TÉCNICO
LISBOA

Optimization and Algorithms Project Report

Raj Maharjan (ist1115593)
Ahmad Ashraf Zargar (ist1115594)
Mohammad Alsalman (ist1115608)
Urho Santeri Kokkonen (ist1115570)

October 2025

1 Task 1

In this task, we solved problem (3) for twelve different values of ρ. The plots below illustrate the resulting trajectories and the trade-off between tracking error and control effort.

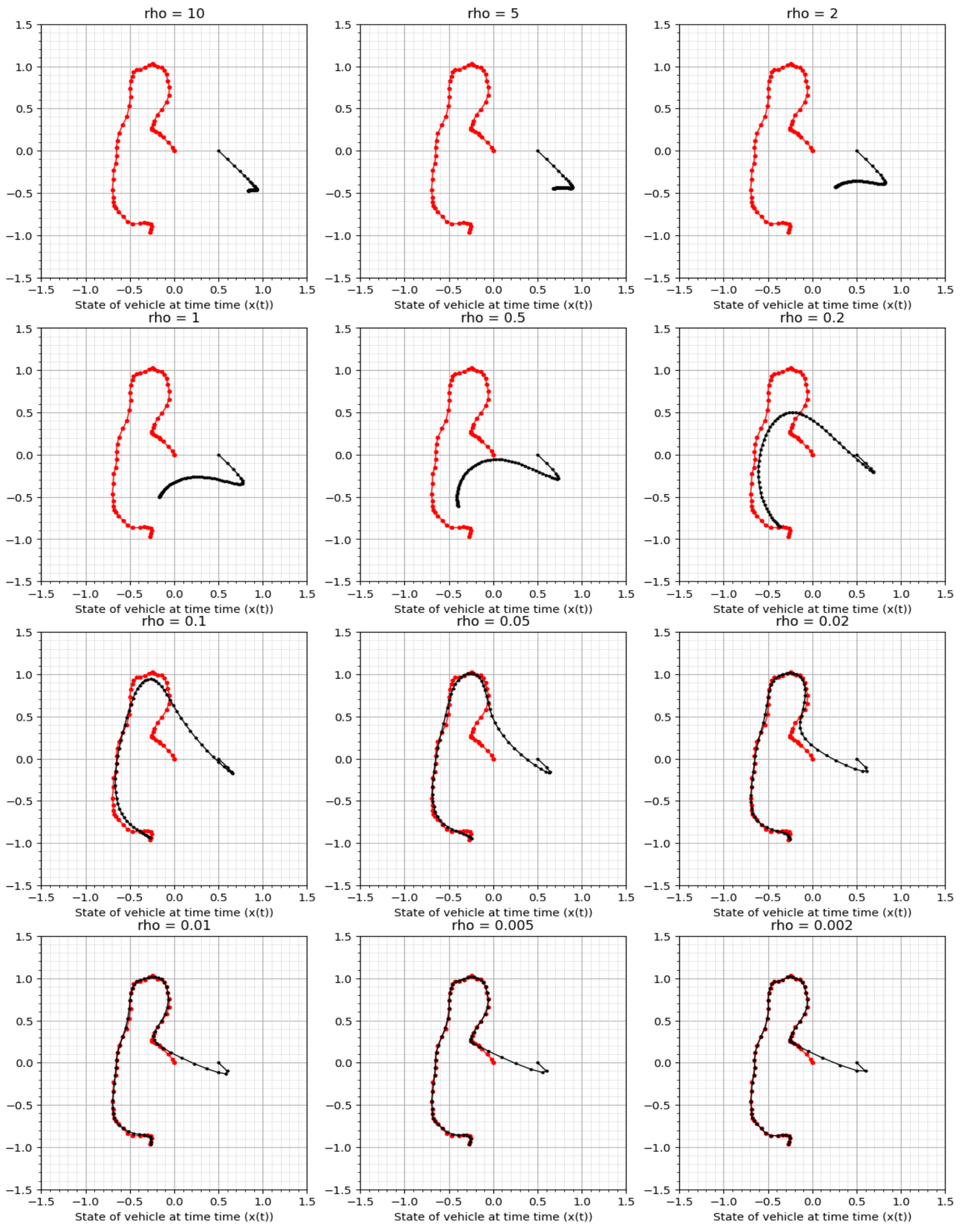


Figure 1: Vehicle trajectory versus target trajectory for different values of ρ. The red path shows the target trajectory, and the black path shows the vehicle trajectory. As ρ decreases, the vehicle trajectory aligns more closely with the target.

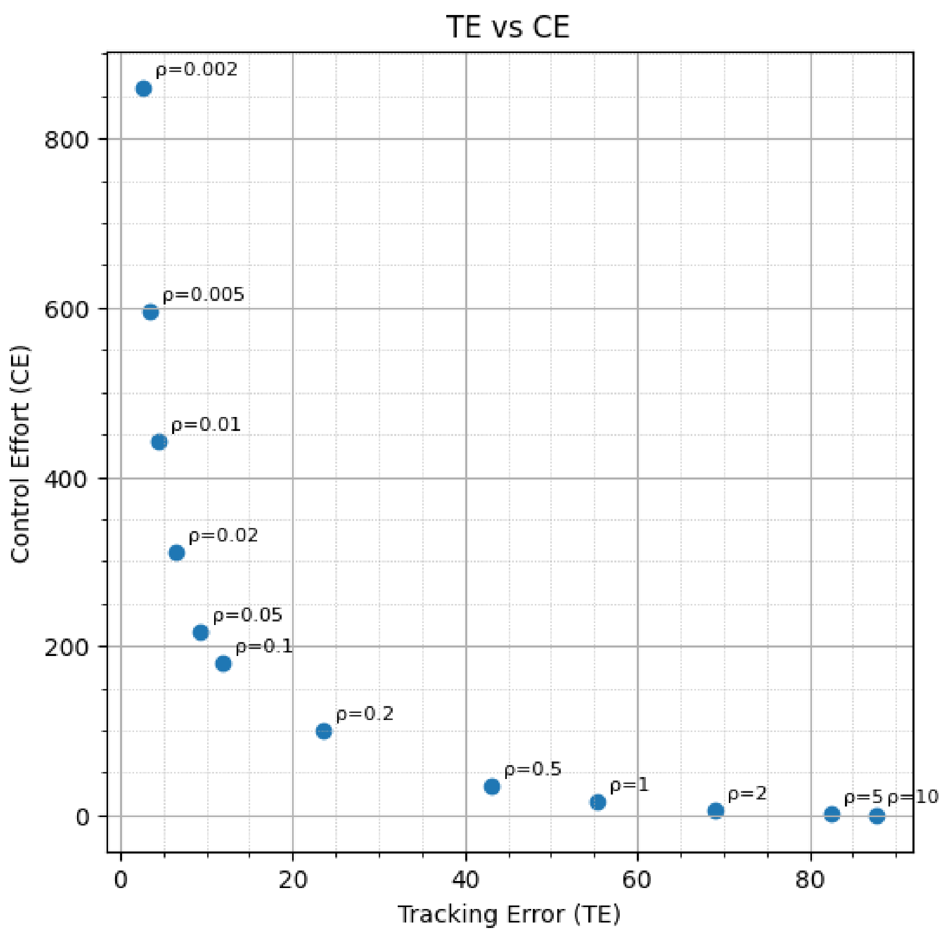


Figure 2: Trade-off between tracking error (TE) and control effort (CE) as ρ varies. Higher ρ reduces control effort but increases tracking error, while lower ρ improves tracking accuracy at the cost of higher control effort.

The results clearly demonstrate the trade-off between tracking accuracy and control effort governed by the weight parameter ρ. For higher values of ρ, the optimization prioritizes minimizing the control effort $u(t)$. As a result, the applied forces are smaller and smoother, but the vehicle deviates more from the target trajectory, leading to a higher tracking error. Conversely, when ρ decreases, the optimization places more emphasis on minimizing the tracking error. The vehicle follows the target path more closely, but this comes at the cost of larger and more aggressive control inputs, which increase the control effort. This aligns with theoretical expectations for the cost function structure. This behavior is evident from the trajectory plots:

- In Figure 1, at large ρ (e.g., 10, 5, 2), the vehicle's trajectory significantly lags behind the target.
- As ρ decreases, the two trajectories converge, showing improved tracking.
- The trade-off curve Figure 2 confirms this inverse relationship. TE increases with ρ, while CE decreases.

2 Task 2

This task is asking us to prove the following, if we optimize with weights ρ_a and ρ_b, and with ρ_a get a better tracking error TE than with ρ_b, then you must have used more control effort CE with ρ_a. Intuitively, this makes sense as achieving a trajectory closer to the target will always require more or equal amount of effort than for a less accurate trajectory.

Let us evaluate the problem at ρ_b:

$$\mathrm{TE}^*_{\rho_b} + \rho_b \cdot \mathrm{CE}^*_{\rho_b} \;\leq\; \mathrm{TE}^*_{\rho_a} + \rho_b \cdot \mathrm{CE}^*_{\rho_a}.$$

The above inequality is true because the optimal cost for ρ_b is better than the cost evaluated at the optimal solution for ρ_a (which is a feasible solution but not necessarily optimal for ρ_b)

Rearrange:

$$\rho_b\big(\mathrm{CE}^\star_{\rho_b} - \mathrm{CE}^\star_{\rho_a}\big) \;\leq\; \mathrm{TE}^\star_{\rho_a} - \mathrm{TE}^\star_{\rho_b}.$$

Use the assumption $\mathrm{TE}^\star_{\rho_a} \leq \mathrm{TE}^\star_{\rho_b}$ so $\mathrm{TE}^\star_{\rho_a} - \mathrm{TE}^\star_{\rho_b} \leq 0$ and this results in:

$$\rho_b\big(\mathrm{CE}^\star_{\rho_b} - \mathrm{CE}^\star_{\rho_a}\big) \;\leq\; 0$$

$$\mathrm{CE}^\star_{\rho_b} - \mathrm{CE}^\star_{\rho_a} \;\leq\; 0$$

$$\mathrm{CE}^\star_{\rho_b} \;\leq\; \mathrm{CE}^\star_{\rho_a}$$

3 Task 3

We have to show that the optimization problem (1) has a unique solution.

$$\begin{aligned} \min_{x,\,u} \quad & \underbrace{\sum_{t=1}^{T} \|Ex(t) - q(t)\|_2}_{\text{TE}} + \rho \underbrace{\sum_{t=1}^{T-1} \|u(t)\|_2^2}_{\text{CE}} \\ \text{subject to} \quad & x(1) = x_{\text{initial}}, \\ & x(t+1) = Ax(t) + Bu(t), \quad \text{for } 1 \le t \le T-1. \end{aligned} \tag{1}$$

Proof. For a unique solution, the function should be strongly convex. Hence we require to prove that the problem given in equation 1 is strongly convex. We begin by rewriting the problem as an unconstrained problem by eliminating the state variable $x(t)$ and expressing the cost only in terms of u.

