Machine Learning 2023/2024 (2nd semester)

Master in Electrical and Computer Engineering

Department of Electrical and Computer Engineering

A. Pedro Aguiar (pedro.aguiar@fe.up.pt), **Aníbal Matos** (anibal@fe.up.pt), **Andry Pinto** (amgp@fe.up.pt), **Daniel Campos** (dfcampos@fe.up.pt), **Maria Inês Pereira** (maria.ines@fe.up.pt)

FEUP, Mar. 2024

Project #01

Note: This is to be done in group of **2** elements. Use this notebook to answer all the questions. At the end of the work, you should **upload** both the **notebook** and a **pdf file** with a printout of the notebook with all the results in the **moodle** platform.

Deadlines: Present the state of your work (and answer questions) on the week of **March 18** in your corresponding practical class. Upload the files until 23:59 of **March 27, 2024**.

In [190…

```
# To make a nice pdf file of this file, you have to do the following:
# - upload this file into the running folder (click on the corresponding left icon)
# Then run this (which will make a html file into the current folder):
!jupyter nbconvert --to html "ML_project1.ipynb"
# Then just download the html file and print it to pdf!
```

Identification

- **Group:** Project_A02_B
- **Name:** Diogo Araújo de Oliveira
- **Student Number:** 202007968
- **Name:** Tiago Marques Claro
- **Student Number:** 202006003

Initial setup: To download the file **data-set.cvs**, run the next cell.

In [191…

```
#!wget -O dataset.csv.zip https://www.dropbox.com/s/9y0s2ogjovkwrbm/data-set.csv.zi
#!unzip dataset.csv.zip -d.
```

In [192…

```
# Then, run this code to get the data-set

import pandas as pd
df = pd.read_csv('data-set.csv', index_col=0)
#df.head()
df

# By convention, values that are zero signify no measurements.
# The units are:
# [m] for x and y
# [m/s] for the velocities vx and vy
# [m] for the LIDAR ranges
```

Out[192]:

	time	x	y	vx	vy	angle -179	angle -178	angle -177	angle -176	angle -175	...	angl 17
0	0.0	-3.946339	-2.912177	0.711051	-0.307325	0.0	0.0	0.0	0.0	0.0	...	0.
1	0.1	0.000000	0.000000	0.678366	-0.308563	0.0	0.0	0.0	0.0	0.0	...	0.
2	0.2	0.000000	0.000000	0.677682	-0.285029	0.0	0.0	0.0	0.0	0.0	...	0.
3	0.3	0.000000	0.000000	0.648523	-0.293170	0.0	0.0	0.0	0.0	0.0	...	0.
4	0.4	0.000000	0.000000	0.644965	-0.277222	0.0	0.0	0.0	0.0	0.0	...	0.
...	...	...	...	...	...	...	...	...	...	...	...	
495	49.5	3.855108	-3.928327	-0.078142	-0.093745	0.0	0.0	0.0	0.0	0.0	...	0.
496	49.6	0.000000	0.000000	-0.088140	-0.103430	0.0	0.0	0.0	0.0	0.0	...	0.
497	49.7	0.000000	0.000000	-0.078002	-0.092986	0.0	0.0	0.0	0.0	0.0	...	0.
498	49.8	0.000000	0.000000	-0.076514	-0.091199	0.0	0.0	0.0	0.0	0.0	...	0.
499	49.9	0.000000	0.000000	-0.078499	-0.092891	0.0	0.0	0.0	0.0	0.0	...	0.

500 rows × 365 columns

Part 1: Kalman filter design

Consider a holonomic mobile robot in the 2D plan and suppose that one can get measurements from its linear velocity every time step $t = 0, 0.1, 0.2, \ldots$ (in seconds) and its position every time step $t = 0, 0.5, 1.0, 1.5 \ldots$ (in seconds). Suppose also that the measurements are corrupted by additive Gaussian noise and furthermore, the linear velocity measurements may also include a unknown but constant bias term. The goal is to obtain an estimate of the position of the robot together with a measure of its uncertainty. To this end, we will implement a Kalman filter (KF)!

Model:

Let (x_t, y_t) be the position of the robot at time step t, and $(v_{x,t}, v_{y,t})$ its linear velocity. Let $(b_{x,t}, b_{y,t})$ be the bias term and w_t and η_t Gaussian noises. Then, a state-space model to design the KF can be written as

x-direction

$$\begin{bmatrix} x_{t+1} \\ b_{x,t+1} \end{bmatrix} = \begin{bmatrix} 1 & h \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ b_{x,t} \end{bmatrix} + \begin{bmatrix} h \\ 0 \end{bmatrix} v_{x,t} + w_{x,t} \quad t = 0, 0.1, 0.2, \dots$$

$$z_{x,t} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_t \\ b_{x,t} \end{bmatrix} + \eta_{x,t}, \quad t = 0, 0.5, 1.0, 1.5 \dots$$

y-direction

$$\begin{bmatrix} y_{t+1} \\ b_{y,t+1} \end{bmatrix} = \begin{bmatrix} 1 & h \\ 0 & 1 \end{bmatrix} \begin{bmatrix} y_t \\ b_{y,t} \end{bmatrix} + \begin{bmatrix} h \\ 0 \end{bmatrix} v_{y,t} + w_{y,t} \quad t = 0, 0.1, 0.2, \dots$$

$$z_{y,t} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ b_{y,t} \end{bmatrix} + \eta_{y,t}, \quad t = 0, 0.5, 1.0, 1.5 \dots$$

where $(z_{x,t}, z_{y,t})$ is the output vector and $h = 0.1\ s$ is the sample time.

Note: We have decomposed the model in two decoupled parts (x and y directions). Thus, it is possible to design a KF for each direction.

1.1 Implement 2 KFs (one for each direction) and display the evolution along time of the estimated position of the robot and the estimated bias term. Display also the estimated trajectory 2D.

In [193…

```
import numpy as np
from numpy import *
import matplotlib.pyplot as plt

time = df["time"].values
x = df["x"].values
y = df["y"].values
vx = df["vx"].values
vy = df["vy"].values
```

In [194…

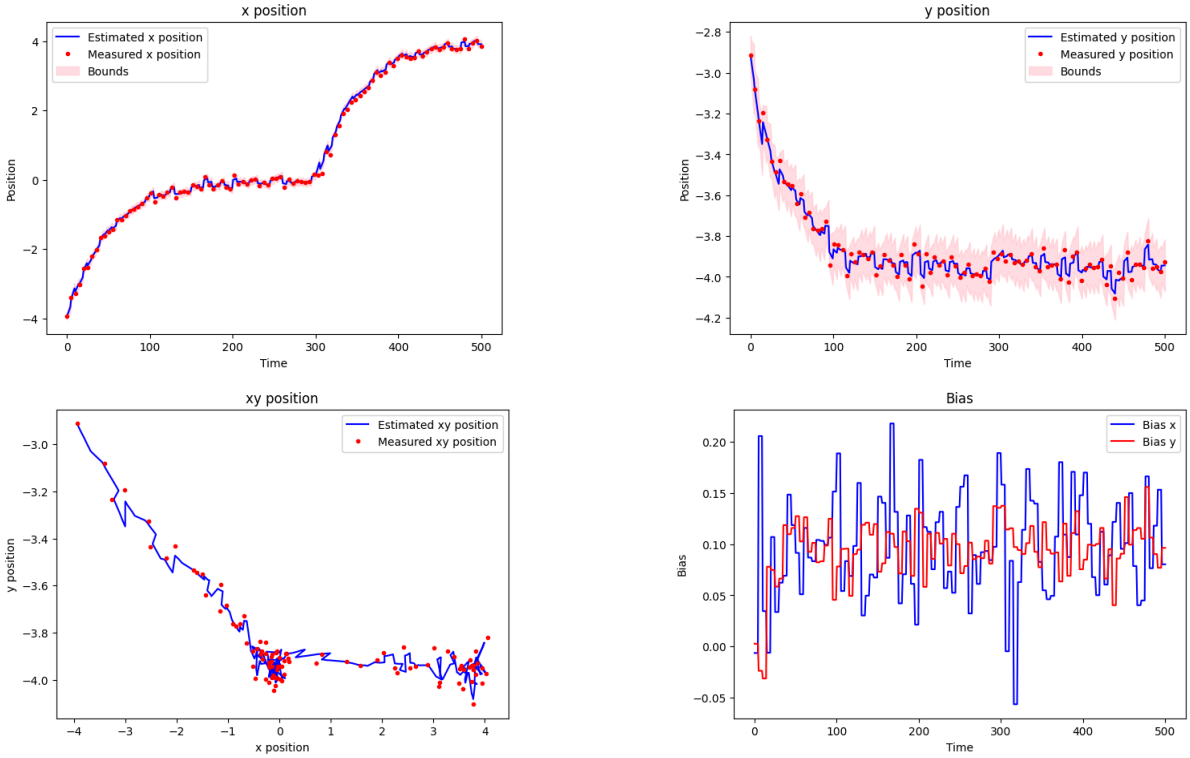
```
# To complete

#
# Kalman Filter loop
#

def kf_predict(X, P, A, Q, B, U):

        X = A @ X + B * U
        P = A @ P @ A.T + Q

        return(X,P)

def kf_update(X_in, P, Y, H, R):

        ek = Y- H @ X_in
        S = H @ P @ H.T + R
        K = P @ H.T @ np.linalg.inv(S)
        X = X_in + K @ ek
        P = P - K @ S @ K.T

        return (X,P)

N_iter = len(time)    # implies dt*N_iter seconds

# sample time
h = 0.1
```

```
# Matrixes for Kalman Filter
A = np.array([[1, h],
                          [0, 1]])

B = np.array([[h],
                          [0]])

H = np.array([[1, 0]])

# Initial Bias
bx0 = 0
by0 = 0

# Bias array
bx = np.zeros((N_iter, 1))
by = np.zeros((N_iter, 1))

# Initial State
x0 = x[0]
y0 = y[0]

# Initial State Vector
X = np.array([[x0],
                          [bx0]])

Y = np.array([[y0],
                          [by0]])

# Initial Covariance Matrix
Px = np.array([[0.1, 0],
                           [0, 0.1]])

Py = Px.copy()

# Process Noise Covariance Matrix
Qx = np.array([[0.001, 0],
                          [0, 0.001]])

Qy = Qx.copy()

# Measurement Noise Covariance Matrix
Rx = np.array([[0.01]])
Ry = Rx.copy()

# Estimation Variables
x_est = np.zeros((len(time), 1))
y_est = x_est.copy()
bx_est = x_est.copy()
by_est = x_est.copy()
vx_est = x_est.copy()
vy_est = x_est.copy()

# Measurement Variables
x_meas = []
y_meas = []

# Bound Variables
x_upperBound = np.zeros((len(time), 1))
x_lowerBound = x_upperBound.copy()
y_upperBound = x_upperBound.copy()
y_lowerBound = x_upperBound.copy()

for t in arange(0, N_iter):
```

```
        # Retrieve the velocity measurements at time t
        Ux = np.array([vx[t]])
        Uy = np.array([vy[t]])

        # Estimate the next state based on the previous state and the process model
        (X, Px) = kf_predict(X, Px, A, Qx, B, Ux)
        (Y, Py) = kf_predict(Y, Py, A, Qy, B, Uy)

        # Store the bias at time t
        bx[t] = X[1][0]
        by[t] = Y[1][0]

        # Store the estimated state
        x_est[t] = X[0][0]
        bx_est[t] = X[1][0]
        y_est[t] = Y[0][0]
        by_est[t] = Y[1][0]

        # Store the estimated velocity
        vx_est[t] = vx[t] + bx_est[t]
        vy_est[t] = vy[t] + by_est[t]

        # Store the bounds
        x_upperBound[t] = x_est[t] + sqrt(Px[0][0])
        x_lowerBound[t] = x_est[t] - sqrt(Px[0][0])
        y_upperBound[t] = y_est[t] + sqrt(Py[0][0])
        y_lowerBound[t] = y_est[t] - sqrt(Py[0][0])


        if t%5 == 0:

                # Retrive the measurements at time t
                Zx = np.array([[x[t]]])
                Zy = np.array([[y[t]]])

                # Update the state based on the measurement
                (X, Px) = kf_update(X, Px, Zx, H, Rx)
                (Y, Py) = kf_update(Y, Py, Zy, H, Ry)

                # Store the bias
                bx[t] = X[1][0]
                by[t] = Y[1][0]

                # Store the measurements
                x_est[t] = X[0][0]
                bx_est[t] = X[1][0]
                y_est[t] = Y[0][0]
                by_est[t] = Y[1][0]

                # Store the estimated velocity
                vx_est[t] = vx[t] + bx_est[t]
                vy_est[t] = vy[t] + by_est[t]

                # Store the bounds
                x_upperBound[t] = x_est[t] + sqrt(Px[0][0])
                x_lowerBound[t] = x_est[t] - sqrt(Px[0][0])
                y_upperBound[t] = y_est[t] + sqrt(Py[0][0])
                y_lowerBound[t] = y_est[t] - sqrt(Py[0][0])

                # Store the measurements
                x_meas.append(x[t])
                y_meas.append(y[t])
```

```
# Plot the results

t = np.linspace(0, N_iter, N_iter)
t_meas = np.linspace(0, N_iter, len(x_meas))

fig = plt.figure(figsize=(40, 5))
fig.subplots_adjust(hspace=0.5, wspace=0.5)
ax1 = fig.add_subplot(1, 4, 1)
ax2 = fig.add_subplot(1, 4, 2)


# Plot the x position
ax1.plot(t, x_est, 'b-', label = 'Estimated x position')
ax1.plot(t_meas, x_meas, 'r.', label = 'Measured x position')
ax1.fill_between(t, x_upperBound[:,0], x_lowerBound[:,0], color = 'pink', alpha = 0
ax1.set_title('x position')
ax1.set_xlabel('Time')
ax1.set_ylabel('Position')
ax1.legend()

# Plot the y position
ax2.plot(t, y_est, 'b-', label = 'Estimated y position')
ax2.plot(t_meas, y_meas, 'r.', label = 'Measured y position')
ax2.fill_between(t, y_upperBound[:,0], y_lowerBound[:,0], color = 'pink', alpha = 0
ax2.set_title('y position')
ax2.set_xlabel('Time')
ax2.set_ylabel('Position')
ax2.legend()

# Figure 2
fig2 = plt.figure(figsize=(40, 5))
fig2.subplots_adjust(hspace=0.5, wspace=0.5)
ax3 = fig2.add_subplot(1, 4, 1)
ax4 = fig2.add_subplot(1, 4, 2)

# Plot the xy position
ax3.plot(x_est, y_est, 'b-', label = 'Estimated xy position')
ax3.plot(x_meas, y_meas, 'r.', label = 'Measured xy position')
ax3.set_title('xy position')
ax3.set_xlabel('x position')
ax3.set_ylabel('y position')
ax3.legend()

# Plot bias
ax4.plot(t, bx, 'b-', label = 'Bias x')
ax4.plot(t, by, 'r-', label = 'Bias y')
ax4.set_title('Bias')
ax4.set_xlabel('Time')
ax4.set_ylabel('Bias')
ax4.legend()

# End For Loop
```

Out[194]: <matplotlib.legend.Legend at 0x22ce51cb5e0>



Part 2: Linear Regression

In this part, the aim is to build a map of the environment by combining the position of the robot with the measurements of the 2D **LIDAR** that is on-board of the robot. The LIDAR measurements consist of range (distance) from the robot to a possible obstacle for each degree of direction, that is,

