum_{i=0}^{n} L_i^n(x)c_i,$$

with $c_i \in \mathbb{R}^n$ for $i = 0, 1, \dots, n$. Recall that the Lagrange basis fulfills the partition of unity, i.e.

$$\sum_{i=0}^{n} L_i^n(x) = 1.$$

We have that

$$\begin{aligned}
\varphi(p(x)) &= \varphi\left(\sum_{i=0}^{n} L_i^n(x)c_i\right) \\
&= A\sum_{i=0}^{n} L_i^n(x)c_i + v \\
&= \sum_{i=0}^{n} L_i^n(x)Ac_i + v \\
&= \sum_{i=0}^{n} L_i^n(x)Ac_i + 1\cdot v \\
&= \sum_{i=0}^{n} L_i^n(x)Ac_i + \sum_{i=0}^{n} L_i^n(x)v \\
&= \sum_{i=0}^{n} L_i^n(x)(Ac_i + v) \\
&= \sum_{i=0}^{n} L_i^n(x)\varphi(c_i)
\end{aligned}$$

which is what we wanted to show. □

Corollary: The Lagrange form in invariant under all affine domain transformations.
Proof: As all affine domain transformations are affine maps, the result follows directly from Claim 1. □

Claim 2: The monomial form is **not** invariant under all affine domain transformations.

Proof: Let $\varphi : [c,d] \to [0,1]$ be the affine domain transformation defined by

$$\varphi(u) = \frac{u-c}{d-c}$$

and set

$$A = \frac{1}{d-c}, \quad v = \frac{c}{d-c},$$

so that

$$\varphi(u) = Au + v.$$

Now let p be a polynomial of degree 1 on such that $\mathrm{Im}(p|_{[a,d]}) = [c,d]$. Then we can express p as

$$\begin{aligned}
p(x) &= \sum_{i=0}^{1} x^i c_i \\
&= c_0 + xc_1, \quad x \in [a,b].
\end{aligned}$$

Then, for $x \in [a, b]$,

$$\begin{aligned}\varphi(p(x)) &= \varphi(c_0 + xc_1)\\ &= A(c_0 + xc_1) + v\end{aligned}$$

Furthermore, for $x \in [a, b]$,

$$\begin{aligned}\sum_{i=0}^{1} x^i \varphi(c_i) &= \varphi(c_0) + x\varphi(c_1)\\ &= Ac_0 + v + x(Ac_1 + v)\end{aligned}$$

Thus, for $x \in [a, b]$, $\varphi(p(x)) = \sum_{i=0}^{1} x^i \varphi(c_i)$ iff $xv = 0$ which is not true in general. Hence, the monomial for is not invariant under all affine domain transformations. $\square$

