Introduction:

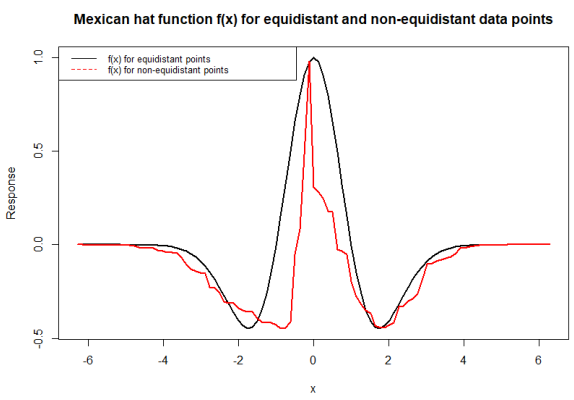
Regression methods like linear regression is a good fit for parametric data where we know the distributions of the data. For non-parametric data, underlying non-linear functions are used for estimation and prediction of response variable. The goal is to estimate non-linear functions (smooth functions) that minimizes the residuals sum of squares (errors). These are local smoothers that allow non-zero small error term. This report explores three such popular local smoothers for two sets of input data: 1) LOESS Smoothing, Kernel Smoothing, and Spline Smoothing.

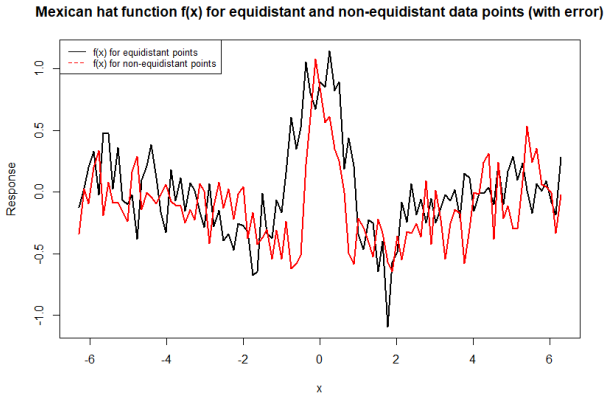
Dataset:

For the first set of input values, I simulated 101 equi-distant datapoints in the range [-2π, 2π] for 1000 Monte Carlo runs that produced a matrix of 1000 columns and 101 rows. For the second set of input values, I used non-equidistant points in the same interval [-2π, 2π] for 1000 Monte Carlo generated using R code. The famous Mexican hat function (as shown below) plus the noise/error term was used to generate y-values to apply the smooth functions on. The mexican hat function without the noise term gave the true y values or the true values of the function f(x).

$$f(x) = (1 - x^2)\exp(-0.5x^2), \qquad -2\pi \leq x \leq 2\pi,$$

Exploratory Data Analysis:





We can observe the difference in f(x) values for the equidistant and non-equidistant input datapoints in the first pic based on their shapes. The second pic is showing the same difference after including errors in the datapoints. We will generate smoothing estimates and assess how different smoothing methods and parameters perform using some evaluation metrics (bias, variance, and MSE).

Methods:

Three smoothing functions were applied to both equi-distant and non-equidistant datasets to compute three different kinds of local smoothing estimates. All three methods were applied to every data point in each 101 dataset for 1000 Monte Carlo runs.

1) LOESS
   LOESS stands for locally estimated/weighted scatterplot smoothing algorithm that combines weighted least squares with k-nearest neighborhood. More weight is assigned to k training data points that are closest neighbors to each data point (x) to fit a local polynomial linear regression to predict the response variable y or function f(x). I fitted the algorithm using loess() funciton in R with various degree of smoothing.

2) Nadaraya-Watson (NW) kernel smoothing with Gaussian Kernel
   NW kernel smoothing method uses weighted average function with Gaussian kernel to decide closest neighbors and predict the response variable y or function f(x). I fitted the algorithm using ksmooth() function in R with various bandwidth values that decide the size of the weights which is used to control smothness ot roughness of the estimates.

3) Spline smoothing (SS) with the default tuning parameter
   Spline smoothing function helps to fit flexible curves or shapes that uses smoothing paramater lambda or Spar to trade-off fitting the data (by minimizing residual errors) and roughness of function. I fitted the algorithm using smooth.spline function in R using default parameter value selected by the dataset and also some other specific values.

