Machine Learning 2022/2023 (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), **Daniel Costa** (danielgcosta@fe.up.pt), **Rui Gonçalves** (rjpg@fe.up.pt)

FEUP, Feb. 2023

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** 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 27** in your corresponding practical class. Upload the files until 23:59 of **April 7, 2023**.

In [ ]:
```
# 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 ic
# 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:** A05E
- **Name:** Filippo Comastri
- **Student Number:** 202211637
- **Name:** Manuel João Videira Silva
- **Student Number:** 201806123

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

In [1]:
```
!wget -O dataset.csv.zip https://www.dropbox.com/s/9y0s2ogjovkwrbm/data-set.csv.
!unzip dataset.csv.zip -d.
```

```
Archive:  dataset.csv.zip
  inflating: ./data-set.csv
  inflating: ./__MACOSX/._data-set.csv
```

In [1]:
```
# Then, run this code to get the data-set

import pandas as pd
df = pd.read_csv('data-set.csv', index_col=0)
df
#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[1]:

	time	x	y	vx	vy	angle -179	angle -178	angle -177	angle -176	angle -175	...
0	0.0	-3.946339	-2.912177	0.711051	-0.307325	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	...
2	0.2	0.000000	0.000000	0.677682	-0.285029	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	...
4	0.4	0.000000	0.000000	0.644965	-0.277222	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	...
496	49.6	0.000000	0.000000	-0.088140	-0.103430	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	...
498	49.8	0.000000	0.000000	-0.076514	-0.091199	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	...

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{align*}

$$\begin{bmatrix} x_{t+1} \\ b_{x,t+1} \end{bmatrix}$$

&=

$$\begin{bmatrix} 1 & h \\ 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x_t \\ b_{x,t} \end{bmatrix}$$

+

$$\begin{bmatrix} h \\ 0 \end{bmatrix}$$

v_{x,t}

- w_{x,t} \quad t=0, 0.1, 0.2, \ldots \

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 \ldots \end{align*}

y-direction \begin{align*}

$$\begin{bmatrix} y_{t+1} \\ b_{y,t+1} \end{bmatrix}$$

&=

$$\begin{bmatrix} 1 & h \\ 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} y_t \\ b_{y,t} \end{bmatrix}$$

+

$$\begin{bmatrix} h \\ 0 \end{bmatrix}$$

v_{y,t}

- w_{y,t} \quad t=0, 0.1, 0.2, \ldots \

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 \ldots \end{align*} 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 [2]:
```
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 [3]:
```
import matplotlib.pyplot as plt
import numpy as np
from numpy import dot
from numpy import *
from numpy.linalg import inv
from numpy.linalg import det
import random
random.seed(3)


# Predict Function
def kf_predict(X, P, A, Q, B, U):
    """
      X : The mean state estimate of the previous step (k-1) - shape(m,1)
      P : The state covariance of previous step (k-1) - shape(m,m)
      A : The transition  matrix - shape(m,m)
      Q : The process noise covariance matrix - shape(m,m)
      B : The input effect matrix - shape(p, m)
      U : The control input - shape(q,1)
    """
    X = A @ X + B @ U
    P = A @ P @ A.T + Q
    return(X,P)

def kf_update(X, P, Y, H, R):
```

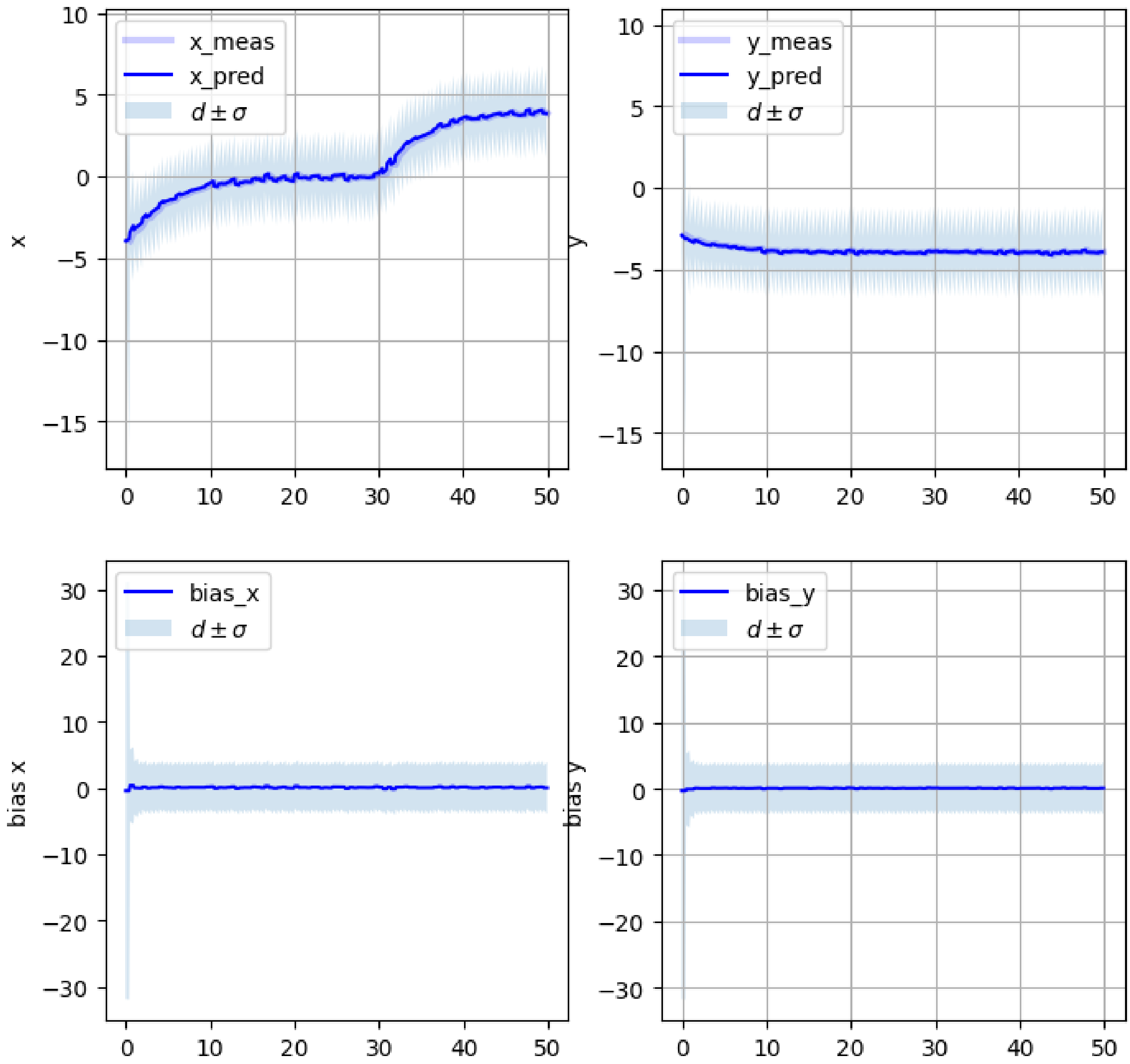
```
    """
      K  : the Kalman Gain matrix
      IS : the Covariance or predictive mean of Y
    """
    IS = H @ P @ H.T + R
    K = P @ H.T @ inv(IS)
    X = X + K @ (Y- H @ X)
    P = P - K @ IS @ K.T
#    P = P - K @ H @ P
    return (X,P)

# time step
h = 0.1

# ini state (Position, Bias) = (0,0)

X_x = np.array( [ [0.0] , [0.0]]  )
X_y = np.array( [ [0.0] , [0.0]]  )

# ini Covar : we start with a very high variance and during it the variance will
P_x = np.array( [ [ 999.0, 0.0 ] ,
                [ 0.0, 999.0 ] ]  )
P_y = np.array( [ [ 999.0, 0.0 ] ,
                [ 0.0, 999.0 ] ]  )