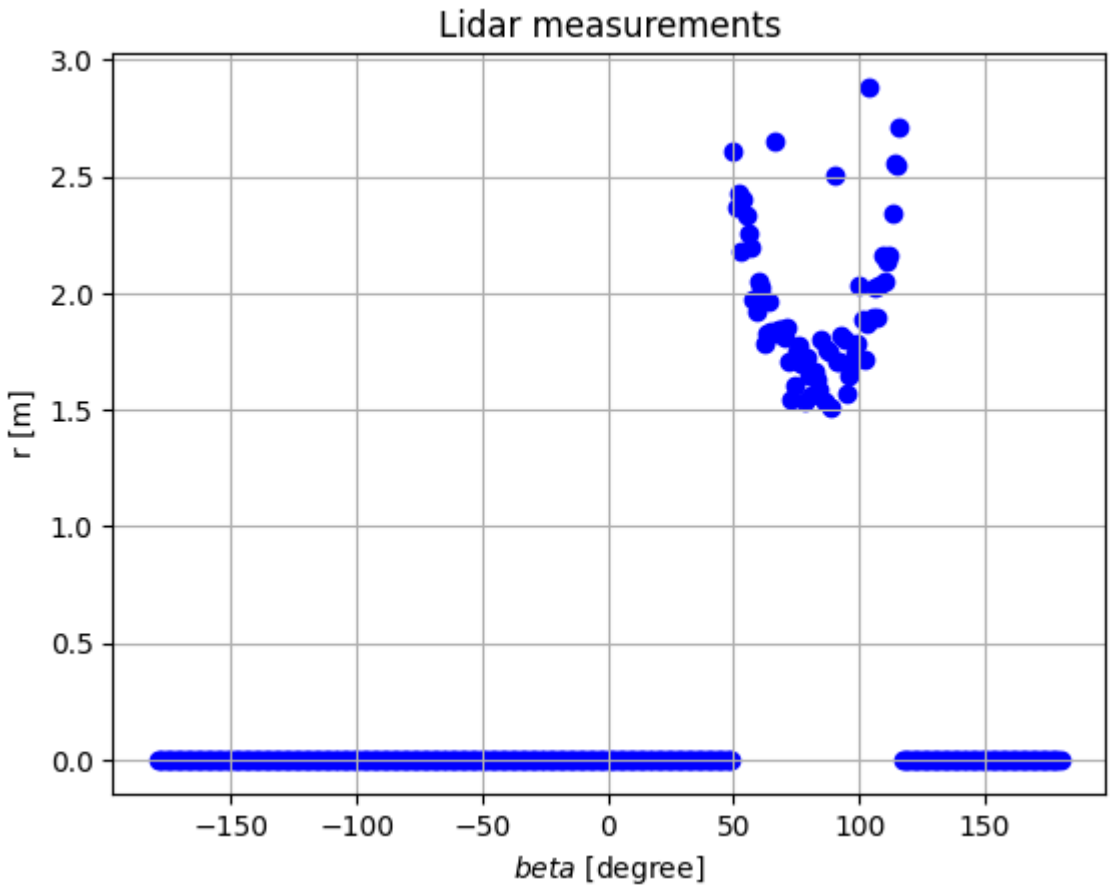
$$r_t = \{r_\beta + \eta_r : \beta = -179^o, -178^o, \ldots, 0^o, \ldots, 180^o\}$$

where η_r is assumed to be Gaussian noise. The sample time is the same, that is, $h = 0.1\,s$, but the LIDAR measurements are outputted every time step $t = 0, 0.5, 1.0, 1.5, \ldots$ (in seconds) like the robot position in the previous exercise. Moreover, if there is no obstacle within the direction of the laser range or if it is far away, that is, if the distance is greater than $5\,m$, by convention the range measurement is set to zero. It may also happen that the LIDAR in some cases may output an *outlier*.

The next figure shows r_t as a function of the angle β for $t = 5.0\,s$.

In [195…

```
time = df["time"].values
Lidar_range = df.iloc[:, np.arange(5,365,1)].values

t=5*10 # t = 5 sec * 1/sample_time
angle = np.linspace(-179, 180, num=360)

plt.figure()
plt.scatter(angle, Lidar_range[t], color='b')
plt.title('Lidar measurements')
plt.ylabel('r [m]')
plt.xlabel('$beta$ [degree]')
plt.grid()
```



2.1 Using the estimated position of the robot (computed in the previous exercise) and the LIDAR data,

1. Obtain the cloud points in the 2D plan that the robot sense at $t = 5\,s$ and plot them. Do not forget to remove the zero ranges and note that

$$\hat{x}_{o,t} = \hat{x}_t + r_t \cos\beta$$
$$\hat{y}_{o,t} = \hat{y}_t + r_t \sin\beta$$

1. Perform a linear regression for the previous data using a model of the type

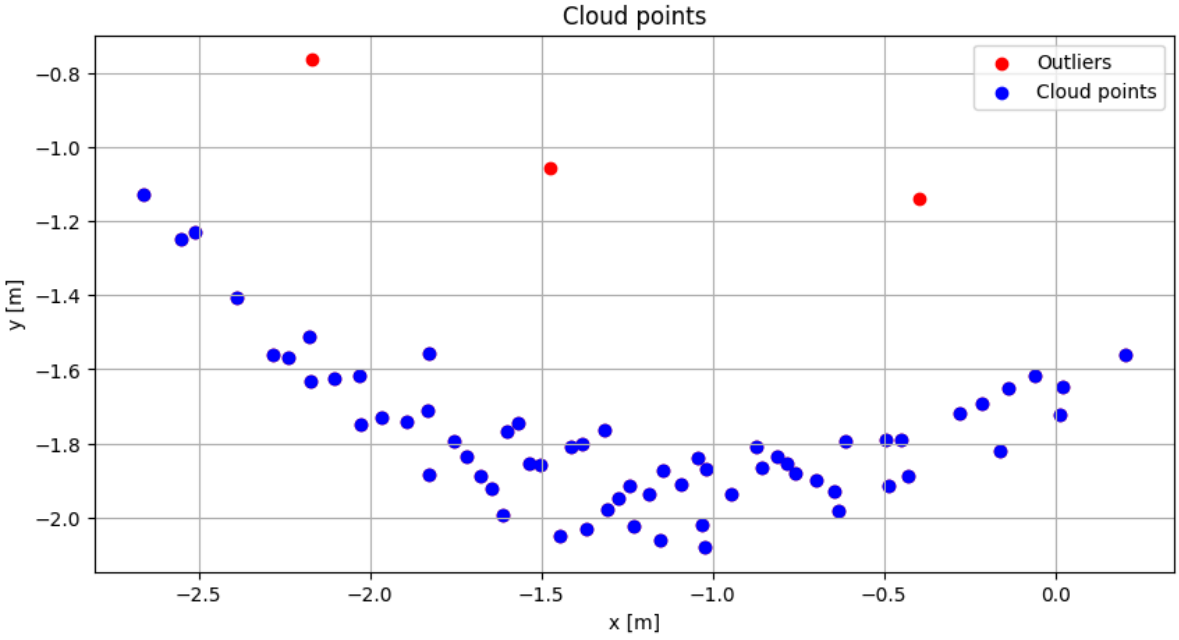
$$y = \theta_0 + \theta_1 x \tag{1}$$

and display the results, that is, display the resulting 2d map, the mean square error, and the optimal parameters for θ. To this end, apply the related Least Square (LS) normal equations and **only use** the sklearn to confirm the obtained values.

In [196…

```
# Part 2.1.1

# function to convert the angle from degrees to radians
def deg2rad(angle):
        return angle*pi/180

# function to append the data of the Lidar measurements to the x and y position
def apendData():
        x_o = []
        y_o = []
        x_NoOutlier = []
        y_NoOutlier = []

        t=5*10 # t = 5 sec * 1/sample_time
```

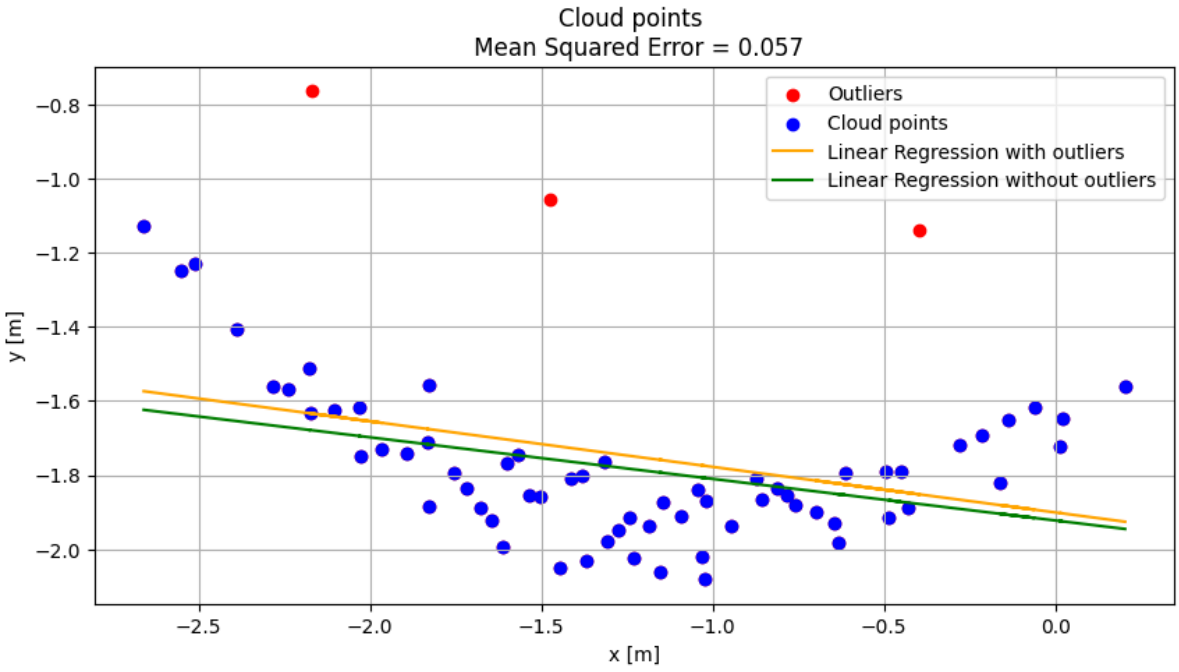
```
        for i in range(len(Lidar_range[t])):

                # Get all the point cloud
                if Lidar_range[t][i] > 0:

                        angle = i - 179
                        Position_X = x_est[t] + Lidar_range[t][i]*cos(deg2rad(angle
                        Position_Y = y_est[t] + Lidar_range[t][i]*sin(deg2rad(angle
                        x_o.append(Position_X)
                        y_o.append(Position_Y)
                        x_NoOutlier.append(Position_X)
                        y_NoOutlier.append(Position_Y)

                        #Remove the outliers
                        if i > 0 and i < len(Lidar_range[t])-1:
                                if(Lidar_range[t][i-1] * Lidar_range[t][i] * Lidar_
                                        if np.abs(Lidar_range[t][i]-Lidar_range[t][
                                                x_NoOutlier.remove(Position_X)
                                                y_NoOutlier.remove(Position_Y)


        return x_o, y_o, x_NoOutlier, y_NoOutlier

# Get the data
x_o, y_o, x_NoOutlier, y_NoOutlier = apendData()

# Plot the results
figure = plt.figure(figsize=(10, 5))
plt.scatter(x_o, y_o, color='r', label='Outliers')
plt.scatter(x_NoOutlier, y_NoOutlier, color='b', label='Cloud points')
plt.title('Cloud points')
plt.ylabel('y [m]')
plt.xlabel('x [m]')
plt.legend()
plt.grid()
plt.show()
```



In [197…

```
# Part 2.1.2
from sklearn.metrics import mean_squared_error

# Get the data
x_o, y_o, x_NoOutlier, y_NoOutlier = apendData()
```

```
# Create X matrix with bias
X_o = np.reshape(x_o.copy(),(len(x_o.copy()),1))
ones = np.ones((len(x_o.copy()), 1))

# Create X matrix without outliers
X_NoOutlier = np.reshape(x_NoOutlier.copy(),(len(x_NoOutlier.copy()),1))
ones_NoOutlier = np.ones((len(x_NoOutlier.copy()), 1))

# Create Y matrix
X = np.concatenate((ones, X_o), axis = 1)
X_NoOutlier_x = np.concatenate((ones_NoOutlier, X_NoOutlier), axis = 1)
Y = np.reshape(y_o,(len(y_o),1))
y_NoOutlier = np.reshape(y_NoOutlier,(len(y_NoOutlier),1))

# Compute the theta
theta1 = np.linalg.inv(X.T @ X) @ X.T @ Y
theta2 = np.linalg.inv(X_NoOutlier_x.T @ X_NoOutlier_x) @ X_NoOutlier_x.T @ y_NoOut
Y_predict1 = X @ theta1
Y_predict2 = X_NoOutlier_x @ theta2

# Compute the mean squared error
MSE = mean_squared_error(Y, Y_predict1)

# Plot the results
figure = plt.figure(figsize=(10, 5))
plt.scatter(x_o, y_o, color='r', label='Outliers')
plt.scatter(x_NoOutlier, y_NoOutlier, color='b', label='Cloud points')
plt.plot(X_o, X @ theta1, color='orange', label='Linear Regression with outliers')
plt.plot(X_NoOutlier, X_NoOutlier_x @ theta2, color='g', label='Linear Regression w
plt.title("Cloud points \n Mean Squared Error = %.3f" % MSE)
plt.ylabel('y [m]')
plt.xlabel('x [m]')
plt.legend()
plt.grid()
plt.show()
```

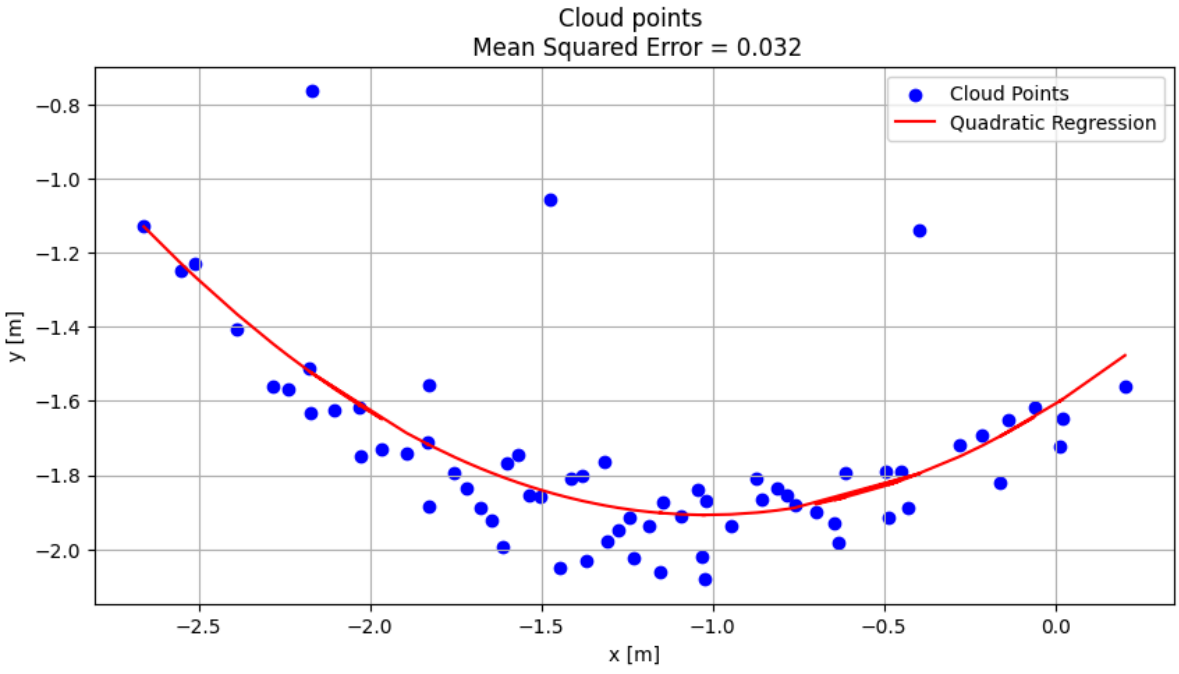


2.2 Repeat the previous exercise but now with a polynomial model of the type

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 \tag{2}$$

In [198…

```
X_pol = []

# Create the matrix X with bias
X_pol = np.concatenate((X.copy(), X_o.copy()**2), axis=1)

# Compute the theta
theta = np.linalg.inv(X_pol.T @ X_pol) @ X_pol.T @ Y

# Compute the prediction
Y_pol_predict = X_pol @ theta

# Compute the mean squared error
MSE_pol = mean_squared_error(Y, Y_pol_predict)

# Plot the results
figure = plt.figure(figsize=(10, 5))
plt.scatter(x_o, y_o, color='b', label='Cloud Points')
plt.plot(X_o, X_pol @ theta, color='r', label='Quadratic Regression')
plt.title("Cloud points \n Mean Squared Error = %.3f" % MSE_pol)
plt.ylabel('y [m]')
plt.xlabel('x [m]')
plt.legend()
plt.grid()
plt.show()
```



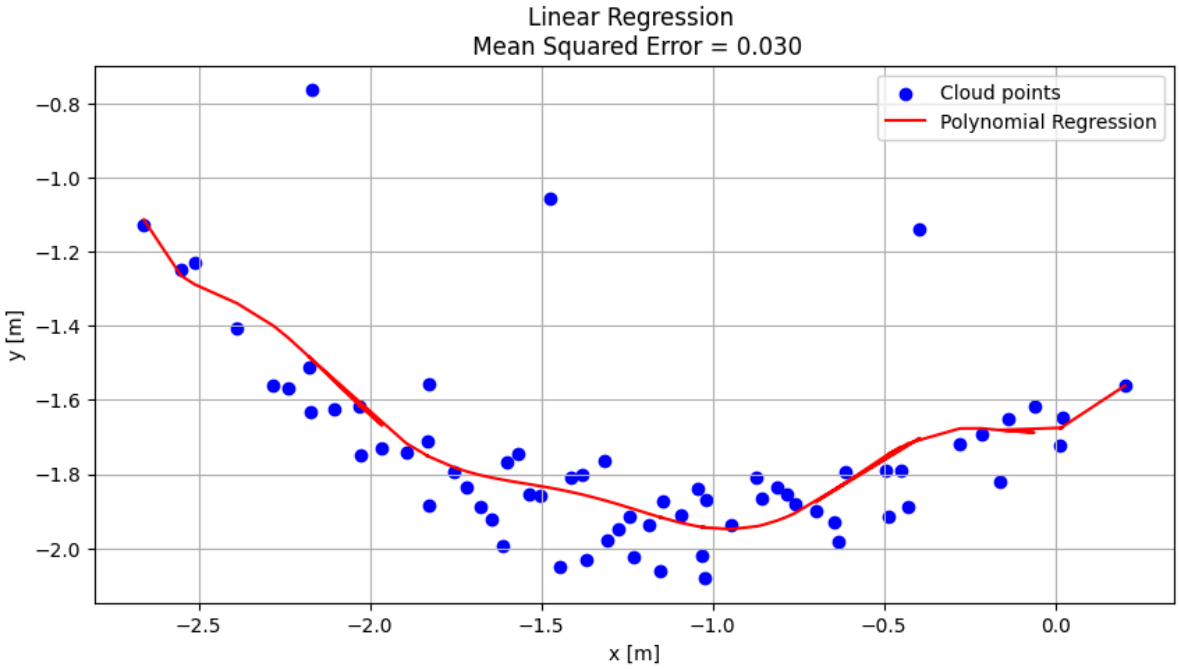
2.3 At this point you can use sklearn! Do the same as the previous exercise (polynomial model) but now with **degree 10**. Moreover, implement also a regression with **Ridge** regularization and a regression with **LASSO** regularization. Do not forget to display the obtained results. What can you conclude?