<u>Methods of evaluations:</u>

To evaluate performances (overfitting, underfitting, good-fitting) of three smoothing methods, three metrics were computed for every data point in each 101 dataset for 1000 Monte Carlo runs.

$$\widehat{Bias\{f(x_i)\}} = \bar{f}_m(x_i) - f(x_i), \quad \text{with } \bar{f}_m(x_i) = \frac{1}{m}\sum_{j=1}^{m} \hat{f}^{(j)}(x_i)$$

$$\widehat{Var\{f(x_i)\}} = \frac{1}{m}\sum_{j=1}^{m} \left(\hat{f}^{(j)}(x_i) - \bar{f}_m(x_i)\right)^2,$$

$$\widehat{MSE\{f(x_i)\}} = \frac{1}{m}\sum_{j=1}^{m} \left(\hat{f}^{(j)}(x_i) - f(x_i)\right)^2,$$

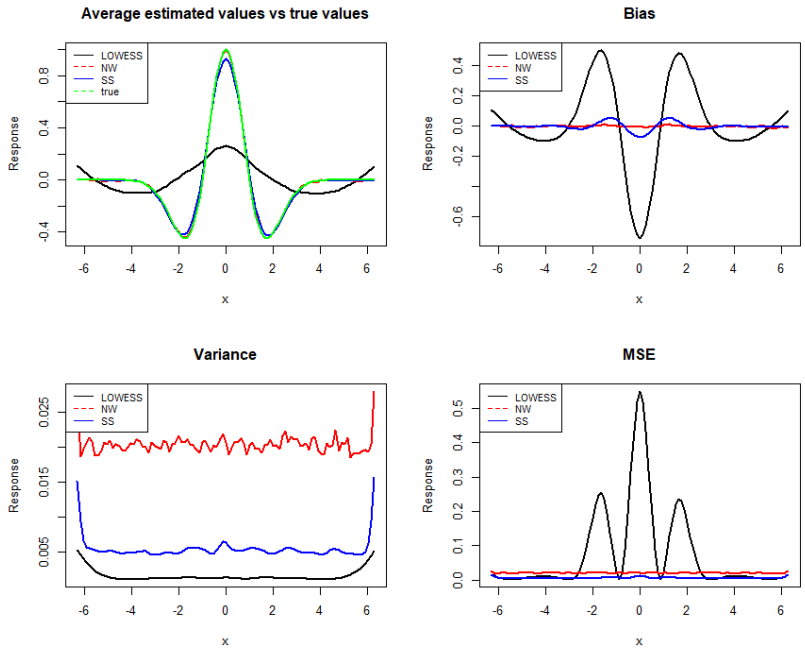
Bias: As noted in the forumula above, bias was calculated by subtracting the true values (without the errors) of y or f(x) from mean of estimated values for all three methods. In other words, each row of 1000 estimated values from 1000 monte carlo runs were averaged producing a total of 101 estimates. Subtracting the true values from these average estimates tells us how different or farther apart were the estimates in terms of represting the true values. We would like low bias as it means it does a better job of predicting the response variable.

Variance: Similarly, variance was calculated by averaging the sum of squared difference between each estimated value and mean of estimated values. This metric helps us know how jumpy our estimated values are compared to the sample values. Low variance is better as it indicates that the algorithm or model (therefore the estimated values) are not affected by changes in the dataset.