Expanding the constraints,

$$\begin{aligned} x(2) &= Ax(1) + Bu(1), \\ x(3) &= Ax(2) + Bu(2) = A(Ax(1) + Bu(1)) + Bu(2) = A^2x(1) + ABu(1) + Bu(2), \\ x(4) &= Ax(3) + Bu(3) = A^3x(1) + A^2Bu(1) + ABu(2) + Bu(3), \\ &\vdots \\ x(t) &= A^{t-1}x(1) + \sum_{k=1}^{t-1} A^{t-1-k}B\,u(k). \end{aligned}$$

Using the generalization, we get the following unconstrained optimization problem.

$$\min_{u(1),\dots,u(T-1)} \underbrace{\sum_{t=1}^{T} \left\| E\left(A^{t-1}x_{\text{initial}} + \sum_{k=1}^{t-1} A^{t-1-k}B\,u(k) \right) - q(t) \right\|_2}_{\text{TE}} + \rho \underbrace{\sum_{t=1}^{T-1} \|u(t)\|_2^2}_{\text{CE}}.$$

Now for this problem to be strongly convex we need to show that the TE part is Convex, while the CE part is Strongly Convex.

$$\min_{u(1),\dots,u(T-1)} \underbrace{\sum_{t=1}^{T} f_t(u)}_{\text{TE}} + \rho \underbrace{\sum_{t=1}^{T-1} \|u(t)\|_2^2}_{\text{CE}},$$

where each term is defined as

$$f_t(u) = \left\| E\left(A^{t-1}x_{\text{initial}} + \sum_{k=1}^{t-1} A^{t-1-k}B\,u(k) \right) - q(t) \right\|_2, \quad t = 1, \ldots, T.$$

Define an affine map $w(u)$ and a function v that takes the norm:

$$\boxed{w(u) := E\sum_{k=1}^{t-1} A^{t-1-k}B\,u(k) + EA^{t-1}x_{\text{initial}} - q(t)}$$

This is affine in u because it is linear in $u(k)$ plus a constant vector.

$$\boxed{v(w) := \|w\|_2}$$

Convexity:

- $w(u)$ is affine.
- $v(z) = \|z\|_2$ is convex.

The composition of a convex function with an affine map is convex. There for the $f_t(u)$ are convex, as they are sum of convex function, the overall tree could be seen in Figure 3

Now considering the CE part,

$$g(u) := \rho\sum_{t=1}^{T-1} \|u(t)\|_2^2,$$

we take the derivate of the function,

$$\nabla_{u(t)}g(u) = \nabla_{u(t)}\big(\rho\|u(t)\|_2^2\big) = 2\rho\,u(t)$$

$$\nabla^2_{u(t)}g(u) = 2\rho\,I_m$$

This is positive definite for $\rho > 0$, i.e. $2\rho I \succ 0$. There for $g(u)$ is strongly convex.

Hence concluding, we know that $f_t(u)$ is convex and $g(u)$ is strongly convex. Therefore, using the theorem (Module 3, slide 18) $f_t(u) + g(u)$ will be strongly convex, and we would have 1 unique global minimizer. Figure 14 in the appendix presents the handwritten analytic solution.

□

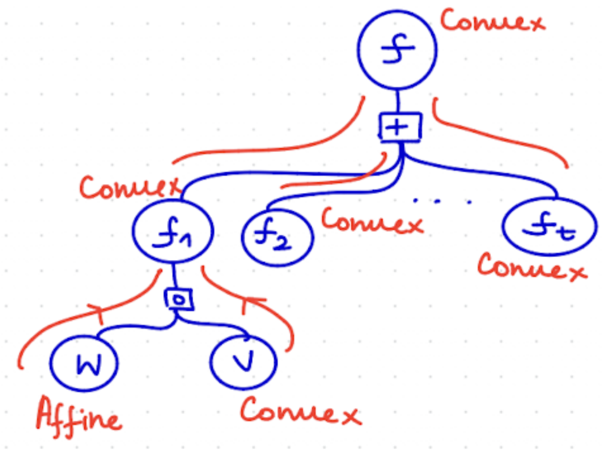


Figure 3: The overall Tree of convexity for $f_t(u)$

4 Task 4

In this task, we solve the problem as described in 2 i.e. minimizing a weighted sum of tracking errors with respect to two possible target trajectories, along with a control effort penalty. The weights p_1 and p_2 represent the probabilities associated with each target path. The plot below (Figure 4) presents the trajectory of the vehicle following the targets with probabilities p_1 and p_2.

$$\begin{aligned} \underset{x,\,u}{\text{minimize}} \quad & \underbrace{p_1 \sum_{t=1}^{T} \|Ex(t) - q_1(t)\|_2}_{\text{TE1}} + \underbrace{p_2 \sum_{t=1}^{T} \|Ex(t) - q_2(t)\|_2}_{\text{TE1}} + \underbrace{\rho \sum_{t=1}^{T-1} \|u(t)\|_2^2}_{\text{CE}} \qquad (2) \\ \text{subject to} \quad & x(1) = x_{\text{initial}}, \\ & x(t+1) = Ax(t) + Bu(t), \quad \text{for } 1 \le t \le T-1. \end{aligned}$$

The results show that for cases when either $p_1 > p_2$ or $p_1 < p_2$ the vehicle will follow the target with higher probability, with the lower probability acting as shift or bias but not playing a major role in the vehicle's direction. While for the case when $p_1 = p_2$, the vehicle follows a intermediate trajectory between the trajectory 1 and trajectory 2.

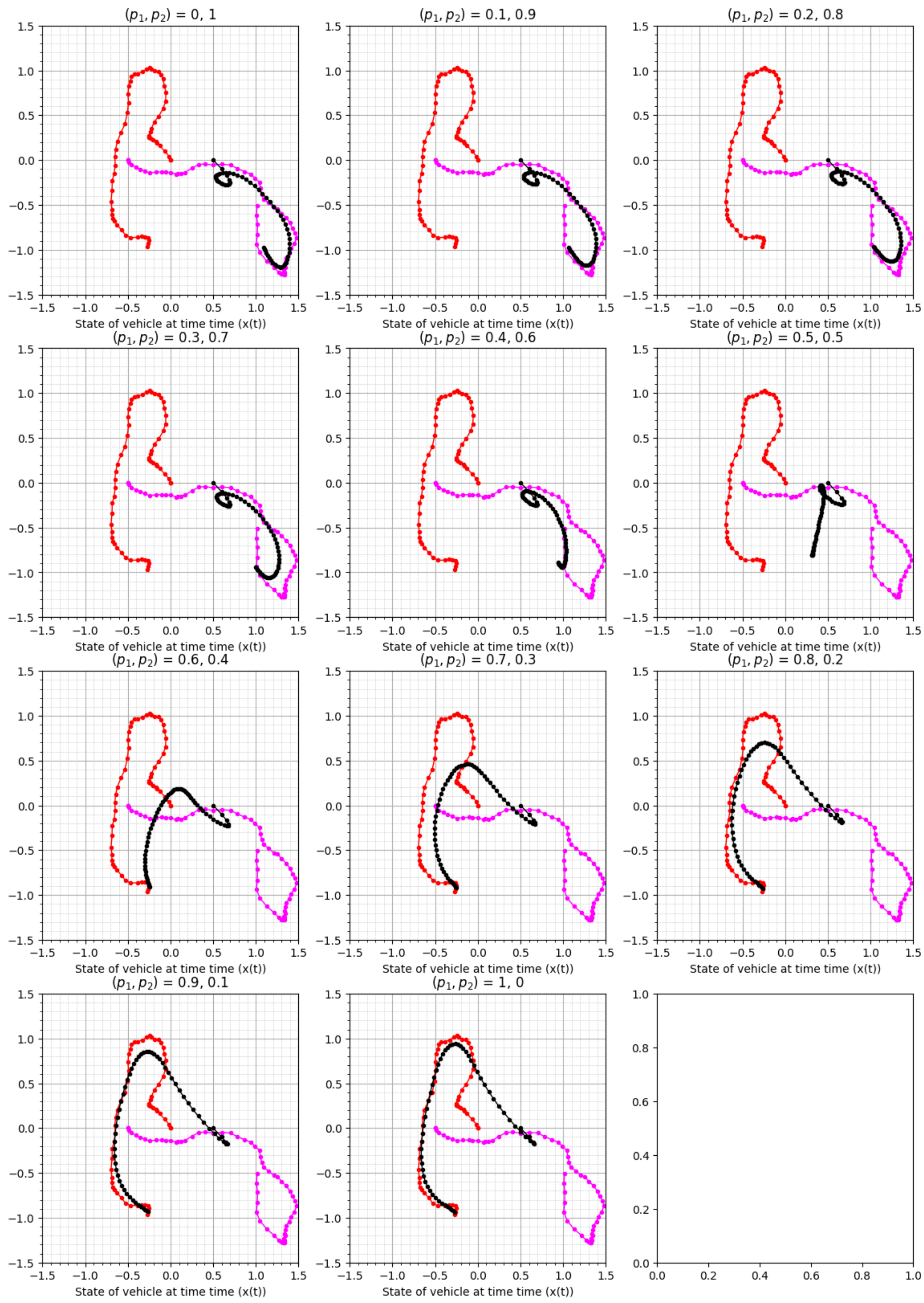


Figure 4: The two known possible trajectories that the target can execute, trajectory 1 (red) and trajectory 2 (magenta), and a trajectory of our vehicle (black) corresponding to a control signal optimized as in (2) for p_1 and p_2.

5 Task 5

The problem follows the same structure as the previous question, but with a twist: we assume that the trajectory—either 1 or 2—that the target executes from time t= 1 is revealed to us at t = 25. That is, from t = 1 to t = 24 we ignore which trajectory the target is executing; we know only their prior probabilities. But, about midway in the time horizon, at t = 25, we are told which trajectory the target has actually chosen to execute since t= 1 (and will keep executing until the end of the time horizon).

$$\begin{aligned}
\underset{x_1, u_1, x_2, u_2}{\text{minimize}} \quad & \sum_{k=1}^{K} p_k \left(\sum_{t=1}^{T} \|Ex_k(t) - q_k(t)\|_2 + \rho \sum_{t=1}^{T-1} \|u_k(t)\|_2^2 \right) \\
\text{subject to} \quad & x_1(1) = x_{\text{initial}}, \\
& x_1(t+1) = Ax_1(t) + Bu_1(t), \quad \text{for } 1 \leq t \leq T-1, \\
& x_2(1) = x_{\text{initial}}, \\
& x_2(t+1) = Ax_2(t) + Bu_2(t), \quad \text{for } 1 \leq t \leq T-1, \\
& \boxed{\text{more constraints needed here.}}
\end{aligned} \tag{3}$$

The problem remains in complete because the condition that both the vehicles should follow the same target in time interval $0 \leq t < 25$, which means that the control effort for the vehicles should be the same. If this constraint is not enforced then vehicles would follow the trajectory of target 1 and the other of target 2, i.e. $u_1(t)$ and $u_2(t)$ are independent decision variables for all t, if we solve this problem as it is, the optimizer will simply pick $u_1(t)$ to optimally follow trajectory 1 and $u_2(t)$ to optimally follow trajectory 2 from the very start ($t = 1$). Which is not the same as described by the problem statement, hence the formulation (3) is incomplete and requires more constraints.

