# hmw1

Group G33: Hejaz Navaser, Mario Edoardo Pandolfo, Salim Sikder

2022-04-10

# Our job

```
# Load the data
wmap <- read.csv("wmap.csv")

# min max normalization
min_max_norm <- function(x) {
    (x - min(x)) / (max(x) - min(x))
}

# We normalize the data
wmap <- cbind(wmap, lapply(wmap[1], min_max_norm))
# We set the column with the normalize data as the main x colomn
colnames(wmap) = c("X", "y", "x")
```

## Part 1

### 1.1

The purpose of using spline and Fourier Series is somehow similar. We fit the splines to different interval of the data to get a better approximation of the non-linear data by dividing it to different sections. In addition, we can use different degree of splines to get a continues model.The Fourier series is a way to represent a better approximation of a periodic function as a sum of sine and cosine functions, and this functions need to be orthogonal. Therefore, the both methods use a sum of more simple function to represent a more complicated function, but in the spline we use different number of degrees (exponential), while in Fourier series we sum odd product of the sine and cosine functions to represent the periodic function.

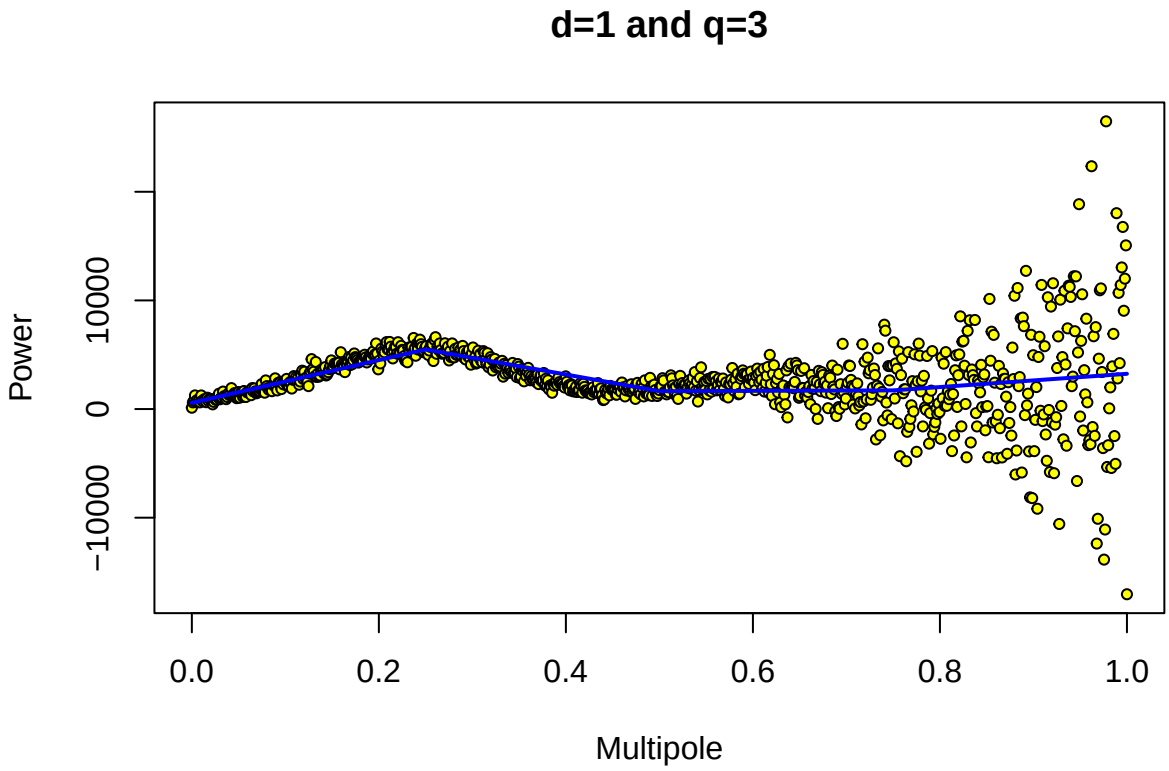
### 1.2

Here are the functions we're going to use:

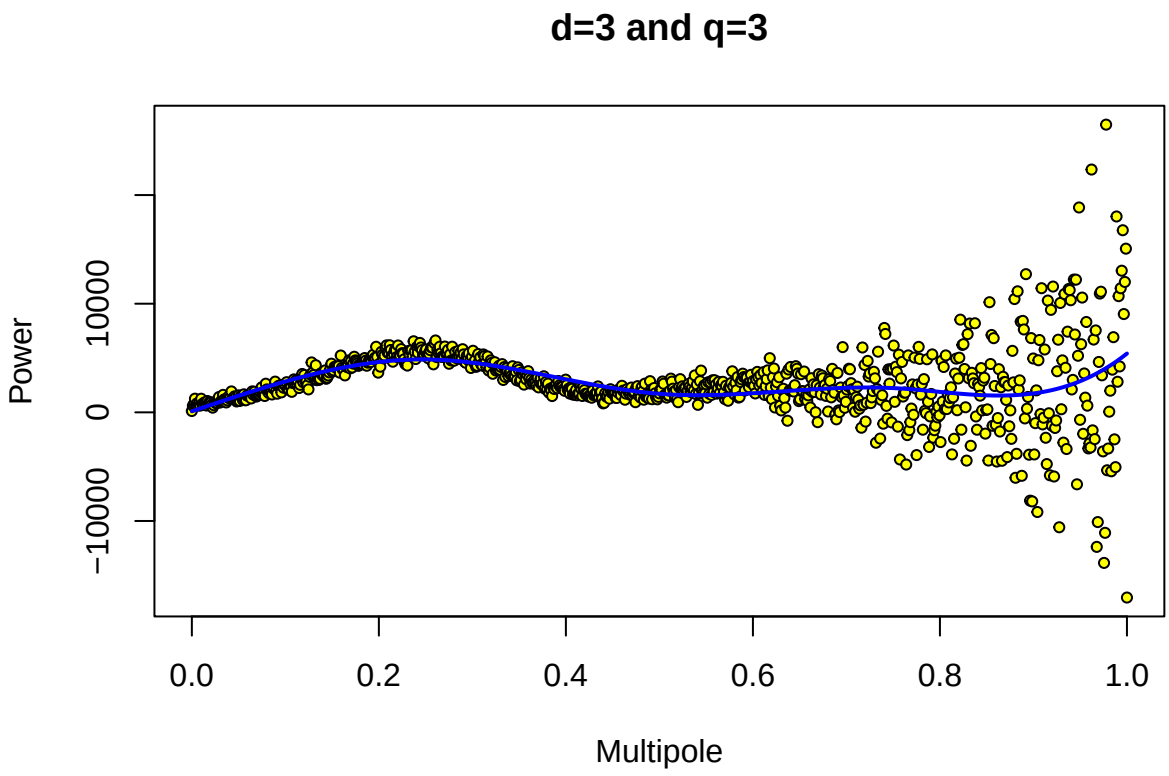
```
knot <- function(x, d, q){
  # The function takes the the features(data$x),
  # a degree and a number of knots as arguments
  # and returns the term (x-q)^d*(x>=q) as an output, which is equal to
  # (x-q)^d if x>=q, otherwise is equal to 0
  return ((x-q)^d*(x>=q))
}


vknots <- function(x, nodes){
  # By passing the data(features) and the number of nodes, the function
  # returns as an output a vector with all the n knots values

  vect <- vector(length = nodes)
  for(i in 1:length(vect)){
    vect[i] <- i*(1/(nodes+1))
  }
  return (vect)
}
```

```
DMatrix <- function(x, degree, nodes){
  mat = matrix(x, byrow = TRUE)
  if(degree > 1){
    for(i in 2:degree){
      mat <- cbind(mat, x^i)
    }
  }
  # Here we build the knots vector
  vect <-vknots(x, nodes)
  for(i in 1:length(vect)){
    mat <- cbind(mat, knot(x, d = degree, q = vect[i]))
  }

  return (mat)
}


# The structure of the matrix:

#  1  x_1  (x_1)^2 ... (x_1)^d  (x_1-q_1)^d*(x_1>=q_1) ... (x_1-q_n)^d*(x_1>=q_n)
#  1  x_2  (x_2)^2 ... (x_2)^d  (x_2-q_1)^d*(x_2>=q_1) ... (x_2-q_n)^d*(x_2>=q_n)
#  1  x_3  (x_3)^2 ... (x_3)^d  (x_3-q_1)^d*(x_3>=q_1) ... (x_3-q_n)^d*(x_3>=q_n)
# ... ...   ...    ...   ...          ...          ...          ...
#  1  x_n  (x_n)^2 ... (x_n)^d  (x_n-q_1)^d*(x_n>=q_1) ... (x_n-q_n)^d*(x_n>=q_n)
```