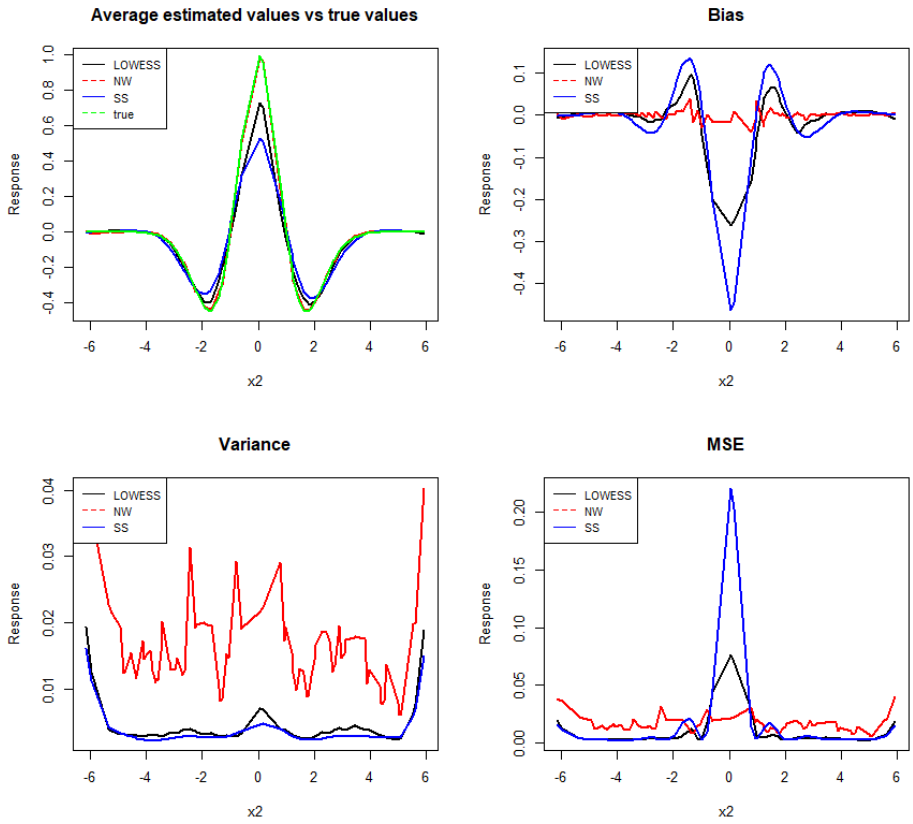
MSE: Mean Squared Error which is a combination of bias and variance was calculated by averaging the sum of squared difference between each estimated value and true value. MSE was calculated for all three sets of smoothing estimates. MSE helps us to gauge the bias-variance tradeoff and find a right balance that prevents both underfitting and overfitting.

Results:

Part 1 (equi-distant points): For comparison purposes, bias, variance, and MSE are calculated and plotted for three kinds of local smoothing estimations. The first graph compares 101 average estimates against 101 true values (green line) which looks like a Mexican hat. Lowess smoothing estimates (black line) seems to be performing significantly different than the true values. NW (red line) and SS (blue line) estimates are more aligned with the true values. Based on other graphs, Lowess estimates seem to have the highest bias and MSE but the lowest variance. NW estimates show the highest variance but the lowest bias. SS estimates are performing the best if we must consider bias-variance tradeoff.



Part 2 (Non Equi-distant points): The average estimated values of all three smoothers based on non-equidistant points in part 2 look better aligned with the true values (in green) compared to equidistant points in part 1. In this case, SS estimates has the highest bias and MSE and the lowest variance. NW estimates show the highest variance and the lowest bias. Lowess estimates seem to be performing the best regarding bias-variance tradeoff. I also noticed that all three sets of estimates look rougher in part 2 than in part 1.



Findings:

I also explored various tuning parameters in the smoothing methods to assess how they affect the performances of the estimates for both equi-distant and non-equidistant points.

I tried different values for the Span parameter for Lowess smoothing method in an increasing order of 0.5, 0.75 (given), 1.0 for equi-distant points and 0.33 (given), 0.5, 1.0 for non-equidistant points. The figures in **Appendix 1** show how bias and MSE were affected by the changes in Span tuning parameter. Larger the values, the smoother the Lowess curves are.

Similarly, I also tried different values for the bandwidth parameter for NW smoothing method in the increasing order of 0.1, 0.2 (given), and 0.5 for both equi and non equi-distant points. The figures in **Appendix 2** show how variances were affected by the changes in bandwidth tuning parameter. Larger the bandwidth size for NW smoother, the smoother NW curves are.

Likewise, I also tried different values for the Spar parameter for SS smoothing method in the order default (given), 0.5, and 1 for equi-distant points and 0.5, 0.7 (given), and 1 for non equi-distant points. The figures in **Appendix 3** show how variances were affected by the changes in Spar tuning parameter. Larger the value of Span, the smoother (almost flat) the SS curves are.

