Assignment 2
MATINF4170
Spline methods

Ivar Haugaløkken Stangeby

March 1, 2017

Abstract

In this assignment we explore further properties of B-splines, especially related to multiple knots and evaluation at knots. We also implement an algorithm for evaluating a spline function f at a parameter x, stemming from the spline matrix representation of spline functions.

Exercise 2.2

In this exercise we want to find the individual polynomial pieces of the cubic B-splines:

i) $B[0,0,0,0,1](x)$;

ii) $B[0,1,1,1,1](x)$;

iii) $B[0,1,1,1,2](x)$.

Instead of computing these directly, we first show the general claim that

$$B[a,\overbrace{b,\dots,b}^{p},c](x) = \frac{(x-a)^p}{(b-a)^p}B[a,b](x) + \frac{(c-x)^p}{(c-b)^p}B[b,c](x),$$

outlined in exercise 2.7 (note that we do allow $a = b$, $b = c$).

Base case We proceed by induction, and the base case consists of $p = 0$. The right hand side then directly evaluates to $B[a,b](x) + B[b,c](x)$, which is equal to $B[a,c]$ as the knot intervals are adjacent.

Induction step For the inductive step, we assume the claim holds for $p = k-1$, and show it must also hold for $p = k$. We then have

$$B[a,\overbrace{b,\dots,b}^{k},c](x) = \frac{(x-a)}{(b-a)}B[a,\underbrace{\overbrace{b,\dots,b}^{k-1},b}_{k}](x) + \frac{(c-x)}{(c-b)}B[\underbrace{b,\overbrace{b,\dots,b}^{k-1}}_{k},c](x),$$

and applying the induction hypothesis yields

$$= \frac{(x-a)^k}{(b-a)^k}B[a,b](x) + \frac{(c-x)^k}{(c-b)^k}B[b,c](x).$$

This closes the induction. It then follows that

i) $B[0,0,0,0,1](x) = (1-x)^3 B[0,1](x)$;

ii) $B[0,1,1,1,1](x) = x^3 B[0,1](x)$;

iii) $B[0,1,1,1,2](x) = x^3 B[0,1](x) + (2-x)^3 B[1,2](x)$,

using the zero-convention. This can be verified by direct computation.

When it comes to smoothness, if we have a knot t_j of multiplicity m, then the derivatives of the spline $B_{j,p}$ of order $0, 1, \dots, p-m$ are continuous at the knot t_j. For the above B-splines, we have knot multiplicity of 4, 4 and 3 respectively, hence for $B[0,0,0,0,1]$ we have a discontinuity at $x = 0$ as $p-m = 3-4 = -1$. This can also be seen by comparing the two limits at 0. By the same token, $B[0,1,1,1,1]$ has a discontinuity at $x = 1$. The B-spline $B[0,1,1,1,2]$ on the other hand exhibits C^0 continuity at the knot $x = 1$.

Exercise 2.6

Induction proof

We wish to show the identity

$$B(t_i \mid t_j, \dots, t_{j+p+1}) = B(t_j \mid t_i, \dots, t_{i-1}, t_{i+1}, \dots, t_{j+i+p}) \tag{1}$$

holds for $i = j, \dots, j+p+1$.

Base case: Note that the B-spline on the right hand side is one degree lower than that on the left. Hence, our base case consists of $p = 1$. With $p = 1$ we have three cases: $t_i = t_j, t_{j+1}$ and t_{j+2}. In general we have

$$B(x \mid t_j, t_{j+1}, t_{j+2}) = \frac{x - t_j}{t_{j+1} - t_j} B(x \mid t_j, t_{j+1}) + \frac{t_{j+2} - x}{t_{j+2} - t_{j+1}} B(x \mid t_{j+1}, t_{j+2}).$$

Plugging in $x = t_j$ both terms evaluates to zero, and since t_j is outside the interval $[t_{j+1}, t_{j+2})$, the right hand side of 1 also evaluates to zero. By symmetry, the same holds for $x = t_{j+2}$. For $x = t_{j+1}$, we get $B(x \mid t_j, t_{j+1}) = 0$, while $B(x \mid t_{j+1}, t_{j+2}) = 1$. By definition, the right hand side of Equation (1) also evaluates to 1. This concludes the base case.

