Design a housing price predictor taking only floor area (plot size), number of bedrooms, and number of bathrooms into considerations. Out of total 546 data , you may take 70% for designing the predictor and 30% for validating the design. The predictor design should be done using the following methods

c) Design Predictor using Batch Gradient Descent Algorithm, Stochastic Gradient Algorithm and mini batch Gradient Descent algorithms (determining minibatch size is your choice- here it could be 10, 20, 30 etc.) with and without regularization and compare their performances in terms of % error in prediction

d) Implement the LWR algorithm on the Housing Price data set with different tau values. Find out the tau value which will provide the best fit predictor and hence compare its results with a) , b) and c) above.

Note: Some Variables are reused so it is suggested to restart the notebook first and then run the notebook in one go from top to down to avoid any error.

# Q6c Solution

In [1]:
```
# Whole Assignment will be done using numpy only
import numpy as np

# pandas is only used to read the csv file since there is no function that allows us to read string data in numpy
import pandas as pd

#Reading data using pandas
url="https://raw.githubusercontent.com/Aditya-2001/ML-Assignments-Semester-5/master/Housing%20Price%20data%20set.csv"
data = pd.read_csv(url)
data
```

Out[1]:

| | price | lotsize | bedrooms | bathrms | stories | driveway | recroom | fullbase | gashw | airco | garagepl | prefarea |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 42000.0 | 5850 | 3 | 1 | 2 | yes | no | yes | no | no | 1 | no |
| **1** | 38500.0 | 4000 | 2 | 1 | 1 | yes | no | no | no | no | 0 | no |
| **2** | 49500.0 | 3060 | 3 | 1 | 1 | yes | no | no | no | no | 0 | no |

| | price | lotsize | bedrooms | bathrms | stories | driveway | recroom | fullbase | gashw | airco | garagepl | prefarea |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **3** | 60500.0 | 6650 | 3 | 1 | 2 | yes | yes | no | no | no | 0 | no |
| **4** | 61000.0 | 6360 | 2 | 1 | 1 | yes | no | no | no | no | 0 | no |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **541** | 91500.0 | 4800 | 3 | 2 | 4 | yes | yes | no | no | yes | 0 | no |
| **542** | 94000.0 | 6000 | 3 | 2 | 4 | yes | no | no | no | yes | 0 | no |
| **543** | 103000.0 | 6000 | 3 | 2 | 4 | yes | yes | no | no | yes | 1 | no |
| **544** | 105000.0 | 6000 | 3 | 2 | 2 | yes | yes | no | no | yes | 1 | no |
| **545** | 105000.0 | 6000 | 3 | 1 | 2 | yes | no | no | no | yes | 1 | no |

546 rows × 12 columns

In [2]:
```
'''
Now we will convert the pandas columns into numpy array because we are not allowed to use any other library.
Note: We will take only those columns into consideration on which we are asked to do prediction.
To convert them into numpy array,
first we will take series object using data[column name]
and then convert it into list using list() function
and then finally we will create the numpy array.'''

# Feature Columns
PlotSize = np.array(list(data["lotsize"]))
Bedrooms = np.array(list(data["bedrooms"]))
Bathrooms = np.array(list(data["bathrms"]))

#Target Column
Price = np.array(list(data["price"]))
```

Approch for preciding in Batch GDA, Stochastic GDA and Mini Batch GDA is similar since in Batch GDA, we take all the samples available for training, whereas in Stochastic GDA, only 1 random sample is taken and in Mini Batch GDA, a random set of batch( like 10 or 20 or 40 or 100 and so on..) is taken

## Now we will not write separate functions for all. Instead we will be writing a single function that take sample size as input and will give the output W.

### 1. For Batch GDA, Sample Size will be 100% (i.e. all samples)

### 2. For Stochastic GDA, Sample Size will be 1

### 3. For Mini Batch GDA, Sample Size will be s (where s will be predefined)

# Part 1

Now we have to create the 3 predictors using GDA without regularization. For that we will create a function that will take feature as well as target columns as input as well as the sample_size and returns the predictor using GDA alogirthm. The function will also take the alpha (learning rate) as input. While calling this function we will take care that the data we pass is training data and later we also have to test our data for which we will write a separate function

