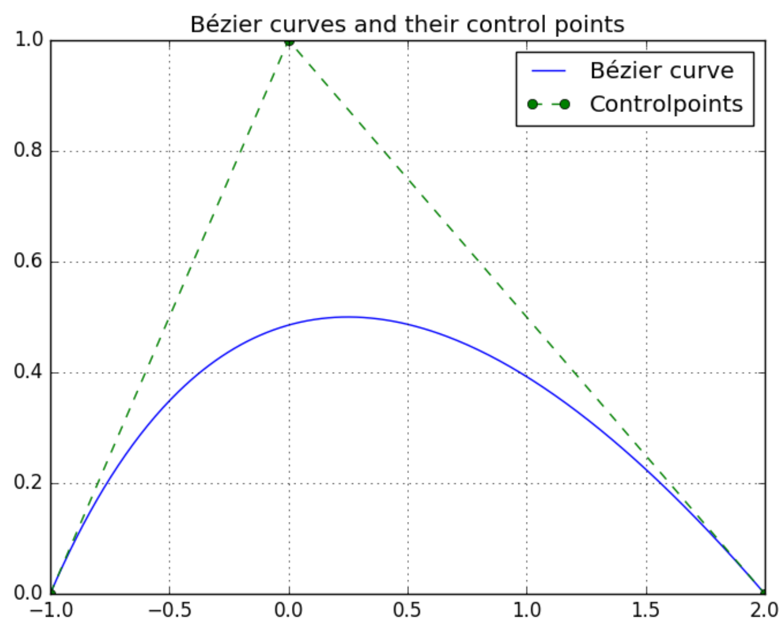
CAGD - Homework 2

Josefine Stål & Erik Ackzell

September 20, 2016

Task 1

In this task we implement subdivision to split a Bézier curve into two different Bézier curves. This is implemented as a method of a Python class, see Appendix 1. Details can be seen in the code. We then test our code by defining a Bézier curve with control points $(-1,0),(0,1),(2,0)$, subdividing at $t = 0.4$. In our test, the control points of the two new Bézier curves were $(-1,0),(-0.6,0.4),(-0.04,0.48)$ and $(-0.04,0.48),(0.8,0.6),(2,0)$. The original curve can be seen in the figure below.



Task 2

In this task we implement degree elevation for the Bézier curve with the same control points as in task 1. This is implemented as a method of a Python class, see Appendix 1. Details can be seen in the code. In our test, the control points of the Bézier curve from task 1 were used, and the degree was increased to four. The control points of the new curve were $(-1, 0), (-0.5, 0.5), (0.17, 0.7), (1, 0.5), (2, 0)$.

Task 3

In this task, we used the trivial reject approach in order to determine the intersection of a Bézier curve and a line. This is implemented as a method of a Python class along with a rectangle class and a line class, see Appendix 1. Details can be seen in the code. In our test, we used the Bézier curve with control points $(0, 0), (9, -4), (7, 5), (2, -4)$ and the line passing through $(4, 5)$ and $(6, -4)$. The intersections found were $(5.32, -0.94)$ and $(5.13, -0.10)$.

Task 4

Let b and c be two Bézier curves with domain $[0, \frac{1}{3}]$ and $[\frac{2}{3}, 1]$ respectively with control points $\{b_0, b_1, ..., b_n\}$ and $\{c_0, c_1, ..., c_m\}$. Bézier curves are always polynomial curves and hence also C^∞. The question is what is required for the composite of the two curves to be C^1 or just G^1? For a curve to be C^1, C^0 must be satisfied to start with. The continuity of the curve is satisfied when $b_m = c_0$. A part from continuity, the curves must be collinear in the meeting point. That is that the tangent at the meeting point, including the points b_{m-1}, $b_m = c_0$ and c_1, is continuous. Finally the derivative at the meeting point must be continuous.

For G^1 on the other hand, the derivatives must not be equal, it is enough for the derivatives to have the same directions. The collinearity must still be fullfilled.

In our case with the two curves b and c we can observe the two domains which do not overlap. Hence it is impossible for b_n and c_0 to coincide. Therefore we can not have either C^1 or G^1 for the composite of the two curves.

Task 5

A basis is invariant under affine domain transformations if the basis functions remains unchanged after applying the affine map $\phi : [a, b] \to [0, 1]$ with

$$\phi(u) = \frac{u - a}{b - a}$$

Bézier polynomials are invariant under affine domain transformations, let us check if the Monomial and/or the Lagrange basis are invariant as well.

Let $C(x) = \sum_{i=0}^{n} c_i x^i$ and $D(x) = \sum_{i=0}^{n} d_i x^i$ be two functions of Monomial form. Then,

$$\alpha C(x) + (1-\alpha) D(x) = \alpha \sum_{i=0}^{n} c_i x^i + (1-\alpha) \sum_{i=0}^{n} d_i x^i$$

With n being finite we have,

$$= \sum_{i=0}^{n} [\alpha c_i + (1-\alpha) d_i] x_i \qquad \square$$

Hence, the monomial form is invariant under affine domain tranformations.

