Math 409: Cubic Splines

Rusty Cain

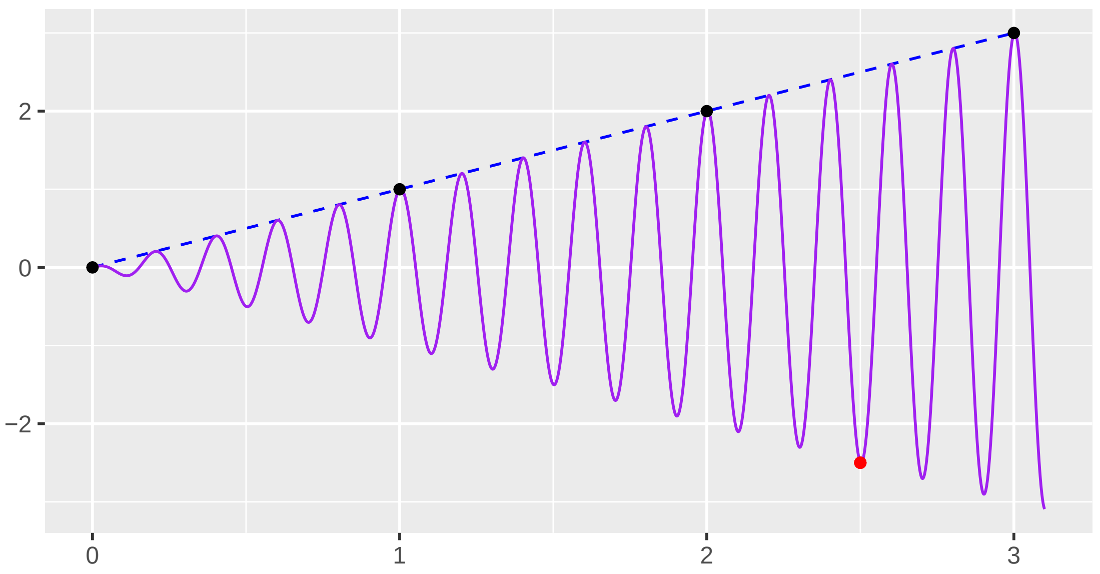
Numerical Methods

Introduction

When grappling with approximation, one must try to squeeze an educated guess out of limited information, perhaps a few (x, y) values. A simple place to start is to assume that these points are entirely representative of the function for their locality and use a linear fit between each successive pairing of points. Occasionally, this can work out quite nicely, as evidenced by Figure 1. With the blue line, this graph shows a familiar function, $y = x$. Therefore, our linear spline, say defined at integer x between 0 and 3, is fit on the corresponding, and here identical, y values. However, Figure 1 also displays in red, the sneaky function $y = x \cdot sin(10 \cdot x)$, which gives us roughly the same (x, y) cooridnates to approximate on. As evidenced by the graph, the linear fit can vary from the true form of the complex function by quite a bit, recommending that $f(2.5) = 2.5$, when it is far closer to -2.5, an error rate which grows with x. This example of two different functions which have drastically different behavior between shared n knots allows us to see one of the shortcomings of this process; dependence on our number of knots, n. Further, sinusoidal functions are far more curved that a piecewise linear function, for spare x, and we might yearn for something which returns a curved approximation between given knots. This paper intends to familiarize the reader with this improvement on piecewise linear approximations, henceforth referred to as basic splines, through brief reference to the history and theory which allowed for the creation of Python code which generates our cubic splines.

```
Figure1
```

Figure 1: Mixed Signals

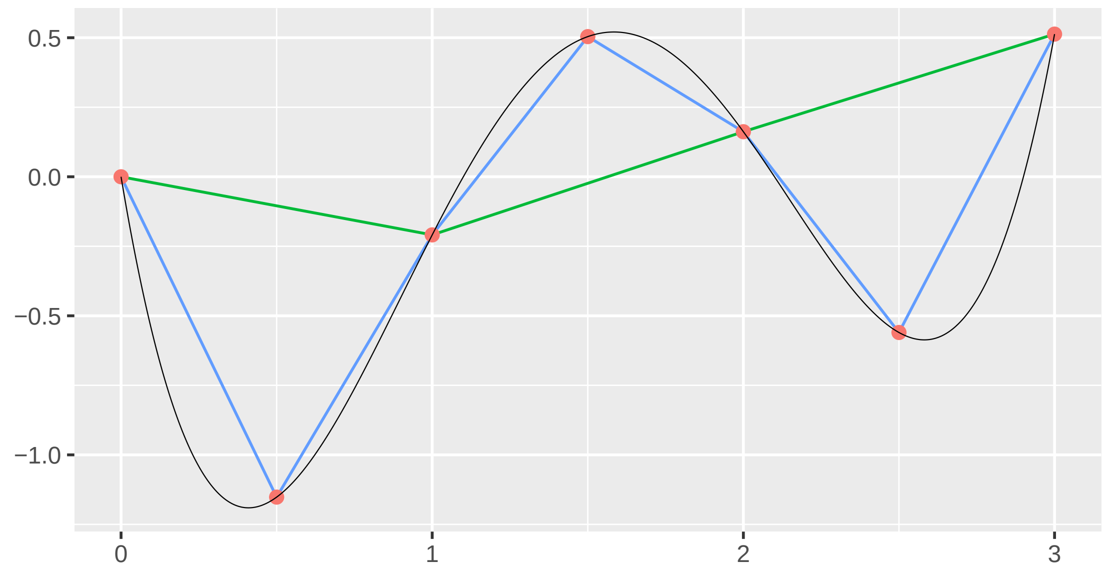


Background

Deriving its name from the plastic rulers which were used to measure and construct the bends of ships, splines were first classified by Isaac Schoenberg in 1946. As the original instrument would be strung about knots, the splines inherited the notion of “fitting” on n such knots. Of course, as n approaches infinity, our spline then perfectly takes the form of our function, f. Of course, if you could produce infinitely many values of the continuous f, you would likely have the desired shape and have no need to approximate it. Equivalently, if you string along the outside of increasingly many points of a ship, eventually you just have the hull you are hoping to construct and, thus, have no need to construct one. So, as we saw with the basic spline, all splines' accuracy are necessarily bounded by n; namely, the world's most intuitive spline will still likely fail to approximate a complex function if only given 2 points to work with. That being said, why not try to find the best means of approximating, given a fixed n.

```
Figure2
```

