Optimization and Algorithms
Project report

Group 25
Matyas Skalicky 94904, Ana Carolina Lima 83993,
Manuel Freitas 85246 and Carina Fernandes 84019

1 Part 1

1.1 Task 1

The code used to solve the task 1 is described in the Listing 1. `vec_sqr_sum` function is described in the Listing 2. The results of the task 1 for $\lambda \in \{10^{-3}, 10^{-2}, 10^{-1}, 10^{0}, 10^{1}, 10^{2}, 10^{3}\}$ are described in the Table 1. The robot positions and control are visualized in the Figure 1.

Listing 1: Code for the Task 1.

```
tau = [10 25 30 40 50 60];
w   = [10 20 30 30 20 10;
       10 10 10 0  0 -10];
A   = [1.0 0.0 0.1 0.0;
       0.0 1.0 0.0 0.1;
       0.0 0.0 0.9 0.0;
       0.0 0.0 0.0 0.9];
B   = [0.0 0.0;
       0.0 0.0;
       0.1 0.0;
       0.0 0.1];
E   = [1 0 0 0;
       0 1 0 0];
p_initial = [ 0;   5];
p_final   = [15; -15];
U_max = 100;
T = 80;

for i = 0:7
    lambda = 10^(-3+i);
    cvx_begin
        variable x(4,T+1);
        variable u(2,T);

```

```
25          % Minimize the objective function
26          Δ = u(:, 2:T) − u(:, 1:T−1);
27          minimize(sum(vec_sqr_sum(E*x(:,tau+1) − w))...
28              + lambda*sum(sum_square(Δ)))
29
30          subject to
31              % Initial and end speed need to be 0
32              x(:,1)     == [p_initial; [0; 0]]
33              x(:,T+1)   == [p_final;   [0; 0]]
34              % Make sure that the robot moves using the constrains
35              x(:,2:T+1) == A*x(:,1:T) + B*u(:,1:T);
36              % Check the actuator force size
37              for ux = u
38                  norm(ux, 2) ≤ U_max;
39              end
40          cvx_end
41
42      % Check how many times the control signal changes
43      count = control_signal_changes(u, T);
44
45      % Get the mean deviation
46      meandev = sum(vecnorm(x(1:2, tau+1) − w, 2, 1)) / length(tau);
47  end
```

Listing 2: vec_sqr_sum function used in the Task 1.

```
1  function [norm_col] = vec_sqr_sum(A)
2      % vec_sqr_sum squares elementwise and sums all the columns for a matrix
3      A = A.^2;
4      for i = 1:length(A(1,:))
5          aux = sum(A(:,i));
6          norm_col(i) = aux;
7      end
8  end
```

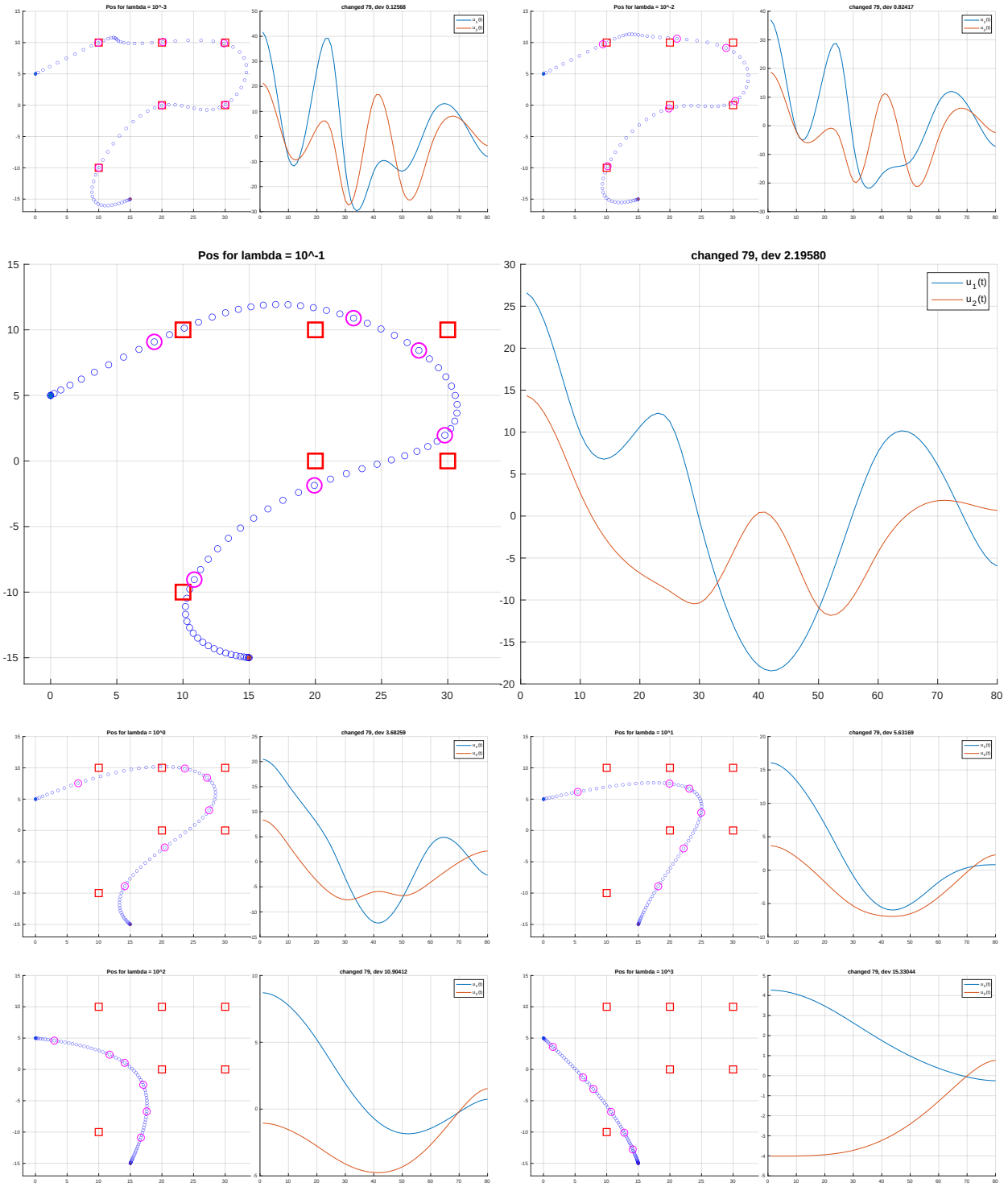

Figure 1: Robot positions and control signal for Task 1.

1.2 Task 2

Code utilized in the task 2 is same as in Task 1, except for the objective function, which is defined as described in Listing 3. The results are described in the Table 2. The robot positions and control signal are visualized in the Figure 2.

Listing 3: Objective function used in Task 2.

```
minimize(sum(vec_sqr_sum(E*x(:,tau+1) - w))...
     + lambda*sum(norms(Δ, 2)));
```

1.3 Task 3

Listing 4: Objective function used in Task 3.

```
minimize(sum(vec_sqr_sum(E*x(:,tau+1) - w)...
                 ) + lambda*sum(norms(Δ, 1)));
```

Code utilized in the task 3 is same as in Task 1, except for the objective function, which is defined as described in Listing 4. The results are described in the Table 3. The robot positions and control signal are visualized in the Figure 3.

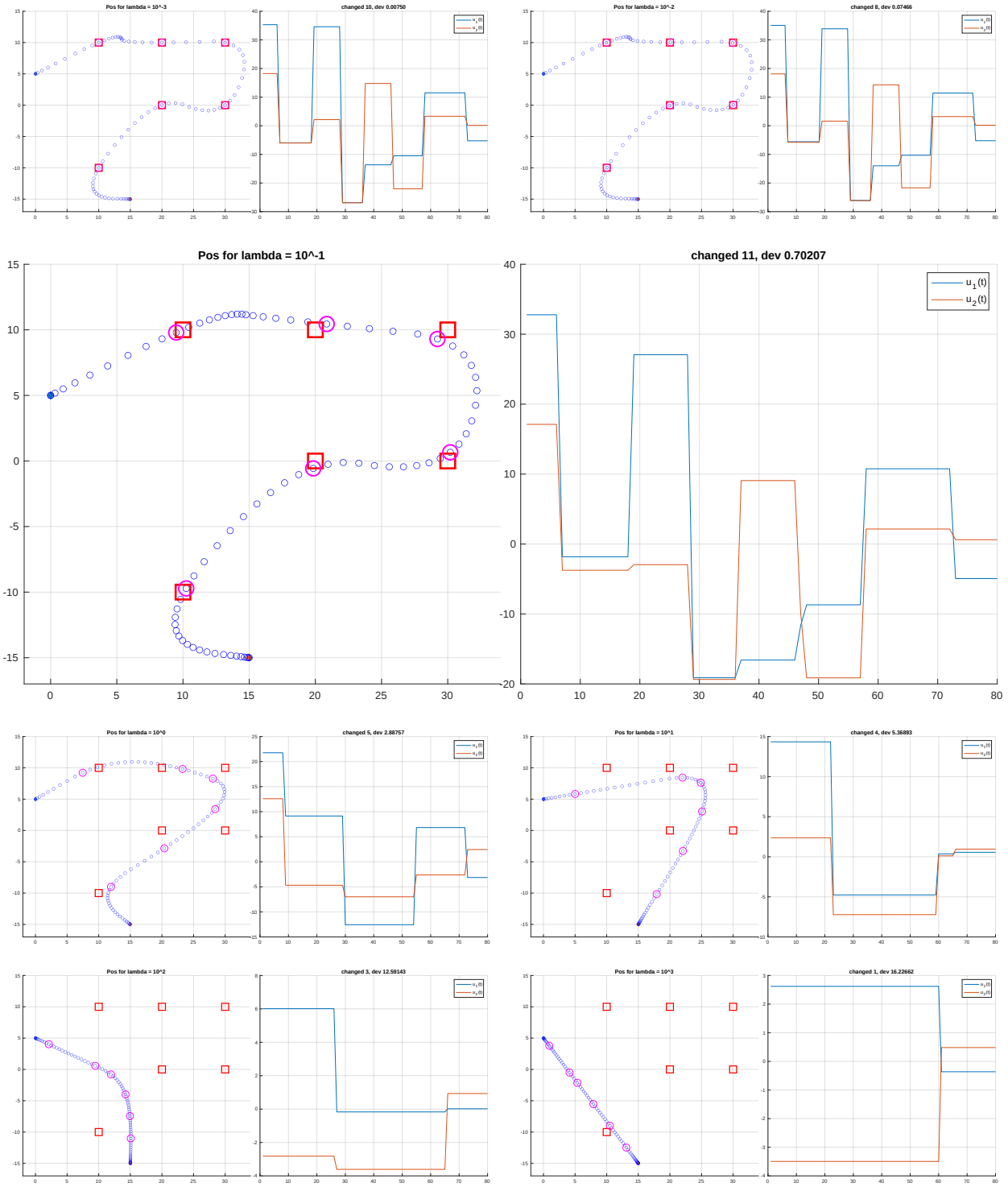

Figure 2: Robot positions and control signal for Task 2.