6 Task 6

The constraint which is required in the problem formulation 3, is to enforce that $u_1(t) = u_2(t), \quad$ for $1 \leq t \leq 24$. Hence the complete formulation is presented below (4).

$$
\begin{array}{ll}
\underset{x_1, u_1, x_2, u_2}{\text{minimize}} & \displaystyle\sum_{k=1}^{K} p_k \left(\sum_{t=1}^{T} \|E x_k(t) - q_k(t)\|_2 + \rho \sum_{t=1}^{T-1} \|u_k(t)\|_2^2 \right) \\
\text{subject to} & x_1(1) = x_{\text{initial}}, \\
& x_1(t+1) = A x_1(t) + B u_1(t), \quad \text{for } 1 \leq t \leq T-1, \\
& x_2(1) = x_{\text{initial}}, \\
& x_2(t+1) = A x_2(t) + B u_2(t), \quad \text{for } 1 \leq t \leq T-1, \\
& \boxed{u_1(t) = u_2(t), \qquad\qquad\quad \text{for } 1 \leq t \leq 24}
\end{array}
\tag{4}
$$

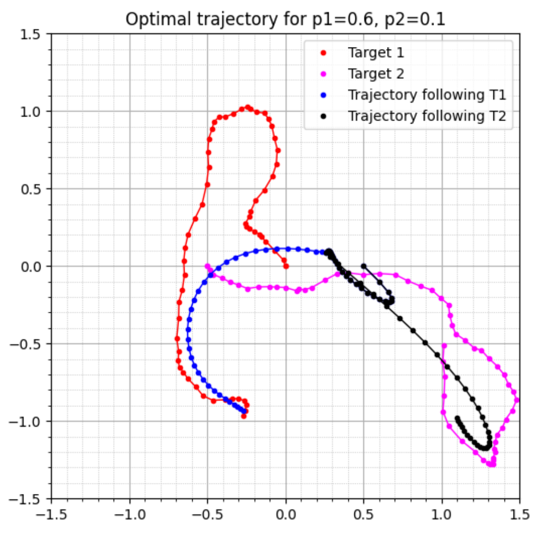


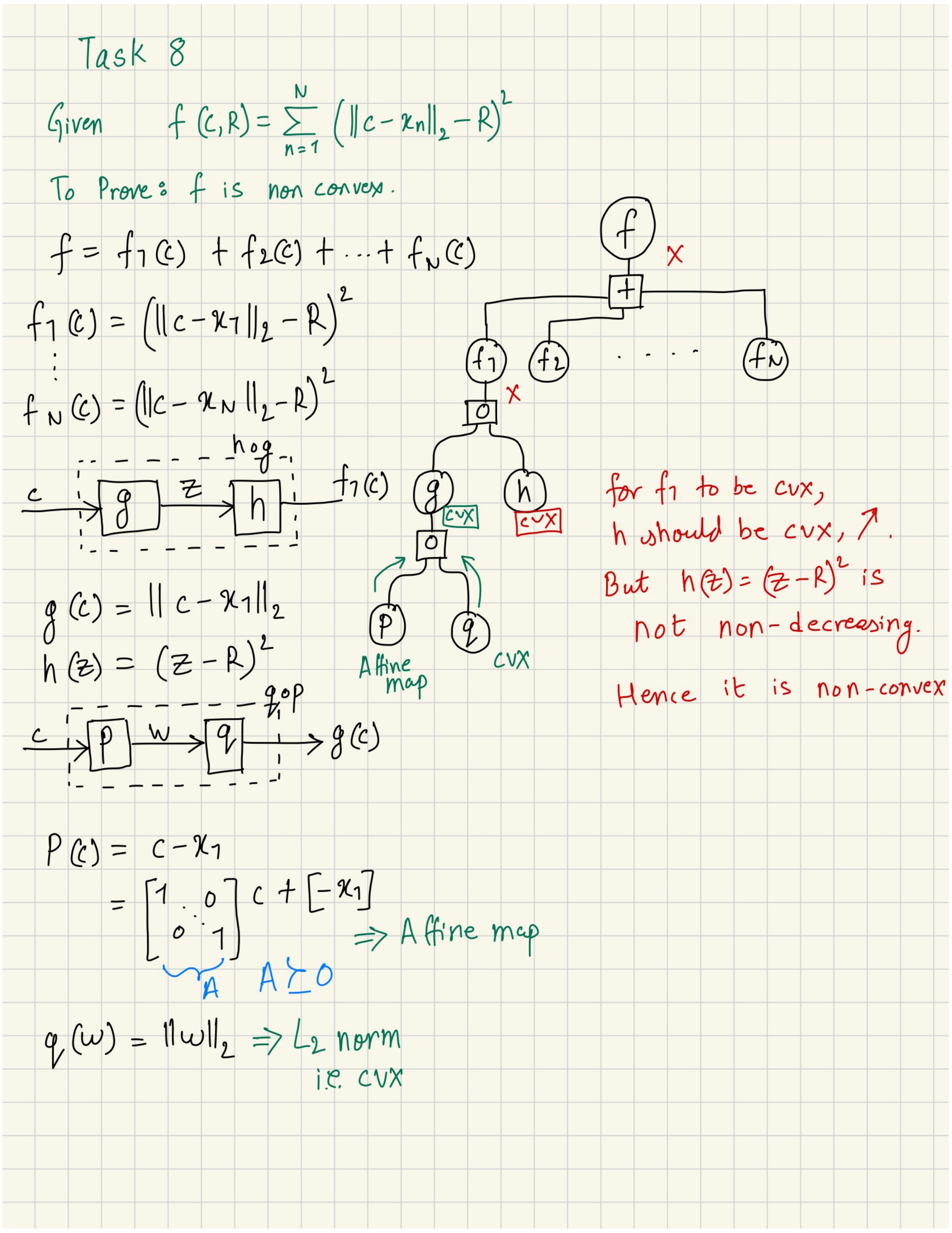
Figure 5: The two known possible trajectories that the target can execute, trajectory 1 (red) and trajectory 2 (magenta), and the two trajectories of our vehicle (blue, and black) corresponding to the control signals optimized as in (5) for $\rho = 0.1, p_1 = 0.6$ and $p_2 = 0.4$. The trajectories x_1 and x_2 split at $t = 25$.

7 Task 7

Figure 5 represents the resulting trajectories obtained by solving the optimization problem in CVXPY with parameters $\rho = 0.1, p_1 = 0.6,$ and $p_2 = 0.4$.

As shown, the two vehicle trajectories ($x_1(t)$ and $x_2(t)$) follow the same target trajectory up to t = 24, since the true target identity is not yet known. After t = 25, once the actual target is revealed, the two trajectories diverge and each vehicle continues to track its corresponding target independently.

8 Task 8



Task 8

Given $f(c,R) = \sum_{n=1}^{N} \left(\|c - x_n\|_2 - R\right)^2$

To Prove: f is non convex.

$$f = f_1(c) + f_2(c) + \dots + f_N(c)$$

$$f_1(c) = \left(\|c - x_1\|_2 - R\right)^2$$

$\vdots$

$$f_N(c) = \left(\|c - x_N\|_2 - R\right)^2$$

$$\begin{tikzcd} c \arrow[r] & \boxed{g} \arrow[r, "z"] & \boxed{h} \arrow[r] & f_1(c) \end{tikzcd}$$

$h \circ g$

$$g(c) = \|c - x_1\|_2$$

$$h(z) = (z - R)^2$$

$$\begin{tikzcd} c \arrow[r] & \boxed{p} \arrow[r, "w"] & \boxed{q} \arrow[r] & g(c) \end{tikzcd}$$

$q \circ p$

$$p(c) = c - x_1 = \underbrace{\begin{bmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{bmatrix}}_{A} c + \left[-x_1\right] \Rightarrow \text{Affine map}$$

$A \succeq 0$

$q(w) = \|w\|_2 \Rightarrow L_2$ norm

i.e. CVX

$$\begin{tikzcd}[column sep=small] & & f & \\ & & \boxed{+} \arrow[u] \arrow[r, phantom, "\times"] & \\ f_1 \arrow[urr] & f_2 \arrow[ur] & \cdots & f_N \arrow[ul] \\ \boxed{\circ} \arrow[u] \arrow[r, phantom, "\times"] & & & \\ g \arrow[u] & h \arrow[ul] & & \\ \boxed{\text{CVX}} & \boxed{\text{CVX}} & & \\ \boxed{\circ} \arrow[uu] & & & \\ p \arrow[u] & q \arrow[ul] & & \end{tikzcd}$$

Affine map

CVX

for f_1 to be cvx,

h should be cvx, $\nearrow$.

But $h(z) = (z-R)^2$ is

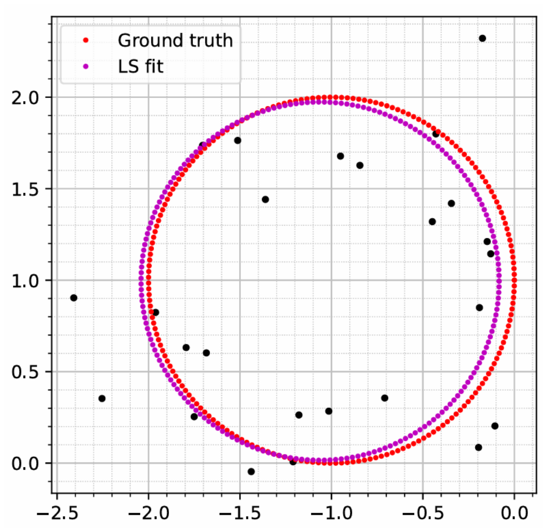
not non-decreasing.

Hence it is non-convex

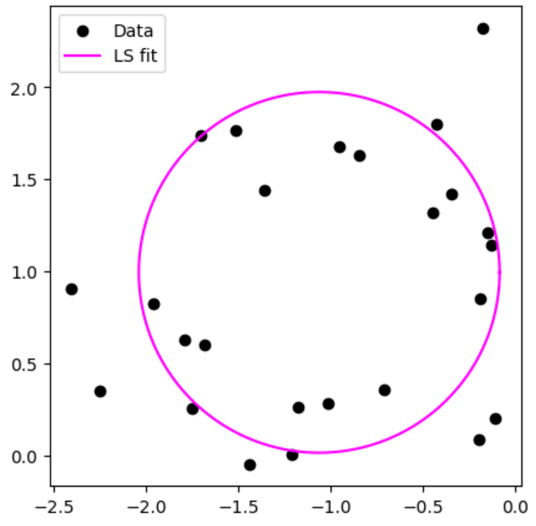
Figure 6: Proof of non convexity of $f(c, R)$

9 Task 9

As a premise to this task, we were encouraged to replicate Figure 7a, which we did and we got the center as (-1.06211003, 0.99475605) with radius 0.97887041. Our replication results can be seen in Figure 7b. So we can confirm that our LS implementation is correct.