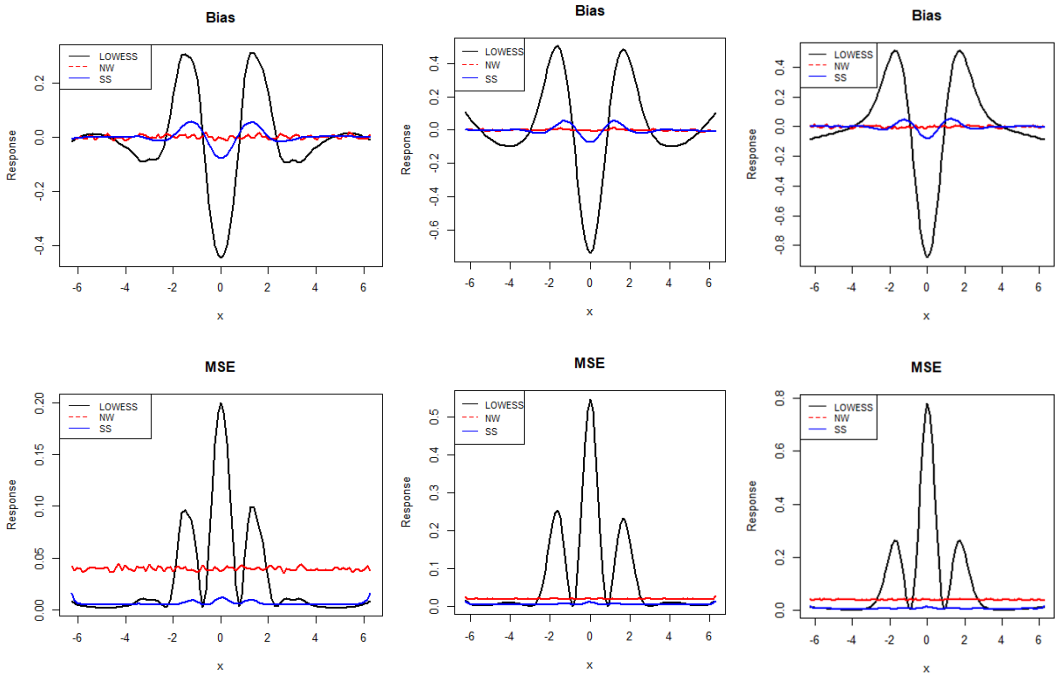
Conclusion:

Smoothing functions are very useful and commonly used in machine learning for non-parametric datasets, but it is equally important to choose the right smoothing function that yields minimum errors and performs well on bias-variance tradeoff. The three different smoothing methods performed differently on the two types of datasets. The method with the highest variance also had the highest bias/MSE and vice versa which makes sense as bias and variance have inverse relationship. SS estimates and Lowess estimates performed best for equi-distant and non-equidistant datapoints respectively balancing bias-variance tradeoff. As the values for smoothing parameters (Span for Lowess, bandwidth for NW and Spar for SS) increased, smoother the curve or lines became. We have to be careful to get the right amount of smoothness, because too smooth curves might not capture trends and too rough curves might capture some seasonal fluctuations that could just be anomalies. These methods can be improved in the future by choosing the tuning parameters using cross-validation.

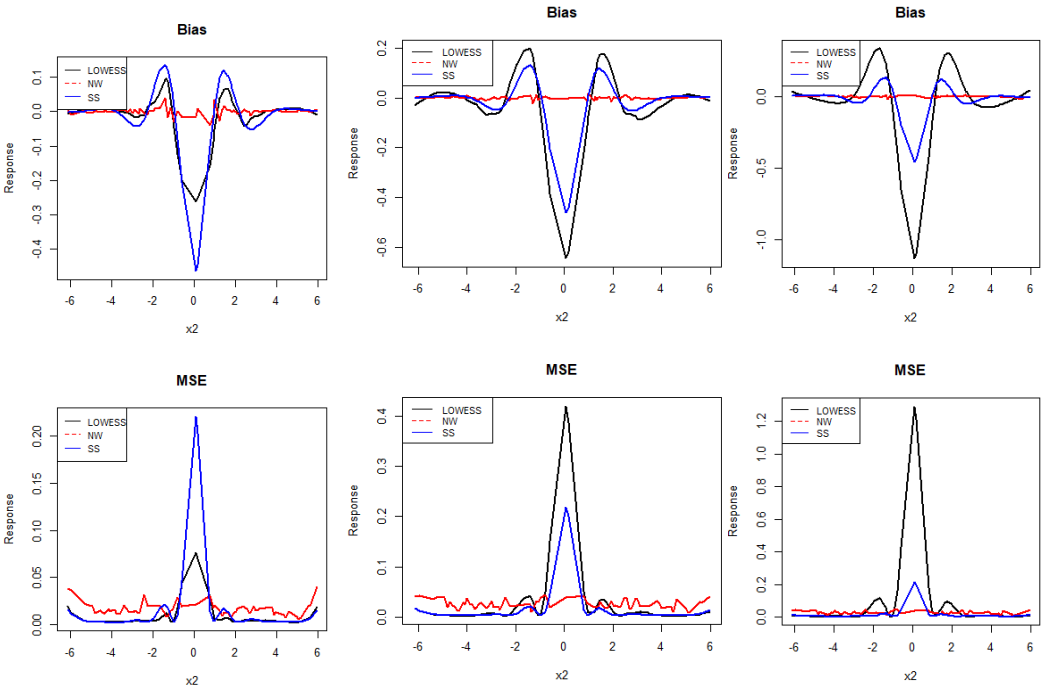
Appendix:

1. Lowess (black line):

Span=0.5 ,0.75 (given), 1.0 (equi-distant points)

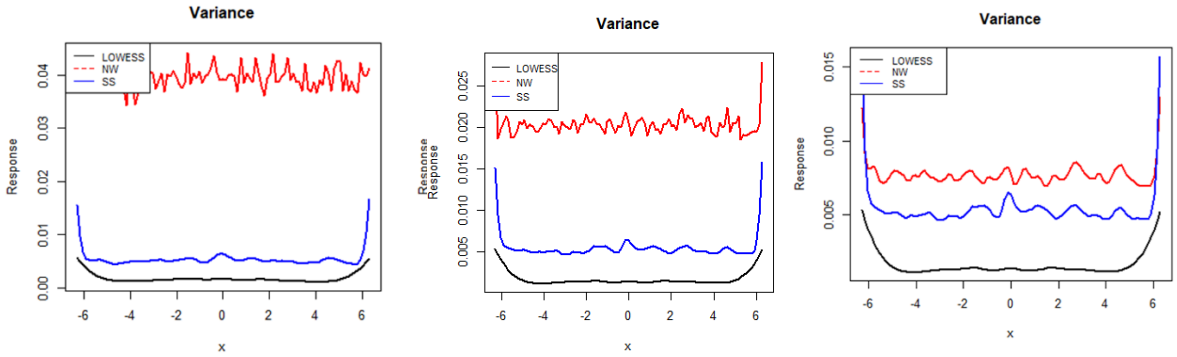


Span=0.33 (given), 0.5, 1.0 (non equi-distant points)

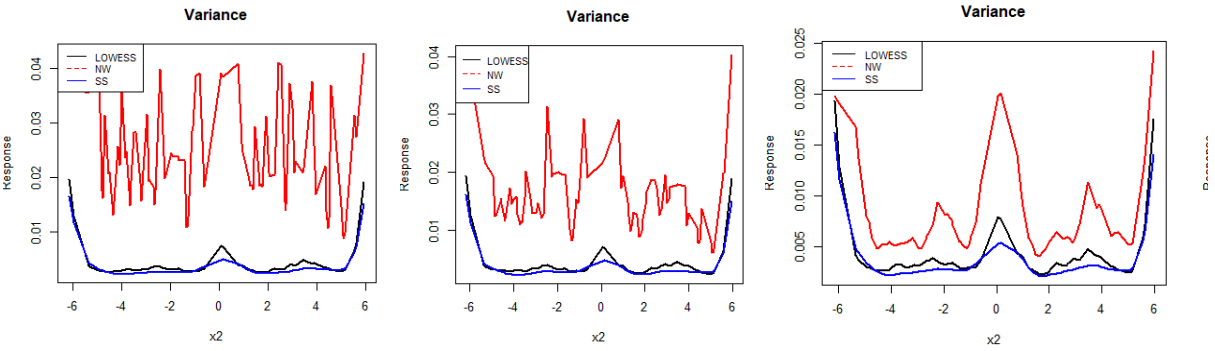


2. NW estimates (red line):

Bandwidth=0.1, 0.2 (given), and 0.5 (equi-distant points)