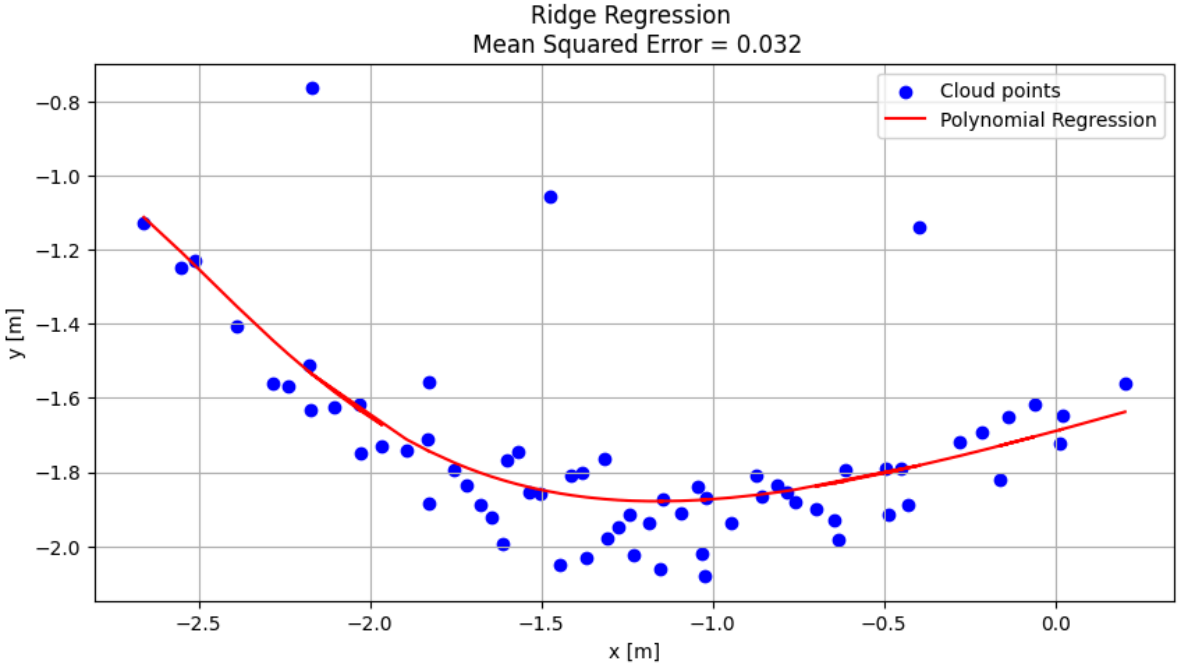
In [199…

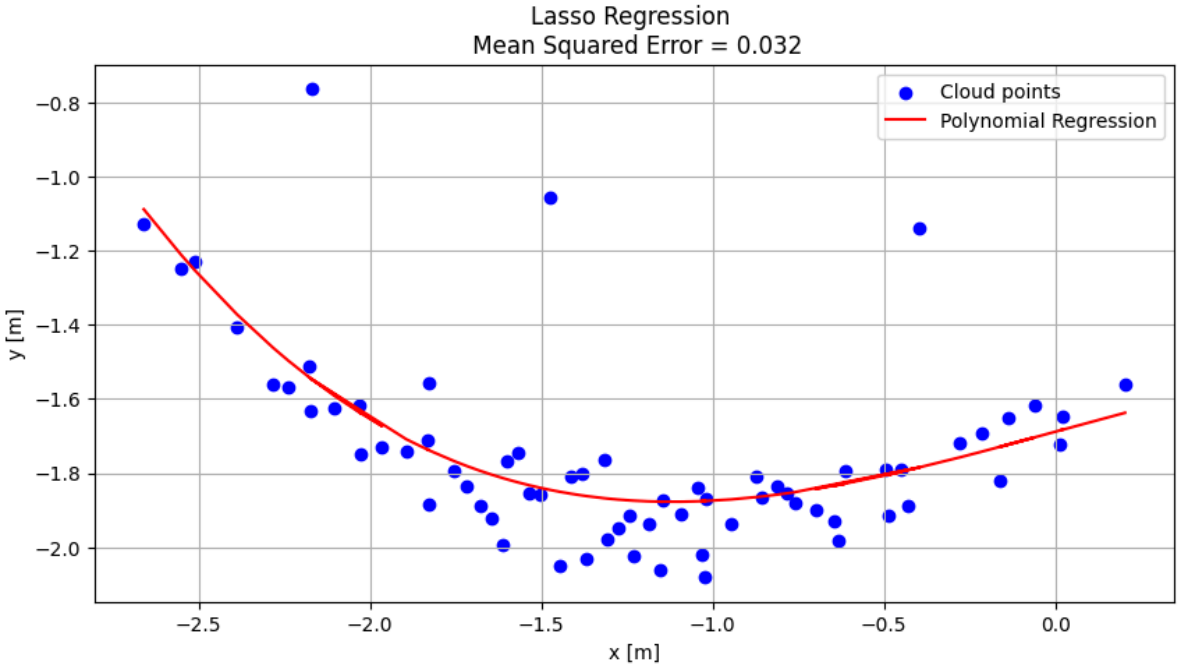
```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import MinMaxScaler

X_reg = []
X_reg = X_pol.copy()

# Create the matrix X with bias degree 10
for i in range(3, 11):
        X_reg = np.concatenate((X_reg.copy(), X_o**i), axis=1)
```

```
# normalize the data
scalerX = MinMaxScaler()
scaler_y = MinMaxScaler()
X_reg = scalerX.fit_transform(X_reg)
Y_norm_orig = scaler_y.fit_transform(Y)


#####################################################
################ LinearRegression ################
#####################################################


# Create the model
model_linear = LinearRegression(fit_intercept=True)

# Fit the model
model_linear.fit(X_reg, Y)
ylinear_fit = scaler_y.inverse_transform(model_linear.predict(X_reg))

#Calculate the mean squared error
MSE_linear = mean_squared_error(Y, model_linear.predict(X_reg))

# Plot the results
figure = plt.figure(figsize=(10, 5))
plt.scatter(x_o, y_o, color='b', label='Cloud points')
plt.plot(X_o, model_linear.predict(X_reg), color='r', label='Polynomial Regression'
plt.title("Linear Regression \n Mean Squared Error = %.3f" % MSE_linear)
plt.ylabel('y [m]')
plt.xlabel('x [m]')
plt.legend()
plt.grid()
plt.show()


#####################################################
################ Ridge Regression ################
#####################################################

from sklearn.linear_model import Ridge

# Create the model
model_ridge = Ridge(alpha=0.1, fit_intercept=True)

# Fit the model
model_ridge.fit(X_reg, Y)

# Calculate the mean squared error
MSE_ridge = mean_squared_error(Y, model_ridge.predict(X_reg))

# Plot the results
figure1 = plt.figure(figsize=(10, 5))
plt.scatter(x_o, y_o, color='b', label='Cloud points')
plt.plot(X_o, model_ridge.predict(X_reg), color='r', label='Polynomial Regression')
plt.title('Ridge Regression \n Mean Squared Error = %.3f' % MSE_ridge)
plt.ylabel('y [m]')
plt.xlabel('x [m]')
plt.legend()
plt.grid()
plt.show()

#####################################################
################ Lasso Regression ################
#####################################################
```

```
from sklearn.linear_model import Lasso

# Create the model
model_lasso = Lasso(alpha=0.001, fit_intercept=True)

# Fit the model
model_lasso.fit(X_reg, Y)

# Calculate the mean squared error
MSE_lasso = mean_squared_error(Y, model_lasso.predict(X_reg))

# Plot the results
figure2 = plt.figure(figsize=(10, 5))
plt.scatter(x_o, y_o, color='b', label='Cloud points')
plt.plot(X_o, model_lasso.predict(X_reg), color='r', label='Polynomial Regression')
plt.title('Lasso Regression \n Mean Squared Error = %.3f' % MSE_lasso)
plt.ylabel('y [m]')
plt.xlabel('x [m]')
plt.legend()
plt.grid()
plt.show()
```







2.4 We now would like to use all the LIDAR data. One simple option (off-line) is to make a data set with all the cloud point positions in 2D and apply the linear regression techniques.

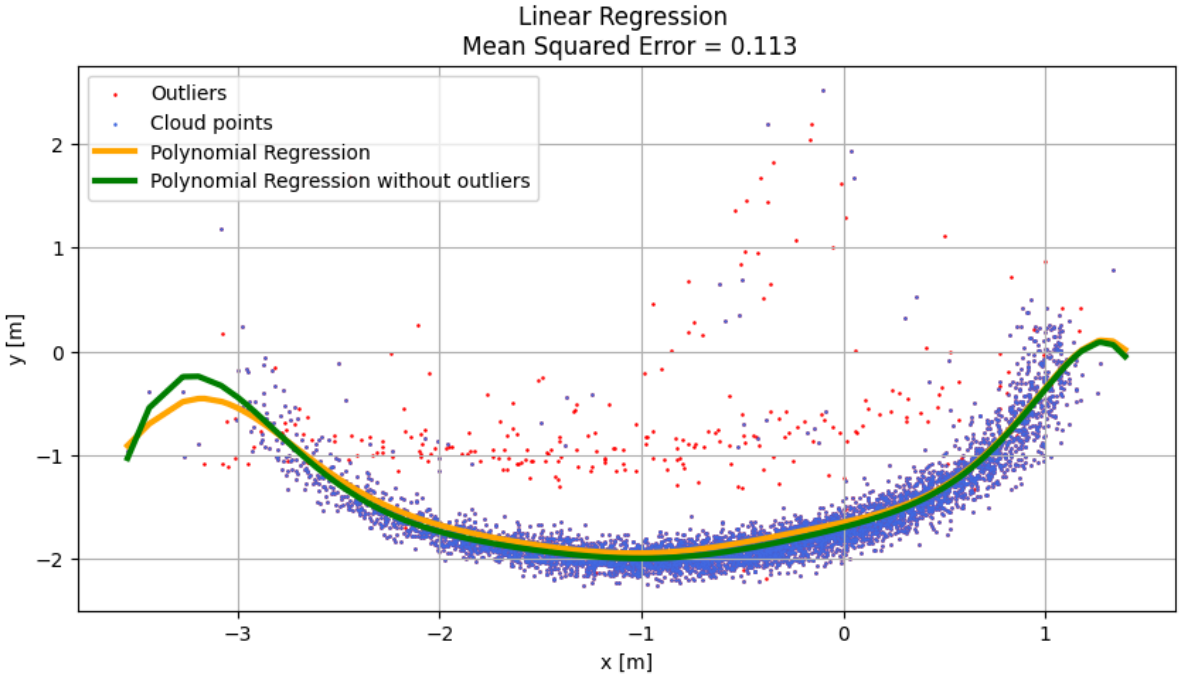
Using sklearn, do this for LS, LS+Ridge, LS+LASSO using the polynomial model of degree 10. Display the results (map 2D) and the optimal values for θ.