(a) Ground truth and LS approach result provided in task description for `circle_data_1.npy`



(b) LS approach result from our implementation for `circle_data_1.npy`

Figure 7: Comparison between the provided LS results and our implementation, confirming the correctness of our Least Squares fitting.

We then applied this LS approach to `circle_data_2.npy` and got center c^*_{ls} as (-0.44638224, 1.50461084) and radius r^*_{ls} as 0.5930429544769911. The figure of the circle with center c^*_{ls} and radius r^*_{ls} superimposed to the dataset can be seen in Figure 8.

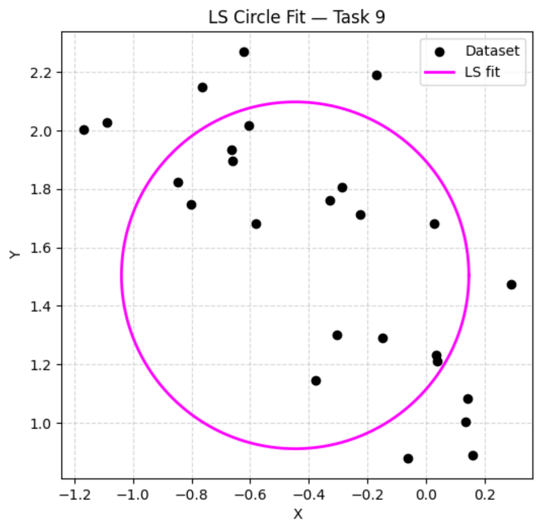
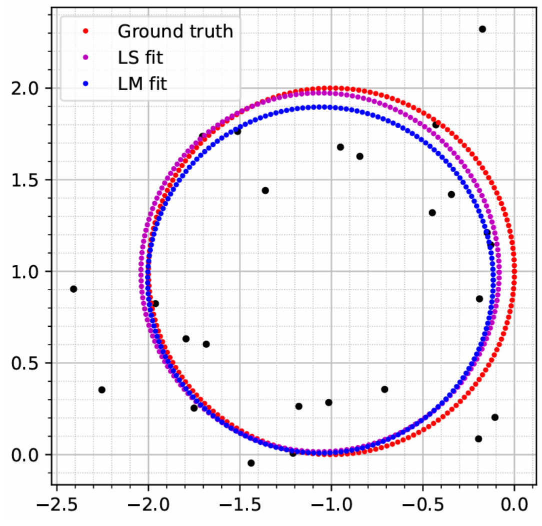


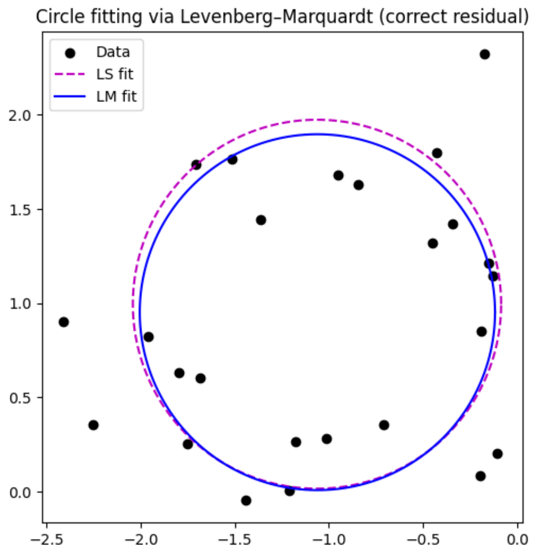
Figure 8: LS approach result from our implementation for `circle_data_2.npy`

10 Task 10

As a premise of this task, we were encouraged to replicate Figure 9a by implementing the Levenberg-Marquardt (LM) algorithm and applying it to `circle_data_1.npy`. Through our implementation, we arrive at center c^*_{lm} = (-1.05996716, 0.9523476) and radius R^*_{lm} = 0.9442665370892928, which is exactly what was expected. The circle obtained from our implementation can be seen in Figure 9b.



(a) Ground truth and circle fitted by the LS and LM approaches as provided in the task description for `circle_data_1.npy`

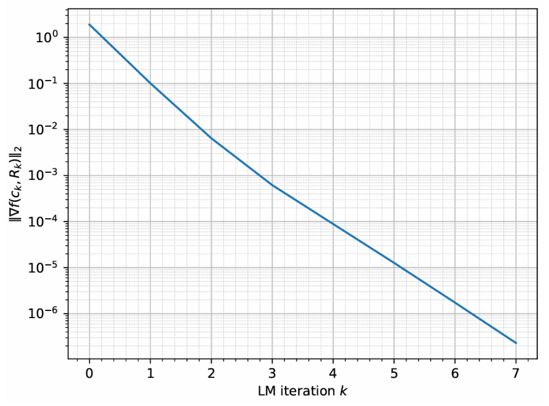


(b) Circles fitted by the LS and LM approaches from our implementation for `circle_data_1.npy`

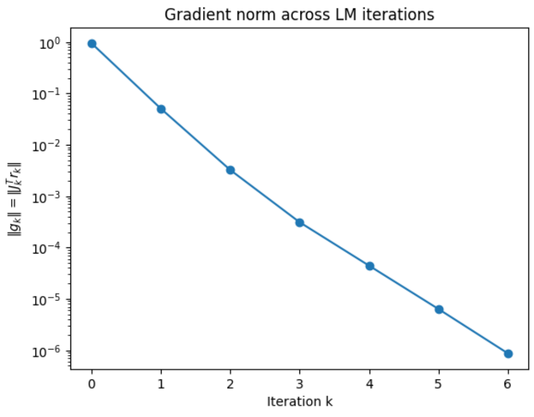
Figure 9: Comparison between the provided and implemented LM results for `circle_data_1.npy`. Both methods produce consistent circle fits, confirming the correctness of our LM implementation.

Furthermore, we were also encouraged to replicate Figure 10a, showing the variation of norm of the gradient of the cost function f across iterations of the LM algorithm. We did replicate it, shown in Figure 10b and it can be seen that the graphs are identical.

Then finally we carried out the actual Task 10, where we applied our implementation of the LM algorithm on `circle_data_2.npy`. We arrived at the center c^*_{lm} = (-0.87918437, 1.10441115) and radius R^*_{lm} = 0.9085570269017816. The plot showing the circle with center c^*_{lm} and radius R^*_{lm} superimposed to the dataset and the circle fitted by the LS approach can be seen in Figure 11a and the plot of the norm of the gradient of f across iterations of the LM algorithm can be seen in Figure 11b.

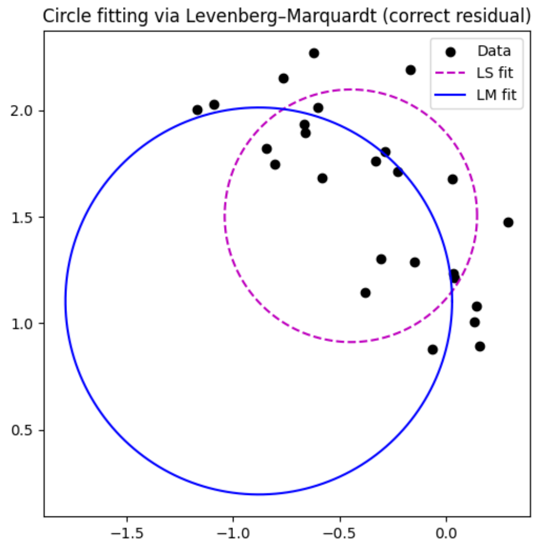


(a) Variation of the norm of the gradient of the cost function f across LM iterations, as provided in the task description for `circle_data_1.npy`

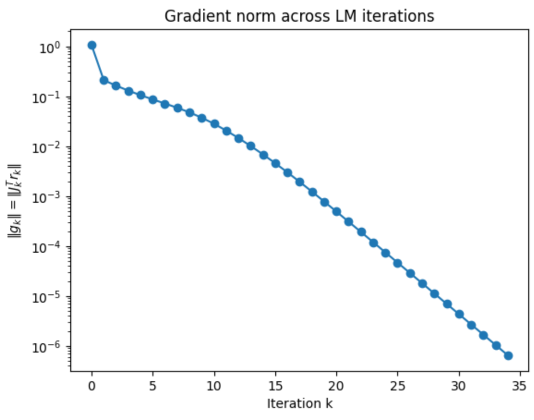


(b) Variation of the norm of the gradient of the cost function f across LM iterations from our own implementation for `circle_data_1.npy`

Figure 10: Comparison of the norm of the gradient of the cost function f across iterations of the LM algorithm for `circle_data_1.npy`. The identical trends in subfigures (a) and (b) confirm the correctness and numerical consistency of our LM implementation.



(a) Circle fitted by LM algorithm and LS approach on `circle_data_2.npy`



(b) Norm of the gradient of the cost function f across iterations of the LM algorithm for `circle_data_2.npy`

Figure 11: Results of applying our implementation of LM algorithm on `circle_data_2.npy`.

11 Task 11

The federated learning approach successfully recovered a circle that closely approximates the ground truth. The ALM algorithm converged in 4 iterations with a penalty parameter c = 100.0. The first ALM iteration required 170 BCD iterations to minimize the

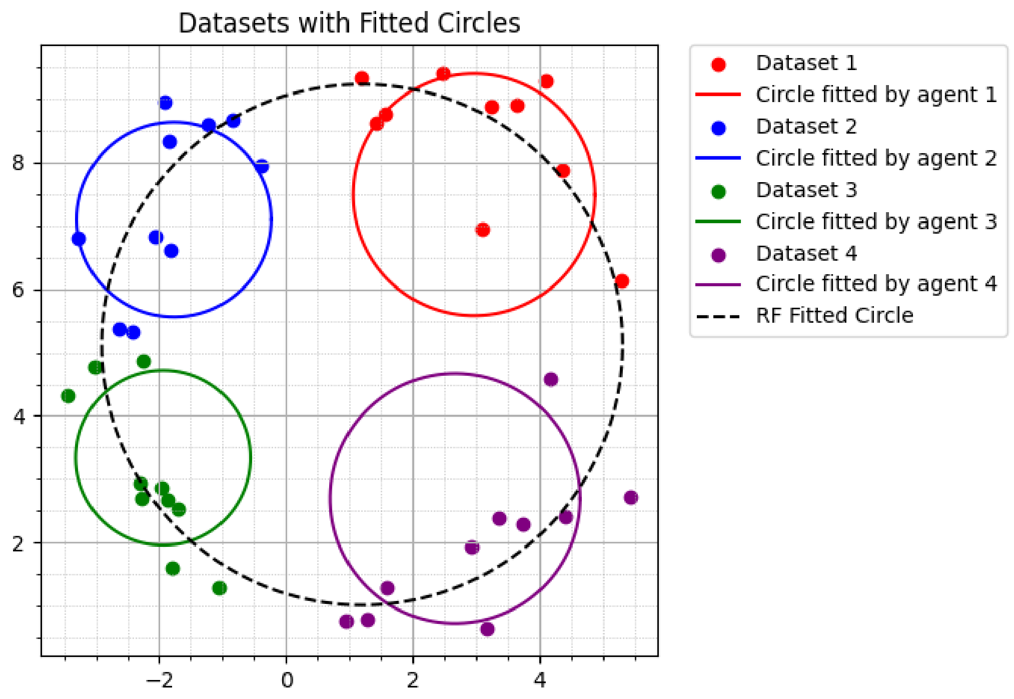


Figure 12: Scatter of the four agents' local datasets (colored points) with their independent least-squares circle fits (solid curves). The black dashed circle is the federated ALM/BCD consensus fit with center (1.221, 5.049) and radius 4.084, which closely matches the pooled-data baseline (1.203, 5.122) with radius 4.108.

augmented Lagrangian, but subsequent iterations converged rapidly (3, 3, and 2 BCD iterations respectively), indicating that the algorithm efficiently approached the solution, which is presented in Table 1. The maximum constraint violation across iterations is plotted in Figure 13.