# state matrix
A = np.array( [ [ 1.0, h ] ,
                [ 0.0, 1.0  ] ] )

# input effect matrix
B = np.array( [ [h], [0] ] )

# meas matrix
H = np.array( [ [ 1.0, 0.0 ] ] )

## Ask for the noise

# meas noise
R = np.array([ [1] ] )

# process noise
Q = np.array(np.eye(2) * 1 )

# every 5 iteration
t_time = []

# means
x_time = []      # x position of robot over time (mean)
y_time = []
bias_x_time = []      # x bias over time (mean)
bias_y_time = []

# std devs
x_sd_time = []      # x position of robot over time (std)
y_sd_time = []
bias_sd_x_time = []      # x bias over time (std)
bias_sd_y_time = []

# up and down
```

```
x_up_time = []   # d mean + one std_dev
y_up_time = []
x_dn_time = []   # d mean - one std_dev
y_dn_time = []
b_x_up_time = []
b_y_up_time = []
b_x_dn_time  = []
b_y_dn_time = []

#
# Kalman Filter Loop
#

N_iter = len(time)    # implies dt*N_iter seconds
tt = 0
for t in arange(0, N_iter):

  U_x = np.array([ [vx[t]] ]) # put the input in the right variable
  U_y = np.array([ [vy[t]] ])

  (X_x, P_x) =  kf_predict(X_x, P_x, A, Q, B, U_x)
  (X_y, P_y) =  kf_predict(X_y, P_y, A, Q, B, U_y)

  if t%5 == 0:
    tt += 0.5
    t_time.append(tt)
    Y_x = np.array([ [x[t]] ])
    Y_y = np.array([ [y[t]] ])

    (X_x, P_x) = kf_update(X_x, P_x, Y_x, H, R)
    (X_y,P_y) =  kf_update(X_y, P_y, Y_y, H, R)

  # mean
  x_time.append( X_x[0].item() )
  bias_x_time.append( X_x[1].item() )
  y_time.append( X_y[0].item() )
  bias_y_time.append( X_y[1].item() )

  # std devs
  x_sd_time.append( P_x[0][0].item() )
  bias_sd_x_time.append( P_x[1][1].item() )
  y_sd_time.append( P_y[0][0].item() )
  bias_sd_y_time.append( P_y[1][1].item() )

  # up and down
  x_up_time.append( X_x[0].item() + sqrt( P_x[0][0]).item() )
  y_up_time.append( X_y[0].item() + sqrt( P_y[0][0]).item() )
  x_dn_time.append( X_x[0].item() - sqrt( P_x[0][0]).item() )
  y_dn_time.append( X_y[0].item() - sqrt( P_y[0][0]).item() )

  b_x_up_time.append( X_x[1].item() + sqrt( P_x[1][1]).item() )
  b_y_up_time.append( X_y[1].item() + sqrt( P_y[1][1]).item() )
  b_x_dn_time.append( X_x[1].item() - sqrt( P_x[1][1]).item() )
  b_y_dn_time.append( X_y[1].item() - sqrt( P_y[1][1]).item() )

x_no_zero = [el for el in x if el !=0]
y_no_zero = [el for el in y if el !=0]
fig = plt.figure(figsize=(8,8))
# x
chart1 = fig.add_subplot(221)
```

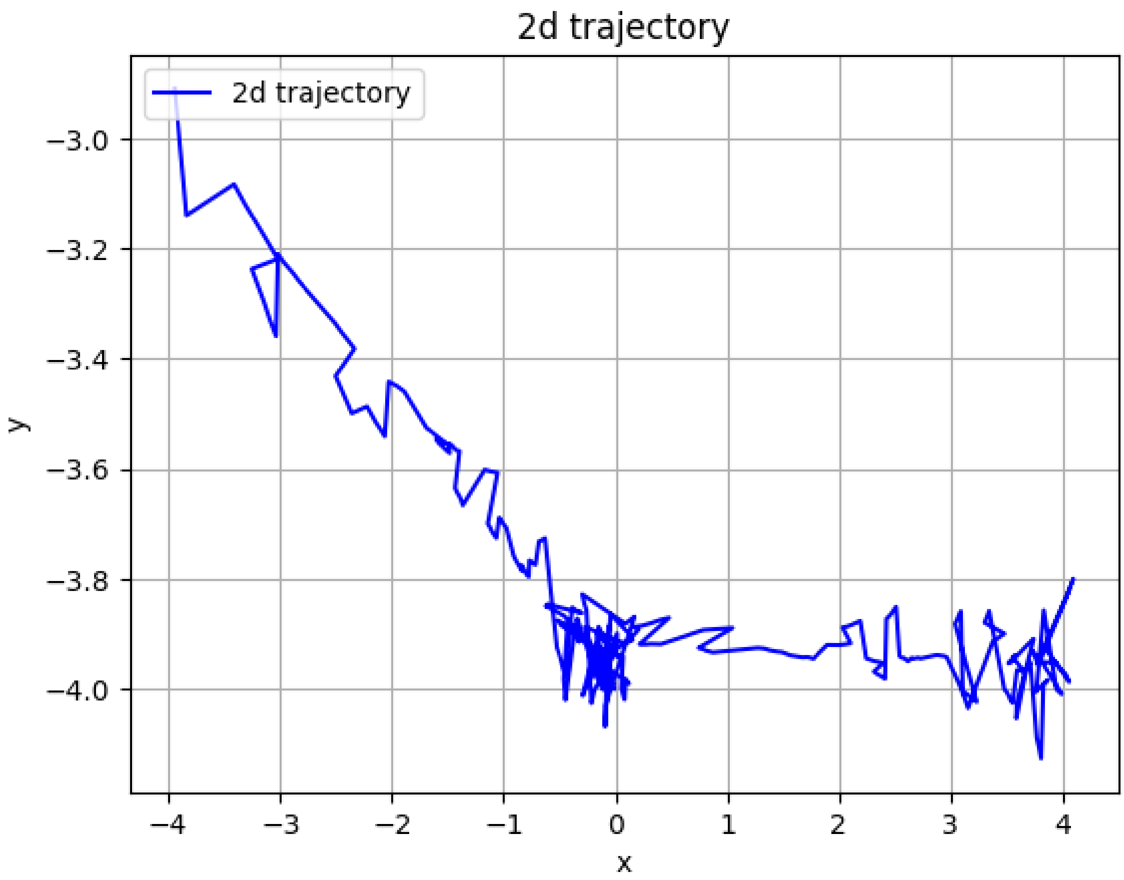
```
chart1.plot(t_time, x_no_zero, label='x_meas', c="b", linewidth=3, alpha=0.2)
chart1.plot(time,x_time, label='x_pred', c="b")
chart1.fill_between(time, x_dn_time, x_up_time, alpha=0.2, linewidth=0, label='$
plt.legend(loc='upper left')
chart1.set_ylabel('x')
plt.grid()

# y
chart2 = fig.add_subplot(222)
chart2.plot(t_time, y_no_zero, label='y_meas', c="b", linewidth=3, alpha=0.2)
chart2.plot(time,y_time, label='y_pred', c="b")
chart2.fill_between(time, y_dn_time, y_up_time, alpha=0.2, linewidth=0, label='$
plt.legend(loc='upper left')
chart2.set_ylabel('y')
plt.grid()

# bias x
chart3 = fig.add_subplot(223)
chart3.plot(time,bias_x_time, label='bias_x', c="b")
chart3.fill_between(time, b_x_dn_time, b_x_up_time, alpha=0.2, linewidth=0, labe
plt.legend(loc='upper left')
chart3.set_ylabel('bias x')

