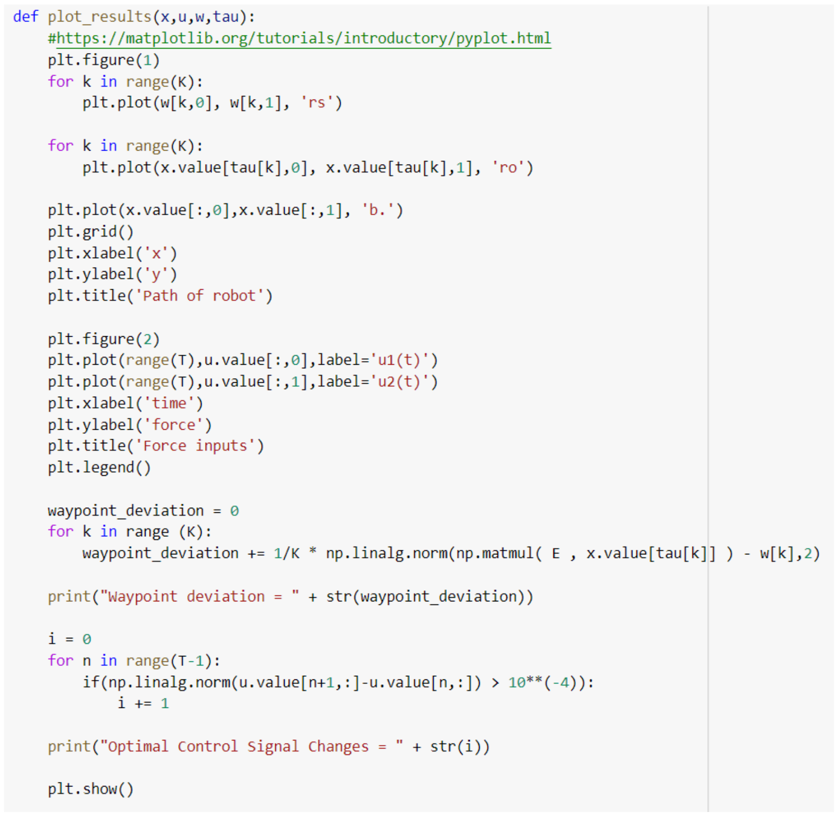
Optimization and Algorithms
Project report

Group 14
António Pereira - 90019, Bernardo Taveira - 90031,
Diogo Alves - 86980 e Miguel Amaral - 90150

Contents

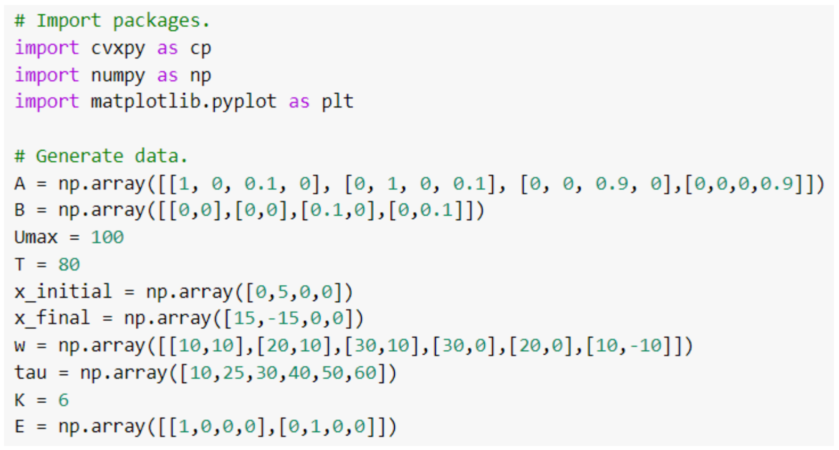
1 Part 1



```python
def plot_results(x,u,w,tau):
    #https://matplotlib.org/tutorials/introductory/pyplot.html
    plt.figure(1)
    for k in range(K):
        plt.plot(w[k,0], w[k,1], 'rs')

    for k in range(K):
        plt.plot(x.value[tau[k],0], x.value[tau[k],1], 'ro')

    plt.plot(x.value[:,0],x.value[:,1], 'b.')
    plt.grid()
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Path of robot')

    plt.figure(2)
    plt.plot(range(T),u.value[:,0],label='u1(t)')
    plt.plot(range(T),u.value[:,1],label='u2(t)')
    plt.xlabel('time')
    plt.ylabel('force')
    plt.title('Force inputs')
    plt.legend()

    waypoint_deviation = 0
    for k in range (K):
        waypoint_deviation += 1/K * np.linalg.norm(np.matmul( E , x.value[tau[k]] ) - w[k],2)

    print("Waypoint deviation = " + str(waypoint_deviation))

    i = 0
    for n in range(T-1):
        if(np.linalg.norm(u.value[n+1,:]-u.value[n,:]) > 10**(-4)):
            i += 1

    print("Optimal Control Signal Changes = " + str(i))

    plt.show()
```

Figure 1: Plotting function used throughout the first part of the project

1.1 Task 1

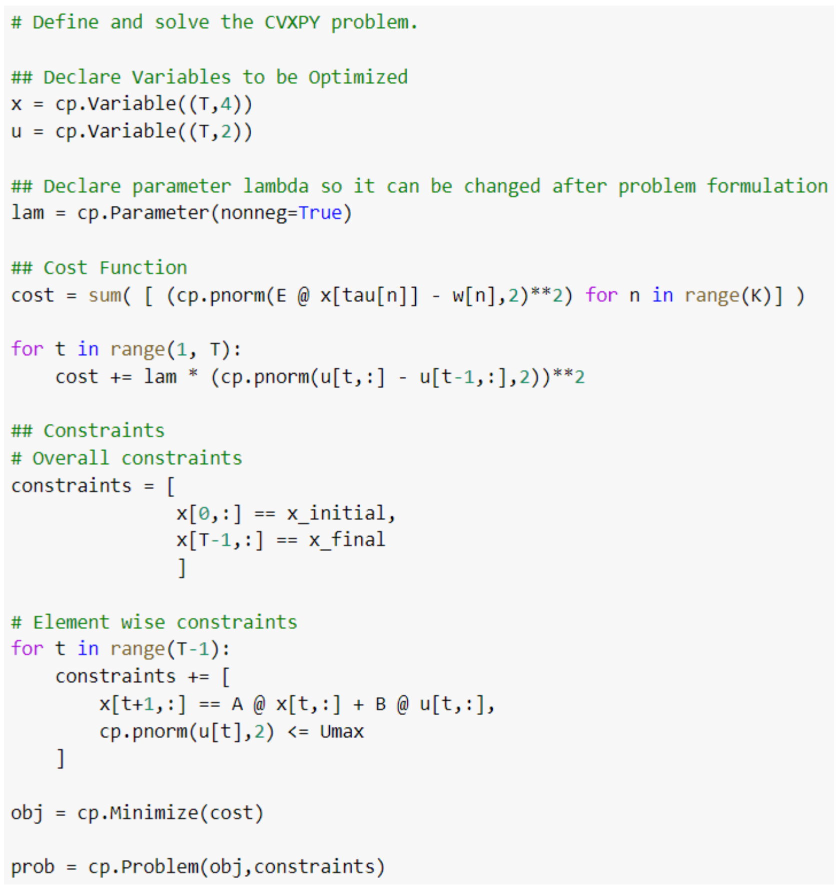


```python
# Import packages.
import cvxpy as cp
import numpy as np
import matplotlib.pyplot as plt

# Generate data.
A = np.array([[1, 0, 0.1, 0], [0, 1, 0, 0.1], [0, 0, 0.9, 0],[0,0,0,0.9]])
B = np.array([[0,0],[0,0],[0.1,0],[0,0.1]])
Umax = 100
T = 80
x_initial = np.array([0,5,0,0])
x_final = np.array([15,-15,0,0])
w = np.array([[10,10],[20,10],[30,10],[30,0],[20,0],[10,-10]])
tau = np.array([10,25,30,40,50,60])
K = 6
E = np.array([[1,0,0,0],[0,1,0,0]])
```

Figure 2: Initialization of variables

Our objective in this task is:

$$\begin{aligned}
\underset{x,u}{\text{minimize}} \quad & \sum_{k=1}^{k} ||Ex(\tau_k) - w_k||^2 + \lambda \sum_{t=1}^{T-1} ||u(t) - u(t-1)||_2^2 \\
\text{subject to} \quad & x(0) = x_{initial} \\
& x(T) = x_{final} \\
& ||u(t)|| \leq U_{max} \\
& x(t+1) = Ax(t) + Bu(t)
\end{aligned} \tag{1}$$



```python
# Define and solve the CVXPY problem.

## Declare Variables to be Optimized
x = cp.Variable((T,4))
u = cp.Variable((T,2))

## Declare parameter lambda so it can be changed after problem formulation
lam = cp.Parameter(nonneg=True)

## Cost Function
cost = sum( [ (cp.pnorm(E @ x[tau[n]] - w[n],2)**2) for n in range(K)] )

for t in range(1, T):
    cost += lam * (cp.pnorm(u[t,:] - u[t-1,:],2))**2

## Constraints
# Overall constraints
constraints = [
               x[0,:] == x_initial,
               x[T-1,:] == x_final
               ]

# Element wise constraints
for t in range(T-1):
    constraints += [
        x[t+1,:] == A @ x[t,:] + B @ u[t,:],
        cp.pnorm(u[t],2) <= Umax
    ]

obj = cp.Minimize(cost)

prob = cp.Problem(obj,constraints)
```

Figure 3: Code used in task 1

Using the code above, we just need to compute the different values of λ in order to obtain both plots using the code in 1

The first plot is the robots position for the optimized input force over time. The red squares are the waypoints (defined as the matrix w) and the red circles are the robot positions ate the appointed time of the waypoints.

The second plot are the input forces. Blue is u1(t) and Orange is u2(t).

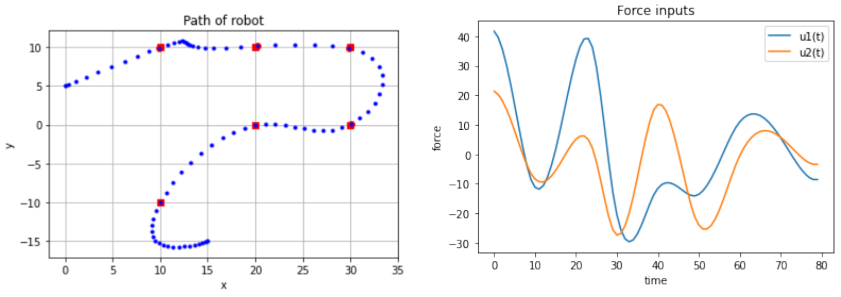


Figure 4: $\lambda = 10^{-3}$

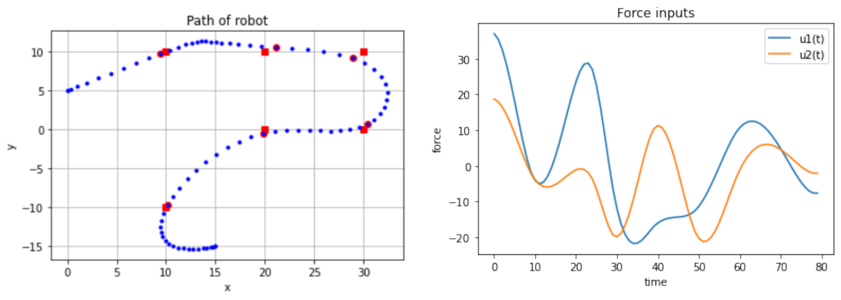


Figure 5: $\lambda = 10^{-2}$

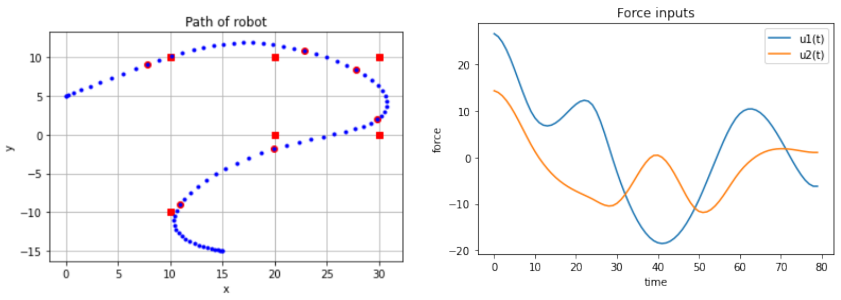


Figure 6: $\lambda = 10^{-1}$

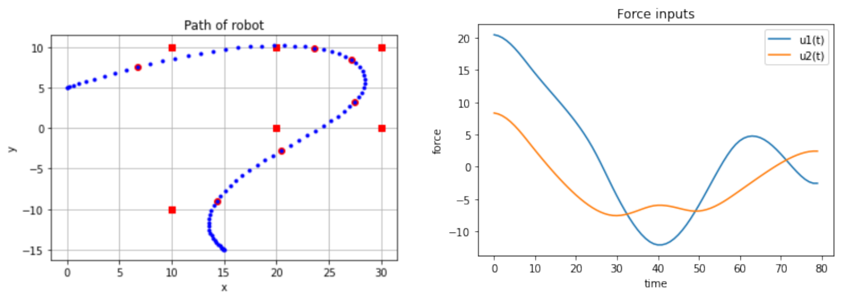


Figure 7: $\lambda = 10^0$

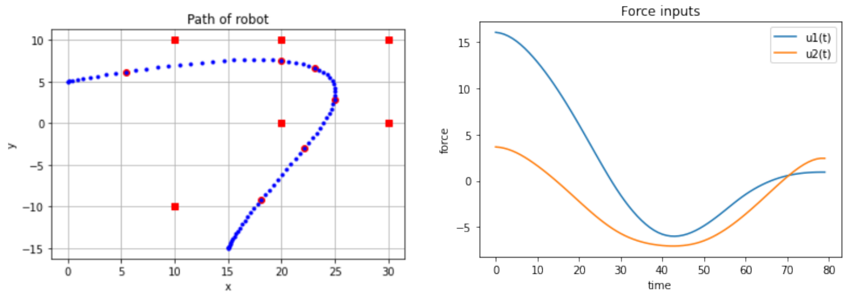


Figure 8: $\lambda = 10^1$

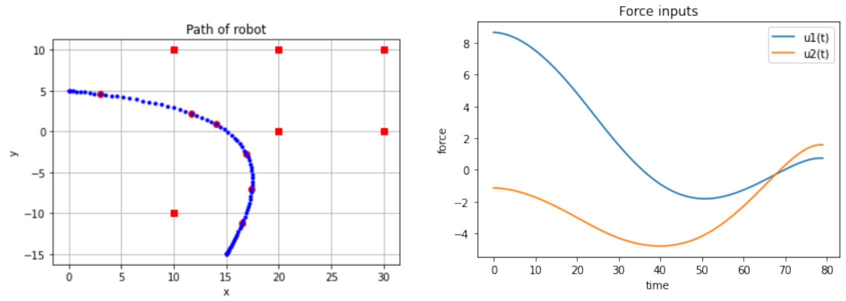


Figure 9: $\lambda = 10^2$

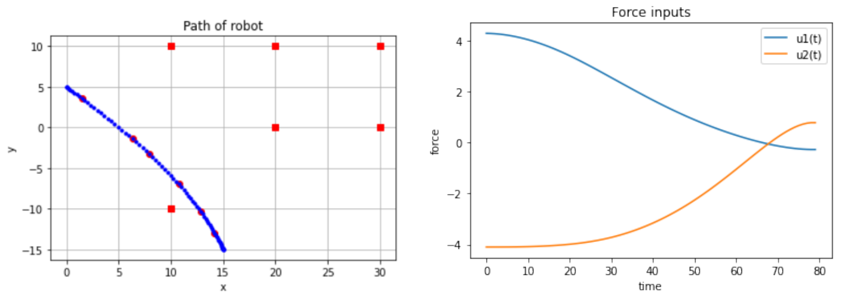


Figure 10: $\lambda = 10^3$

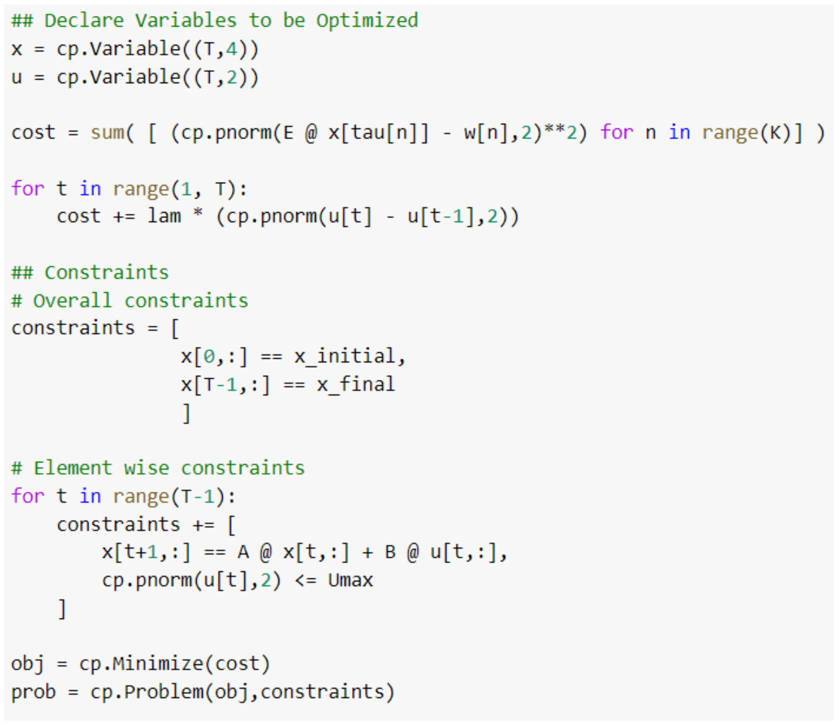
λ	Waypoint deviation	Optimal Control Signal Changes
10^{-3}	0.12653634833380814	79
10^{-2}	0.8291418701691656	79
10^{-1}	2.206581300320847	79
10^{0}	3.7140231509139032	79
10^{1}	5.627997634931546	79
10^{2}	11.029778990641217	79
10^{3}	15.394723280425408	79