The federated circle fit achieved a center of (1.221, 5.049) and radius of 4.084, whereas the centralized LS fit on the combined data gives center (1.203, 5.122) and radius 4.108.

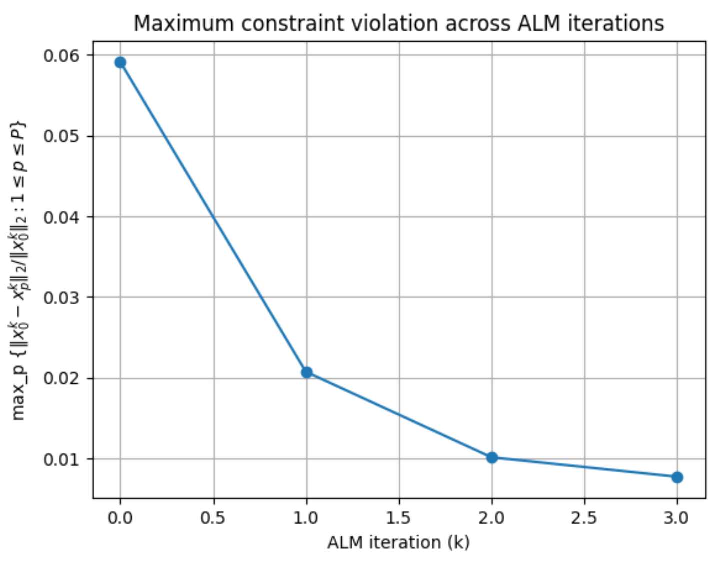


Figure 13: Maximum constraint violation across ALM iterations, for the case c = 100 ; ALM terminated after 4 iterations with BCD inner counts [170, 3, 3, 2].

Table 1: Number of BCD iterations per ALM iteration, for the case c = 100.

ALM iteration k	**Number of BCD iterations**
1	170
2	3
3	3
4	2

Appendix: Code Listings

Below are the code snippets related to the numerical tasks.

Task 1 Code

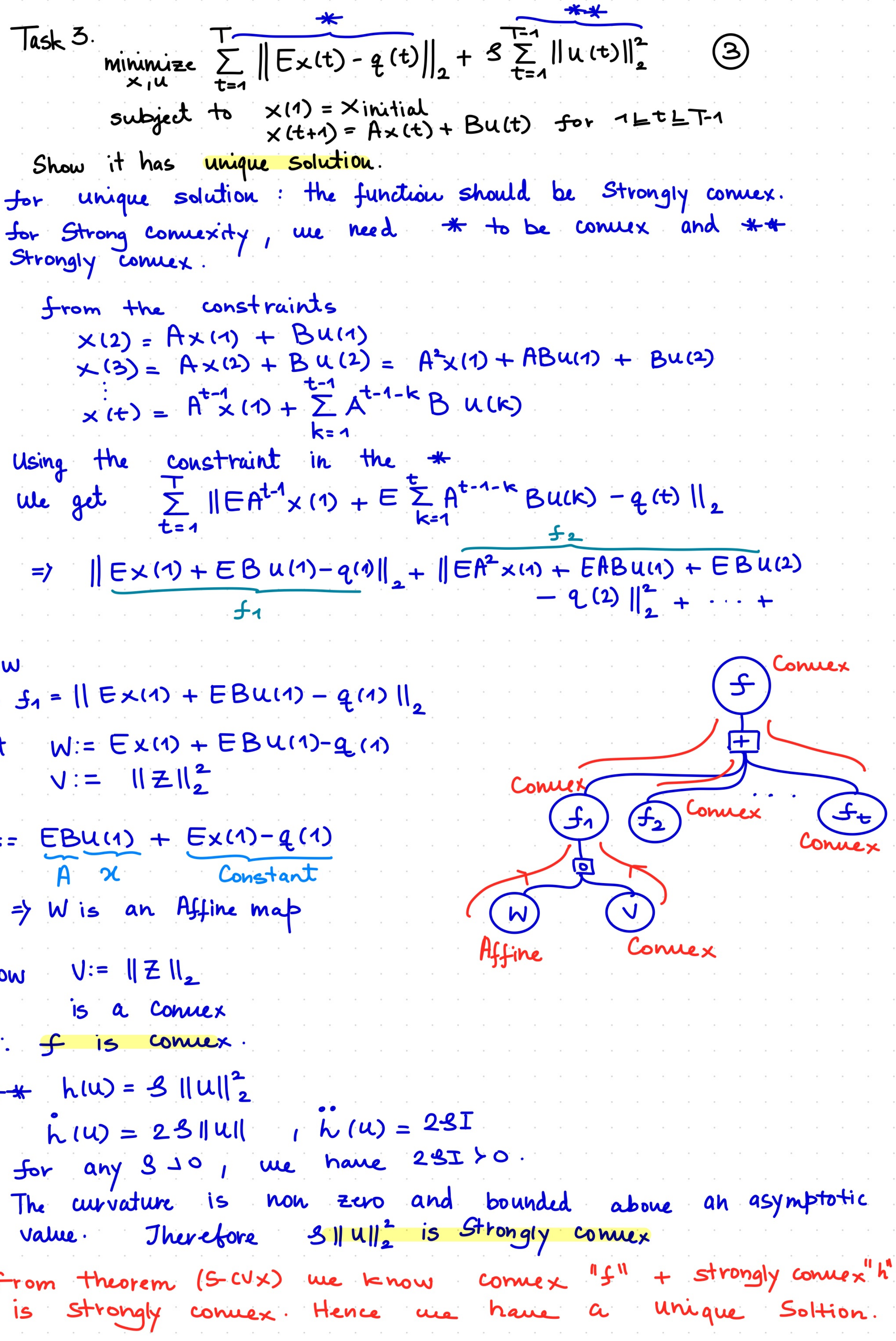
Listing 1: Python code for Task 1

```
import numpy as np
import matplotlib.pyplot as plt
import cvxpy as cp
import time

q_t = np.load("target_1.npy") # \in \mathcal{R}^{2}
print(q_t.shape)

```

```python
T = 60          # number of time steps
n = 4           # state dimension (p in R^2, v in R^2)

# State trajectory variable: shape (n, T)
x = cp.Variable((n, T))
x_rho = np.zeros((12, n, T))  # reference trajectory variable:
   shape (n, T)
u_rho = np.zeros((12, 2, T-1))  # reference control input
   variable: shape (n), T-1)

u = cp.Variable((2, T-1))  # control input variable: shape (2, T
   -1)

rho = [10, 5, 2, 1, .5, .2, .1, 0.05, 0.02, 0.01, 0.005, 0.002]
       # weight for control effort in cost function

E = np.array([[1, 0, 0, 0],
              [0, 1, 0, 0]])  # matrix to extract position from
                 state

A = np.array([[1, 0, 0.1, 0],
                [0, 1, 0, 0.1],
                [0, 0, 0.8, 0],
                [0, 0, 0, 0.8]])  # state transition matrix

B = np.array([[0, 0],
              [0, 0],
              [0.1, 0],
              [0, 0.1]])  # control input matrix

TE = cp.sum([cp.norm(E @ x[:, t] - q_t[:, t],2) for t in range(T)
   ])
# Control Effort term (sum of squared norms)
CE = cp.sum([cp.sum_squares(u[:, t]) for t in range(T-1)])

def cost_function(TE, CE, rho):
    J = TE + rho * CE

    objectives = cp.Minimize(J)

    return objectives
```

```python
constraints = []
for t in range(T-1):
    constraints += [x[:, t+1] == A @ x[:, t] + B @ u[:, t]]  #
        dynamics constraints

# Initial condition
constraints += [x[:, 0] == np.array([0.5, 0, 1, -1])]  # starting
    at origin with zero velocity

for i in range(len(rho)):
    r = rho[i]
    objectives = cost_function(TE, CE, r)
    prob = cp.Problem(objectives, constraints)
    time_start = time.time()
    result = prob.solve()
    time_end = time.time()
    x_rho[i, :, :] = x.value
    u_rho[i, :, :] = u.value
    print("Optimal value: ", result)
    print("Solve time: ", time_end - time_start)

# Create 3x4 subplots (3 rows, 4 cols)
fig, axes = plt.subplots(4, 3, figsize=(14, 20))

# Flatten axes for easy iteration
axes = axes.flatten()

# Plot each trajectory in a separate subplot
for i in range(12):
    ax = axes[i]
    ax.plot(q_t[0, :], q_t[1, :], linewidth=1, color='red',
        linestyle='-')
    ax.plot(q_t[0, :], q_t[1, :], 'o', color='red', markersize=3)
    ax.plot(x_rho[i, 0, :], x_rho[i, 1, :], linewidth=1, color='
        black', linestyle='-')
    ax.plot(x_rho[i, 0, :], x_rho[i, 1, :], 'o', color='black',
        markersize=2)
    ax.set_title(f'rho = {rho[i]}')
    ax.set_xlim([-1.5, 1.5])
    ax.set_ylim([-1.5, 1.5])
    ax.grid(True)
    ax.minorticks_on()
    ax.grid(which='major', linestyle='-', linewidth=0.8)
    ax.grid(which='minor', linestyle=':', linewidth=0.5)
    ax.set_xlabel('State of vehicle at time time (x(t))')

list = []
for i in range(12):
    TE = np.linalg.norm(E @ x_rho[i, :, :T-1] - q_t[0:2, 0:T-1],
       axis=0)
    CE = (np.linalg.norm(u_rho[i, :, :], axis=0))**2
    list.append((np.sum(TE), np.sum(CE)))

plt.figure(figsize=(6, 6))
x = [item[0] for item in list]  # TE
y = [item[1] for item in list]  # CE

plt.scatter(x, y, marker='o')
#plt.xscale('log')
#plt.yscale('log')
plt.xlabel('Tracking Error (TE)')
plt.ylabel('Control Effort (CE)')
plt.title('TE vs CE')

# Annotate each point
for (xi, yi, r) in zip(x, y, rho):
    plt.annotate(r"$\rho$"+f"={r}", (xi, yi), textcoords="offset
       points", xytext=(5,5), fontsize=8)

plt.grid(True)
plt.minorticks_on()
plt.grid(which='major', linestyle='-', linewidth=0.8)
plt.grid(which='minor', linestyle=':', linewidth=0.5)
plt.show()
```