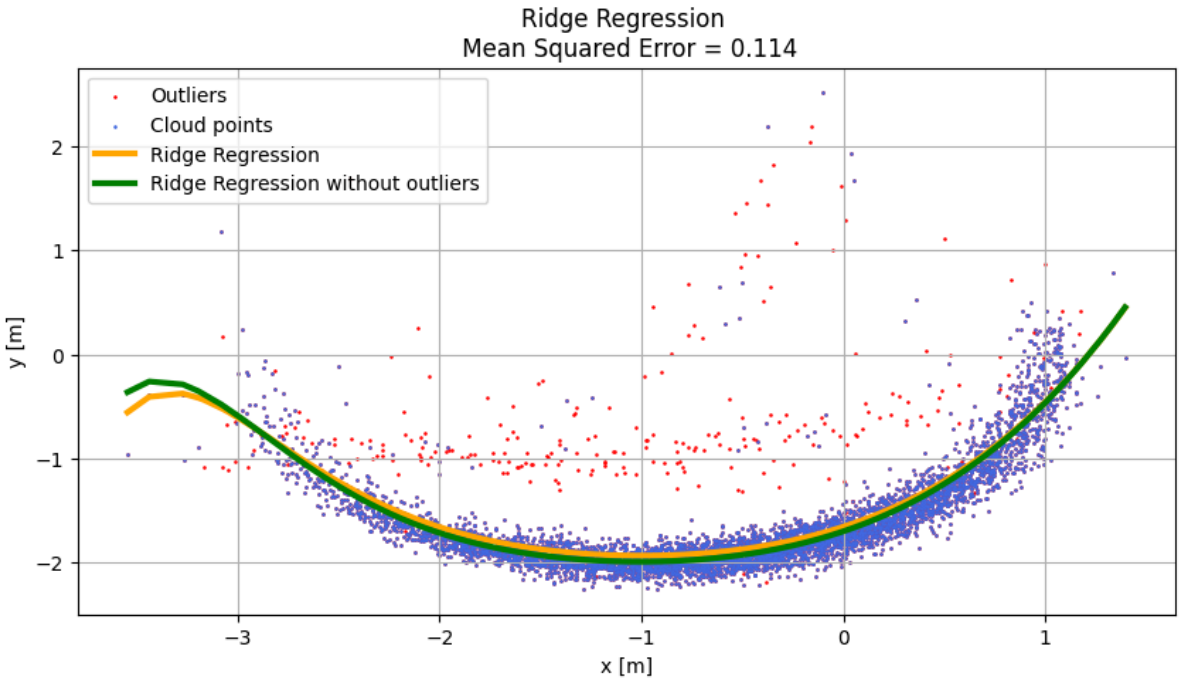
In [200…

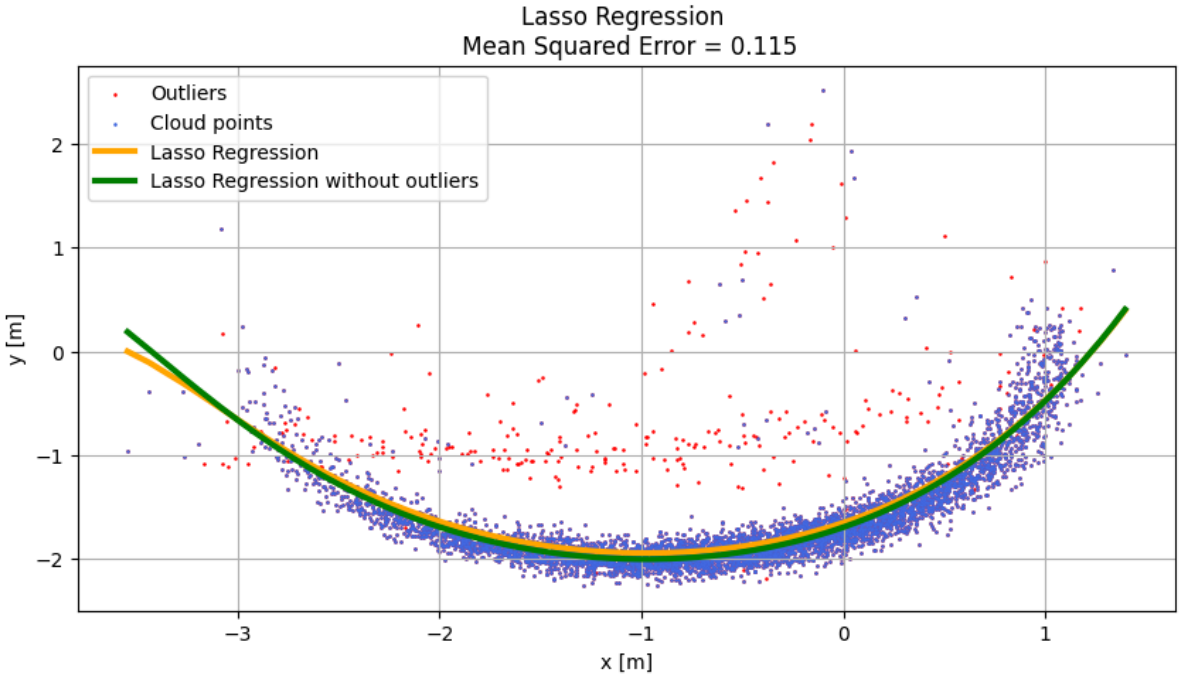
```
import numpy as np
import sklearn
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.preprocessing import MinMaxScaler


#Now using all the LIDAR data
x_all, y_all = [], []
x_noOut, y_noOut = [], []
x_t, y_t = [], []
x_data, y_data = [], []

t=5*10 # t = 5 sec * 1/sample_time

for t in range(0, time.size, 5): #Read every 0.5s from the 500 samples (each sample
        x_t, y_t = [], []
        for i in range(len(Lidar_range[t])):
                if Lidar_range[t][i] > 0:
                        angle = i - 179
                        Position_X = x_est[t] + Lidar_range[t][i]*cos(deg2rad(angle
                        Position_Y = y_est[t] + Lidar_range[t][i]*sin(deg2rad(angle
                        x_all.append(Position_X)
                        y_all.append(Position_Y)
                        x_noOut.append(Position_X)
                        y_noOut.append(Position_Y)
                        x_t.append(Position_X)
                        y_t.append(Position_Y)

                        #Remove the outliers
                        if i > 0 and i < len(Lidar_range[t])-1:
                                if(Lidar_range[t][i-1] * Lidar_range[t][i] * Lidar_
                                        if np.abs(Lidar_range[t][i]-Lidar_range[t][
                                                x_noOut.remove(Position_X)
```

```
                                                y_noOut.remove(Position_Y)
        x_data.append(x_t)
        y_data.append(y_t)


# Order the x in ascending order and get the corresponding y
x_all, y_all = zip(*sorted(zip(x_all, y_all)))
x_noOut, y_noOut = zip(*sorted(zip(x_noOut, y_noOut)))

# Create X matrix with bias
X_all = np.reshape(x_all,(len(x_all),1))
ones = np.ones((len(x_all), 1))

# Create X matrix without outliers
X_noOut = np.reshape(x_noOut,(len(x_noOut),1))
ones_noOut = np.ones((len(x_noOut), 1))

# Create Y matrix
X_full = np.concatenate((ones, X_all), axis = 1)
Y_full = np.reshape(y_all,(len(y_all),1))

# Create Y matrix without outliers
X_noOutlier = np.concatenate((ones_noOut, X_noOut), axis = 1)
Y_noOutlier = np.reshape(y_noOut,(len(y_noOut),1))

# Create the matrix X with bias degree 10
for i in range(2, 11):
        X_full = np.concatenate((X_full, X_all**i), axis=1)

# Create the matrix X with bias degree 10
for i in range(2, 11):
        X_noOutlier = np.concatenate((X_noOutlier, X_noOut**i), axis=1)

# normalize the data
scalerX = MinMaxScaler()
X_full = scalerX.fit_transform(X_full)
scalerX_noOut = MinMaxScaler()
X_noOutlier = scalerX_noOut.fit_transform(X_noOutlier)

#####################################################
############### Quadratic  Regression #############
#####################################################

# Create the model
model_linear = LinearRegression(fit_intercept=True)
model_linear_noOut = LinearRegression(fit_intercept=True)

# Fit the model
model_linear.fit(X_full, Y_full)
model_linear_noOut.fit(X_noOutlier, Y_noOutlier)

# Calculate the mean squared error
MSE_linear = mean_squared_error(Y_full, model_linear.predict(X_full))

# Plot the results
figure2 = plt.figure(figsize=(10, 5))
plt.scatter(x_all, y_all, color='r', label='Outliers', s=0.7)
plt.scatter(x_noOut, y_noOut, color='royalblue', label='Cloud points', s=0.7)
plt.plot(X_all, model_linear.predict(X_full), color='orange', label='Polynomial Reg
plt.plot(X_noOut, model_linear_noOut.predict(X_noOutlier), color='g', label='Polync
plt.title('Linear Regression \n Mean Squared Error = %.3f' % MSE_linear)
plt.ylabel('y [m]')
plt.xlabel('x [m]')
plt.legend()
```

```
plt.grid()
plt.show()

#####################################################
################ Ridge Regression ################
#####################################################

# Create the model
model_ridge = Ridge(alpha=0.1, fit_intercept=True)
model_ridge_noOut = Ridge(alpha=0.1, fit_intercept=True)

# Fit the model
model_ridge.fit(X_full, Y_full)
model_ridge_noOut.fit(X_noOutlier, Y_noOutlier)

# Calculate the mean squared error
MSE_ridge = mean_squared_error(Y_full, model_ridge.predict(X_full))

# Plot the results
figure3 = plt.figure(figsize=(10, 5))
plt.scatter(x_all, y_all, color='r', label='Outliers', s=0.7)
plt.scatter(x_noOut, y_noOut, color='royalblue', label='Cloud points', s=0.7)
plt.plot(X_all, model_ridge.predict(X_full), color='orange', label='Ridge Regressio
plt.plot(X_noOut, model_ridge_noOut.predict(X_noOutlier), color='g', label='Ridge R
plt.title('Ridge Regression \n Mean Squared Error = %.3f' % MSE_ridge)
plt.ylabel('y [m]')
plt.xlabel('x [m]')
plt.legend()
plt.grid()
plt.show()

#####################################################
################ Lasso Regression ################
#####################################################

# Create the model
model_lasso = Lasso(alpha=0.0001, fit_intercept=True)
model_lasso_noOut = Lasso(alpha=0.0001, fit_intercept=True)

# Fit the model
model_lasso.fit(X_full, Y_full)
model_lasso_noOut.fit(X_noOutlier, Y_noOutlier)

# Calculate the mean squared error
MSE_lasso = mean_squared_error(Y_full, model_lasso.predict(X_full))

# Plot the results
figure4 = plt.figure(figsize=(10, 5))
plt.scatter(x_all, y_all, color='r', label='Outliers', s=0.7)
plt.scatter(x_noOut, y_noOut, color='royalblue', label='Cloud points', s=0.7)
plt.plot(X_all, model_lasso.predict(X_full), color='orange', label='Lasso Regressio
plt.plot(X_noOut, model_lasso_noOut.predict(X_noOutlier), color='g', label='Lasso R
plt.title('Lasso Regression \n Mean Squared Error = %.3f' % MSE_lasso)
plt.ylabel('y [m]')
plt.xlabel('x [m]')
plt.legend()
plt.grid()
plt.show()
```


Linear Regression
Mean Squared Error = 0.113
Outliers
Cloud points
Polynomial Regression
Polynomial Regression without outliers
y [m]
x [m]


Ridge Regression
Mean Squared Error = 0.114
Outliers
Cloud points
Ridge Regression
Ridge Regression without outliers
y [m]
x [m]



2.5 (Extra) Another option (on-line) is to make a linear regression with only the LIDAR data that is being acquired at each snapshot of time $t = 0, 0.5, 1.0, \ldots$ and update the optimal value θ using a gradient descent rule

$$\theta_{t+1} = \theta_t - \gamma \nabla J(\theta_t),$$

where $\gamma > 0$ is the learning rate, and $\nabla J(\theta_t)$ is the gradient at each snapshot of the cost

$$J(\theta) = \sum_{n=1}^{N} \left(y_n - \theta^T \phi(x_n)\right)^2$$

where N is the number of valid (that is non zero) range measurements at instant t.

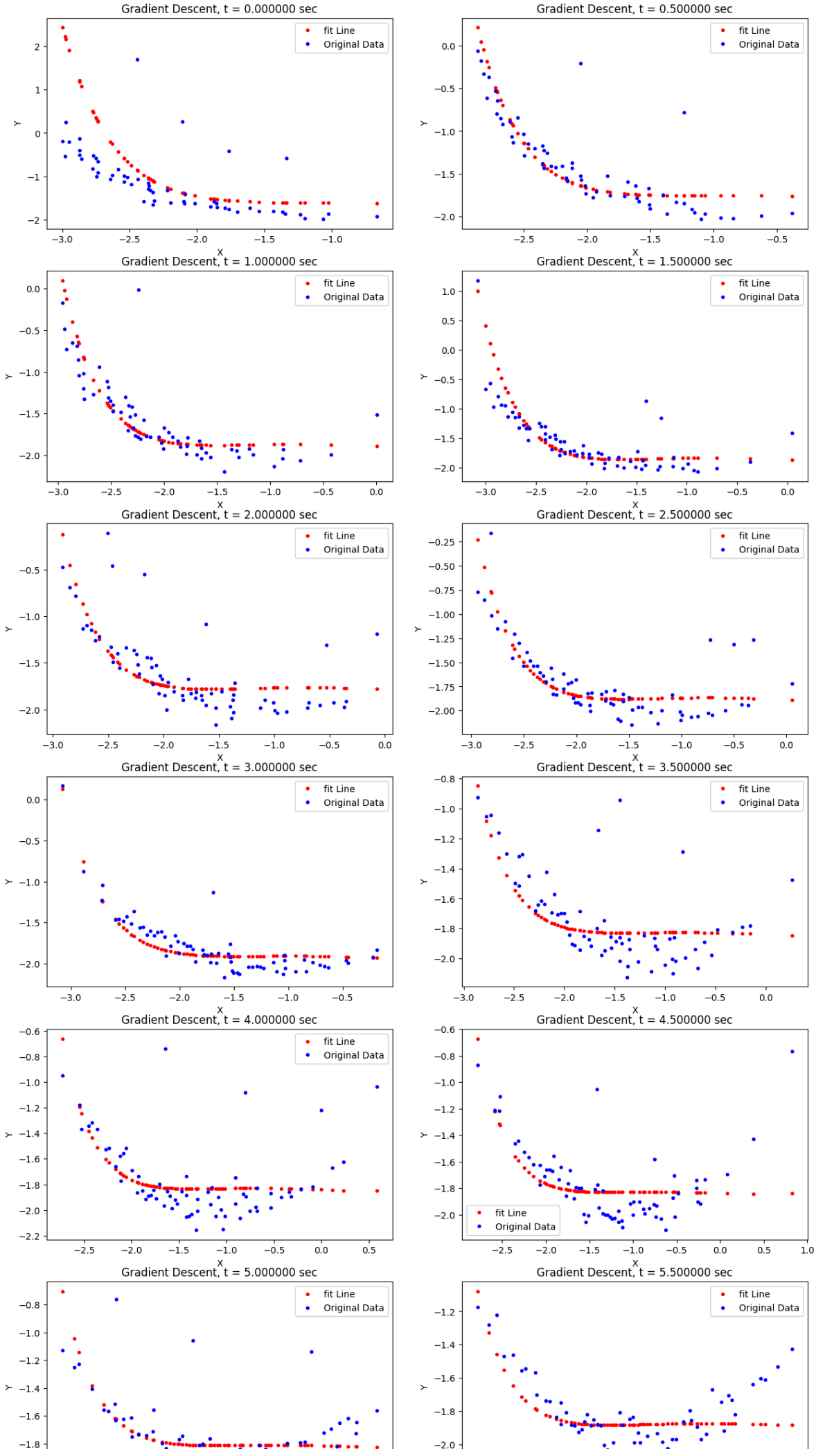
Implement this strategy and plot the results.

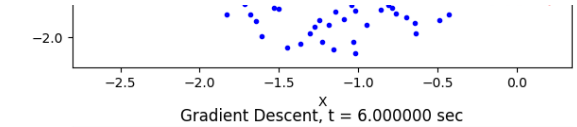
Note: This question is optional. If you solve it, you get extra 15 points (in 100).

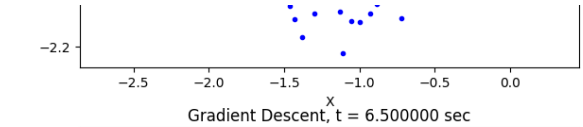
In [203…

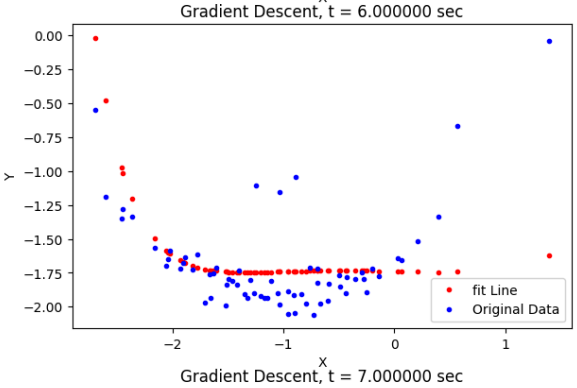
```
# Gradient Descent Method

learning_rate = 0.001
N_iter = 500

# Initialize prediction and time values
predictedValues = []
timeValues = []
X_data = []
Y_data = []
X = []
Y = []

polynomialDegree = 10

# Initialize theta
theta = np.random.rand(polynomialDegree+1, 1)

for t in range(len(x_data)):

        x_data[t] = np.reshape(x_data[t], (len(x_data[t]),1))
```

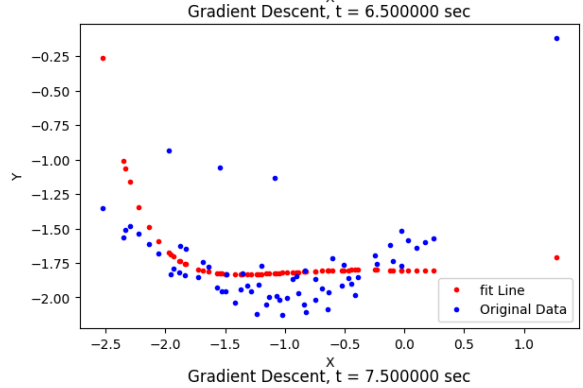
```
        y_data[t] = np.reshape(y_data[t], (len(y_data[t]),1))

        # Create polynomial features up to degree 10
        X_data = np.ones((len(x_data[t]), 1), dtype=float)
        for degree in range(1, polynomialDegree+1):
                X_data = np.concatenate((X_data, x_data[t]**degree), axis=1)

        Y_data = y_data[t]

        # Normalize the data
        x_scaler = MinMaxScaler()
        y_scaler = MinMaxScaler()
        X = x_scaler.fit_transform(X_data)
        Y = y_scaler.fit_transform(Y_data)

        # Gradient descent loop
        for i in range(N_iter):
                Y_pred = X @ theta
                residuals = np.subtract(Y_pred, Y)

                # Gradient calculation
                gradient = 2 * X.T @ residuals / X.shape[0]
                gradient = np.reshape(gradient, (X.shape[1], 1))

                theta = theta - learning_rate * gradient

        Y_pred = y_scaler.inverse_transform(X.dot(theta)) # Reverse transformation
        predictedValues.append(Y_pred)
        timeValues.append(x_data[t])

ncols = 2
nlines = (int)(len(timeValues)/ncols)
fig, ax = plt.subplots(nlines, ncols, figsize=(15,5*nlines))

for l in range(nlines):
        for c in range(ncols):
                t = l*ncols + c
                ax[l][c].plot(timeValues[t], predictedValues[t], 'r.', label = 'fit
                ax[l][c].plot(x_data[t], y_data[t], 'b.', label = 'Original Data')
                ax[l][c].set_title("Gradient Descent, t = %f sec" % (t*0.5))
                ax[l][c].set_xlabel("X")
                ax[l][c].set_ylabel("Y")
                ax[l][c].legend(loc = 'best')
plt.show()
```