1.2 Task 2

Now we remove the square from regularizer and compare the results.

$$\begin{aligned} \underset{x,u}{\text{minimize}} \quad & \sum_{k=1}^{k} ||Ex(\tau_k) - w_k||^2 + \lambda \sum_{t=1}^{T-1} ||u(t) - u(t-1)|| \\ \text{subject to} \quad & x(0) = x_{initial} \\ & x(T) = x_{final} \\ & ||u(t)|| \leq U_{max} \\ & x(t+1) = Ax(t) + Bu(t) \end{aligned}$$

Using the following code:



```
## Declare Variables to be Optimized
x = cp.Variable((T,4))
u = cp.Variable((T,2))

cost = sum( [ (cp.pnorm(E @ x[tau[n]] - w[n],2)**2) for n in range(K)] )

for t in range(1, T):
    cost += lam * (cp.pnorm(u[t] - u[t-1],2))

## Constraints
# Overall constraints
constraints = [
               x[0,:] == x_initial,
               x[T-1,:] == x_final
               ]

# Element wise constraints
for t in range(T-1):
    constraints += [
        x[t+1,:] == A @ x[t,:] + B @ u[t,:],
        cp.pnorm(u[t],2) <= Umax
    ]

obj = cp.Minimize(cost)
prob = cp.Problem(obj,constraints)
```

Figure 11: Code used in task 2

We obtain the following results:

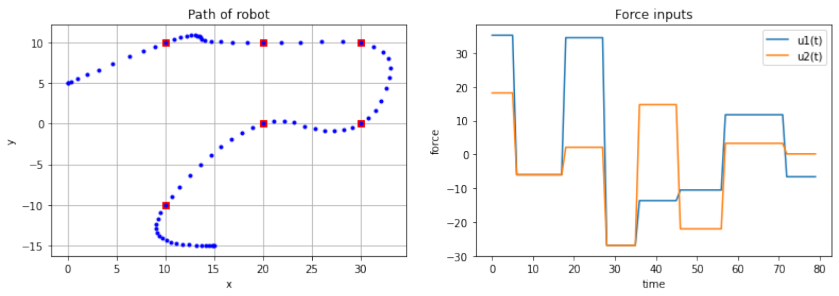


Figure 12: $\lambda = 10^{-3}$

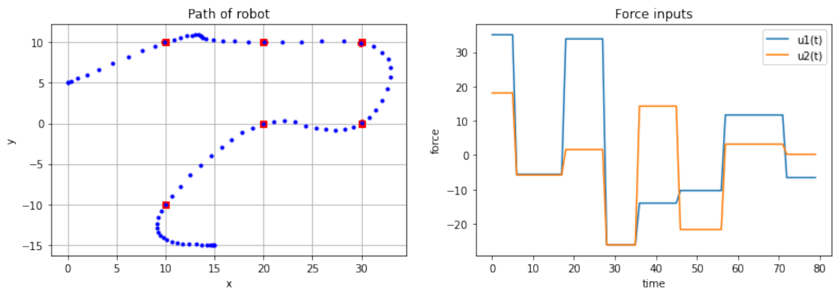


Figure 13: $\lambda = 10^{-2}$

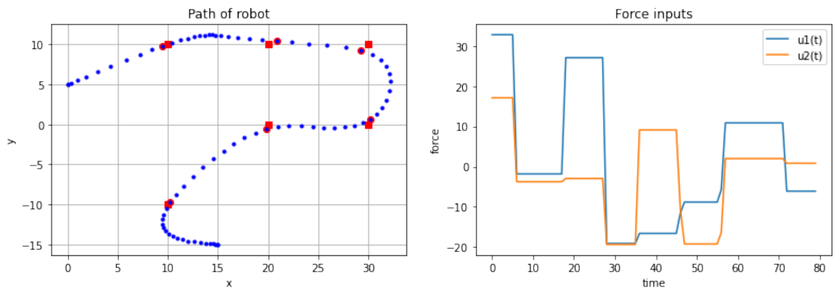


Figure 14: $\lambda = 10^{-1}$

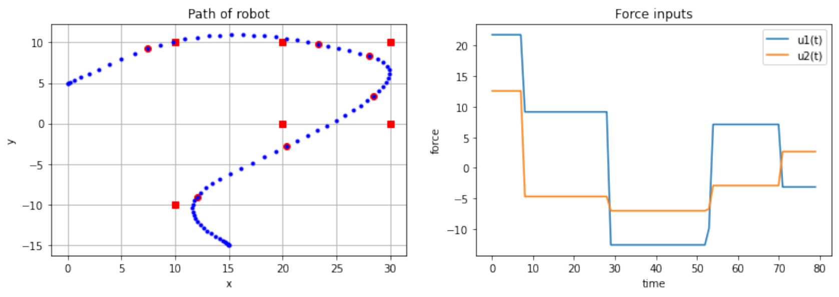


Figure 15: $\lambda = 10^{0}$

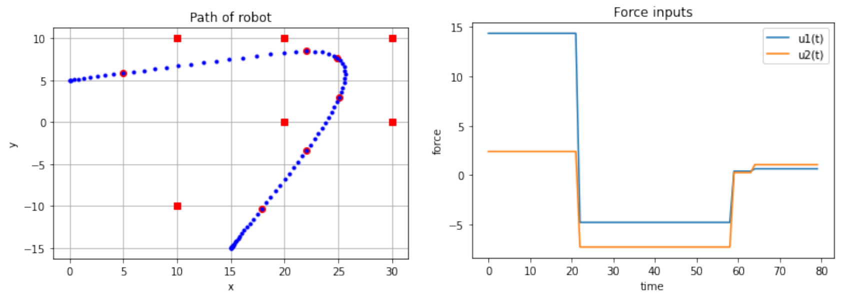


Figure 16: $\lambda = 10^1$

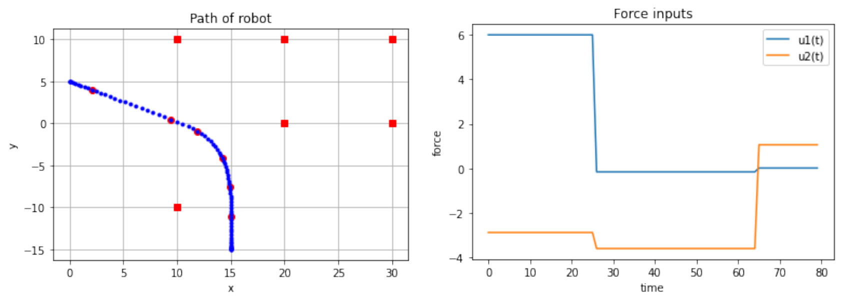


Figure 17: $\lambda = 10^2$

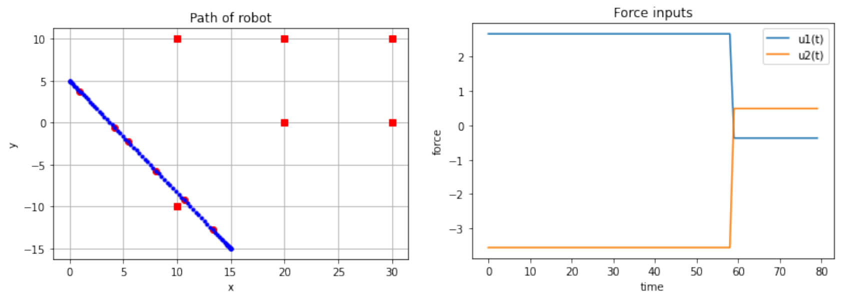
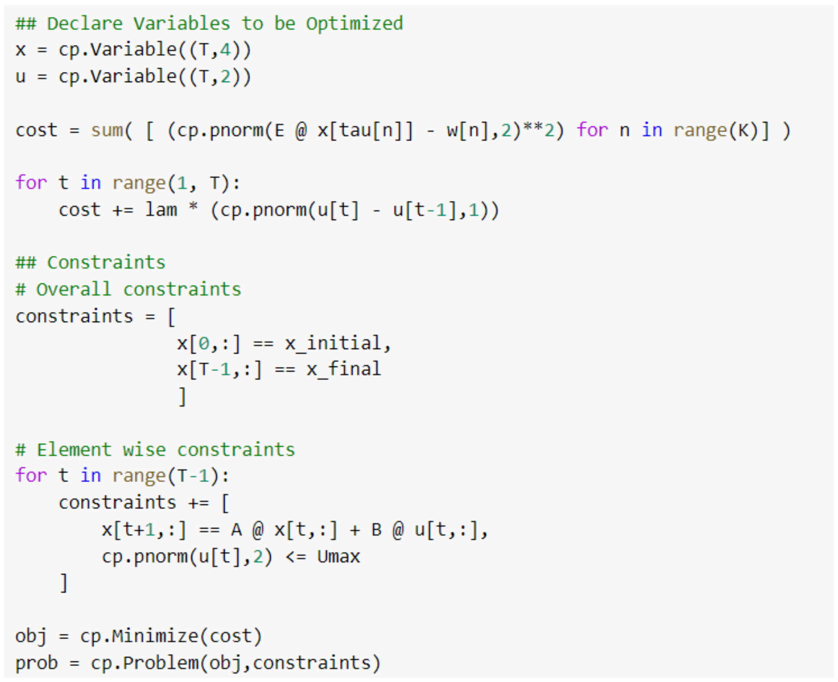


Figure 18: $\lambda = 10^3$

λ	Waypoint deviation	Optimal Control Signal Changes
10^{-3}	0.007526051506597102 6	7
10^{-2}	0.07490042785104145	7
10^{-1}	0.7042847620936804	9
10^{0}	2.902559005393649	5
10^{1}	5.360819048216747	3
10^{2}	12.656066807343135	2
10^{3}	16.27485839271584	1

1.3 Task 3



```
## Declare Variables to be Optimized
x = cp.Variable((T,4))
u = cp.Variable((T,2))

cost = sum( [ (cp.pnorm(E @ x[tau[n]] - w[n],2)**2) for n in range(K)] )

for t in range(1, T):
    cost += lam * (cp.pnorm(u[t] - u[t-1],1))

## Constraints
# Overall constraints
constraints = [
               x[0,:] == x_initial,
               x[T-1,:] == x_final
               ]

# Element wise constraints
for t in range(T-1):
    constraints += [
        x[t+1,:] == A @ x[t,:] + B @ u[t,:],
        cp.pnorm(u[t],2) <= Umax
    ]

obj = cp.Minimize(cost)
prob = cp.Problem(obj,constraints)
```

Figure 19: Code used in task 3

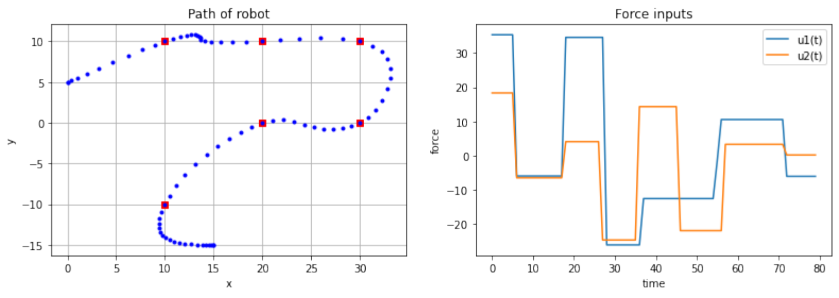


Figure 20: $\lambda = 10^{-3}$

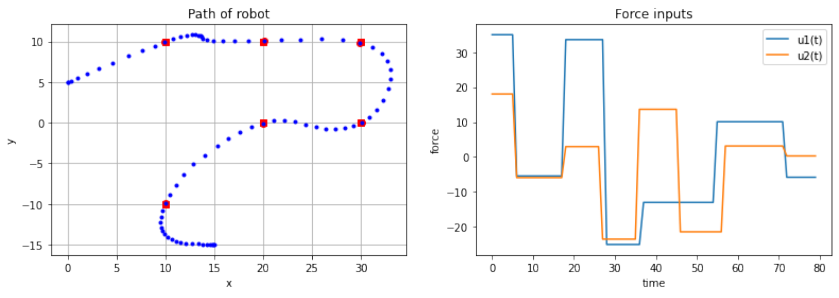


Figure 21: $\lambda = 10^{-2}$

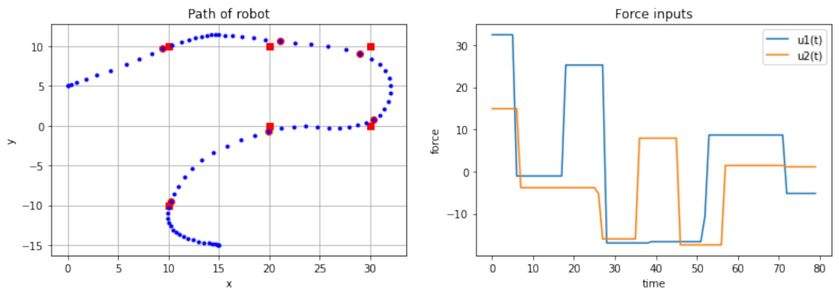


Figure 22: $\lambda = 10^{-1}$

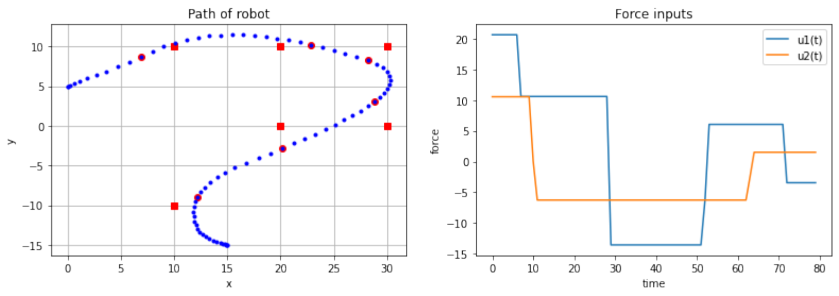


Figure 23: $\lambda = 10^0$

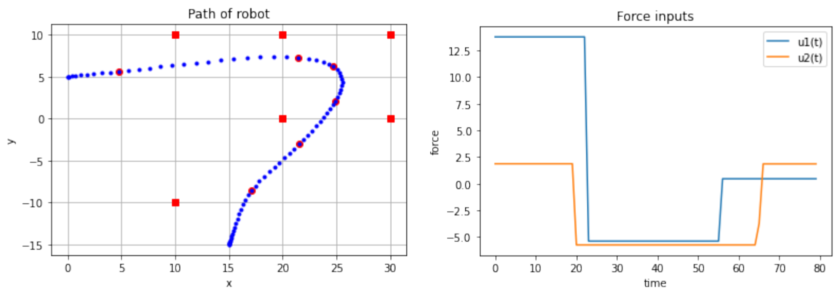


Figure 24: $\lambda = 10^1$

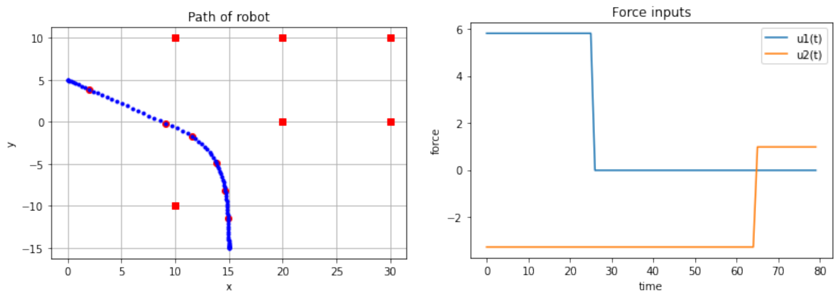


Figure 25: $\lambda = 10^2$

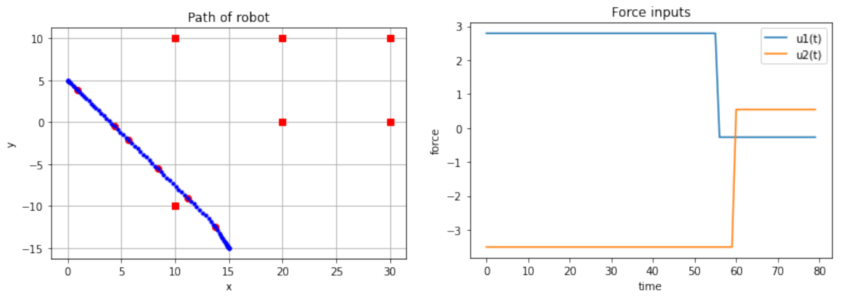


Figure 26: $\lambda = 10^3$

λ	Waypoint deviation	Optimal Control Signal Changes
10^{-3}	0.010834711962522931	11
10^{-2}	0.10738083162217865	10
10^{-1}	0.8922241950430558	13
10^{0}	2.8940849000558297	9
10^{1}	5.421924524677606	5
10^{2}	13.094034792975402	2
10^{3}	16.075380063793776	2

1.4 Task 4

$$\sum_{k=1}^{k} ||Ex(\tau_k) - w_k||_2^2 + \lambda \sum_{t=1}^{T-1} ||u(t) - u(t-1)|| \tag{2}$$

Our cost function is composed of two sums, one that computes the divergence of our robot from its intended path and other that computes the changes to our control signal.