# bias y
chart4 = fig.add_subplot(224)
chart4.plot(time,bias_y_time, label='bias_y', c="b")
chart4.fill_between(time, b_y_dn_time, b_y_up_time, alpha=0.2, linewidth=0, labe
plt.legend(loc='upper left')
chart4.set_ylabel('bias y')

plt.grid()

'''# v
chart2 = fig.add_subplot(212)
chart2.plot(t_time, train_v_time, label='train_v', c="g", linewidth=3, alpha=0.2
chart2.plot(t_time,v_time, label='v', c="g")
chart2.fill_between(t_time,v_dn_time,v_up_time, alpha=0.2, label='$v\pm\sigma$')
chart2.set_ylabel('v [m/s]')
chart2.set_xlabel('t [s]')
plt.legend(loc='upper left')
plt.grid()
plt.show()
'''

# End For Loop
```

Out[3]:

```
'# v\nchart2 = fig.add_subplot(212)\nchart2.plot(t_time, train_v_time, label=
\'train_v\', c="g", linewidth=3, alpha=0.2)\nchart2.plot(t_time,v_time, label=
\'v\', c="g")\nchart2.fill_between(t_time,v_dn_time,v_up_time, alpha=0.2, label
=\'$v\\pm\\sigma$\')\nchart2.set_ylabel(\'v [m/s]\')\nchart2.set_xlabel(\'t [s]
\')\nplt.legend(loc=\'upper left\')\nplt.grid()\nplt.show()\n'
```



In [5]:
```
# 2d trajectory
plt.figure()
plt.plot(x_time, y_time, label='2d trajectory', c="b")
plt.title('2d trajectory')
plt.ylabel('y')
plt.xlabel('x')
plt.legend(loc='upper left')
plt.grid()
```



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,

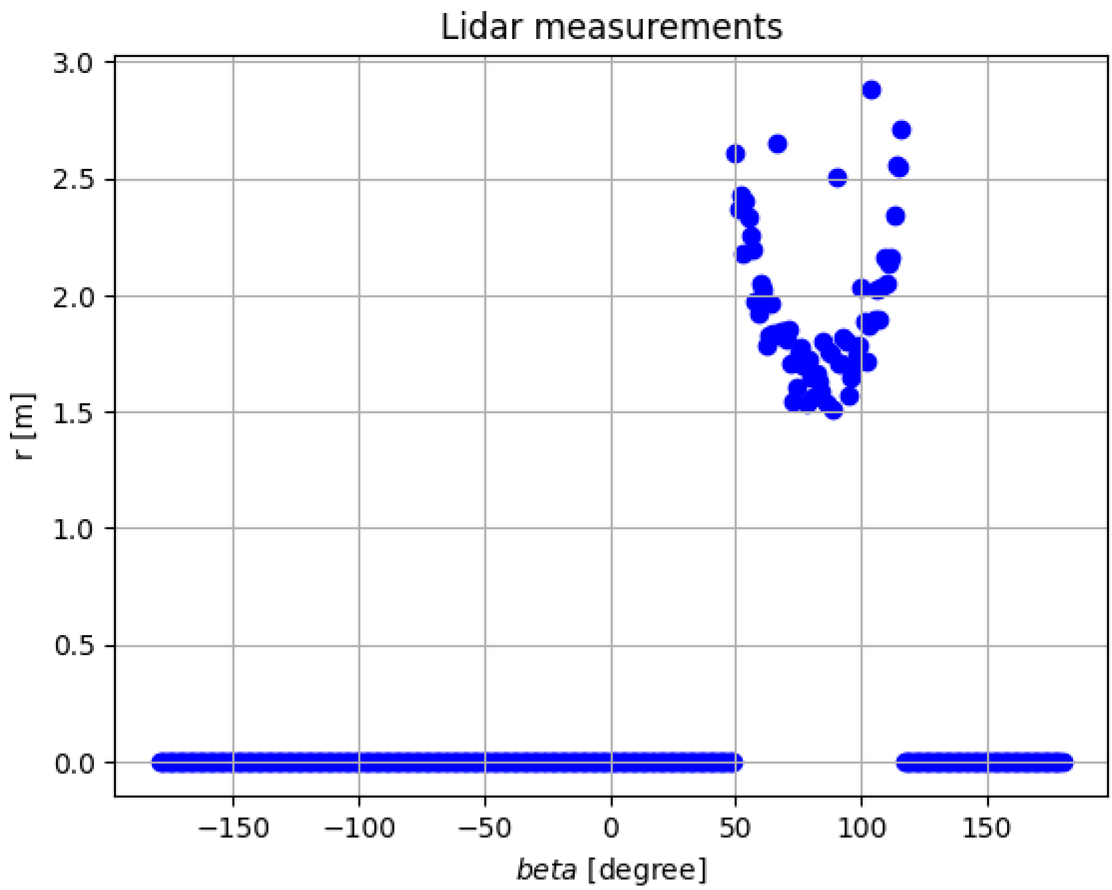
$$r_t = \{r_\beta + \eta_r : \beta = -179^o, -178^o, \dots, 0^o, \dots, 180^o\}$$

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 $t = 0, 0.5, 1.0, 1.5, \dots$ (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 [4]:
```
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

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

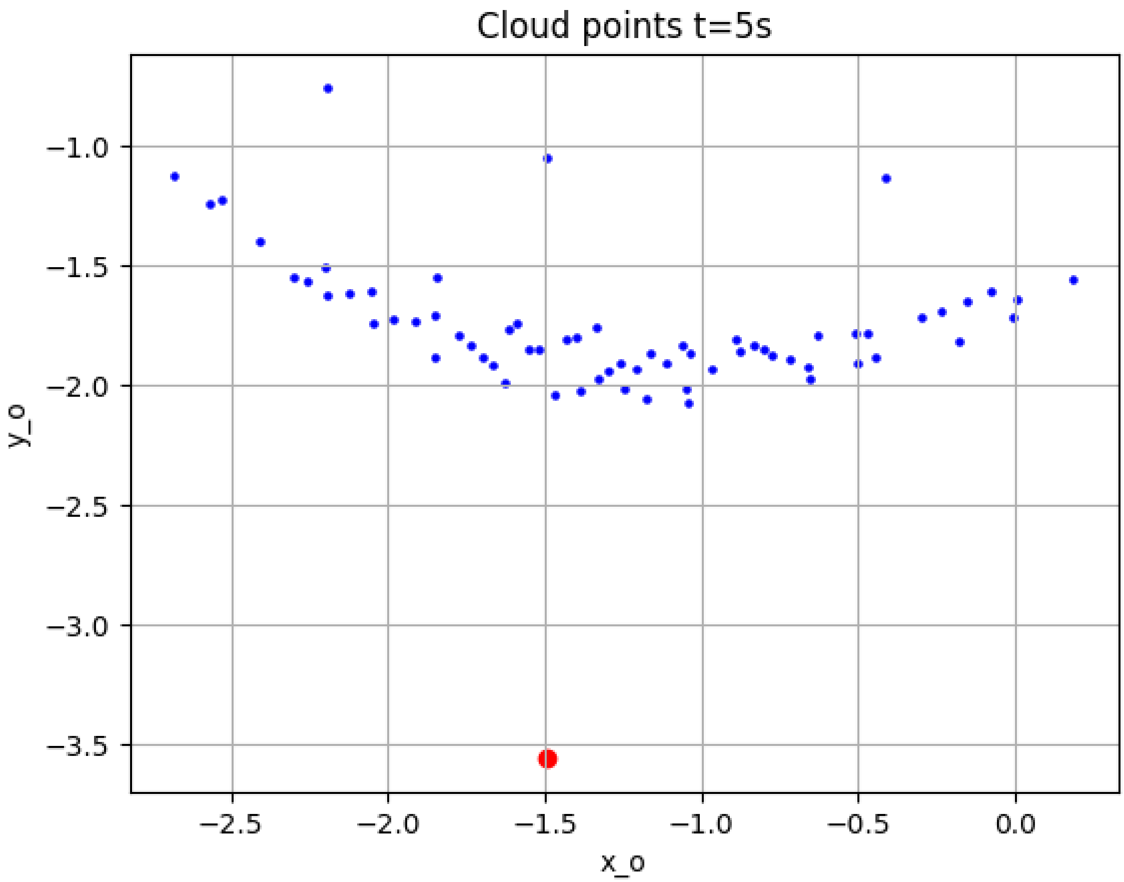
2. Perform a linear regression for the previous data using a model of the type