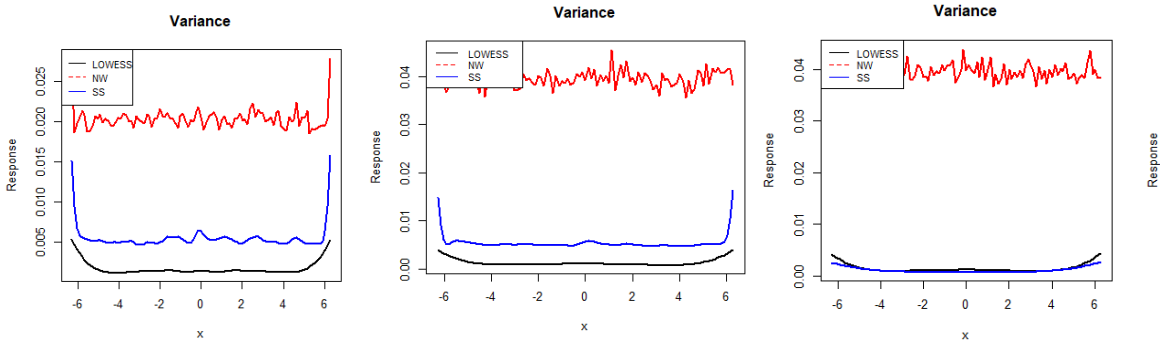


Bandwidth=0.1, 0.2 (given), and 0.5 (non equi-distant points)

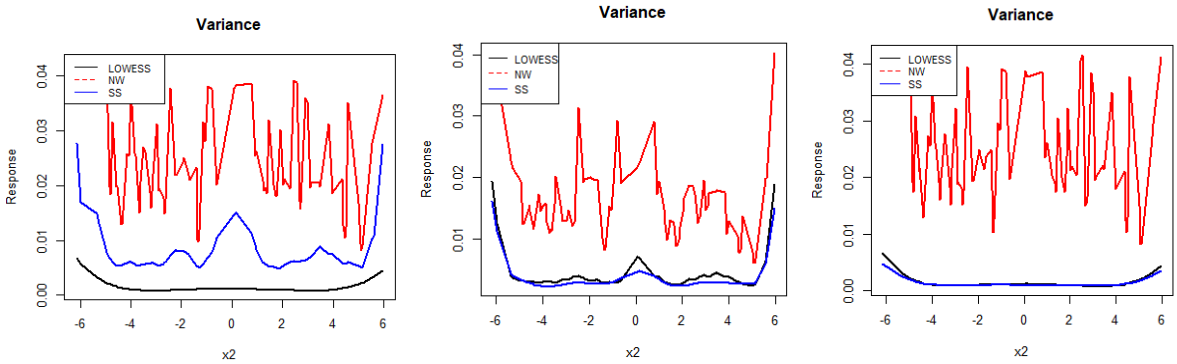


3. SS estimates (blue line):

Spar= default, 0.5, 1 (equi-distant points)



Spar= 0.5, 0.7 (given), and 1 (non equi-distant points)