In order to minimize our cost function, we tried 3 different approaches, given by each task: the l_2^2 regularizer on task 1, the l_2 regularizer on task 2 and the l_1 regularizer on task 3. Each variation corresponded to alterations to the second sum in our cost function whereas the first sum remained constant throughout.

For each case, we first saw the impact of the λ factor, which, as it increased, multiplied our difference between consecutive control signal inputs, therefore making our robot deviate further away from its course, although it did result in a simpler control system.

By comparing the three tasks in terms of control signal changes, we can easily distinguish the first task from the other two, as it has a higher, constant number of changes for each value of λ, while the other two tasks have significantly fewer optimal control signal changes and this number decreases inversely proportionally to λ. This can be attributed to the fact that the l_2^2 regularizer does not allow the difference between consecutive control signals to reach zero so $u(t) \neq u(t-1)$ for any value of t, which does not apply for the other two tasks, as we can clearly see many instances of t.

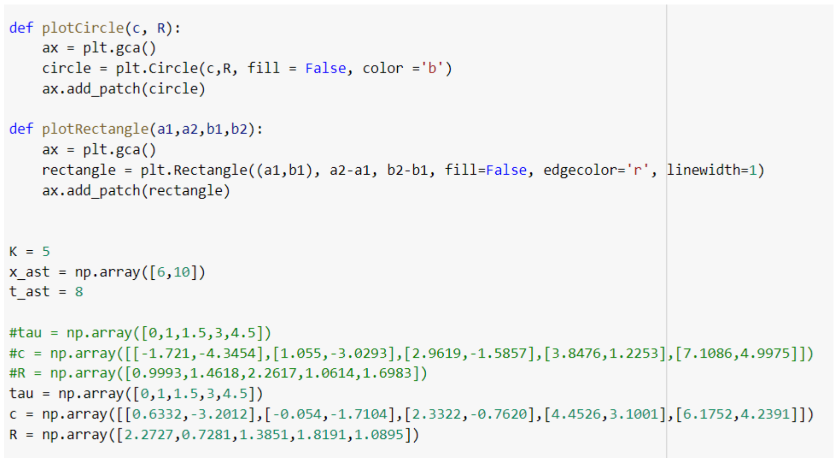
When comparing the three methods, we can see that the l_1 method is the least efficient, since it operates whenever any of the coordinates change, so the optimal control signal sum could have multiple solutions based on both coordinates, whereas the first method is the most efficient as it is the only one that does not allow both multiple and constant solutions, however it is constitutes a much more complex control system than the other two.

Therefore, when analyzing this problem from the point of view of the wishes formulated in the beginning of this part, we can conclude that while the first two wishes are always fulfilled, the other two are dependent on other factors. The third wish is dependent on the λ factor. To fulfill the last wish, we can either raise the value of the regularization parameter or use either of the last two regularizers.

1.5 Task 5

Variable initialization and function declaration:

$$\begin{aligned} \underset{p_0, v}{\text{minimize}} \quad & ||p_0 + t^* v - x^*||^2 \\ \text{subject to} \quad & R_k \geq ||p_0 + \tau_k v - c_k|| \end{aligned}$$



```python
def plotCircle(c, R):
    ax = plt.gca()
    circle = plt.Circle(c,R, fill = False, color ='b')
    ax.add_patch(circle)

def plotRectangle(a1,a2,b1,b2):
    ax = plt.gca()
    rectangle = plt.Rectangle((a1,b1), a2-a1, b2-b1, fill=False, edgecolor='r', linewidth=1)
    ax.add_patch(rectangle)


K = 5
x_ast = np.array([6,10])
t_ast = 8

#tau = np.array([0,1,1.5,3,4.5])
#c = np.array([[-1.721,-4.3454],[1.055,-3.0293],[2.9619,-1.5857],[3.8476,1.2253],[7.1086,4.9975]])
#R = np.array([0.9993,1.4618,2.2617,1.0614,1.6983])
tau = np.array([0,1,1.5,3,4.5])
c = np.array([[0.6332,-3.2012],[-0.054,-1.7104],[2.3322,-0.7620],[4.4526,3.1001],[6.1752,4.2391]])
R = np.array([2.2727,0.7281,1.3851,1.8191,1.0895])
```

Formulation and resolution of the optimization problem

```
# Define and solve the CVXPY problem.

## Declare Variables to be Optimized
p_0 = cp.Variable(2)
v = cp.Variable(2)

cost = cp.pnorm(p_0 + t_ast*v - x_ast,2)**2


## Constraints

# Element wise constraints
constraints = []
for k in range(K):
    constraints += [
        cp.pnorm(p_0 + tau[k]*v - c[k],2) <= R[k]
    ]

obj = cp.Minimize(cost)

prob = cp.Problem(obj,constraints)

prob.solve()

print("p_0="+str(p_0.value) +
" v=" + str(v.value) +
" p[t*]="+ str(p_0.value + t_ast*v.value) +
" distance=" + str(np.linalg.norm(p_0.value + t_ast * v.value - x_ast,2)))

plt.figure(1)
for k in range(K):
    plotCircle(c[k],R[k])
    plt.plot(p_0.value[0]+v.value[0]*tau[k], p_0.value[1]+v.value[1]*tau[k], 'rs')

plt.plot(p_0.value[0]+v.value[0]*t_ast, p_0.value[1]+v.value[1]*t_ast, 'rs')
plt.plot(x_ast[0], x_ast[1], 'k.')
ax = plt.gca()
ax.axis("scaled")
ax.set(xlim=(-6, 12), ylim=(-6, 12))
plt.grid()
```

1.6 Task 6

We used the following optimization to define the rectangle:

$$
\begin{aligned}
a_1 = \min_{p_0, v} \quad & p_0^{(1)} + t^* v^{(1)} \\
\text{subject to} \quad & R_k \geq ||p_0 + \tau_k v - c_k|| \\
a_2 = \max_{p_0, v} \quad & p_0^{(1)} + t^* v^{(1)} \\
\text{subject to} \quad & R_k \geq ||p_0 + \tau_k v - c_k||
\end{aligned}
$$

$$
\begin{aligned}
b_1 = \min_{p_0, v} \quad & p_0^{(2)} + t^* v^{(2)} \\
\text{subject to} \quad & R_k \geq ||p_0 + \tau_k v - c_k|| \\
b_2 = \max_{p_0, v} \quad & p_0^{(2)} + t^* v^{(2)} \\
\text{subject to} \quad & R_k \geq ||p_0 + \tau_k v - c_k||
\end{aligned}
$$

```python
cost = p_0[0] + t_ast*v[0]
obj = cp.Minimize(cost)
prob = cp.Problem(obj,constraints)
prob.solve()
a1 = p_0.value[0] + t_ast*v.value[0]

cost = p_0[0] + t_ast*v[0]
obj = cp.Maximize(cost)
prob = cp.Problem(obj,constraints)
prob.solve()
a2 = p_0.value[0] + t_ast*v.value[0]

cost = p_0[1] + t_ast*v[1]
obj = cp.Minimize(cost)
prob = cp.Problem(obj,constraints)
prob.solve()
b1 = p_0.value[1] + t_ast*v.value[1]

cost = p_0[1] + t_ast*v[1]
obj = cp.Maximize(cost)
prob = cp.Problem(obj,constraints)
prob.solve()
b2 = p_0.value[1] + t_ast*v.value[1]



print('a1='+str(a1)+' a2='+str(a2)+' b1='+str(b1)+' b2='+str(b2))

plt.figure(2)
for k in range(K):
    plotCircle(c[k],R[k])

plotRectangle(a1,a2,b1,b2)
ax = plt.gca()
ax.axis("scaled")
ax.set(xlim=(-6, 18), ylim=(-6, 18))
plt.grid()
plt.show()
```

Which gave us the following results:

$Variable$	Value
p_0	[-0.60420065, -3.25237183]
v	[1.26495079, 1.68075889]
p[t*]	[9.51540567 10.19369932]
distance	3.5207380498715874
a1	9.512187841372409
a2	14.190188891604238
b1	7.520306067674468
b2	12.945777974441587

Which gave us the following plots:

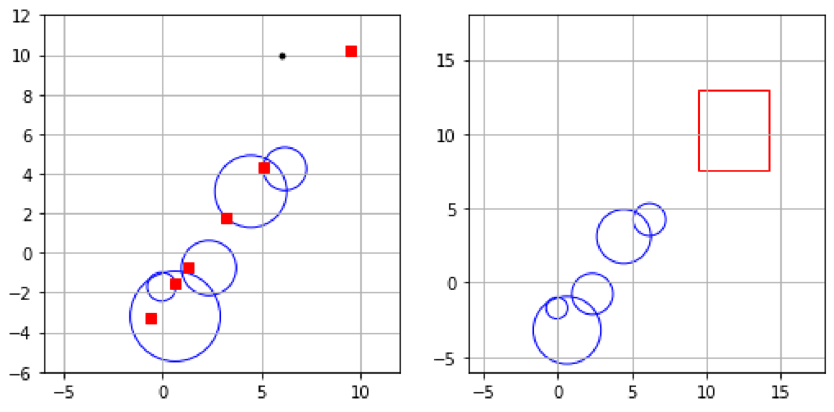


Figure 27: Distance from a critical point and smallest enclosing rectangle

2 Part 2

2.1 Task 1

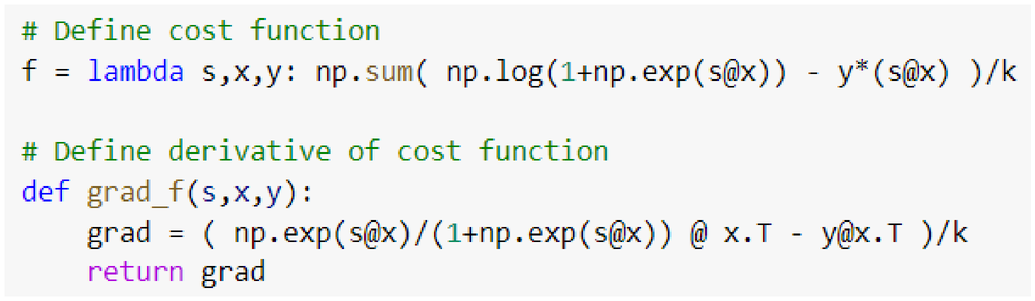
$$\underset{(s,r)\in\mathbf{R}^n\times\mathbf{R}}{\text{minimize}} \quad \underbrace{\frac{1}{K}\sum_{k=1}^{K}\left(\log\left(1+\exp(s^T x_k - r)\right) - y_k\left(s^T x_k - r\right)\right)}_{f(s,r)}. \qquad (1)$$

Figure 28: Cost function for part 2

By analyzing this function, we can prove its convexity by decomposing the elements in the sum into simpler convex functions, as both the sum itself and the multiplying factor $\frac{1}{K}$ do not influence its convexity.

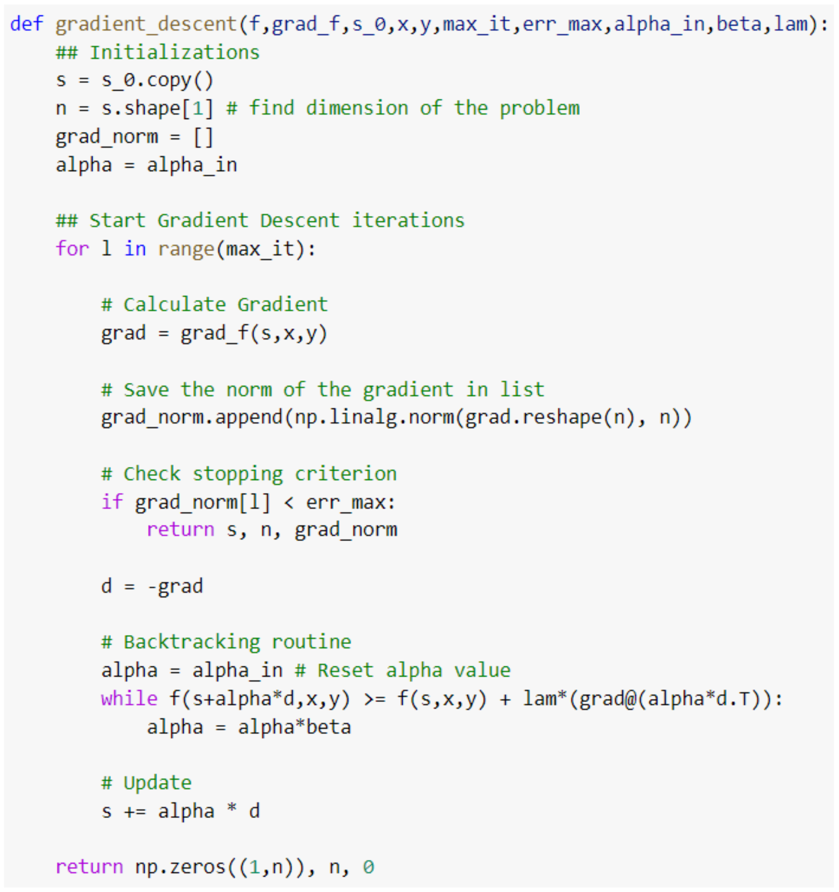
We can see that our cost function is composed by two known convex functions: both expressions of the form $log(1 + \exp x)$ and $ax + b$ are convex, therefore, since our cost function is a linear combination of these functions, we can conclude that our function is convex.

2.2 Task 2



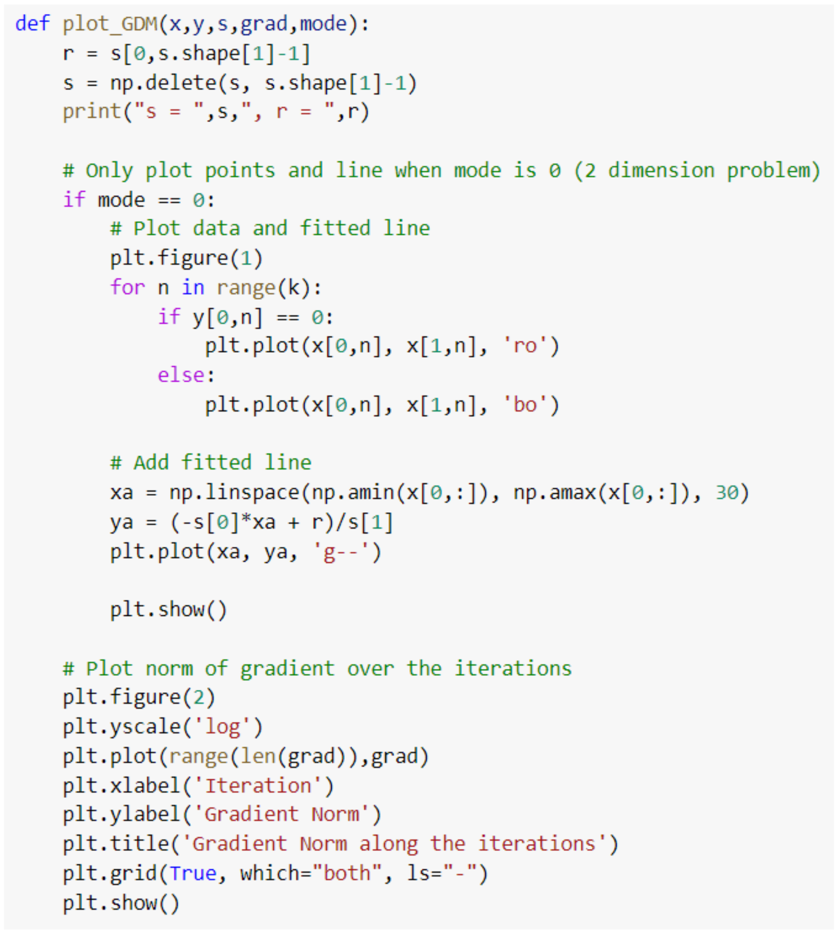
```python
# Define cost function
f = lambda s,x,y: np.sum( np.log(1+np.exp(s@x)) - y*(s@x) )/k

# Define derivative of cost function
def grad_f(s,x,y):
    grad = ( np.exp(s@x)/(1+np.exp(s@x)) @ x.T - y@x.T )/k
    return grad
```

Figure 29: Code to implement the cost function for part 2



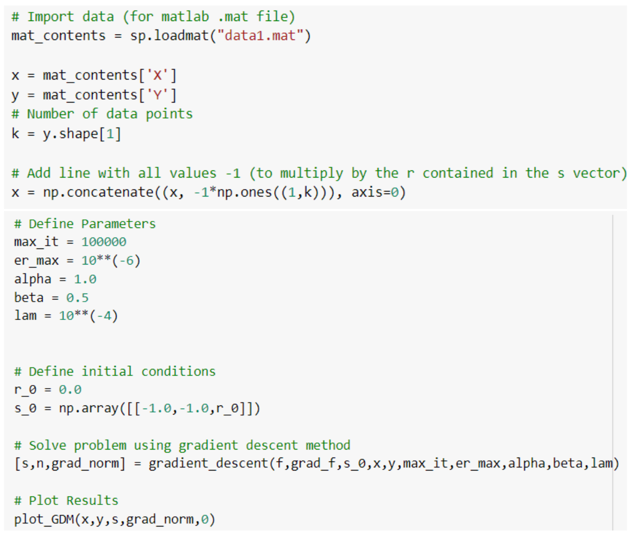
```python
def gradient_descent(f,grad_f,s_0,x,y,max_it,err_max,alpha_in,beta,lam):
    ## Initializations
    s = s_0.copy()
    n = s.shape[1] # find dimension of the problem
    grad_norm = []
    alpha = alpha_in

    ## Start Gradient Descent iterations
    for l in range(max_it):

        # Calculate Gradient
        grad = grad_f(s,x,y)

        # Save the norm of the gradient in list
        grad_norm.append(np.linalg.norm(grad.reshape(n), n))

        # Check stopping criterion
        if grad_norm[l] < err_max:
            return s, n, grad_norm

        d = -grad

        # Backtracking routine
        alpha = alpha_in # Reset alpha value
        while f(s+alpha*d,x,y) >= f(s,x,y) + lam*(grad@(alpha*d.T)):
            alpha = alpha*beta

        # Update
        s += alpha * d

    return np.zeros((1,n)), n, 0
```