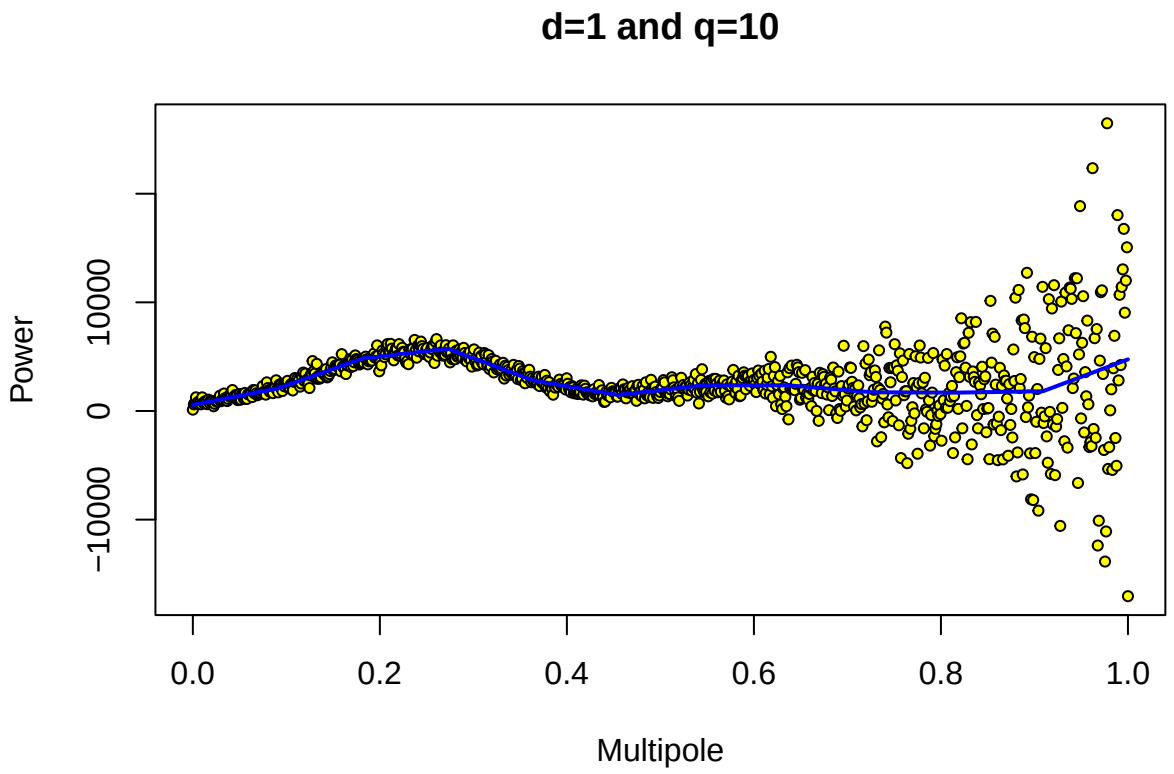
```
reg<-lm(y~x+I((x-0.25)*(x>=0.25))+I((x-0.50)*(x>=0.50))+I((x-0.75)*(x>=0.75)),
        data = wmap)
plot(y ~x, wmap, pch = 21, bg = "yellow", cex = .7, main = "d=1 and q=3",
     xlab = "Multipole", ylab = "Power")
lines(wmap$x, predict(reg), col= 'blue', lwd=2)
```


d=1 and q=3
Power
−10000
0
10000
0.0
0.2
0.4
0.6
0.8
1.0
Multipole

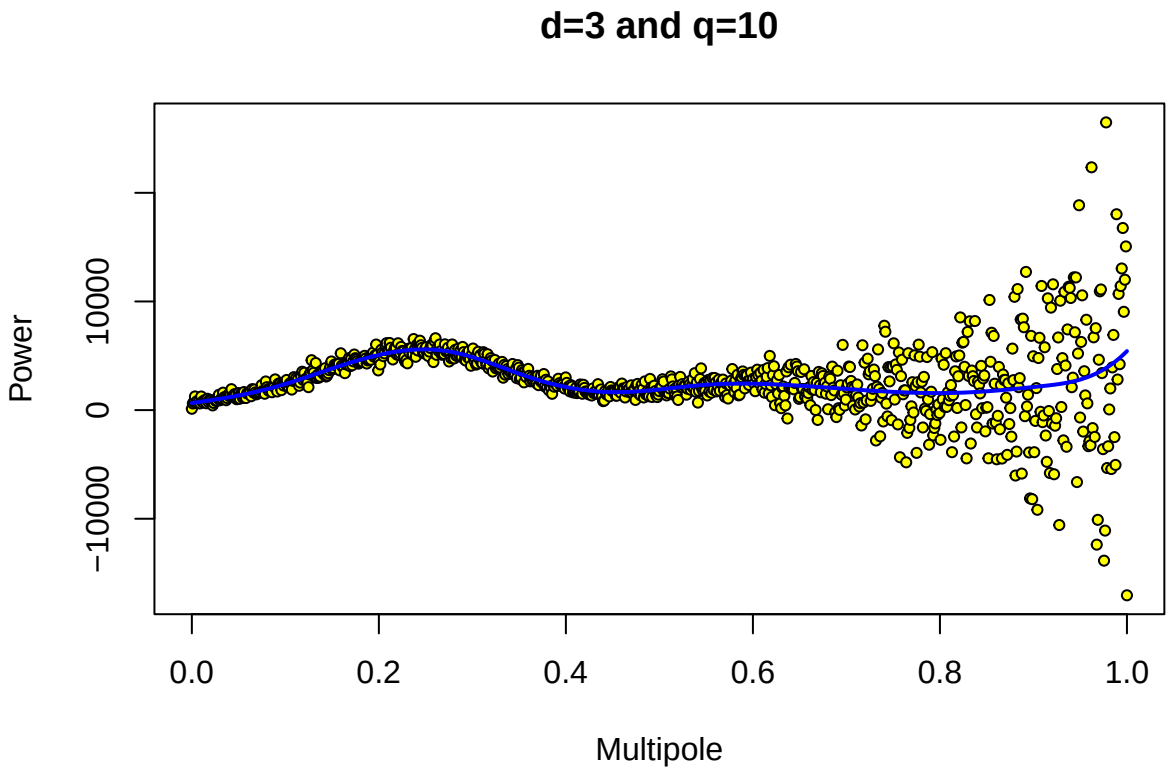
```
reg<-lm(y~x+I(x^2)+I(x^3)+I((x-0.25)^3*(x>=0.25))+I((x-0.50)^3*(x>=0.50))+
          I((x-0.75)^3*(x>=0.75)), data = wmap)
plot(y ~x, wmap, pch = 21, bg = "yellow", cex = .7, main = "d=3 and q=3",
     xlab = "Multipole", ylab = "Power")
lines(wmap$x, predict(reg), col= 'blue', lwd=2)
```



```r
# fitting a model with degree=1 and q(number of knots)=10 to the data

Mat1_10 = DMatrix(wmap$x, 1, 10)

reg<-lm(y~ Mat1_10, data = wmap)
plot(y ~x, wmap, pch = 21, bg = "yellow", cex = .7, main = "d=1 and q=10",
     xlab = "Multipole", ylab = "Power")
lines(wmap$x, predict(reg), col= 'blue', lwd=2)
```



```r
# fitting a model with degree=3 and q(number of knots)=10 to the data
Mat3_10 = DMatrix(wmap$x, 3, 10)

reg<-lm(y~Mat3_10, data = wmap)
plot(y ~x, wmap, pch = 21, bg = "yellow", cex = .7, main = "d=3 and q=10",
     xlab = "Multipole", ylab = "Power")
lines(wmap$x, predict(reg), col= 'blue', lwd=2)
```



### 1.3 Model Evaluation

The functions that we're going to use to evaluate our model:

