Chapter 4

Contents

Lagrangian Polynomial

The first thing in Chapter 4 of substance is the use of Lagrangian polynomials to fit data points. Recall that for $n+1$ points, we can write a unique n^{th} order polymial

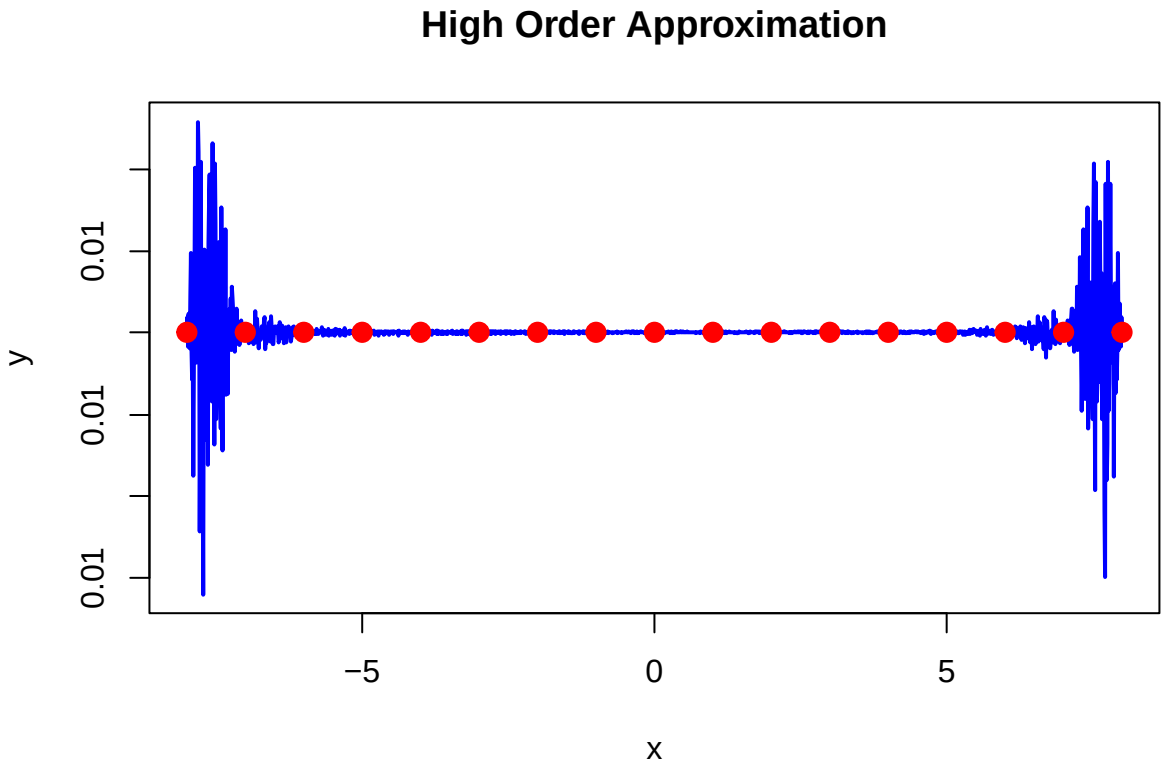
$$y = P(x) = y_0 L_0(x) + y_1 L_1(x) + ... + y_n L_n(x)$$

where

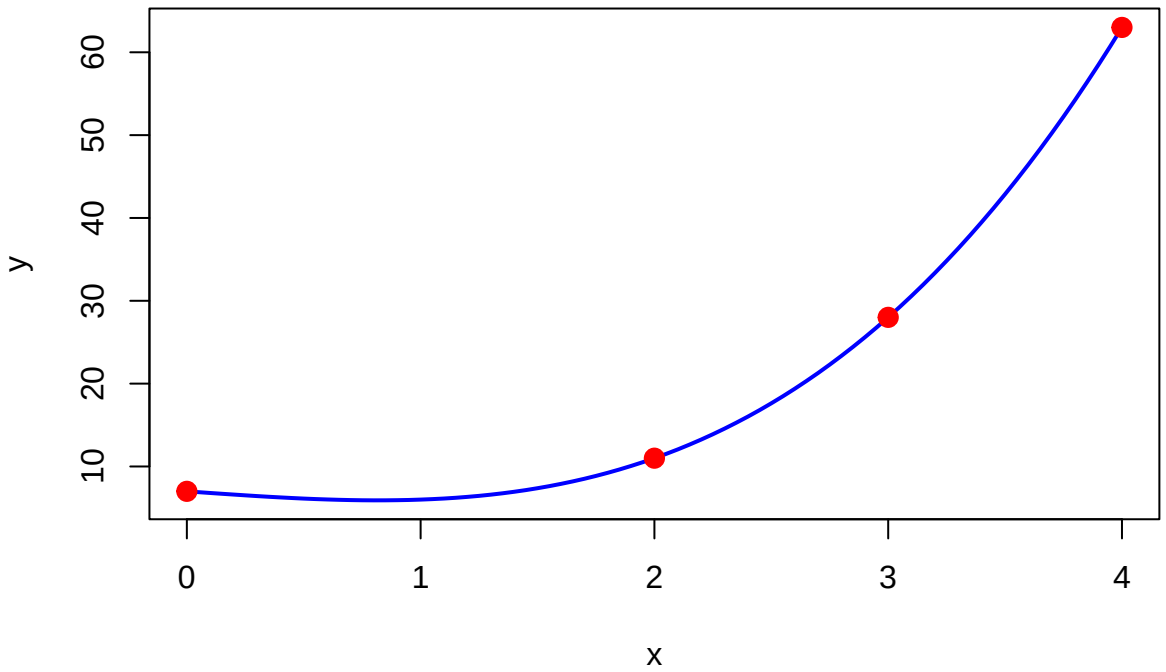
$$L_k(x) = \frac{(x-x_0)(x-x_1)...(x-x_{k-1})(x-x_{k+1})...(x-x_n)}{(x_k-x_0)(x_k-x_1)...(x_k-x_{k-1})(x_k-x_{k+1})...(x_k-x_n)}.$$

Let's use R to calculate $P(x)$ for some data set.

```
#z is the point to evaluate at
#x is the x data
#y is the y data
#i is index of the polynomial, i.e., L_k=L_i
Lk <- function(z, x, y){
  sub <- rep(NA, length(x))
  for(i in seq_along(sub)) sub[i]  <- y[i]*prod(z - x[-i]) / prod(x[i] - x[-i])
  sum(sub)
}

#v are values to evaluate the polynomial at
#x and y are the data
lagrange  <- function(v,x,y){
  if(length(x) != length(y)) return(print("Error: x and y have different lengths."))
  Px <- rep(NA, length(v))
  v <- sort(v)
  for(i in seq_along(v)) Px[i] <- Lk(v[i],x,y)
  out <- matrix(c(v, Px), ncol = 2)
  out
}

### Book Figure 4.12

plot(lagrange(seq(-8,8,by=0.01), sample(-8:8), rep(0.01,17)), type="l", xlab="x", ylab="y", main = "Hig
points(-8:8, rep(0.01,17), col="red", cex=2, pch=20)
```




```
plot(lagrange(seq(0,4,by=0.05), c(0, 2, 3, 4), c(7, 11, 28, 63)), type = "l", xlab="x", ylab="y", col="
points(c(0, 2, 3, 4), c(7, 11, 28, 63), col="red", cex=2, pch=20)
```