Appendix I

```
import scipy
from matplotlib import pyplot as plt


class NURBS:
    def __init__(self, grid, controlpoints, degree, weights):
        if len(controlpoints) != len(weights):
            raise ValueError('Not same number of controlpoints and
                weights')
        try:
            grid = scipy.array(grid)
            grid = grid.reshape((len(grid), 1))
        except ValueError:
            raise ValueError('Grid should be a one-dimensional list or
                array')
        self.grid = grid.reshape((len(grid), 1))
        self.controlpoints = controlpoints
        self.degree = degree
        self.weights = weights
        self.dim = scipy.shape(controlpoints)[1] + 1

    def __call__(self, u):
        index = self.get_index(u)
        r = self.get_mult(index, u)
        controlpoints, weights =
            self.get_controlpoints_and_weights(index,
                                                                 r,
                                                                 u)
        current_controlpoints = self.convert_to_homogeneous(
                                        controlpoints=controlpoints,
                                        weights=weights)
        num_controlpoints = len(current_controlpoints)
        d = scipy.zeros((num_controlpoints, num_controlpoints, self.dim))
        d[0] = current_controlpoints
        for s in range(1, num_controlpoints): # column
            for j in range(s, num_controlpoints): # rows
                left = index - self.degree + j
                right = index + j - s + 1
                a = (u - self.grid[left])\
                    / (self.grid[right] - self.grid[left])
                d[s, j] = (1 - a) * d[s-1, j-1] + a * d[s-1, j]
        return self.convert_to_cartesian(
            d[-1, -1].reshape((1, len(d[-1, -1])))).flatten()

    def get_mult(self, index, u):
        if u == self.grid[index]:
            return len([i for i in self.grid if i == self.grid[index]])
```

```
        elif u == self.grid[-1]:
            return len([i for i in self.grid if i == self.grid[-1]])
        else:
            return 0

    def convert_to_homogeneous(self, controlpoints, weights):
        homogeneous_controlpoints = scipy.zeros((controlpoints.shape[0],
                                                controlpoints.shape[1] + 1))
        for i in range(len(homogeneous_controlpoints)):
            homogeneous_controlpoints[i] = weights[i] *\
                                    scipy.concatenate((controlpoints[i],
                                                       scipy.ones(1)))
        return homogeneous_controlpoints

    def convert_to_cartesian(self, homogeneous_controlpoints):
        cartesian_controlpoints = scipy.zeros(
            (homogeneous_controlpoints.shape[0],
             homogeneous_controlpoints.shape[1] - 1))

        for i in range(len(cartesian_controlpoints)):
            cartesian_controlpoints[i] = \
                        (1 / homogeneous_controlpoints[i][-1]) *\
                        homogeneous_controlpoints[i][:-1]
        return cartesian_controlpoints

    def get_controlpoints_and_weights(self, index, r, u):
        if r > self.degree:
            if u == self.grid[0]:
                current_controlpoints = self.controlpoints[:self.degree +
                    1]
                current_weights = self.weights[:self.degree + 1]
            else:
                current_controlpoints = self.controlpoints[-(self.degree
                    + 1):]
                current_weights = self.weights[- (self.degree + 1) :]
        else:
            current_controlpoints = \
                self.controlpoints[index - self.degree:index - r + 1]
            current_weights = self.weights[index - self.degree:index - r
                + 1]
        return current_controlpoints, current_weights

    def get_index(self, u):
        if u == self.grid[-1]:
            index = (self.grid < u).argmin() - 1
        else:
            index = (self.grid > u).argmax() - 1
        return index

    def plot(self, title, points=500, controlpoints=True):
```

```
        fig = plt.figure()
        ax = fig.add_subplot(111)
        ulist = scipy.linspace(self.grid[0], self.grid[-1], points)
        ax.plot(*zip(*[self(u) for u in ulist]), label='NURBS curve')
        if controlpoints:
            ax.plot(*zip(*self.controlpoints), 'o--', label='Control
                Points')

        lgd = ax.legend(loc='best')
        plt.title(title)
        return fig

if __name__ == '__main__':
    controlpoints = scipy.array([[-1, 0],
                                 [-1, 1],
                                 [0, 1],
                                 [1, 1],
                                 [1, 0],
                                 [1, -1],
                                 [0, -1],
                                 [-1, -1],
                                 [-1, 0]])

    grid = scipy.array([0, 0, 0, 0.25, 0.25, 0.5, 0.5, 0.75, 0.75, 1, 1,
        1])

    weights = scipy.ones(len(controlpoints))
    for i in [1, 3, 5, 7]:
        weights[i] = (2 ** 0.5) / 2

    nurbscurve = NURBS(controlpoints=controlpoints,
                       degree=2,
                       grid=grid,
                       weights=weights)
    fig = nurbscurve.plot('Circle constructed using NURBS')
    fig.show()
```