After training the model on the training data, we should evaluate our model performance on new data (unseen/evaluation data). To evaluate our model we used the cross validation method.

```
ourLOOCV <- function(mat, mydata){
  n <- nrow(mydata)
  oneout <- vector()
  for (i in 1:n){
    # We build a vector (vect) that contains the subsets of the data on which we want
    # to train the model(all the data excluding the i-element)

    if(i==1){
      vect <- c(2:n)
    }else if(i==n){
      vect <- c(1:(n-1))
    }else{
      vect <- c(1:(i-1))
      vect1 <- c((i+1):n)
      vect <- c(vect, vect1)
    }
    fit_i  <- lm( y ~ mat, data = mydata, subset = vect )
    yhat_i <- predict(fit_i, mydata)
    oneout[i] <- ( mydata$y[i] -  yhat_i[i] )^2
  }
  return (mean(oneout)) # LOOCV-score
}

# Implementing a grid search
ourGridSearch <- function(mydata, qmin, qmax){
  if(qmin>qmax){
    return("error, qmin greater than qmax")
  }
  degrees <- c(1,3)
  points <- c(qmin:qmax)

  #The Design matrix for the first case
  Mat <- DMatrix(mydata$x, 1, qmin)

  # We set the minimum score(error) to be equal to the score of the first combination
  min_score <-ourLOOCV(Mat, mydata)

  # To get the best score and the corresponding combination
  # Of degree and number of knots (d, q)
  best = c(1,qmin)
  for(i in 1:length(degrees)){
    for(j in 1:length(points)){
      if(i!=1 && j!= 1){ # We don't have to recalculate the score for (1, qmin)
        d <- degrees[i]
        q <- points[j]
        Mat <- DMatrix(mydata$x, d, q)
        score <- ourLOOCV(Mat, mydata)
```

```
        if(score<min_score){
          min_score <- score
          best <- c(d, q)
        }
      }
    }
  }
  return(best)
}

# To evaluate the polynomial model performance
ourPoly3LOOCV <- function(mydata, d){
  n <- nrow(mydata)
  oneout <- vector()
  for (i in 1:n){
    # We build a vector (vect) that contains the subset of the data on which we want
    # To train (all the data except for the i-element)
    if(i==1){
      vect <- c(2:n)
    }else if(i==n){
      vect <- c(1:(n-1))
    }else{
      vect <- c(1:(i-1))
      vect1 <- c((i+1):n)
      vect <- c(vect, vect1)
    }
    fit_i  <- lm( y ~ poly(x,d), data = mydata, subset = vect )
    yhat_i <- predict(fit_i, mydata)
    oneout[i] <- ( mydata$y[i] -  yhat_i[i] )^2
  }
  return (mean(oneout)) # LOOCV-score
}


GTunedPolyRegression <- function(mydata){
  d <- 1
  min_score <- ourPoly3LOOCV(mydata, d)
  while(TRUE){
    d <- d + 1
    score <- ourPoly3LOOCV(mydata, d)
    if(score>min_score){
      # We have a kind of momentum to avoid local minimums
      k <- 0
      for(i in 1:5){
        k <- k + 1
        d <- d + 1
        score <- ourPoly3LOOCV(mydata, d)
        if(score<min_score){
          break
        }
      }
      if(k == 5){
        return(d-6)
```

```
      }
    }
    min_score <- score
  }
}
```

Now we are going to use `ourGridSearch(mydata, qmin, qmax)` to find the spline with the combination of $d \in \{1, 3\}$ and $q \in [q_{min}, q_{max}]$ that best fits the data *mydata*. To choose the best combination of d and q, we will run our implementation of the LOOCV method: `ourLOOCV(mat, mydata)`, which takes a design matrix *mat* and the data *mydata* as input.

This is the best combination of $d$ and $q$, which gives us the best score:

```
best_couple <- ourGridSearch(wmap, 3, 10) # From qmin = 3 to qmax = 10
best_couple
```

```
## [1] 3 4
```

Afterwards, we can use `DMatrix()` with those $d$ and $q$ (the best combination) to recall the design matrix of the spline that best fits our data *wmap*. We train the model(the best design matrix) using `lm()` and return the coefficients resulted from training the model.

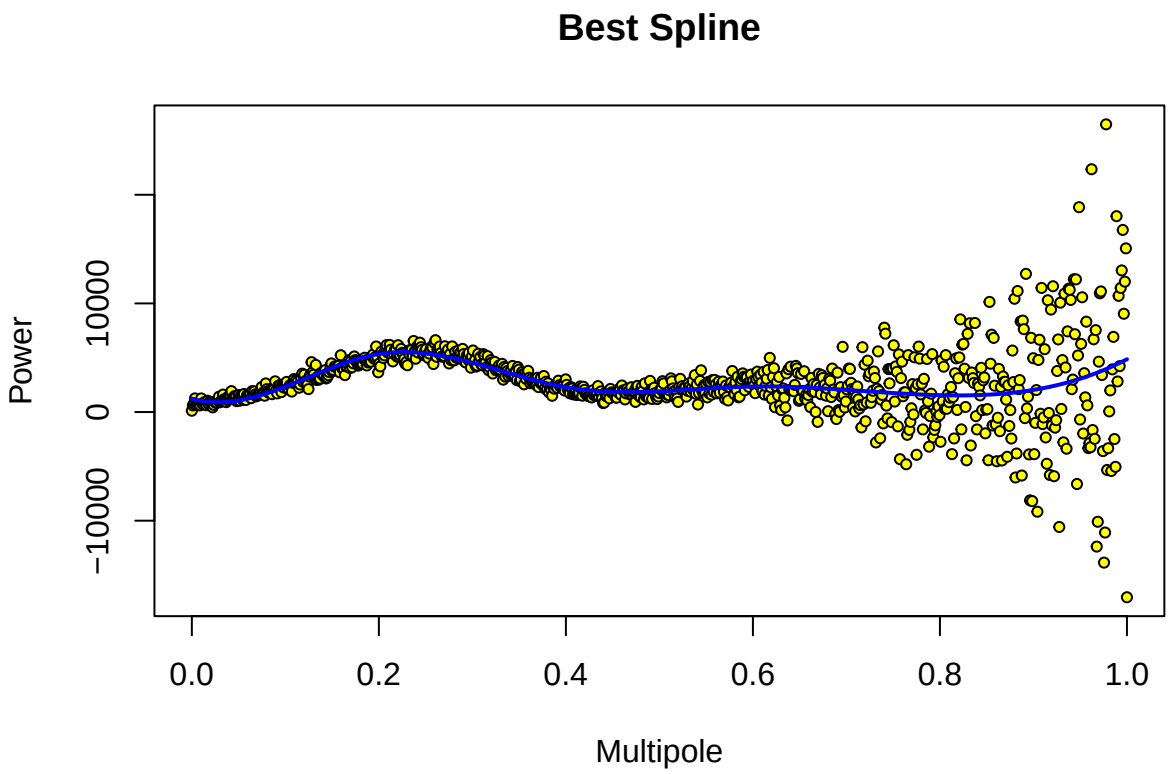
The coefficients:

```
best_mat <- DMatrix(wmap$x, best_couple[1], best_couple[2])
best_spline <- lm(y ~ best_mat, data = wmap )

best_coefficients <- best_spline$coefficients
best_coefficients
```

```
##  (Intercept)    best_mat1    best_mat2    best_mat3    best_mat4    best_mat5
##     1239.072   -24343.431   482249.626 -1289585.132  2020665.579 -1083449.201
##    best_mat6    best_mat7
##   536564.349    50096.746
```

Plotting the splines:

```
plot(y ~x, wmap, pch = 21, bg = "yellow", cex = .7,
     main = "Best Spline",
     xlab = "Multipole", ylab = "Power")
lines(wmap$x, predict(best_spline), col= 'blue', lwd=2)
```




Showing the LOOCV score of our spline by directly calling `ourLOOCV()` function:

```
bestSpline_score = ourLOOCV(best_mat, wmap)
bestSpline_score
```

```
## [1] 9706695
```

### 1.4

Using `GTunedPolyRegression(mydata)` we can find the LOOCV-tuned polynomial regression that best fits the data *mydata*. The idea is that as long as the LOOCV-score of the current p-regression is less than the minimum score, we continue updating our current minimum score and searching for our best p-regression by increasing $d$ of 1.