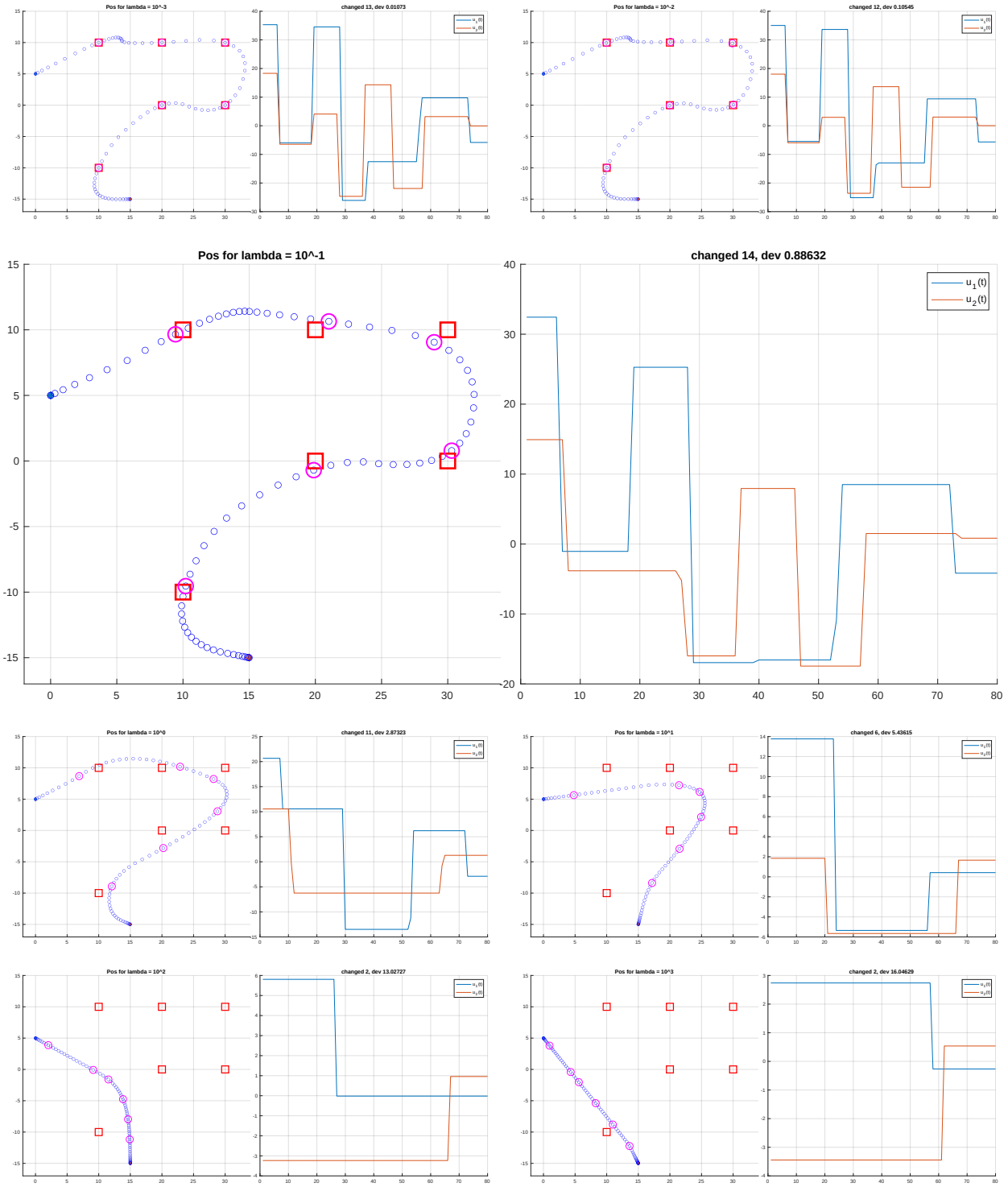

Figure 3: Robot positions and control signal for Task 3.

1.4 Task 4

Mean deviation and effect of the regularizer based on the λ in Tasks 1 to 3.

λ	changes	m. dev.
10^{-3}	79	0.1257
10^{-2}	79	0.8242
10^{-1}	79	2.1958
10^{+0}	79	3.6826
10^{+1}	79	5.6317
10^{+2}	79	10.9041
10^{+3}	79	15.3304

Table 1: Task 1 results.

λ	changes	m. dev.
10^{-3}	10	0.0075
10^{-2}	8	0.0747
10^{-1}	11	0.7021
10^{+0}	5	2.8876
10^{+1}	4	5.3689
10^{+2}	3	12.5914
10^{+3}	1	16.2266

Table 2: Task 2 results.

λ	changes	m. dev.
10^{-3}	13	0.0107
10^{-2}	12	0.1055
10^{-1}	14	0.8863
10^{+0}	11	2.8732
10^{+1}	6	5.4361
10^{+2}	2	13.0273
10^{+3}	2	16.0463

Table 3: Task 3 results.

The objective function which we try to minimize in the tasks 1 to 3 consists of sum of two parts. The first part, common to all of three tasks measures how far away the robot is from the waypoints at given times. This is defined as $\sum_{k=1}^{K} \|Ex(\tau_k) - w_k\|_2^2$.

Second part of the objective function, the regularizer, differs among the first three tasks. Regularizer enforces the fourth wish (simple control) by penalizing the deviations of the control signal from its previous value. By changing the λ parameter, we can change the importance of this wish. If the λ becomes a large number, the objective function's value shifts towards the regularizer which enforces least number of changes to the control signal. Making the λ a large value naturally increases the mean deviation as we put larger importance to minimizing the number of changes of the control signal instead of capturing all of the points.

$$\sum_{t=1}^{T-1} \|u(t) - u(t-1)\|_2^2 \quad (1) \qquad \sum_{t=1}^{T-1} \|u(t) - u(t-1)\|_2 \quad (2) \qquad \sum_{t=1}^{T-1} \|u(t) - u(t-1)\|_1 \quad (3)$$

Effect of this behavior is shown in the Figure 5. The number of changes in the control signal does not change for the Task 1 as it stays 79. The number of changes for Task 2 and 3 is visualized in the Figure 4.

Task 1 utilizes the squared L2 regularizer as shown in Equation 1. Benefit of using the squared L2 norm can be in easier computation, as no square root needs to be computed. The differences between control signals are absolutely limited by the U_Max constraint which results in them being often lower than 1. Squaring a small number leads to even smaller value. Since the value is too small, the contribution to the objective function is neglected and the optimizer rather optimizes capturing the waypoints.

Task 2 utilizes the L2 norm, described in Equation 2. This can be interpreted as euclidean distance from the previous actuator vector to the current one. Calculating the square root of a small value (< 0) makes this value larger. The contribution to the objective function is therefore larger than in the Task 1. The lowest number of updates (1)

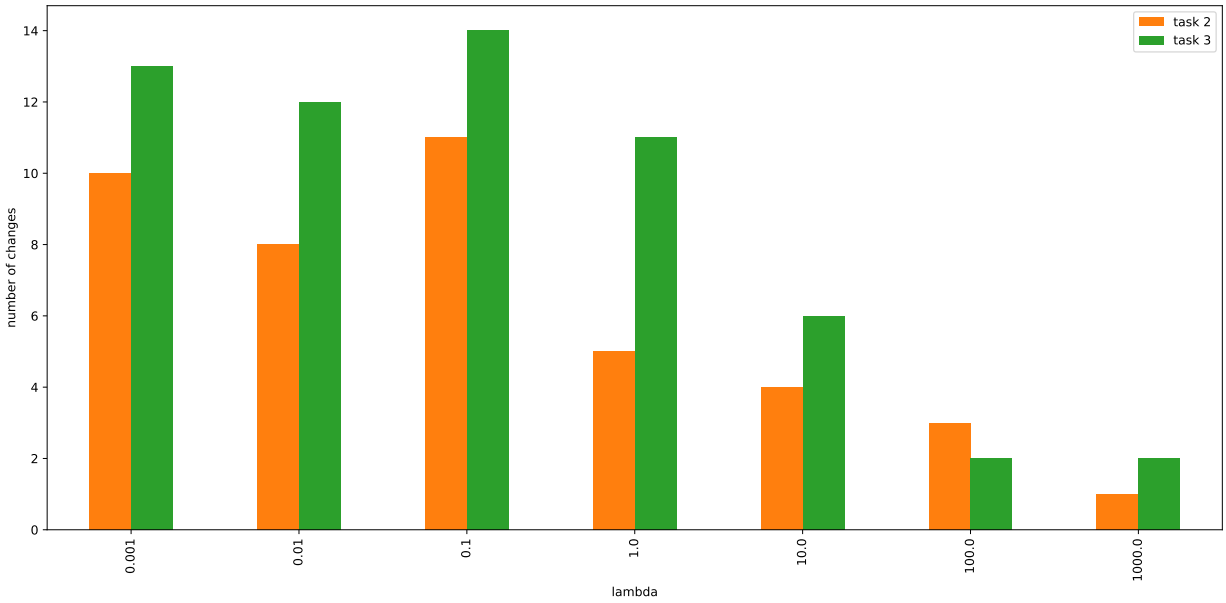

Figure 4: Control signal changes for Tasks 2 and 3.

was achieved by using the $\lambda = 10^3$. We can see, that the control signal's function is no longer smooth, but rather has a form of squares as the robot utilizes large control signals which change rarely.

Task 3 relies on the L1 norm, as described in Equation 3. This norm is not differentiable at 0, however it induces a behavior which was called "L1 magic" - the property of producing many coefficients with zero values or very small values with few large coefficients[1].

Surprisingly, the performance of the L2 norm in this task is superficial to the L1 norm as the robot achieves lower mean deviation with the same number of points captured.