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

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 [15]:
```
# Part 2.1.1
import math
Lidar_range = df.iloc[:, np.arange(5,365,1)].values

#Build the cloud points in 2D plan
x_o, y_o = [], []
angle = np.linspace(-179, 180, num=360)

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

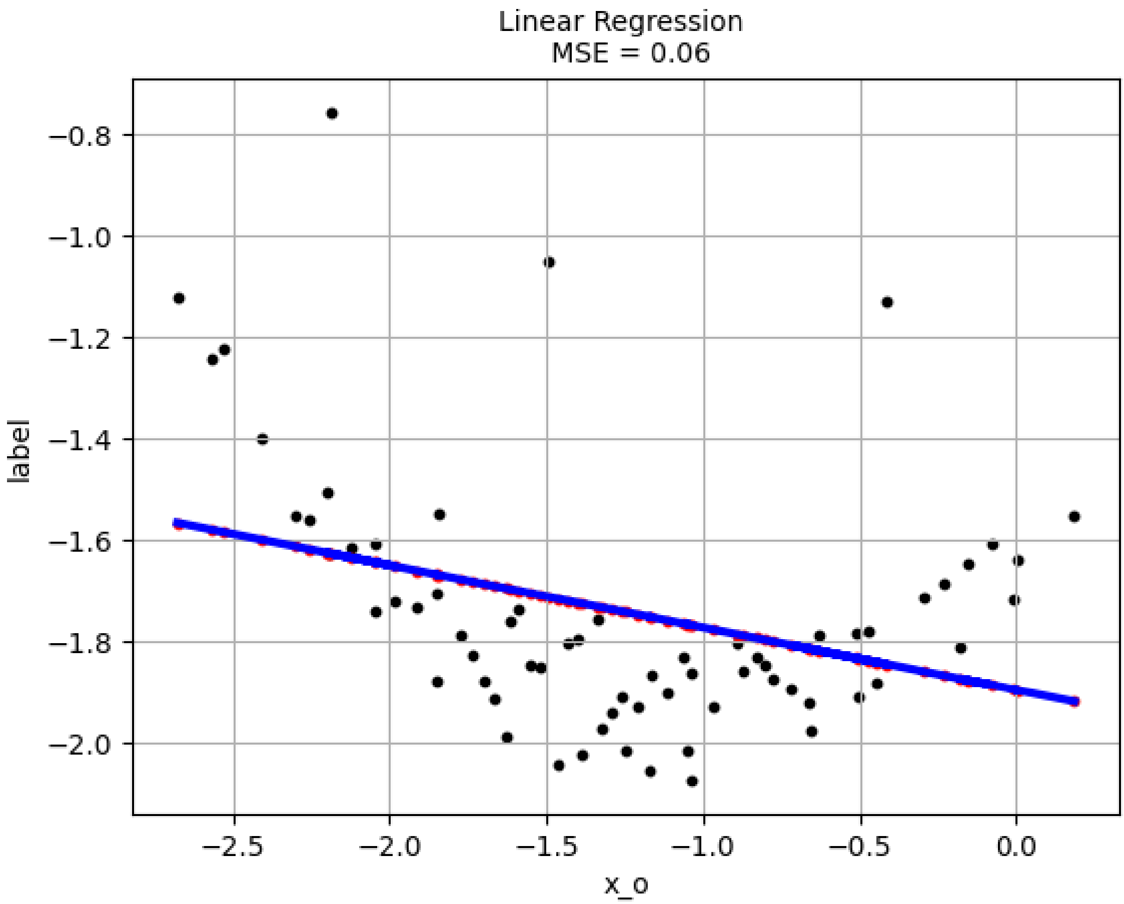
```
for i in range(len(Lidar_range[t])):
  if Lidar_range[t][i] > 0:
    x_o.append(x_time[t]+Lidar_range[t][i]*np.cos(angle[i]*math.pi/180))
    y_o.append(y_time[t]+Lidar_range[t][i]*np.sin(angle[i]*math.pi/180))

plt.figure()
plt.scatter(x_o, y_o, color='b',s=5)
plt.scatter(x_time[t], y_time[t], color='r')
plt.title('Cloud points t=5s')
plt.ylabel('y_o')
plt.xlabel('x_o')
plt.grid()
```



In [16]:
```
# Part 2.1.2

# Linear regression

import numpy as np
from scipy import linalg
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

x_o_mat = np.array(x_o).reshape(len(x_o),1)

#Create X matrix with ones
X = np.ones((len(x_o), 1), dtype=float)
X = np.concatenate((X, x_o_mat), axis = 1)
#Create Y matrix

Y = np.array(y_o).reshape(len(y_o),1)
```

```
print(" --- Linear Regression --- ")

# Normal Equation: (X.t X)^-1 X.t Y
theta = np.linalg.inv(np.transpose(X) @ X) @ np.transpose(X) @ Y

print("Parameters theta =\n", theta.T)

# Precticted values
Y_predict = X @ theta

#Model's error
MSE = mean_squared_error(Y, Y_predict)
print('MSE ',MSE)

### Plot
plt.scatter(x_o, Y, color="black",s=10)
plt.scatter(x_o,Y_predict,color="red",s=10)
plt.plot(x_o, Y_predict,  color="blue", linewidth=3)
plt.grid()

title = 'MSE = {}'.format(round(MSE,2))
plt.title("Linear Regression \n " + title, fontsize=10)
plt.xlabel('x_o')
plt.ylabel('label')
plt.show()


# Using sklearn
from sklearn import linear_model
from sklearn.metrics import mean_squared_error, r2_score
model = linear_model.LinearRegression()
model.fit(x_o_mat, Y)
print('--- Using SKLEARN ---')
print("Intercept = ", model.intercept_)
print("Coef = ", model.coef_)
```

```
 --- Linear Regression ---
Parameters theta =
 [[-1.89576421 -0.12289806]]
MSE  0.056577742699758266
```