Figure 30: Code to implement the gradient descent method



```python
def plot_GDM(x,y,s,grad,mode):
    r = s[0,s.shape[1]-1]
    s = np.delete(s, s.shape[1]-1)
    print("s = ",s,", r = ",r)

    # Only plot points and line when mode is 0 (2 dimension problem)
    if mode == 0:
        # Plot data and fitted line
        plt.figure(1)
        for n in range(k):
            if y[0,n] == 0:
                plt.plot(x[0,n], x[1,n], 'ro')
            else:
                plt.plot(x[0,n], x[1,n], 'bo')

        # Add fitted line
        xa = np.linspace(np.amin(x[0,:]), np.amax(x[0,:]), 30)
        ya = (-s[0]*xa + r)/s[1]
        plt.plot(xa, ya, 'g--')

        plt.show()

    # Plot norm of gradient over the iterations
    plt.figure(2)
    plt.yscale('log')
    plt.plot(range(len(grad)),grad)
    plt.xlabel('Iteration')
    plt.ylabel('Gradient Norm')
    plt.title('Gradient Norm along the iterations')
    plt.grid(True, which="both", ls="-")
    plt.show()
```

Figure 31: Code to plot the gradient descent method



```
# Import data (for matlab .mat file)
mat_contents = sp.loadmat("data1.mat")

x = mat_contents['X']
y = mat_contents['Y']
# Number of data points
k = y.shape[1]

# Add line with all values -1 (to multiply by the r contained in the s vector)
x = np.concatenate((x, -1*np.ones((1,k))), axis=0)
```

```
# Define Parameters
max_it = 100000
er_max = 10**(-6)
alpha = 1.0
beta = 0.5
lam = 10**(-4)


# Define initial conditions
r_0 = 0.0
s_0 = np.array([[-1.0,-1.0,r_0]])

# Solve problem using gradient descent method
[s,n,grad_norm] = gradient_descent(f,grad_f,s_0,x,y,max_it,er_max,alpha,beta,lam)

# Plot Results
plot_GDM(x,y,s,grad_norm,0)
```

Figure 32: Code to load and fit data to the gradient descent method

Where we obtained:

- s = [1.34953921 1.05397136]
- r = 4.881542559496009

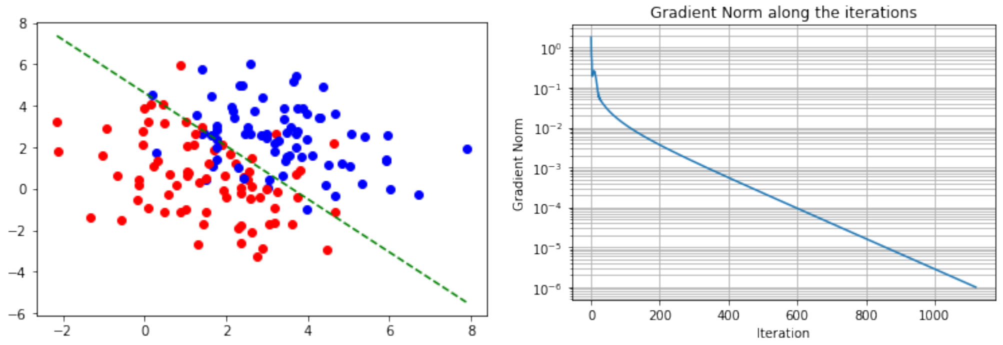
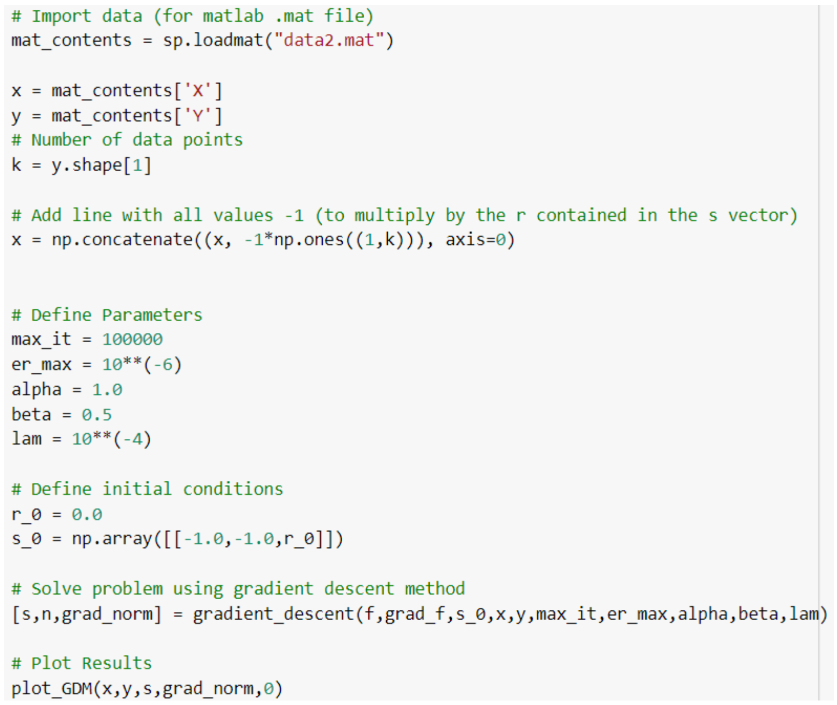


Figure 33: Plot of line along with dataset; Plot of the gradient of the cost function across iterations

2.3 Task 3



```
# Import data (for matlab .mat file)
mat_contents = sp.loadmat("data2.mat")

x = mat_contents['X']
y = mat_contents['Y']
# Number of data points
k = y.shape[1]

# Add line with all values -1 (to multiply by the r contained in the s vector)
x = np.concatenate((x, -1*np.ones((1,k))), axis=0)


# Define Parameters
max_it = 100000
er_max = 10**(-6)
alpha = 1.0
beta = 0.5
lam = 10**(-4)

# Define initial conditions
r_0 = 0.0
s_0 = np.array([[-1.0,-1.0,r_0]])

# Solve problem using gradient descent method
[s,n,grad_norm] = gradient_descent(f,grad_f,s_0,x,y,max_it,er_max,alpha,beta,lam)

# Plot Results
plot_GDM(x,y,s,grad_norm,0)
```

Figure 34: Adapting the code to the new dataset

Using data2.mat, we obtained the following results:

- s = [0.74021451 2.35765419]
- r = 4.555298359498374

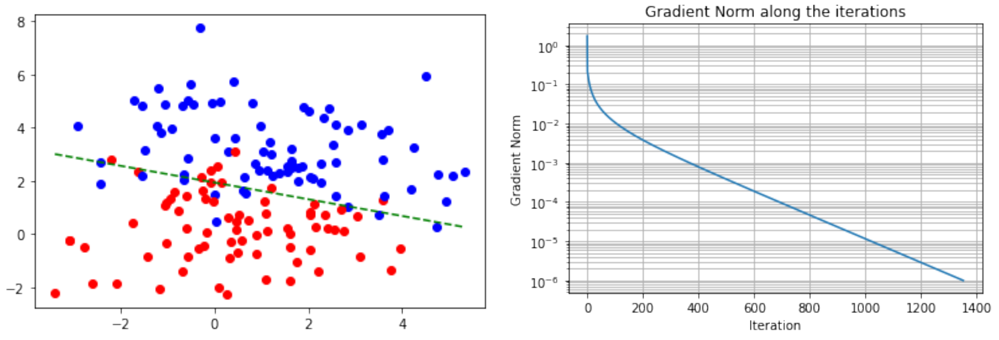
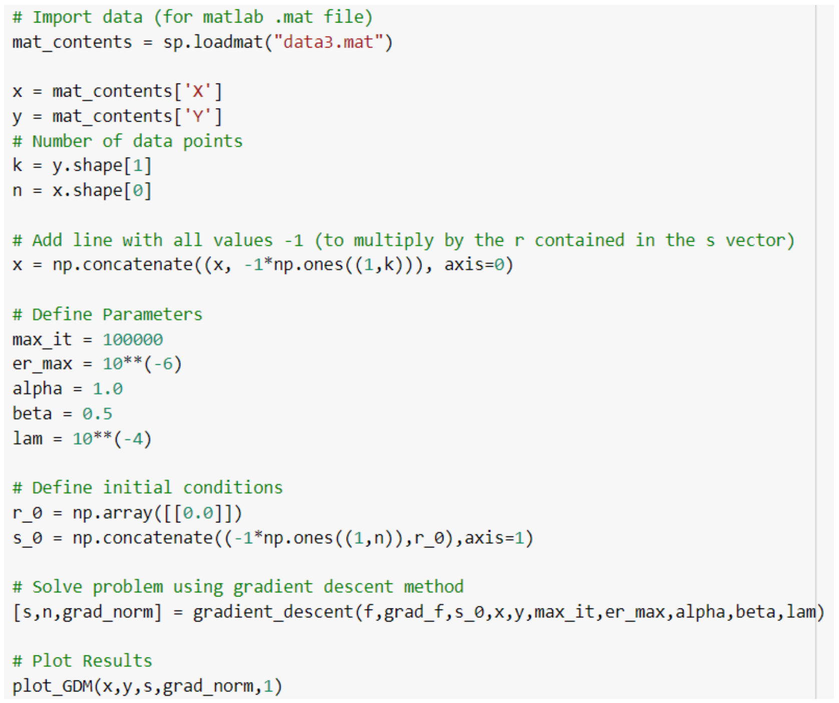


Figure 35: Plot of line along with dataset; Plot of the gradient of the cost function across iterations

2.4 Task 4



```
# Import data (for matlab .mat file)
mat_contents = sp.loadmat("data3.mat")

x = mat_contents['X']
y = mat_contents['Y']
# Number of data points
k = y.shape[1]
n = x.shape[0]

# Add line with all values -1 (to multiply by the r contained in the s vector)
x = np.concatenate((x, -1*np.ones((1,k))), axis=0)

# Define Parameters
max_it = 100000
er_max = 10**(-6)
alpha = 1.0
beta = 0.5
lam = 10**(-4)

# Define initial conditions
r_0 = np.array([[0.0]])
s_0 = np.concatenate((-1*np.ones((1,n)),r_0),axis=1)

# Solve problem using gradient descent method
[s,n,grad_norm] = gradient_descent(f,grad_f,s_0,x,y,max_it,er_max,alpha,beta,lam)

# Plot Results
plot_GDM(x,y,s,grad_norm,1)
```

Figure 36: Adapting the code to data3.mat

Using data3.mat, we obtained the following results:

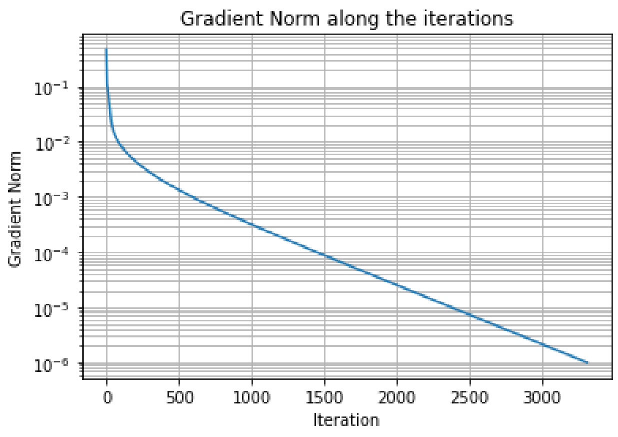


Figure 37: Plot of the gradient of the cost function across iterations

In this task, we were also asked to fit data4.mat:

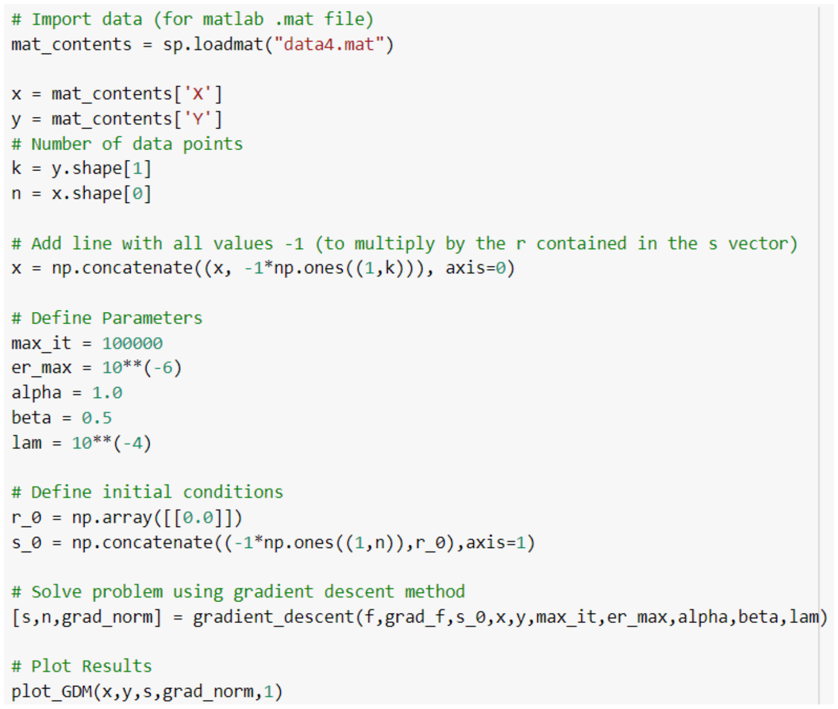

```
# Import data (for matlab .mat file)
mat_contents = sp.loadmat("data4.mat")

x = mat_contents['X']
y = mat_contents['Y']
# Number of data points
k = y.shape[1]
n = x.shape[0]

# Add line with all values -1 (to multiply by the r contained in the s vector)
x = np.concatenate((x, -1*np.ones((1,k))), axis=0)

# Define Parameters
max_it = 100000
er_max = 10**(-6)
alpha = 1.0
beta = 0.5
lam = 10**(-4)

# Define initial conditions
r_0 = np.array([[0.0]])
s_0 = np.concatenate((-1*np.ones((1,n)),r_0),axis=1)

# Solve problem using gradient descent method
[s,n,grad_norm] = gradient_descent(f,grad_f,s_0,x,y,max_it,er_max,alpha,beta,lam)

# Plot Results
plot_GDM(x,y,s,grad_norm,1)
```

Figure 38: Adapting the code to data4.mat

Which gave us:

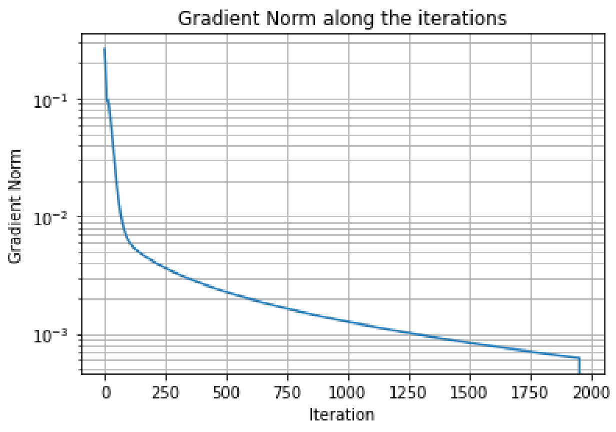


Figure 39: Plot of the gradient of the cost function across iterations

2.5 Task 5

Given the equation:

$$p(x) = \sum_{k=1}^{K} \phi(a_k^T x) \tag{3}$$

Its gradient is given by:

$$\nabla p(x) = \nabla \sum_{k=1}^{K} \phi(a_k^T x) = \sum_{k=1}^{K} \nabla \phi(a_k^T x) = \sum_{k=1}^{K} a_k \dot{\phi}(a_k^T x) = Av \tag{4}$$

Since both a_k and p have three dimensions, the resulting gradient is a 3 by 1 matrix.

