In [ ]:
```
import cv2 as cv
import numpy as np
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import math
from math import nan
from IPython.display import Image
from matplotlib import cm
```

Project 1.

Patrik Dominik Pördi

Problem 1.

1.

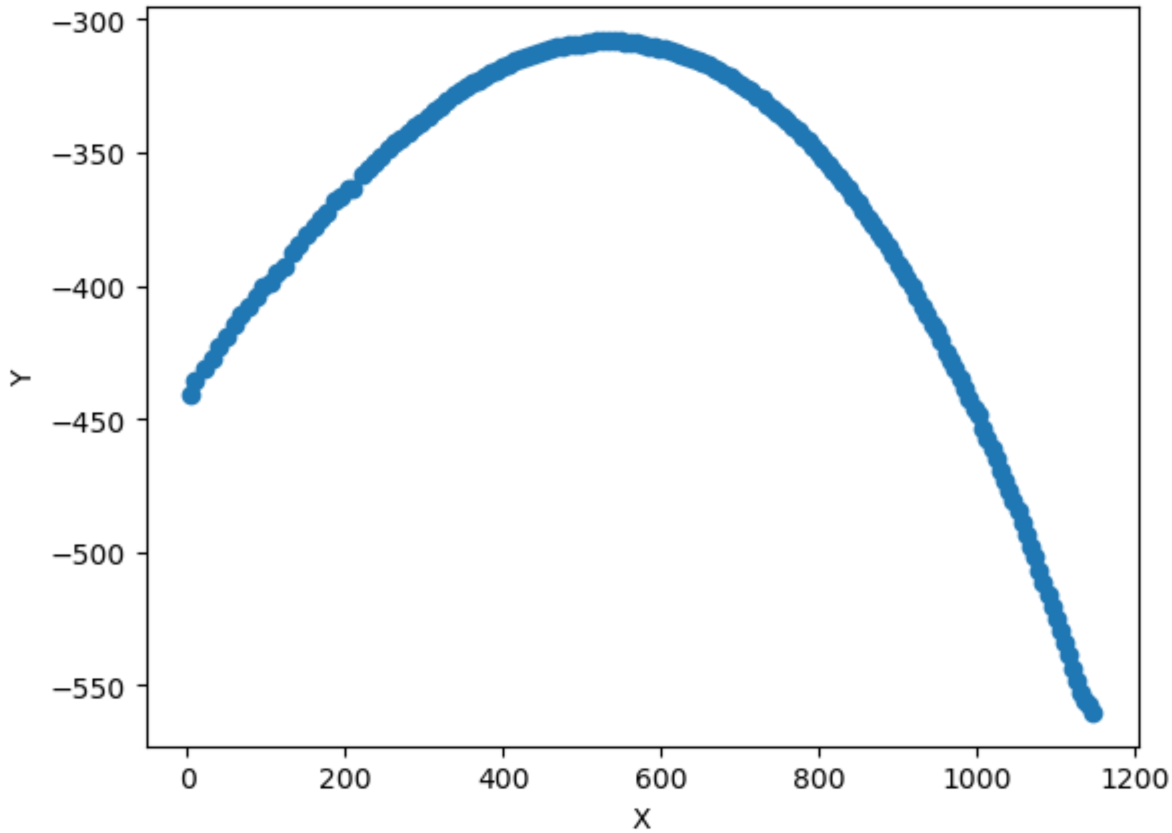
First the video an the corresponding frames are inserted, after that using various image operations the path of the ball was extracted, using standard mean calculations the centeroid of the ball can be found on each frame and plotted. For filtering the movement of the ball the treshold was iterated.

In [ ]:
```
# Create a video capture object, in this case we are reading the video from
vid_capture = cv.VideoCapture('/home/pordipatrik/Perception/ball.mov')
if (vid_capture.isOpened() == False):
  print("Error opening the video file")
# Read fps and frame count
else:
  # Get frame rate information
  # You can replace 5 with CAP_PROP_FPS as well, they are enumerations
  fps = vid_capture.get(5)
  print('Frames per second : ', fps,'FPS')

  # Get frame count
  # You can replace 7 with CAP_PROP_FRAME_COUNT as well, they are enumeratio
  frame_count = vid_capture.get(7)
  print('Frame count : ', frame_count)

# Creating two lists to store the path of the ball
x=[]
y=[]
while(vid_capture.isOpened()):
    # vid_capture.read() methods returns a tuple, first element is a bool
    # and the second is frame
    ret, frame = vid_capture.read()
    if ret == True:

        # Bluring, conversion to hsv, creating limits, creating the mask, cr
        blurred = cv.GaussianBlur(frame, (11, 11), 0)
        hsv=cv.cvtColor(blurred, cv.COLOR_BGR2HSV)
        lower_red=np.array([0,170,90])
        upper_red=np.array([5,255,255])
        mask=cv.inRange(hsv,lower_red, upper_red)
        res=cv.bitwise_and(frame,frame,mask=mask)
        cv.imshow('mask', mask)
        cv.imshow('frame', frame)
        cv.imshow('res',res)

        # Extracting the points that represents the path of the ball, and ca
        yis, xis = np.nonzero(mask)
        a=xis.mean()
        b=yis.mean()
        x.append(a)
        y.append(-b)

        key = cv.waitKey(30)
        if key == ord('q'):
            break
    else:
        break

# Release the video capture object
vid_capture.release()
cv.destroyAllWindows()

#Plotting the path of the ball
plt.scatter(x,y)
plt.xlabel("X")
```

```
plt.ylabel("Y")
plt.show()
```

```
Frames per second :  30.067789822945567 FPS
Frame count :  438.0
```

```
/tmp/ipykernel_8382/1907966289.py:39: RuntimeWarning: Mean of empty slice.
  a=xis.mean()
/usr/lib/python3/dist-packages/numpy/core/_methods.py:161: RuntimeWarning: invalid value encountered in double_scalars
  ret = ret.dtype.type(ret / rcount)
/tmp/ipykernel_8382/1907966289.py:40: RuntimeWarning: Mean of empty slice.
  b=yis.mean()
```