1.5 Task 5

Distance between point p and disc $D(c, r)$ is defined as:

$$d(p, D(c, r)) = min\{\|p - y\| : y \in D(c, r)\} \tag{4}$$

Based on whether the point is within the disc or not, the following cases might occur:

$$d(p, D(c, r)) = \begin{cases} 0 & \text{if } \|p - c\|_2 \leq r \\ \|p - c\|_2 - r & \text{if } \|p - c\|_2 > r \end{cases} \tag{5}$$

[1] http://www.chioka.in/differences-between-the-l1-norm-and-the-l2-norm-least-absolute-deviations-and-least-squares/

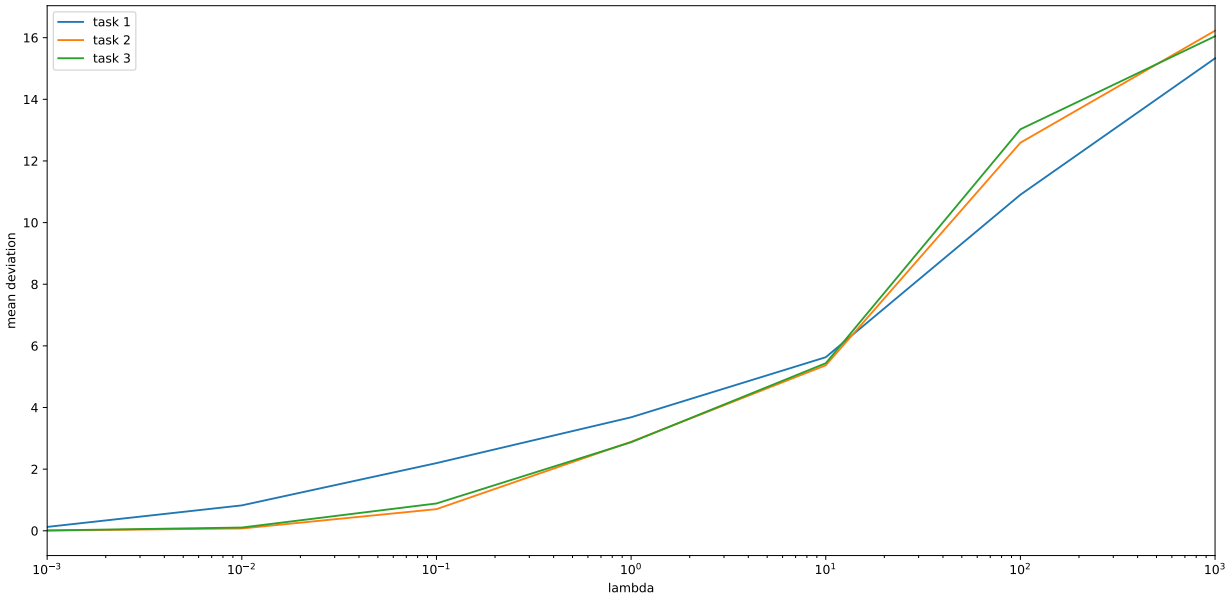

Figure 5: Waypoint mean deviation for Tasks 1 to 3.

If the result of $\|p - c\|_2 - r$ is negative, the point is within the circle and the distance should be 0. If the point is outside of the disc, then the result is equal to $\|p - c\|_2 - r$. Based on this, we conclude that the equation can be written in a closed form as:

$$d\left(p, D\left(c, r\right)\right) = max\left(0, \|p - c\|_2 - r\right) \tag{6}$$

1.6 Task 6

Mean deviation and number of changes for the Task 6 is shown in the Table **??**. The resulting robot's optimal positions are shown in the Figure 6. The snippet of the CVX code is shown in the Listing 5.

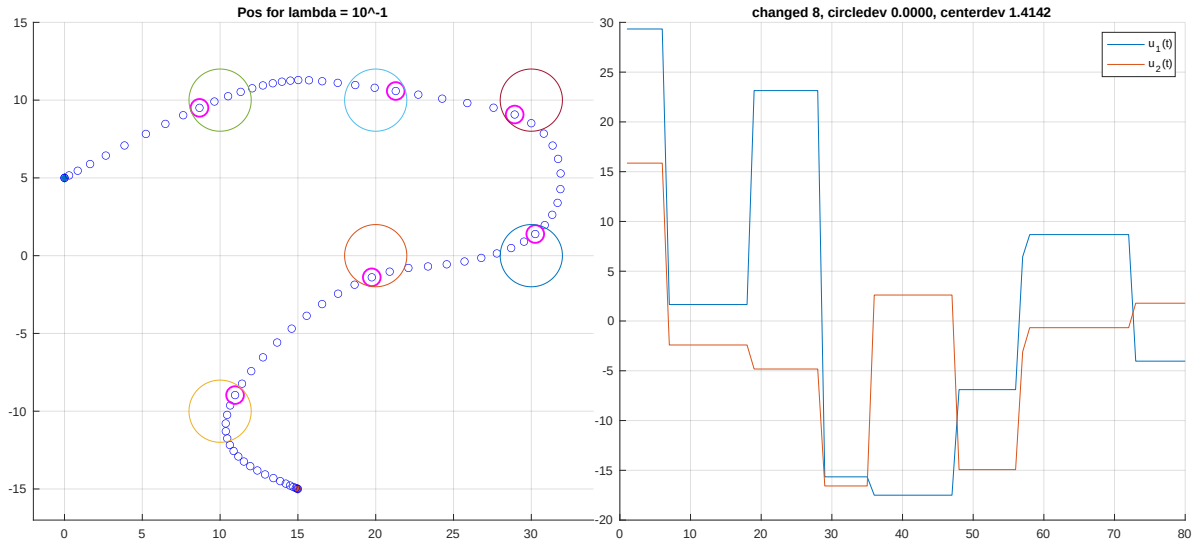

Figure 6: Robot positions and control signal for Task 6.

The robot is now not rewarded for directly capturing the waypoint, it only needs to be within the disc's radius at the given time. With $\lambda = 10^{-}1$ the robot achieves 8 changes as compared to 11 changes in the Task 2. The center mean deviation is 1.4141 as compared to 0.7021 achieved in the Task 2. We can see, that by not forcing the robot not to be at exact position at the given time, we have relaxed one of our wishes and as a tradeoff, the robot is able to achieve the task with lower number of signal changes. Of course, this comes at a cost of larger mean deviation.

λ	changes	circle m. dev.	center m. dev.
10^{-1}	8	0	1.4142

Table 4: Mean deviation and number of changes for Task 6.

Listing 5: CVX code for task 6.

```
1 minimize(sum(max(0, vec_sqr_sum(c - x(1:2, tau+1)) - r))...
2     + lambda*sum(norms(Δ, 2)));
```

1.7 Task 7

We implemented the task 7 in CVX code as shown in the listing 6. The variables are same as in the previous exercises with the exception of U_Max = 15.

Listing 6: CVX code for task 7.

```
cvx_begin
    variable x(4,T+1);
    variable u(2,T);
    minimize(0);
    subject to
        % Initial and end speed need to be 0
        x(:,1)     == [p_initial; [0; 0]];
        x(:,T+1)   == [p_final;   [0; 0]];
        % Make sure that the robot moves using the constrains
        x(:,2:T+1) == A*x(:,1:T) + B*u(:,1:T);
        % Check the actuator force size
        for ux = u
            norm(ux, 2) ≤ U_max;
        end
        % Enforce the correct positions of the robot
        for ti = 1:length(tau)
            E*x(:, tau(ti)) ==  w(:, ti);
        end
cvx_end
```

The CVX tries to solve this task, but fails to do so and reports the following:

```
Status: Infeasible
Optimal value (cvx_optval): +Inf
```

This indicates, that CVX has not found a feasible solution to the given constrains. This is likely due to the actuator force U_Max = 15 being too low.

We have investigated setting the U_Max as another CVX variable and optimizing it with `minimize(U_max)` which returns 38.8989. We can see, that with value of U_Max = 38.8989 this feasibility problem has a solution.

1.8 Task 8

The function ϕ outputs zero if its argument is zero. Otherwise it outputs one. Function $\phi : \mathbf{R}^2 \to \mathbf{R}$, is defined as

$$\phi(x) = \begin{cases} 0, & \text{if } x = 0, \\ 1, & \text{if } x \neq 0. \end{cases}$$

$$\forall x_1, x_2 \in X, \forall t \in [0,1] : \qquad f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2) \tag{7}$$

Function f is convex if for any two points x_1, x_2 and a parameter $t \in [0,1]$ the above inequality is true. We select $x_1 = 0$, $x_2 = 1$ and $t = 0.5$. The inequality for this setting and function ϕ is described below.

$$\begin{aligned} \phi(t \cdot x_1 + (1-t) \cdot x_2) &\leq t \cdot \phi(x_1) + (1-t) \cdot \phi(x_2) \\ \phi(0.5 \cdot 0 + 0.5 \cdot 1) &\leq 0.5 \cdot \phi(0) + 0.5 \cdot f(1) \\ \phi(0.5) &\leq 0.5 \cdot \phi(0) + 0.5 \cdot f(1) \\ 1 &\leq 0.5 \end{aligned} \tag{8}$$

As we can see above, this inequality is a contradiction for selected x_1, x_2 and t. Since ϕ does not fulfill the requirements for a convex function, we conclude that it is non convex.

1.9 Task 9

The minimized function can be expressed in CVX as shown on Listing 7.

Listing 7: CVX code for task 9.

```
minimize(sum(vec_sqr_sum(E*x(:,tau+1) - w)));
```

The robot positions are shown in the Figure 7. The robot did not capture any point.

1.10 Task 10

The minimized function can be expressed in CVX as shown on Listing 8. The robot has captured a single point along its journey.

Listing 8: CVX code for task 10.

```
minimize(sum(norms(E*x(:,tau+1) - w, 2)));
```

The resulting robot positions are visualized in the Figure 8.

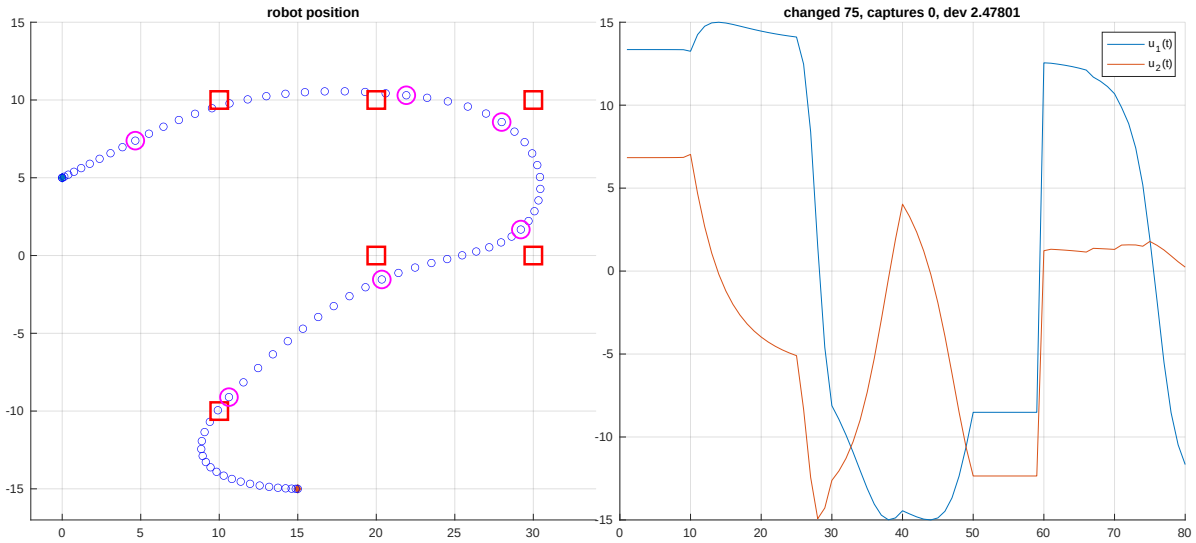

Figure 7: Robot positions and control signal for Task 9.