Now let the basis functions be the Lagrange polynomials

$$L_i^n(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}$$

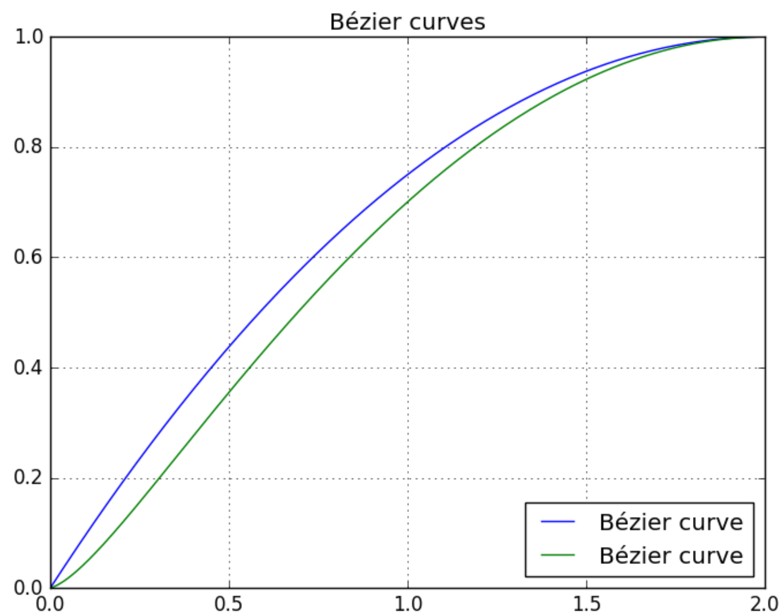
and $C(x) = \sum_{i=0}^{n} c_i L_i^n$ and $D(x) = \sum_{i=0}^{n} d_i L_i^n$. With the same reasoning as above we can show that

$$\alpha C(x) + (1-\alpha) D(x) = \sum_{i=0}^{n} [\alpha c_i + (1-\alpha) d_i] L_i^n(x)$$

and the Lagrange form is indeed invariant under affine domain transformations. $\square$

Task 6

In this task, we change the appearance of a curve (top curve in figure below to the lower curve) by first applying a degree elevation followed by modifying the control points. It is clear that the lower curve can not be a quadratic curve due to its s-shape, that is why the degree elevation is needed. The control points of the original curve is given by $(0,0), (1,1), (2,1)$ and the second curve has control points $(0,0), (0.25, 0.05), (1,1), (2,1)$.


Bézier curves
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.5
1.0
1.5
2.0
Bézier curve
Bézier curve