In [3]:
```
'''
1. X will be the numpy array of feature columns and Y will be target column
2. In this function we need to W to optimize it further so, we will use normal equation for that.
    And for that we have normal equation function from previous part A.
'''

def GDA(X, Y, learning_rate, sample_size, W):
    '''
        We have our previous W
        For GDA we have
        new W = current W - (learning_rate/sample_size)*(∑ ((h(X)-Y).X*) ), note: ∑ is for sample_size
        Here, current W will be an numpy array and similarly X* will also be a numpy array
    For that we will simply iterate over X and add 1 over each row
    '''

    #Added 1 in each row as done in Normal Equation function
    X1=[]
    for i in range(len(X)):
        X1.append(list(np.insert(X[i],0,1)))
    X=np.array(X1)

    #Will pick s random samples from np array
    total_index=list(range(10))
    sample_index=[]
    for i in range(10):
        sample_index.append(np.random.choice(total_index))
        total_index.remove(sample_index[-1])

    #Calculating (∑ ((h(X)-Y).X*) ) and storing it into value and will be used later on
    value=np.zeros((len(W)),dtype=int)
```

```
    for i in range(len(sample_index)):
        current_index=sample_index[i]
        predicted_value=0
        for j in range(len(W)):
            predicted_value+=W[j]*X[current_index][j]
        original_value=Y[current_index]
        result=np.multiply(X[current_index],(predicted_value-original_value))
        value=np.add(value,result)

    #We will now overwrite value from (∑ ((h(X)-Y).X*) ) to (∑ ((h(X)-Y).X*) ) * (learning_rate/sample_size)
    value=np.multiply(value,learning_rate/sample_size,dtype=float)

    #Finally we have to subtract W and value np matrixes and return W as result
    W=np.subtract(W,value)
    return W

def LRNormalEquation(X, Y):
    X1=[]
    for i in range(len(X)):
        X1.append(list(np.insert(X[i],0,1)))
    X=np.array(X1)
    result1=np.dot(X.transpose(),X)
    result1=np.linalg.inv(result1)
    result2=np.dot(X.transpose(),Y)
    result=np.dot(result1,result2)
    return result

'''To call the function first we have to merge the numpy arrays into 1
So this function merges cells so that data for each index becomes as row for that part only'''
def mergeCells(cell):
    n=len(cell[0])
    m=len(cell)
    result=np.ones((n,m),dtype=int)
    for i in range(n):
        for j in range(m):
            result[i][j]=cell[j][i]
    return result
```

In [4]:

```
'''
Now we have to divide our data into testing and trainging where test size will be 30%.
For that we will take first 70% for training and rest for testing
We also have to predict data for which we can use the same predict function used in part A/
'''
train_size=int(0.7*len(PlotSize))
```

```
train_X=mergeCells([PlotSize[:train_size], Bedrooms[:train_size], Bathrooms[:train_size]])
test_X=mergeCells([PlotSize[train_size:], Bedrooms[train_size:], Bathrooms[train_size:]])
train_Y=Price[:train_size]
test_Y=Price[train_size:]

def predict(X,Y,W):
    error=0
    for i in range(len(Y)):
        predicted=abs(W[0] + W[1]*X[i][0] + W[2]*X[i][1] + W[3]*X[i][2])
        actual=abs(Y[i])
        error+=abs(actual-predicted)/actual
    error=error/len(Y)
    error=error*100
    return error
```

In [5]:

```
#Now we have to predict for testing data. The below predict function will take testing data and W and return the mean squ
#Defining epochs and learning rate for 3 algorithms
epochs=1000
learning_rate=0.0000000001
```

## Batch Gradient Descent Algorithm without regularization

In [6]:

```
W=LRNormalEquation(train_X,train_Y)
for i in range(epochs):
    W=GDA(train_X,train_Y,learning_rate,len(train_Y),W)
print("Model using Batch GDA without regularization is given below")
print(round(W[0]),"+",round(W[1]),"* PlotSize +",round(W[2]),"* Bedrooms +",round(W[3]),"* Bathrooms")
print("% error in the model on testing data using Batch GDA without regularization is",end=" ")
print(round(predict(test_X,test_Y,W),10),end="%")
```

```
Model using Batch GDA without regularization is given below
-4895 + 6 * PlotSize + 5529 * Bedrooms + 19010 * Bathrooms
% error in the model on testing data using Batch GDA without regularization is 18.6745878332%
```