2.

Using the system of equations(refer to the link and the picture below) we can find a, b, c and write up the equation of the parabola.

https://www.efunda.com/math/leastsquares/lstsqr2dcurve.cfm

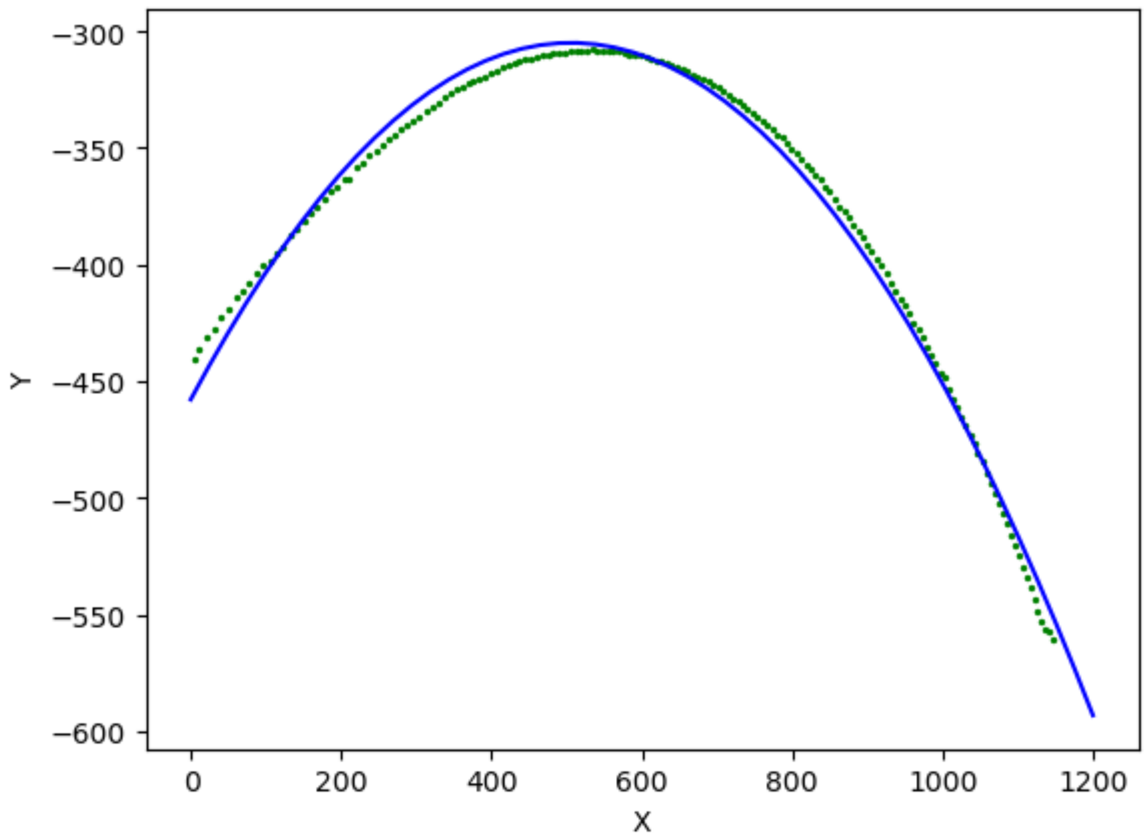
In [ ]:
```
Image("p_1.png")
```

Out[ ]:

$$\begin{cases} \sum_{i=1}^{n} y_i = a\sum_{i=1}^{n} 1 + b\sum_{i=1}^{n} x_i + c\sum_{i=1}^{n} x_i^{\ 2} \\ \sum_{i=1}^{n} x_i y_i = a\sum_{i=1}^{n} x_i + b\sum_{i=1}^{n} x_i^{\ 2} + c\sum_{i=1}^{n} x_i^{\ 3} \\ \sum_{i=1}^{n} x_i^{\ 2} y_i = a\sum_{i=1}^{n} x_i^{\ 2} + b\sum_{i=1}^{n} x_i^{\ 3} + c\sum_{i=1}^{n} x_i^{\ 4} \end{cases}$$