```
--- Using SKLEARN ---
Intercept =  [-1.89576421]
Coef =  [[-0.12289806]]
```

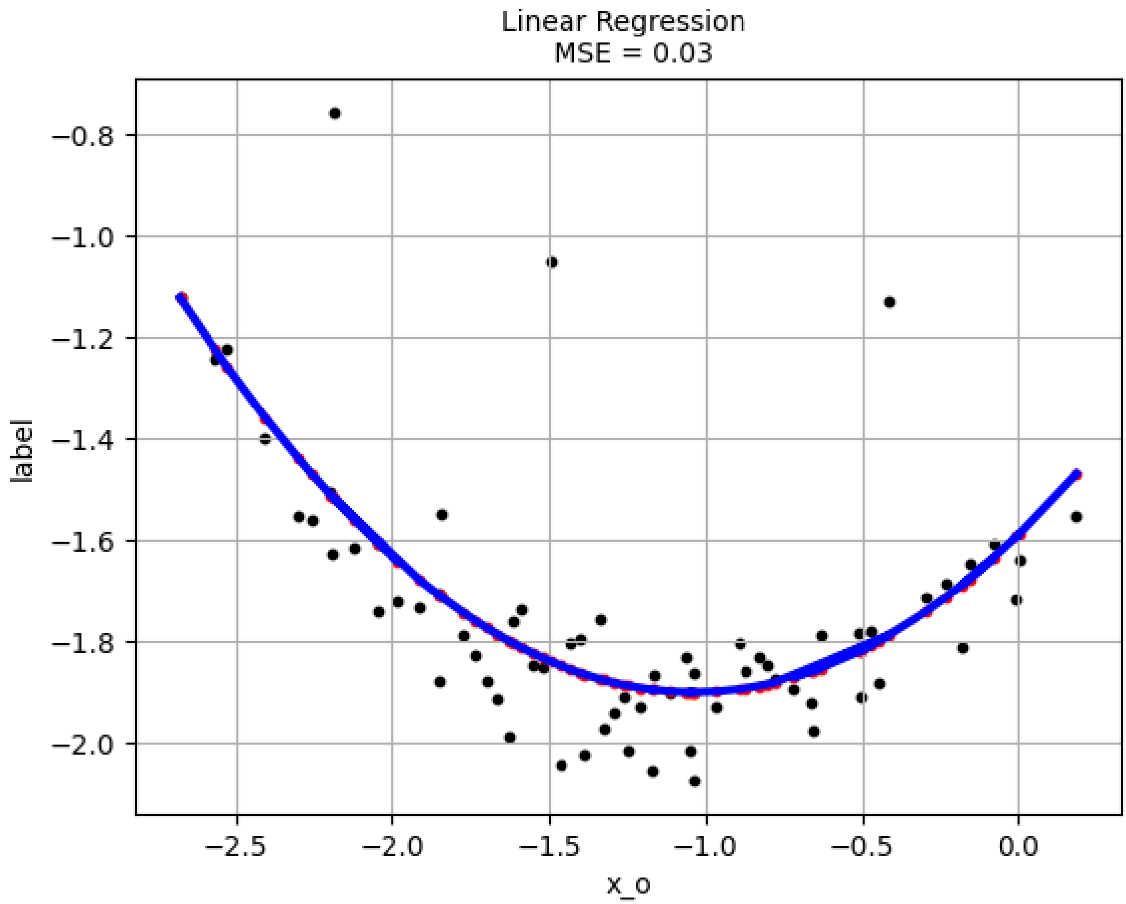
2.2 Repeat the previous exercise but now with a polynomial model of the type

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

In [17]:
```
# Linear regression

import numpy as np
from scipy import linalg
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

x_o_squared=np.array(x_o).reshape(len(x_o),1) ** 2

#Create X matrix with ones
X = np.ones((len(x_o), 1), dtype=float)
X = np.concatenate((X, np.array(x_o).reshape(len(x_o),1)), axis = 1)
X = np.concatenate((X, x_o_squared), axis = 1)

#Create Y matrix

Y = np.array(y_o).reshape(len(y_o),1)

print(" --- Linear Regression --- ")

# Normal Equation: (X.t X)^-1 X.t Y
theta = np.linalg.inv(np.transpose(X) @ X) @ np.transpose(X) @ Y

print("Parameters theta =\n", theta.T)
```

```
# Prectected values
Y_predict = X @ theta

#Model's error
MSE = mean_squared_error(Y, Y_predict)
print('MSE ',MSE)
### Plot
plt.scatter(x_o, Y, color="black",s=10)
plt.scatter(x_o,Y_predict,color="red",s=10)
plt.plot(x_o, Y_predict,  color="blue", linewidth=3)
plt.grid()

title = 'MSE = {}'.format(round(MSE,2))
plt.title("Linear Regression \n " + title, fontsize=10)
plt.xlabel('x_o')
plt.ylabel('label')
plt.show()


# Using sklearn
from sklearn import linear_model
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import PolynomialFeatures

nb_degree=2
polynomial_feats = PolynomialFeatures(degree=nb_degree)
X_TRANSF = polynomial_feats.fit_transform(np.array(x_o).reshape(len(x_o),1))
model = linear_model.LinearRegression()
model.fit(X_TRANSF[:,[1,2]], Y)
print('--- Using SKLEARN ---')
print("Intercept = ", model.intercept_)
print("Coef = ", model.coef_)
```

```
 --- Linear Regression ---
Parameters theta =
 [[-1.59080603  0.59737324  0.28829671]]
MSE  0.03168382684626477
```



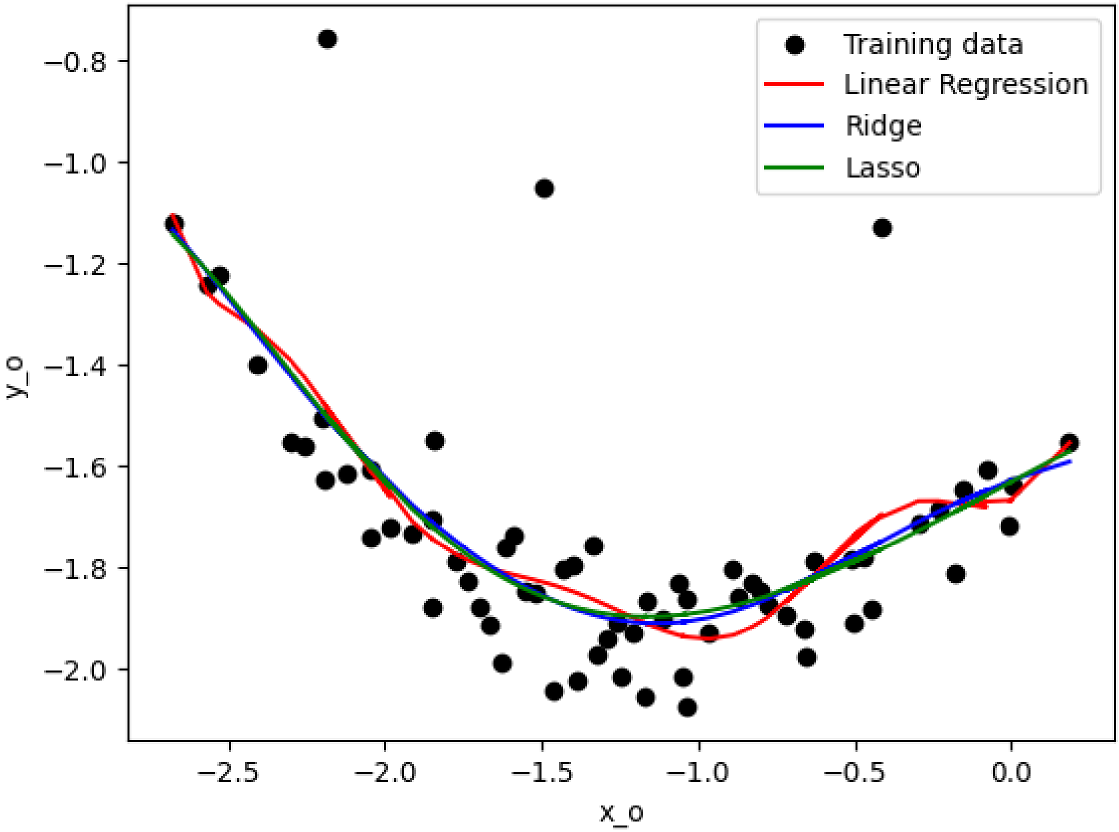

```
--- Using SKLEARN ---
Intercept =  [-1.59080603]
Coef =  [[0.59737324 0.28829671]]
```

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 [19]:
```
# Using sklearn
from sklearn import linear_model
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import Ridge, Lasso
from tabulate import tabulate

nb_degree=10
polynomial_feats = PolynomialFeatures(degree=nb_degree)
X_TRANSF = polynomial_feats.fit_transform(np.array(x_o).reshape(len(x_o),1))
Y = np.array(y_o).reshape(len(y_o),1)