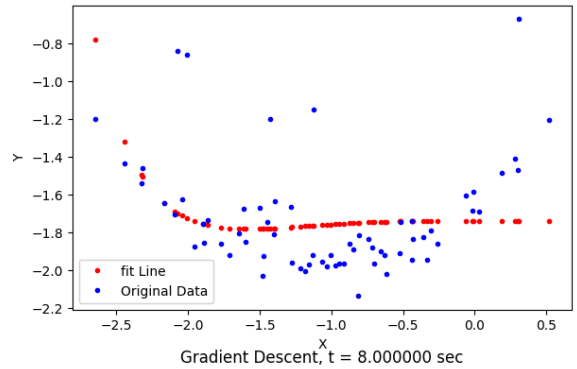

Gradient Descent, t = 0.000000 sec
Gradient Descent, t = 0.500000 sec
Gradient Descent, t = 1.000000 sec
Gradient Descent, t = 1.500000 sec
Gradient Descent, t = 2.000000 sec
Gradient Descent, t = 2.500000 sec
Gradient Descent, t = 3.000000 sec
Gradient Descent, t = 3.500000 sec
Gradient Descent, t = 4.000000 sec
Gradient Descent, t = 4.500000 sec
Gradient Descent, t = 5.000000 sec
Gradient Descent, t = 5.500000 sec
fit Line
Original Data
X
Y