In [ ]:

```
# First we filter out the useless data
x=[number for number in x if str(number)!="nan"]
y=[number for number in y if str(number)!="nan"]

## Creating the required parts for the system of equations
second=[number**2 for number in x]
third=[number**3 for number in x]
fourth=[number**4 for number in x]
xy=[]
for i in range(0,len(x)):
    xy.append(x[i]*y[i])
xxy=[]
for i in range(0,len(x)):
    xxy.append(second[i]*y[i])

# Creating the matrices for the system of equations
A=np.array([[len(x),np.sum(x),np.sum(second)],
[np.sum(x),np.sum(second),np.sum(third)],
[sum(second),np.sum(third),np.sum(fourth)]])
B=np.array([np.sum(y),np.sum(xy),np.sum(xxy)])
h=np.linalg.solve(A,B)
def f(x):
    return h[0]+h[1]*x+h[2]*x**2
g=np.linspace(0,1200,50)
plt.scatter(x,y, Color="g",s=2)
plt.plot(g,f(g), Color="b")
plt.xlabel("X")
plt.ylabel("Y")
# Plotting the curve
plt.show()
print("Equation of the curve: x^2*" ,h[2], "x*",h[1],"+",h[0])
```




```
Equation of the curve: x^2* -0.000597895258757051 x* 0.6046909775458277 + -
457.7840595912796
```

3.

Using the equation of the parabola and the given y-value the x-value can be approximated

In [ ]:
```
# Finding the solutions of the equation using the given information
x_1=(-h[1]+np.sqrt(h[1]**2-4*(h[2]*(h[0]-y[0]+300))))/(2*h[2])
x_2=(-h[1]-np.sqrt(h[1]**2-4*(h[2]*(h[0]-y[0]+300))))/(2*h[2])
print(max(x_1,x_2),",",y[0]-300)
```

```
1359.3495290023293 , -740.6071428571429
```

Problems faced: Finding the right limits for generating the mask without the noise, getting rid of the nan values from the path, calculating the coordinates upside down.

Problem 2.

1.

a.

The covariance matrix can be calculated using the formula below, after that we can find the eigenvectors and eigenvalues using inbuilt functions.

https://towardsdatascience.com/5-things-you-should-know-about-covariance-26b12a0516f1

In [ ]:
```
Image("p_2.png")
```

Out[ ]:

$$\begin{array}{c|ccc} & x & y & z \\ \hline x & & & \\ y & & \left[\begin{array}{ccc} var(x) & cov(x,y) & cov(x,z) \\ cov(x,y) & var(y) & cov(y,z) \\ cov(x,z) & cov(y,z) & var(z) \end{array}\right] & \\ z & & & \end{array}$$

In [ ]:
```
#The texts have to be inserted and the x,y,z coordinates are stored separate
A=np.loadtxt("pc1.csv",delimiter=",",dtype=float)
B=np.loadtxt("pc2.csv",delimiter=",",dtype=float)
x=A[:,0]
y=A[:,1]
z=A[:,2]


# Means and the deifference between actual values and means are calculated
x_m=np.mean(x, axis=0).reshape(-1,1)
y_m=np.mean(y, axis=0).reshape(-1,1)
z_m=np.mean(z, axis=0).reshape(-1,1)
x_d=x-x_m
y_d=y-y_m
z_d=z-z_m

# Variancies are calculated
var_x=(x_d@x_d.T)/len(x)
var_y=(y_d@y_d.T)/len(x)
var_z=(z_d@z_d.T)/len(x)

# Covariancies are calculated
cov_xy=(x_d@y_d.T)/len(x)
cov_yz=(y_d@z_d.T)/len(x)
cov_xz=(x_d@z_d.T)/len(x)
cov_m=np.array([[var_x[0,0],cov_xy[0,0],cov_xz[0,0]],[cov_xy[0,0],var_y[0,0]
print("The covariance matrix is the following:")
print(cov_m)
```

```
The covariance matrix is the following:
[[ 33.6375584   -0.82238647 -11.3563684 ]
 [ -0.82238647  35.07487427 -23.15827057]
 [-11.3563684  -23.15827057  20.5588948 ]]
```

b.

The eigenvector of the covariance matrix with the largest eigenvalues always points into the direction of the largest variance of the data, and the magnitude of this vector equals the corresponding eigenvalue. The eigenvector corresponding to the smalest eigenvalue will represent the surface normal.

In [ ]:
```
# Calculating the eigenvalues and eigenvectors of the covariance matrix and
E_val,E_vec=np.linalg.eig(cov_m)
print("Direction:",E_vec[np.where(min(E_val[0],E_val[1],E_val[2]))[0][0]])
print("Magnitude:",np.linalg.norm(E_vec[np.where(min(E_val[0],E_val[1],E_val
```

```
Direction: [ 0.28616428  0.90682723 -0.30947435]
Magnitude: 0.9999999999999999
```

Problems faced: Reshaping the matrices, figuring out which eigenvector is needed.

2.

a.

The standared least square method can be calculated using the equations below. The data has to be restructured, and then the parameters of the plane can be found.