The Hessian of p is given by:

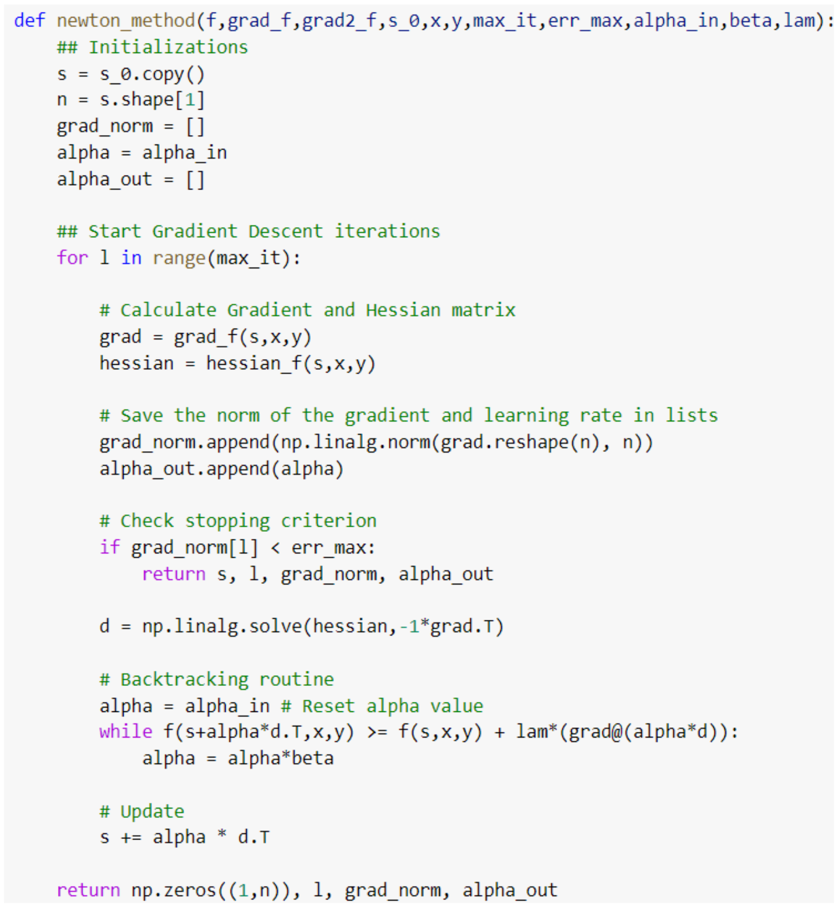
$$\nabla^2 p(x) = \nabla \sum_{k=1}^{K} a_k \dot{\phi}(a_k^T x) = \sum_{k=1}^{K} a_k \nabla \dot{\phi}(a_k^T x) = \sum_{k=1}^{K} a_k^2 \ddot{\phi}(a_k^T x) = ADA^T \tag{5}$$

Taking into account the dimensions of our variables, the resulting hessian is a three by three matrix in which the result is:

$$\begin{bmatrix} \sum_{k=1}^{K} a_{k1}^2 \ddot{\phi}(a_k^T x) & \sum_{k=1}^{K} a_{k1} * a_{k2} \ddot{\phi}(a_k^T x) & \sum_{k=1}^{K} a_{k1} * a_{k3} \ddot{\phi}(a_k^T x) \\ \sum_{k=1}^{K} a_{k2} * a_{k1} \ddot{\phi}(a_k^T x) & \sum_{k=1}^{K} a_{k2}^2 \ddot{\phi}(a_k^T x) & \sum_{k=1}^{K} a_{k2} * a_{k3} \ddot{\phi}(a_k^T x) \\ \sum_{k=1}^{K} a_{k3} * a_{k1} \ddot{\phi}(a_k^T x) & \sum_{k=1}^{K} a_{k3} * a_{k2} \ddot{\phi}(a_k^T x) & \sum_{k=1}^{K} a_{k3}^2 \ddot{\phi}(a_k^T x) \end{bmatrix}$$

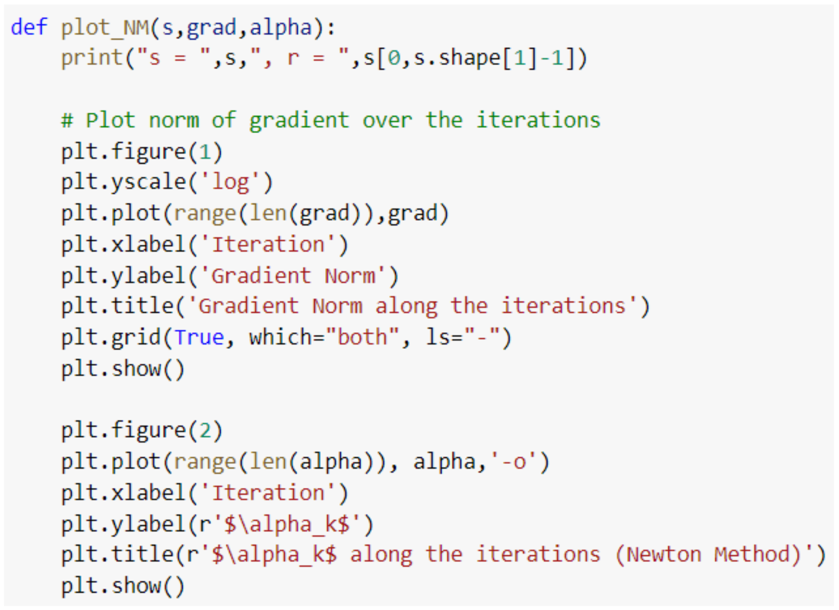
Both these expressions are equivalent to their matrix counterparts indicated in the task statement, therefore confirming their veracity.

2.6 Task 6



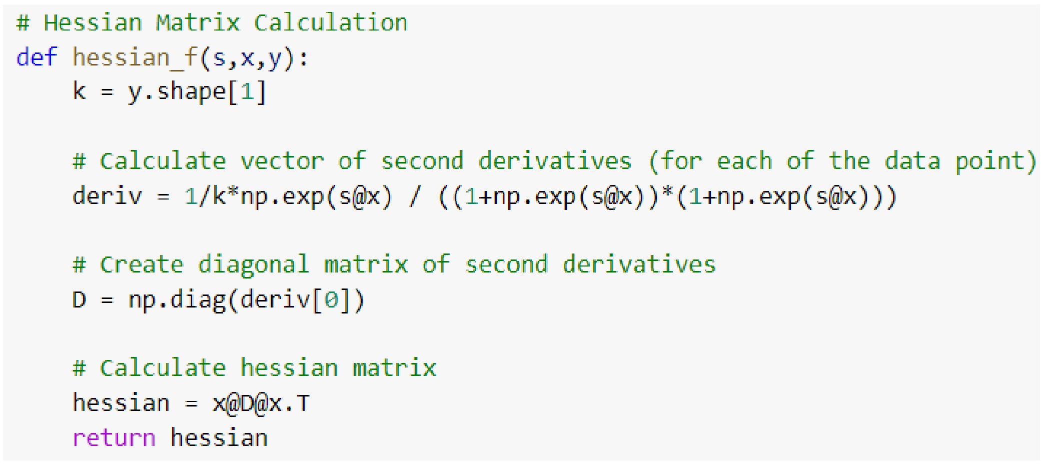
```
def newton_method(f,grad_f,grad2_f,s_0,x,y,max_it,err_max,alpha_in,beta,lam):
    ## Initializations
    s = s_0.copy()
    n = s.shape[1]
    grad_norm = []
    alpha = alpha_in
    alpha_out = []

    ## Start Gradient Descent iterations
    for l in range(max_it):

        # Calculate Gradient and Hessian matrix
        grad = grad_f(s,x,y)
        hessian = hessian_f(s,x,y)

        # Save the norm of the gradient and learning rate in lists
        grad_norm.append(np.linalg.norm(grad.reshape(n), n))
        alpha_out.append(alpha)

        # Check stopping criterion
        if grad_norm[l] < err_max:
            return s, l, grad_norm, alpha_out

        d = np.linalg.solve(hessian,-1*grad.T)

        # Backtracking routine
        alpha = alpha_in # Reset alpha value
        while f(s+alpha*d.T,x,y) >= f(s,x,y) + lam*(grad@(alpha*d)):
            alpha = alpha*beta

        # Update
        s += alpha * d.T

    return np.zeros((1,n)), l, grad_norm, alpha_out
```

Figure 40: Code to implement the Newton method



```python
def plot_NM(s,grad,alpha):
    print("s = ",s,", r = ",s[0,s.shape[1]-1])

    # Plot norm of gradient over the iterations
    plt.figure(1)
    plt.yscale('log')
    plt.plot(range(len(grad)),grad)
    plt.xlabel('Iteration')
    plt.ylabel('Gradient Norm')
    plt.title('Gradient Norm along the iterations')
    plt.grid(True, which="both", ls="-")
    plt.show()

    plt.figure(2)
    plt.plot(range(len(alpha)), alpha,'-o')
    plt.xlabel('Iteration')
    plt.ylabel(r'$\alpha_k$')
    plt.title(r'$\alpha_k$ along the iterations (Newton Method)')
    plt.show()
```

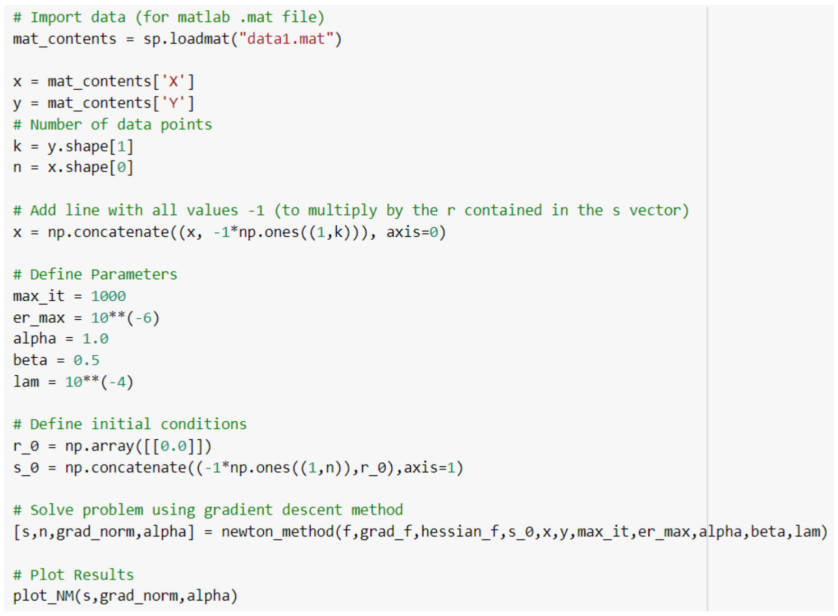
Figure 41: Code to plot the Newton method



```python
# Hessian Matrix Calculation
def hessian_f(s,x,y):
    k = y.shape[1]

    # Calculate vector of second derivatives (for each of the data point)
    deriv = 1/k*np.exp(s@x) / ((1+np.exp(s@x))*(1+np.exp(s@x)))

    # Create diagonal matrix of second derivatives
    D = np.diag(deriv[0])

    # Calculate hessian matrix
    hessian = x@D@x.T
    return hessian
```

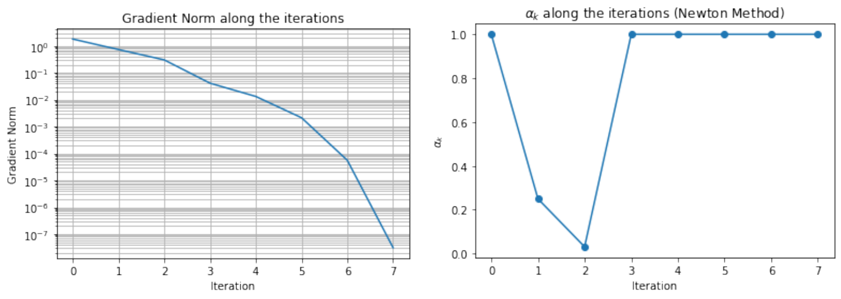
Figure 42: Code to calculate the hessian matrix

2.6.1 data1.mat

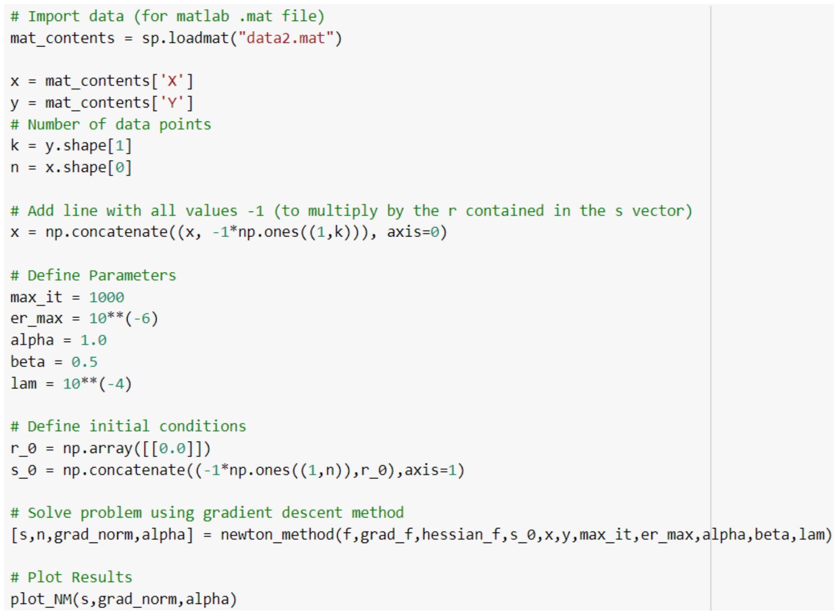


```
# Import data (for matlab .mat file)
mat_contents = sp.loadmat("data1.mat")

x = mat_contents['X']
y = mat_contents['Y']
# Number of data points
k = y.shape[1]
n = x.shape[0]

# Add line with all values -1 (to multiply by the r contained in the s vector)
x = np.concatenate((x, -1*np.ones((1,k))), axis=0)

# Define Parameters
max_it = 1000
er_max = 10**(-6)
alpha = 1.0
beta = 0.5
lam = 10**(-4)

# Define initial conditions
r_0 = np.array([[0.0]])
s_0 = np.concatenate((-1*np.ones((1,n)),r_0),axis=1)

# Solve problem using gradient descent method
[s,n,grad_norm,alpha] = newton_method(f,grad_f,hessian_f,s_0,x,y,max_it,er_max,alpha,beta,lam)

# Plot Results
plot_NM(s,grad_norm,alpha)
```

Figure 43: Loading and fitting for data1.mat

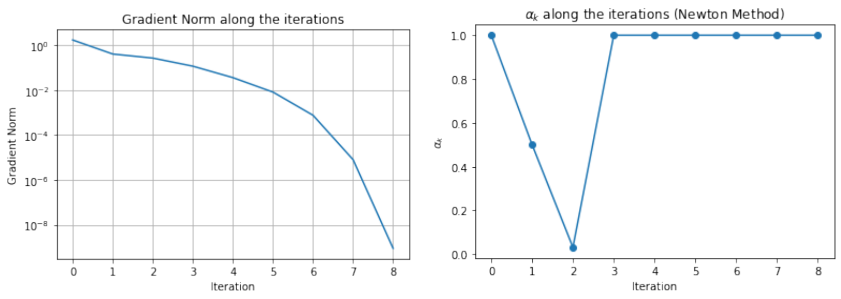


2.6.2 data2.mat

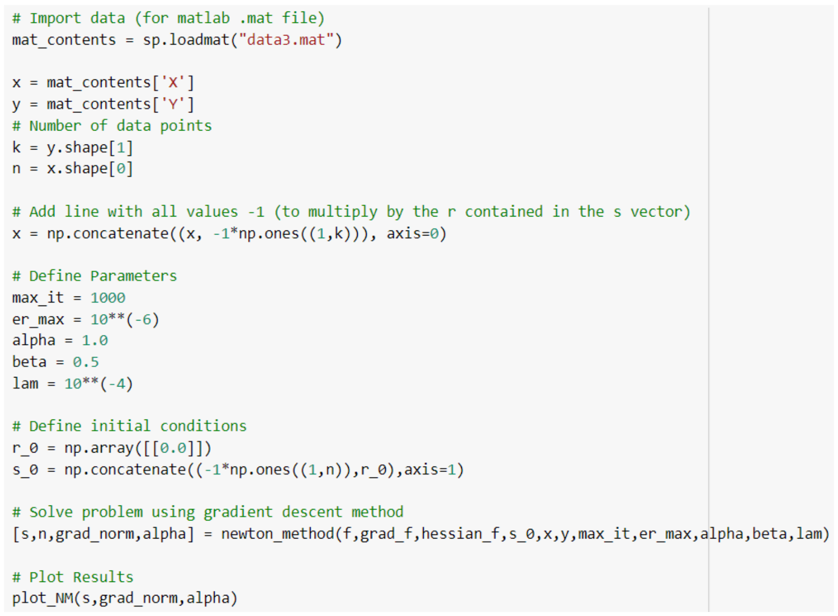


```
# Import data (for matlab .mat file)
mat_contents = sp.loadmat("data2.mat")

x = mat_contents['X']
y = mat_contents['Y']
# Number of data points
k = y.shape[1]
n = x.shape[0]

# Add line with all values -1 (to multiply by the r contained in the s vector)
x = np.concatenate((x, -1*np.ones((1,k))), axis=0)

# Define Parameters
max_it = 1000
er_max = 10**(-6)
alpha = 1.0
beta = 0.5
lam = 10**(-4)

# Define initial conditions
r_0 = np.array([[0.0]])
s_0 = np.concatenate((-1*np.ones((1,n)),r_0),axis=1)

# Solve problem using gradient descent method
[s,n,grad_norm,alpha] = newton_method(f,grad_f,hessian_f,s_0,x,y,max_it,er_max,alpha,beta,lam)

# Plot Results
plot_NM(s,grad_norm,alpha)
```