#for plotting
thetas = []
mses = []
name_thetas = ['theta {}'.format(i) for i in range(nb_degree+1)]
name_thetas.append('MSE')
name_thetas = np.array(name_thetas).reshape(-1,1).T
headers = ['','Linear Regression','Ridge','Lasso']

# LINEAR REGRESSION
lin_reg = linear_model.LinearRegression()
```

```
lin_reg.fit(X_TRANSF[:,1:11], Y)
Y_predict_lr = lin_reg.predict(X_TRANSF[:,1:11])
MSE = mean_squared_error(Y,Y_predict_lr)
mses.append(MSE)
th = list(np.concatenate((lin_reg.intercept_,lin_reg.coef_[0])))
thetas.append(th)

# Using Ridge
alpha_ridge = 0.1
ridge = Ridge(alpha_ridge)
ridge.fit(X_TRANSF[:,1:11], Y)
Y_predict_ridge = ridge.predict(X_TRANSF[:,1:11])
MSE = mean_squared_error(Y,Y_predict_ridge)
mses.append(MSE)
th = list(np.concatenate((ridge.intercept_,ridge.coef_[0])))
thetas.append(th)
# Using Lasso
alpha_lasso = 0.001
lasso = Lasso(alpha_lasso)
lasso.fit(X_TRANSF[:,1:11], Y)
Y_predict_lasso = lasso.predict(X_TRANSF[:,1:11])
MSE = mean_squared_error(Y,Y_predict_lasso)
mses.append(MSE)
th = list(np.concatenate((lasso.intercept_,lasso.coef_)))
thetas.append(th)


# Plotting
plt.xlabel('x_o')
plt.ylabel('y_o')
plt.scatter(x_o,y_o,color='black',label='Training data')
plt.plot(x_o,Y_predict_lr,color='r',label='Linear Regression')
plt.plot(x_o,Y_predict_ridge,color='b',label='Ridge')
plt.plot(x_o,Y_predict_lasso,color='g',label='Lasso')
plt.legend()
plt.show()

# Printing parameters
thetas_mses = np.concatenate((np.array(thetas),np.array(mses).reshape(-1,1)),axi
thetas_mses = np.concatenate((name_thetas,thetas_mses),axis=0)
print(tabulate(np.array(thetas_mses).T,headers=headers,tablefmt='pipe', straligr
```

```
C:\Users\comas\venvpy3107\lib\site-packages\sklearn\linear_model\_coordinate_de
scent.py:631: ConvergenceWarning: Objective did not converge. You might want to
increase the number of iterations, check the scale of the features or consider
increasing regularisation. Duality gap: 1.020e+00, tolerance: 4.299e-04
  model = cd_fast.enet_coordinate_descent(
```



	Linear Regression	Ridge	Lasso
theta 0	-1.66881	-1.62883	-1.63203
theta 1	0.3067	0.227817	0.329771
theta 2	2.22726	-0.170157	-0
theta 3	1.92792	0.0121907	-0.0523157
theta 4	-16.6141	0.114436	0.021187
theta 5	-43.4748	-0.0885444	-0
theta 6	-46.5841	-0.0357167	-0.000502892
theta 7	-26.6159	0.0787454	0.000366103
theta 8	-8.47669	0.0626455	-7.64076e-05
theta 9	-1.41584	0.0171853	7.38985e-06
theta 10	-0.0962434	0.00165837	4.10852e-06
MSE	0.0301782	0.0310161	0.0312185

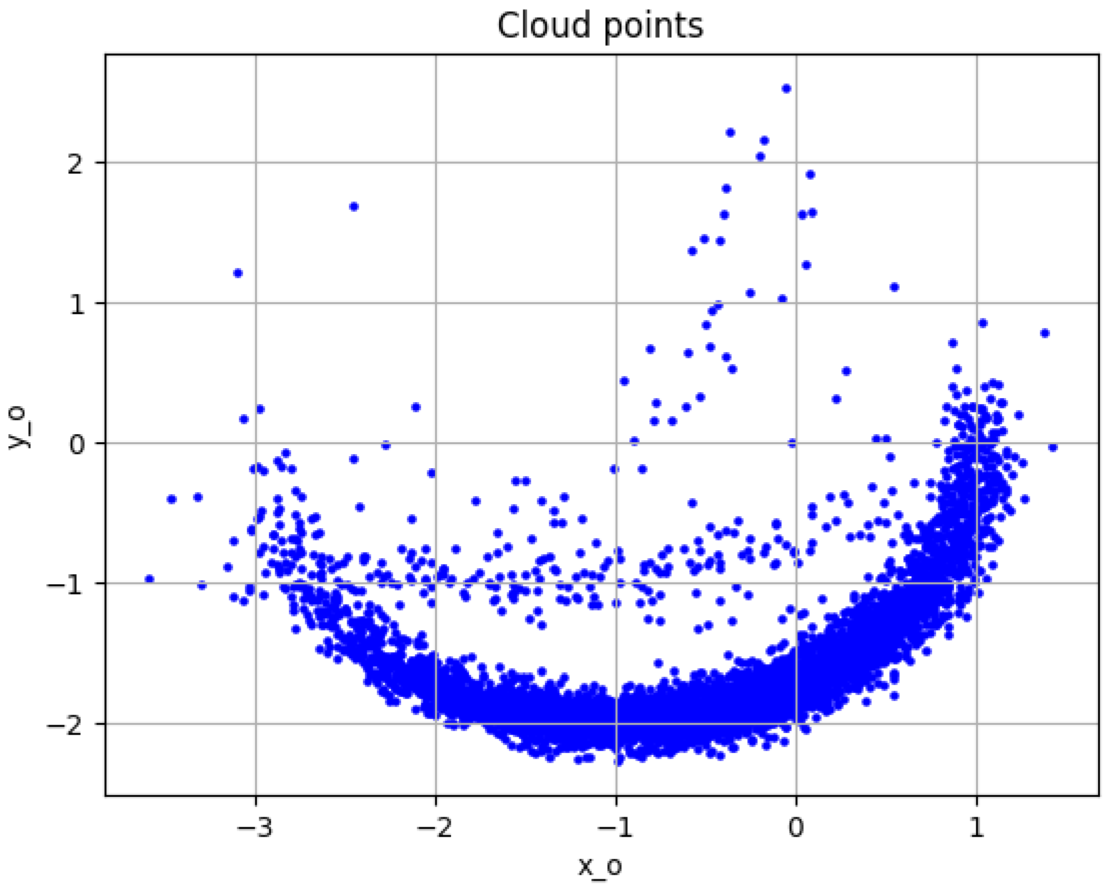
Without using normalization the regression curve is more "wavy" and tries to fit perfectly some of the training samples. Using regularization instead we obtain a smoother curve that represents better the points cloud's shape.

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 [20]:
```
import math

Lidar_range = df.iloc[:, np.arange(5,365,1)].values
angle = np.linspace(-179, 180, num=360)