R Code:

```
# Part #1 deterministic equidistant design

## Generate n=101 equidistant points in [-2\pi, 2\pi]


##EQUIDISTANT DATA FOR PART 1

x <- 2*pi*seq(-1, 1, length=n)


##NON-EQUIDISTANT DATA FOR PART 2

x2 <- round(2*pi*sort(c(0.5, -1 + rbeta(50,2,2), rbeta(50,2,2))), 8)


##EDA OF MEXICAN HAT FUNCTION

#y <- (1-x^2)*exp(-0.5*x^2)

y <- ((1-x^2)*exp(-0.5*x^2) + rnorm(length(x), sd=0.2))

#y2 <- (1-x2^2)*exp(-0.5*x2^2)

y2 <- ((1-x2^2)*exp(-0.5*x2^2) + rnorm(length(x), sd=0.2))


dmin= min(y,y2)

dmax = max(y,y2)


matplot(x, y, "l", ylim=c(dmin, dmax), ylab="Response",
col="black",lwd=2, lty=1, main="Mexican hat function f(x) for
equidistant and non-equidistant data points (with error)")

matlines(x, y2, col="red",lwd=2, lty=1)

legend("topleft", legend=c("f(x) for equidistant points", "f(x) for
non-equidistant points"),

       col=c("black", "red"), lty=1:2, cex=0.8)


##########################################################################
##

#PART 1
```

```
m <- 1000

n <- 101


## Initialize the matrix of fitted values for three methods

fvlp <- fvnw <- fvss <- matrix(0, nrow= n, ncol= m)



##Generate data, fit the data and store the fitted values


for (j in 1:m){

  ## simulate y-values


  y <- ((1-x^2)*exp(-0.5*x^2) + rnorm(length(x), sd=0.2));   #Mexican
hat function


  ## Get the estimates and store them

  fvlp[,j] <- predict(loess(y ~ x, span = 0.75), newdata = x); #LOESS
with span = 0.75

  fvnw[,j] <- ksmooth(x, y, kernel="normal", bandwidth= 0.2,
x.points=x)$y; #nadaraya-watson (NW) kernel smoothing with gaussian
kernel and bandwith=0.2

  fvss[,j] <- predict(smooth.spline(y ~ x), x=x)$y #spline smoothing
with default tuning parameter

}


##true y value from mexican hat function

y <- ((1-x^2)*exp(-0.5*x^2))



par(mfrow=c(2,2))
```

```
## Below is the sample R code to plot the mean of three estimators in
a single plot

meanlp = apply(fvlp,1,mean);

meannw = apply(fvnw,1,mean);

meanss = apply(fvss,1,mean);


dmin = min(meanlp, meannw, meanss);

dmax = max( meanlp, meannw, meanss);


matplot(x, meanlp, "l", ylim=c(dmin, dmax), ylab="Response",
col="black",lwd=2, lty=1, main="Average estimated values vs true
values")

matlines(x, meannw, col="red",lwd=2, lty=1)

matlines(x, meanss, col="blue",lwd=2, lty=1)

matlines (x, y, col="green",lwd=2, lty=1)

legend("topleft", legend=c("LOWESS", "NW","SS","true"),

       col=c("black", "red","blue","green"), lty=1:2, cex=0.8)




## plot the empirical bias/variance/MSE




##bias


Biaslp <-  meanlp - y

Biasnw <- meannw - y

Biasss <- meanss - y


Biasmin = min( Biaslp, Biasnw, Biasss);
```

```
Biasmax = max( Biaslp, Biasnw, Biasss);


matplot(x, Biaslp, "l", ylim=c(Biasmin, Biasmax),
ylab="Response",col="black",lwd=2, lty=1,main="Bias")
matlines(x, Biasnw, col="red",lwd=2, lty=1)
matlines(x, Biasss, col="blue",lwd=2, lty=1)
legend("topleft", legend=c("LOWESS", "NW","SS"),
       col=c("black", "red","blue"), lty=1:2, cex=0.8)



##variance


varlp <- 1/1000*(apply((fvlp- meanlp)^2,1,sum))
varnw <- 1/1000*(apply((fvnw- meannw)^2,1,sum))
varss <- 1/1000*(apply((fvss- meanss)^2,1,sum))



varmin = min( varlp, varnw, varss);
varmax = max( varlp, varnw, varss);


matplot(x, varlp, "l", ylim=c(varmin, varmax),
ylab="Response",col="black",lwd=2, lty=1,main="Variance")
matlines(x, varnw, col="red",lwd=2, lty=1)
matlines(x, varss, col="blue",lwd=2, lty=1)
legend("topleft", legend=c("LOWESS", "NW","SS"),
       col=c("black", "red","blue"), lty=1:2, cex=0.8)



##MSE
```

```
MSElp <- 1/1000*(apply((fvlp - y)^2,1,sum))

MSEnw <- 1/1000*(apply((fvnw - y)^2,1,sum))

MSEss <- 1/1000*(apply((fvss - y)^2,1,sum))


MSEmin = min(MSElp, MSEnw, MSEss);

MSEmax = max(MSElp, MSEnw, MSEss);


matplot(x, MSElp, "l", ylim=c(MSEmin, MSEmax),
ylab="Response",col="black",lwd=2, lty=1,main="MSE")