## Stochastic Gradient Descent Algorithm without regularization

In [7]:

```
W=LRNormalEquation(train_X,train_Y)
for i in range(epochs):
    W=GDA(train_X,train_Y,learning_rate,1,W)
print("Model using Stochastic GDA without regularization is given below")
print(round(W[0]),"+",round(W[1]),"* PlotSize +",round(W[2]),"* Bedrooms +",round(W[3]),"* Bathrooms")
```

```
print("% error in the model on testing data using Stochastic GDA without regularization is",end=" ")
print(round(predict(test_X,test_Y,W),10),end="%")
```

```
Model using Stochastic GDA without regularization is given below
-4895 + 6 * PlotSize + 5529 * Bedrooms + 19010 * Bathrooms
% error in the model on testing data using Stochastic GDA without regularization is 18.8001486202%
```

### Mini Batch Gradient Descent Algorithm without regularization

In [8]:
```
batch_size=50
W=LRNormalEquation(train_X,train_Y)
for each in range(epochs):
    W=GDA(train_X,train_Y,learning_rate,batch_size,W)
print("Model using Mini Batch GDA without regularization is given below")
print(round(W[0]),"+",round(W[1]),"* PlotSize +",round(W[2]),"* Bedrooms +",round(W[3]),"* Bathrooms")
print("% error in the model on testing data using Stochastic GDA without regularization is",end=" ")
print(round(predict(test_X,test_Y,W),10),end="%")
```

```
Model using Mini Batch GDA without regularization is given below
-4895 + 6 * PlotSize + 5529 * Bedrooms + 19010 * Bathrooms
% error in the model on testing data using Stochastic GDA without regularization is 18.7169049039%
```

# Part 2

In [9]:
```
'''
Now we have to create the 3 predictors using GDA with regularization.
We have GDA without regularization and for regularization we will add an addition term of alpha*λ/m.
'''
def GDA_with_Regularization(X, Y, learning_rate, sample_size, W, λ):
    '''
        We have our previous W
        For GDA with regularization we have
        new W = current W * (1-alpha*λ/m) - (learning_rate/sample_size)*(∑ ((h(X)-Y).X*) ), note: ∑ is for sample_size
        Here, current W will be an numpy array and similarly X* will also be a numpy array
    For that we will simply iterate over X and add 1 over each row
    '''

    #Added 1 in each row as done in Normal Equation function
    X1=[]
    for i in range(len(X)):
        X1.append(list(np.insert(X[i],0,1)))
    X=np.array(X1)

    #Will pick s random samples from np array
```

```
    total_index=list(range(10))
    sample_index=[]
    for i in range(10):
        sample_index.append(np.random.choice(total_index))
        total_index.remove(sample_index[-1])

    #Calculating (∑ ((h(X)-Y).X*) ) and storing it into value and will be used later on
    value=np.zeros((len(W)),dtype=int)
    for i in range(len(sample_index)):
        current_index=sample_index[i]
        predicted_value=0
        for j in range(len(W)):
            predicted_value+=W[j]*X[current_index][j]
        original_value=Y[current_index]
        result=np.multiply(X[current_index],(predicted_value-original_value))
        value=np.add(value,result)

    #We will now overwrite value from (∑ ((h(X)-Y).X*) ) to (∑ ((h(X)-Y).X*) ) * (learning_rate/sample_size)
    value=np.multiply(value,learning_rate/sample_size,dtype=float)

    #We will now change W into W * (1-alpha*λ/m)
    flag=1-float((learning_rate*λ)/sample_size)
    W=np.multiply(W,flag)

    #Finally we have to subtract W and value np matrixes and return W as result
    W=np.subtract(W,value)
    return W
```

In [10]:

```
#Defining lamda/regulizer that will be passed in GDA function
λ1=-1000
λ2=1000
```

## Batch Gradient Descent Algorithm with regularization

In [11]:

```
W=LRNormalEquation(train_X,train_Y)

print("CASE-1: λ is -1000")
for i in range(epochs):
    W=GDA_with_Regularization(train_X,train_Y,learning_rate,len(train_Y),W,λ1)
print("Model using Batch GDA with regularization is given below")
print(round(W[0],2),"+",round(W[1],2),"* PlotSize +",round(W[2],2),"* Bedrooms +",round(W[3],2),"* Bathrooms")
print("% error in the model on testing data using Batch GDA with regularization is",end=" ")
print(round(predict(test_X,test_Y,W),10),end="%")
```

```
print("\n\nCASE-2: λ is +1000")
for i in range(epochs):
    W=GDA_with_Regularization(train_X,train_Y,learning_rate,len(train_Y),W,λ2)
print("Model using Batch GDA with regularization is given below")
print(round(W[0],2),"+",round(W[1],2),"* PlotSize +",round(W[2],2),"* Bedrooms +",round(W[3],2),"* Bathrooms")
print("% error in the model on testing data using Batch GDA with regularization is",end=" ")
print(round(predict(test_X,test_Y,W),10),end="%")
```

```
CASE-1: λ is -1000
Model using Batch GDA with regularization is given below
-4894.92 + 5.92 * PlotSize + 5528.57 * Bedrooms + 19010.02 * Bathrooms
% error in the model on testing data using Batch GDA with regularization is 18.6745826331%

CASE-2: λ is +1000
Model using Batch GDA with regularization is given below
-4894.92 + 5.92 * PlotSize + 5528.57 * Bedrooms + 19010.02 * Bathrooms
% error in the model on testing data using Batch GDA with regularization is 18.6821784209%
```