Figure 44: Loading and fitting for data2.mat

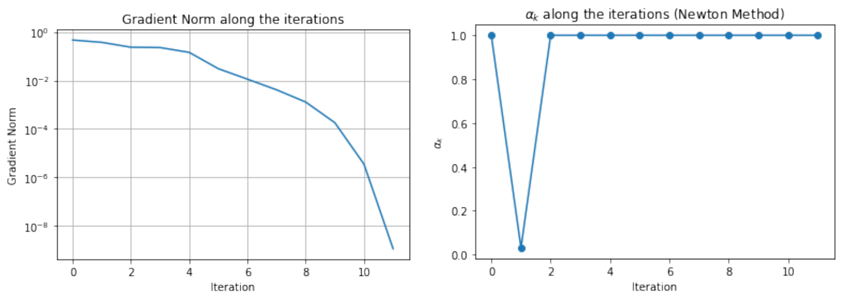


2.6.3 data3.mat

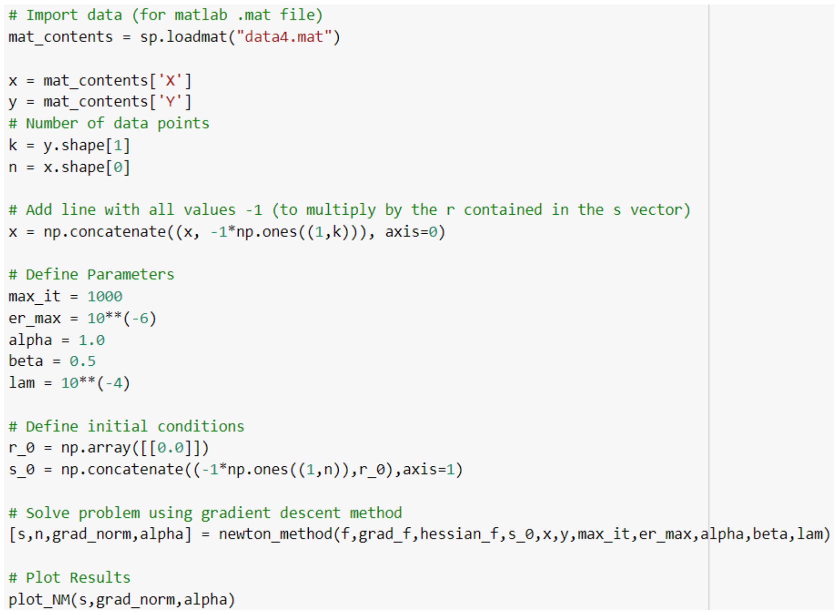


```
# Import data (for matlab .mat file)
mat_contents = sp.loadmat("data3.mat")

x = mat_contents['X']
y = mat_contents['Y']
# Number of data points
k = y.shape[1]
n = x.shape[0]

# Add line with all values -1 (to multiply by the r contained in the s vector)
x = np.concatenate((x, -1*np.ones((1,k))), axis=0)

# Define Parameters
max_it = 1000
er_max = 10**(-6)
alpha = 1.0
beta = 0.5
lam = 10**(-4)

# Define initial conditions
r_0 = np.array([[0.0]])
s_0 = np.concatenate((-1*np.ones((1,n)),r_0),axis=1)

# Solve problem using gradient descent method
[s,n,grad_norm,alpha] = newton_method(f,grad_f,hessian_f,s_0,x,y,max_it,er_max,alpha,beta,lam)

# Plot Results
plot_NM(s,grad_norm,alpha)
```

Figure 45: Loading and fitting for data3.mat

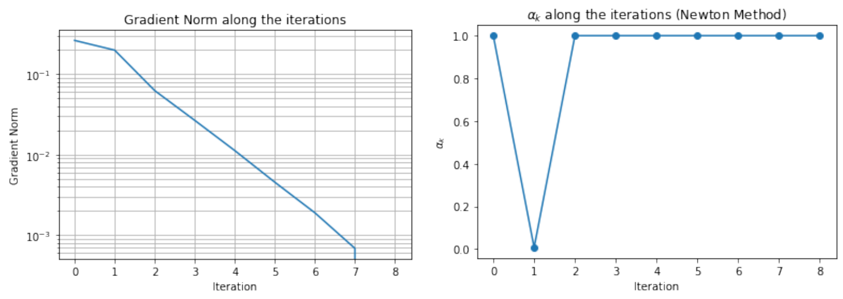


2.6.4 data4.mat



```
# Import data (for matlab .mat file)
mat_contents = sp.loadmat("data4.mat")

x = mat_contents['X']
y = mat_contents['Y']
# Number of data points
k = y.shape[1]
n = x.shape[0]

# Add line with all values -1 (to multiply by the r contained in the s vector)
x = np.concatenate((x, -1*np.ones((1,k))), axis=0)

# Define Parameters
max_it = 1000
er_max = 10**(-6)
alpha = 1.0
beta = 0.5
lam = 10**(-4)

# Define initial conditions
r_0 = np.array([[0.0]])
s_0 = np.concatenate((-1*np.ones((1,n)),r_0),axis=1)

# Solve problem using gradient descent method
[s,n,grad_norm,alpha] = newton_method(f,grad_f,hessian_f,s_0,x,y,max_it,er_max,alpha,beta,lam)

# Plot Results
plot_NM(s,grad_norm,alpha)
```

Figure 46: Loading and fitting for data4.mat



2.7 Task 7

Comparing the results obtained using the Gradient Descent Method and the Newton Method we can see a big difference in execution time and iterations needed. Our results of the tests are expressed in the table bellow containing the time it took to find the solution and the iterations needed.

We decided to evaluate the time as well because the time it takes to do a single iteration is dependant on the algorithm used. Since the Newton

File	n	k	iterations GDM	Execution time GDM	iterations NM	Execution time NM
data1.mat	2	150	1121	0.24806	8	0.011002
data2.mat	2	150	1353	0.201756	9	0.007001
data3.mat	30	500	3309	0.64412	12	0.03249
data4.mat	100	8000	1955	6.77544	9	4.29659
data3_mod.mat	30	150	-	-	21	0.02699
data4_mod.mat	100	150	285	0.071908	18	0.105

Method needs to calculate the Hessian matrix as well it is only natural that an iteration can take longer in average than the gradient descent (ignoring the backtracking). Algorithms can also be more prone to need longer backtracking routines and more often.

For the "data1.mat" we can see that the GDM (gradient descent method) finds a solution after 1121 iterations over 0.24806 seconds while using the NM (newton method) a solution is discovered after only 8 iterations taking 0.011002 seconds. This is a huge difference in time but if we compare the time per iteration we can conclude that the time per iteration is much higher in the newton method. If the newton method was to take 1121 iterations we can estimate that it would take around 1.5 seconds. This is because of the extra computational cost in calculating the Hessian matrix and solving for the vector d_k (equation 6).

$$d_k = -(\nabla^2 f(x_k))^{-1}\nabla^2 f(x_k) \tag{6}$$

We also created two new data files. These were copies of the data3 and data4 but with the amount of points reduced to the same number as the data1 and data2. Our objective was trying to isolate the effect the dimension of the problem has on the time and iterations needed to find the solution. In the GDM we found that it didn't find a solution in less than 100000 iterations for the modified data 3 but it found the solution for the modified data4 although in a suspicious small number of iterations. What we can conclude is that the number of points (150) is a very small number for the optimization to work with. In the NM it increased the number of iterations by about double and in the gradient method it decreased drastically to about one eighth of the unmodified. Due to this not many conclusions could be taken apart that the GM behaves better than the NM for a number o points closer to the dimension number.

2.7.1 Variation of Stopping Criterion

We also choose to analyse how the max error changes the iterations needed and time taken. From this analysis we got the results expressed in the following table.

File	Max error	Iterations GDM	Time GDM	Iterations NM	Time NM
data4.mat	10^{-1}	10	0.023036	3	1.11299
data4.mat	10^{-2}	69	0.30073	6	1.69218
data4.mat	10^{-3}	1282	4.03788	8	2.525
data4.mat	10^{-6}	1955	6.67895	9	3.79

From the data obtained we concluded that despite the newton method taking less iterations in almost all situations the extra computation required can be a downside. While for the regular stopping criterion of 10^{-6} it takes half of the time of the gradient method, for higher errors such as 10^{-2} it takes 1.6 seconds while the gradient method takes only 0.3 seconds.

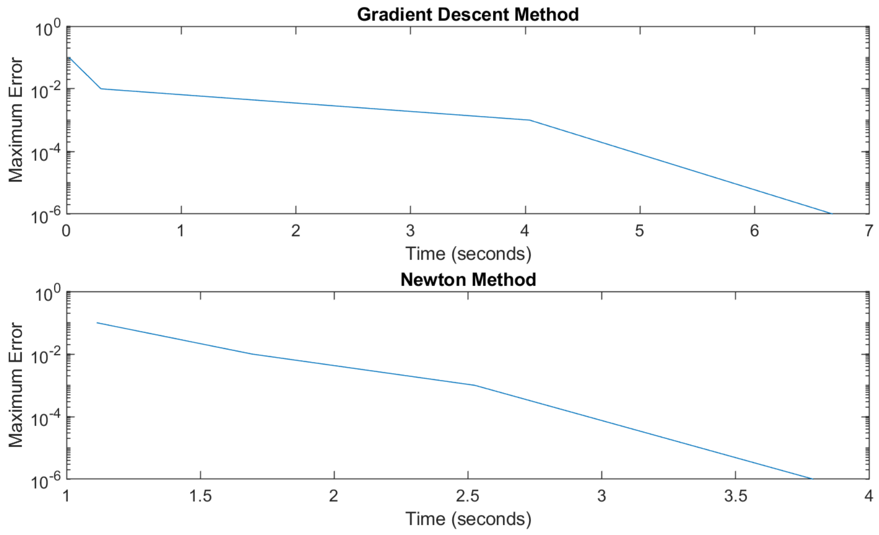


Figure 47: Time for different stopping criterion

Just as we concluded before the newton method seems to only require a few iterations but each iteration take quite a long time compared to other methods.

We can then conclude that it is worth switching to the Newton Method when the number of iterations required by the gradient method is high (over 1000 for example).

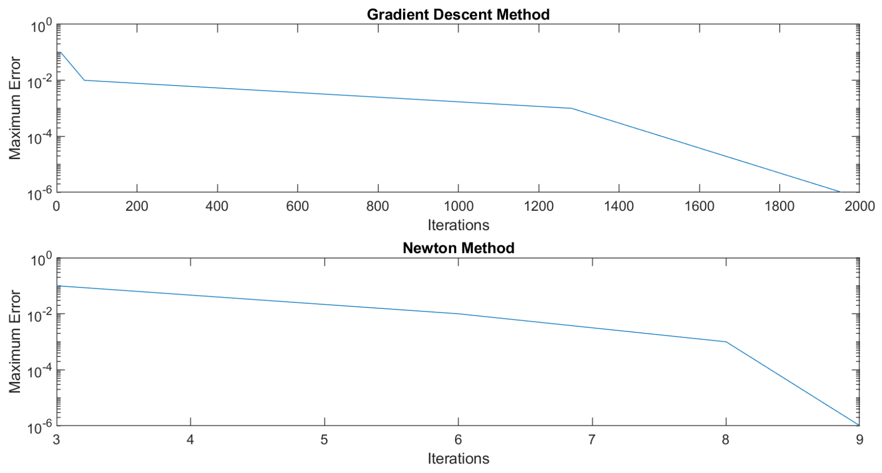
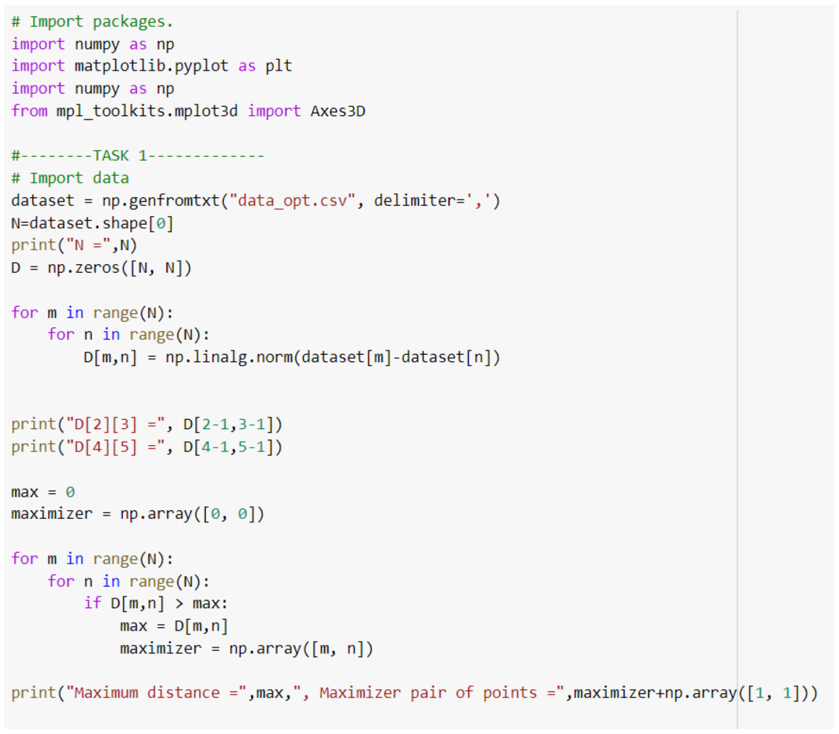


Figure 48: Iterations for different stopping criterion

3 Part 3

3.1 Task 1



```
# Import packages.
import numpy as np
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

#--------TASK 1-------------
# Import data
dataset = np.genfromtxt("data_opt.csv", delimiter=',')
N=dataset.shape[0]
print("N =",N)
D = np.zeros([N, N])

for m in range(N):
    for n in range(N):
        D[m,n] = np.linalg.norm(dataset[m]-dataset[n])


print("D[2][3] =", D[2-1,3-1])
print("D[4][5] =", D[4-1,5-1])

max = 0
maximizer = np.array([0, 0])

for m in range(N):
    for n in range(N):
        if D[m,n] > max:
            max = D[m,n]
            maximizer = np.array([m, n])

print("Maximum distance =",max,", Maximizer pair of points =",maximizer+np.array([1, 1]))
```

Figure 49: Code used in task 1

Which gave us the following results:

- N = 200
- D[2][3] = 5.874898878227682
- D[4][5] = 24.376910709439635
- Maximum distance = 83.00299253869042
- Maximizer pair of points = [33 134]

3.2 Task 2

$$f(y) = \sum_{m=1}^{N}\sum_{n>m}(||y_m - y_n|| - D_{mn})^2 = \sum_{m=1}^{N}\sum_{n>m}(f_{nm}(y))^2 \tag{7}$$

$$\begin{aligned} f_{nm}(y) &= ||y_m - y_n|| - D_{mn} = \sqrt{(y_m - y_n)^T(y_m - y_n)} - D_{mn} \\ &= \sqrt{y_m^T y_m - 2y_m^T y_n + y_n^T y_n} - D_{mn} \end{aligned} \tag{8}$$

Now that we have defined both $f_{nm}(y)$ and $f(y)$, we can compute their gradients:

$$\frac{\partial f_{nm}}{\partial y_m} = \frac{1}{2}(y_m^T y_m - 2y_m^T y_n + y_n^T y_n)^{\frac{-1}{2}}(2y_m - 2y_n) = \frac{y_m - y_n}{||y_m - y_n||} \tag{9}$$

$$\frac{\partial f_{nm}}{\partial y_n} = \frac{y_n - y_m}{||y_m - y_n||} \tag{10}$$

We can conclude that $\nabla f_{nm}(y)$ will be given by a column matrix where line m will be its y_m derivative and its line n will be its y_n derivative, with all other values being 0.

Using this information, we can calculate $\nabla f(y)$:

$$\begin{aligned} \nabla f(y) &= \sum_{m=1}^{N}\sum_{n>m} 2f_{nm}(y)\begin{bmatrix} 0 \\ \frac{\partial f_{nm}}{\partial y_m} \\ \frac{\partial f_{nm}}{\partial y_n} \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} \frac{2(||y_1-y_2||-D_{12})(y_1-y_2)}{||y_1-y_2||} + ... + \frac{2(||y_1-y_N||-D_{1N})(y_1-y_N)}{||y_1-y_N||} \\ \frac{2(||y_1-y_2||-D_{12})(y_2-y_1)}{||y_1-y_2||} + ... + \frac{2(||y_2-y_N||-D_{2N})(y_2-y_N)}{||y_2-y_N||} \\ . \\ . \\ . \\ \frac{2(||y_1-y_N||-D_{1N})(y_1-y_N)}{||y_N-y_1||} + \frac{2(||y_2-y_N||-D_{2N})(y_N-y_2)}{||y_2-y_N||} + ... \end{bmatrix} \end{aligned} \tag{11}$$

Now in order to compute A and b:

$$\underset{y}{\text{minimize}} \sum_{m=1}^{N}\sum_{n>m}(f_{mn}(y_k) + \nabla f_{mn}(y_k)^T(y - y_k))^2 + \lambda_k ||y - y_k||^2$$

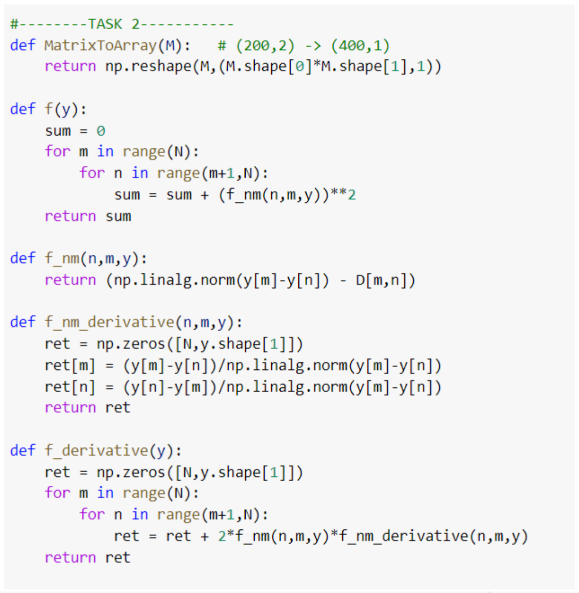
$$\leftrightarrow \underset{y}{\text{minimize}} ||Ay - b||^2$$

We can obtain A and b, where A is a $\frac{N^2-N}{2}$+N*K by N*K matrix and b is a $\frac{N^2-N}{2}$+N*K by 1 matrix. In this step, we have to "flatten" the matrices y_k and $\nabla f_{nm}(y_k)$ so that their shape changes from (N,K) to (N*K,1):