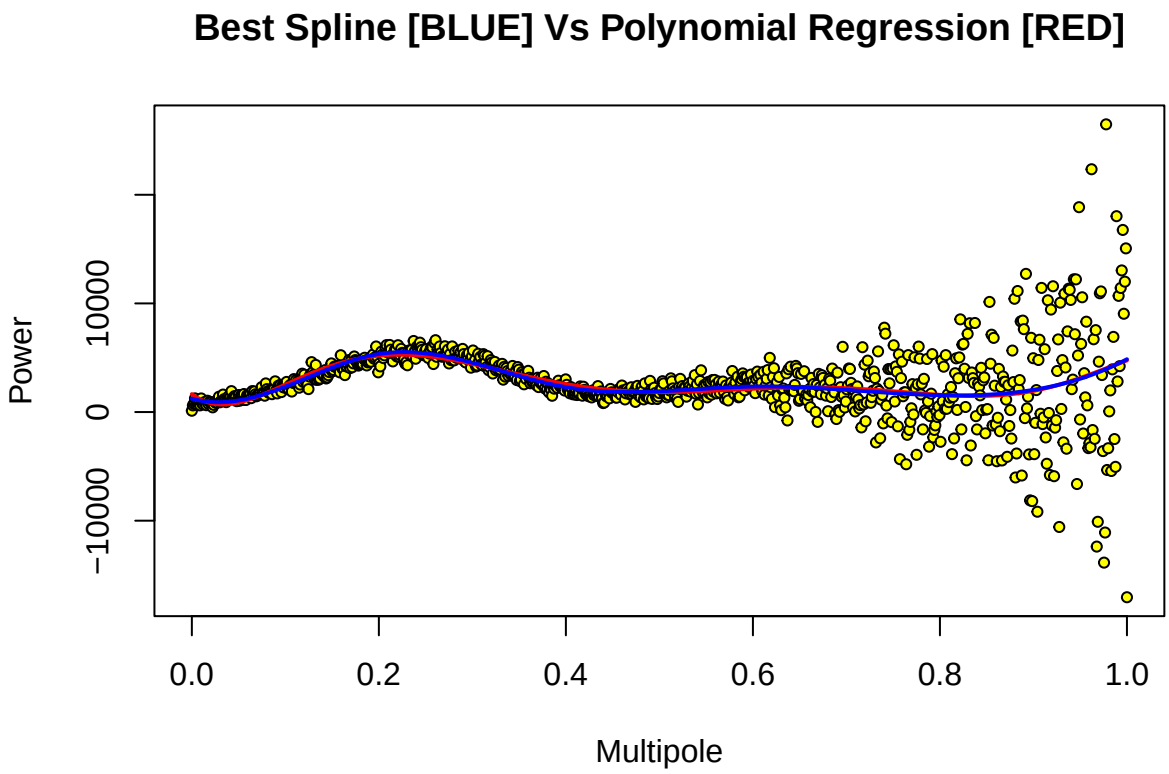
If the current regression has a LOOCV-score greater than the previous one, we check if any of the following $K$ (we have set $K = 5$) p-regression have a LOOCV-score less than our minimum score; if so, we continue our search from that degree on, otherwise we return the degree of the p-regression with score equal to the founded minimum score. In this way, there is less possibility to fall into a local minimum (for example p-regression of degree 2 has a LOOCV-score less than the one of the degree 2 p-regression).

Then we can return the degree of the best polynomial regression and plot it with our best spline to compare their result:

```
gT_d = GTunedPolyRegression(wmap)
# We look at the degree of the GCV-tuned pol-regression that we have found
gT_d
```

```
## [1] 7
```

```
best_rpoly<-lm(y~poly(x, gT_d), data = wmap)
plot(y ~x, wmap, pch = 21, bg = "yellow", cex = .7,
     main = "Best Spline [BLUE] Vs Polynomial Regression [RED]",
     xlab = "Multipole", ylab = "Power")
lines(wmap$x, predict(best_rpoly), col= 'red', lwd=2)
lines(wmap$x, predict(best_spline), col= 'blue', lwd=2)
```




It can be seen that they look very similar, indeed if we plot the corresponding LOOCV scores:

```
bestRPoly_score <- ourPoly3LOOCV(wmap, gT_d)
bestRPoly_score
```

```
## [1] 9795824
```

```
bestSpline_score
```

```
## [1] 9706695
```

```
if(bestRPoly_score>bestSpline_score){
  print("Best spline has a better LOOCV-score")
}else if(bestRPoly_score<bestSpline_score){
  print("Best p-regression has a better LOOCV-score")
}else{
  print("Same score")
}
```

```
## [1] "Best spline has a better LOOCV-score"
```

```
best_coefficients
```

```
##  (Intercept)    best_mat1    best_mat2    best_mat3    best_mat4    best_mat5
##     1239.072   -24343.431   482249.626 -1289585.132  2020665.579 -1083449.201
##    best_mat6    best_mat7
##   536564.349    50096.746
```

```
# best Coefficients of polynomial regression model
best_rpoly$coefficients
```

```
##    (Intercept) poly(x, gT_d)1 poly(x, gT_d)2 poly(x, gT_d)3 poly(x, gT_d)4
##       2687.808      -9892.706      -1313.531      29200.926     -10169.335
## poly(x, gT_d)5 poly(x, gT_d)6 poly(x, gT_d)7
##       7324.114      13769.993      -9852.154
```

By looking at the coefficients, We can notice that both of them have 7 coefficients. This is not surprising because the number of coefficients for a polynomial regression is equal to its degree (in our case 7) and the number of coefficients for a spline is equal to sum of its degree and number of knots (in our case 3 and 4, so 7).

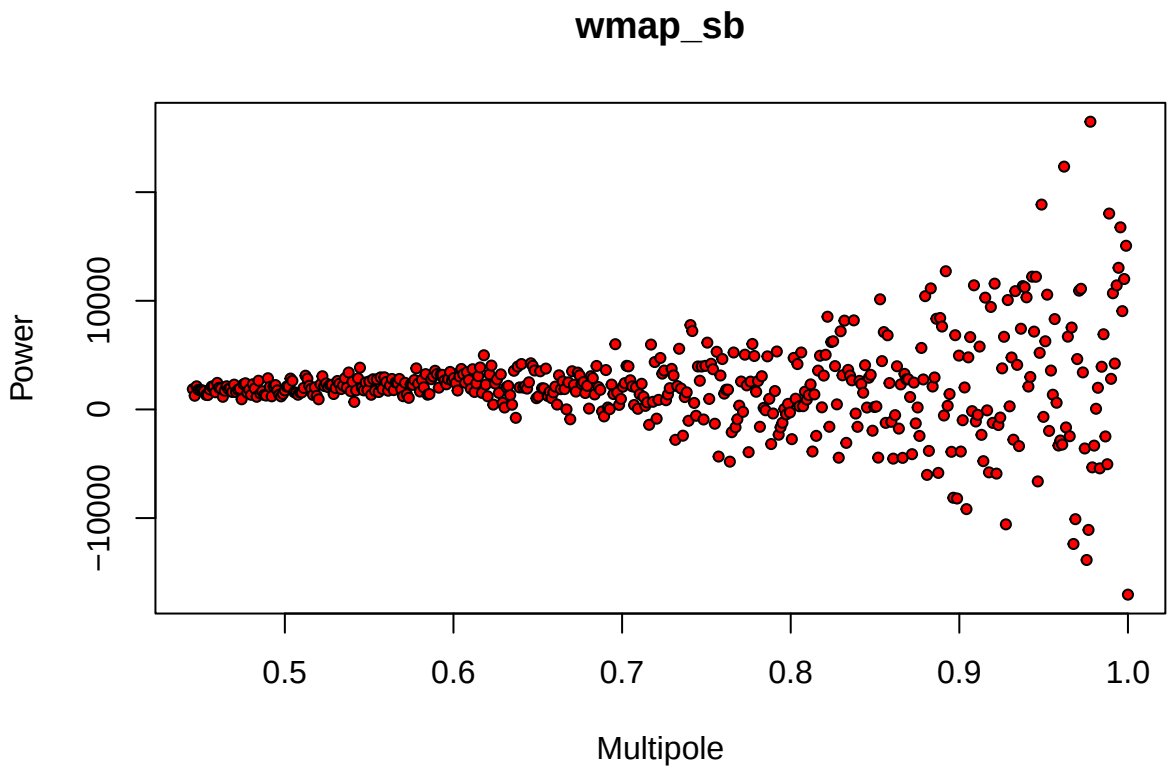
We should prefer to use the spline not only because it has a better LOOCV-score, but also because it has a smaller number of degrees (3 instead of 7) without losing any degree of freedom (still 7, 3+4), thanks to the addition of the 4 elements corresponding to the 4 knots.