In [ ]:
```
Image("p_3.png")
```

Out[ ]:

$$\begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ & \cdots & \\ x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} z_0 \\ z_1 \\ \cdots \\ z_n \end{bmatrix}$$

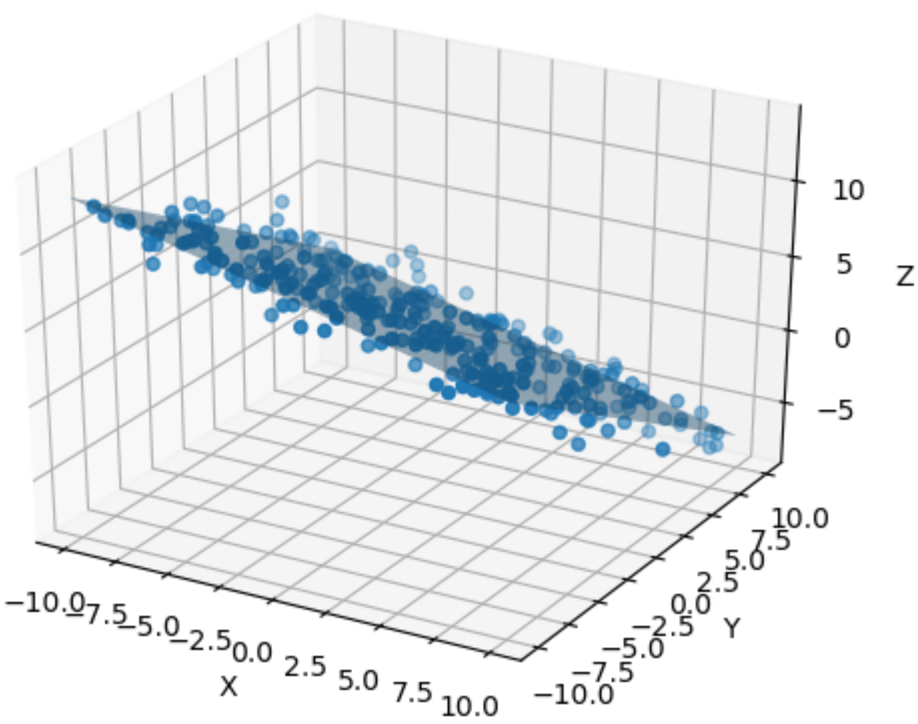
In [ ]:
```
Image("p_4.png")
```

Out[ ]:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = (A^T A)^{-1} A^T B$$

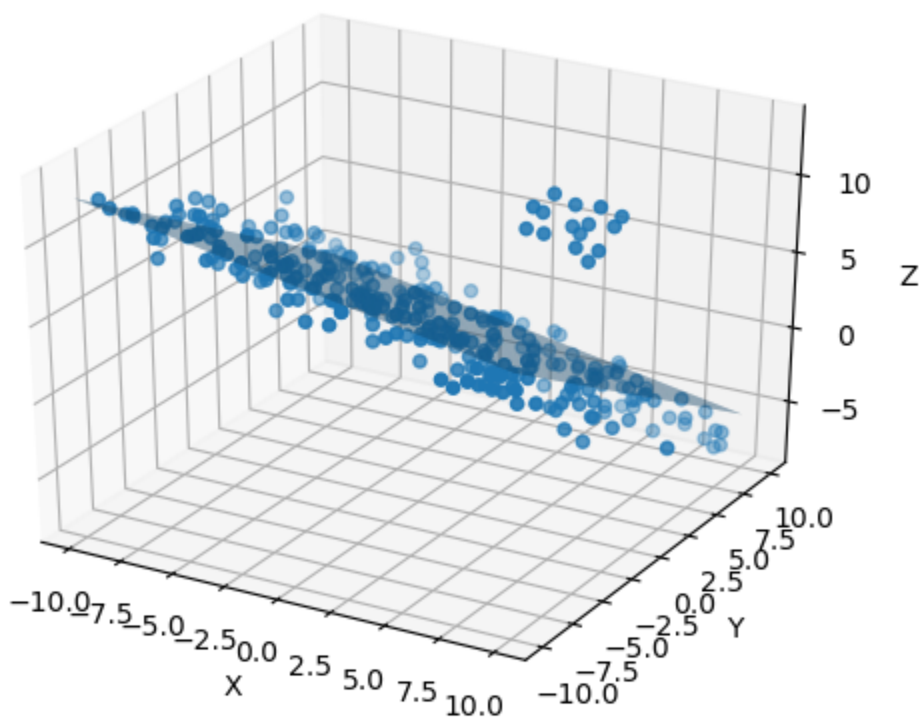
pc1.csv

In [ ]:
```
# The texts have to be inserted and the x,y,z coordinates are stored separat
A=np.loadtxt("pc1.csv",delimiter=",",dtype=float)
x=A[:,0]
y=A[:,1]
z=A[:,2]

# System matrix is made and the a,b,c values for the plane equation are calc
M=np.column_stack((x,y, np.ones(len(A))))
a,b,c= np.linalg.inv(M.T@ M) @ M.T @z

# Creating the meshgrid for ploting the plane and calculating the correspond
x_f, y_f=np.meshgrid(np.linspace(np.min(x),np.max(x),1000),np.linspace(np.mi
z_f=a*x_f+b*y_f+c

# Ploting the points and corresponding plane
fig=plt.figure()
ax=fig.add_subplot(111,projection='3d')
ax.scatter(x,y,z)
ax.plot_surface(x_f,y_f,z_f,alpha=0.5)
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
plt.show()
```