#Build the cloud points in 2D plan with ALL LIDAR DATA
```

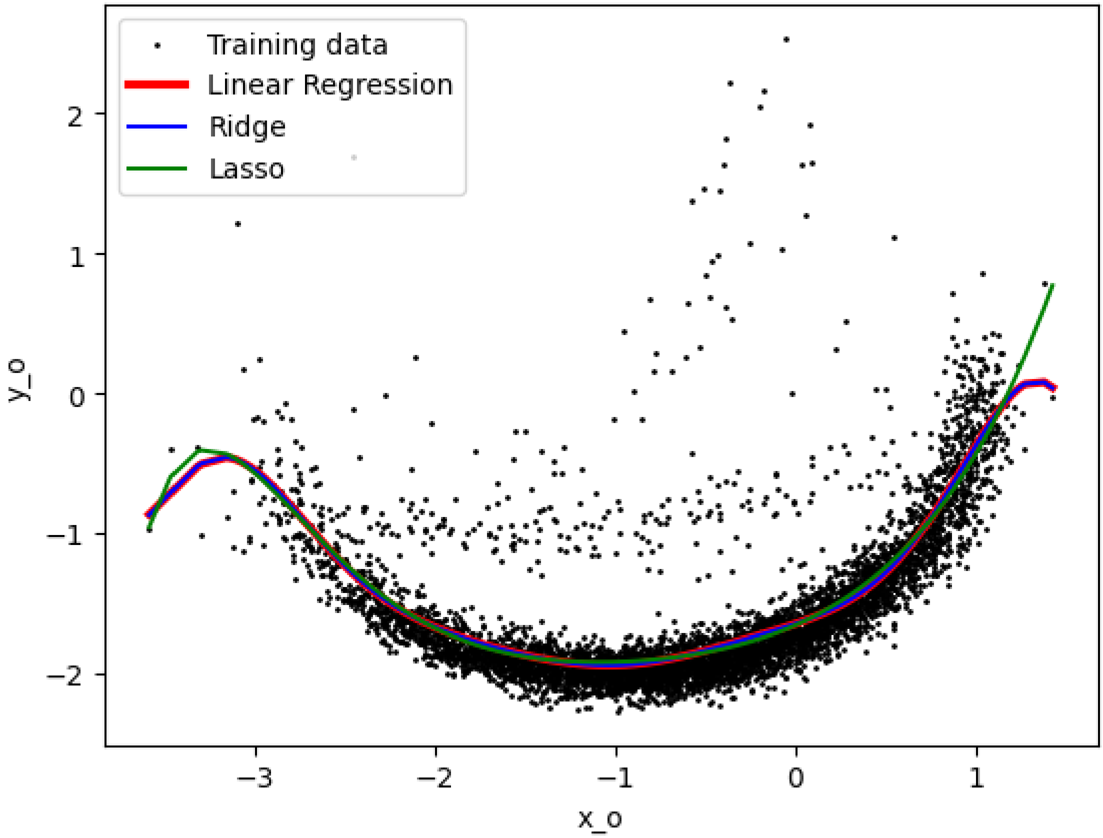
```
x_o, y_o = [], []
for t in range(len(Lidar_range)):
    for i in range(len(Lidar_range[t])):
        if Lidar_range[t][i] > 0:
            x_o.append(x_time[t]+Lidar_range[t][i]*np.cos(angle[i]*math.pi/180))
            y_o.append(y_time[t]+Lidar_range[t][i]*np.sin(angle[i]*math.pi/180))

x_o, y_o = zip(*sorted(zip(x_o, y_o)))
plt.figure()
plt.scatter(x_o, y_o, color='b',s=5)
plt.title('Cloud points')
plt.ylabel('y_o')
plt.xlabel('x_o')
plt.grid()
```



In [22]:
```
# Using sklearn
from sklearn import linear_model
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import Ridge, Lasso
from tabulate import tabulate

nb_degree=10
polynomial_feats = PolynomialFeatures(degree=nb_degree)
X_TRANSF = polynomial_feats.fit_transform(np.array(x_o).reshape(len(x_o),1))
Y = np.array(y_o).reshape(len(y_o),1)

#for plotting
thetas = []
mses = []
name_thetas = ['theta {}'.format(i) for i in range(nb_degree+1)]
name_thetas.append('MSE')
```

```
name_thetas = np.array(name_thetas).reshape(-1,1).T
headers = ['','Linear Regression','Ridge','Lasso']

# LINEAR REGRESSION
lin_reg = linear_model.LinearRegression()
lin_reg.fit(X_TRANSF[:,1:11], Y)
Y_predict_lr = lin_reg.predict(X_TRANSF[:,1:11])
MSE = mean_squared_error(Y,Y_predict_lr)
mses.append(MSE)
th = list(np.concatenate((lin_reg.intercept_,lin_reg.coef_[0])))
thetas.append(th)

# Using Ridge
alpha_ridge = 0.1
ridge = Ridge(alpha_ridge)
ridge.fit(X_TRANSF[:,1:11], Y)
Y_predict_ridge = ridge.predict(X_TRANSF[:,1:11])
MSE = mean_squared_error(Y,Y_predict_ridge)
mses.append(MSE)
th = list(np.concatenate((ridge.intercept_,ridge.coef_[0])))
thetas.append(th)
# Using Lasso
alpha_lasso = 0.001
lasso = Lasso(alpha_lasso)
lasso.fit(X_TRANSF[:,1:11], Y)
Y_predict_lasso = lasso.predict(X_TRANSF[:,1:11])
MSE = mean_squared_error(Y,Y_predict_lasso)
mses.append(MSE)
th = list(np.concatenate((lasso.intercept_,lasso.coef_)))
thetas.append(th)


# Plotting
plt.xlabel('x_o')
plt.ylabel('y_o')
plt.scatter(x_o,y_o,color='black',label='Training data',s=1)
plt.plot(x_o,Y_predict_lr,color='r',label='Linear Regression',linewidth=3)
plt.plot(x_o,Y_predict_ridge,color='b',label='Ridge')
plt.plot(x_o,Y_predict_lasso,color='g',label='Lasso')
plt.legend()
plt.show()

# Printing parameters
thetas_mses = np.concatenate((np.array(thetas),np.array(mses).reshape(-1,1)),axi
thetas_mses = np.concatenate((name_thetas,thetas_mses),axis=0)
print(tabulate(np.array(thetas_mses).T,headers=headers,tablefmt='pipe', straligr
```