## Part 2

### 2.1

In this part, we are using part of the data by dropping the first 400 observations and saving the others in a data_frame called `wmap_sb`.

```
wmap_sb <- data.frame(x = wmap$x[-(1:400)], y = wmap$y[-(1:400)])

plot(wmap_sb, pch = 21, bg = "red", cex = .7,
main = "wmap_sb", xlab = "Multipole", ylab = "Power")
```




### 2.2

We fit a simple linear regression model to the data and plot the result:

```
# Fitting a linear regression model to the data
lin_fit <- lm(y ~ x, data = wmap_sb)
summary(lin_fit)
```

```
##
## Call:
## lm(formula = y ~ x, data = wmap_sb)
##
```

```
## Residuals:
##      Min       1Q   Median       3Q      Max
## -19651.3  -1311.8     60.4   1208.1  23909.8
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   1071.7      855.2   1.253    0.211
## x             1533.6     1155.2   1.328    0.185
##
## Residual standard error: 4139 on 497 degrees of freedom
## Multiple R-squared:  0.003534,   Adjusted R-squared:  0.001529
## F-statistic: 1.762 on 1 and 497 DF,  p-value: 0.1849
```

```
# The residual error of the model
residuals(lin_fit)
```

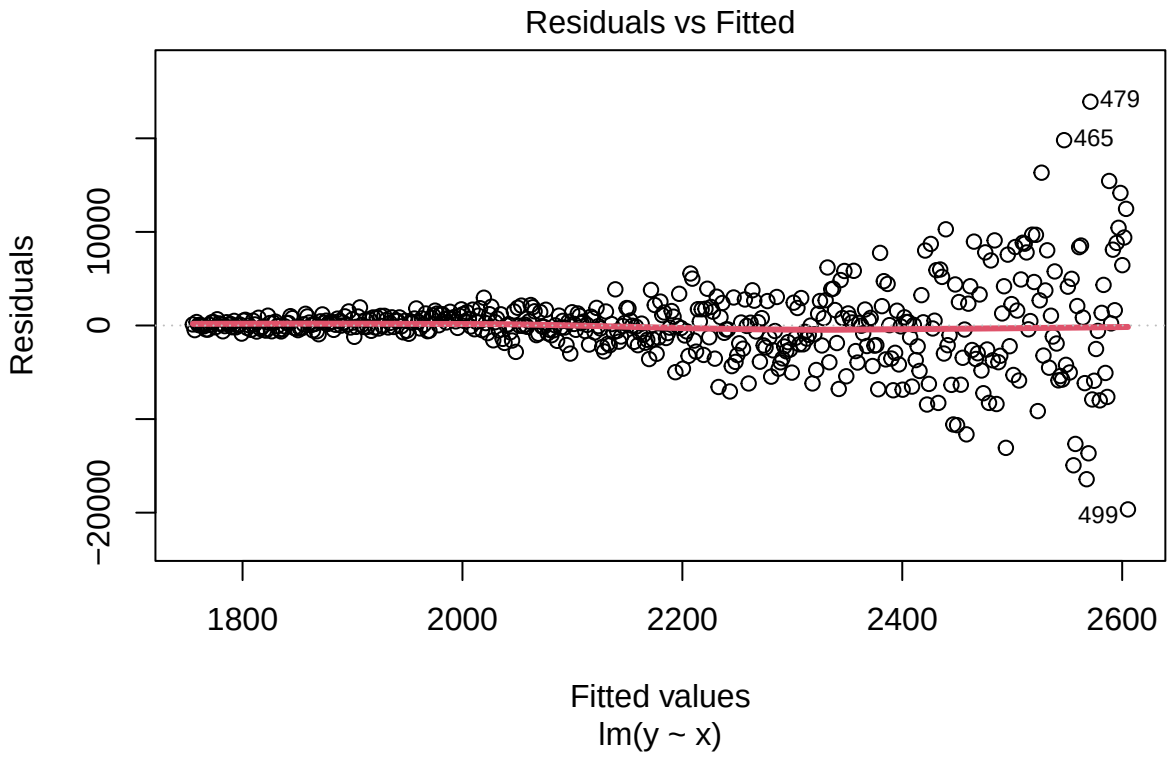
```
##            1            2            3            4            5
##    110.755376   -494.310381    369.310862     45.862105     10.264348
##            6            7            8            9           10
##     -2.815409    -84.636166   -423.578923   -444.668780     65.257564
##           11           12           13           14           15
##    330.199807    367.821150   -192.578707    664.425636    217.074879
##           16           17           18           19           20
##    144.501022   -621.927735    -64.048392    348.225752     91.239095
##           21           22           23           24           25
##    174.199238   -184.738519    490.057824   -193.708033      9.359210
##           26           27           28           29           30
##    104.121453   -864.749704    520.169940    607.755083   -266.196574
##           31           32           33           34           35
##     14.406769   -459.680088    513.563255    115.042398   -650.203359
##           36           37           38           39           40
##    826.277884   -363.610973   -142.778629   -484.435386   -544.200143
##           41           42           43           44           45
##   1048.127000    258.906343   -618.882414    280.613829    412.317972
##           46           47           48           49           50
##    -80.553785   -471.685441   -634.884198   -410.283955    -94.413712
##           51           52           53           54           55
##    266.563631    266.791874    994.929117    787.353160   -258.857497
##           56           57           58           59           60
##   -335.918253   -497.681010   -274.616767   -293.521524    128.536719
##           61           62           63           64           65
##   1247.578062    967.388105    316.171348     39.259691   -526.104965
##           66           67           68           69           70
##    168.763178   -618.436579   -936.969036    441.447007   1182.406250
##           71           72           73           74           75
##    278.882293    516.068636    177.209979    362.591122    392.111366
##           76           77           78           79           80
##   -465.266391     78.081852    729.643095    463.027438    -14.401419
##           81           82           83           84           85
##    311.563824    976.528167     99.793310   1502.235454   -215.507203
##           86           87           88           89           90
##    646.302140  -1210.439017   -146.076574    957.951769   1936.620012
##           91           92           93           94           95
##     64.707255   -155.017502    574.431842   -147.172115    668.925228
```

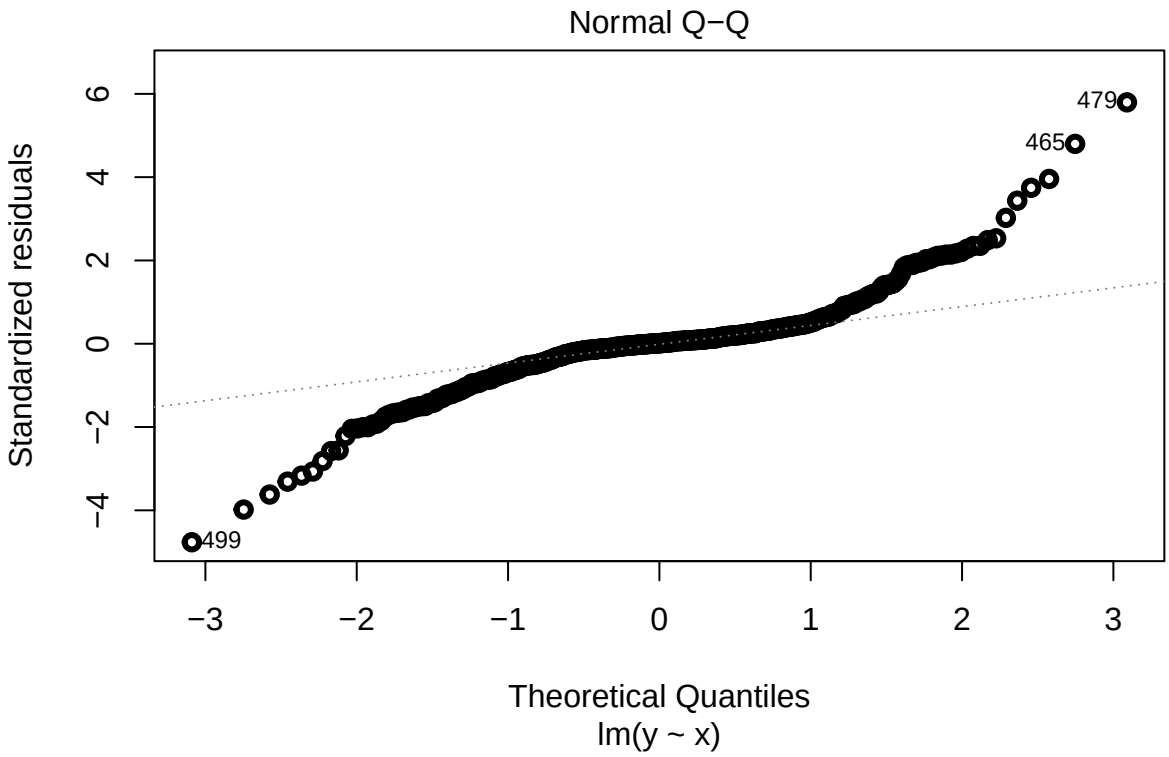
```
##           96           97           98           99          100
##  -567.994529   858.400814  -209.038943   848.080300  -320.898557
##          101          102          103          104          105
##  1025.156686   -56.856071  1028.383273   593.927516  -209.208341
##          106          107          108          109          110
##   273.792002   868.980245  -157.211612   549.200731   492.682874
##          111          112          113          114          115
##   865.464117   -91.237639  -728.127396   501.189747  -562.392910
##          116          117          118          119          120
##  -885.898767   360.370676   307.143919   748.399062  1821.358305
##          121          122          123          124          125
##   405.232549  -313.366208   759.106035    68.695278  1282.344621
##          126          127          128          129          130
##  -610.077236  -555.317993   811.770150  1063.246493  1583.891737
##          131          132          133          134          135
##  1271.511880    25.038223  1211.124466  1197.165709   679.690852
##          136          137          138          139          140
##   327.591295   808.573438  1463.875681   681.930924   888.794168
##          141          142          143          144          145
##   440.958511  -256.748446  1038.835997  1723.628240  1224.245383
##          146          147          148          149          150
##   453.582526  1424.068869   690.473212  -107.840644  1691.955599
##          151          152          153          154          155
##  -402.518158   387.971985  1048.803328  1847.566471  -495.306186
##          156          157          158          159          160
##  2968.223257   255.947400  -813.638456  1205.128787  2010.377130
##          161          162          163          164          165
## -1583.466727   353.679616   733.788759  -513.902998  1193.941345
##          166          167          168          169          170
## -1428.325012 -1860.951469    10.635975   125.104218  -731.440639
##          171          172          173          174          175
## -1610.368096  1509.227047 -2820.811810  1894.457533   -28.320324
##          176          177          178          179          180
##  2123.723919   -44.284837    85.722406  -141.260351   437.456792
##          181          182          183          184          185
##  2186.233935  1946.521278  1516.076721 -1030.669136  -880.002893
##          186          187          188          189          190
##  1439.678251  -122.687406  -146.224163  1681.646080  -837.758677
##          191          192          193          194          195
##  -470.224434 -1039.090191  -520.251948    16.736195 -1639.138462
##          196          197          198          199          200
##  1052.913682  -223.421975   489.499168  -220.124489 -2086.788346
##          201          202          203          204          205
##   396.767997 -3003.208260  1415.015583   165.327826  -460.567030
##          206          207          208          209          210
##  1284.645213  1029.384456   251.229699   407.103042  -600.277815
##          211          212          213          214          215
##    60.365428 -2037.495429   944.368914   763.817158  -741.947599
##          216          217          218          219          220
##  1880.884544   -64.105213  -319.697870 -2310.623127 -2773.115884
##          221          222          223          224          225
##  1493.583759 -1955.315098 -2110.236354   151.762489  -723.506168
##          226          227          228          229          230
##  3863.880975  -584.785682 -1725.611639 -1180.625196   -52.765953
```