Divided Difference Tables

First, let's make some simple data for us to use with our divided differences. We'll take a look at a data set without noise, and then add some noise into it. We will use 20 data points. Our data will follow the polynomial

$$y = 2x^3 - 5x^2 + x + 5$$

```
# x data
set.seed(19624)
x <- sort(runif(20,-5,5)) #note the sorting
y <- 2*x^3 - 5*x^2 + x + 5

# y with (small) noise
y.noise <- y + rnorm(20,0,0.2)
```

We will be using the `diff` command to find our divided differences. There are two options for the `diff` command: `lag` and `differences`. `lag` specifies how far apart the two elements are in a vector to be differenced. The `differences` argument specifies if you want first differences, second differences, etc. Let's use the the command to get our table for the clean table:

```
first <- diff(y)/diff(x)
second <- diff(first)/diff(x,lag=2)
third <- diff(second)/diff(x,lag=3)
fourth <- diff(third)/diff(x,lag=4)

data.frame(first, second = c(0,second), third = c(0,0,third), fourth = c(0,0,0,fourth))
```

```
##          first     second third        fourth
## 1  171.7112529   0.000000     0  0.000000e+00
## 2  143.9137883 -31.353988     0  0.000000e+00
## 3  130.0638771 -28.871372     2  0.000000e+00
## 4  116.4222924 -27.634129     2  5.426361e-13
## 5  110.7858915 -26.499250     2 -5.599448e-12
## 6   96.5568700 -25.087057     2  6.095765e-12
## 7   68.1465976 -22.403209     2 -2.368991e-12
## 8   49.8303489 -19.541851     2  2.681711e-13
## 9   40.6848464 -16.910755     2 -3.166904e-14
## 10  28.9564832 -15.024425     2  3.541540e-14
## 11   4.4722739  -8.521319     2 -8.454881e-15
## 12  -2.3279415  -2.724249     2  6.776265e-16
## 13  -2.0662427   2.432763     2 -7.746968e-15
## 14  -0.7116301   3.303390     2  2.200393e-14
## 15   8.0595543   6.267144     2 -1.267039e-14
## 16  19.2255930   9.319023     2 -2.581808e-14
## 17  27.2157832  12.719055     2  2.792316e-14
## 18  37.3883355  14.386056     2 -3.146576e-14
## 19  65.2545751  18.415999     2 -1.555399e-15
```

It's pretty clear that third divided difference is a constant, thus suggesting that a 3^{rd} order polynomial would fit our data. So our divided difference table gave us the correct answer. Let's do the same thing with the noisy data:

```
first <- diff(y.noise)/diff(x)
second <- diff(first)/diff(x,lag=2)
third <- diff(second)/diff(x,lag=3)
fourth <- diff(third)/diff(x,lag=4)

data.frame(first, second = c(0,second), third = c(0,0,third), fourth = c(0,0,0,fourth))
```

```
##          first      second       third       fourth
```

```
## 1  171.8835632   0.0000000   0.0000000    0.0000000
## 2  144.3621449 -31.0426234   0.0000000    0.0000000
## 3  129.8785116 -30.1924222   0.6849237    0.0000000
## 4  113.2445509 -33.6958665  -5.6633082   -4.5994395
## 5  117.4052256  19.5611989  93.8550315  143.7272096
## 6   94.9505054 -39.5897105 -83.7717192 -167.4404050
## 7   68.8262753 -20.6005276  14.1507164   66.1265256
## 8   49.3609843 -20.7677796  -0.1169039   -9.4834926
## 9   40.1826969 -16.9713772   2.8857958    1.6599187
## 10  30.4933666 -12.4123562   4.8337474    1.1339125
## 11   4.1923078  -9.1536426   1.0022021   -1.1222714
## 12   1.8032622  -0.9570808   2.8278295    0.5571374
## 13   0.1119795 -15.7222404  -5.7262452   -2.8696653
## 14  -0.7741360  -2.1609021  31.1530539   12.6896644
## 15   8.1977487   6.4105477   5.7841837  -16.8326470
## 16  19.7617858   9.6511867   2.1237013   -2.2760430
## 17  26.2375502  10.3083405   0.3865575   -0.8566826
## 18  37.8663314  16.4454596   7.3630643    3.6616117
## 19  65.3490562  18.1625450   0.8521636   -3.0405445
```

At this point, we see that with our noisy data, we still have some fairly large differences. Let's take a few more divided differences and see if things get better:

```
fifth <- diff(fourth)/diff(x,lag=5)
sixth<- diff(fifth)/diff(x,lag=6)
seventh<- diff(sixth)/diff(x,lag=7)
fourth
```

```
##  [1]   -4.5994395  143.7272096 -167.4404050   66.1265256   -9.4834926
##  [6]    1.6599187    1.1339125   -1.1222714    0.5571374   -2.8696653
## [11]   12.6896644  -16.8326470   -2.2760430   -0.8566826    3.6616117
## [16]   -3.0405445
```

```
fifth
```

```
##  [1]  102.0122405 -262.4099381  127.2446505  -46.0088577    5.9187403
##  [6]   -0.2378720   -0.5386183    0.4882933   -1.0201471    4.7026899
## [11]   -7.4212897    8.9112330    0.6726561    2.0233725   -2.0858393
```

```
sixth
```

```
##  [1] -187.13226045  198.74806664  -86.70830969   25.68582386   -2.69425530
##  [6]   -0.06423137    0.24368567   -0.42833083    1.55222238   -2.76777162
## [11]    3.97922760   -3.85822309    0.58336763   -1.16049580
```