```
C:\Users\comas\venvpy3107\lib\site-packages\sklearn\linear_model\_coordinate_de
scent.py:631: ConvergenceWarning: Objective did not converge. You might want to
increase the number of iterations, check the scale of the features or consider
increasing regularisation. Duality gap: 2.368e+02, tolerance: 1.558e-01
  model = cd_fast.enet_coordinate_descent(
```



	Linear Regression	Ridge	Lasso
theta 0	-1.65055	-1.65053	-1.66042
theta 1	0.42828	0.430707	0.558426
theta 2	0.240955	0.241698	0.426973
theta 3	0.515874	0.506671	0.177828
theta 4	0.487273	0.482488	0.0524003
theta 5	-0.110696	-0.102727	0.00251696
theta 6	-0.264448	-0.258329	0.00107561
theta 7	-0.0647257	-0.065618	-4.42951e-05
theta 8	0.0234899	0.0215816	1.37984e-06
theta 9	0.0118465	0.0112574	1.55435e-05
theta 10	0.00132205	0.00126428	-8.29829e-06
MSE	0.115809	0.115809	0.116536

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.

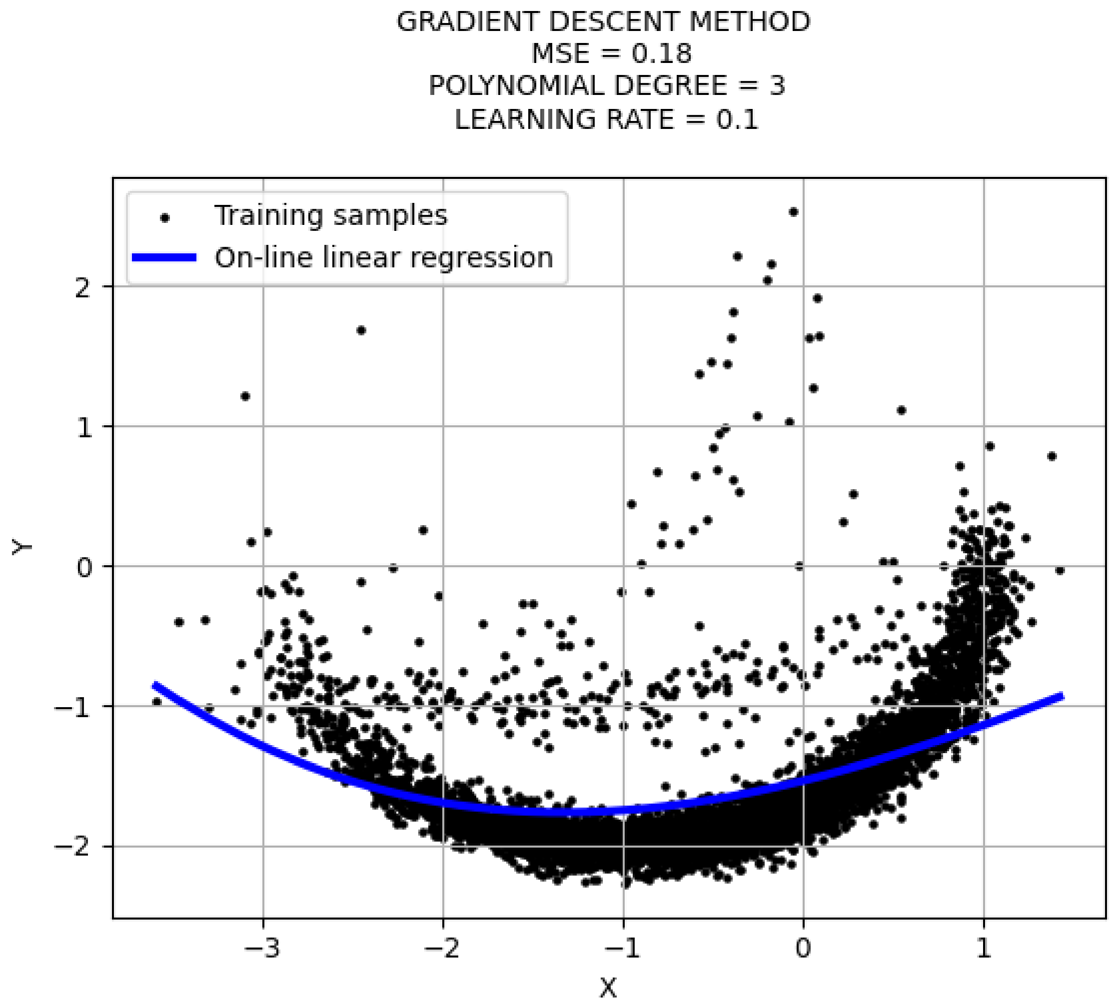
Implement this strategy and plot the results.

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

In [23]:
```
from sklearn.preprocessing import MinMaxScaler

# x_o, y_o at each snapshot time t = 0,0.5,1 ...
x_o_snap, y_o_snap = [], []
for t in range(len(Lidar_range)):
    if t%5==0:
        for i in range(len(Lidar_range[t])):
            if Lidar_range[t][i] > 0:
                x_o_snap.append(x_time[t]+Lidar_range[t][i]*np.cos(angle[i]*math
                y_o_snap.append(y_time[t]+Lidar_range[t][i]*np.sin(angle[i]*math
x_o_snap, y_o_snap = zip(*sorted(zip(x_o, y_o)))

# normalization
scaler_x = MinMaxScaler()
scaler_y = MinMaxScaler()
init_l_rate = 0.1
l_rate = init_l_rate
n_epochs = len(df) / 5
nb_degree=3

theta = np.zeros((nb_degree+1,1))

factor_low = 0.1
factor_high = 1.3

for itr in range(int(n_epochs)):
    # time istant
    t = itr * 5

    # for each istant i compute x_o and y_o
    x_o_n, y_o_n = [],[]
    for i in range(len(Lidar_range[t])):
        if Lidar_range[t][i] > 0:
            x_o_n.append(x_time[t]+Lidar_range[t][i]*np.cos(angle[i]*math.pi/180
            y_o_n.append(y_time[t]+Lidar_range[t][i]*np.sin(angle[i]*math.pi/180

    # I get X and Y matrixes and i normalize them
    polynomial_feats = PolynomialFeatures(degree=nb_degree)
    X_TRANSF = scaler_x.fit_transform(polynomial_feats.fit_transform(np.array(x_
    Y = scaler_y.fit_transform( np.array(y_o_n).reshape(len(y_o_n),1))
    # I compute theta
    theta_base = theta
    Y_predict = X_TRANSF @ theta_base
    Y_residuals = np.subtract(Y_predict,Y)
    Loss = np.sum((Y_residuals**2))
    #grad_loss = 2 * np.transpose(X_TRANSF) @ Y_predict - 2 * np.transpose(X_TRA
    grad_loss = -2 * X_TRANSF.T @ (Y - X_TRANSF @ theta)
    grad_loss = np.reshape(grad_loss, (X_TRANSF.shape[1],1))
    theta_new= theta_base - l_rate*grad_loss

    # Compute the new Loss to adjust learning rate
    Y_pred_new = X_TRANSF @ theta_new
    Y_residuals_new = np.subtract(Y_pred_new,Y)
    Loss_new = np.sum((Y_residuals_new**2))

    if(Loss_new >= Loss) :
        # if the loss with computed theta is >= than the previous loss
        # we lower the learning rate
        while Loss_new >= Loss:
```

```
            l_rate *= factor_low
            # compute new theta with new lr
            theta_new = theta_base - l_rate*grad_loss
            Y_pred_new = X_TRANSF @ theta_new
            Y_residuals_new = np.subtract(Y_pred_new,Y)
            # Compute new loss
            Loss_new = np.sum((Y_residuals_new**2))
            # While the new loss is >= than the previous one
            # we continue lowering the l_rate
    else :
        # if the new Loss is < previous loss
        while True :
            # we increase l_rate
            l_rate *= factor_high
            # compute a new theta
            theta_new_new = theta_base - l_rate*grad_loss
            Y_pred_new_new = X_TRANSF @ theta_new_new
            Y_residuals_new_new = np.subtract(Y_pred_new_new,Y)
            # compute a new loss
            Loss_new_new = np.sum((Y_residuals_new_new**2))
            # if the new loss < of the previous one we update theta
            # and we continue to increase theta till the new loss
            # is >= of the previous one
            if Loss_new_new >= Loss_new : break
            theta_new , Loss_new = theta_new_new, Loss_new_new

    theta = theta_new
    Loss = Loss_new
    #print("Itr ",itr," Loss ",Loss)



X_TRANSF = scaler_x.fit_transform(polynomial_feats.fit_transform(np.array(x_o_sr
Y = np.array(y_o_snap).reshape(len(y_o_snap),1)
Y_pred = scaler_y.inverse_transform(X_TRANSF @ theta)
MSE = mean_squared_error(Y,Y_pred)

plt.scatter(x_o_snap, Y, color="black",s=5,label='Training samples')
plt.plot(x_o_snap, Y_pred,  color="blue", linewidth=3,label='On-line linear regr
plt.grid()

print('THETA', theta.T)

title = 'MSE = {}\n'.format(round(MSE,2))
degree = 'POLYNOMIAL DEGREE = {}\n'.format(nb_degree)
lr = 'LEARNING RATE = {}\n'.format(init_l_rate)
plt.title("GRADIENT DESCENT METHOD \n " + title + degree + lr, fontsize=10)
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()
```

```
THETA [[ 0.          0.46226309  0.34622031 -0.19347677]]
```


GRADIENT DESCENT METHOD
MSE = 0.18
POLYNOMIAL DEGREE = 3
LEARNING RATE = 0.1
Training samples
On-line linear regression
Y
X
2
1
0
−1
−2
−3
−2
−1
0
1