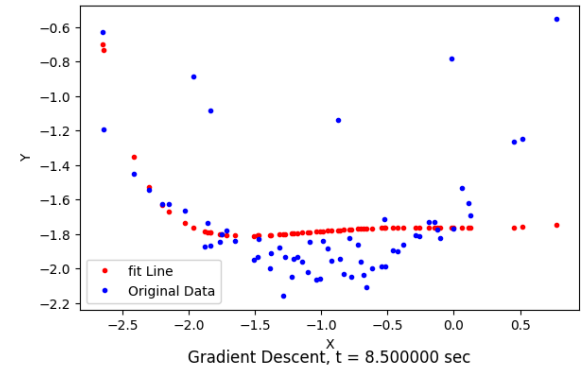


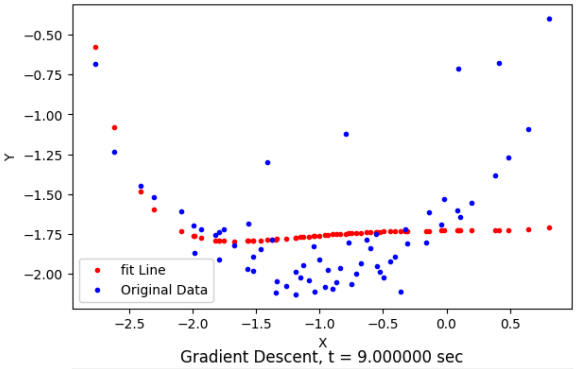


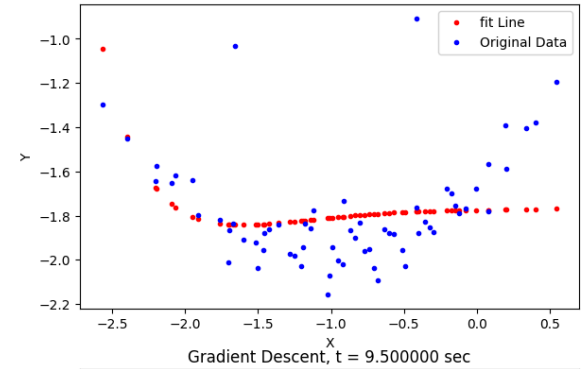

Gradient Descent, t = 6.000000 sec
fit Line
Original Data
X
Y

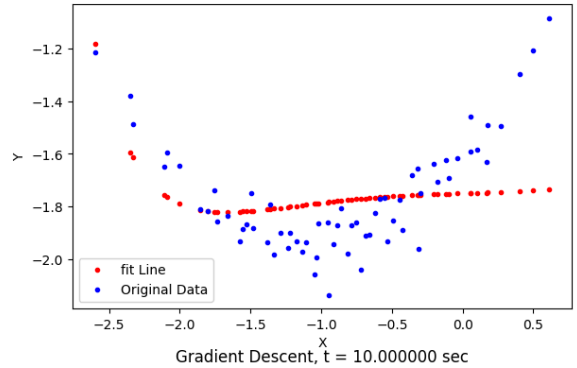

Gradient Descent, t = 6.500000 sec
fit Line
Original Data
X
Y

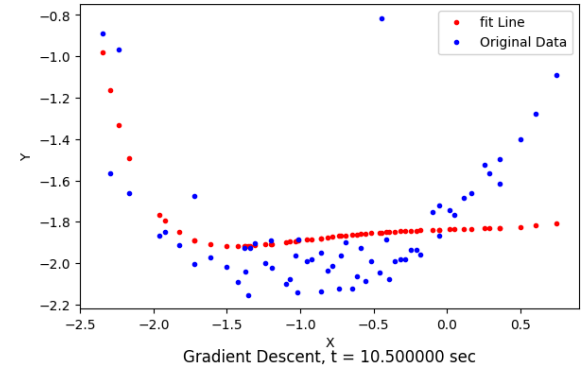

Gradient Descent, t = 7.000000 sec
fit Line
Original Data
X
Y

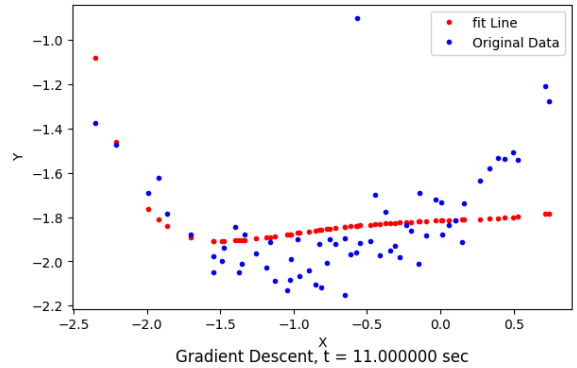

Gradient Descent, t = 7.500000 sec
fit Line
Original Data
X
Y

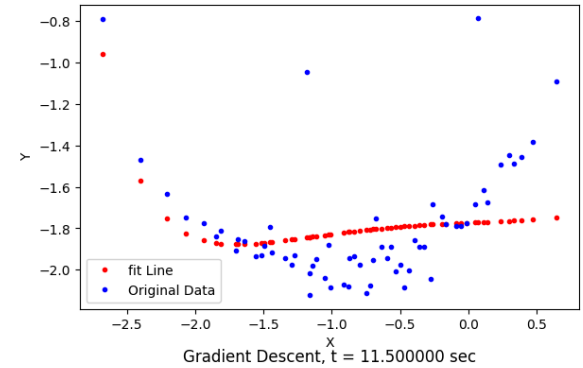

Gradient Descent, t = 8.000000 sec
fit Line
Original Data
X
Y


Gradient Descent, t = 8.500000 sec
fit Line
Original Data
X
Y


Gradient Descent, t = 9.000000 sec
fit Line
Original Data
X
Y

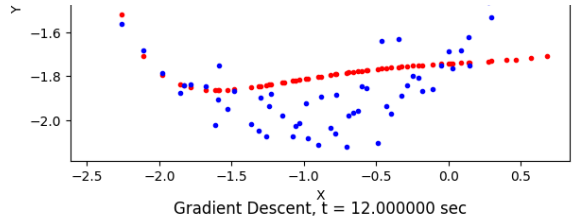

Gradient Descent, t = 9.500000 sec
fit Line
Original Data
X
Y

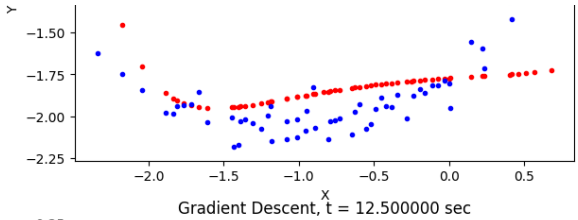

Gradient Descent, t = 10.000000 sec
fit Line
Original Data
X
Y

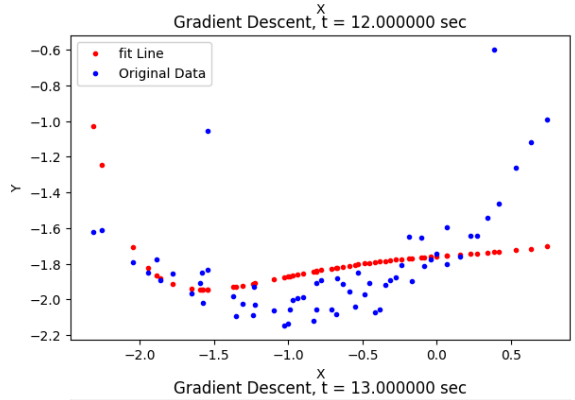

Gradient Descent, t = 10.500000 sec
fit Line
Original Data
X
Y

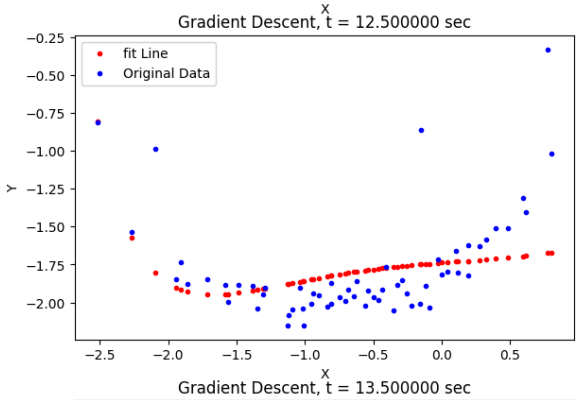

Gradient Descent, t = 11.000000 sec
fit Line
Original Data

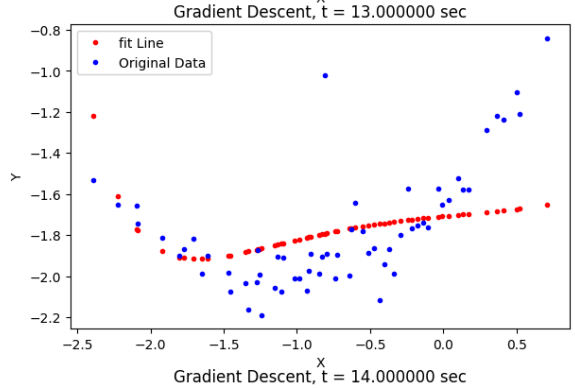

Gradient Descent, t = 11.500000 sec
fit Line
Original Data

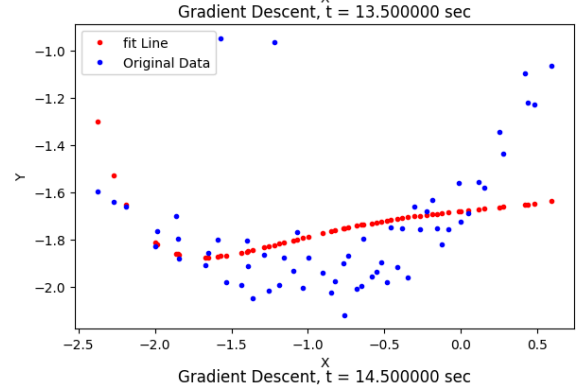

X
Y

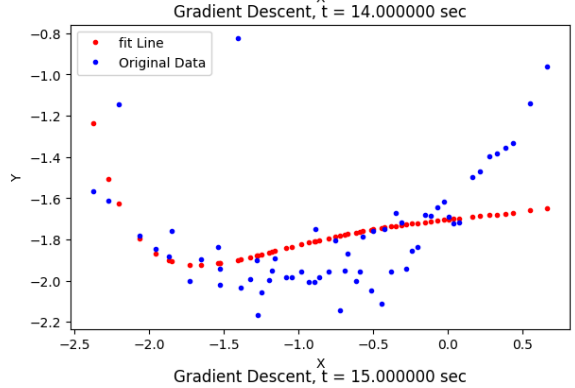

X
Y

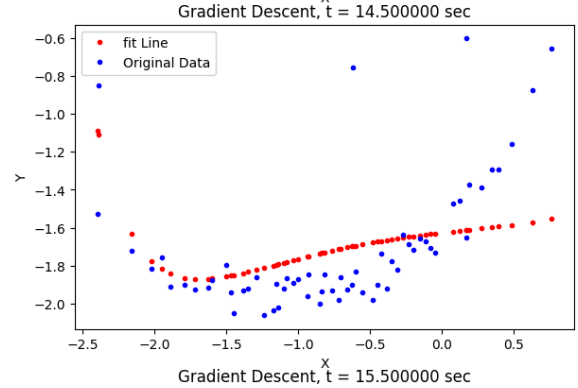

Gradient Descent, t = 12.000000 sec
fit Line
Original Data
X
Y

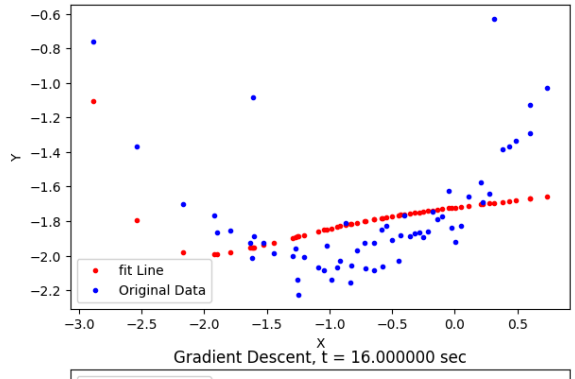

Gradient Descent, t = 12.500000 sec
fit Line
Original Data
X
Y

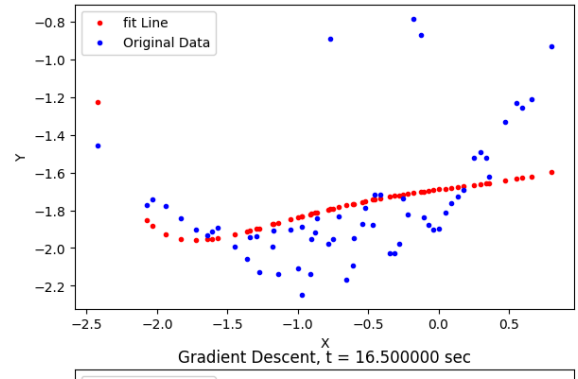

Gradient Descent, t = 13.000000 sec
fit Line
Original Data
X
Y

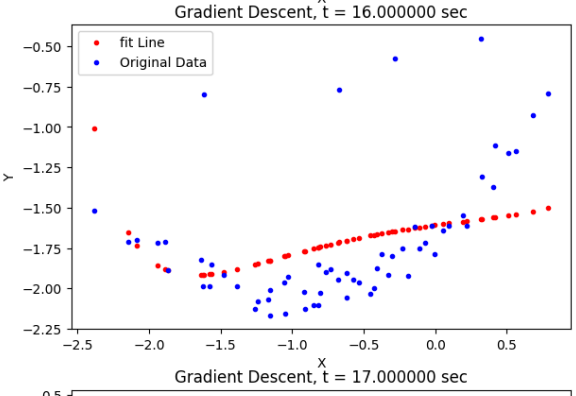

Gradient Descent, t = 13.500000 sec
fit Line
Original Data
X
Y

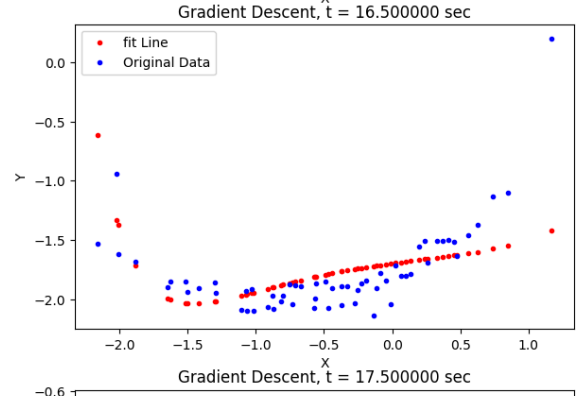

Gradient Descent, t = 14.000000 sec
fit Line
Original Data
X
Y


Gradient Descent, t = 14.500000 sec
fit Line
Original Data
X
Y

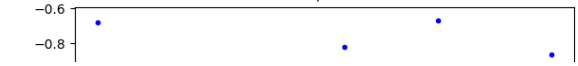

Gradient Descent, t = 15.000000 sec
fit Line
Original Data
X
Y

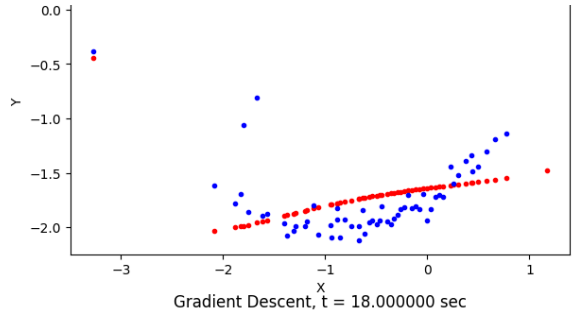

Gradient Descent, t = 15.500000 sec
fit Line
Original Data
X
Y

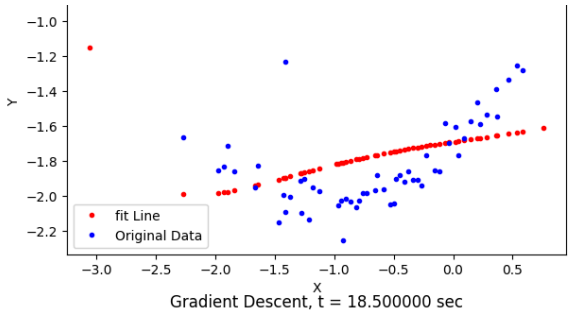

Gradient Descent, t = 16.000000 sec
fit Line
Original Data
X
Y


Gradient Descent, t = 16.500000 sec
fit Line
Original Data
X
Y

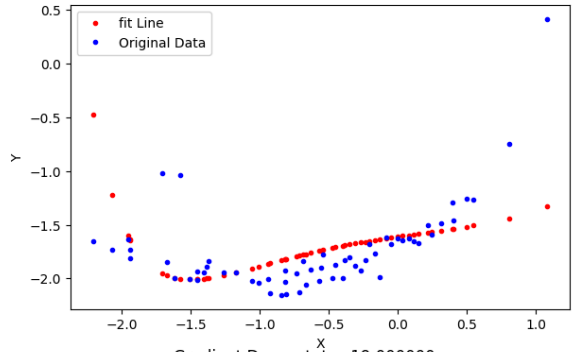

Gradient Descent, t = 17.000000 sec
fit Line
Original Data


Gradient Descent, t = 17.500000 sec

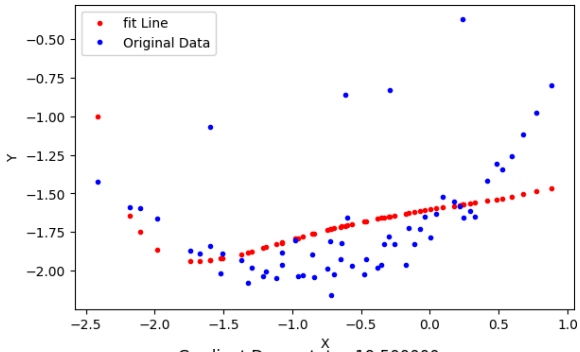




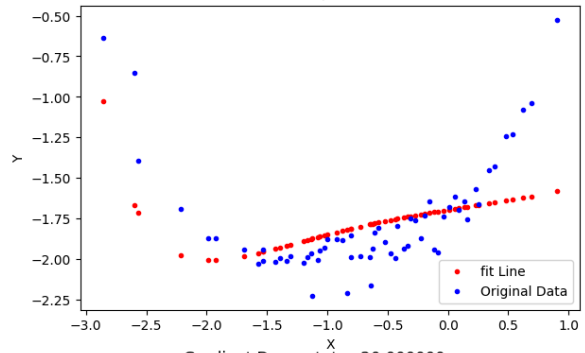
Gradient Descent, t = 18.000000 sec



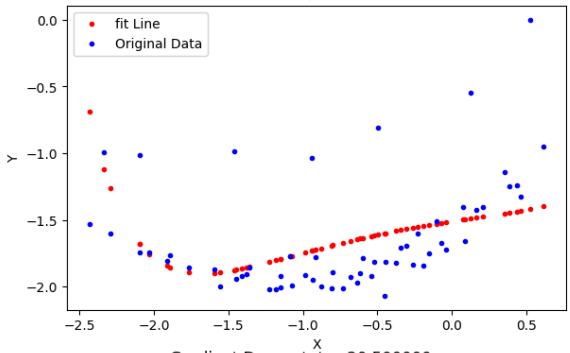
Gradient Descent, t = 18.500000 sec



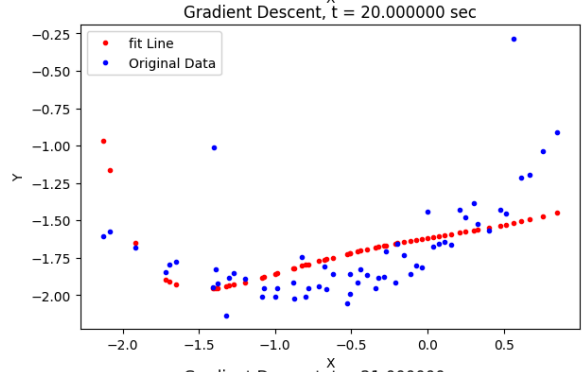
Gradient Descent, t = 19.000000 sec



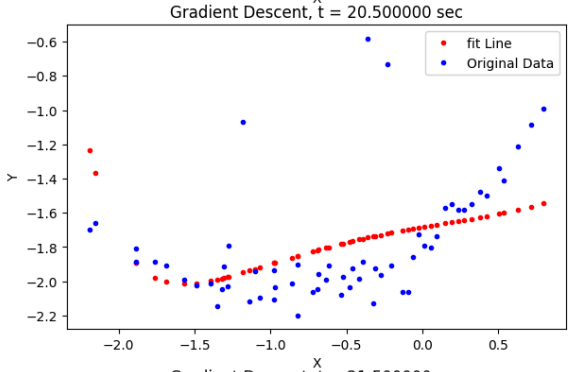
