Assignment 1
MATINF4170
Spline methods

Ivar Haugaløkken Stangeby

January 26, 2017

Exercise 1.3

In this exercise we take a look at the Neville-Aitken algorithm for evaluating points on an interpolating curve. In the general setup we are given $p+1$ control points, $\{c_j\}_{j=0}^{p}$, as well as $p+1$ strictly increasing parameter values $\{t_j\}_{j=0}^{p}$. The integer p ends up being the degree of the interpolating polynomial. The algorithm was implemented in Python and the source code is given in Listing 1. We wish to interpolate N points on the semicircle $\mathcal{C} \coloneqq \{(\cos(t), \sin(t)) \mid t \in [0, \pi]\}$ and study the effect of varying the number of interpolation points N, as well as the effect of varying the parameter values.

We chose two distinct parametrizations, one being the uniform parametrization given by

$$t_{j+1} - t_j = 1$$

for $j = 0, \ldots, N-1$ and with $t_0 = 0$. The other one being tailored to misbehave for increasing N was generated using a cumulative sum of random numbers, that is:

$$t_{j+1} - t_j = r_j$$

where r_j is a random integer in $\mathbb{Z}_5$ and $t_0 = 0$. The uniform parametrization behaved quite nicely under increasing N, as can be seen in Figure 1. The random parametrization on the other hand was very unstable under increasing N and hence yields a very bad interpolation to the semi circle for large N.

Exercise 1.4

In this exercise we look at the de Casteljau algorithm for computing points on a Bézier curve. It is very similar to the Neville-Aitken algorithm, however there is one big difference. While the Neville-Aitken algorithm parametrizes on adjacent parameter intervals, yielding convex combinations only in the last step of the algorithm, the de Casteljau algorithm parametrizes each line segment

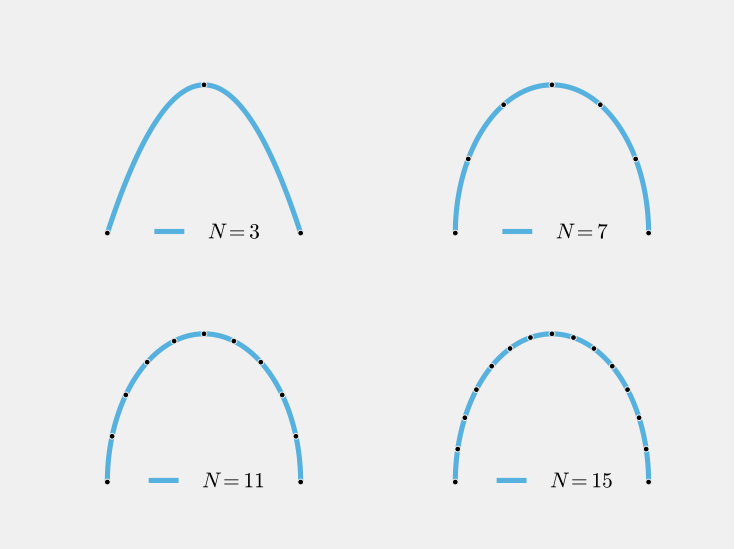

Figure 1: Uniform parametrization of the parameter values, i.e.,

$$\{t_j\}_{j=0}^{N-1} = \{0, 1, \ldots, N-1\}\,.$$

As N increases, we see that the interpolating polynomial converges fairly quickly to the unit semi circle. The uniform parametrization yields a good interpolation.

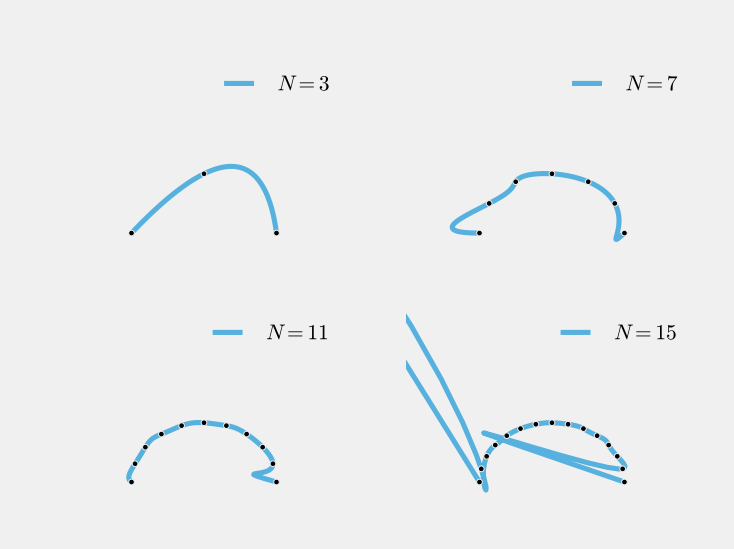

Figure 2: This parametrization was generated using a random generator. The parameter values for $N = 15$ ended up being