Induction step: For the inductive step, we assume that Equation (1) holds for degree $p-1$. For degree p, the left hand side of Equation (1) then reads:

$$B(t_i \mid t_j, \dots, t_{j+p+1}) = \frac{t_i - t_j}{t_{j+p} - t_j} B(t_i \mid t_j \dots t_{j+p}) + \frac{t_{j+p+1} - t_i}{t_{j+p+1} - t_{j+1}} B(t_i \mid t_{j+1} \dots t_{j+p+1}),$$

which by the induction hypothesis yields

$$\frac{t_i - t_j}{t_{j+p} - t_j} B(t_i \mid t_j \dots, t_{i-1}, t_{i+1}, \dots, t_{j+p}) + \frac{t_{j+p+1} - t_i}{t_{j+p+1} - t_{j+1}} B(t_i \mid t_{j+1}, \dots, t_{i-1}, t_{i+1}, \dots, t_{j+p+1}).$$

Evaluating the right hand side of Equation (1) using the recursive definition, yields the same result. This closes the induction.

Application

We use the result proved above to evaluate the second order B-spline $B_{j,2}$ at its interior knots, t_{j+1} and t_{j+2}. The first interior knot gives

$$B(t_{j+1} \mid t_j, \ldots, t_{j+3}) = B(t_{j+1} \mid t_j, t_{j+2}, t_{j+3}) = \frac{t_{j+1} - t_j}{t_{j+2} - t_j} \underbrace{B(t_{j+1} \mid t_j, t_{j+2})}_{=1} = \frac{t_{j+1} - t_j}{t_{j+2} - t_j}.$$

For the second interior knot, we have

$$B(t_{j+2} \mid t_j, \ldots, t_{j+3}) = B(t_{j+2} \mid t_j, t_{j+1}, t_{j+3}) = \frac{t_{j+3} - t_{j+2}}{t_{j+3} - t_{j+1}} \underbrace{B(t_{j+2} \mid t_{j+1}, t_{j+3})}_{=1} = \frac{t_{j+3} - t_{j+2}}{t_{j+3} - t_{j+1}}.$$

Repeated interior knots

Assume now that we all interior knots are equal to some number z, i.e., we have knot multiplicity p at interior knots. That is $t_j < z < t_{j+p+1}$. By repeated use of the above result, we have that the jth B-spline can be evaluated as such:

$$B(z \mid t_j, \underbrace{z, \ldots, z}_{p}, t_{j+p+1}) = B(z \mid t_j, t_{j+p+1}) = 1$$

as z lies in the interval $[t_j, t_{j+p+1})$. If we instead consider the B-splines $B_{i,p}$ where $i \neq j$ evaluated at z, then we have two cases:

i) No equal knots, in which case $B_{i,p}(z) = 0$,

ii) $k \leqslant p$ equal knots, either at the left or right end of the active knots:

$$B(z \mid t_i, \ldots, t_{i+p-k}, \underbrace{z, \ldots, z}_{k}) = B(z \mid t_i, t_{i+p-k}) = 0$$

or

$$B(z \mid \underbrace{z, \ldots, z}_{k}, t_{i+k}, \ldots, t_i + p + 1) = B(z \mid t_{i+k}, \ldots, t_{i+p+1}) = 0$$

by repeated application of above result.

Exercise 2.11

Given a knot vector $t = \{t_j\}_{j=1}^{n+p+1}$ and a real number $x \in [t_1, \ldots, t_{n+p+1})$ we often find ourselves needing to determine exactly what knot interval the number x lies in, e.g., determine the index μ such that $t_\mu \leqslant x < t_{\mu+1}$. There are many ways of achieving this, and sometimes one can tailor the procedure to current context in order to achieve better performance. We examine a few methods, demonstrated in PYTHON, and for all of these we assume $x \in [t_1, \ldots, t_{n+p+1})$. These have yet to be tested in a proper setting.