Gradient Descent, t = 19.500000 sec



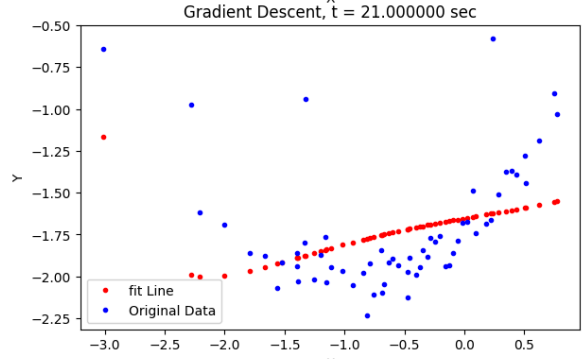
Gradient Descent, t = 20.000000 sec



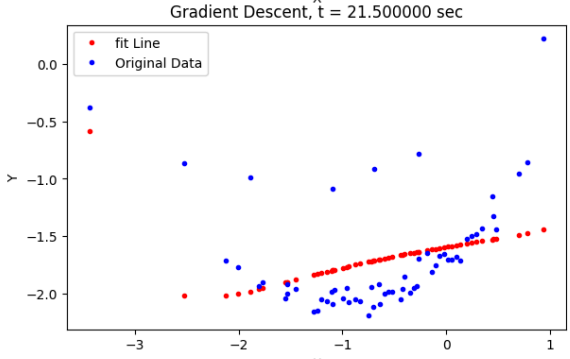
Gradient Descent, t = 20.500000 sec



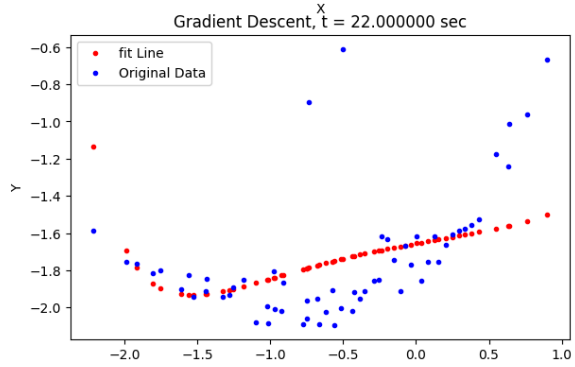
Gradient Descent, t = 21.000000 sec



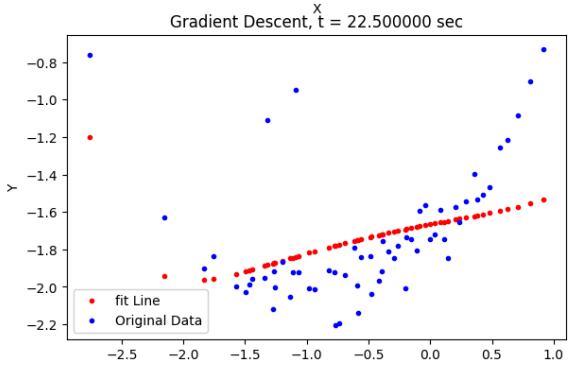
Gradient Descent, t = 21.500000 sec

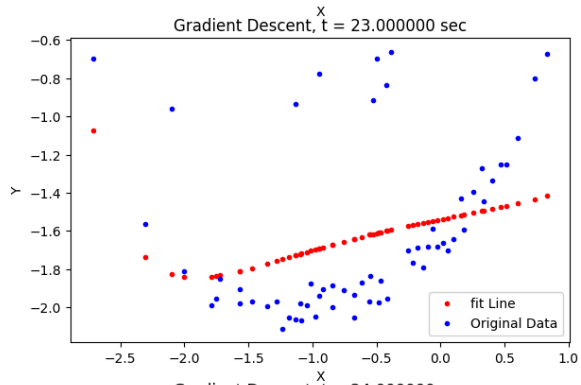


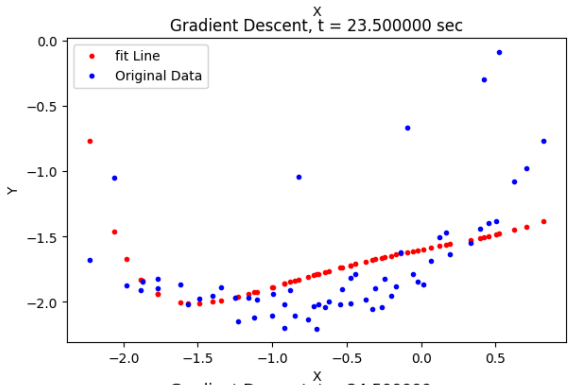
Gradient Descent, t = 22.000000 sec

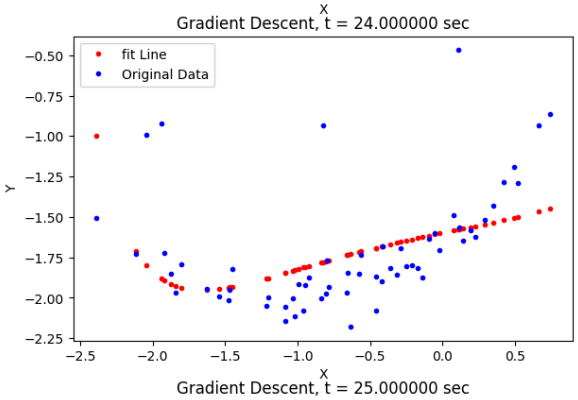


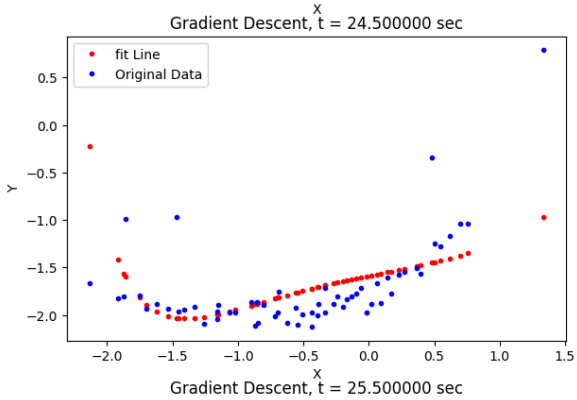
Gradient Descent, t = 22.500000 sec

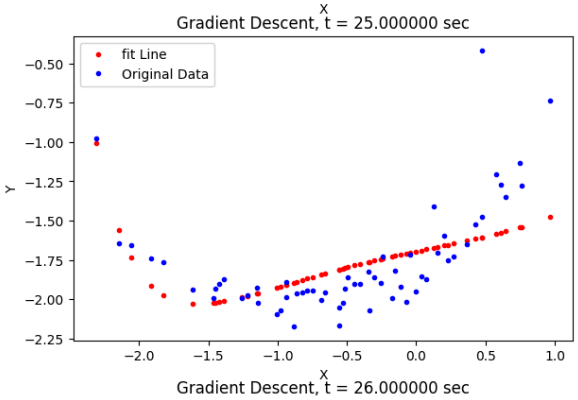


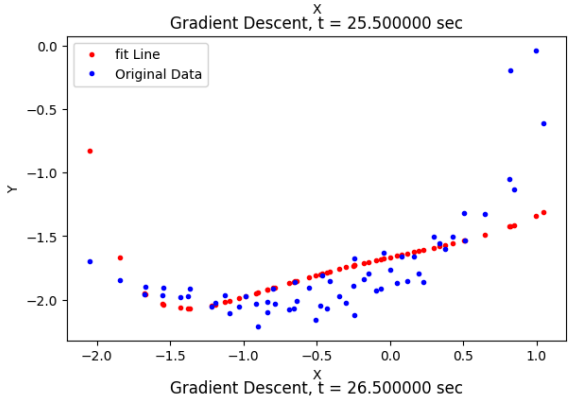

Gradient Descent, t = 23.000000 sec
fit Line
Original Data
X
Y

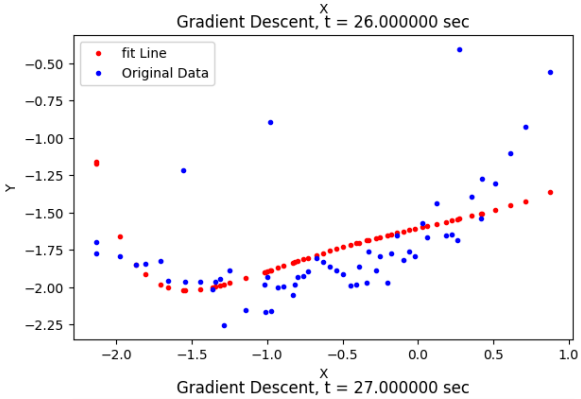

Gradient Descent, t = 23.500000 sec
fit Line
Original Data
X
Y

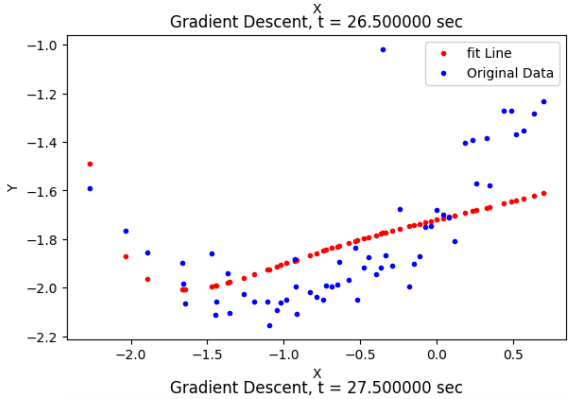

Gradient Descent, t = 24.000000 sec
fit Line
Original Data
X
Y

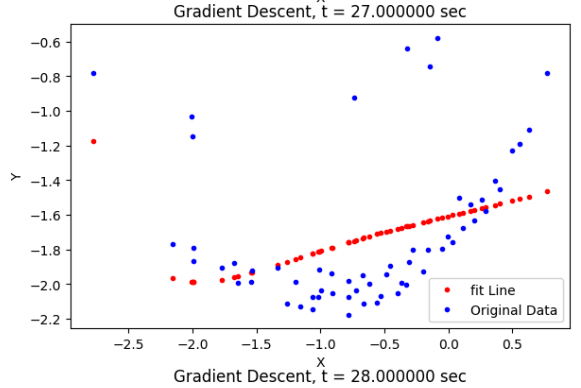

Gradient Descent, t = 24.500000 sec
fit Line
Original Data
X
Y

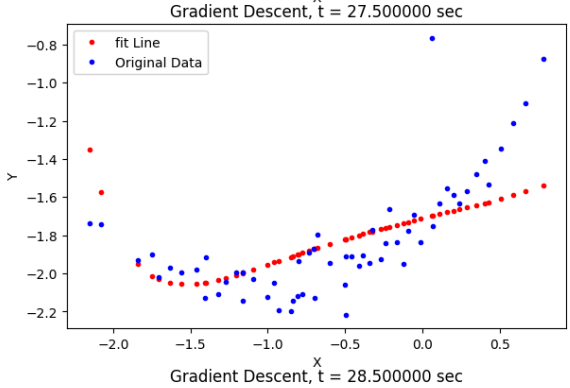

Gradient Descent, t = 25.000000 sec
fit Line
Original Data
X
Y

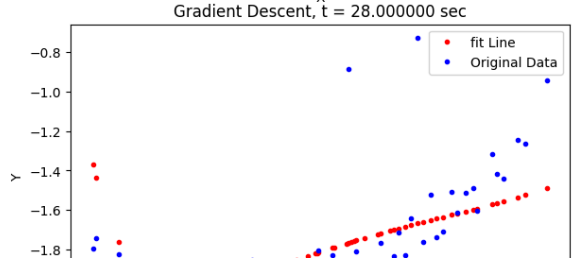

Gradient Descent, t = 25.500000 sec
fit Line
Original Data
X
Y

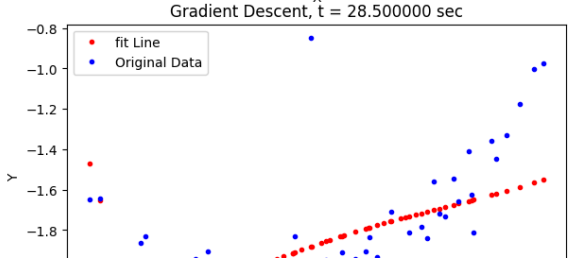

Gradient Descent, t = 26.000000 sec
fit Line
Original Data
X
Y

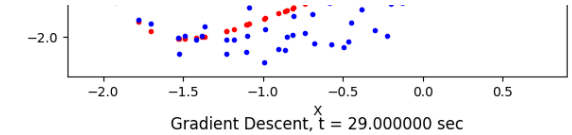

Gradient Descent, t = 26.500000 sec
fit Line
Original Data
X
Y

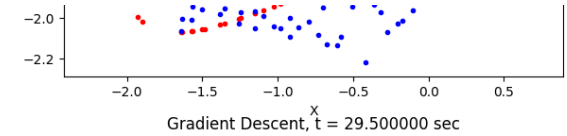

Gradient Descent, t = 27.000000 sec
fit Line
Original Data
X
Y

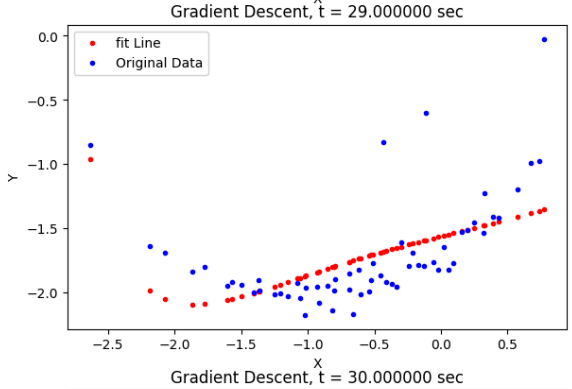

Gradient Descent, t = 27.500000 sec
fit Line
Original Data
X
Y

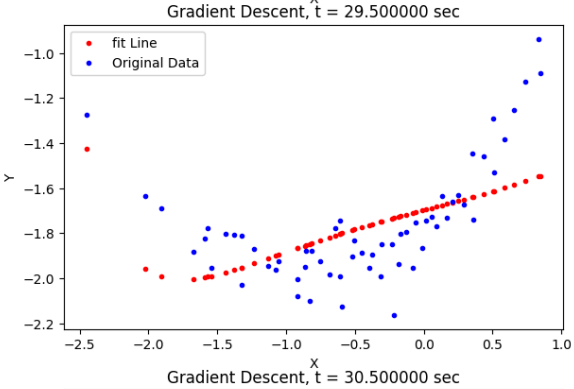

Gradient Descent, t = 28.000000 sec
fit Line
Original Data
X
Y

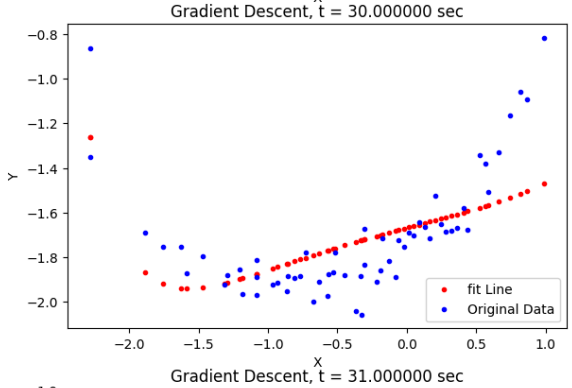

Gradient Descent, t = 28.500000 sec
fit Line
Original Data
X
Y

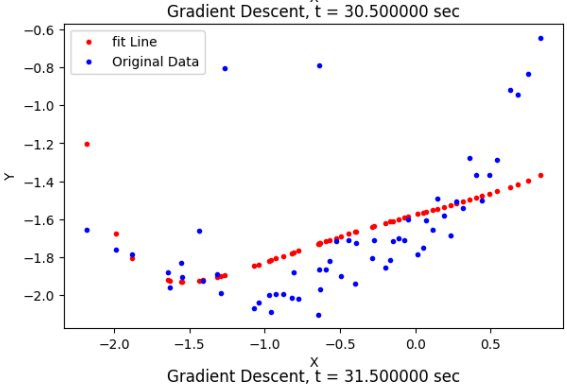


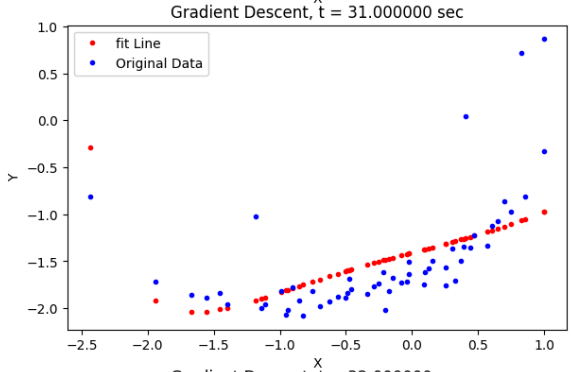


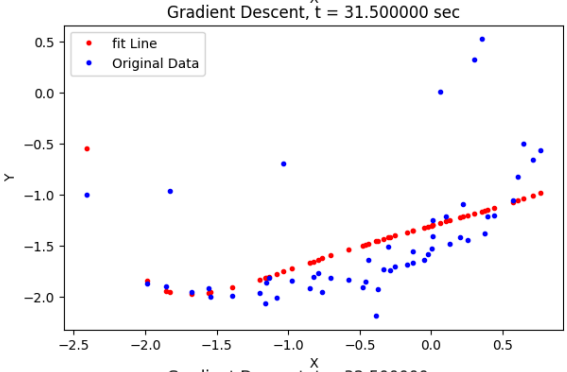

Gradient Descent, t = 29.000000 sec
fit Line
Original Data
X
Y

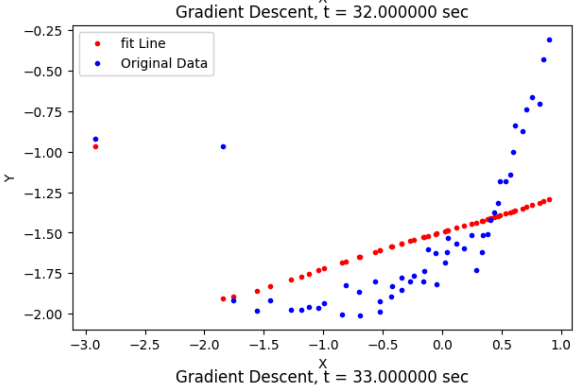

Gradient Descent, t = 29.500000 sec
fit Line
Original Data
X
Y

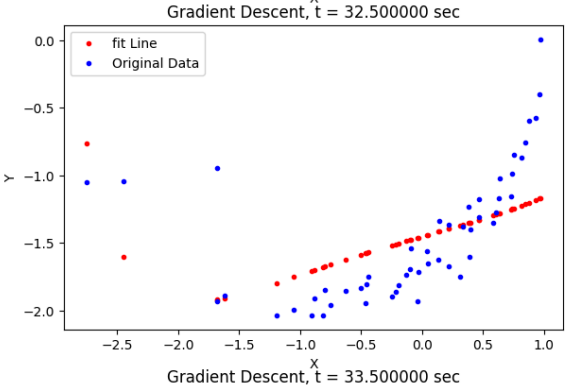

Gradient Descent, t = 30.000000 sec
fit Line
Original Data
X
Y

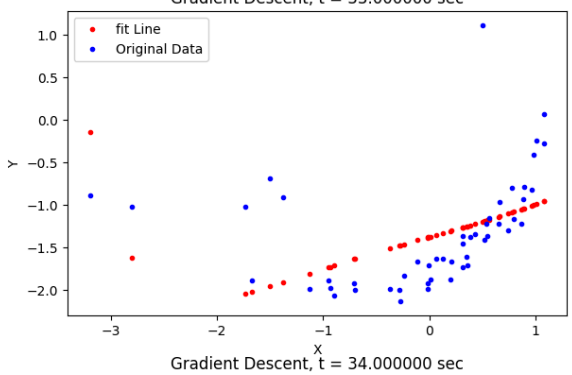

Gradient Descent, t = 30.500000 sec
fit Line
Original Data
X
Y

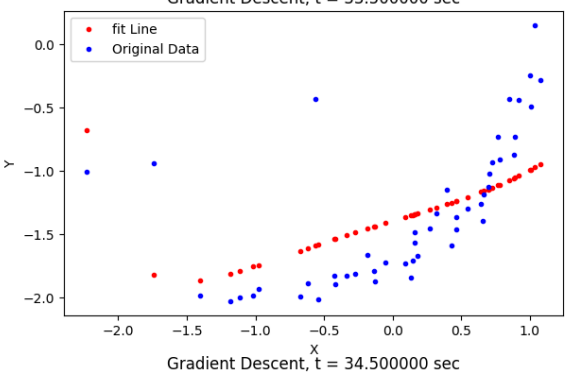

Gradient Descent, t = 31.000000 sec
fit Line
Original Data
X
Y

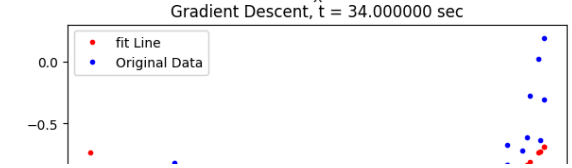

Gradient Descent, t = 31.500000 sec
fit Line
Original Data
X
Y

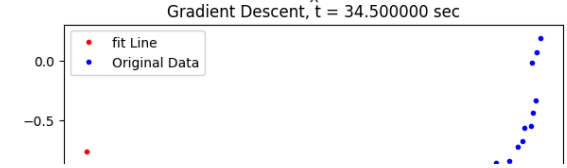

Gradient Descent, t = 32.000000 sec
fit Line
Original Data
X
Y

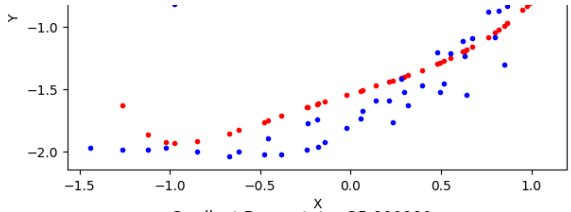

Gradient Descent, t = 32.500000 sec
fit Line