Task 3 Solution



Task 3.

$$\underset{x,u}{\text{minimize}} \quad \overbrace{\sum_{t=1}^{T} \|Ex(t) - q(t)\|_2}^{*} + \delta \overbrace{\sum_{t=1}^{T-1} \|u(t)\|_2^2}^{**} \qquad ③$$

subject to $x(1) = x_{initial}$
$x(t+1) = Ax(t) + Bu(t)$ for $1 \leq t \leq T-1$

Show it has unique solution.

for unique solution : the function should be strongly convex.
for strong convexity , we need $*$ to be convex and $**$ strongly convex.

from the constraints

$$x(2) = Ax(1) + Bu(1)$$
$$x(3) = Ax(2) + Bu(2) = A^2x(1) + ABu(1) + Bu(2)$$
$$\vdots$$
$$x(t) = A^{t-1}x(1) + \sum_{k=1}^{t-1} A^{t-1-k} B\,u(k)$$

Using the constraint in the $*$
we get

$$\sum_{t=1}^{T} \|EA^{t-1}x(1) + E\sum_{k=1}^{t} A^{t-1-k}Bu(k) - q(t)\|_2$$

$$\Rightarrow \|\underbrace{Ex(1) + EBu(1) - q(1)}_{f_1}\|_2 + \overbrace{\|EA^2x(1) + EABu(1) + EBu(2) - q(2)\|_2^2}^{f_2} + \cdots +$$

now
for $f_1 = \|Ex(1) + EBu(1) - q(1)\|_2$

let $W := Ex(1) + EBu(1) - q(1)$
$V := \|Z\|_2^2$

$$W := \underbrace{EB}_{A}\underbrace{u(1)}_{x} + \underbrace{Ex(1) - q(1)}_{\text{Constant}}$$

$\Rightarrow$ W is an Affine map

now $V := \|Z\|_2$
is a convex

$\therefore$ f is convex.

$**$ $h(u) = \delta\|u\|_2^2$

$\dot{h}(u) = 2\delta\|u\|$, $\ddot{h}(u) = 2\delta I$

for any $\delta > 0$, we have $2\delta I > 0$.
The curvature is non zero and bounded above an asymptotic value. Therefore $\delta\|u\|_2^2$ is strongly convex

from theorem (S-CVX) we know convex "f" + strongly convex "h" is strongly convex. Hence we have a unique Soltion.

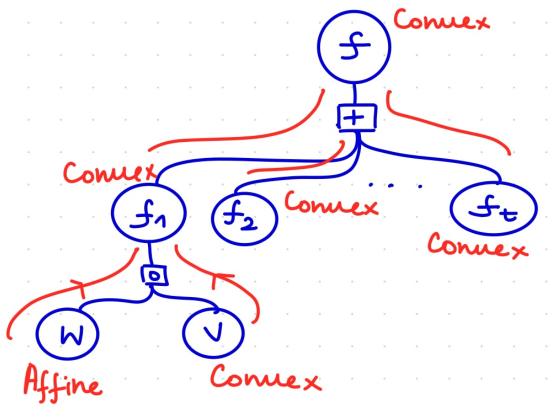


Figure 14: Proof for task 3

Task 4 Code

Listing 2: Python code for Task 4

```
import numpy as np
import matplotlib.pyplot as plt
import cvxpy as cp
import time

q_t1 = np.load("target_1.npy") # \in \mathcal{R}^{2}
print(q_t1.shape)
q_t2 = np.load("target_2.npy") #\in \mathcal{R}^{2}
print(q_t2.shape)

T = 60          # number of time steps
n = 4           # state dimension (p in R^2, v in R^2)

# State trajectory variable: shape (n, T)
x = cp.Variable((n, T))
x_i = np.zeros((11, n, T))  # reference trajectory variable:
   shape (n, T)
u_i = np.zeros((11, 2, T-1))  # reference control input variable:
    shape (n), T-1)

u = cp.Variable((2, T-1))  # control input variable: shape (2, T
   -1)



prior = [[0, 1], [0.1, 0.9], [0.2, 0.8], [0.3, 0.7], [0.4, 0.6],
   [0.5, 0.5], [0.6, 0.4], [0.7, 0.3], [0.8, 0.2], [0.9, 0.1],
   [1, 0]]     # prior probabilities

E = np.array([[1, 0, 0, 0],
              [0, 1, 0, 0]])  # matrix to extract position from
                 state

A = np.array([[1, 0, 0.1, 0],
                [0, 1, 0, 0.1],
                [0, 0, 0.8, 0],
                [0, 0, 0, 0.8]])  # state transition matrix

B = np.array([[0, 0],
              [0, 0],
              [0.1, 0],
              [0, 0.1]])  # control input matrix

# Initial state constraint
TE_1 = cp.sum([cp.norm(E @ x[:, t] - q_t1[:, t], 2) for t in
   range(T)])
TE_2 = cp.sum([cp.norm(E @ x[:, t] - q_t2[:, t], 2) for t in
   range(T)])
CE = cp.sum([cp.sum_squares(u[:, t]) for t in range(T-1)])

def cost_function(TE_1, TE_2, u, p1, p2, rho = 0.1):
    J = p1 * TE_1 + p2 * TE_2 + rho * CE

    objectives = cp.Minimize(J)

    return objectives


constraints = []
for t in range(T-1):
    constraints += [x[:, t+1] == A @ x[:, t] + B @ u[:, t]]  #
       dynamics constraints

# Initial condition
constraints += [x[:, 0] == np.array([0.5, 0, 1, -1])]  # starting
    at origin with zero velocity


for i in range(len(prior)):
    p1 = prior[i][0]
    p2 = prior[i][1]
    objectives = cost_function(TE_1, TE_2, u, p1, p2)
    prob = cp.Problem(objectives, constraints)
    time_start = time.time()
    result = prob.solve()
    time_end = time.time()
    x_i[i, :, :] = x.value
    u_i[i, :, :] = u.value
    print("Optimal value: ", result)
    print("Solve time: ", time_end - time_start)
```

```
# Create 3x4 subplots (3 rows, 4 cols)
fig, axes = plt.subplots(4, 3, figsize=(14, 20))

# Flatten axes for easy iteration
axes = axes.flatten()

# Plot each trajectory in a separate subplot
for i in range(11):
    ax = axes[i]
    ax.plot(q_t1[0, :], q_t1[1, :], linewidth=1, color='red',
       linestyle='-')
    ax.plot(q_t1[0, :], q_t1[1, :], 'o', color='red', markersize
       =3)
    ax.plot(q_t2[0, :], q_t2[1, :], linewidth=1, color='magenta',
        linestyle='-')
    ax.plot(q_t2[0, :], q_t2[1, :], 'o', color='magenta',
       markersize=3)
    ax.plot(x_i[i, 0, :], x_i[i, 1, :], linewidth=1, color='black
       ', linestyle='-')
    ax.plot(x_i[i, 0, :], x_i[i, 1, :], 'o', color='black',
       markersize=3)
    ax.set_title(f'$(p_1, p_2)$ = {prior[i][0]}, {prior[i][1]}')
    ax.set_xlim([-1.5, 1.5])
    ax.set_ylim([-1.5, 1.5])
    ax.grid(True)
    ax.minorticks_on()
    ax.grid(which='major', linestyle='-', linewidth=0.8)
    ax.grid(which='minor', linestyle=':', linewidth=0.5)
    ax.set_xlabel('State of vehicle at time time (x(t))')
```

Task 7 Code

Listing 3: Python code for Task 7

```
########################
Starting block same as Task 4 Code without the prior list
########################
# Initial state
TE_1 = cp.sum([cp.norm(E @ x1[:, t] - q_t1[:, t], 2) for t in
   range(T)])
TE_2 = cp.sum([cp.norm(E @ x2[:, t] - q_t2[:, t], 2) for t in
   range(T)])
CE_1 = cp.sum([cp.sum_squares(u1[:, t]) for t in range(T-1)])
CE_2 = cp.sum([cp.sum_squares(u2[:, t]) for t in range(T-1)])


def cost_function(TE_1, TE_2, p1=0.6, p2=0.4, rho = 0.1):
    J = p1 * (TE_1 + rho * CE_1) + p2 * (TE_2 + rho * CE_2)

    objectives = cp.Minimize(J)

    return objectives


constraints = []
for t in range(T-1):
    constraints += [x1[:, t+1] == A @ x1[:, t] + B @ u1[:, t]]  #
        dynamics constraints
    constraints += [x2[:, t+1] == A @ x2[:, t] + B @ u2[:, t]]  #
        dynamics constraints
# Initial condition
constraints += [x1[:, 0] == np.array([0.5, 0, 1, -1])]  #
   starting at origin with zero velocity
constraints += [x2[:, 0] == np.array([0.5, 0, 1, -1])]  #
   starting at origin with zero velocity

# Common control before t = 25
constraints += [u1[:, 0:24] == u2[:, 0:24]]


objectives = cost_function(TE_1, TE_2)
prob = cp.Problem(objectives, constraints)
```

```python
time_start = time.time()
result = prob.solve()
time_end = time.time()
print("Optimal value: ", result)
print("Solve time: ", time_end - time_start)

# Extract solutions
x1_val = x1.value
x2_val = x2.value

plt.figure(figsize=(6, 6))
plt.xlim([-1.5, 1.5])
plt.ylim([-1.5, 1.5])
plt.plot(q_t1[0,:], q_t1[1,:], linewidth=1, color='red',
   linestyle='-')
plt.plot(q_t1[0,:], q_t1[1,:], 'o', color='red', markersize=3,
   label='Target 1')
plt.plot(q_t2[0,:], q_t2[1,:], linewidth=1, color='magenta',
   linestyle='-')
plt.plot(q_t2[0,:], q_t2[1,:], 'o', color='magenta', markersize
   =3, label='Target 2')
plt.plot(x1_val[0,:], x1_val[1,:], linewidth=1, color='blue',
   linestyle='-')
plt.plot(x1_val[0,:], x1_val[1,:], 'o', color='blue', markersize
   =3, label='Trajectory following T1')
plt.plot(x2_val[0,:], x2_val[1,:], linewidth=1, color='black',
   linestyle='-')
plt.plot(x2_val[0,:], x2_val[1,:], 'o', color='black', markersize
   =3, label='Trajectory following T2')
plt.legend()
plt.grid(True)
plt.minorticks_on()
plt.grid(which='major', linestyle='-', linewidth=0.8)
plt.grid(which='minor', linestyle=':', linewidth=0.5)

plt.title('Optimal trajectory for p1=0.6, p2=0.1')
plt.show()
```