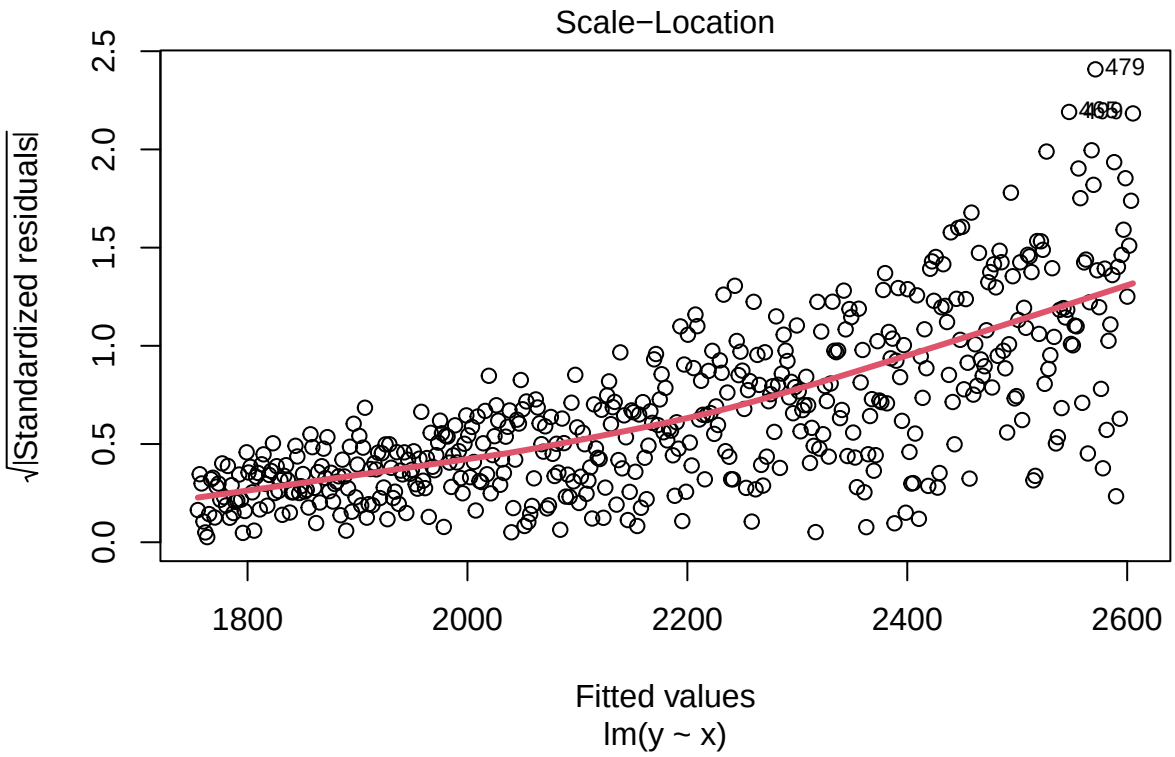
```
##          231          232          233          234          235
##   271.906390  1863.105633  1812.096777   532.006020   -28.235737
##          236          237          238          239          240
## -1749.630994  -127.182251 -2103.846108  -764.774765   198.748478
##          241          242          243          244          245
## -1000.308279 -1839.172135 -1527.958692 -3586.204549  3798.304694
##          246          247          248          249          250
## -1471.270263  2186.125980 -3025.778077 -1296.283434  2554.035909
##          251          252          253          254          255
##  1111.592053  1391.689496 -1336.896061  -809.848118  -231.276975
##          256          257          258          259          260
##  1542.960268   938.641611 -4992.856146   -47.867803  3386.176140
##          261          262          263          264          265
##  -272.986416 -4616.088273 -1065.004930  -631.613687 -3250.687344
##          266          267          268          269          270
##  5557.800699  5005.029942 -1614.362015 -2793.845772  1741.091672
##          271          272          273          274          275
##   424.506015  1733.493358 -3145.794299  1788.678644  3926.510887
##          276          277          278          279          280
## -1258.199970  1977.025573  1473.548816 -3544.664140  3076.425303
##          281          282          283          284          285
## -6571.053454   889.159689  2401.639832  -783.002825  -425.871582
##          286          287          288          289          290
##  -423.704339 -7049.047096 -4342.053952  2993.124291 -3891.804366
##          291          292          293          294          295
## -3155.752623 -1903.390280   316.905363 -2478.803294  2782.674649
##          296          297          298          299          300
##   -45.575808 -6194.982565   299.936579  3761.150722  2652.373165
##          301          302          303          304          305
##  -641.006692   344.171551 -3867.944206   784.868037 -2131.186220
##          306          307          308          309          310
## -2350.784877  2606.659567 -1304.594890 -5467.704647 -2655.956304
##          311          312          313          314          315
##  -591.964261  3046.570982 -4616.812775 -3933.688532 -3519.388389
##          316          317          318          319          320
## -2248.149145 -2757.248602 -1783.554459 -2578.611116 -5035.643073
##          321          322          323          324          325
##  2438.287070 -1319.083687  1869.776856 -2012.812501  2932.846242
##          326          327          328          329          330
## -2011.019914  -673.749371 -1402.331528  -995.306785   -10.975542
##          331          332          333          334          335
## -6191.193299  -939.504156 -4755.204013  1250.013230  2616.922474
##          336          337          338          339          340
## -2134.223583   785.544060  2698.844103  6199.157346 -3929.104211
##          341          342          343          344          345
##  3870.239932  3936.948175  1649.956418 -1874.865238 -6784.166795
##          346          347          348          349          350
##  4853.339248   796.114391  5825.611734 -5427.717123  1288.363220
##          351          352          353          354          355
##   768.337363   325.097606  5841.308350 -2734.935507 -3959.505464
##          356          357          358          359          360
##   267.523679   -24.346178  -829.322835  1706.275408 -2193.581749
##          361          362          363          364          365
##   548.400894   812.100137 -4330.958619 -2151.488176 -2103.680433
```