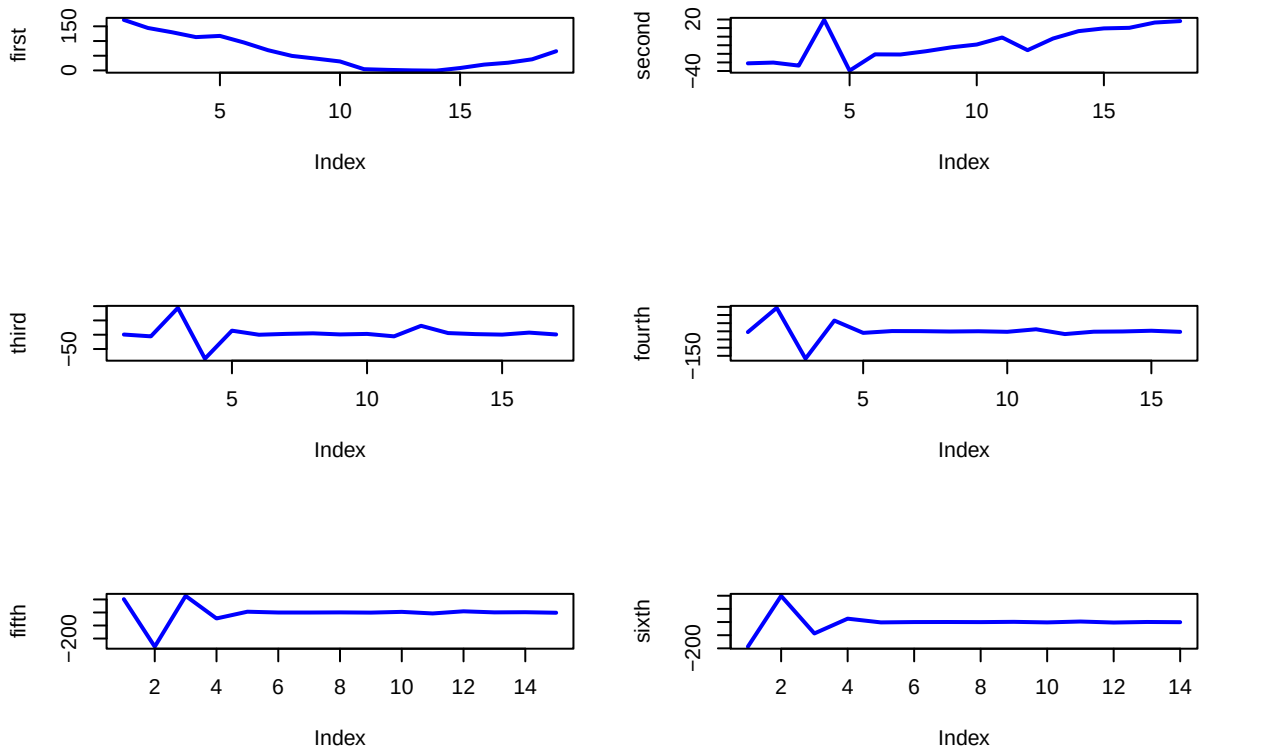
```
seventh
```

```
##  [1]  141.75611466 -134.45323172   47.29629448  -11.70795660    0.55298813
##  [6]    0.06541019   -0.15641352    0.51450800   -0.90781418    1.49707073
## [11]   -1.70147701    1.89760989   -0.48129981
```

Even with a small amount of noise, our differences are not going to zero, i.e., the errors are propagating through the table. It becomes difficult to tell what order is appropriate. One thing we can do is look at plots of the divided differences.

```
par(mfrow = c(3,2))
plot(first, type="l",lwd=2,col="blue")
plot(second, type="l",lwd=2,col="blue")
plot(third, type="l",lwd=2,col="blue")
```

```
plot(fourth, type="l",lwd=2,col="blue")
plot(fifth, type="l",lwd=2,col="blue")
plot(sixth, type="l",lwd=2,col="blue")
```



From the plots above, we can see definite trends in the first and second divided differences (first looks quadratic; second looks linear). The third divided difference *might* be constant. Going to the fourth, fifth, and sixth difference does not really change the shape of the curve. This suggests, that perhaps a third difference would be sufficient. If we add more noise, do you think this will work?

Splines

As discussed in class, smoothing splines are piecewise polynomial functions of some order. Probably the most common order used is a 3^{rd} order (cubic) polynomial. The polynomials are joined at points called knots (i.e., two consecutive knots define the domain of one of the cubic polynomials). More formally, for some sequence of $(n+1)$ knots $k_0, k_1, ..., k_n$ where $k_0 = \min(x)$ and $k_n = \max(x)$, then curves $S_1, S_2, ...S_n$ given by

$$S_1 = a_1+b_1x+c_1x^2+d_1x^3 \qquad x \in [k_0, k_1) S_2 = a_2+b_2x+c_2x^2+d_2x^3 \qquad x \in [k_1, k_2) \vdots S_n = a_n+b_nx+c_nx^2+d_nx^3 \qquad x \in [k_{n-1},$$

define the spline $S(x)$. Because we wish the spline to be smooth, we require that $S_i'(k_i) = S_{i+1}'(k_i)$. Optionally (among many options), it can also be required that $S_i''(k_i) = S_{i+1}''(k_i)$. At the endpoints k_0, k_n, we can impose special boundary conditions. A "natural" spline is one where we require that $S_0''(k_0) = S_n''(k_n) = 0$, i.e., the first derivative stays constant at the exterior boundary. If we happen to have information about the first derivative at the endpoints, given by $f'(k_0)$ and $f'(k_n)$, then we can create a "clamped" spline by setting $S_1'(k_0) = f'(k_0)$ and $S_n'(k_n) = f'(k_n)$.

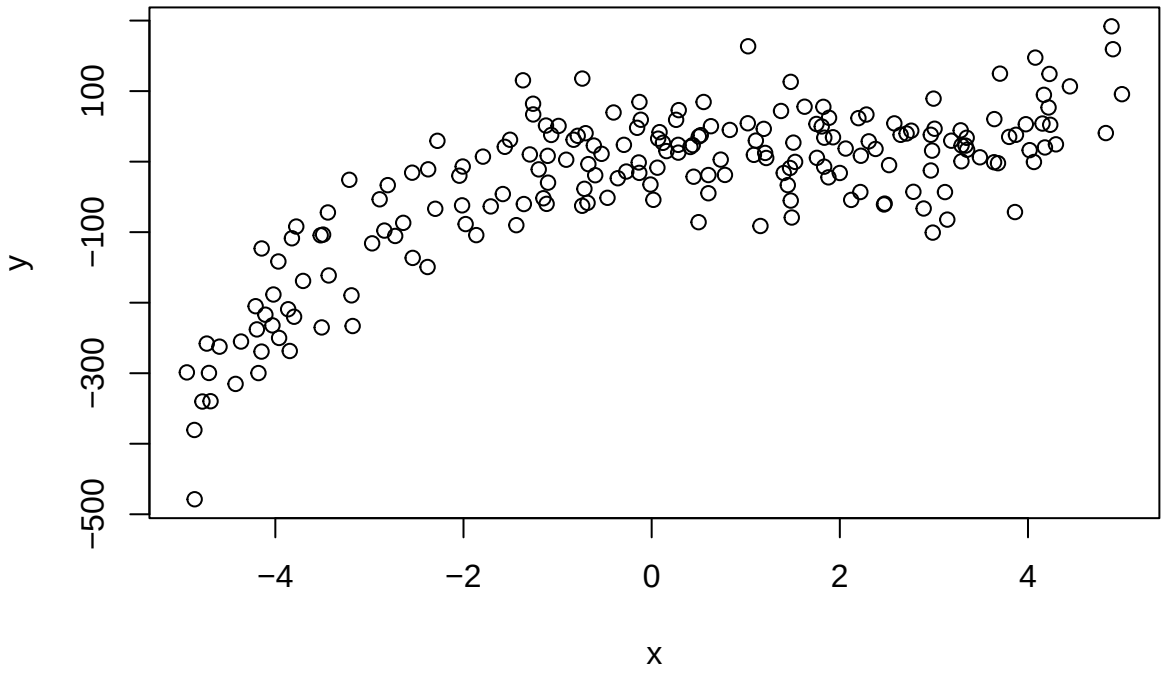
The fit of a spline is often assessed using a *penalized* log-likelihood. The penalized log-likelihood is defined by

$$\mathcal{L} = (y - f)'W(y - f) + \lambda c'\Sigma c$$

where W is a weights matrix (if you want certain observations to be more important), y are the data, f are the modeled values, λ is a smoothing parameter, c are the coefficients of penalized regression, and Σ is a variance-covariance matrix of the spline functions. The important thing to note is the first term $(y-f)'W(y-f)$ is just a least-squares regression that can be weighted by W. The second thing is that λ controls **how much smoothing** you wish to do. If $\lambda = 0$, we get a least-squares fit for each segment S_i of our spline. If insteads, we use the **spar** flag (**s**moothing **par**ameter) and set it to 1, we get a highly smoothed spline.