$$\{t_j\}_0^{14} = \{2, 4, 5, 6, 8, 10, 12, 14, 16, 17, 18, 20, 21, 23, 25\}\,,$$

and for lower N a subsequence of this was used.

```python
def NevAit(T, c_points, t_points, eps=1.0E-14):
    """
    The Neville Aitkens Given p + 1 control points c_points and p+1
    strictly increasing parameter values t_points, computes the polynomial
    curve of degree p that interpolates the control points.
    """
    t = t_points
    c = np.array(c_points, dtype=np.float64)
    p = len(c) - 1
    for k in range(1, p + 1):
        for j in range(0, p - k + 1):
            denum = (t[j+k] - t[j])
            if abs(denum) <= eps:
                c[j] = 0
                continue
            l_one = (t[j+k] - T) / denum
            l_two = (T - t[j]) / denum
            c[j] = l_one*c[j] + l_two * c[j+1]
    return c[0]
```

Listing 1: The Neville-Aitken algorithm implemented in PYTHON. The algorithm uses in-place replacement of the newly computed values in order to save memory, as well as time. No other numerical optimizations has been made.

over the same interval. This has the added benefit of convex combinations in every step of the algorithm and yields higher numerical stability. Due to this, the resulting curve interpolates only the first and the last control point, hence the interpolatory nature of the Neville-Aitken is lost. The algorithm was implemented in PYTHON and the code can be seen in Listing 2.

Using the same data as in the previous exercise the resulting curve can be seen in Figure 3. As we can see the interpolation property has been lost for the interior control points.

```
def deCasteljau(T, c_points, interval_start=0, interval_stop=1):
    """
    The de Casteljau algorithm.
    Given p + 1 control points c_points, computes the point on the bezier curve
    at parameter value T.
    """
    T = (interval_stop - T) / float(interval_stop - interval_start)
    c = np.array(c_points, dtype=np.float64)
    p = len(c_points) - 1

    for k in range(1, p+1):
        for j in range(0, p - k + 1):
            c[j] = (1-T)*c[j] + T * c[j+1]

    return c[0]
```

Listing 2: The de Casteljau algorithm for computing points on a Bézier curve. Again, the implementation uses in place replacement of newly computed values. No other numerical optimizations has been made.

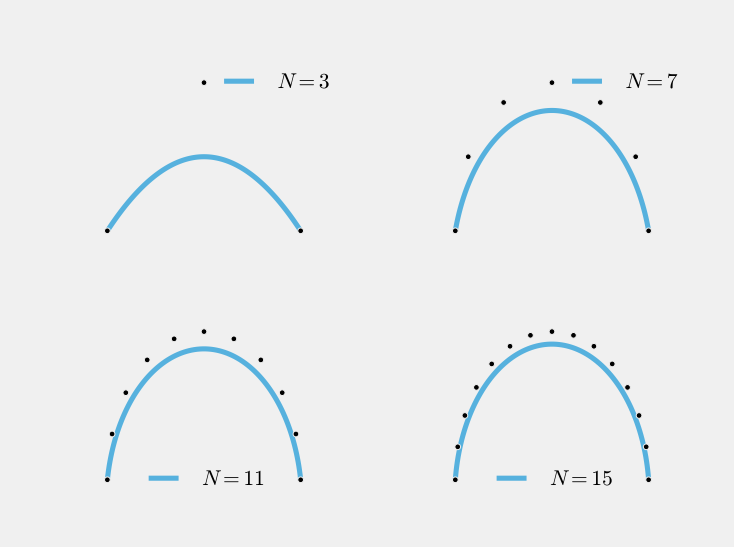

Figure 3: Computing convex combinations of the line segments yields what we call a Bézier curve. The curve has nice geometric properties due to it always lying in the convex hull of its control points. It therefore resembles the control polygon nicely. It does however not interpolate the interior control points.