Appendix II

```
import scipy
from matplotlib import pyplot as plt
import numpy as np


class Bspline:
    def __init__(self, knots, degree=None, controlpoints=None):
        self.knots = knots

        if controlpoints is not None:
            self.controlpoints = controlpoints
            self.degree = len(self.knots) - 1 - len(self.controlpoints)
            if degree:
                assert degree == self.degree, \
                       'Given degree is wrong. Check the knots and
                           control' + \
                       'points or do not define a degree yourself.'
        else:
            self.degree = degree

    def __call__(self, u):
        """
        Evaluates the spline at a point u, using the spline definition.
        """
        S = sum([self.controlpoints[i] *
            self.get_basisfunc(k=self.degree,j=i)(u)
                for i in range(len(self.controlpoints))])
        return S

    def has_full_support(self, u):
        """
        This method checks if the point u is in an interval with full
            support.
        """
        if min(scipy.count_nonzero(self.knots < u),
              scipy.count_nonzero(self.knots > u)) > self.degree:
            return True
        else:
            return False

    def get_basisfunc(self, k, j):
        """
        Method that returns a function which evaluates the basis
            function of
        degree k with index j at point u.
        """
        def basisfunction(u, k=k, j=j):
```

```
        """
        Method to evaluate the the basis function N^k with index j at
        point u.
        u (float): the point where to evaluate the basis function
        k (int): the degree of the basis function
        j (int): the index of the basis function we want to evaluate
        knots (array): knot sequence u_i, where i=0,...,K
        """
        if k == 0:
            if self.knots[j] <= u <= self.knots[j+1]:
                return 1
            else:
                return 0
        else:
            try:
                a0 = 0 if self.knots[j+k] == self.knots[j] \
                      else (u - self.knots[j]) / (self.knots[j+k] -
                                                  self.knots[j])
                a1 = 0 if self.knots[j+k+1] == self.knots[j+1] \
                      else (self.knots[j+k+1] - u) /
                           (self.knots[j+k+1] -
                                                    self.knots[j+1])
                basisfunc = a0 * basisfunction(u, k=k-1, j=j) + \
                           a1 * basisfunction(u, k=k-1, j=j+1)
            except IndexError:
                numBasisfunc = len(self.knots) - 1 - k
                raise IndexError('Invalid index. There are no more
                    than {} basis functions for the given problem,
                    choose an ' \
                           'index lower than the number of basis
                               functions.'.format(numBasisfunc))

            return basisfunc
    return basisfunction

def plot(self, points=None):
    """
    This method plots the spline.
    """
    ulist = scipy.linspace(self.knots[0], self.knots[-1], 1000)
    ulist = [u for u in ulist if self.has_full_support(u=u)]
    plt.plot(*zip(*[self(u=u) for u in ulist]))
    plt.plot(*zip(*self.controlpoints), 'o--', label='control
        points')
    plt.plot(*zip(*[self(u=u) for u in self.knots]), 'x',
        label='knots')
    if points is not None:
        plt.plot(*zip(*points), 'o', label='Interpolation Points')
    plt.legend(loc='upper left')
    plt.grid()
```

```
        plt.title('B-spline curve and its control polygon')
        plt.show()

class interpolation:
    def __init__(self, points, domain):
        self.points = points
        self.a = domain[0]
        self.b = domain[1]
        self.s = len(points) - 1

    def get_parameter_values(self, method):
        if method == 'uniform':
            return self.uniform_method()
        elif method == 'chord':
            return self.chord_method()
        elif method == 'centripetal':
            return self.centripetal()

    def uniform_method(self):
        t = scipy.array([])
        for k in range(self.s + 1):
            if k == 0:
                val = self.a
            elif k == self.s:
                val = self.b
            else:
                val = self.a + k * (self.b - self.a)/self.s
            t = np.append(t,val)
        return t

    def chord_method(self):
        t = scipy.array([])
        for k in range(self.s + 1):
            if k == 0:
                val = self.a
            elif k == self.s:
                val = self.b
            else:
                Lk = sum([np.linalg.norm(self.points[i] - self.points[i -
                    1]) for i in range(1,k+1)]) / \
                    sum([np.linalg.norm(self.points[i] - self.points[i -
                        1]) for i in range(1,self.s+1)])
                val = self.a + Lk * (self.b - self.a)
            t = np.append(t,val)
        return t

    def centripetal(self):
        t = scipy.array([])
        for k in range(self.s + 1):
            if k == 0:
```

```
                val = self.a
            elif k == self.s:
                val = self.b
            else:
                Lk = sum([np.sqrt(np.linalg.norm(self.points[i] -
                    self.points[i - 1])) for i in range(1,k+1)]) / \
                    sum([np.sqrt(np.linalg.norm(self.points[i] -
                        self.points[i - 1])) for i in range(1,self.s+1)])
                val = self.a + Lk * (self.b - self.a)
            t = np.append(t,val)
        return t

    def get_knots(self, degree, parameters):
        numKnots = degree + self.s + 2
        knots = scipy.array([])
        for j in range(numKnots):
            if j <= degree:
                knots = np.append(knots, 0)
            elif j >= self.s + 1:
                knots = np.append(knots, 1)
            else:
                jj = j - degree
                u = 1/degree * sum([parameters[i] for i in range(jj,jj +
                    degree - 1 + 1)])
                knots = np.append(knots, u)
        return knots

def get_controlpoints(points, tvals, knots, degree):
    s = len(points) - 1
    N = scipy.zeros((s+1,s+1))
    bspline = Bspline(knots)
    for i in range(s+1): # columns
        basisfunc = bspline.get_basisfunc(degree,i)
        for j in range(s+1): # rows
            N[j,i] = basisfunc(tvals[j])
    print(N)
    controlpoints = np.linalg.lstsq(N,points)[0]
    print(len(controlpoints))
    return controlpoints


if __name__ == '__main__':
    methods = ['uniform', 'chord', 'centripetal']
    points = scipy.array([[0,0],
                          [6,10],
                          [7,10.2],
                          [9,8]])
    degree = [2,3]
    interpolate = interpolation(points, [0, 1])
    fig, axarr = plt.subplots(len(degree),1, figsize=(8,8))
```

```
for i, deg in enumerate(degree):
    for method in methods:
        parameters = interpolate.get_parameter_values(method)
        knots = interpolate.get_knots(deg,parameters)
        controlpoints = get_controlpoints(points,parameters,knots,
            deg)
        bspline = Bspline(knots,controlpoints=controlpoints)

        ulist = scipy.linspace(knots[0], knots[-1], 300)
        ulist = [u for u in ulist if bspline.has_full_support(u=u)]
        axarr[i].plot(*zip(*[bspline(u=u) for u in ulist]),
            label=method)
        axarr[i].plot(*zip(*controlpoints), 'o--', label='{} -
            Control Points'.format(method))
        axarr[i].plot(*zip(*[bspline(u=u+1e-10) for u in knots]),
            'x', label='{} - Knots'.format(method))
    axarr[i].plot(*zip(*points), 'o', label='Interpolation Points')
    axarr[i].set_title('B-Spline Curve of Degree {} that
        Interpolates \n '
            '$(0,0$, $(6,10)$, $(7,10.2)$ and $(9,8)$'.format(deg))
    lgd = axarr[i].legend(loc='upper left', bbox_to_anchor=(1, 1))
    axarr[i].grid()
fig.tight_layout()
fig.savefig('task2', bbox_extra_artists=(lgd,), bbox_inches='tight')
plt.show()
```