Naive approach More often than not, when we repeatedly evaluate B-splines it is because we want to compute a set of points on the curve. We can use this, as well as the fact that knot sequences are increasing to our advantage. Letting the function start searching at the previous index, we significantly reduce the number of iterations.

```python
def IndexEvaluation(x, knots, previous=0):
    for i in range(previous, len(knots-1)):
        if knots[i] <= x < knots[i+1]:
            return i
```

Binary approach We could also hope to combine the binary search method with the one using the previous index as a starting point, to further narrow down our search range.

```
def IndexBinaryEvaluation(x, knots, previous=0):
    a = previous
    b = len(knots)-1
    while a <= b:
        c = (a + b) // 2
        if knots[c] <= x < knots[c+1]:
            return c
        else:
            if x < knots[c]:
                b = c - 1
            else:
                a = c + 1
```

Exercise 2.13

In this exercise we want to implement the algorithm for evaluating a spline function f given by

$$f(x) = \sum_{i=1}^{n} c_i B_{i,p}(x)$$

where n is the number of basis functions of degree p, and c_i are the given spline coefficients. The code was implemented in PYTHON, and can be seen in Listing 1. If no μ was supplied, one of the above routines for determining μ is used. The algorithm was used to compute and visualize the cubic *variation diminishing spline approximation* $Q[f]$ to the function $f(x) = \sin(x)$ on the clamped knot vector

$$t = (-1, -1, -1, -1, 0, 1, 2, 3, 4, 5, 5, 5, 5),$$

and can be seen in Figure 1.

Exercise 2.17

In this exercise, we examine what happens when the algorithm in exercise 2.13 is called with an x *not* in the knot interval $[t_\mu, t_{\mu+1})$. I found it hard to determine exactly what is computed by the algorithms in this case, so following is a general discussion.

Some of the benefits of the spline algorithms are that all combinations are *convex*. By applying the spline algorithms to an x outside of the relevant knot interval $[t_\mu, t_{\mu+1})$, we lose this property, and we get weird results that depend on exactly which interval x lies in.

Take for instance the last step in Algorithm 2.20, that is

$$f(x) = \mathbf{c}_p = \begin{pmatrix} \frac{t_{\mu+1}-x}{t_{\mu+1}-t_\mu} & \frac{x-t_\mu}{t_{\mu+1}-t_\mu} \end{pmatrix} \begin{pmatrix} c_{\mu-1,p-1} \\ c_{\mu,p-1}. \end{pmatrix}$$

If we happen to have $x < t_\mu$, then the left matrix coefficient becomes greater than one, while the right one becomes negative, and hence $c_{\mu-1,p-1}$ is weighted higher than $c_{\mu,p-1}$. Similarly, for $x > t_{\mu+1}$ the exact opposite happens. Affine, non-convex combinations like these, for every step of the algorithm, does not suit computers well and the results become unstable.