Appendix I

Code for task 1-3.

```
"""
This program consists of three different class definitions, a line class,
a rectangle class and a Bzier curve class. The line and rectangle classes
are used to implement the trivial reject method to determine the
    intersection
between a Bzier curve and a line.
"""
import numpy
import scipy
import pylab


class rectangle(object):
    """
    This is a rectangle class.
    """
    def __init__(self, xlow, xhigh, ylow, yhigh):
        """
        An object of the class is initialized with two x-values, one for
            the
        lower bound of the rectangle and one for the upper bound, as
            well as
        two y-values, one for the lower bound of the rectangle and one
            for the
        upper bound.
        """
        # corners of the rectangle
        self.corners = scipy.array([[xlow, ylow],
                                   [xlow, yhigh],
                                   [xhigh, yhigh],
                                   [xhigh, ylow]])
        # lower/higher bounds for x and y
        self.xlow = xlow
        self.xhigh = xhigh
        self.ylow = ylow
        self.yhigh = yhigh

    def plot(self):
        """
        This method plots the rectangle.
        """
        # adding the first corner of the list to the end of the corner
            array,
        # for easier plotting
        rectangle_update = scipy.vstack((self.corners, self.corners[0]))
        pylab.plot(rectangle_update[:, 0], rectangle_update[:, 1])
```

```python
    def get_diagonal_length(self):
        """
        This method calculates and returns the length of the diagonal of
            the
        rectangle, using the two-norm.
        """
        return scipy.linalg.norm(self.corners[0] - self.corners[2], 2)

    def get_center(self):
        """
        This method calculates and returns the center of the rectangle.
        """
        xval = 0.5 * (self.xlow + self.xhigh)
        yval = 0.5 * (self.ylow + self.yhigh)
        return scipy.array([xval, yval])


class line(object):
    """
    This is a line class.
    """
    def __init__(self, p, q):
        """
        An object of the class is initialized with two points through
            which
        the line passes.
        """
        self.p = p
        self.q = q
        self.Lx, self.Ly = self.get_functions_from_points(p, q)

    def get_functions_from_points(self, p, q):
        """
        This method returns two polynomials, describing the line as a
            function
        of x and a function of y, respectively.
        """
        # if the line is parallel with the y axis
        if p[0] == q[0]:
            # coefficients for polynomial of y
            Lycoeff = scipy.polyfit([p[1], q[1]], [p[0], q[0]], 1)
            # no function of x
            Lx = None

            def Ly(y):
                """
                Line as a function of y
                """
                return Lycoeff[0] * y + Lycoeff[1]
```

```python
    # if the line is paralell with the x axis
    elif p[1] == q[1]:
        Lxcoeff = scipy.polyfit([p[0], q[0]], [p[1], q[1]], 1)

        def Lx(x):
            """
            Line as a function of x
            """
            return Lxcoeff[0] * x + Lxcoeff[1]

        Ly = None
    # if the line is non-parallel with neither the x or the y axis
    else:
        Lxcoeff = scipy.polyfit([p[0], q[0]], [p[1], q[1]], 1)
        Lycoeff = scipy.polyfit([p[1], q[1]], [p[0], q[0]], 1)

        def Lx(x):
            """
            Line as a function of x
            """
            return Lxcoeff[0] * x + Lxcoeff[1]

        def Ly(y):
            """
            Line as a function of y
            """
            return Lycoeff[0] * y + Lycoeff[1]

    return Lx, Ly

def crosses_line_segment(self, segmentpoints):
    """
    This method checks whether the line intersects a line segment,
        parallel
    to the x or y axis.
    """
    # if the line segment is parallel to the x axis
    if segmentpoints[0, 1] == segmentpoints[1, 1]:
        # if the line is not parallel to the y axis
        if self.Lx:
            # check if line crosses the line segment
            if (
                    (self.Lx(segmentpoints[0, 0]) - segmentpoints[0, 1])
                        *
                    (self.Lx(segmentpoints[1, 0]) - segmentpoints[0,
                        1])) <= 0:
                return True
            else:
                return False
        else:
```

```
            if segmentpoints[0, 0] <= self.p[0] <= segmentpoints[1,
                 0]:
                return True
            else:
                return False
    # if the line segment is parallel to the y axis
    else:
        # if the line is not parallel to the x axis
        if self.Ly:
            if (
                 (self.Ly(segmentpoints[0, 1]) - segmentpoints[0, 0]) *
                 (self.Ly(segmentpoints[1, 1]) - segmentpoints[0, 0]))
                     <= 0:
                return True
            else:
                return False
        else:
            if segmentpoints[0, 1] <= self.p[1] <= segmentpoints[1,
                 1]:
                return True
            else:
                return False

def intersects_rectangle(self, rectangle):
    """
    This method checks whether the line intersects a rectangle.
    """
    result = False
    # check if the line intersects any of the sides of the rectangle,
    # the first three sides
    for i in range(3):
        segmentpoints = rectangle.corners[i:i+2]
        if self.crosses_line_segment(segmentpoints=segmentpoints):
            result = True
    # checking the last side
    segmentpoints = scipy.array([rectangle.corners[0],
                                 rectangle.corners[3]])
    if self.crosses_line_segment(segmentpoints=segmentpoints):
        result = True
    return result

def plot(self, parammin, parammax):
    """
    This method plots the line.
    """
    paramlist = scipy.linspace(parammin, parammax, 200)
    if self.Lx:
        vallist = [self.Lx(x) for x in paramlist]
        pylab.plot(paramlist, vallist)
    else:
```

```
        vallist = [self.Ly(y) for y in paramlist]
        pylab.plot(vallist, paramlist)


class beziercurve(object):
    """
    This is a class for Bzier curves.
    """
    def __init__(self, controlpoints):
        """
        An object of the class is initialized with a set of control
            points in
        the plane.
        """
        self.controlpoints = controlpoints
        self.xlow = min(self.controlpoints[:, 0])
        self.xhigh = max(self.controlpoints[:, 0])
        self.ylow = min(self.controlpoints[:, 1])
