Optimization and Algorithms Project

Authors:

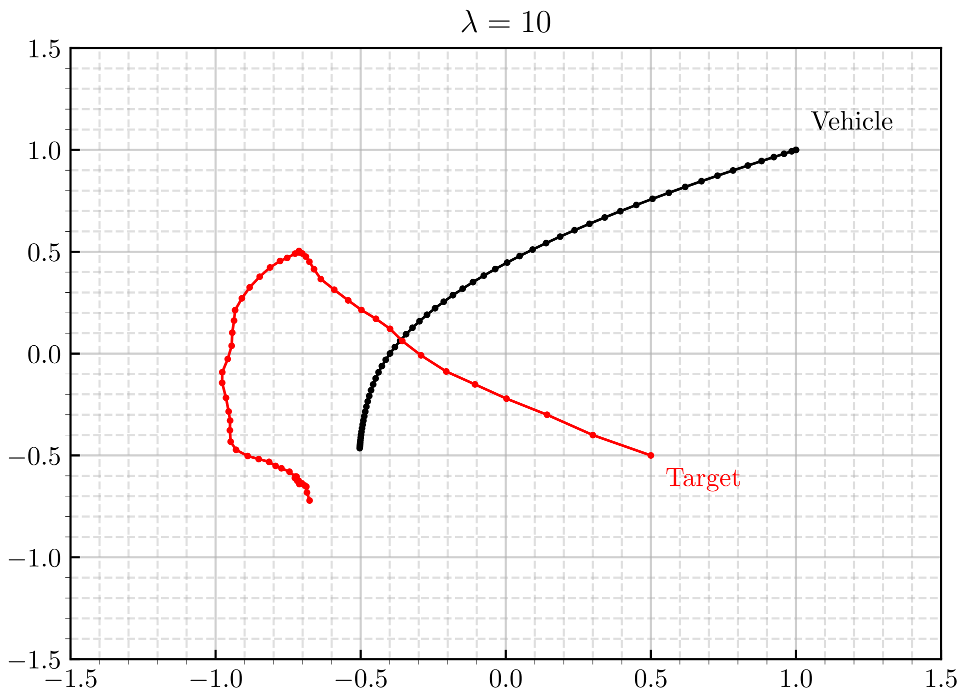
Duarte Calado de Almeida (95565)
Francisco Manuel Leal Mithá Ribeiro (95578)
José Pedro Baptista de Figueiredo (96259)
João De Assis Marcos Soares Nabais (97349)