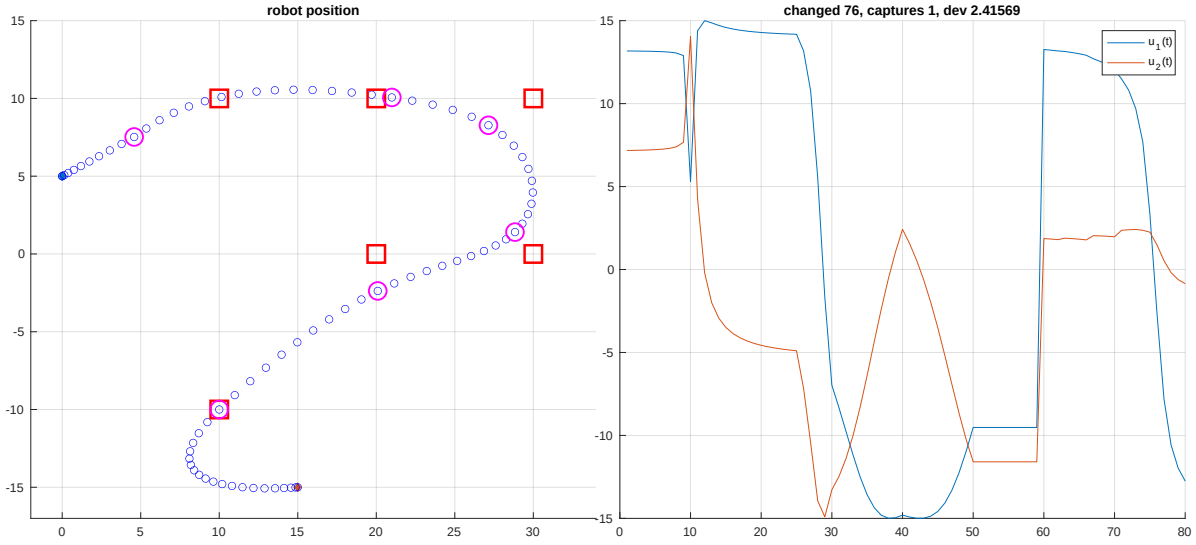

Figure 8: Robot positions and control signal for Task 10.

1.11 Task 11

The solution to the reweighting problem is visualized in the Figure 9. The results are described in the Table 5.

m	captured	mean deviation
0	1	2.4157
1	2	2.4567
2	2	2.5737
3	3	2.7362
4	3	2.7355
5	3	2.7355
6	3	2.7355
7	3	2.7355
8	3	2.7355
9	3	2.7355

Table 5: Results of the reweighting technique in Task 11.

1.12 Task 12

The optimizer tries to optimize the utility function of each waypoint divided by the previous value. The utility values for first 5 iterations are shown in Listing 9.

Listing 9: Unweighted utility function for each waypoint.

```
1     5.9580    0.9926    3.3379    1.8267    2.3790    0.0000
2     6.0346    0.0000    4.3660    1.3281    3.0118         0
3     6.0892         0    4.4750    0.3854    4.4923         0
4     6.2897         0    4.4125    0.0000    5.7152         0
5     6.2540         0    4.3987    0.0000    5.7602         0
6     6.2514         0    4.3983    0.0000    5.7630         0
7     6.2512         0    4.3983    0.0000    5.7632         0
8     6.2512         0    4.3983    0.0000    5.7632         0
9     6.2512         0    4.3983    0.0000    5.7632         0
10    6.2512         0    4.3983    0.0000    5.7632         0
```

By investigating the resulting utility function sizes for each waypoint, we can see that if the robot captures the waypoint, the utility becomes 0. Since the optimizer tries to minimize the sum of the utility function for each waypoint, thanks to the reweighting technique, it already has the information that the waypoint can be captured (in form of the previous iteration's utility value). The previous utility value was 0 and when we divide any number by a very small number ($\epsilon = 10^{-6}$) this fraction explodes thus forcing the optimizer to keep this value small by setting the numerator 0 again.

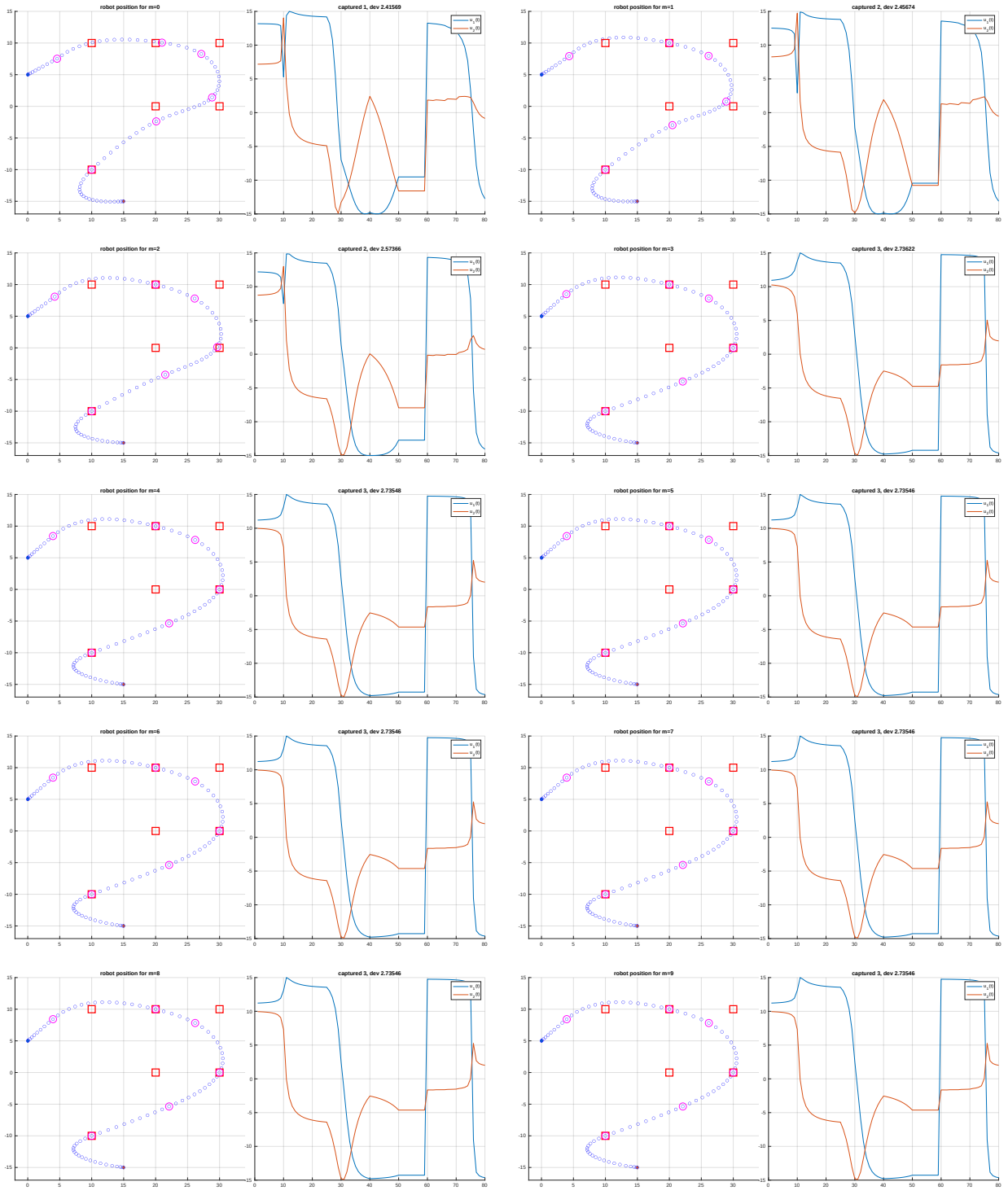

Figure 9: Robot positions and control signal for Task 11.

2 Part 2

In this part, it is required to minimize $f(s,r)$, represented in equation 9,using various methods to give optimal values of (s,r).

$$f(s,r) = \sum_{k=1}^{K} \log(1 + \exp(s^T x_k - r)) - y_k(s^T x_k - r) \tag{9}$$

2.1 Task 1

To prove the convexity of the function, it is easier to breakdown function F, in equation 9 and figure 10 as a composition of smaller functions:

$$\begin{aligned} F &= F_1 + \ldots + F_k \\ F_1 &= G_1 + H_1 \\ G_1 &= Q_1(F_1) \end{aligned} \tag{10}$$

Where

$$\begin{aligned} H_1 &= -y_1(s^T x_1 - r) \\ P_1 &= s^T x_1 - r \\ Q_1 &= log(1 + e^z) \end{aligned} \tag{11}$$

P_1 can be expressed as an inner product between the vector of variables (s,r) (equation 12) and the vector constants x_1 . This is also the trick used in the implementation of the Gradient and Newton methods, so that only matrix and vector calculations take place. It is affine, therefore convex.

$$\begin{bmatrix} x_1^T & -1 \end{bmatrix} \begin{bmatrix} s \\ r \end{bmatrix} \tag{12}$$

The composed function $Q_1(F_1)$ is convex, because $\ddot{Q}_1(z) > 0, \forall z \in \mathbb{R}$.
The same thought process of equation 12 can be applied to H_1, but in this case the constants also take into account y_1. The sum of a convex function to another $(H_1 + G_1)$ only changes the offset of the function images, so it still remains convex.

$$\begin{bmatrix} -y_1 x_1^T & y_1 \end{bmatrix} \begin{bmatrix} s \\ r \end{bmatrix} \tag{13}$$

If F_1 is convex, then all of the F_k are convex too, since all of them have the same structure. Therefore, their sum is also convex, making F a convex function.