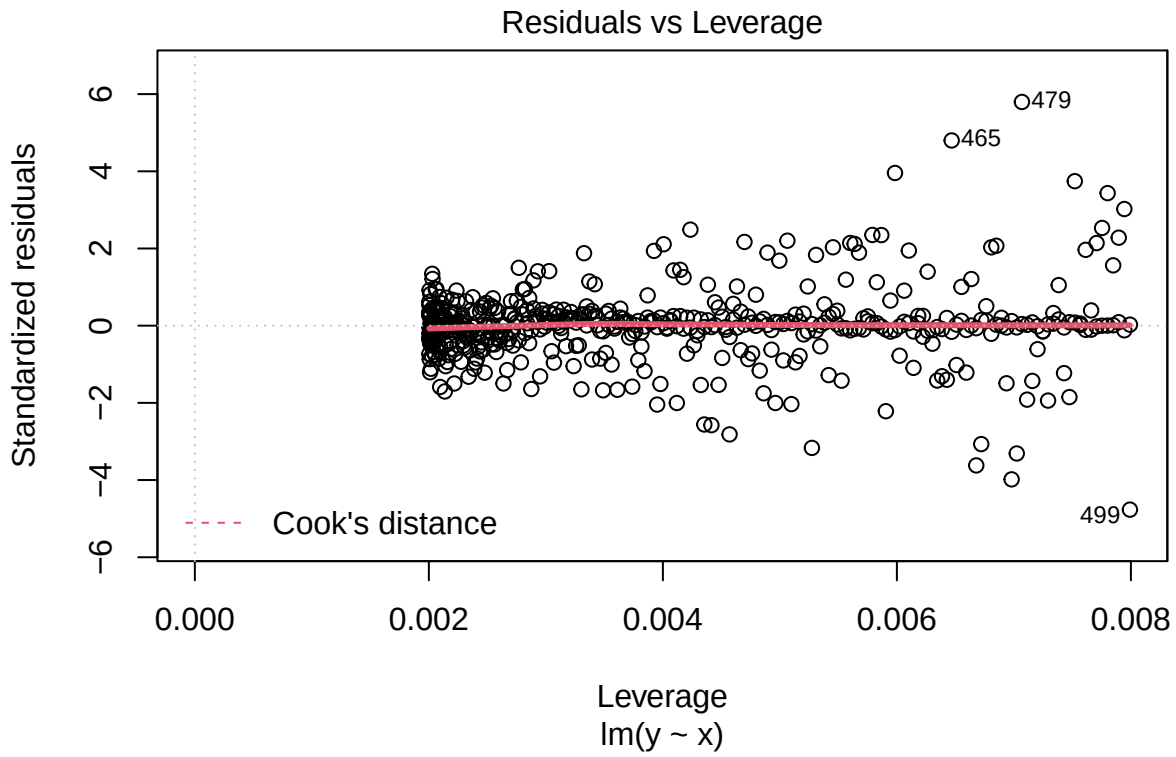
```
##          366          367          368          369          370
##  -6814.684990  7758.245453  2066.521596  4736.706839 -3622.875918
##          371          372          373          374          375
##   4439.518425    38.456669 -3529.946188 -6919.891945 -2920.161002
##          376          377          378          379          380
##   1577.981541 -4161.581116   -93.342873 -6858.157830   873.207413
##          381          382          383          384          385
##    368.465757   374.542900 -1265.048757 -6527.269314    58.683829
##          386          387          388          389          390
##  -3696.843028 -2233.062885 -4856.554442  3243.039501   339.827844
##          391          392          393          394          395
##   8006.499488 -8445.936769 -6242.038326  8714.895717  -319.174740
##          396          397          398          399          400
##    512.853303  5911.736346 -8275.491911  5981.581632  5191.203376
##          401          402          403          404          405
##  -2997.717581 10279.624362 -2105.271495 -1027.436652 -6344.982309
##          406          407          408          409          410
## -10575.398066  4385.471277 -10649.362480  2504.887764 -6328.382193
##          411          412          413          414          415
##  -3449.189150  -433.268707 -11632.593564  2332.586779  4195.131022
##          416          417          418          419          420
##  -2619.816935  8961.457208 -3577.469148 -2967.396905  3314.296338
##          421          422          423          424          425
##  -4810.597419 -7236.141176  7812.700867 -2563.018790 -8271.514447
##          426          427          428          429          430
##   6951.714296 -3719.907161  9093.642583 -8391.786374 -3924.071231
##          431          432          433          434          435
##  -3236.809488  1289.081055  4193.532498 -13074.424259  7572.467584
##          436          437          438          439          440
##  -2213.275273  2284.939271 -5287.583386  8384.846457  1597.231000
##          441          442          443          444          445
##  -5880.628557  4919.379486  8842.125829  8741.198372  7806.700515
##          446          447          448          449          450
##   -409.775541   471.158802  9703.067545  4644.226488  9688.971331
##          451          452          453          454          455
##  -9147.733026  2683.715017 16328.948660 -3213.702697  3743.972546
##          456          457          458          459          460
##   8032.255090 -4509.974767  1043.733576 -1182.366281  5776.003162
##          461          462          463          464          465
##  -1925.278995 -5863.119652 -5403.257309 -5780.141166 19801.705578
##          466          467          468          469          470
##  -4205.839579  4147.587864 -5020.033193  4985.498050 -14938.553907
##          471          472          473          474          475
## -12657.821264  2078.402679  8372.423422  8542.034966   842.598709
##          476          477          478          479          480
##  -6159.013148 -16427.157605 -13658.115362  23909.777481 -7904.755976
##          481          482          483          484          485
##  -5905.171033 -2514.016990  -584.797447 -8003.179303  1348.980140
##          486          487          488          489          490
##   4336.970083 -5072.562574 -7631.941031 15440.699912   226.164255
##          491          492          493          494          495
##   8102.113498  1629.563941  8825.108085 10439.629828 14162.172071
##          496          497          498          499
##   6446.364714  9399.987057 12462.039000 -19651.338657
```

```
plot(lin_fit, lwd=3)
```

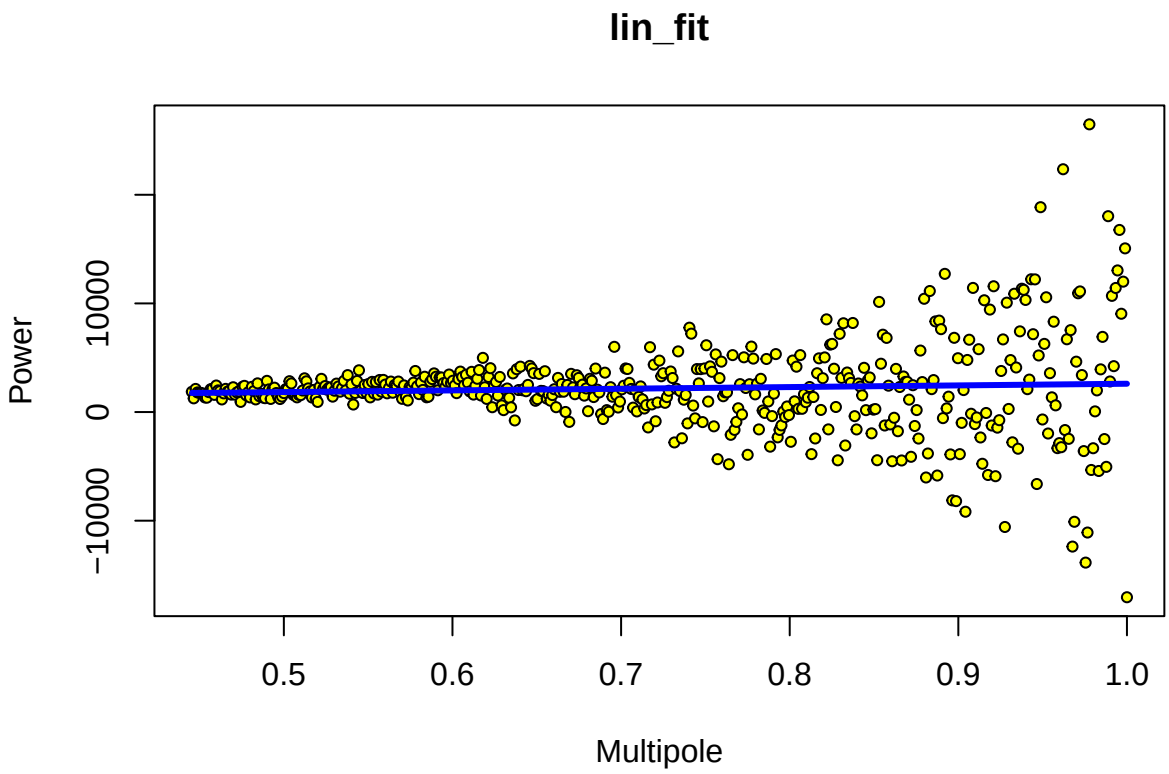



Normal Q−Q
Standardized residuals
Theoretical Quantiles
lm(y ~ x)
479
465
499
−4
−2
0
2
4
6
−3
−2
−1
0
1
2
3


Scale−Location
√|Standardized residuals|
0.0
0.5
1.0
1.5
2.0
2.5
479
1800
2000
2200
2400
2600
Fitted values
lm(y ~ x)




```
# Plotting the data and the model fitted on
plot(wmap_sb, pch = 21, bg = "yellow", cex = .7,
main = "lin_fit", xlab = "Multipole", ylab = "Power")
lines(wmap_sb$x, predict(lin_fit), col= 'blue', lwd=3)
```




As we can see linear regression can not fit these data and does not perform well. The p-value of the slope is high, which means that we can not reject the null hypothesis and the slope is equal to 0. So, linear regression can not show the relationship between x and y. Additionally, the high residual shows that the model cannot catch the variations among the data; therefore, it can't give us good predictions.