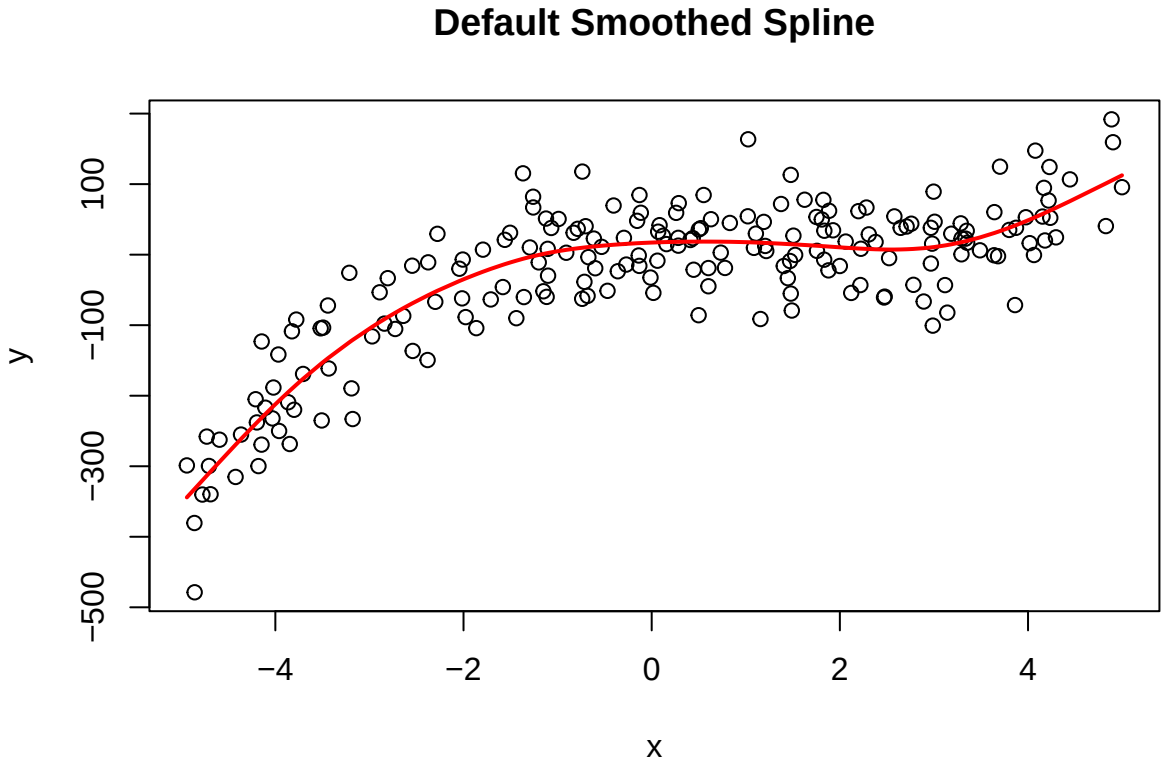
Let's try out a smoothing spline in R using a larger data set based on the same polynomial we used earlier:

```
set.seed(8652)
x <- sort(runif(200,-5,5)) #note the sorting
y <- 2*x^3 - 5*x^2 + x + 5

# y with (small) noise
y.noise <- y + rnorm(200,0,50)

plot(x, y.noise, ylab="y")
```



Okay, we have a fairly noisy data set. Let's try the function `smooth.spline` to find a nice fit:

```
s1 <- smooth.spline(x,y.noise)
m1 <- predict(s1, x)

plot(x, y.noise, ylab="y", main = "Default Smoothed Spline")
points(m1, col="red", type="l", lwd = 2)
```



Okay, that's not bad!!! Right out of the box, we didn't need to do much to get a pretty smoothed spline. Let's play with some of the options we have, namely, the amount of smoothing we wish to do and where the knots are. First, let's get some info from the spline object:

```
s1$spar #how much smothing was done?
```

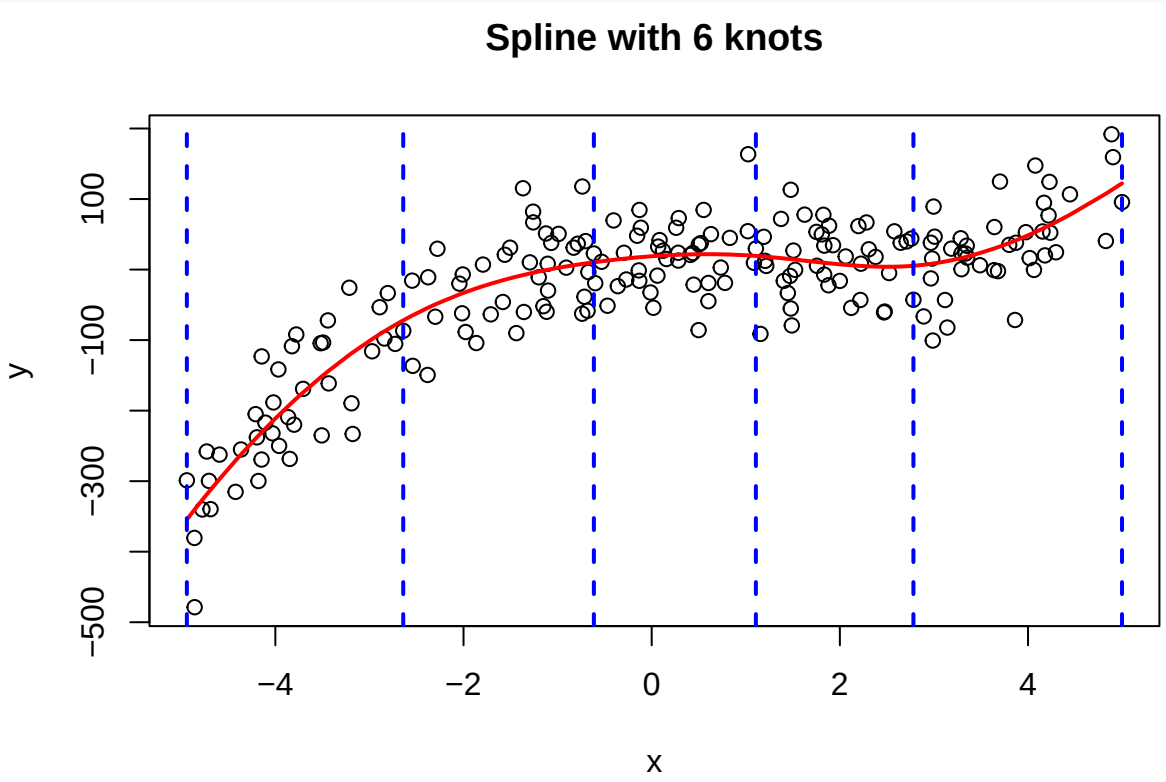
```
## [1] 0.9413255
```

```
s1$fit$knot #see how many knots and where, note these are scaled between xmin and xmax
```

```
##   [1] 0.000000000 0.000000000 0.000000000 0.000000000 0.008338182 0.021309728
##   [7] 0.025326028 0.052112255 0.073558493 0.076581319 0.080036762 0.091548414
##  [13] 0.097870998 0.108347097 0.112362688 0.117025371 0.143097518 0.145715711
##  [19] 0.151689183 0.176080485 0.198135915 0.211231783 0.222778582 0.240951927
##  [25] 0.257425626 0.265654548 0.291431162 0.295015480 0.309509262 0.324979383
##  [31] 0.340022125 0.352324811 0.360389577 0.370240477 0.376288213 0.384043168
##  [37] 0.386236211 0.397501381 0.413601491 0.422735013 0.425140848 0.428508289
##  [43] 0.435223782 0.443495739 0.456324453 0.467376862 0.481528796 0.483827212
##  [49] 0.485455027 0.498860160 0.504021526 0.509115644 0.523225768 0.525541972
##  [55] 0.538450957 0.541929842 0.547503176 0.552616011 0.557718902 0.570919008
##  [61] 0.580655585 0.600201118 0.608472371 0.616961458 0.619573357 0.638294705
##  [67] 0.645019126 0.645799222 0.648616556 0.673259921 0.678956291 0.681513945
##  [73] 0.686120893 0.690967583 0.704512441 0.718158229 0.720773002 0.729235876
##  [79] 0.745533818 0.750963510 0.763317829 0.774651321 0.787932179 0.795612808
##  [85] 0.797661005 0.799598302 0.813192184 0.827549807 0.828456287 0.833934960
##  [91] 0.848097300 0.863498178 0.869516543 0.885578008 0.897190561 0.905848469
##  [97] 0.914986884 0.917603368 0.922446991 0.929394140 0.982723689 1.000000000
## [103] 1.000000000 1.000000000 1.000000000
```