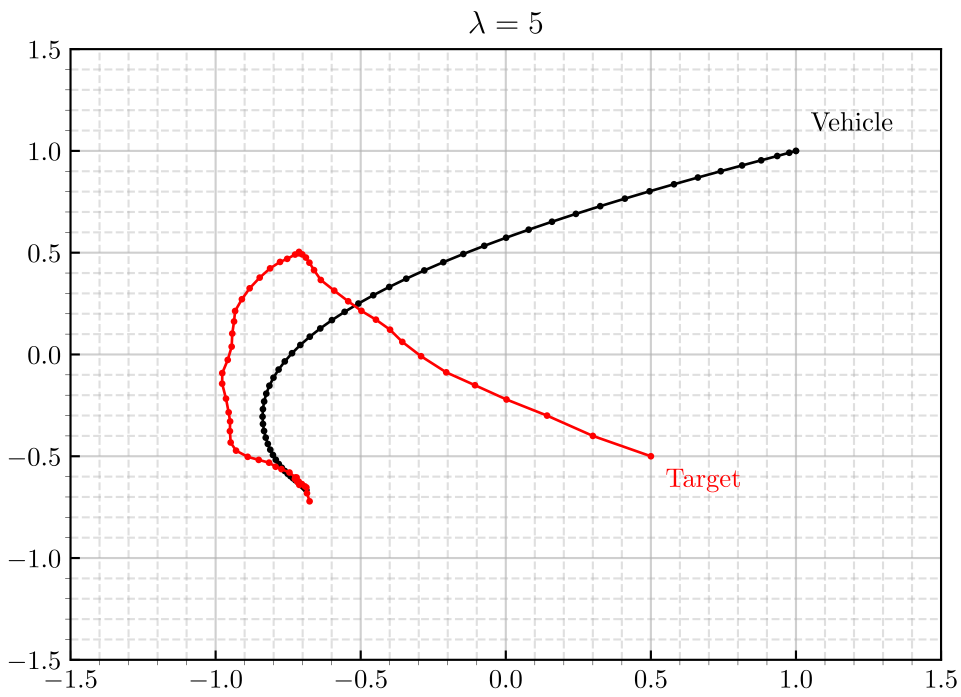
Group 8

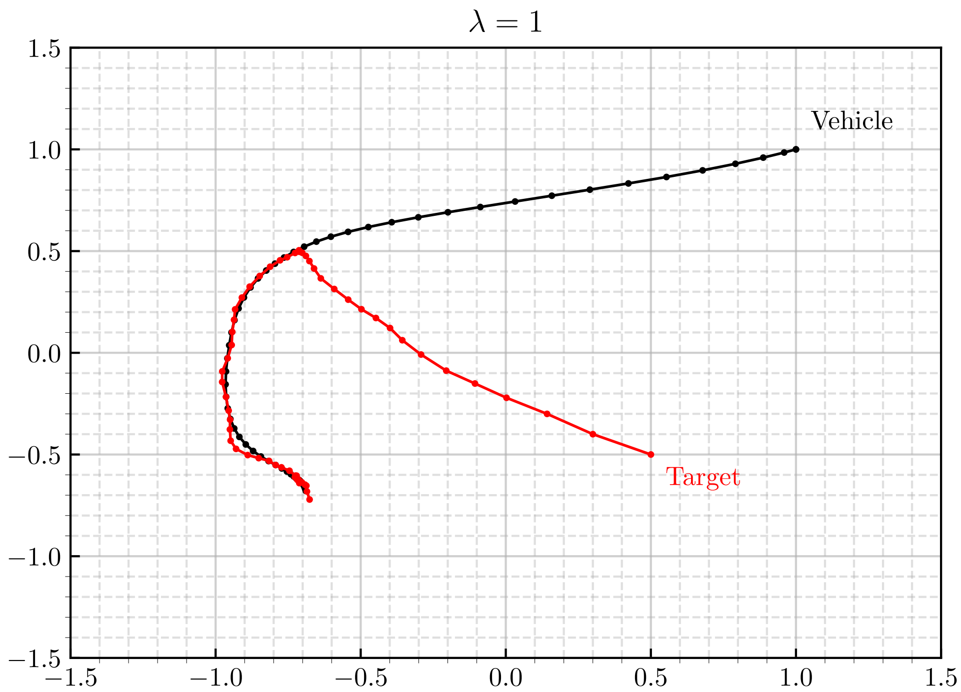
2022/2023 – 1º Semester, P1

Contents

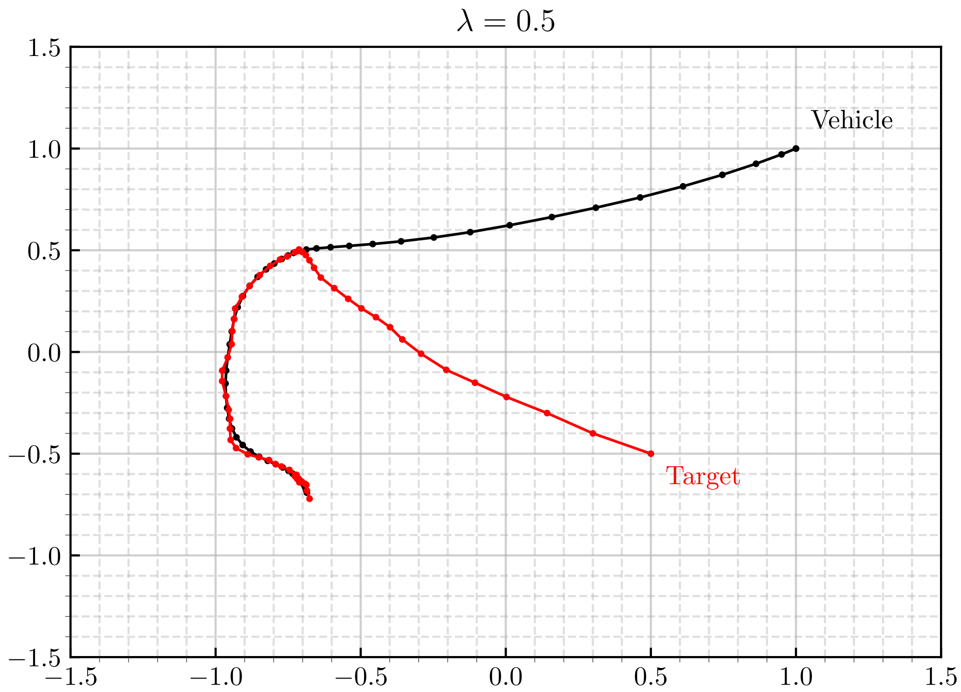
1 Task 1


$\lambda = 10$
Vehicle
Target