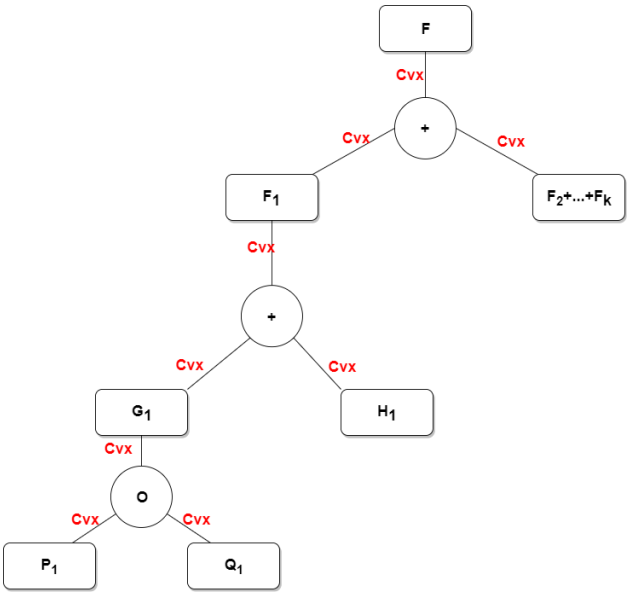

Figure 10: Schematic breakdown of $f(s, r)$, represented by F

2.2 Task 2

The code for the Gradient method can be observed at listing 10. It is worth noticing that a strategy like equation 12 was enforced, so as to simplify and optimize the calculations, without for-loops.

Listing 10: Matlab code for the Gradient method.

```
function [S] = grad_descent_2(X,Y,s0,r0,e)
%grad_descent_2 applies the gradient method
%   Inputs : X,Y,e= dataset,labels,error tolerance
%            s0,r0= initial s, initial r
%   Outputs: S = [X_features+1]*[k] matrix, where S(:,end) is the final
%            value; [S(1:end-1,:); S(end,:)]= [values of s ; values of r]
%            troughout the method
k=1;
%Compactly defines an affine mapping, so that matricial notation can be
%employed . [s;r]'*[x;-1] = s'*x - r
S(:,1)=[s0;r0];
X(end+1,:)=-1;
%define functions f, and its first derivative in order of s'*x
d_f=@(x,s,y)(exp(s'*x)./(1+exp(s'*x))) - y ;
f=@(x,s,y) sum(log(1+ exp(s'*x)) - y.*(s'*x))/length(x);
while(1)
    %calculate gradient
    g=(X*(d_f(X,S(:,k),Y)'))/length(X);
    grad_norm(k)=norm(g);
    if(norm(g) < e)
        break;
    else
```

```
        d=-g;
        %backtracking routine
        a(k)=1;
        while(1)
            cond2= f(X,S(:,k),Y)+(10^-4)*g'*(a(k)*d);
            cond1= f(X,S(:,k)+a(k)*d,Y);
            if(cond1<cond2)
                break;
            end
            %step size recalculation
            a(k)=a(k)*0.5;
        end
        % calculation of k(th) iteration of the values of s and r
        S(:,k+1)=S(:,k)+a(k)*d;
        k=k+1;
    end
end
%Norm graph
figure
semilogy(1:k,grad_norm);
xlabel('$k$','interpreter','latex')
title('$||\nabla{f(s_k,r_k)}||$ (Gradient Method)','interpreter','latex')
grid on;
end
```

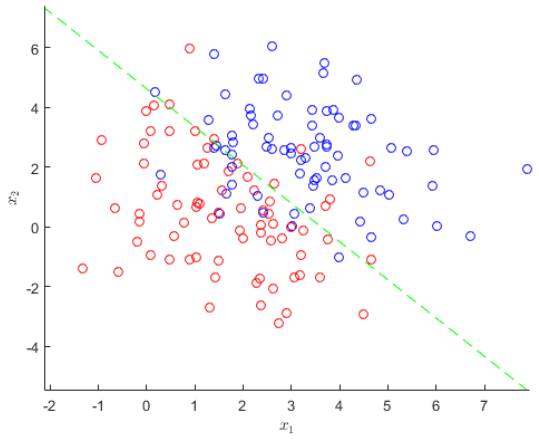

Figure 11: `data1.mat` (blue,red) with optimized hyperplane (green) using Gradient method

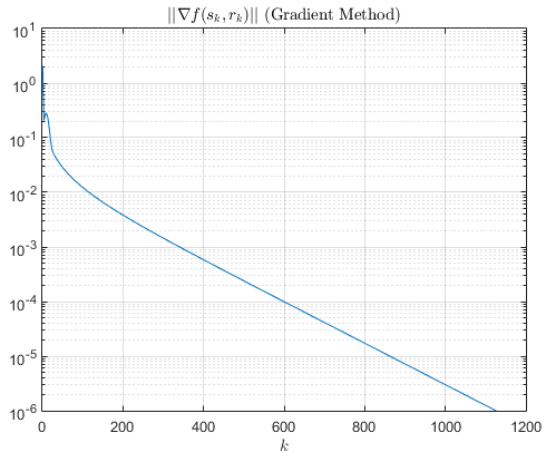

Figure 12: `data1.mat` $||\nabla f(s_k, r_k)||$ evolution

2.3 Task 3

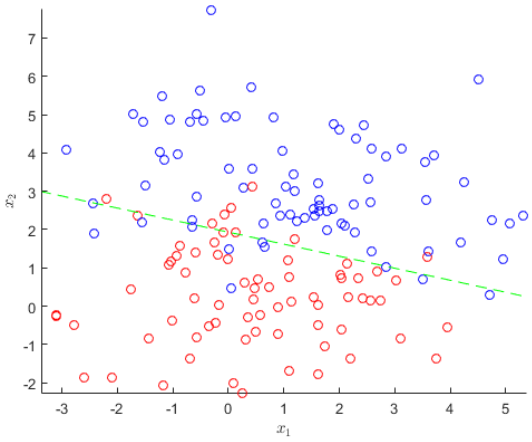

Figure 13: `data2.mat` (blue,red) with optimized hyperplane (green) using Gradient method

	s_1	s_2	r
data1.mat	1.3495	1.0540	4.8815
data2.mat	0.7402	2.3577	4.5553

Table 6: Gradient method values of s and r for tasks 2 and 3.

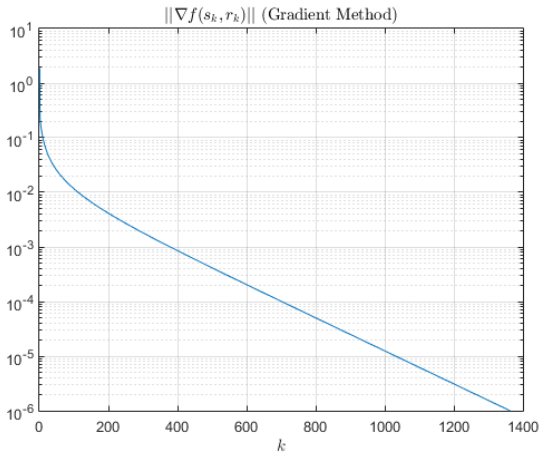

Figure 14: `data2.mat` $||\nabla f(s_k, r_k)||$ evolution

2.4 Task 4

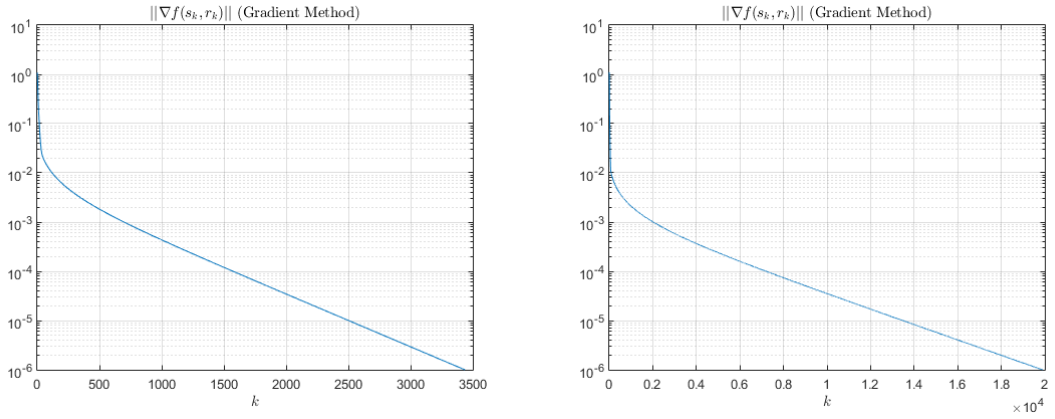

Figure 15: `data3.mat` (left) and `data4.mat` (right) $||\nabla f(s_k, r_k)||$ evolution

2.5 Task 5

Given $p(x) = \sum_{k=1}^{K} \phi(a_k^T x)$ where $\phi : R \to R$ is a twice differentiable function.
To compute the gradient of the function, $\nabla p(x)$, chain rule of differentiation can be used :

$$\nabla p(x) = \sum_{k=1}^{K} \frac{\partial \phi(a_k^T x)}{\partial a_k^T x} \cdot \frac{\partial a_k^T x}{\partial x} \tag{14}$$

$$\sum_{k=1}^{K} \frac{\partial \phi(a_k^T x)}{\partial a_k^T x} \cdot \frac{\partial a_k^T x}{\partial x} = \sum_{k=1}^{K} \dot{\phi}(a_k^T x) \cdot a_k^T = \sum_{k=1}^{K} a_k \dot{\phi}(a_k^T x) = [a_1 a_2 \dots a_k] \begin{bmatrix} \dot{\phi}(a_1^T x) \\ \dot{\phi}(a_2^T x) \\ \vdots \\ \dot{\phi}(a_k^T x) \end{bmatrix} \tag{15}$$

We can then simplify to:

$$Av = [a_1 a_2 \dots a_k] \begin{bmatrix} \dot{\phi}(a_1^T x) \\ \dot{\phi}(a_2^T x) \\ \vdots \\ \dot{\phi}(a_k^T x) \end{bmatrix} = \nabla p(x) \tag{16}$$

With dimensions 3×1, since A is $3 \times K$ and v is $K \times 1$
To prove the structure of $\nabla^2 p(x)$ we are going to refer once more to the chain rule of derivation, keeping in mind some concepts used in former passages:

$$\nabla^2 p(x) = \frac{\partial Av}{\partial v} \cdot \frac{\partial v}{\partial x} = A \cdot AD \tag{17}$$

The Hessian matrix has to have 3×3 dimensions. It just so happens that the term on the right of the dot product in the result has the same dimensions as the gradient, which will invert, since we have a dot product. Knowing that $\ddot{\phi}(a_k^T x)$ is a scalar the second term looks very much like a multiplication between a vector and a diagonal matrix.
Thus we can elaborate more of the structure

$$A \cdot AD = AD^T A^T = ADA^T \tag{18}$$

With D being, in this specific case:

$$\begin{bmatrix} \ddot{\phi}(a_1^T x) & & & \\ & \ddot{\phi}(a_2^T x) & & \\ & & \ddots & \\ & & & \ddot{\phi}(a_k^T x) \end{bmatrix} \tag{19}$$

2.6 Task 6

The code for the Gradient method can be observed at listing 11.