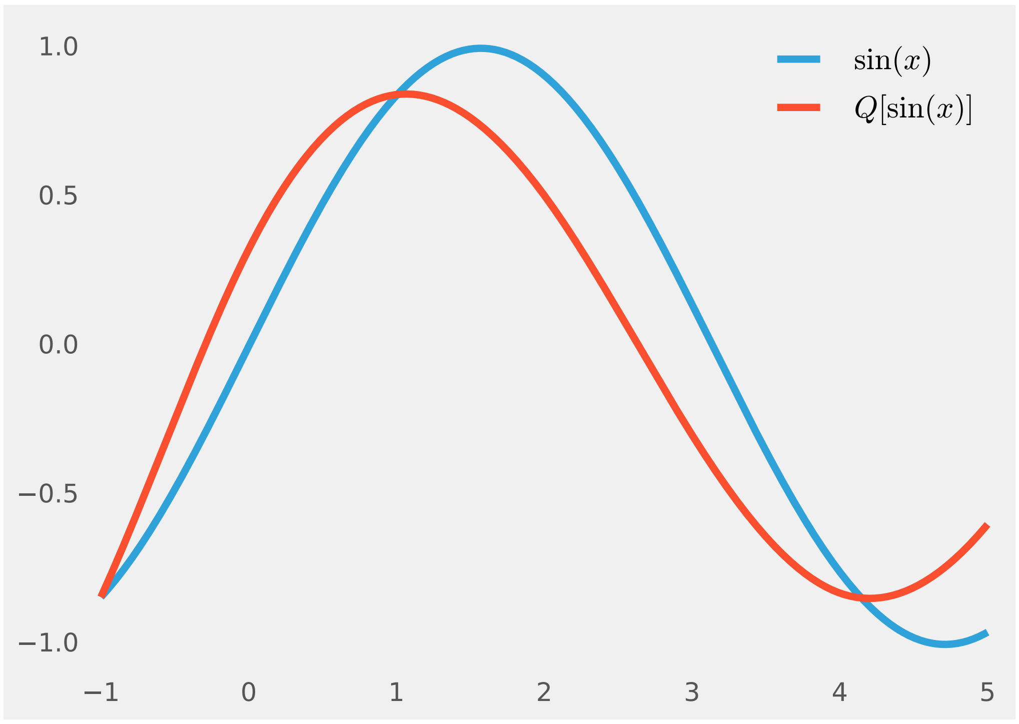

Figure 1: The variation diminishing spline approximation to $\sin(x)$. This was computed using the algorithm implemented in Listing 1. The n coefficients were computed according to the rule $t_j^\star = (t_{j+1} + t_{j+2})/2$ for $j = 1, \dots, n$, where $c_j = \sin(t_j^\star)$.

```python
def SplineEvaluation(x, p, knots, coefficients, mu=None):
    """
    Given a polynomial degree p, a list of n + p + 1 knots, a list of n
    coefficients, and a parameter x. If no index mu is supplied, find the
    appropriate one using a method. Returns the value the spline function f
    with given coefficients at the parameter x.
    """
    if mu is None:
        mu = index(x, knots)
    knots = np.array(knots, dtype=np.float64)
    c = np.array(coefficients[mu - (p):mu + 1], dtype=np.float64)
    for i in range(0, p):
        k = p - i
        t1 = knots[mu - k + 1:mu + 1]
        t2 = knots[mu + 1:mu + k + 1]
        omega = (x - t1) / (t2 - t1)
        c = (1 - omega) * c[:-1] + omega * c[1:]
    return c
```

Listing 1: An implementation of the spline function evaluation algorithm using vector operations as defined by the numpy-library. If no μ is given, it finds a suitable μ using one of the routines discussed in Exercise 2.11. Numpy views are used to save memory.

Exercise 2.19

Given the knot vector $t = (0, 1, 2, 3, 4)$, we can only define *one* B-spline of degree three. Say we want to evaluate $B_{1,3,t}(x)$ for some $x \in (0, 4)$. We can do this using the recursive definition, however, if we wish to evaluate this using Algorithm 2.20 or Algorithm 2.21, we run into trouble. In order to evaluate the splines at a point $x \in [t_\mu, t_{\mu+1})$ the algorithms are reliant on the knots $t_{\mu-p+1}, \dots, t_{\mu+p}$. Say for example, $\mu = 1$, then we need the knots t_{-2}, t_{-1}, t_0 in addition, and if $\mu = 5$, we need the knots t_6, t_7, t_8.

Recall that the spline space $\mathbb{S}_{p,t}$ is defined by

$$\mathbb{S}_{p,t} := \left\{ \sum_{j=1}^{n} c_j B_{j,p} \mid c_j \in \mathbb{R}, 1 \leqslant j \leqslant n \right\}.$$

We can resolve the above problem by introducing the augmented knot vector

$$\tau = (-1, -1, -1, -1, 0, 1, 2, 3, 4, 5, 5, 5, 5).$$

and instead consider $B_{1,3,t}$ as a function in the spline space $\mathbb{S}_{p,\tau}$. With respect to this new knot vector, the B-spline $B_{1,3,t}$ can be equivalently expressed as $B_{5,3,\tau}$, where our relevant index now is $\mu' = 5$. Now, the algorithm implemented in Exercise 2.13 computes arbitrary functions in a spline space, given the knot vector, polynomial degree, and set of coefficients. With the coefficients $c = (0, 0, 0, 0, 1, 0, 0, 0, 0)$, knot vector τ with index $\mu = 5$, and degree p, the algorithm computes the spline function $B_{5,3,\tau}(x)$. Hence, we can reclaim the functions in the original spline space in the augmented spline space by choosing suitable knots and considering them as spline functions in a larger spline space.