        self.yhigh = max(self.controlpoints[:, 1])

    def __call__(self, t):
        """
        This method returns the point on the line for some t.
        """
        deCasteljauArray = self.get_deCasteljauArray(t)
        return deCasteljauArray[-1, -2:]

    def subdivision(self, t):
        """
        This method implements subdivision at t.
        """
        # getting the de Casteljau array using t
        deCasteljauArray = self.get_deCasteljauArray(t)
        # extracting the new controlpoints from the array
        controlpoints1 = scipy.array([deCasteljauArray[i, 2 * i:2 * i+2]
                                    for i in
                                        range(len(self.controlpoints))])
        controlpoints2 = scipy.array([deCasteljauArray[-1, 2 * i:2 * i+2]
                                    for i in
                                        range(len(self.controlpoints))])
        controlpoints2 = controlpoints2[::-1]
        curve1 = beziercurve(controlpoints1)
        curve2 = beziercurve(controlpoints2)

        return (curve1, curve2)

    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
        deCasteljauArray = scipy.column_stack((
                              numpy.copy(self.controlpoints),
                              scipy.zeros((len(self.controlpoints),
                                          2 * len(self.controlpoints) -
                                              2))
                                         ))
        # filling the array
        for i in range(1, len(deCasteljauArray)):
            for j in range(1, i + 1):
                deCasteljauArray[i, j*2:j*2+2] = (
                        (1 - t) * deCasteljauArray[i-1, (j-1)*2:(j-1)*2+2]
                            +
                        t * deCasteljauArray[i, (j-1)*2:(j-1)*2+2])
        return deCasteljauArray

    def intersects_line(self, line1):
        """
        This method checks if the curve intersects a line.
        """
        # rectangle with sides parallel to the x and y axis, containing
            the
        # convex hull of the control points of the curve
        rectangle1 = rectangle(xlow=self.xlow,
                             xhigh=self.xhigh,
                             ylow=self.ylow,
                             yhigh=self.yhigh)
        # initial check
        if not line1.intersects_rectangle(rectangle1):
            return False
        else:
            # list of rectangles
            rectangle_list = [rectangle1]
            # list of intersections
            intersection_list = []
            # list of curves
            curve_list = [self]
            # as long as there are rectangles in the rectangle list
            while rectangle_list:
                # list to update curve list with
                updated_curve_list = []
                # list to update rectangle list with
                updated_rectangle_list = []
                # going through all the curves in the list
                for C in curve_list:
                    # subdividing
                    C1, C2 = C.subdivision(t=0.5)
```

```
                    # getting rectangles corresponding to the new curves
                    R1 = rectangle(xlow=C1.xlow,
                                   xhigh=C1.xhigh,
                                   ylow=C1.ylow,
                                   yhigh=C1.yhigh)
                    R2 = rectangle(xlow=C2.xlow,
                                   xhigh=C2.xhigh,
                                   ylow=C2.ylow,
                                   yhigh=C2.yhigh)
                    # checking if the line intersects any of the new
                        rectangles
                    for RC in [(R1, C1), (R2, C2)]:
                        # if intersection, use to update rectangle/curve
                            list
                        if line1.intersects_rectangle(RC[0]):
                            updated_rectangle_list.append(RC[0])
                            updated_curve_list.append(RC[1])
                # update curve and rectangle lists
                curve_list = updated_curve_list
                rectangle_list = updated_rectangle_list
                # list of indices of rectangles and curves to remove from
                # the rectangle and curve list
                poplist = []
                for i, R in enumerate(rectangle_list):
                    # if the rectangles are very small
                    if R.get_diagonal_length() < 1e-7:
                        intersection_list.append(R.get_center())
                        poplist.append(i)
                # remove elements from rectangle and curve lists
                for i in poplist[::-1]:
                    rectangle_list.pop(i)
                    curve_list.pop(i)
            # list of indices of calculated intersections to remove
            poplist = []
            for i, I in enumerate(intersection_list[:-2]):
                for j, I2 in enumerate(intersection_list[i + 1:]):
                    # if the distance of intersections is too small
                    if scipy.linalg.norm(I - I2, 2) < 1e-7:
                        poplist.append(j + i)
            # remove duplicates
            for i in poplist[::-1]:
                intersection_list.pop(i)
            return intersection_list

    def degree_elevation(self):
        """
        This method implements degree elevation.
        """
        n = len(self.controlpoints)
        # initializing the array to hold control points
```

```
        new_controlpoints = scipy.zeros((n + 1, 2))
        # the first and last control points are the same as before
        new_controlpoints[0] = numpy.copy(self.controlpoints[0])
        new_controlpoints[-1] = numpy.copy(self.controlpoints[-1])
        # calculating the new control points
        for i in range(1, n):
            new_controlpoints[i] = (
                        (1 - i/n) * numpy.copy(self.controlpoints[i]) +
                        (i/n) * numpy.copy(self.controlpoints[i - 1])
                                  )
        return beziercurve(new_controlpoints)

    def plot(self, controlpoints=True):
        """
        This method plots the curve.
        """
        # list of u values for which to plot
        tlist = scipy.linspace(0, 1, 300)
        pylab.plot(*zip(*[self(t) for t in tlist]), label='Bzier curve')
        title = 'Bzier curves'
        if controlpoints: # checking whether to plot control points
            pylab.plot(*zip(*self.controlpoints), 'o--',
                label='Controlpoints')
            title += ' and their control points'
        pylab.legend()
        pylab.title(title)
```