Listing 11: Matlab code for the Newton method.

```
function [ S ] = Newton_method( X,Y,s0,r0,e )
%Newton_method applies the newton method
%   Inputs : X,Y,e= dataset,labels,error tolerance
%            s0,r0= initial s, initial r
%   Outputs: S = [X_features+1]*[k] matrix, where S(:,end) is the final
%            value; [S(1:end-1,:); S(end,:)]= [values of s ; values of r]
%            troughout the method
k=1;
%Compactly defines an affine mapping, so that matricial notation can be
%employed . [s;r]'*[x;-1] = s'*x - r
S(:,1)=[s0;r0];
X(end+1,:)=-1;
```

```
%define functions f, and its first and second derivatives in order of s'*x
f=@(x,s,y) (1/length(x))*sum(log(1+ exp(s'*x)) - y.*(s'*x));
d_f=@(x,s,y)(exp(s'*x)./(1+exp(s'*x))) - y ;
h=@(x,s) exp(s'*x)./((1+exp(s'*x)).^2) ;

while(1)
    %calculate gradient
    g= X*(d_f(X,S(:,k),Y))'/length(X);
    grad_norm(k)=norm(g);
    if(norm(g) < e)
        break;
    else
        %calculate hessian
        hess= X*diag(h(X,S(:,k)))*X'/length(X);
        d= -(hess)^-1*g;
        %backtracking routine
        a(k)=1;
        while(1)
            cond2= f(X,S(:,k),Y)+(10^-4)*g'*(a(k)*d);
            cond1= f(X,S(:,k)+a(k)*d,Y);
            if(cond1<cond2)
                break;
            end
            %step size recalculation
            a(k)=a(k)*0.5;
        end
        % calculation of k(th) iteration of the values of s and r
        S(:,k+1)=S(:,k)+a(k)*d;
        k=k+1;
    end
end

%Norm graph
figure
semilogy(1:k,grad_norm);
xlabel('$k$','interpreter','latex')
xlim([1 k]);
title('$||\nabla{f(s_k,r_k)}||$ (Newton method)','interpreter','latex')
grid on;
%stepsize graph
figure
stem(a,'filled');
xlabel('$k$','interpreter','latex')
title('$ \alpha_k$ (Newton method)','interpreter','latex')
xlim([1 k-1]);
ylim([0 max(a)]);
end
```

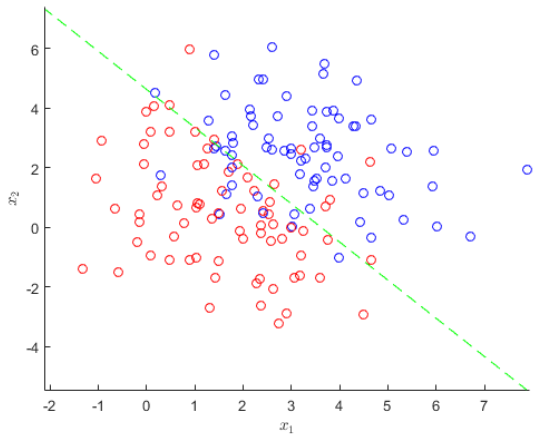

Figure 16: `data1.mat` (blue,red) with optimized hyperplane (green) using Newton method

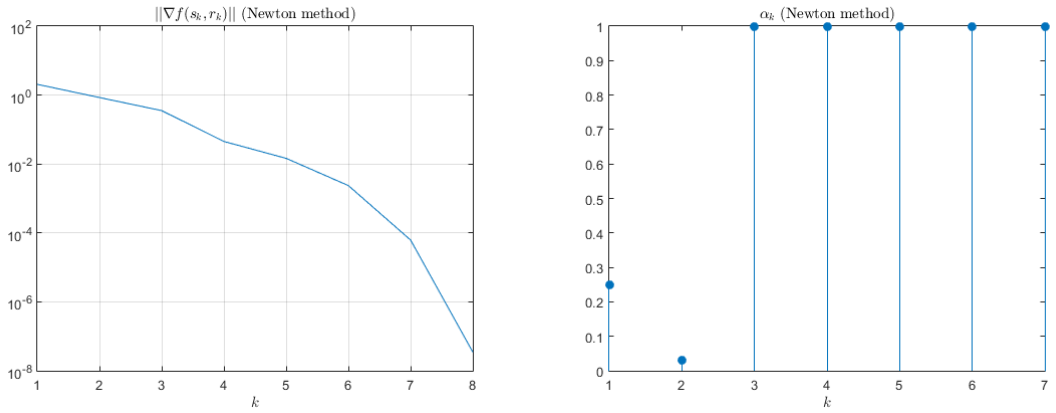

Figure 17: `data1.mat` $||\nabla f(s_k, r_k)||$ (left) and step size (right) evolution

	s_1	s_2	r
data1.mat	1.3496	1.0540	4.8817
data2.mat	0.7402	2.3577	4.5554

Table 7: Newton method values of s and r for data1.mat and data2.mat.

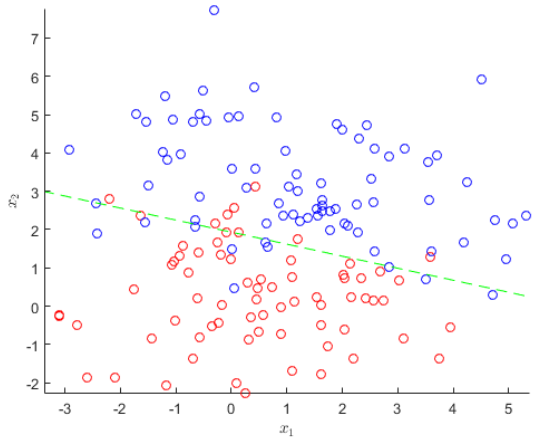

Figure 18: data2.mat (blue,red) with optimized hyperplane (green) using Newton method

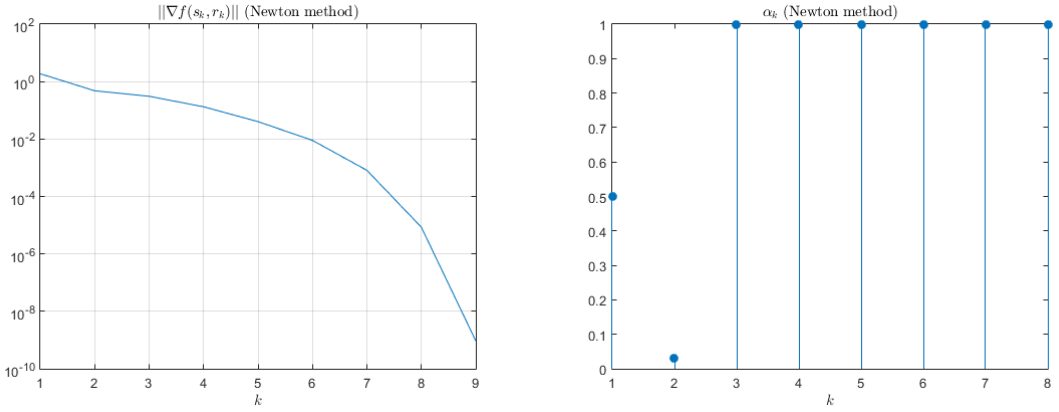

Figure 19: data2.mat $||\nabla f(s_k, r_k)||$ (left) and step size (right) evolution

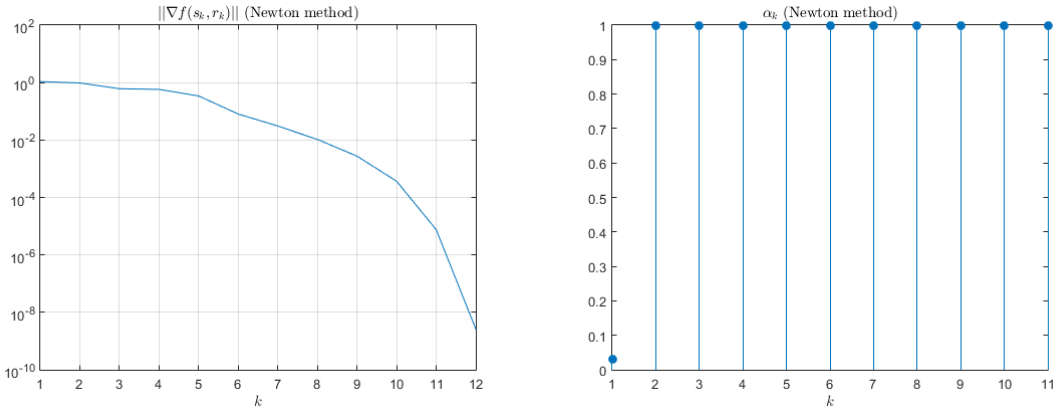

Figure 20: data3.mat gradient norm (left) and step size (right) evolution

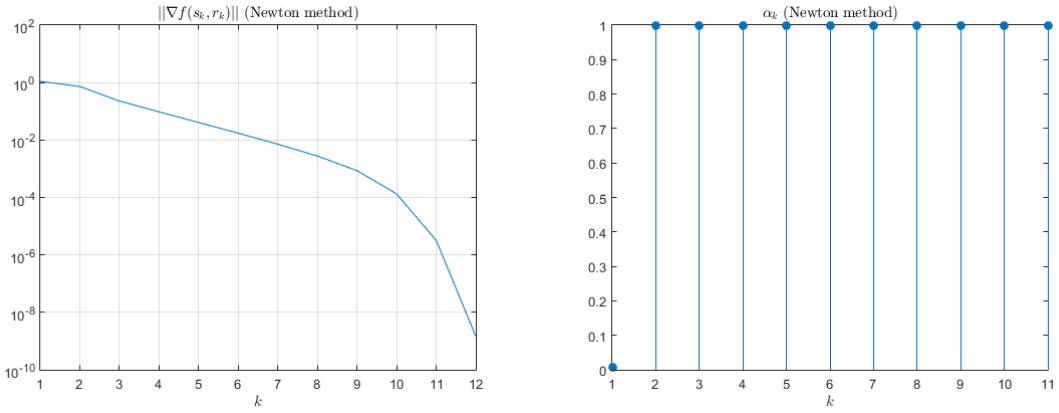

Figure 21: data4.mat $||\nabla f(s_k, r_k)||$ (left) and step size (right) evolution

2.7 Task 7

As can be consistently seen throughout the results, the Newton method converges in quite less iterations than the gradient method. This happens because in each iteration of the Newton method, the direction of the step and stepsize are calculated with a second order approximation of the function to minimize, which in its turn allows for a faster convergence on the long run.It was also a faster method time-wise, mainly due to the data sets tested.
We do believe that the time-wise results would favor the gradient method in datasets with very large number of features of X, mainly due to the complexity of the Gradient and Newton methods ($\mathcal{O}(n)$ and $\mathcal{O}(n^3)$, respectively).
The Gradient method also has an advantage in terms of computational cost (space used) in each iteration, so it has a lot better performance when its used in situations like deep learning, where milions of parameters exist. That is why it is a better alternative to the

Newton method, in these specific cases. If you have low to reasonable number of features and enough memory space, it's favourable to go for the Newton method.