Now that we know a bit about the "out-of-the-box" spline, we can mess around with it some. Let's try a spline with just a few knots, say 6, and not a lot of smoothing.

```
s2 <- smooth.spline(x,y.noise, nknots = 6, spar = 0.1)
m2 <- predict(s2, x)

plot(x, y.noise, ylab="y", main="Spline with 6 knots")
points(m2, col="red", type="l", lwd = 2)
abline(v = unique(s2$fit$knot) * diff(range(x)) + min(x), col="blue", lty = 2, lwd = 2)
```



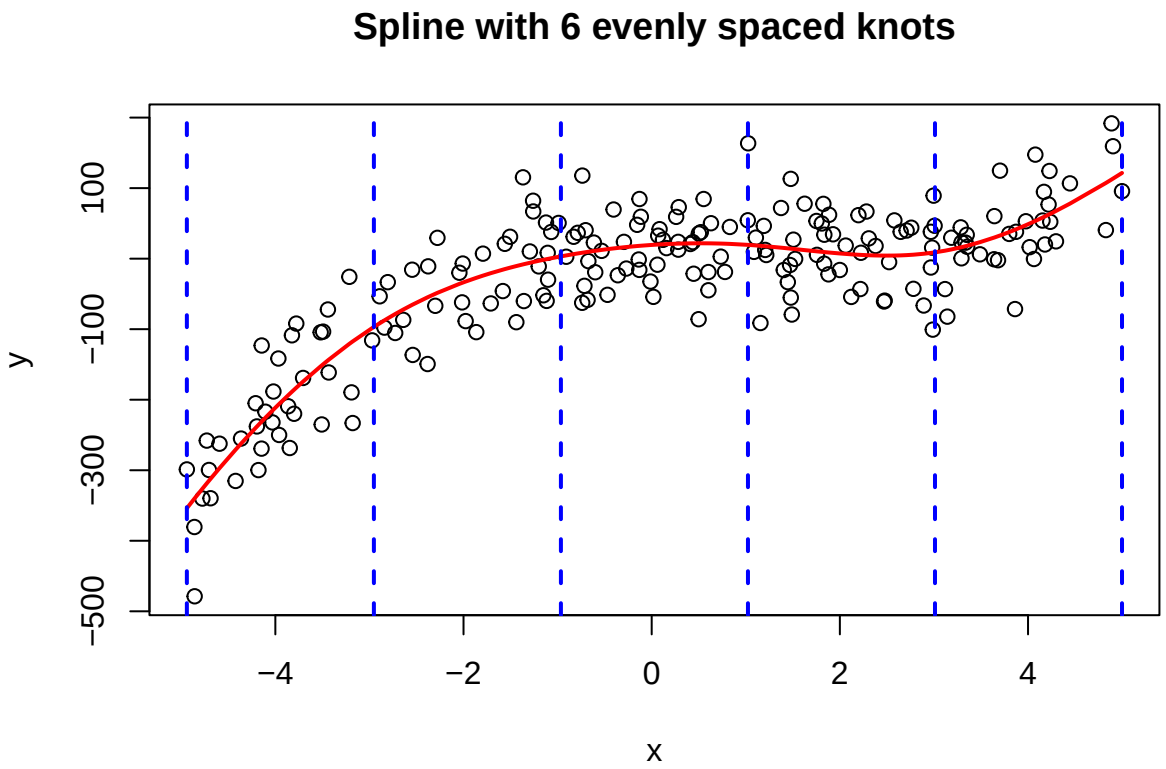
Note that the knots are not even spaced! How do you think they are spaced?

```
table(cut(x, unique(s2$fit$knot) * diff(range(x)) + min(x), include.lowest=T))
```

```
##
##  [-4.94,-2.64] (-2.64,-0.615]  (-0.615,1.11]    (1.11,2.78]       (2.78,5]
##             40              40             40             40             40
```

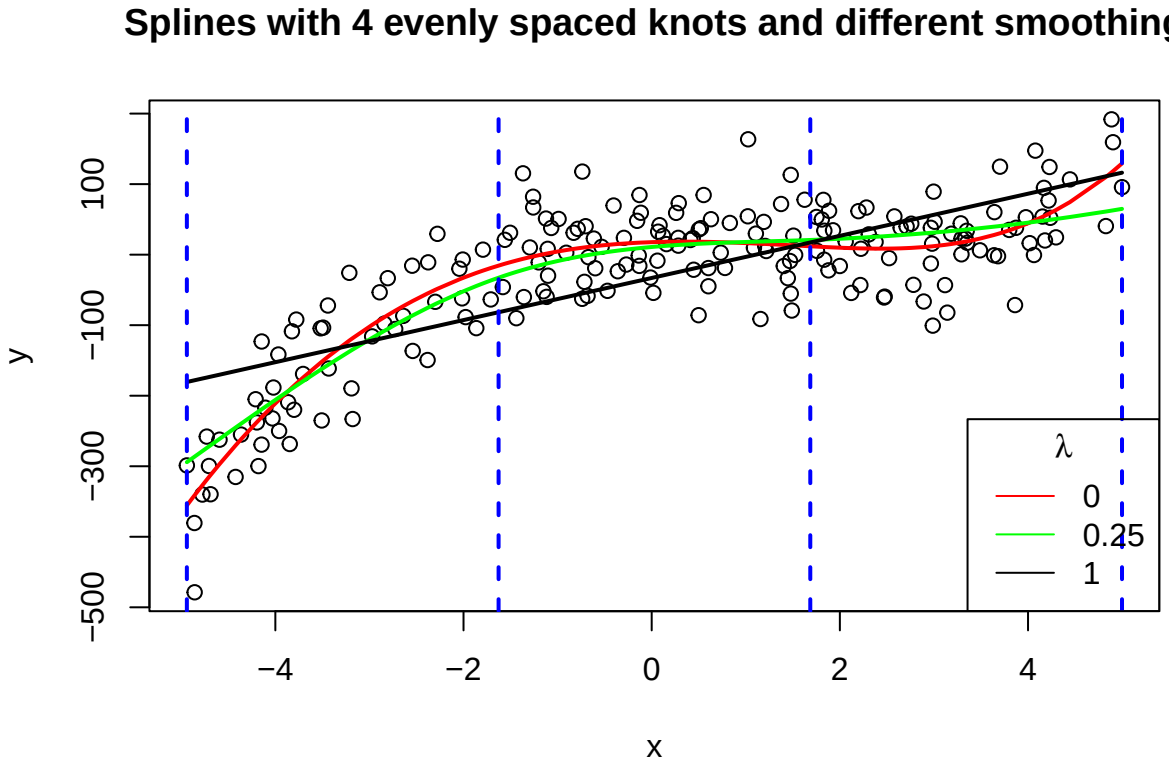
Let's set the knots ourselves:

```
s3 <- smooth.spline(x,y.noise, all.knots = seq(0,1,length=6), spar = 0.1)
m3 <- predict(s3, x)

plot(x, y.noise, ylab="y", main="Spline with 6 evenly spaced knots")
points(m3, col="red", type="l", lwd = 2)
abline(v = unique(s3$fit$knot) * diff(range(x)) + min(x), col="blue", lty = 2, lwd = 2)
```