pc2.csv

In [ ]:
```
# The texts have to be inserted and the x,y,z coordinates are stored separat
A=np.loadtxt("pc2.csv",delimiter=",",dtype=float)
x=A[:,0]
y=A[:,1]
z=A[:,2]

# System matrix is made and the a,b,c values for the plane equation are calc
M=np.column_stack((x,y, np.ones(len(A))))
a,b,c= np.linalg.inv(M.T@ M) @ M.T @z

# Creating the meshgrid for ploting the plane and calculating the correspond
x_f, y_f=np.meshgrid(np.linspace(np.min(x),np.max(x),1000),np.linspace(np.mi
z_f=a*x_f+b*y_f+c

# Ploting the points and corresponding plane
fig=plt.figure()
ax=fig.add_subplot(111,projection='3d')
ax.scatter(x,y,z)
ax.plot_surface(x_f,y_f,z_f,alpha=0.5)
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
plt.show()
```



The total least square method can be calculated using the equations below. The data has to be restructured, and then the parameters of the plane can be found.

In [ ]:
```
Image("p_6.png")
```

Out[ ]: Solution to $(U^TU)N = 0$, subject to $||N||^2 = 1$: eigenvector of U^TU associated with the smallest eigenvalue (least squares solution to *homogeneous linear system* $UN = 0$)

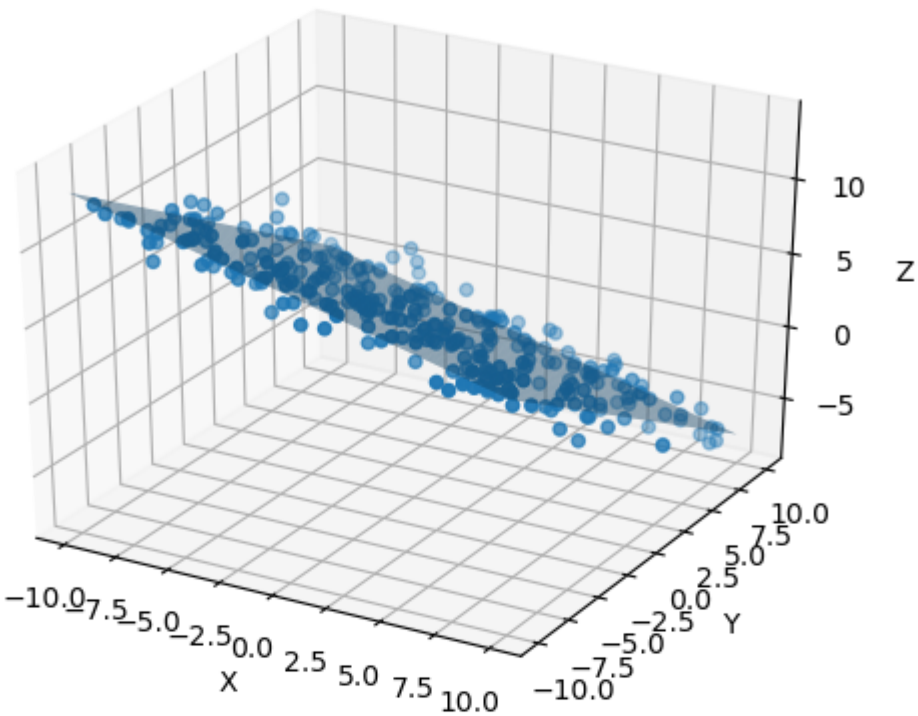
In [ ]:
```
Image("p_5.png")
```

Out[ ]:

$$U = \begin{bmatrix} x_1 - \bar{x} & y_1 - \bar{y} \\ \vdots & \vdots \\ x_n - \bar{x} & y_n - \bar{y} \end{bmatrix} \quad U^TU = \begin{bmatrix} \sum_{i=1}^{n} (x_i - \bar{x})^2 & \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y}) \\ \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y}) & \sum_{i=1}^{n} (y_i - \bar{y})^2 \end{bmatrix}$$