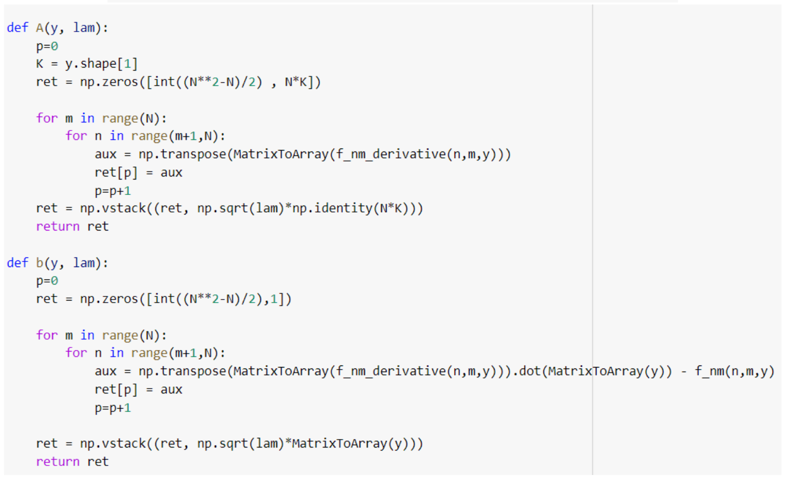
$$A = \begin{bmatrix} \nabla f_{12}(y_k)^T \\ \nabla f_{13}(y_k)^T \\ . \\ . \\ . \\ \nabla f_{1N}(y_k)^T \\ \nabla f_{23}(y_k)^T \\ . \\ . \\ . \\ \sqrt{\lambda_k} I \end{bmatrix} \tag{12}$$

$$b = \begin{bmatrix} \nabla f_{12}(y_k)^T y_k - f_{12}(y_k) \\ \nabla f_{13}(y_k)^T y_k - f_{13}(y_k) \\ . \\ . \\ . \\ \nabla f_{1N}(y_k)^T y_k - f_{1N}(y_k) \\ \nabla f_{23}(y_k)^T y_k - f_{23}(y_k) \\ . \\ . \\ . \\ \sqrt{\lambda_k} y_k \end{bmatrix} \tag{13}$$

Knowing what A and b matrices are, we can translate our problem to code:



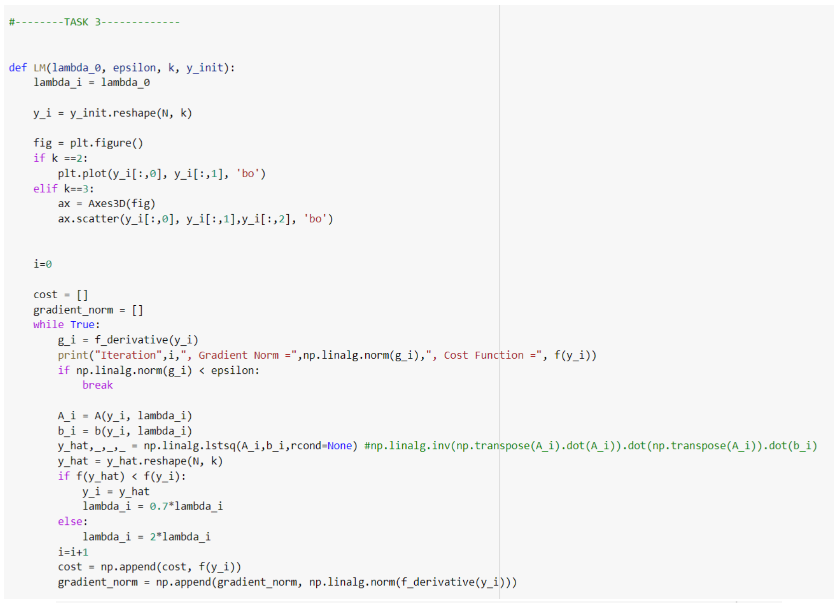
```python
#--------TASK 2-----------
def MatrixToArray(M):   # (200,2) -> (400,1)
    return np.reshape(M,(M.shape[0]*M.shape[1],1))

def f(y):
    sum = 0
    for m in range(N):
        for n in range(m+1,N):
            sum = sum + (f_nm(n,m,y))**2
    return sum

def f_nm(n,m,y):
    return (np.linalg.norm(y[m]-y[n]) - D[m,n])

def f_nm_derivative(n,m,y):
    ret = np.zeros([N,y.shape[1]])
    ret[m] = (y[m]-y[n])/np.linalg.norm(y[m]-y[n])
    ret[n] = (y[n]-y[m])/np.linalg.norm(y[m]-y[n])
    return ret

def f_derivative(y):
    ret = np.zeros([N,y.shape[1]])
    for m in range(N):
        for n in range(m+1,N):
            ret = ret + 2*f_nm(n,m,y)*f_nm_derivative(n,m,y)
    return ret
```



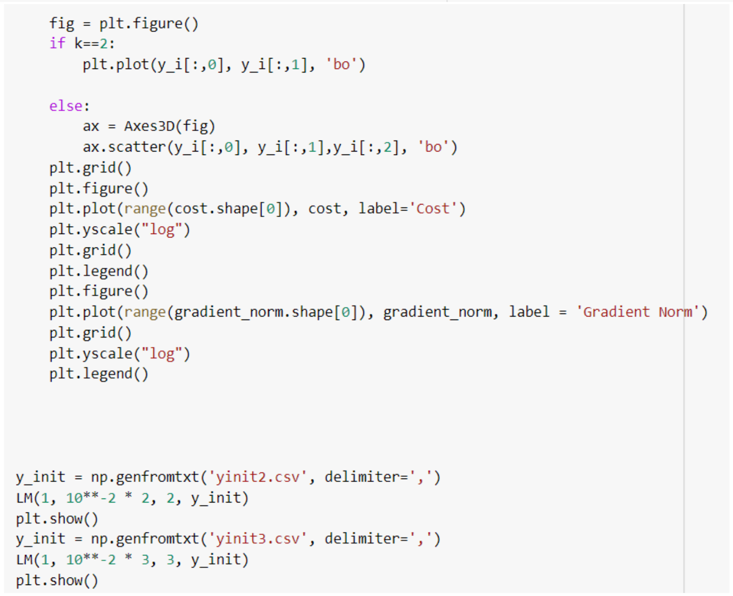
```python
def A(y, lam):
    p=0
    K = y.shape[1]
    ret = np.zeros([int((N**2-N)/2) , N*K])

    for m in range(N):
        for n in range(m+1,N):
            aux = np.transpose(MatrixToArray(f_nm_derivative(n,m,y)))
            ret[p] = aux
            p=p+1
    ret = np.vstack((ret, np.sqrt(lam)*np.identity(N*K)))
    return ret

def b(y, lam):
    p=0
    ret = np.zeros([int((N**2-N)/2),1])

    for m in range(N):
        for n in range(m+1,N):
            aux = np.transpose(MatrixToArray(f_nm_derivative(n,m,y))).dot(MatrixToArray(y)) - f_nm(n,m,y)
            ret[p] = aux
            p=p+1

    ret = np.vstack((ret, np.sqrt(lam)*MatrixToArray(y)))
    return ret
```

Figure 50: Code for task 2

3.3 Task 3



```
#--------TASK 3-------------


def LM(lambda_0, epsilon, k, y_init):
    lambda_i = lambda_0

    y_i = y_init.reshape(N, k)

    fig = plt.figure()
    if k ==2:
        plt.plot(y_i[:,0], y_i[:,1], 'bo')
    elif k==3:
        ax = Axes3D(fig)
        ax.scatter(y_i[:,0], y_i[:,1],y_i[:,2], 'bo')


    i=0

    cost = []
    gradient_norm = []
    while True:
        g_i = f_derivative(y_i)
        print("Iteration",i,", Gradient Norm =",np.linalg.norm(g_i),", Cost Function =", f(y_i))
        if np.linalg.norm(g_i) < epsilon:
            break

        A_i = A(y_i, lambda_i)
        b_i = b(y_i, lambda_i)
        y_hat,_,_,_ = np.linalg.lstsq(A_i,b_i,rcond=None) #np.linalg.inv(np.transpose(A_i).dot(A_i)).dot(np.transpose(A_i)).dot(b_i)
        y_hat = y_hat.reshape(N, k)
        if f(y_hat) < f(y_i):
            y_i = y_hat
            lambda_i = 0.7*lambda_i
        else:
            lambda_i = 2*lambda_i
        i=i+1
        cost = np.append(cost, f(y_i))
        gradient_norm = np.append(gradient_norm, np.linalg.norm(f_derivative(y_i)))
```



```
    fig = plt.figure()
    if k==2:
        plt.plot(y_i[:,0], y_i[:,1], 'bo')

    else:
        ax = Axes3D(fig)
        ax.scatter(y_i[:,0], y_i[:,1],y_i[:,2], 'bo')
    plt.grid()
    plt.figure()
    plt.plot(range(cost.shape[0]), cost, label='Cost')
    plt.yscale("log")
    plt.grid()
    plt.legend()
    plt.figure()
    plt.plot(range(gradient_norm.shape[0]), gradient_norm, label = 'Gradient Norm')
    plt.grid()
    plt.yscale("log")
    plt.legend()




y_init = np.genfromtxt('yinit2.csv', delimiter=',')
LM(1, 10**-2 * 2, 2, y_init)
plt.show()
y_init = np.genfromtxt('yinit3.csv', delimiter=',')
LM(1, 10**-2 * 3, 3, y_init)
plt.show()
```

Figure 51: Code for task 3

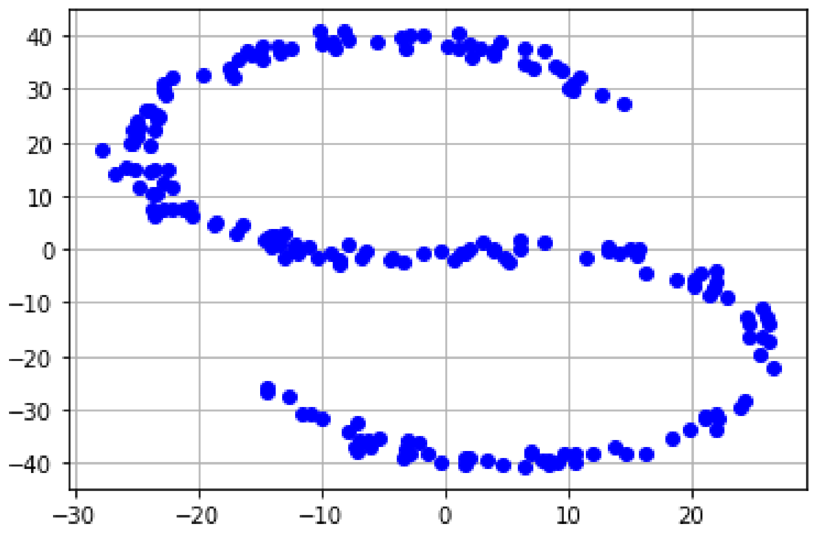


Figure 52: Dimensionality reduction output for k = 2.

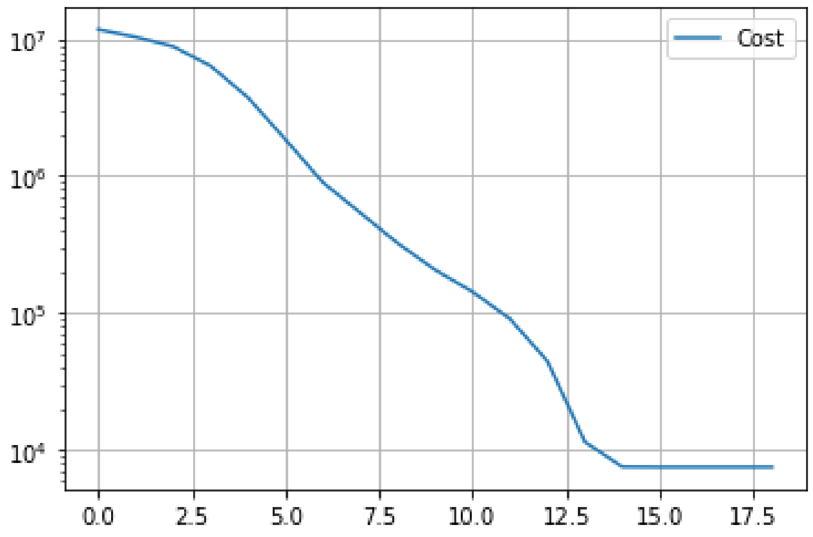


Figure 53: Cost value along iterations of the LM method, when the LM method is applied to problem (1) with the data of file data_opt.csv, k = 2.

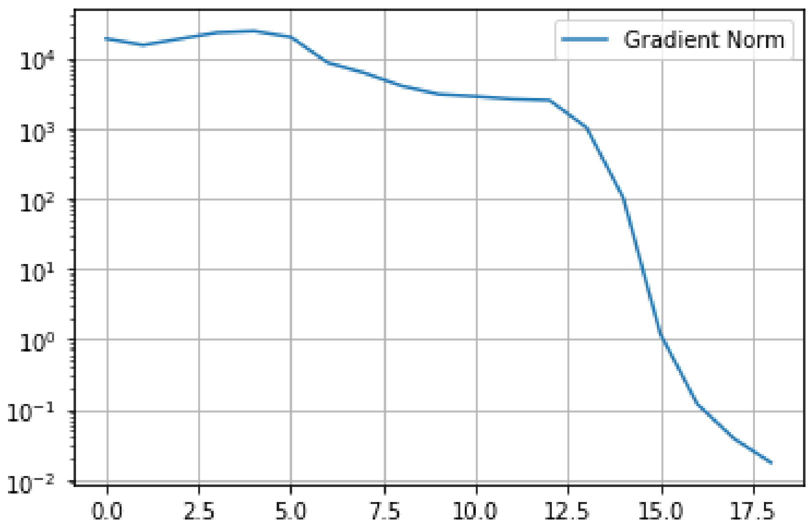


Figure 54: Gradient norm value along iterations of the LM method, k = 2.

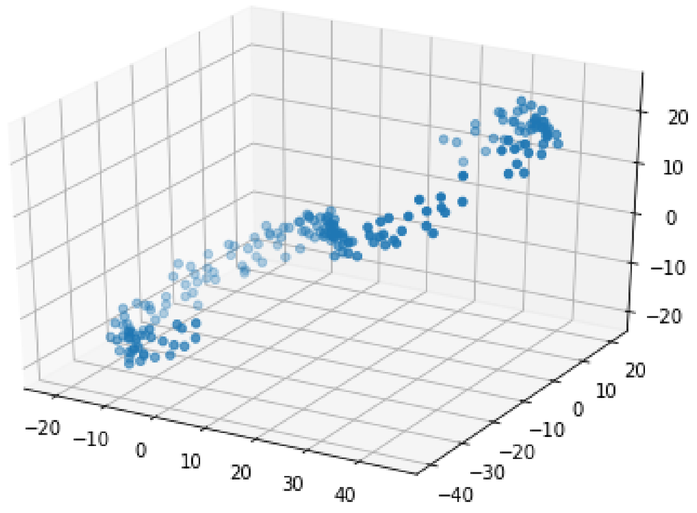


Figure 55: Dimensionality reduction output for k = 3.

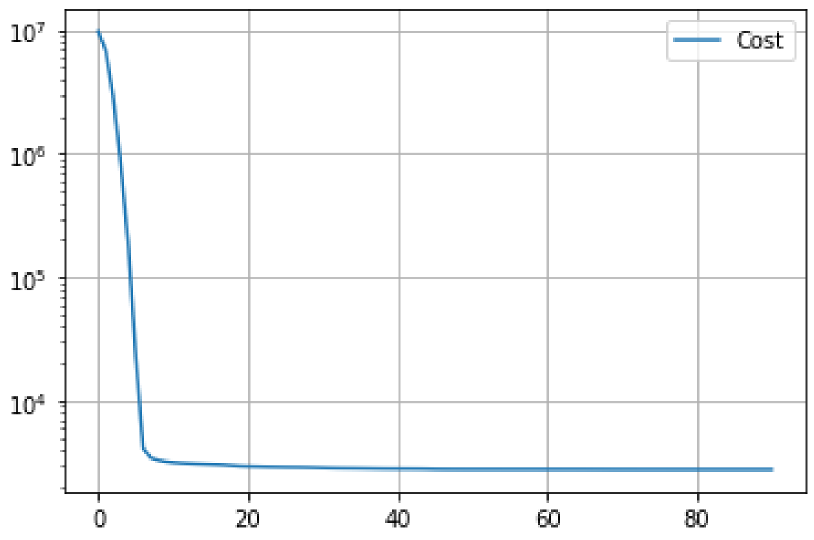


Figure 56: Cost value along iterations of the LM method, when the LM method is applied to problem (1) with the data of file data_opt.csv, k = 3.