Task 9 Code

9.1 Solving the premise

Listing 4: Python code for premise task of Task 9

```
import numpy as np
import matplotlib.pyplot as plt
import cvxpy as cp
import time

x = np.load("circle_data_1.npy") # \in \mathcal{R}^{2}

A = np.array([[1, -2*x[0,0], -2*x[1,0]]])
for k in range(1,T):
    A = np.concatenate((A, [ [1, -2*x[0,k], -2*x[1,k]] ]), axis
       =0)

b = np.array( [[ -np.inner(x[:,0], x[:,0]) ]] )
for k in range(1,T):
    b = np.concatenate((b, [[-np.inner(x[:,k], x[:,k]) ]]), axis
       =0)

y_c = cp.Variable((3,1))

J_inside = A@y_c - b

J = cp.sum_squares(J_inside)

constraints = []

objective = cp.Minimize(J)
prob = cp.Problem(objective, constraints)
result = prob.solve()

c1 = y_c.value.T

c = c1[0][1:3]
R = np.sqrt(np.inner(c,c) - y_c[0].value)

print("Status: ", prob.status, "Optimal: ", prob.value, "y_c: ",
   y_c.value, "R: ", R)
```

```
# Plot result
theta = np.linspace(0, 2*np.pi, 200)
circle_x = c[0] + R * np.cos(theta)
circle_y = c[1] + R * np.sin(theta)

plt.figure(figsize=(5,5))
plt.scatter(x[0], x[1], color='black', label='Data')
plt.plot(circle_x, circle_y, color='magenta', label='LS fit')
plt.gca().set_aspect('equal')
plt.legend()
plt.show()
```

9.2 Solving the actual task

Listing 5: Python code for Task 9

```
x = np.load('circle_data_2.npy')

A = np.array([[1, -2*x[0,0], -2*x[1,0]]])
for k in range(1,T):
    A = np.concatenate((A, [ [1, -2*x[0,k], -2*x[1,k]] ]), axis
       =0)

b = np.array( [[ -np.inner(x[:,0], x[:,0]) ]] )
for k in range(1,T):
    b = np.concatenate((b, [[-np.inner(x[:,k], x[:,k]) ]]), axis
       =0)

prob = np.linalg.lstsq(A, b, rcond=None)
c1 = prob[0].flatten()
c = c1[1:3]
y = c1[0]

R = np.sqrt(np.inner(c, c) - y)

print("y:", y)
print("Center:", c)
print("Radius:", R)

theta = np.linspace(0, 2*np.pi, 500)
```

```
circle_x = c[0] + R * np.cos(theta)
circle_y = c[1] + R * np.sin(theta)

plt.figure(figsize=(6,6))
plt.scatter(x[0,:], x[1,:], color='black', label='Dataset',
   zorder=3)
plt.plot(circle_x, circle_y, color='magenta', label='LS fit',
   linewidth=2)
plt.gca().set_aspect('equal')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('LS Circle Fit-Task 9')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```

Task 10 Code

10.1 Solving the premise

Listing 6: Python code for premise of Task 10

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

x = np.load("circle_data_1.npy")

c = np.array([-1.06211003, 0.99475605])   # LS center
R = 0.97887041                            # LS radius
lam = 1.0                                 #       = 1
eps = 1e-6                                # stopping tolerance
max_iter = 100
eps_norm = 1e-12

grad_norms = []

print(f"{'Iter':>4} | {'  ':>8} | {'Cost':>12} | {'||g||':>10}")
print("-"*46)


for k in range(max_iter):
    # residuals: r_n = ||c - x_n|| - R
    d = np.sqrt((c[0] - x[0])**2 + (c[1] - x[1])**2)
    r = d - R

    # Jacobian: J_n = [ (c1 - x1n)/d_n, (c2 - x2n)/d_n, -1 ]
    # handle very small d by replacing with eps_norm
    d_safe = np.where(d < eps_norm, eps_norm, d)
    J = np.vstack([(c[0] - x[0]) / d_safe,
                   (c[1] - x[1]) / d_safe,
                   -np.ones_like(d_safe)]).T

    g = J.T @ r
    grad_norm = np.linalg.norm(g)
    grad_norms.append(grad_norm)
    f_old = np.sum(r**2)
    print(f"{k:4d} | {lam:8.3e} | {f_old:12.6e} | {grad_norm:10.3
       e}")

    if grad_norm < eps:
        break

    # Solve (J^T J + lambda I) delta = -J^T r
    H = J.T @ J + lam * np.eye(3)
    delta = np.linalg.lstsq(H, -g, rcond=None)[0]

    c_new = c + delta[:2]
    R_new = R + delta[2]

    # compute new cost
    d_new = np.sqrt((c_new[0] - x[0])**2 + (c_new[1] - x[1])**2)
    r_new = d_new - R_new
    f_new = np.sum(r_new**2)

    if f_new < f_old:           # valid step
        c, R = c_new, R_new
        lam *= 0.7
    else:                       # null step
        lam *= 2.0

print("-"*46)
print("Final LM center:", c)
print("Final LM radius:", R)

#Plot circles
theta = np.linspace(0, 2*np.pi, 200)
circle_ls_x = -1.06211003 + 0.97887041 * np.cos(theta)
circle_ls_y =  0.99475605 + 0.97887041 * np.sin(theta)
circle_lm_x = c[0] + R * np.cos(theta)
circle_lm_y = c[1] + R * np.sin(theta)

plt.figure(figsize=(6,6))
plt.scatter(x[0], x[1], color='black', label='Data')
plt.plot(circle_ls_x, circle_ls_y, 'm--', label='LS fit')
plt.plot(circle_lm_x, circle_lm_y, 'b', label='LM fit')
plt.gca().set_aspect('equal')
plt.legend()
```

```python
plt.title("Circle fitting via Levenberg-Marquardt (correct
    residual)")
plt.show()

#Plot gradient norm
plt.semilogy(grad_norms, 'o-')
plt.xlabel("Iteration k")
plt.ylabel(r"$\|g_k\| = \|J_k^T r_k\|$")
plt.title("Gradient norm across LM iterations")
plt.show()
```

10.2 Solving the actual task

Listing 7: Python code for Task 10

```python
import numpy as np
import matplotlib.pyplot as plt

x = np.load("circle_data_2.npy")

c = np.array([-0.44638224, 1.50461084])   # LS center
R = 0.5930429544769911                    # LS radius
lam = 1.0                                 #       = 1
eps = 1e-6                                # stopping tolerance
max_iter = 100
eps_norm = 1e-12

grad_norms = []

print(f"{'Iter':>4} | {'  ':>8} | {'Cost':>12} | {'||g||':>10}")
print("-"*46)


for k in range(max_iter):
    # residuals: r_n = ||c - x_n|| - R
    d = np.sqrt((c[0] - x[0])**2 + (c[1] - x[1])**2)
    r = d - R

    # Jacobian: J_n = [ (c1 - x1n)/d_n, (c2 - x2n)/d_n, -1 ]
    # handle very small d by replacing with eps_norm
    d_safe = np.where(d < eps_norm, eps_norm, d)
    J = np.vstack([(c[0] - x[0]) / d_safe,
                   (c[1] - x[1]) / d_safe,
                   -np.ones_like(d_safe)]).T

    g = J.T @ r
    grad_norm = np.linalg.norm(g)
    grad_norms.append(grad_norm)
    f_old = np.sum(r**2)

    print(f"{k:4d} | {lam:8.3e} | {f_old:12.6e} | {grad_norm:10.3
       e}")

    if grad_norm < eps:
        break

    # Solve (J^T J + lambda I) delta = -J^T r
    H = J.T @ J + lam * np.eye(3)
    delta = np.linalg.lstsq(H, -g, rcond=None)[0]

    c_new = c + delta[:2]
    R_new = R + delta[2]

    # compute new cost
    d_new = np.sqrt((c_new[0] - x[0])**2 + (c_new[1] - x[1])**2)
    r_new = d_new - R_new
    f_new = np.sum(r_new**2)

    if f_new < f_old:            # valid step
        c, R = c_new, R_new
        lam *= 0.7
    else:                        # null step
        lam *= 2.0

print("-"*46)
print("Final LM center:", c)
print("Final LM radius:", R)

#Plot circles
theta = np.linspace(0, 2*np.pi, 200)
circle_ls_x = -0.44638224 + 0.5930429544769911 * np.cos(theta)
circle_ls_y =  1.50461084 + 0.5930429544769911 * np.sin(theta)
```

```
circle_lm_x = c[0] + R * np.cos(theta)
circle_lm_y = c[1] + R * np.sin(theta)

plt.figure(figsize=(6,6))
plt.scatter(x[0], x[1], color='black', label='Data')
plt.plot(circle_ls_x, circle_ls_y, 'm--', label='LS fit')
plt.plot(circle_lm_x, circle_lm_y, 'b', label='LM fit')
plt.gca().set_aspect('equal')
plt.legend()
plt.title("Circle fitting via Levenberg Marquardt (correct
    residual)")
plt.show()

#Plot gradient norm
plt.semilogy(grad_norms, 'o-')
plt.xlabel("Iteration k")
plt.ylabel(r"$\|g_k\| = \|J_k^T r_k\|$")
plt.title("Gradient norm across LM iterations")
plt.show()
```