Finally, let's compare 3 splines where we either have no smoothing, some smoothing, or full smoothing and we use a small number of knots, 4.

```
s4 <- smooth.spline(x,y.noise, all.knots = seq(0,1,length=4), spar = 0)
m4 <- predict(s4, x)

s5 <- smooth.spline(x,y.noise, all.knots = seq(0,1,length=4), spar = 0.25)
m5 <- predict(s5, x)

s6 <- smooth.spline(x,y.noise, all.knots = seq(0,1,length=4), spar = 1)
m6 <- predict(s6, x)

plot(x, y.noise, ylab="y", main="Splines with 4 evenly spaced knots and different smoothing")
points(m4, col="red", type="l", lwd = 2)
points(m5, col="green", type="l", lwd = 2)
points(m6, type="l",lwd=2)
abline(v = unique(s4$fit$knot) * diff(range(x)) + min(x), col="blue", lty = 2, lwd = 2)
legend("bottomright", legend = c("0", "0.25", "1"), lty = 1, col = c("red","green","black"), title = ex
```



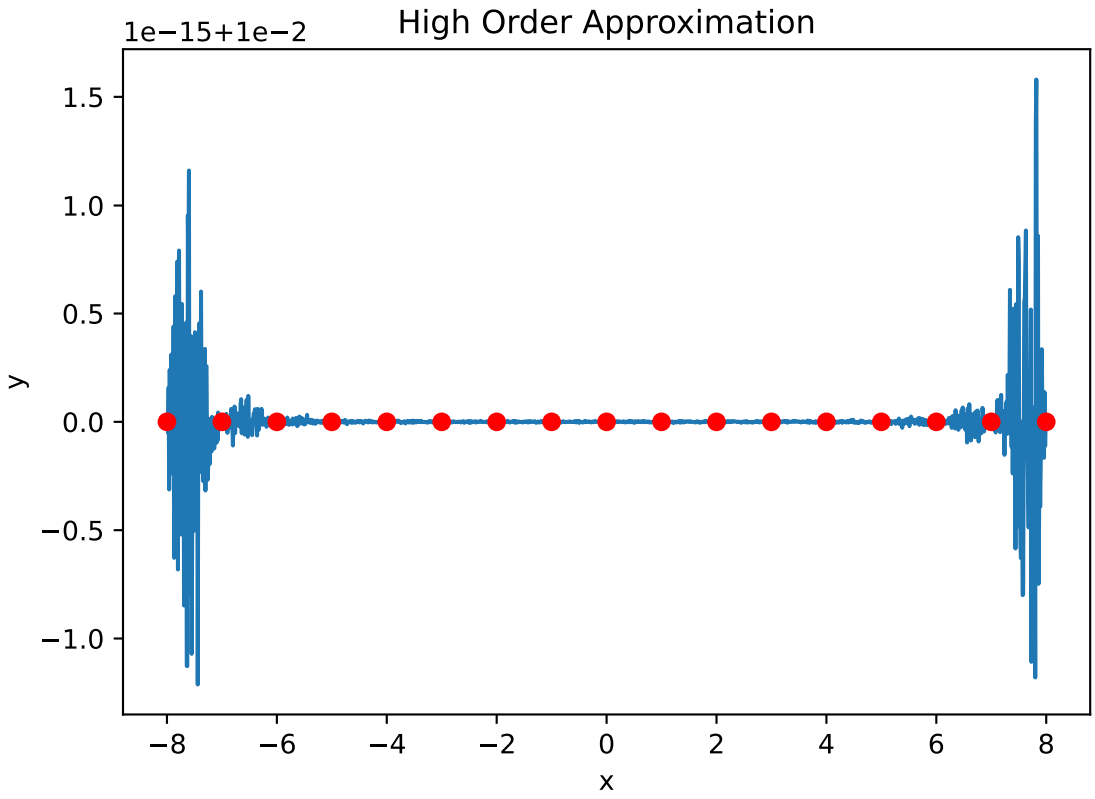
Why do we get a straight line when $\lambda = 1$? Why don't we see three distinct segments to the line?

Doing it All in Python

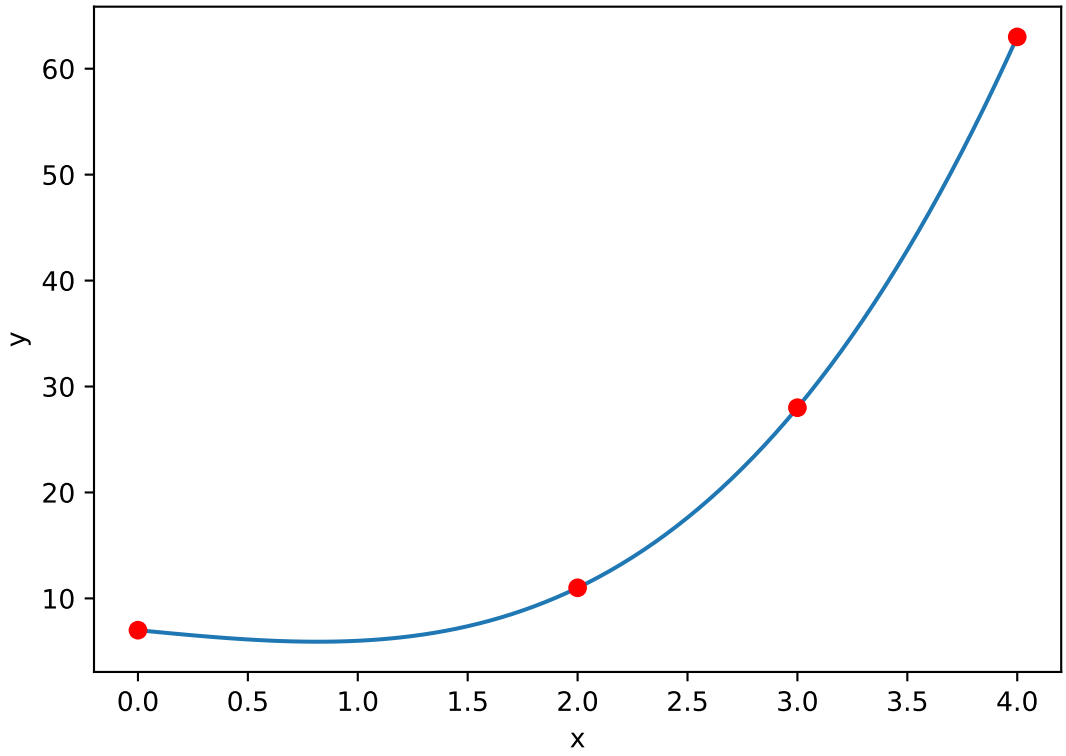
Defining some functions to calculate Lagrangian polynomials