## Stochastic Gradient Descent Algorithm with regularization

In [12]:

```
W=LRNormalEquation(train_X,train_Y)

print("CASE-1: λ is -1000")
for i in range(epochs):
    W=GDA_with_Regularization(train_X,train_Y,learning_rate,1,W,λ1)
print("Model using Stochastic GDA with regularization is given below")
print(round(W[0]),"+",round(W[1]),"* PlotSize +",round(W[2]),"* Bedrooms +",round(W[3]),"* Bathrooms")
print("% error in the model on testing data using Stochastic GDA with regularization is",end=" ")
print(round(predict(test_X,test_Y,W),10),end="%")

print("\n\nCASE-2: λ is +1000")
for i in range(epochs):
    W=GDA_with_Regularization(train_X,train_Y,learning_rate,1,W,λ2)
print("Model using Stochastic GDA with regularization is given below")
print(round(W[0]),"+",round(W[1]),"* PlotSize +",round(W[2]),"* Bedrooms +",round(W[3]),"* Bathrooms")
print("% error in the model on testing data using Stochastic GDA with regularization is",end=" ")
print(round(predict(test_X,test_Y,W),10),end="%")
```

```
CASE-1: λ is -1000
Model using Stochastic GDA with regularization is given below
-4895 + 6 * PlotSize + 5529 * Bedrooms + 19012 * Bathrooms
% error in the model on testing data using Stochastic GDA with regularization is 18.7996525631%

CASE-2: λ is +1000
Model using Stochastic GDA with regularization is given below
```

```
-4895 + 6 * PlotSize + 5529 * Bedrooms + 19010 * Bathrooms
% error in the model on testing data using Stochastic GDA with regularization is 18.8002212587%
```

### Mini Batch Gradient Descent Algorithm with regularization

In [13]:

```
batch_size=50
W=LRNormalEquation(train_X,train_Y)

print("CASE-1: λ is -1000")
for each in range(epochs):
    W=GDA_with_Regularization(train_X,train_Y,learning_rate,batch_size,W,λ1)
print("Model using Mini Batch GDA with regularization is given below")
print(round(W[0]),"+",round(W[1]),"* PlotSize +",round(W[2]),"* Bedrooms +",round(W[3]),"* Bathrooms")
print("% error in the model on testing data using Stochastic GDA with regularization is",end=" ")
print(round(predict(test_X,test_Y,W),10),end="%")

print("\n\nCASE-2: λ is +1000")
for each in range(epochs):
    W=GDA_with_Regularization(train_X,train_Y,learning_rate,batch_size,W,λ2)
print("Model using Mini Batch GDA with regularization is given below")
print(round(W[0]),"+",round(W[1]),"* PlotSize +",round(W[2]),"* Bedrooms +",round(W[3]),"* Bathrooms")
print("% error in the model on testing data using Stochastic GDA with regularization is",end=" ")
print(round(predict(test_X,test_Y,W),10),end="%")
```

```
CASE-1: λ is -1000
Model using Mini Batch GDA with regularization is given below
-4895 + 6 * PlotSize + 5529 * Bedrooms + 19010 * Bathrooms
% error in the model on testing data using Stochastic GDA with regularization is 18.7168707596%

CASE-2: λ is +1000
Model using Mini Batch GDA with regularization is given below
-4895 + 6 * PlotSize + 5529 * Bedrooms + 19010 * Bathrooms
% error in the model on testing data using Stochastic GDA with regularization is 18.7476596132%
```

# CONCLUSION

## Result: % error is less when Regularization is done

The arrangement becomes:

# % error : GDA with Regularization < GDA without Regularization

# Maximum Error : 18.8001486202%

# Minimum Error : 18.6745826331%

### Note: I have taken learning_rate=0.0000000001, epochs=1000, as λ -1000 and +1000 and batch_size as 50 for minibatch GDA.