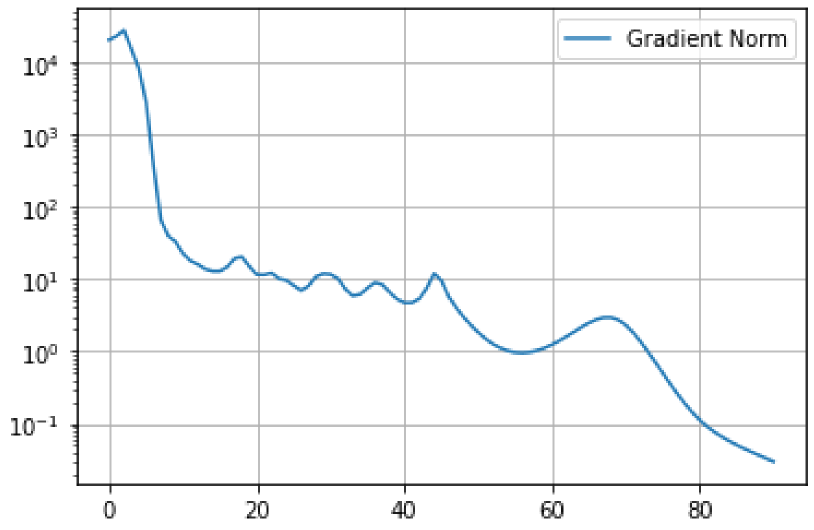
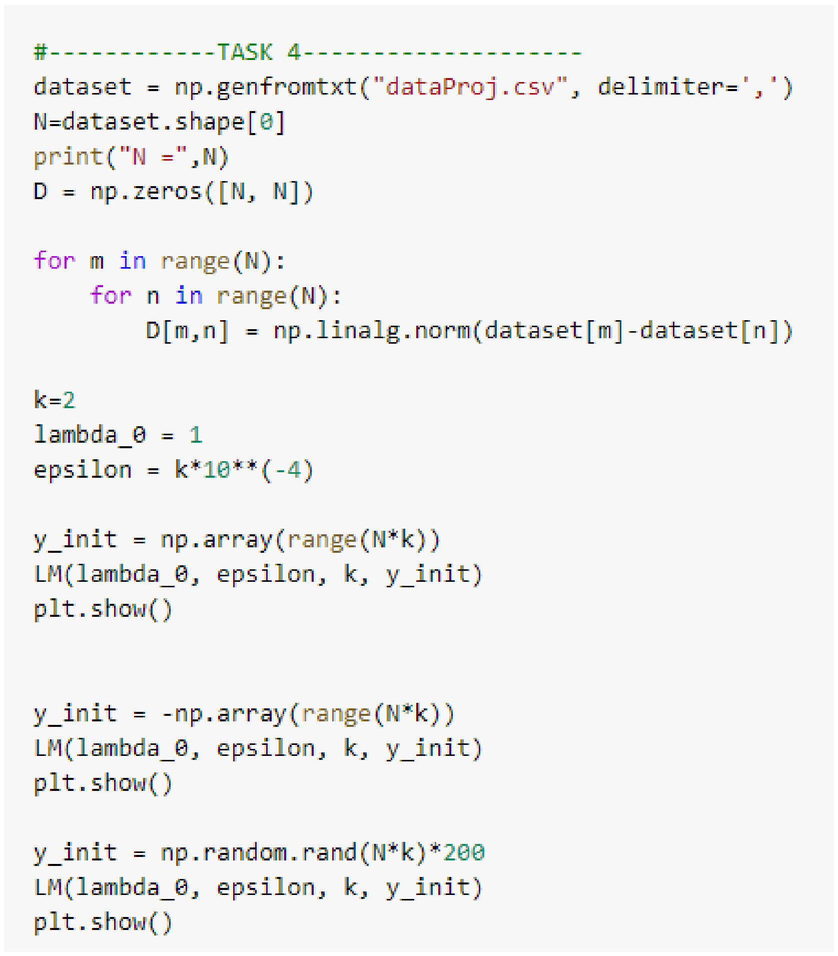


Figure 57: Gradient norm value along iterations of the LM method, k = 3.

3.4 Task 4



```
#------------TASK 4--------------------
dataset = np.genfromtxt("dataProj.csv", delimiter=',')
N=dataset.shape[0]
print("N =",N)
D = np.zeros([N, N])

for m in range(N):
    for n in range(N):
        D[m,n] = np.linalg.norm(dataset[m]-dataset[n])

k=2
lambda_0 = 1
epsilon = k*10**(-4)

y_init = np.array(range(N*k))
LM(lambda_0, epsilon, k, y_init)
plt.show()


y_init = -np.array(range(N*k))
LM(lambda_0, epsilon, k, y_init)
plt.show()

y_init = np.random.rand(N*k)*200
LM(lambda_0, epsilon, k, y_init)
plt.show()
```

Figure 58: Code for task 4

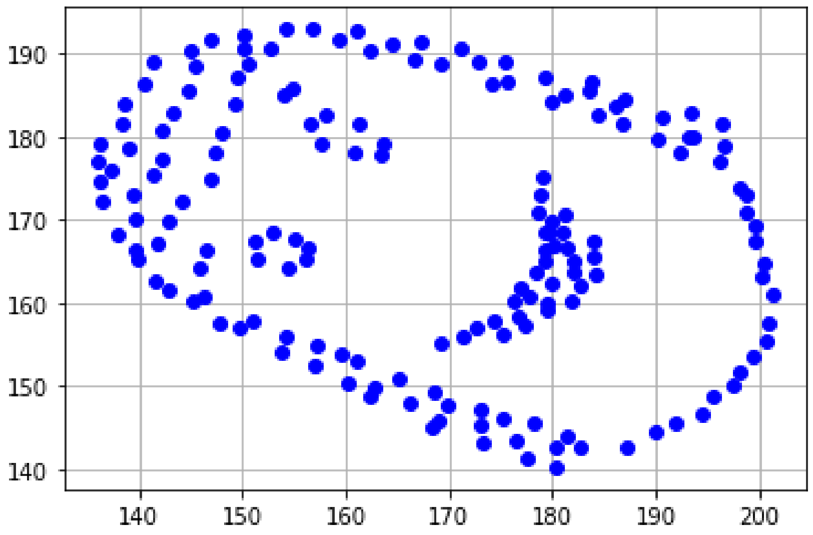


Figure 59: Dimensionality reduction output for our first solution.

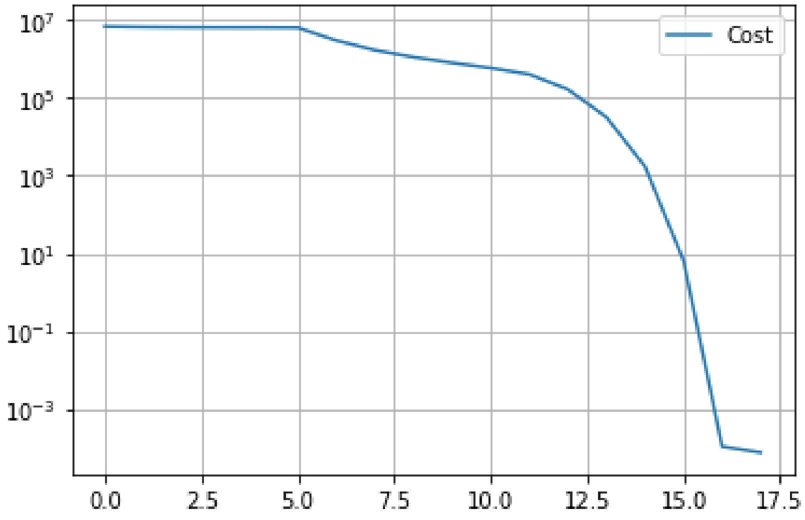


Figure 60: Cost value along iterations of the LM method for our first solution

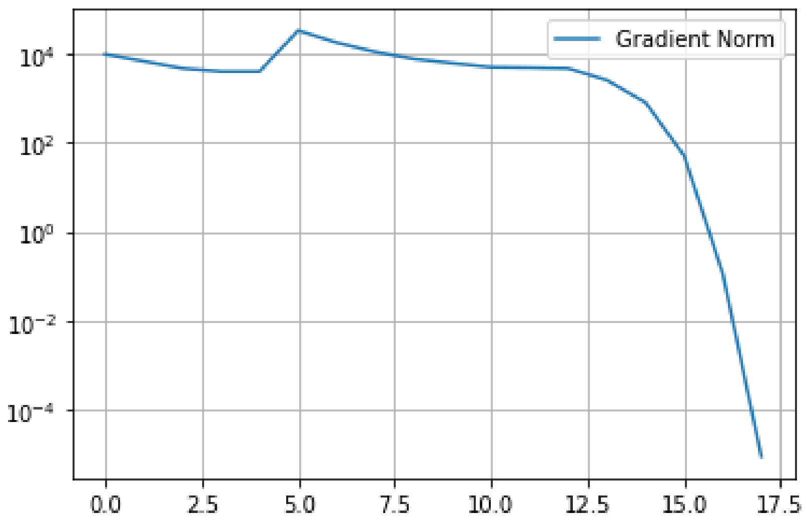


Figure 61: Gradient norm value along iterations of the LM method for our first solution

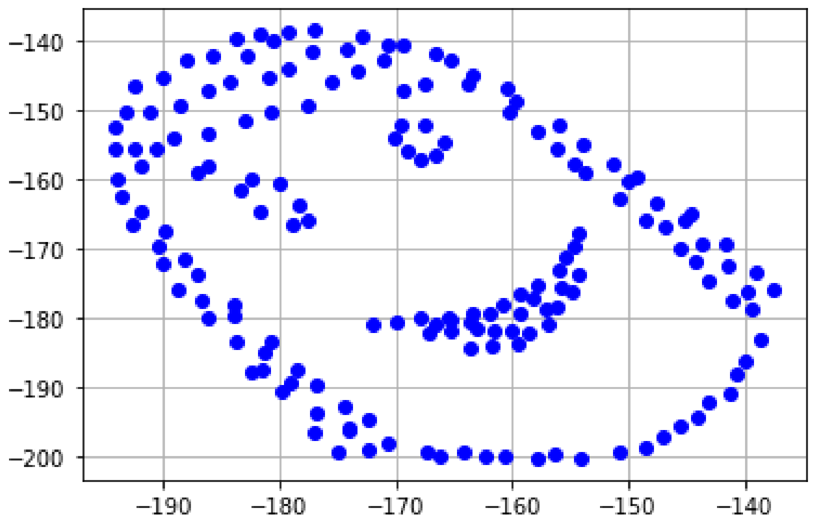


Figure 62: Dimensionality reduction output for our second solution.

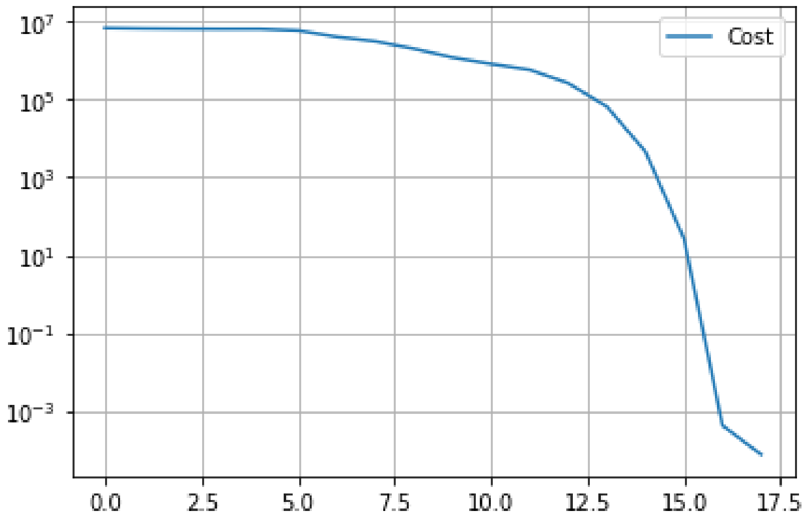


Figure 63: Cost value along iterations of the LM method for our second solution

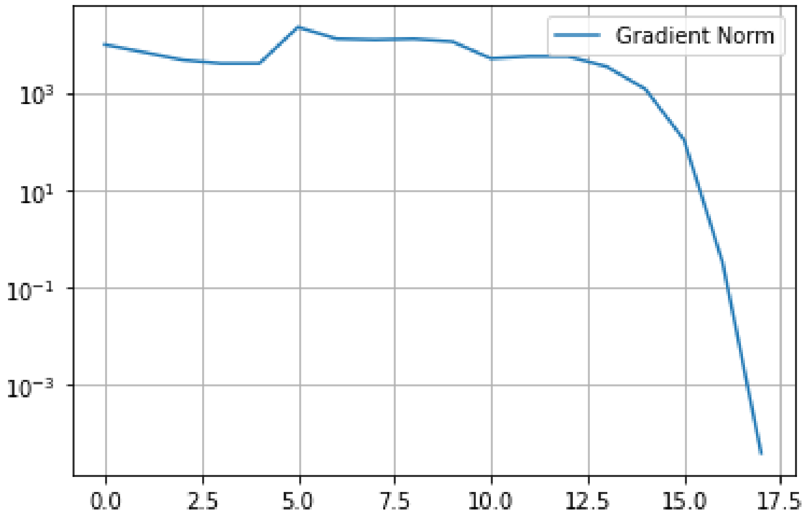


Figure 64: Gradient norm value along iterations of the LM method for our second solution

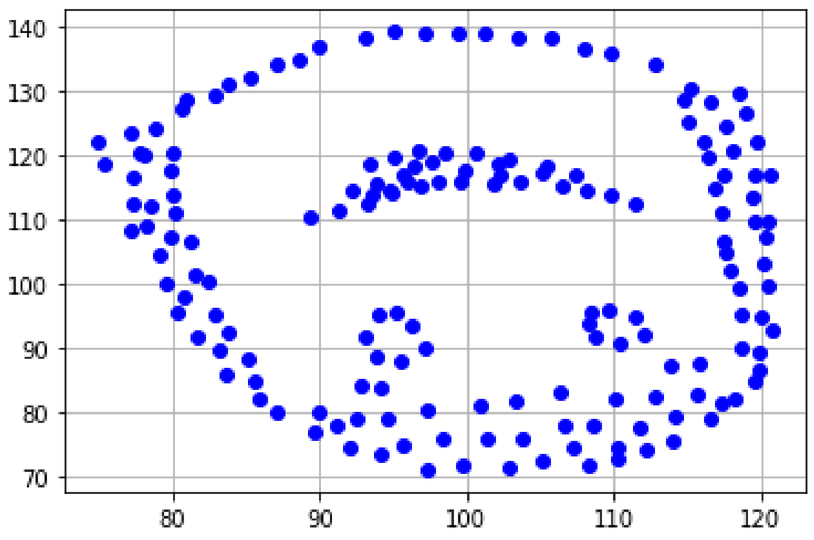


Figure 65: Dimensionality reduction output for our third solution.

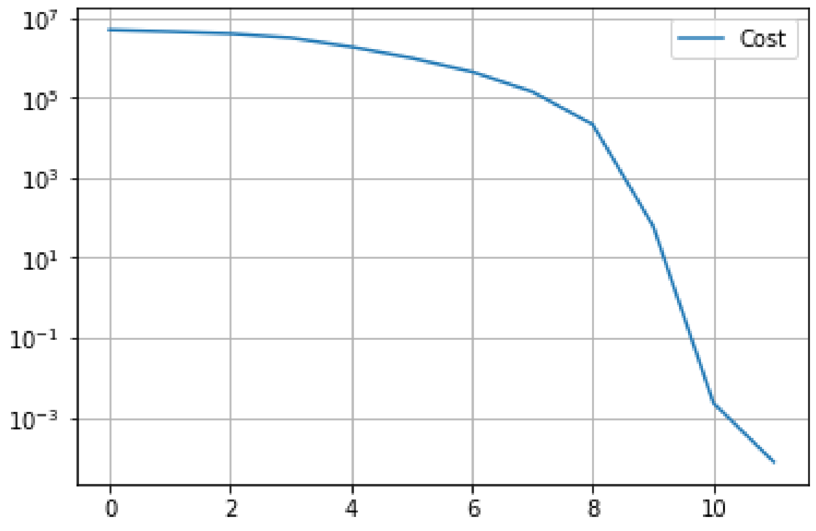


Figure 66: Cost value along iterations of the LM method for our third solution

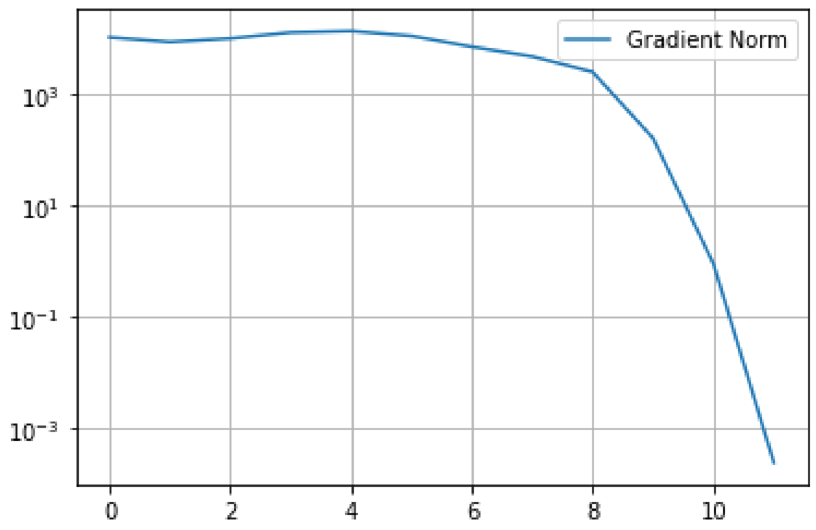


Figure 67: Gradient norm value along iterations of the LM method for our third solution

As we can see, for different initializations, we obtain different solutions for this problem. This leads us to conclude that the function

$$f(y) = \sum_{m=1}^{N} \sum_{n>m} (||y_m - y_n|| - D_{mn})^2$$

where matrix D is obtained from the data in the file "dataProj.csv", has more than one minimizer and the LM method reaches a different minimizer depending on its initialization. This means that the solution for this problem is not unique. This can also be concluded just by looking at function f. This function does not depend on the absolute values of the vectors y_i, only on the relative positions of each pair of vectors (y_n,y_m), so a rotation/translation of the vectors will produce the same value for f, which is exactly what is happening in the figures obtained.