Task 11 Code

Listing 8: Python code for Task 11

```python
import numpy as np
import matplotlib.pyplot as plt

# Load datasets for the agents
x1 = np.load("dataset1.npy") # \in \mathcal{R}^{2}
x2 = np.load("dataset2.npy") # \in \mathcal{R}^{2}
x3 = np.load("dataset3.npy") # \in \mathcal{R}^{2}
x4 = np.load("dataset4.npy") # \in \mathcal{R}^{2}


def fit_circle(x):
    """
    Fit a circle to 2D points using least squares.

    Parameters
    ----------
    x : np.ndarray
        2 T  array of points [ [x1, x2, ...], [y1, y2, ...] ].

    Returns
    -------
    center : np.ndarray
        (2,) array representing the circle center.
    radius : float
        Estimated radius of the circle.
    circle_points : tuple of np.ndarray
        (circle_x, circle_y) for plotting the circle.
    """

    T = x.shape[1]
    A = np.array([[1, -2*x[0,0], -2*x[1,0]]])
    for k in range(1,T):
        A = np.concatenate((A, [ [1, -2*x[0,k], -2*x[1,k]] ]),
           axis=0)

    b = np.array( [[ -np.inner(x[:,0], x[:,0]) ]] )
    for k in range(1,T):
        b = np.concatenate((b, [[-np.inner(x[:,k], x[:,k]) ]]),
           axis=0)

    prob = np.linalg.lstsq(A, b, rcond=None)
    c1 = prob[0].flatten()
    c = c1[1:3]
    y = c1[0]

    R = np.sqrt(np.inner(c, c) - y)

    R = np.sqrt(np.inner(c, c) - y)

    # Precompute circle coordinates for convenience
    theta = np.linspace(0, 2*np.pi, 500)
    circle_x = c[0] + R * np.cos(theta)
    circle_y = c[1] + R * np.sin(theta)

    return c, R, (circle_x, circle_y)

plt.figure(1)
colors = ['red', 'blue', 'green', 'purple', "black"]
datasets = [x1, x2, x3, x4, np.concatenate((x1, x2, x3, x4), axis
   =1)]

for i, (x, color) in enumerate(zip(datasets, colors)):
    # Fit circle
    center, radius, (circle_x, circle_y) = fit_circle(x)
    print(f"Dataset {i+1}: Center = {center}, Radius = {radius:.3
       f}")

    # Scatter data points
    if color != "black":  # Avoid duplicate colors for combined
       dataset
        plt.scatter(x[0], x[1], color=color, marker='o', label=f'
           Dataset {i+1}')
    # Plot fitted circle
    if color == "black":
        plt.plot(circle_x, circle_y, color=color, linestyle='--',
            label=f'Ground Truth Circle')
    else:
        plt.plot(circle_x, circle_y, color=color, linestyle='--',
            label=f'Circle fitted by agent {i+1}')

plt.grid(True)
plt.minorticks_on()
plt.grid(which='major', linestyle='-', linewidth=0.8)
plt.grid(which='minor', linestyle=':', linewidth=0.5)
plt.title("Datasets with Fitted Circles")

# Legend outside the plot
plt.legend(loc='upper left', bbox_to_anchor=(1.05, 1),
   borderaxespad=0.)
plt.gca().set_aspect('equal', adjustable='box')
plt.tight_layout()
plt.show()

# helper: build local A_p and b_p from 2xT dataset of size (2,
   10)
def circle_ls_form(x):
    """
    x: 2 x T numpy array
    returns A: T x 3, b: T x 1 (column)
    A row for point k: [1, -2 x_k, -2 y_k]
    b row: - (x_k^2 + y_k^2)

    corresponds to the equation:
    (x - a)^2 + (y - b)^2 = r^2 Equation of a circle with center
       (a, b) and radius r, expanding:
    => x^2 + y^2 - 2 a x - 2 b y + (a^2 + b^2 - r^2) = 0
    => 1 * (a^2 + b^2 - r^2) + (-2 x) * a + (-2 y) * b = - (x^2 +
        y^2)
    => [1, -2x, -2y] [ (a^2 + b^2 - r^2), a, b ]^T = - (x^2 + y
       ^2)
    Vector form: A c = b, where c = [ (a^2 + b^2 - r^2), a, b ]^T
    """
    T = x.shape[1]
    A = np.column_stack((np.ones(T), -2 * x[0, :], -2 * x[1, :]))
         # T x 3
    b = -np.sum(x**2, axis=0).reshape(-1, 1)  # T x 1
    return A, b
```

```
def x_to_circle(x):
    """
    x: (3,) where x[0] = y, x[1:3] = c (center)
    returns center (2,), radius (float)
    radius = sqrt(||c||^2 - y)
    """
    x = x.flatten()
    y = x[0]
    c_x = x[1]
    c_y = x[2]
    c = np.array([c_x, c_y])
    val = np.dot(c, c) - y
    if val < 0:
        # numerical issues: clamp to zero
        val = max(val, 0.0)
    R = np.sqrt(val)
    return (c_x, c_y), R


def ALM(data, c=100.0, eps_ALM=0.01, eps_BCD=0.01, verbose=True):
    """
    data: list of P arrays, each 2 x T (2x10)
    c: ALM penalty parameter
    eps_ALM: stopping tol for ALM (on max constraint violation)
    """

    P = len(data)
    A_list = []
    b_list = []
    for p in range(P):
        A_p, b_p = circle_ls_form(data[p])
        A_list.append(A_p)
        b_list.append(b_p)
    # Initialize
    x0 = np.zeros((3,1)) # global variable
    x_p_list = [np.zeros((3, 1)) for _ in range(P)]  # local
       variables
    lambda_list = [np.zeros((3, 1)) for _ in range(P)]  # dual
       variables

    # History and counters
    alm_iter = 0
    alm_history_violation = []
    bcd_iters_per_alm = []

    while True:
        alm_iter += 1
        # Step 1: Minimize L_c over x_p (local variables)
        bcd_iter = 0
        # initialize inner variables for BCD using current outer
           state
        u0 = x0.copy()
        u_p = [xp.copy() for xp in x_p_list]
        lam_bcd = [lam.copy() for lam in lambda_list]  # lambda
           are fixed in inner minimization
        while True:
            bcd_iter += 1
            max_change = 0.0
            # Update each x_p
            for p in range(P):
                A_p = A_list[p]
                b_p = b_list[p]
                lam_p = lam_bcd[p]
                # Solve for u_p:
                # solve: \sum_p ||A_p x_p - b_p||^2 + (c/2) ||u_p
                    - u0||^2 + lam_p^T (u_p - u0)
                # => (c/2) ||u0 - u_p||^2 + lam_p^T (u0 - u_p) =
                   c/2 ||u_p - (u0 - lam_p/c)||^2 - (1/2c) ||
                   lam_p||^2
                # => min_u_p ||A_p u_p - b_p||^2 + (c/2) ||u_p -
                   (u0 - lam_p/c)||^2 - (1/2c) ||lam_p||^2
                # => min_u_p ||A_p u_p - b_p||^2 + (c/2) ||u_p -
                   d_p||^2, where d_p = u0 - lam_p/c
                # => min_u_p ||[A_p; sqrt(c/2) I] u_p - [b_p;
                   sqrt(c/2) d_p]||^2
                d_p = u0 - (lam_p / c)
                A_aug = np.vstack((A_p, np.sqrt(c / 2) * np.eye
                   (3)))
                b_aug = np.vstack((b_p, np.sqrt(c / 2) * d_p))

                # Solve least-squares for u_p_new
                u_p_new, *_ = np.linalg.lstsq(A_aug, b_aug, rcond
                   =None)


                ########### Alternative direct solution
                   ############
                # Exact analytical minimizer:
                # (2 A_p^T A_p + c I) x_p = 2 A_p^T b_p + c d_p
                #M = 2 * (A_p.T @ A_p) + c * np.eye(3)
                #rhs = 2 * (A_p.T @ b_p).flatten() + c * d_p.
                   flatten()
                #u_p_new = np.linalg.solve(M, rhs).reshape(3, 1)
                change = (np.linalg.norm(u_p_new - u_p[p])/np.
                   linalg.norm(u_p[p]+0.0000001))  # relative
                   change


                u_p[p] = u_p_new
                if change > max_change:
                    max_change = change
            # Update u0 from the derivative condition:
            # \sum_p [ - lam_p - c (u_p - u0) ] = 0
            # => \sum_p [ c u0 ] = \sum_p [ lam_p + c u_p ]
            # => P c u0 = \sum_p [ lam_p + c u_p ]
            # => u0 = (1/(P c)) \sum_p [ lam_p + c u_p ]
            change0 = np.sum([lam_bcd[p] + c * u_p[p] for p in
               range(P)], axis=0) / (P * c)
            change0_norm = np.linalg.norm(change0 - u0)
            u0 = change0
            if change0_norm > max_change:
                max_change = change0_norm
            # Check BCD stopping criterion
            if max_change < eps_BCD:
                break
        # End of BCD inner loop
        # Update outer variables
        bcd_iters_per_alm.append(bcd_iter)
        x_p_list = [u_p[p].copy() for p in range(P)]
        x0 = u0.copy()
        # Step 2: Update dual variables
        max_violation = 0.0
```

```
        for p in range(P):
            violation = x_p_list[p] - x0
            lambda_list[p] += c * violation
            violation_norm = np.linalg.norm(violation) / np.
               linalg.norm(x0)  # relative violation
            if violation_norm > max_violation:
                max_violation = violation_norm
        alm_history_violation.append(max_violation)
        print(f"ALM Iteration {alm_iter}: BCD iters = {bcd_iter}"
           )
        # Check ALM stopping criterion
        if max_violation < eps_ALM:
            break
    info = {
        "alm_iters": alm_iter,
        "bcd_iters_per_alm": bcd_iters_per_alm,
        "violation_history": alm_history_violation
    }

    return x0, x_p_list, lambda_list, info

data = [x1, x2, x3, x4]

# Run ALM+BCD
x0_final, x_p_final, lam_final, info = ALM(data, c=100.0, eps_ALM
   =0.01, eps_BCD=0.01, verbose=True)

plt.figure(1)
colors = ['red', 'blue', 'green', 'purple', "orange"]
datasets = [x1, x2, x3, x4, np.concatenate((x1, x2, x3, x4), axis
   =1)]

for i, (x, color) in enumerate(zip(datasets, colors)):
    # Fit circle
    center, radius, (circle_x, circle_y) = fit_circle(x)

    # Scatter data points
    if color != "orange":  # Avoid duplicate colors fobr combined
        dataset
        print(f"Dataset {i+1}: Center = {center}, Radius = {
           radius:.3f}")
        plt.scatter(x[0], x[1], color=color, marker='o', label=f'
           Dataset {i+1}')
    # Plot fitted circle
    if color == "orange":
        print(f"Ground Truth: Center = {center}, Radius = {radius
           :.3f}")
        #plt.plot(circle_x, circle_y, color=color, linestyle
           ='--', label=f'Dataset Circle')
    else:
        plt.plot(circle_x, circle_y, color=color, linestyle='-',
           label=f'Circle fitted by agent {i+1}')
(c_x, c_y), R_final = x_to_circle(x0_final)
print(f"RF fitted: Center = {[c_x, c_y]}, Radius = {R_final:.3f}"
   )
plt.plot(circle_x, circle_y, color='black', linestyle='--', label
   ='RF Fitted Circle')
plt.grid(True)
plt.minorticks_on()
plt.grid(which='major', linestyle='-', linewidth=0.8)
plt.grid(which='minor', linestyle=':', linewidth=0.5)
plt.title("Datasets with Fitted Circles")

# Legend outside the plot
plt.legend(loc='upper left', bbox_to_anchor=(1.05, 1),
   borderaxespad=0.)
plt.gca().set_aspect('equal', adjustable='box')
plt.tight_layout()
plt.show()

# plot violation history
plt.figure()
plt.plot(info['violation_history'], marker='o')
plt.yscale('linear')
plt.xlabel('ALM iteration (k)')
plt.ylabel('max_p {$\| x_0^k - x_p^k\|_2/\|x_0^k\|_2: 1\leq p \
   leq P$}')
plt.title('Maximum constraint violation across ALM iterations')
plt.grid(True)
plt.show()
# print summary
print("\nALM summary:")
print("ALM iterations:", info['alm_iters'])
print("BCD iters per ALM iter:", info['bcd_iters_per_alm'])
# final center/radius info
c_final, R_final = x_to_circle(x0_final)
print(f"Federated (ALM) center: {c_final}, radius: {R_final:.6f}"
    )
```