# Q6d

In [14]:
```
# Whole Assignment will be done using numpy only
import numpy as np

# pandas is only used to read the csv file since there is no function that allows us to read string data in numpy
import pandas as pd

#Reading data using pandas
url="https://raw.githubusercontent.com/Aditya-2001/ML-Assignments-Semester-5/master/Housing%20Price%20data%20set.csv"
data = pd.read_csv(url)
data
```

Out[14]:

| | price | lotsize | bedrooms | bathrms | stories | driveway | recroom | fullbase | gashw | airco | garagepl | prefarea |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 42000.0 | 5850 | 3 | 1 | 2 | yes | no | yes | no | no | 1 | no |
| **1** | 38500.0 | 4000 | 2 | 1 | 1 | yes | no | no | no | no | 0 | no |
| **2** | 49500.0 | 3060 | 3 | 1 | 1 | yes | no | no | no | no | 0 | no |
| **3** | 60500.0 | 6650 | 3 | 1 | 2 | yes | yes | no | no | no | 0 | no |
| **4** | 61000.0 | 6360 | 2 | 1 | 1 | yes | no | no | no | no | 0 | no |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **541** | 91500.0 | 4800 | 3 | 2 | 4 | yes | yes | no | no | yes | 0 | no |
| **542** | 94000.0 | 6000 | 3 | 2 | 4 | yes | no | no | no | yes | 0 | no |
| **543** | 103000.0 | 6000 | 3 | 2 | 4 | yes | yes | no | no | yes | 1 | no |
| **544** | 105000.0 | 6000 | 3 | 2 | 2 | yes | yes | no | no | yes | 1 | no |

| | price | lotsize | bedrooms | bathrms | stories | driveway | recroom | fullbase | gashw | airco | garagepl | prefarea |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **545** | 105000.0 | 6000 | 3 | 1 | 2 | yes | no | no | no | yes | 1 | no |

546 rows × 12 columns

In [15]:
```
'''
Now we will convert the pandas columns into numpy array because we are not allowed to use any other library.
Note: We will take only those columns into consideration on which we are asked to do prediction.
To convert them into numpy array,
first we will take series object using data[column name]
and then convert it into list using list() function
and then finally we will create the numpy array.'''

# Feature Columns
PlotSize = np.array(list(data["lotsize"]))
Bedrooms = np.array(list(data["bedrooms"]))
Bathrooms = np.array(list(data["bathrms"]))

#Target Column
Price = np.array(list(data["price"]))
```

In [16]:
```
'''
This function takes X and Y columns as input where
X are the values in feature columns and Y is the value in
target column. It applies locally weighted regression and
predicts the value acc to the algorithm
'''
def LocallyWeightedLR(X, Y, Tau):
    prediction=[]

    #Added 1 in each row as done in Normal Equation function
    X1=[]
    for i in range(len(X)):
        X1.append(list(np.insert(X[i],0,1)))
    X=np.array(X1)

    #Appying the LWR algorithm for each sample to obtain the predicted value
    #And finally adding the value into the prediction list
    for i in range(X.shape[0]):
        xi=X[i]
        X_T=np.transpose(X)
```

```
        W=kernel(X, xi, Tau)
        X_T_W=X_T * W
        X_T_WX=np.matmul(X_T_W, X)
        InverseX_T_WX=np.linalg.pinv(X_T_WX)
        X_T_WXXTW=np.matmul(InverseX_T_WX, X_T_W)
        X_T_WXXTWY=np.matmul(X_T_WXXTW, Y)
        X_T_WXXTWYT=np.transpose(X_T_WXXTWY)
        prediction.append(X_T_WXXTWYT.dot(xi))
    return prediction

def kernel(X, xi, Tau):
    return np.exp(-np.sum((xi - X) ** 2, axis = 1) / (2 * Tau * Tau))

'''To call the function first we have to merge the numpy arrays into 1
So this function merges cells so that data for each index becomes as row for that part only'''
def mergeCells(cell):
    n=len(cell[0])
    m=len(cell)
    result=np.ones((n,m),dtype=int)
    for i in range(n):
        for j in range(m):
            result[i][j]=cell[j][i]
    return result
```