pc1.csv

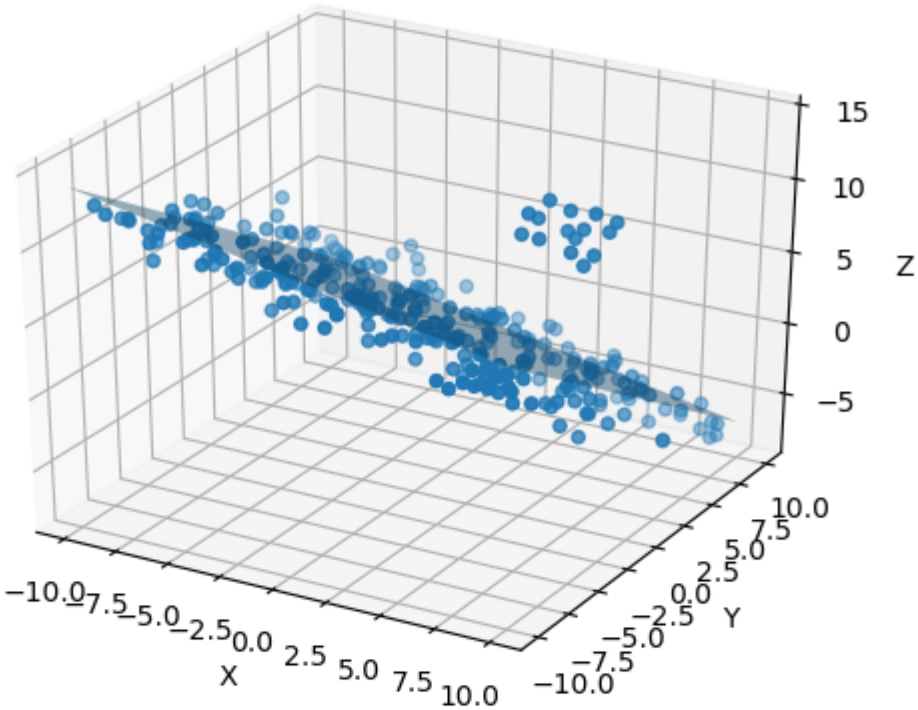
In [ ]:
```
# The texts have to be inserted and the x,y,z coordinates are stored separat
A=np.loadtxt("pc1.csv",delimiter=",",dtype=float)
x=A[:,0]
y=A[:,1]
z=A[:,2]

# Calculating U and its eigenvalues and the eigenvector corresponding to the
cent_of_mass = np.mean(A,axis=0)
Am=A-cent_of_mass
U = Am.T@Am
E_val, E_vec = np.linalg.eig(U)
E_vec_s = E_vec[:,np.argmin(E_val)]

# Creating the meshgrid for ploting the plane and calculating the correspond
x_f, y_f=np.meshgrid(np.linspace(np.min(x),np.max(x),1000),np.linspace(np.mi
d=E_vec_s[0]*cent_of_mass[0]+E_vec_s[1]*cent_of_mass[1]+E_vec_s[2]*cent_of_m
z_f = (-E_vec_s[0]*x_f - E_vec_s[1]*y_f + d)/E_vec_s[2]

# Ploting the points and corresponding plane
fig1 = plt.figure()
ax = fig1.add_subplot(111, projection='3d')
ax.scatter(x, y, z)
ax.plot_surface(x_f, y_f, z_f, alpha= 0.5)
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
plt.show()
print(E_vec_s[0],E_vec_s[1],E_vec_s[2],d)
```




```
0.2861642761209515 0.539712338307339 0.7917200256094298 2.5344641945425836
```

pc2.csv

In [ ]:
```
# The texts have to be inserted and the x,y,z coordinates are stored separat
A=np.loadtxt("pc2.csv",delimiter=",",dtype=float)
x=A[:,0]
y=A[:,1]
z=A[:,2]

# Calculating U and its eigenvalues and the eigenvector corresponding to the
cent_of_mass = np.mean(A,axis=0)
Am=A-cent_of_mass
U = Am.T@Am
E_val, E_vec = np.linalg.eig(U)
E_vec_s = E_vec[:,np.argmin(E_val)]

# Creating the meshgrid for ploting the plane and calculating the correspond
x_f, y_f=np.meshgrid(np.linspace(np.min(x),np.max(x),1000),np.linspace(np.mi
d=E_vec_s[0]*cent_of_mass[0]+E_vec_s[1]*cent_of_mass[1]+E_vec_s[2]*cent_of_m
z_f = (-E_vec_s[0]*x_f - E_vec_s[1]*y_f + d)/E_vec_s[2]

# Ploting the points and corresponding plane
fig1 = plt.figure()
ax = fig1.add_subplot(111, projection='3d')
ax.scatter(x, y, z)
ax.plot_surface(x_f, y_f, z_f, alpha= 0.5)
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")

plt.show()
```



My conclusion is that both the least square and the total least square methodes work pretty well when we need to fit a model to the given data set, except when there are outliers, in that case both have problems. In general, the solution of the total least square seems slightly more accurate. I also have to admit a question, we were told that total least square is more computationally intensive than the simple one, but for me calculating eigenvectors seems less intensive than inverting matrices.

Problems faced: Creating the meshgrids, calculating d and z correctly

b.

For implementing the RANSAC algorithm we had the following guidelines: Choose a small subset of points uniformly at random, Fit a model to that subset, Find all remaining points that are “close” to the model and reject the rest as outliers, Do this many times and choose the best model.