Appendix III

```
import scipy
import numpy
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

class RationalBezierCurve(object):
    """
    This is a class for Bzier curves.
    """
    def __init__(self, controlpoints, weights):
        """
        An object of the class is initialized with a set of control
            points in
        the plane.
        """
        self.weights = weights
        self.original_controlpoints = controlpoints
        self.controlpoints = self.create_new_controlpoints(controlpoints)


    def __call__(self, t):
        """
        This method returns the point on the line for some t.
        """
        deCasteljauArray = self.get_deCasteljauArray(t)
        # turn into homogeneous coordinates
        point = deCasteljauArray[-1, -1]
        return 1/point[-1] * point

    def create_new_controlpoints(self, controlpoints):
        return
            scipy.array([numpy.append(controlpoints[i]*self.weights[i],
            self.weights[i])
                          for i in range(len(controlpoints))])

    def get_deCasteljauArray(self, t):
        """
        This method calculates and returns a matrix with the lower left
            corner
        containing the de Casteljau array, calculated for the specified
            t.
        """
        # initializing the array
        dim = numpy.shape(self.controlpoints)
        deCasteljauArray = scipy.zeros((dim[0],dim[0],dim[1]))
        deCasteljauArray[0] = self.controlpoints
```

```
        # filling the array
        for i in range(1, len(deCasteljauArray)):
            for j in range(i, len(deCasteljauArray)):
                deCasteljauArray[i,j] = (
                        (1 - t) * deCasteljauArray[i-1, j-1] +
                        t * deCasteljauArray[i-1,j])
        return deCasteljauArray


if __name__ == '__main__':
    ### Task 4 ###
    controlpoints = scipy.array([[0,0],
                                [4,3],
                                [3,1],
                                [5,1]])
    weights = scipy.array([1,2,5,4])

    # (index of changed weight, the new values)
    changed_weights3 = (2, scipy.array([0,2,3,4,6]))
    changed_weights4 = (-1, scipy.array([0,2,4,6,8]))
    fig, axarr = plt.subplots(2,1,sharex='col')
    for i,plot in enumerate((changed_weights3,changed_weights4)):
        tlist = scipy.linspace(0, 1, 300)
        for new_weight in plot[1]:
            weights[plot[0]] = new_weight
            curve = RationalBezierCurve(controlpoints, weights)
            axarr[i].plot(*zip(*[curve(t)[:2] for t in tlist]),
                label='Weight w={}'.format(new_weight))

        axarr[i].plot(*zip(*curve.original_controlpoints), 'o--',
            label='Controlpoints')
        lgd = axarr[i].legend(loc='upper left', bbox_to_anchor=(1, 1))
        title = 'Rational Bzier Curve with Control Points $(0,0)$,
            $(4,3)$, $(3,1)$, $(5,1)$ \n ' \
                'and Varying Weights $[1,2,{},{}]$'.format('w' if i == 0
                    else 3,
                                                    'w' if i ==1 else
                                                        4)
        axarr[i].set_title(title)
    fig.subplots_adjust(hspace=0.5)
    fig.savefig('task4', bbox_extra_artists=(lgd,), bbox_inches='tight')
    plt.show()
```