In [17]:

```
#Taking all the target columns as X
X=mergeCells([PlotSize[:], Bedrooms[:], Bathrooms[:]])

#Taking all the values of price column for Y
Y=Price[:]

#This function takes Y and Y predicted and calculates error
def predict(Y, Y_prediction):
    error=0
    for i in range(len(Y)):
        error+=abs((Y[i]-Y_prediction[i])/Y[i])
    error=error/len(Y)
    error=error*100
    return error
```

In [18]:

```
#Let's assume tau as given value and predict for it
tau=0.01
predictionLWR=LocallyWeightedLR(X,Y,tau)
print("Below are initial 10 value using LWR\nOriginal Value\t\tPredicted Value")
```

```
for i in range(10):
    print(Y[i],"\t\t",predictionLWR[i])
print("After performing LWR Algorithm the % error when tau =",tau,"is",round(predict(Y, predictionLWR),2),end="%")
```

```
Below are initial 10 value using LWR
Original Value          Predicted Value
42000.0                  42000.0
38500.0                  46333.33333333335
49500.0                  49500.00000000003
60500.0                  60499.999999999985
61000.0                  61966.666666666664
66000.0                  67499.99999999999
66000.0                  66000.0
69000.0                  67499.99999999999
83800.0                  83399.99999999996
88500.0                  82999.99999999997
After performing LWR Algorithm the % error when tau = 0.01 is 5.41%
```

In [19]:

```
# % Error for different tau values
print("Tau\t\t% Error")
max_iteration=100
req_values=[]
for i in range(1,max_iteration):
    req_values.append(i/100)
for tau in req_values:
    predictionLWR=LocallyWeightedLR(X,Y,tau)
    print(tau,end="\t\t")
    print(round(predict(Y, predictionLWR),10))
```

```
Tau             % Error
0.01            5.4073055366
0.02            5.4073055366
0.03            5.4073055366
0.04            5.4073055366
0.05            5.4073055366
0.06            5.4073055366
0.07            5.4073055366
0.08            5.4073055366
0.09            5.4073055366
0.1             5.4073055366
0.11            5.4073055366
0.12            5.4073055366
0.13            5.4073055366
0.14            5.4073055367
0.15            5.407305539
0.16            5.4073055785
0.17            5.4073059139
```

```
0.18        5.4073058008
0.19        5.4073066095
0.2         5.407309645
0.21        5.4073186611
0.22        5.4073414849
0.23        5.407392172
0.24        5.4074927324
0.25        5.4076749731
0.26        5.4079808234
0.27        5.4084611374
0.28        5.409173895
0.29        5.410181229
0.3         5.4115459215
0.31        5.413327913
0.32        5.4155808926
0.33        5.4183494088
0.34        5.4216669112
0.35        5.4255545161
0.36        5.4300201298
0.37        5.435059262
0.38        5.4406555379
0.39        5.4467822096
0.4         5.4534040641
0.41        5.4604793443
0.42        5.4682072568
0.43        5.4763114554
0.44        5.484734308
0.45        5.4934263462
0.46        5.5023408545
0.47        5.5114354394
0.48        5.5206723165
0.49        5.5300193053
0.5         5.5394501438
0.51        5.5488804049
0.52        5.5585456043
0.53        5.5683928852
0.54        5.5782668421
0.55        5.5881643087
0.56        5.5983436177
0.57        5.6084596915
0.58        5.618876394
0.59        5.6293339353
0.6         5.6398362935
0.61        5.6503873291
0.62        5.6606535508
0.63        5.6713005789
0.64        5.6820043214
0.65        5.6930157002
0.66        5.7045441665
```

```
0.67        5.7161902559
0.68        5.7278762814
0.69        5.7395977227
0.7         5.7513491401
0.71        5.7631241331
0.72        5.7749157665
0.73        5.7867163812
0.74        5.7985179227
0.75        5.81031194
0.76        5.8220897849
0.77        5.8310087408
0.78        5.8364602958
0.79        5.8475458417
0.8         5.8585585535
0.81        5.8694892544
0.82        5.8803309163
0.83        5.8910771241
0.84        5.9021811093
0.85        5.9132002381
0.86        5.9241220304
0.87        5.9349381141
0.88        5.9456442045
0.89        5.9562333727
0.9         5.966701986
0.91        5.9770452325
0.92        5.9874358285
0.93        5.9977469001
0.94        6.0079208464
0.95        6.0179562568
0.96        6.0278507426
0.97        6.0376027211
0.98        6.047211277
0.99        6.0566877695
```

# CONCLUSION

## Result: % error is less when using LWR Algorithm.

### In comparison to (a), (b) and (c) parts, my observation is that the minimum error is in LWR Algorithm.