Figure 2: Successive Basic Splines



Theory

The simplest nonlinear approximation for n+1 observations, would be "connecting the dots", fitting a linear model between each successive pairing of points. Albeit simple, this is not a substantial improvement on the flexibility of the linear model. We will first view this linear approximation of a curve, $y = x^3 - 2x^2 - x + 3$, and later compare it to the cubic spline fit on the same curve later one.

< Linear spline S(x) is continuous function, the derivative S'(x) is however in general discontinuous at interior nodes. > From a statistical standpoint, this is akin to "overfitting" a model, which begets model weakness when exposed to new random observations. For this reason, our theory should be looking for a 'smooth'er model, hence the introduction of the cubic spline.

Code

This section will have us putting our code (included in the Appendix) to the test on some randomly generated data.

Appendix

Python Code:

```
#Spline code for final paper.
def sequence(a, b, step = 1): #modeling after R's nice seq() function
    """Returns a list of floats from a to b by step"""
    """Later used for data/knot creation"""
    answer = [a]
    assert step != 0, "How will you ever get to b?"
    while len(answer) < (abs(b-a)*(1/step)):
        answer.append(answer[-1] + step)
    answer.append(b)
    return(answer)
def f(x):
  #"""This is our 'hidden' function"""
  return((x-2.1)*(x-1.1)*(x-2.9)*(x))
def g(x, values, run):
  #"""Generates cubic spline y values from our code`s output"""
  return values[0][run]*(x**3) + values[1][run]*(x**2) + values[2][run]*x
def cubic_spline(x, y):
  #"""The chief interest of this paper, calculates approximating"""
  #""" cubic functions for each integer step of our passed data """
    n = len(x) - 1

    delta_x, delta_y = [x[1]-x[0]], [y[1]-y[0]]

    diff, z, l = [0],[0],[1]

    for i in range(1,n):
        delta_x.append(x[i+1]-x[i])
        delta_y.append(y[i+1]-y[i])
        diff.append(3*(delta_y[i]/delta_x[i] - delta_y[i-1]/delta_x[i-1]))
        l.append(2*(delta_x[i] + delta_x[i-1]) - (delta_x[i-1]**2)/l[i-1])
        z.append((diff[i] - delta_x[i-1]*z[i-1])/l[i])
    a, b, c = [0], [0], [0]
    for i in range(1,n+1):
      b.append(z[n-i] - (delta_x[n-i]/l[n-i])*b[i-1])
      a.append((b[i-1]-b[i])/(3*delta_x[n-i]))
      c.append((delta_y[n-i])/delta_x[n-i] - delta_x[n-i]*(b[i-1] + 2*b[i])/3)

    a.reverse(), b.reverse(), c.reverse()
    return(a, b, c)
x = sequence(0,3,.5)
y = [f(i) for i in x]
values = cubic_spline(x, y)
#our approximations, g_n(x), for each subdivision of our interval
print(values)
#From here, these values are passed into R for consistent (and improved) visualizations
#in ggplot2, which I find favorable to matlibplot graphs.
```

R Visualizations & Data Creation:

```r
## FIGURE ONE CODE: ###
x = seq(0,3.1,.001)
sinus <- data.frame(sinx = x, y = x*sin(10*pi*x + 1.5))
f = seq(0,3)
basic <- data.frame(x = f, y = f)
Figure1 <- ggplot(data = basic, aes(x = x, y = y)) +
  geom_line(linetype = 'dashed', color = 'blue')
Figure1 <- Figure1 + geom_line(data = sinus, aes(x = sinx, y = y),
                               color = 'purple') +
  geom_point(x = 2.5, y = -2.5, color = 'red') +
  labs(title = "Figure 1: Mixed Signals") +
  xlab("") + ylab("") + geom_point()


### FIGURE TWO CODE: ###
f <- function(x){
  return((x-2.1)*(x-1.1)*(x-2.9)*(x))
}

lin_x = 0:3
lin_y = f(lin_x)
lin_app = data.frame(x = lin_x, y = lin_y)
Figure2 <- ggplot(data = lin_app,
                  aes(lin_x, lin_y, color = "green")) + geom_line()

lin_x2 = seq(0,3,.5)
lin_y2 = f(lin_x2)
lin_app2 = data.frame(x = lin_x2, y = lin_y2)
Figure2 <- Figure2 + geom_line(data = lin_app2, aes(x = lin_x2, y = lin_y2,
                                                    color = "purple")) +
  geom_point(data = lin_app2, aes(x = lin_x2, y = lin_y2, color = "black"),
             lwd = 2)

x = seq(0,3, 0.001)
y = f(x)
g = data.frame(x = x, y = y)
Figure2 <- Figure2 + geom_line(data = g, aes(x = x, y = y), color = 'black',
                               lwd = .2) + xlab("") + ylab("") +
  labs(title = "Figure 2: Successive Basic Splines") +
  theme(legend.position = "none")
```