2.8 Task 8

The code for the 8th task is in Listing 12 and the results obtained after running it for the `lmdata1.m` dataset are in Figs. 22 and 23.

Listing 12: Matlab code for task 8.

```
function [result, f_est] = lm(dataset, xinit, print)
    load(dataset, 'A', 'iA', 'y', 'z', 'iS', 'S');

    clc;
    close all
    xk = xinit;

    % because vecnorm(x, dim) isn't available before R2017a
    vnorm = @(mat, dim) sqrt(sum(mat.^2, dim));

    % represents sp - sq
    sel = [1 0 -1 0; 0 1 0 -1];

    norms_f = [];
    lambda = 1;
    epsilon = 10^(-6);
    while 1
        [am, sp1, sp2, sq] = variables(xk, iA, A, iS);

        [f, fa, fs] = fx(am, sp1, sp2, sq, y, z);

        grad_fa = zeros(size(am, 1), length(xk));
        for ii = 1:length(iA)
            grad_fa(ii, [2*iA(ii,2)-1 2*iA(ii,2)]) = ...
            (sp1(ii,:) - am(ii,:)) ./ vnorm(am(ii,:) - sp1(ii,:), 2);
        end

        grad_fs = zeros(size(sq, 1), length(xk));
        for ii = 1:length(iS)
            grad_fs(ii, [2*iS(ii,1)-1 2*iS(ii,1) ...
                2*iS(ii,2)-1 2*iS(ii,2)]) = ((sel'*sel*[sp2(ii,:) ...
                sq(ii,:)]') ./ vnorm(sp2(ii,:) - sq(ii,:), 2))';
        end

        g_k = [2.*grad_fa.*fa; 2.*grad_fs.*fs];
        norm_gk = norm(sum(g_k, 1));

        norms_f = [norms_f norm_gk];
        if norm_gk < epsilon
           break;
```

```
        end

        x_est = [grad_fa; grad_fs; sqrt(lambda).*eye(size(g_k,2))]...
            \ [grad_fa*xk - fa; grad_fs*xk - fs; sqrt(lambda) * xk];

        [am, sp1, sp2, sq] = variables(x_est, iA, A, iS);
        [f_est, ~, ~] = fx(am, sp1, sp2, sq, y, z);

        f_sum = sum(f);
        f_est = sum(f_est);

        if f_est < f_sum
            xk = x_est;
            lambda = 0.7 * lambda;
        else
            lambda = 2 * lambda;
        end
    end

    result = reshape(xk, [2, length(xk)/2]);

    % === plot results ===
    % -- 1) network localization setup
    if print
        figure(1)
        scatter(A(1,:), A(2,:), 80, 'r', 's', 'LineWidth', 2)
        grid on;
        hold on;
        % real locations of sensors (.)
        if exist('S', 'var')
            scatter(S(1,:), S(2,:), 50, 'b', 'o', 'filled');

            % pink lines
            for ii = 1:length(iA)
                plot([A(1,iA(ii,1)) S(1,iA(ii,2))], ...
                [A(2,iA(ii,1)) S(2,iA(ii,2))], '--m');
            end
            for ii = 1:length(iS)
                plot(S(1,iS(ii,:)), S(2,iS(ii,:)), '--m');
            end
        end
        % xinit (*)
        scatter(xinit(1:2:length(xinit)-1), ...
        xinit(2:2:length(xinit)), 50, 'b', '*', 'LineWidth', 1.5);

        % solution obtained (o)
        scatter(result(1,:), result(2,:), 80, 'b', 'o', 'LineWidth', 2);
        xlim([min(A(1,:), [], 2)-3 max(A(1,:), [], 2)+3])
        title(['Network localization']);

        % -- 2) norm of the gradient of the cost function
```

```
92          figure(2);
93          semilogy(1:length(norms_f), norms_f, 'LineWidth', 2);
94          xlim([0 length(norms_f)+2])
95
96          grid on;
97          grid minor;
98
99          title('$||\nabla f(x_k)||$ (LM Method)','Interpreter','latex')
100         xlabel('$k$', 'Interpreter', 'latex');
101     end
102 end
103
104 function [am, sp1, sp2, sq] = variables(xk, iA, A, iS)
105     am = A(:,iA(:,1))';
106     sp1 = [xk(iA(:,2)*2 - 1,:) xk(iA(:,2)*2,:)];
107
108     sp2 = [xk(iS(:,1)*2 - 1,:) xk(iS(:,1)*2,:)];
109     sq = [xk(iS(:,2)*2 - 1,:) xk(iS(:,2)*2,:)];
110 end
111
112 function [f, fa, fs] = fx(am, sp1, sp2, sq, y, z)
113     vnorm = @(mat, dim) sqrt(sum(mat.^2, dim));
114
115     fa = vnorm(am - sp1,2) - y;
116     fs = vnorm(sp2 - sq,2) - z;
117
118     f = [fa.^2;fs.^2];
119 end
```

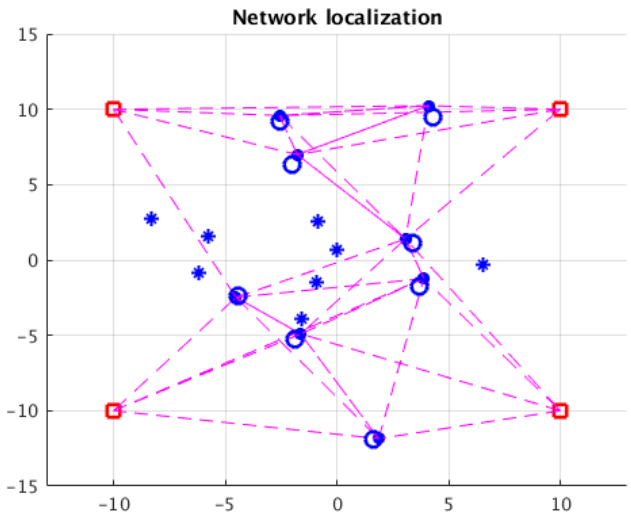

Figure 22: Network localization setup with the red squares being the anchors, the blue stars representing the initial guesses for the sensors, the blue dots being the real location of the sensors and the blue circles being the final guess.

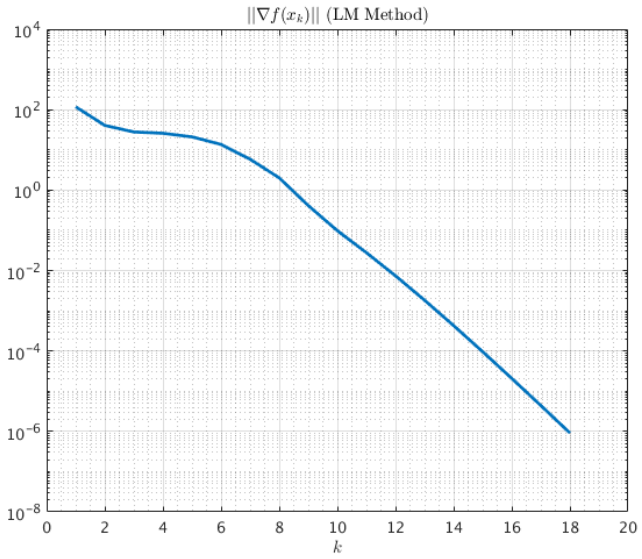

Figure 23: Evolution of $||\nabla f(x_k)||$ through the iterations of the LM method for the first dataset.

2.9 Task 9

The code used for generating the random `xinit` values and using them in the LM method is in Listing 13.

Listing 13: Matlab code for task 9.

```
function [result, f, best_xinit] = no_xinit(dataset)
    load(dataset, 'A');

    % number of xinits to be generated
    n = 10000;

    results = [];
    f_list = [];
    xinits = [];

    for ii = 1:n
        l = min(A(:))+min(A(:))/2;
        u = max(A(:))+max(A(:))/2;
        xinit = (l-u).*rand(16,1) + u;

        [result, f] = lm(dataset, xinit, 0);

        results = [results; result];
        f_list = [f_list f];
        xinits = [xinits xinit];
    end

```

```
23      [f, idx] = min(f_list(:));
24      best_xinit = xinits(:,idx);
25      result = results([2*idx-1 2*idx],:);
26  end
```

We tested 10000 random values for the `xinit`, and ultimately the best result which is the of 4.4945 was found with the `xinit` vector that can be found in Table 8.

Table 8: `xinit` that lead to the best results.

x_1	10.6518	x_9	-8.2749
x_2	-3.8599	x_{10}	-14.7822
x_3	1.7606	x_{11}	-4.4136
x_4	-8.6097	x_{12}	5.5214
x_5	-0.4788	x_{13}	2.2146
x_6	4.6891	x_{14}	-5.3173
x_7	7.6006	x_{15}	-10.6129
x_8	0.3717	x_{16}	-11.1809

Table 9: Results for the best xinit

Sensor	x	y
s_1	-5.9916	9.1769
s_2	-8.8767	-1.9330
s_3	4.2174	0.4480
s_4	-7.6749	3.9664
s_5	0.7416	-4.0953
s_6	2.0511	3.6757
s_7	10.9249	-0.8923
s_8	-1.3723	-2.6921

In general and with the 10^4 `xinit` vectors we generated, the cost function seemed to converge towards two values: 4.4945 and 107.2565 which proves that there is more than one local minimum.

3 Part 3

3.1 Task 1

A constrained optimization problem can be written in a general form as seen in (20).