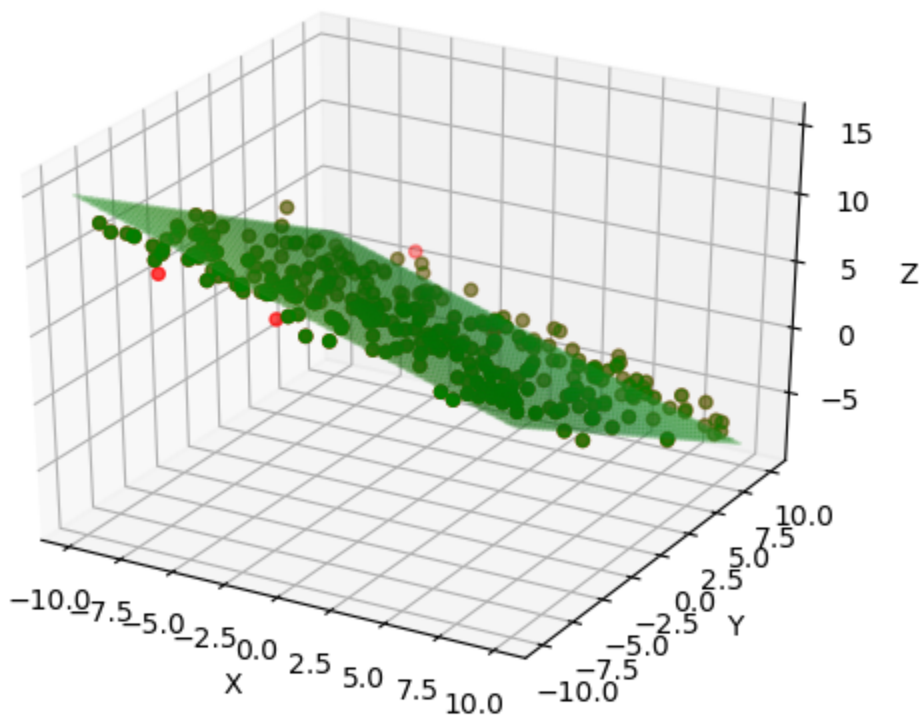
In this case 3 points were selected and the corresponding plane was constructed, after the treshold, and the acceptable distance were iterated to find the result that fully separates the out/in-liers. The algorithm runs for a choosen number of iterations.

pc1.csv

In [ ]:
```
# The texts have to be inserted and the x,y,z coordinates are stored separat
A=np.loadtxt("pc1.csv",delimiter=",",dtype=float)
x=A[:,0]
y=A[:,1]
z=A[:,2]

# Function for constructing the plane
def plane(sample):
    v1 = np.array(sample[1])-np.array(sample[0])
    v2 = np.array(sample[2])-np.array(sample[0])
    a,b,c = np.cross(v1,v2)
    l = -np.dot(np.cross(v1,v2),np.array(sample[0]))
    return a,b,c,l

# Function for calculating the distance of a point from a plane
def distance(a,b,c,l,x,y,z):
    distance=(abs(a*x+b*y+c*z+l))/(math.sqrt(a*a+b*b+c*c))
    return distance

# Function for RANSAC
def ransac (Data):
    #Initializing the variables,3 points,200 iterations,2 distance, 90%tresh
    n,k,d,l = 3,200,2,len(Data)*0.9
    t_plane = None
    t_points = []
    for i in range(k):
        #Creating a plane based on random points
        plane_temp = plane(Data[np.random.choice(len(Data), n, replace=False
        points_in=[]
        # Checking wheteher the distance is less than the limit, if yes the
        for points in Data:
            dist = distance(plane_temp[0],plane_temp[1], plane_temp[2], plan
            if(dist<d):
                points_in.append(points)
        # Checking if we are in the first iteration or the current iteratior
        in_q = len(points_in)
        if(in_q>=l) and (t_plane is None or in_q>len(t_points)):
            t_plane = plane(points_in)
            t_points = points_in
            print(in_q/len(Data))

    # Saving the plane and the points as np.array to be able to return it
    t_plane_np = np.array(t_plane)
    t_points_np = np.array(t_points)

    return t_plane_np, t_points_np

# Extracting the plane and the inlier points
r_plane, r_points = ransac(A)

# Creating the meshgrid for ploting the plane and calculating the correspond
x_f, y_f=np.meshgrid(np.linspace(np.min(x),np.max(x),1000),np.linspace(np.mi
z = (-r_plane[0] * x_f - r_plane[1] * y_f - r_plane[3]) / r_plane[2]

# Ploting the points and corresponding plane
fig = plt.figure()
```

```
ax = fig.add_subplot(111, projection='3d')
ax.scatter(A[:,0], A[:,1], A[:,2], c='r', marker='o')
ax.scatter(r_points[:,0], r_points[:,1], r_points[:,2], c='g', marker='o')

ax.plot_surface(x_f, y_f, z,alpha= 0.2, Color='g', cmap=cm.twilight)
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
plt.show()
```

```
0.9766666666666667
0.9833333333333333
0.9866666666666667
```