Original Data
X
Y

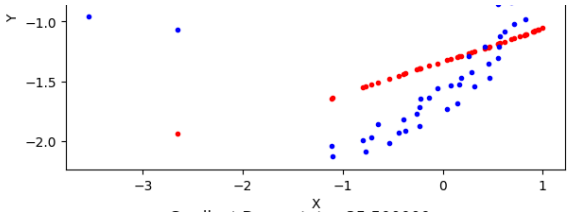

Gradient Descent, t = 33.000000 sec
fit Line
Original Data
X
Y

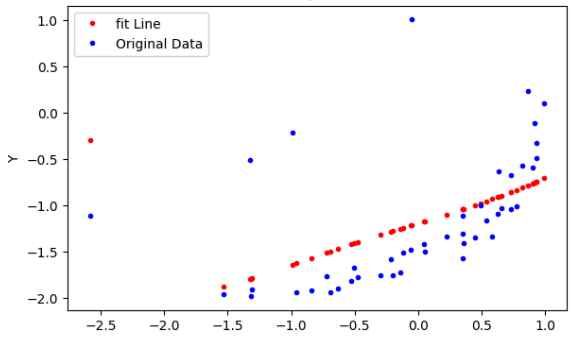

Gradient Descent, t = 33.500000 sec
fit Line
Original Data
X
Y

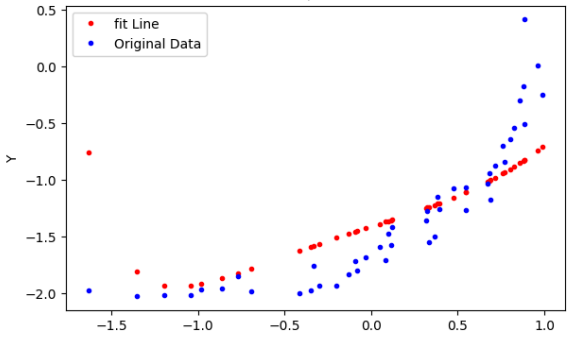

Gradient Descent, t = 34.000000 sec
fit Line
Original Data

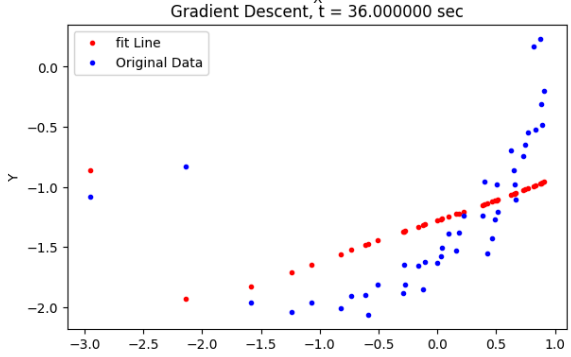

Gradient Descent, t = 34.500000 sec
fit Line
Original Data

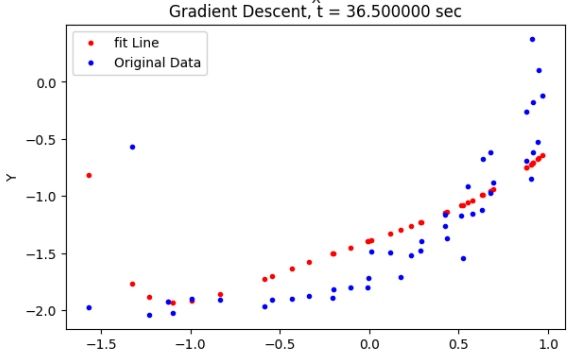

X
Y

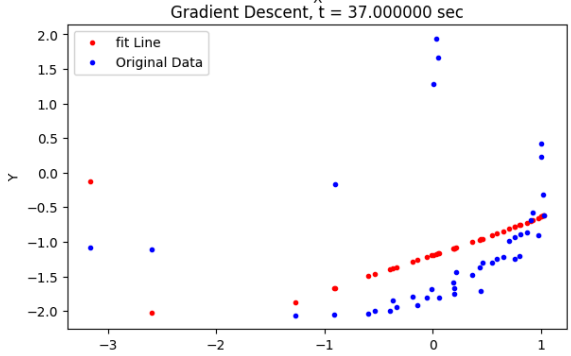

X
Y

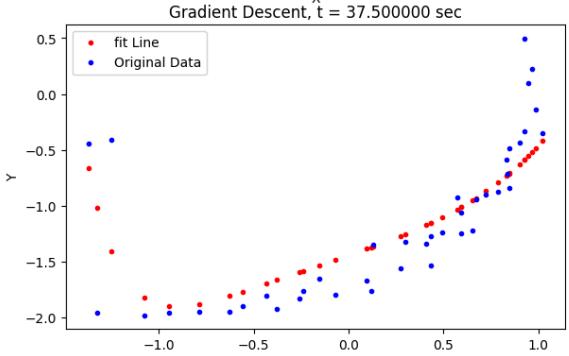

Gradient Descent, t = 35.000000 sec
fit Line
Original Data
X
Y

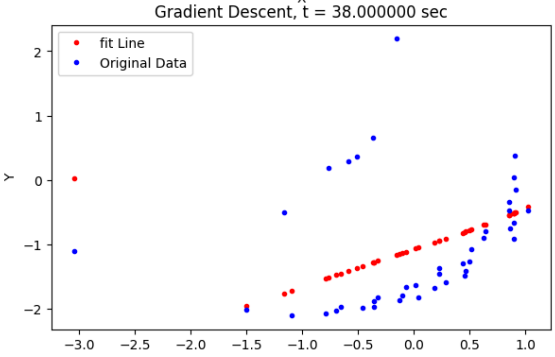

Gradient Descent, t = 35.500000 sec
fit Line
Original Data
X
Y

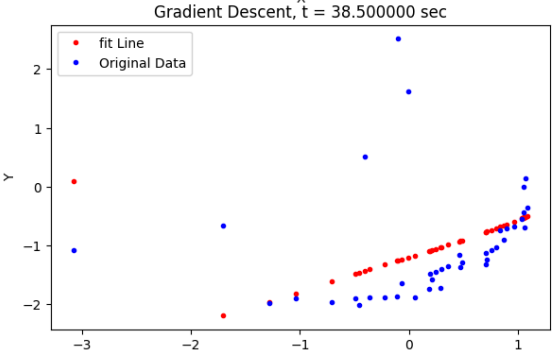

Gradient Descent, t = 36.000000 sec
fit Line
Original Data
X
Y

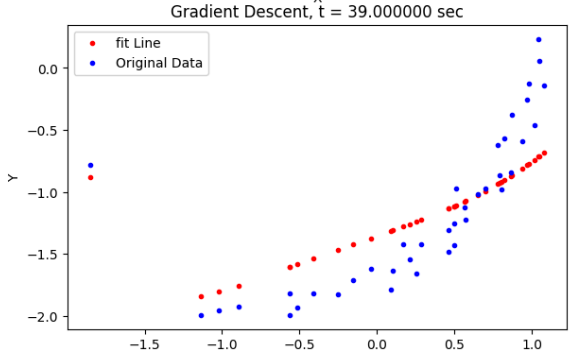

Gradient Descent, t = 36.500000 sec
fit Line
Original Data
X
Y

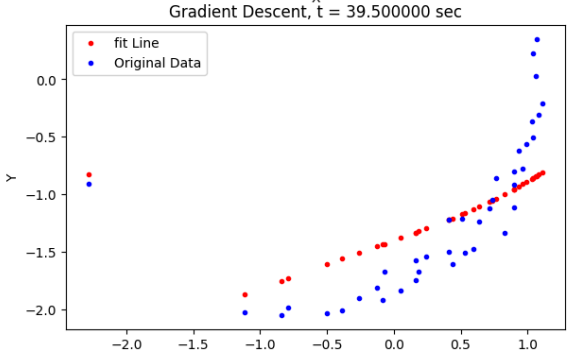

Gradient Descent, t = 37.000000 sec
fit Line
Original Data
X
Y

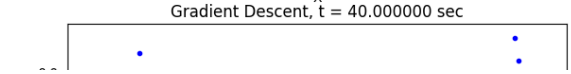

Gradient Descent, t = 37.500000 sec
fit Line
Original Data
X
Y

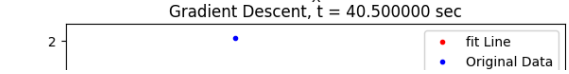

Gradient Descent, t = 38.000000 sec
fit Line
Original Data
X
Y

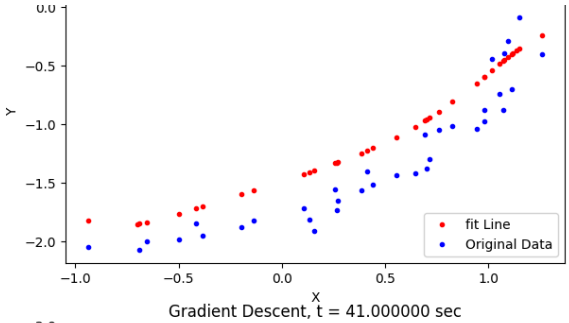

Gradient Descent, t = 38.500000 sec
fit Line
Original Data
X
Y

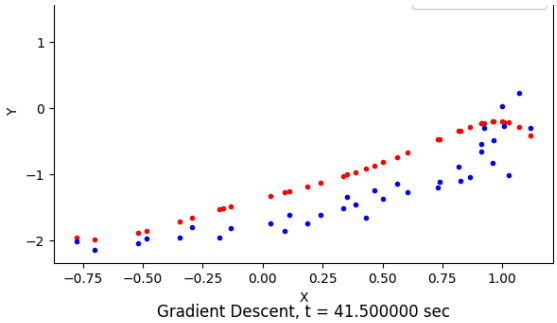

Gradient Descent, t = 39.000000 sec
fit Line
Original Data
X
Y


Gradient Descent, t = 39.500000 sec
fit Line
Original Data
X
Y

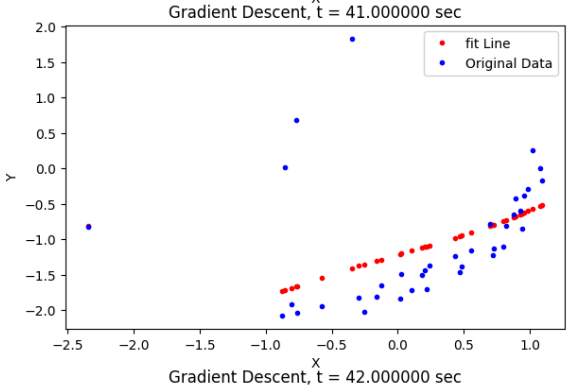

Gradient Descent, t = 40.000000 sec


Gradient Descent, t = 40.500000 sec
fit Line
Original Data

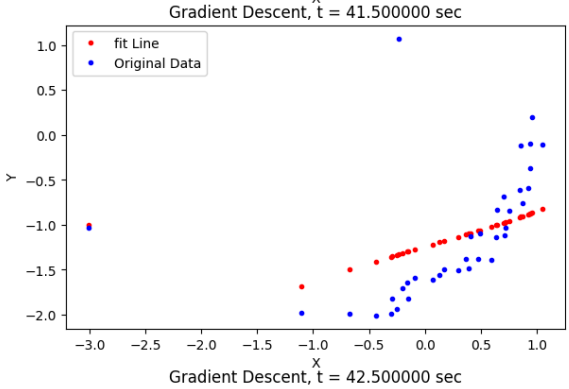




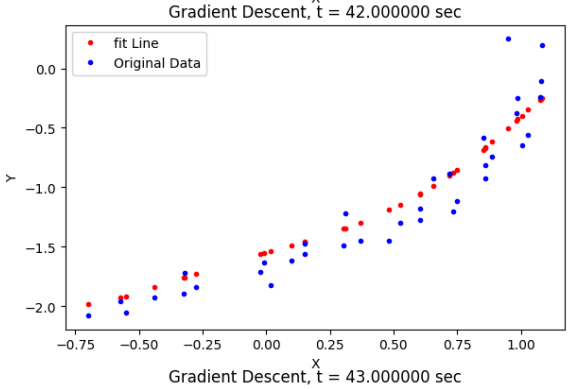
Gradient Descent, t = 41.000000 sec



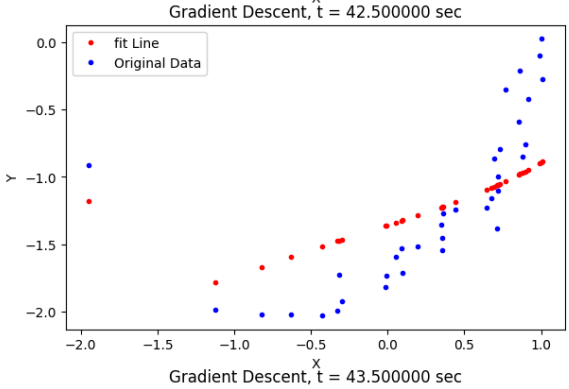
Gradient Descent, t = 41.500000 sec



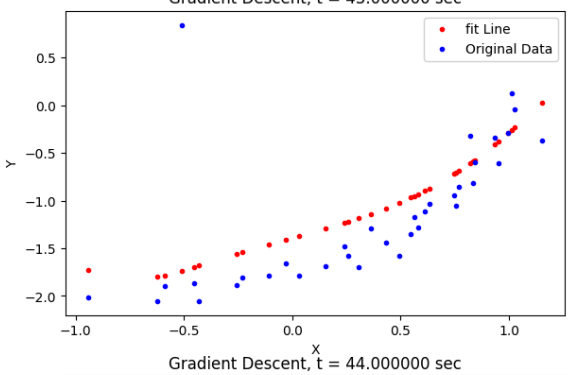
Gradient Descent, t = 42.000000 sec



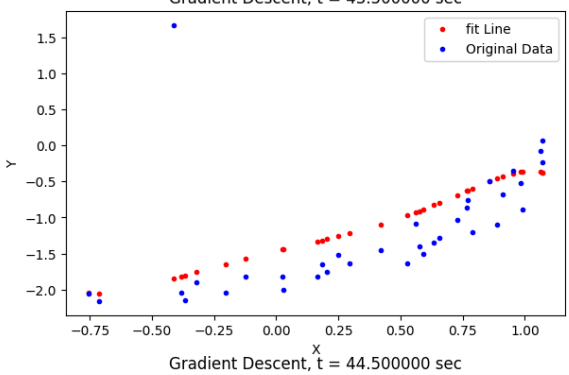
Gradient Descent, t = 42.500000 sec



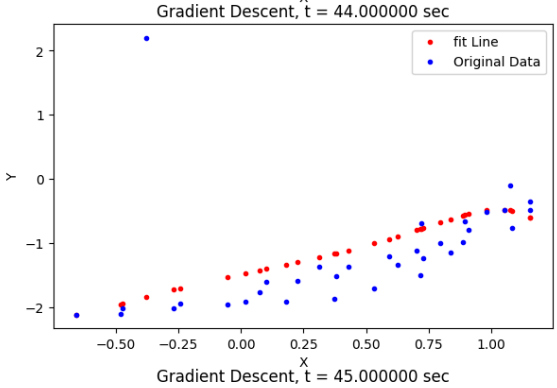
Gradient Descent, t = 43.000000 sec



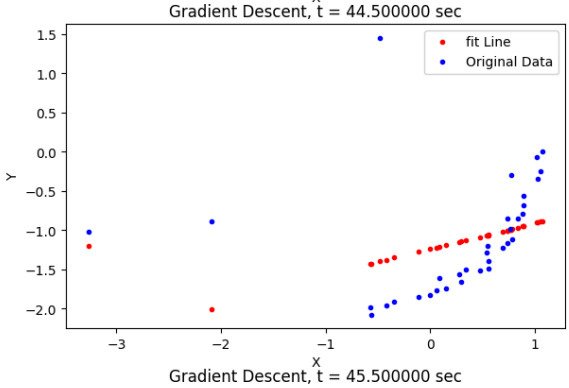
Gradient Descent, t = 43.500000 sec



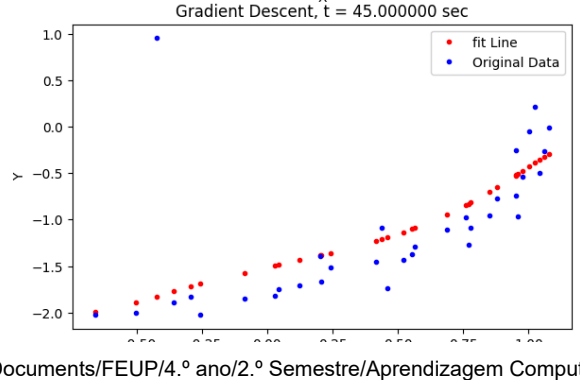
Gradient Descent, t = 44.000000 sec



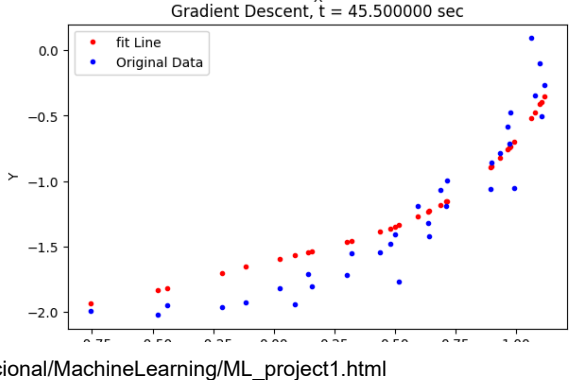
Gradient Descent, t = 44.500000 sec

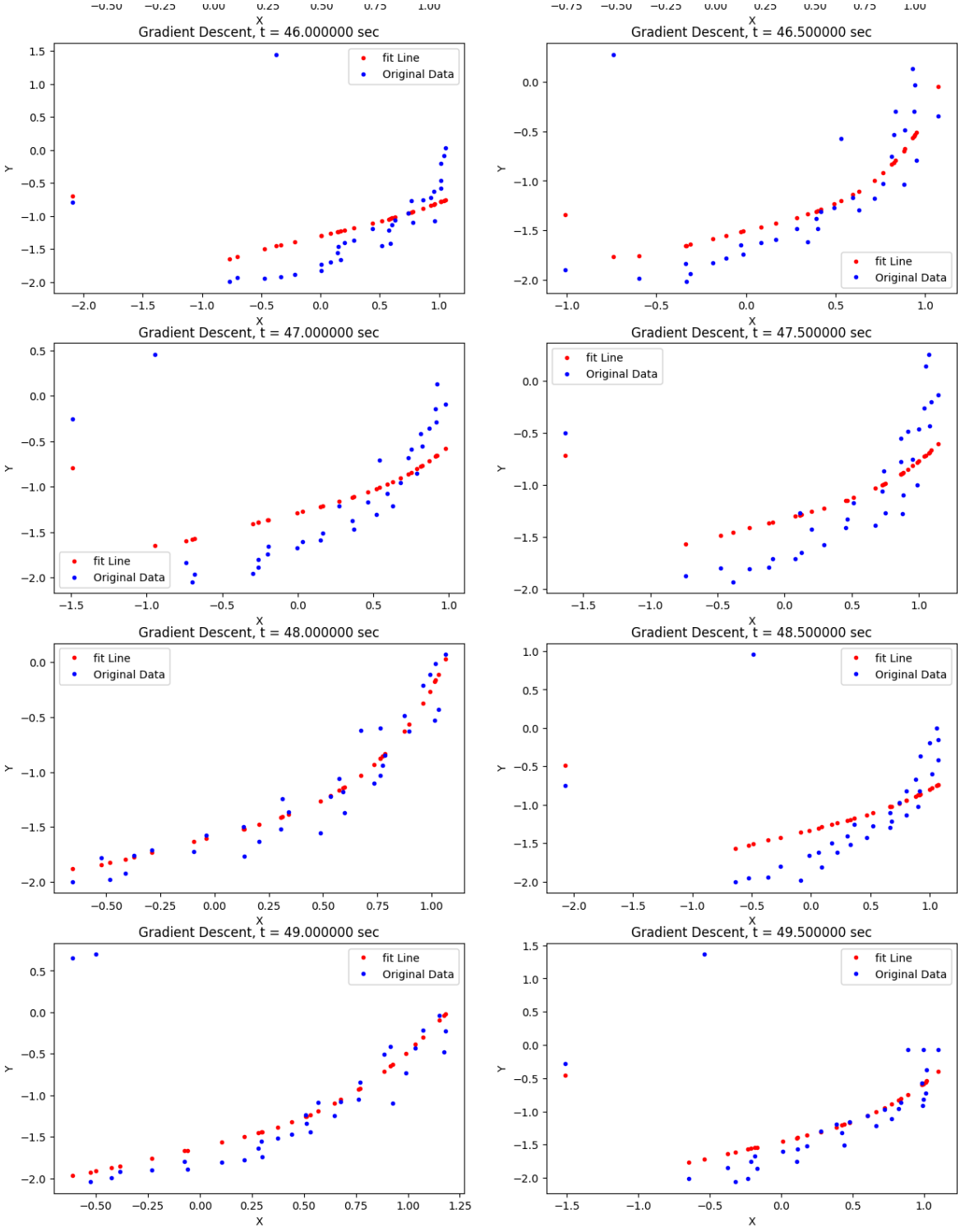


Gradient Descent, t = 45.000000 sec



Gradient Descent, t = 45.500000 sec




Gradient Descent, t = 46.000000 sec
Gradient Descent, t = 46.500000 sec
Gradient Descent, t = 47.000000 sec
Gradient Descent, t = 47.500000 sec
Gradient Descent, t = 48.000000 sec
Gradient Descent, t = 48.500000 sec
Gradient Descent, t = 49.000000 sec
Gradient Descent, t = 49.500000 sec
fit Line
Original Data
X
Y