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy.interpolate as interp

def Lk(z,x,y):
  sub = np.zeros(x.size)
  for i in range(sub.size): sub[i] = y[i]*np.prod(z - np.delete(x,i))/np.prod(x[i] - np.delete(x,i))
  return np.sum(sub)

def lagrange(v,x,y):
  if x.size != y.size: return print("Error: x and y have different lengths.")
  Px = np.zeros(v.size)
  v.sort()
  for i in range(v.size): Px[i] = Lk(v[i],x,y)
  return np.stack([v,Px],axis=1)

lg_data = lagrange(np.arange(-8,8.01,0.01), np.arange(-8,9), np.repeat(0.01,17))


plt.plot(lg_data[:,0], lg_data[:,1])
plt.scatter(np.arange(-8,9), np.repeat(0.01,17), color = 'red', zorder = 10)
plt.xlabel('x')
plt.ylabel('y')
plt.title('High Order Approximation')
```

```
plt.show()
```




```
plt.close()

lg_data = lagrange(np.arange(0,4.05,0.05), np.array([0,2,3,4]), np.array([7,11,28,63]))

plt.plot(lg_data[:,0], lg_data[:,1])
plt.scatter(np.array([0,2,3,4]), np.array([7,11,28,63]), color = 'red', zorder = 10)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```




```
plt.close()
```

Doing divided differences tables:

```
rng = np.random
rng.seed(19624)

x = rng.uniform(-5,5,20)
x.sort()
y = 2*np.power(x,3) - 5*np.power(x,2) + x + 5
#y with (small) noise
y_noise = y + rng.normal(0,0.2,20)

#numpy has a diff() function but it can't do lags
#pandas.Series also has a diff() function and can do lags (keyword periods)
first = pd.Series(y).diff()/pd.Series(x).diff()
second = first.diff()/pd.Series(x).diff(periods=2)
third = second.diff()/pd.Series(x).diff(periods=3)
fourth = third.diff()/pd.Series(x).diff(periods=4)

pd.DataFrame(data={'first': first, 'second': second,'third': third, 'fourth':fourth})
```

```
##          first     second  third        fourth
## 0          NaN        NaN    NaN           NaN
## 1   108.157374        NaN    NaN           NaN
## 2    88.358488 -24.747024    NaN           NaN
## 3    69.295892 -21.975981    2.0           NaN
```

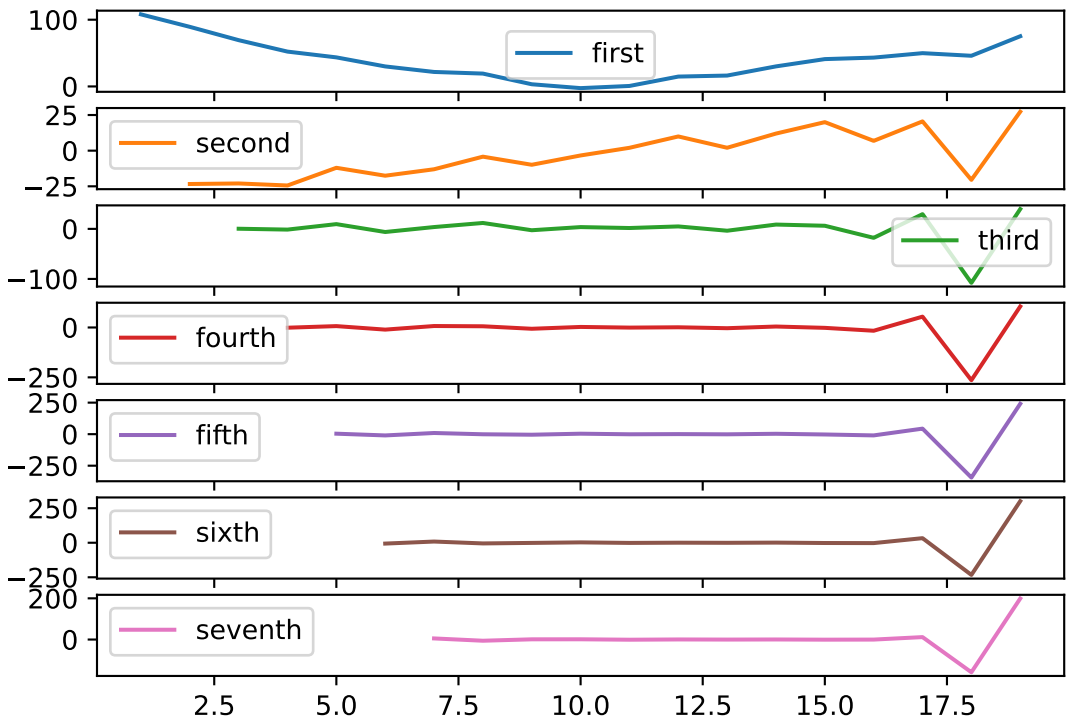
```
## 4    55.252686 -20.008328    2.0  8.530401e-14
## 5    42.716799 -17.396164    2.0 -9.899467e-14
## 6    30.664584 -15.620212    2.0  7.896547e-14
## 7    22.318996 -13.141689    2.0 -4.258563e-14
## 8    15.557251 -11.645720    2.0  5.353247e-14
## 9     3.467879  -7.476304    2.0 -3.302956e-14
## 10   -3.084912  -3.816517    2.0  7.441509e-15
## 11    0.831529   2.423590    2.0 -5.494468e-16
## 12    8.842610   5.673408    2.0  3.533984e-14
## 13   16.312509   9.977024    2.0 -1.509922e-13
## 14   30.400746  12.287929    2.0  2.169633e-13
## 15   38.506438  14.858515    2.0 -2.165142e-13
## 16   43.845905  16.326102    2.0 -1.247678e-13
## 17   49.461812  17.254613    2.0 -2.261240e-13
## 18   52.924183  18.015526    2.0  5.515203e-12
## 19   74.810838  20.432294    2.0 -2.259473e-12
```

Repeating the process with the noisy y data:

```
first = pd.Series(y_noise).diff()/pd.Series(x).diff()
second = first.diff()/pd.Series(x).diff(periods=2)
third = second.diff()/pd.Series(x).diff(periods=3)
fourth = third.diff()/pd.Series(x).diff(periods=4)
fifth = fourth.diff()/pd.Series(x).diff(periods=5)
sixth = fifth.diff()/pd.Series(x).diff(periods=6)
seventh = sixth.diff()/pd.Series(x).diff(periods=7)

div_diff = pd.DataFrame(data={'first': first, 'second': second,'third': third, 'fourth':fourth,
                    'fifth': fifth, 'sixth': sixth, 'seventh': seventh})
div_diff.plot(subplots=True)
```

```
## array([<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>,
##        <AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>], dtype=object)
```