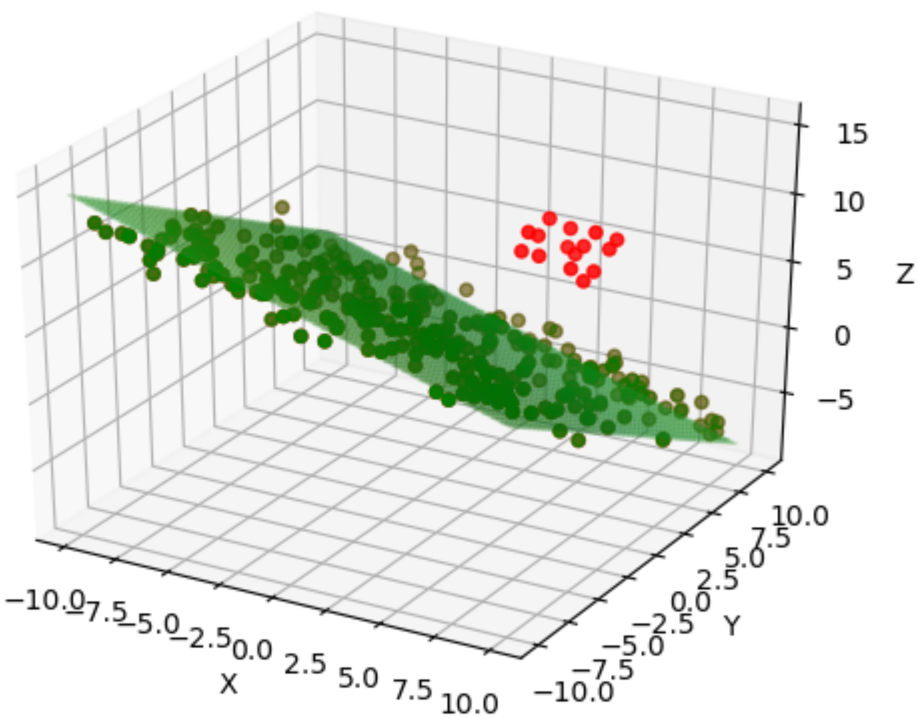


pc2.csv

In [ ]:
```
# The texts have to be inserted and the x,y,z coordinates are stored separat
A=np.loadtxt("pc2.csv",delimiter=",",dtype=float)
x=A[:,0]
y=A[:,1]
z=A[:,2]

# Function for constructing the plane
def plane(sample):
    v1 = np.array(sample[1])-np.array(sample[0])
    v2 = np.array(sample[2])-np.array(sample[0])
    a,b,c = np.cross(v1,v2)
    l = -np.dot(np.cross(v1,v2),np.array(sample[0]))
    return a,b,c,l

# Function for calculating the distance of a point from a plane
def distance(a,b,c,l,x,y,z):
    distance=(abs(a*x+b*y+c*z+l))/(math.sqrt(a*a+b*b+c*c))
    return distance

# Function for RANSAC
def ransac (Data):
    #Initializing the variables,3 points,200 iterations,2.4 distance, 90%tre
    n,k,d,l = 3,200,2.4,len(Data)*0.9
    t_plane = None
    t_points = []
    for i in range(k):
        #Creating a plane based on random points
        plane_temp = plane(Data[np.random.choice(len(Data), n, replace=False
        points_in=[]
        # Checking wheteher the distance is less than the limit, if yes the
        for points in Data:
            dist = distance(plane_temp[0],plane_temp[1], plane_temp[2], plan
            if(dist<d):
                points_in.append(points)
        # Checking if we are in the first iteration or the current iteration
        in_q = len(points_in)
        if(in_q>=l) and (t_plane is None or in_q>len(t_points)):
            t_plane = plane(points_in)
            t_points = points_in
            print(in_q/len(Data))

    # Saving the plane and the points as np.array to be able to return it
    t_plane_np = np.array(t_plane)
    t_points_np = np.array(t_points)

    return t_plane_np, t_points_np

# Extracting the plane and the inlier points
r_plane, r_points = ransac(A)

# Creating the meshgrid for ploting the plane and calculating the correspond
x_f, y_f=np.meshgrid(np.linspace(np.min(x),np.max(x),1000),np.linspace(np.mi
z = (-r_plane[0] * x_f - r_plane[1] * y_f - r_plane[3]) / r_plane[2]

# Ploting the points and corresponding plane
fig = plt.figure()
```

```
ax = fig.add_subplot(111, projection='3d')
ax.scatter(A[:,0], A[:,1], A[:,2], c='r', marker='o')
ax.scatter(r_points[:,0], r_points[:,1], r_points[:,2], c='g', marker='o')

ax.plot_surface(x_f, y_f, z,alpha= 0.2, Color='g', cmap=cm.twilight)
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
plt.show()
```

```
0.9428571428571428
0.946031746031746
0.9523809523809523
```



According to the results RANSAC seems the best methode to me for detecting outliers and fiting models based on the separation. On the other hand, finding the right variables (iterations,distance, treshold) can be a bit hard when it comes to more complex data. All in all, if there are outliers, for instance data set 2 we should use RANSAC, while in other cases simplier methodes like least square and total least square can provide almost the same results.

Problems faced: Creating the RANSAC function was challenging, and finding the right number of iterations, distance and treshold, storing the inliers separately and plotting everything correctly