matlines(x, MSEnw, col="red",lwd=2, lty=1)

matlines(x, MSEss, col="blue",lwd=2, lty=1)

legend("topleft", legend=c("LOWESS", "NW","SS"),

       col=c("black", "red","blue"), lty=1:2, cex=0.8)
```

```
#########################################################################
#########
```

```
## Part #2 non-equidistant design
```

```
set.seed(79)

x2 <- round(2*pi*sort(c(0.5, -1 + rbeta(50,2,2), rbeta(50,2,2))), 8)
```

```
## assume you save the file "HW04part2.x.csv" in the local folder
"C:/temp",

#x2 <- read.table(file= "C:/temp/HW04part2.x.csv", header=TRUE);
```

```
##Generate data, fit the data and store the fitted values
m <- 1000
n <- 101
#x2 <- read.table(file= "HW04part2-1.x.csv", sep = ",", header=TRUE);

## Initialize the matrix2 of fitted values for three methods
fvlp <- fvnw <- fvss <- matrix(0, nrow= n, ncol= m)



##Generate data, fit the data and store the fitted values


for (j in 1:m){
  ## simulate y-values


  y <- ((1-x2^2)*exp(-0.5*x2^2) + rnorm(length(x2), sd=0.2));
#Mex2ican hat function


  ## Get the estimates and store them
  fvlp[,j] <- predict(loess(y ~ x2, span = 0.3365), newdata = x2);
#LOESS with span = 0.75
  fvnw[,j] <- ksmooth(x2, y, kernel="normal", bandwidth= 0.2,
x.points=x2)$y; #nadaraya-watson (NW) kernel smoothing with gaussian
kernel and bandwith=0.2
  fvss[,j] <- predict(smooth.spline(y ~ x2, spar= 0.7163), x=x2)$y
#spline smoothing with default tuning parameter
}


##true y value from mex2ican hat function
```

```
y <- ((1-x2^2)*exp(-0.5*x2^2))



par(mfrow=c(2,2))


## Below is the sample R code to plot the mean of three estimators in
a single plot

meanlp = apply(fvlp,1,mean);

meannw = apply(fvnw,1,mean);

meanss = apply(fvss,1,mean);


dmin = min(meanlp, meannw, meanss);

dmax = max( meanlp, meannw, meanss);


matplot(x2, meanlp, "l", ylim=c(dmin, dmax),
ylab="Response",col="black", lwd=2, lty=1, main="Average estimated
values vs true values")

matlines(x2, meannw, col="red", lwd=2, lty=1)

matlines(x2, meanss, col="blue", lwd=2, lty=1)

matlines (x2,y, col="green", lwd=2, lty=1)

legend("topleft", legend=c("LOWESS", "NW","SS","true"),

       col=c("black", "red","blue","green"), lty=1:2, cex=0.8)






## plot the empirical bias/variance/MSE


##bias


Biaslp <-  meanlp - y
```

```
Biasnw <- meannw - y

Biasss <- meanss - y


Biasmin = min( Biaslp, Biasnw, Biasss);

Biasmax = max( Biaslp, Biasnw, Biasss);


matplot(x2, Biaslp, "l", ylim=c(Biasmin, Biasmax), ylab="Response",
col="black", lwd=2, lty=1,main="Bias")

matlines(x2, Biasnw, col="red", lwd=2, lty=1)

matlines(x2, Biasss, col="blue", lwd=2, lty=1)

legend("topleft", legend=c("LOWESS", "NW","SS"),

       col=c("black", "red","blue"), lty=1:2, cex=0.8)




##variance


varlp <- 1/1000*(apply((fvlp- meanlp)^2,1,sum))

varnw <- 1/1000*(apply((fvnw- meannw)^2,1,sum))

varss <- 1/1000*(apply((fvss- meanss)^2,1,sum))



varmin = min( varlp, varnw, varss);

varmax = max( varlp, varnw, varss);


matplot(x2, varlp, "l", ylim=c(varmin, varmax), ylab="Response",
col="black", lwd=2, lty=1,main="Variance")

matlines(x2, varnw, col="red", lwd=2, lty=1)

matlines(x2, varss, col="blue", lwd=2, lty=1)

legend("topleft", legend=c("LOWESS", "NW","SS"),

       col=c("black", "red","blue"), lty=1:2, cex=0.8)
```

```
##MSE

MSElp <- 1/1000*(apply((fvlp - y)^2,1,sum))
MSEnw <- 1/1000*(apply((fvnw - y)^2,1,sum))
MSEss <- 1/1000*(apply((fvss - y)^2,1,sum))

MSEmin = min(MSElp, MSEnw, MSEss);
MSEmax = max(MSElp, MSEnw, MSEss);

matplot(x2, MSElp, "l", ylim=c(MSEmin, MSEmax),
ylab="Response",col="black", lwd=2, lty=1,main="MSE")
matlines(x2, MSEnw, col="red", lwd=2, lty=1)
matlines(x2, MSEss, col="blue", lwd=2, lty=1)
legend("topleft", legend=c("LOWESS", "NW","SS"),
       col=c("black", "red","blue"), lty=1:2, cex=0.8)
```