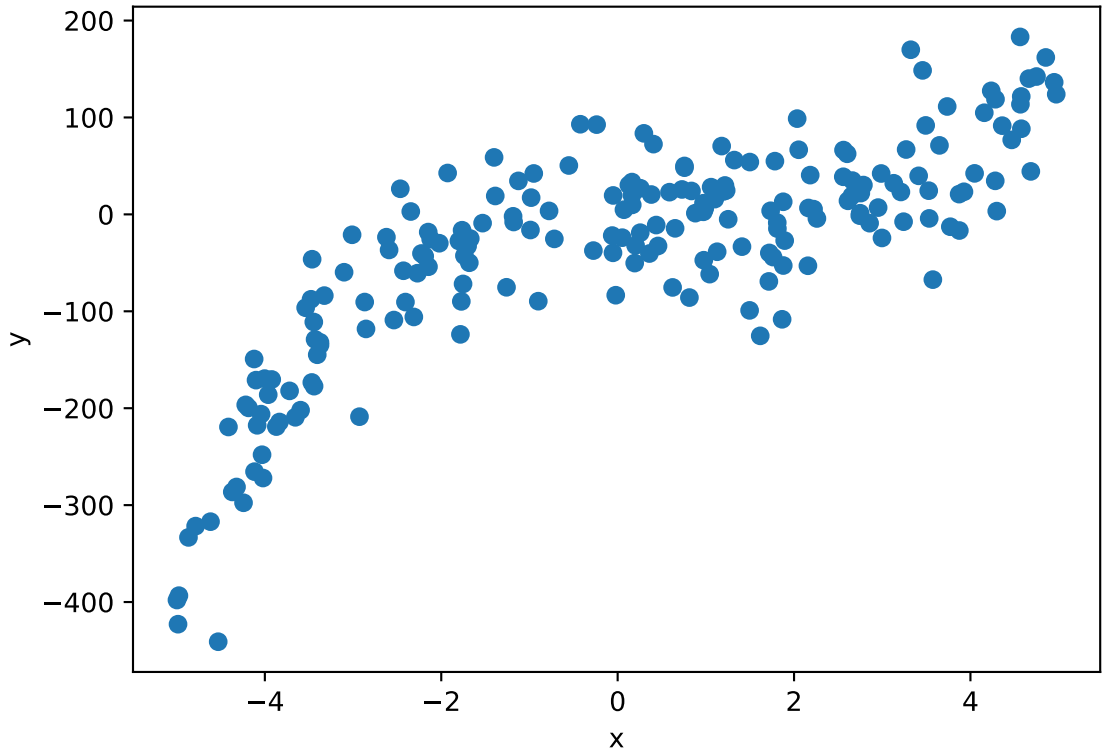
```
plt.show()
```




```
plt.close()
```

Working with splines: Smoothing splines are much more difficult to work with in Python. I'd recommend following the code use in R. `scipy.interpolate` offers some functionality, but smoothing parameters do not work as expected.

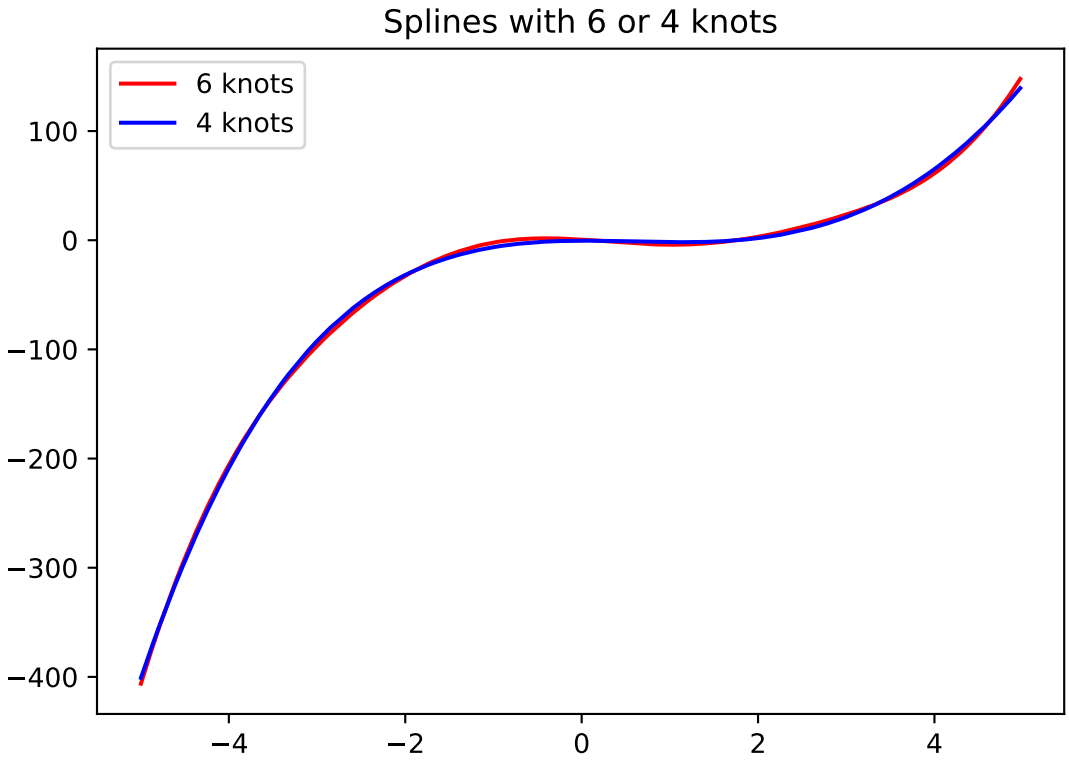
```
rng = np.random
rng.seed(8652)

x = rng.uniform(-5,5,200)
x.sort()
y = 2*np.power(x,3) - 5 * np.power(x,2) + x + 5
y_noise = y + rng.normal(0,50,200)

plt.scatter(x,y_noise)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```




```
knots = np.linspace(-3,3,4) #note that Python automatically includes knots at the min/max x values, so
weight = np.repeat(1, x.size)
s1 = interp.splrep(x, y_noise, w=weight, t = knots)
m1 = interp.splev(x,s1)

#here are the knots:
s1[0]
```

```
## array([-4.9965523 , -4.9965523 , -4.9965523 , -4.9965523 , -3.        ,
##        -1.        ,  1.        ,  3.        ,  4.97271453,  4.97271453,
##         4.97271453,  4.97271453])
```

```
knots2 = np.array([-5 + 10/3, -5 + 20/3])
s2 = interp.splrep(x, y_noise, w=weight, t = knots2)
m2 = interp.splev(x,s2)

plt.plot(x, m1, color='r')
plt.plot(x, m2, color='b')
plt.title('Splines with 6 or 4 knots')
plt.legend(["6 knots","4 knots"])
plt.show()
```


Splines with 6 or 4 knots
6 knots
4 knots
100
0
−100
−200
−300
−400
−4
−2
0
2
4

```
plt.close()
```