We evaluate the training mean-squared error and we store it into a variable called `MSEp_hat`

```
MSEp_hat <- mean(summary(lin_fit)$residuals^2)
MSEp_hat
```

```
## [1] 17065283
```

**2.3**

```
# We find the best spline for wmap_sb
best_2couple <- ourGridSearch(wmap_sb, 1, 10) # From qmin = 1 to qmax = 10
best_2couple
```

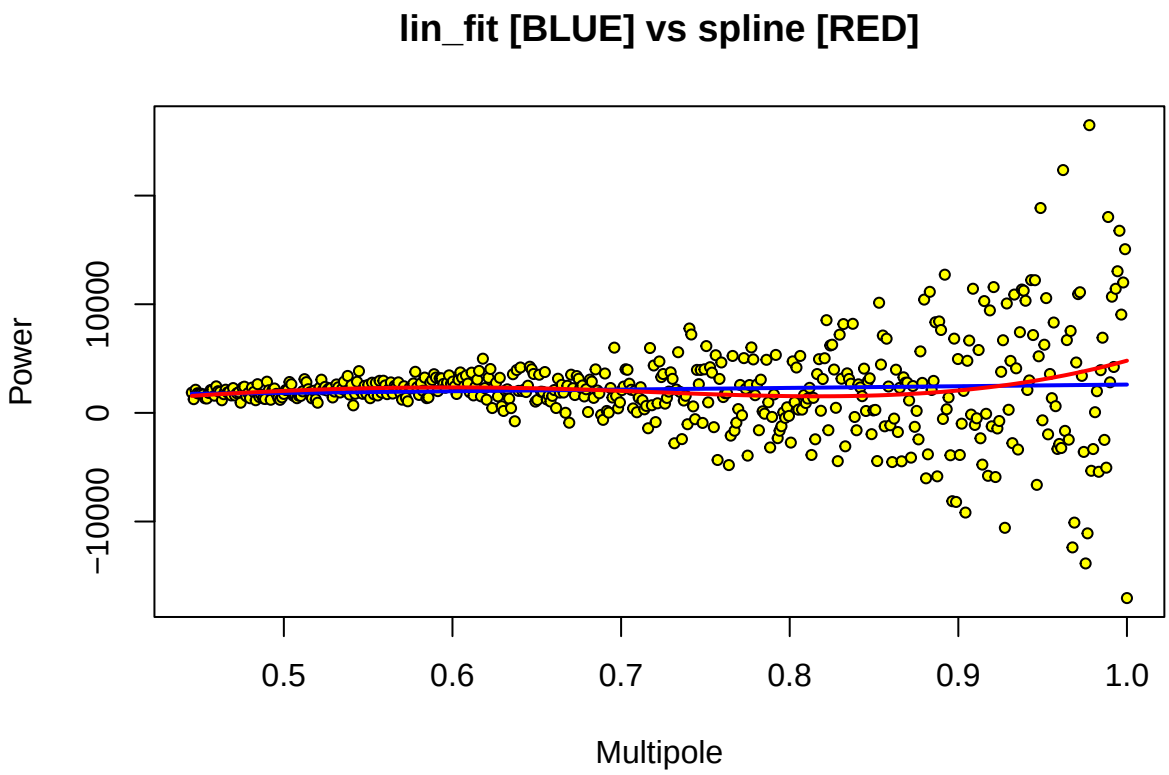
```
## [1] 3 2
```

```
# We find the spline using the founded combination of d and q
best_2mat <- DMatrix(wmap_sb$x, best_2couple[1], best_2couple[2])
best_2spline <- lm(y ~ best_2mat, data = wmap_sb )
```

```
# We calculate the MSE
MSEnp_hat <- mean(summary(best_2spline)$residuals^2)
MSEnp_hat
```

```
## [1] 16711038
```

```
plot(wmap_sb, pch = 21, bg = "yellow", cex = .7,
main = "lin_fit [BLUE] vs spline [RED]", xlab = "Multipole", ylab = "Power")
lines(wmap_sb$x, predict(lin_fit), col= 'blue', lwd=2)
lines(wmap_sb$x, predict(best_2spline), col= 'red', lwd=2)
```




By looking at the plot, we notice that the linear regression model and piece-wise regression spline have almost the same performance on these data with some slight difference at the end. the piece-wise spline model catch some variations of the data. As we saw even the MSE of the both model are somehow equal, which shows that they had a poor performance fitting the data.

**2.4**

```
t_hat <- MSEp_hat - MSEnp_hat
t_hat
```

```
## [1] 354245.8
```

### 2.5

We're going to use this function to simulate new data form the previous data by adding some noise to them:

```
# Inputs: linear model (lin_fit), x values at which to simulate (sim_x)
# Outputs: Data frame with columns x and y
sim_lm = function(lin_fit, sim_x) {
n = length(sim_x)
sim_fr = data.frame(x = sim_x)
sigma = summary(lin_fit)$sigma
y_sim = predict(lin_fit, newdata = sim_fr)
y_sim = y_sim + rnorm(n, 0, sigma) # Add noise
sim_fr = data.frame(sim_fr, y = y_sim) # Adds y column
return(sim_fr)
}
```

The above function `sim_lm()`, takes the linear regression model and the features of the data to simulate new data using the fitted parametric model(linear regression). This function takes the parameter of the data (sigma) that was obtained by the fitted linear regression model to simulate some data with noise, that is to say it uses this parameter to add some add some noise to previous data using the standard normal distribution. These new data will be used to evaluate the performance of the parametric and nonparametric models. Next, we will train both models with simulated data and calculate their MSE to see how they perform on these new data.

```
B <- 500

t_tilde <- c(1:B)
MSEp_tilde <- c(1:B)
MSEnp_tilde <- c(1:B)

for(b in 1:B){
  sim_data <- sim_lm(lin_fit, wmap_sb$x)

  lin_fit_b <- lm(y ~ x, data = sim_data)
  MSEp_tilde[b] <- mean(summary(lin_fit_b)$residuals^2)

  best_couple_b <- ourGridSearch(sim_data, 1, 10)
  best_mat_b <- DMatrix(sim_data$x, best_couple_b[1], best_couple_b[2])
  best_spline_b <- lm(y ~ best_mat_b, data = sim_data )
  MSEnp_tilde[b] <- mean(summary(best_spline_b)$residuals^2)

  t_tilde[b] <- MSEp_tilde[b] - MSEnp_tilde[b]

  print(b)
}
```

### 2.6

```
delta <- c(rep(1, B))

for(b in 1:B){ delta[b] <- delta[b] * (t_tilde[b]>t_hat) }
```

```r
p_value <- mean(delta)
p_value
```

```
## [1] 0.092
```

In eveluating the *p-value* we decided to run the simulation multiple times, most of them (pratically all them) returned a *p-value* greater then the *alpha*, so we can say that there is not enough evidence to reject the null hypotheses, which means that we can not say that there are secondary bumps or that a linear regression model is not a correct choice.

Looking at the original scatterplot, and thinking about those secondary bumps, a spline should model their behavior by setting accordinly the number of knots and the number of degrees: the best spline we've found before the *parametric boostrap*, has degree 3 and number of knots equals to 2, which makes sense if we relate this with the presence of 2 secondary bumps; so the test of comparing the best possible model for this data (a tuned spline), for which we assume true the hypotheses **"There are secondary bumps"**, with a linear model for which that hypotheses is trivialy false seems reasonable.

For the robustness of our conclusions we can say only that the *p-value* most of the time is greater than 0.05, this is due by its proximity with the value of *alpha*.