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & h_1(x) = 0 \\ & \vdots \\ & h_p(x) = 0 \\ & g_1(x) \leq 0 \\ & \vdots \\ & g_m(x) \leq 0 \end{aligned} \tag{20}$$

The first thing to do is to reformulate the problem to match the formulation given in (20). Then, we can see that $f(x) = ||p - x||_2$. Since the point has to belong to the disk, then $g(x) = ||x - c||_2 - r$.

With these expressions we can then use the Karush-Kuhn-Tucker (KKT) theorem, which declares that for an optimization problem that is formulated as shown in (20), with f, h and g being continuously differentiable, if there is an x^* that is a regular local minimum then there exists $\lambda^* \in \mathbf{R}^p$ and $\mu^* \in \mathbf{R}^m$ that fulfill the conditions in (21).

$$\begin{cases} \nabla f(x^*) + \nabla h(x^*)\lambda^* + \nabla g(x^*)\mu^* = 0 & \text{(stationarity)} \\ h(x^*) = 0, g(x^*) \leq 0 & \text{(primal feasibility)} \\ \mu^* \geq 0 & \text{(dual feasibility)} \\ g(x^*)^T \mu^* = 0 & \text{(complementary slackness)} \end{cases} \tag{21}$$

There are no equality conditions, so the conditions are slightly simplified. Even then, there are two cases to consider, depending on whether $g(x) = 0$ or $g(x) < 0$.

Considering $g(x) < 0$ first, the principle of primal feasibility states that $g(x) \leq 0$, then in order for the complementary slackness condition to hold then it stands that $\mu = 0$. However, that leads to the equation in (22), which leads to an impossible result.

$$\nabla f(x) + \cancel{\nabla h(x)\lambda} + \cancel{\nabla g(x)\mu} = 0 \Leftrightarrow \frac{x - p}{||p - x||} = 0 \tag{22}$$

Therefore all that's left to do is to consider $g(x) = 0$. In order to simplify the calculations, we decided to square the norm. This is a rough approximation, but it's a valid one because x represents the closest point from a point to the disk - even if the distance is squared, the closest point is the same regardless if the distance is squared or not.

If we were using the original expressions, then $\nabla f(x) = \frac{x-p}{||p-x||_2}$ and $\nabla g(x) = \frac{c-x}{||x-c||_2}$. Through the chain rule, we get the equivalent expressions with the norm squared, $\nabla(f(x)^2) = 2(x-p)$ and $\nabla(g(x)^2) = 2(c-x)$. We can then substitute those expressions on the stationarity condition and solve that for x as seen in equation (23).

$$\nabla(f(x)^2) + \nabla(g(x)^2)\mu = 0 \Leftrightarrow x - p + (c - x)\mu = 0 \Leftrightarrow x = \frac{p - \mu c}{1 - \mu} \tag{23}$$

Now we need to find the value for μ. For that we're going to input the result found in (23) in the primal feasibility condition.

$$\begin{aligned} g(x) = 0 \Leftrightarrow ||x - c||_2 - r = 0 \Leftrightarrow ||\frac{p - \mu c}{1 + \mu} - c||_2 = r \Leftrightarrow \frac{||p - c||_2}{|1 - \mu|} = r \\ \Leftrightarrow \mu = 1 \pm \frac{||p - c||_2}{r} \end{aligned} \tag{24}$$

Then we substitute (24) back in (23), which leads to two different values for x. One of them is going to be within the boundaries of the disk, the other is going to be outside them.

$$x = \frac{p - (1 \pm \frac{||p-c||_2}{r}) \cdot c}{1 - 1 \pm \frac{||p-c||_2}{r}} = c \pm \frac{r \cdot (p - c)}{||p - c||_2} \tag{25}$$

In this case, the correct answer is the one where $(p - c)/||p - c||_2)$ is positive. This is because x represents the projection of p in the disk - the larger the x, the smaller $||p - x||_2$ will be in the end. This solution is only valid in the stated conditions, which means that $g(x) \leq 0 \Leftrightarrow ||x - c||_2 - r$ has to hold. In

We still have to take into account that since $g(x) = 0$ leads to an impossible solutions we have to make sure that that doesn't happen - which means that x and p have to be different.

The closed-form solution for the stated problem can be seen in (26).

$$x = c + \frac{r \cdot (p - c)}{||p - c||_2} \wedge x \neq p \wedge ||x - c||_2 < 0 \tag{26}$$

3.2 Task 2

Considering the problem presented in (27), the goal becomes to find a closed-form solution.

$$\begin{aligned} \underset{x,u}{\text{minimize}} \quad & \sum_{t=1}^{T-1} \|x(t) - x_{des}(t)\|_2^2 + \lambda \sum_{t=0}^{T-1} \|u(t)\|_2^2 \\ \text{subject to} \quad & x(0) = x_{\text{initial}} \\ & x(T) = x_{\text{final}} \\ & x(t+1) = Ax(t) + Bu(t), \text{ for } 0 \leq t \leq T - 1 \end{aligned} \tag{27}$$

The vectors x and x_{des} are considered to be $[n \times 1]$ and the vector u to be $[m \times 1]$ with $m < n$.

Now, given the presented problem, in order to solve it using the KKT theorem, the problem has to be rewritten to fit the form presented in (20).

In order to do so, the chosen problem variable will be the variable y, defined as presented in (28).

$$y(t) = \left[\begin{array}{c} x(0) \\ x(1) \\ \vdots \\ x(T-1) \\ x(T) \\ \hdashline u(0) \\ \vdots \\ u(T-1) \end{array}\right] \tag{28}$$

Likewise, the function $f(x(t), u(t))$ to be minimized will be as in (29).

$$f(x(t), y(t)) = \sum_{t=1}^{T-1} ||x(t) - x_{\text{des}}(t)||_2^2 + \lambda \sum_{t=0}^{T-1} ||u(t)||_2^2 \tag{29}$$

Therefore, since the function f is a sum of squared norms, it will be a convex function and therefore, the problem is a convex problem.

In addition, considering that the only constraints specified are equality constraints that can be rewritten as affine functions, as in (33), it becomes possible to say that no other regularity conditions need to be verified in order for the solution to satisfy the KKT conditions.

And so, the final solution will be the y^* that satisfies the conditions presented in (30), for a certain constant λ.

$$\begin{cases} \nabla f(y^*) + \lambda \nabla h(y^*) = 0 \\ h(y^*) = 0 \end{cases} \tag{30}$$

Now, to write the equality constraints in their affine form and in relation to the problem variable y, it becomes necessary to define the matrix c^T, as in (31) and the matrix d, as in (32).

$$c^T = \left[\begin{array}{ccccc:ccc} -I_{n\times n} & 0_{n\times n} & ... & ... & ... & ... & 0_{n\times m} & ... \\ \hdashline A_{n\times n} & -I_{n\times n} & 0_{n\times n} & ... & 0_{n\times n} & B_{n\times m} & ... & 0_{n\times m} \\ & & \vdots & & & & \vdots & \\ 0_{n\times n} & 0_{n\times n} & ... & A_{n\times n} & -I_{n\times n} & 0_{n\times m} & ... & B_{n\times m} \\ \hdashline 0_{n\times n} & ... & ... & 0_{n\times n} & -I_{n\times n} & ... & 0_{n\times m} & ... \end{array}\right] \tag{31}$$

$$d = \left[\begin{array}{c} x_{\text{initial}_{n\times 1}} \\ 0_{n\times 1} \\ \vdots \\ 0_{n\times 1} \\ x_{\text{final}_{n\times 1}} \end{array}\right] \tag{32}$$

And so, the equality constraints have been compiled and represented as in (33), resulting in a column vector.

$$c^T y + d = 0_{(T+1)n \times 1} \tag{33}$$

Therefore, the gradient $\nabla h(y(t))$ will simply be as in (34).

$$\nabla h(y(t)) = \nabla[c^T y + d] = (c^T)^T = c \tag{34}$$

As in (30), it is also necessary to compute the gradient $\nabla f(y(t))$ to determine a solution y^*, which implies computing the gradients with respect to all of the components of the vector $y(t)$, which allows for the simplification of the sum since all of the remaining partial derivatives will be zero. And so, since the variables $x(t)$ and $u(t)$ are independent of each other it's possible to compute their gradients separately, as presented in (35).

$$\begin{cases} \nabla f_{x(t)} = \nabla ||x(t) - x_{\text{des}}(t)||_2^2 = 2(x(t) - x_{\text{des}}(t))) \\ \nabla f_{u(t)} = \nabla \lambda ||u(t)||_2^2 = 2\lambda u(t) \end{cases} \tag{35}$$

And so, the gradient $\nabla f(y(t))$ is as in (36)

$$\nabla f(y(t)) = \begin{bmatrix} 0 \\ 2x(1) - 2x_{\text{des}}(1) \\ \vdots \\ 2x(T-1) - 2x_{\text{des}}(T-1) \\ 0 \\ 2\lambda u(0) \\ \vdots \\ 2\lambda u(T-1) \end{bmatrix} = Gy + l \tag{36}$$

With the matrix G as in (37) and the matrix l as in (38).

$$G = \left[\begin{array}{c:c:c} 0 & 0_{n\times 1} & 0 \\ \hdashline 2I_{n(T-1)\times n(T-1)} & 0_{n\times 1} & 0 \\ \hdashline 0 & 0_{n\times 1} & 0 \\ \hdashline 0 & 0_{n\times 1} & 2\lambda I_{m(T-1)\times m(T)} \end{array}\right] \tag{37}$$

$$l = \left[\begin{array}{c} 0_{n\times 1} \\ -2x_{\text{des}}(1)_{n\times 1} \\ \vdots \\ -2x_{\text{des}}(T-1)_{n\times 1} \\ 0_{n\times 1} \\ \hdashline 0_{m\times 1} \\ \vdots \\ 0_{m\times 1} \end{array}\right] \tag{38}$$

Solving the original system (30) leads to (39) and ultimately to (40).

$$\begin{cases} (Gy + l) + \lambda c = 0 \\ c^T y + d = 0 \end{cases} \Leftrightarrow \begin{cases} G^T G y = G^T(-\lambda c - l) \\ c^T y + d = 0 \end{cases} \tag{39}$$

$$\begin{cases} y = (G^T G)^{-1} G^T(-\lambda c - l) \\ c(G^T G)^{-1} G^T(-\lambda c - l) + d = 0 \end{cases} \Leftrightarrow \begin{cases} y = -(G^T G)^{-1} G^T [c^T (G^T G)^{-1} G^T]^{-1}(-d) \\ (-\lambda c - l) = [c^T (G^T G)^{-1} G^T]^{-1}(-d) \end{cases} \tag{40}$$

And so, the final closed form solution is the one presented in (41).

$$y = -(G^T G)^{-1} G^T [c^T (G^T G)^{-1} G^T]^{-1}(-d) \tag{41}$$