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5 −1.0 −0.5 0.0 0.5 1.0 1.5

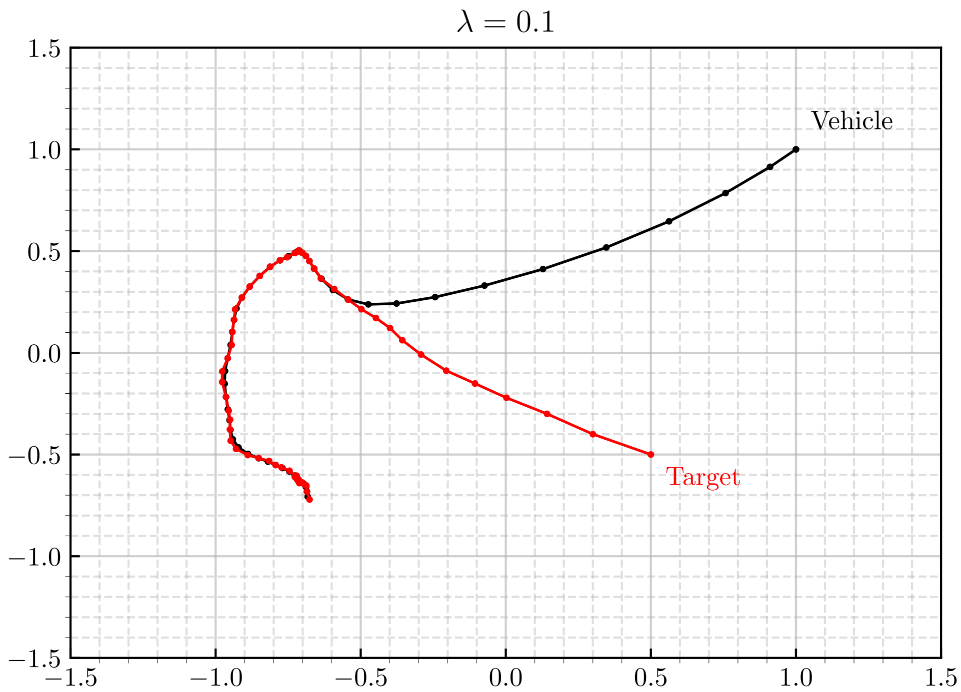

$\lambda = 5$
Vehicle
Target
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5 −1.0 −0.5 0.0 0.5 1.0 1.5

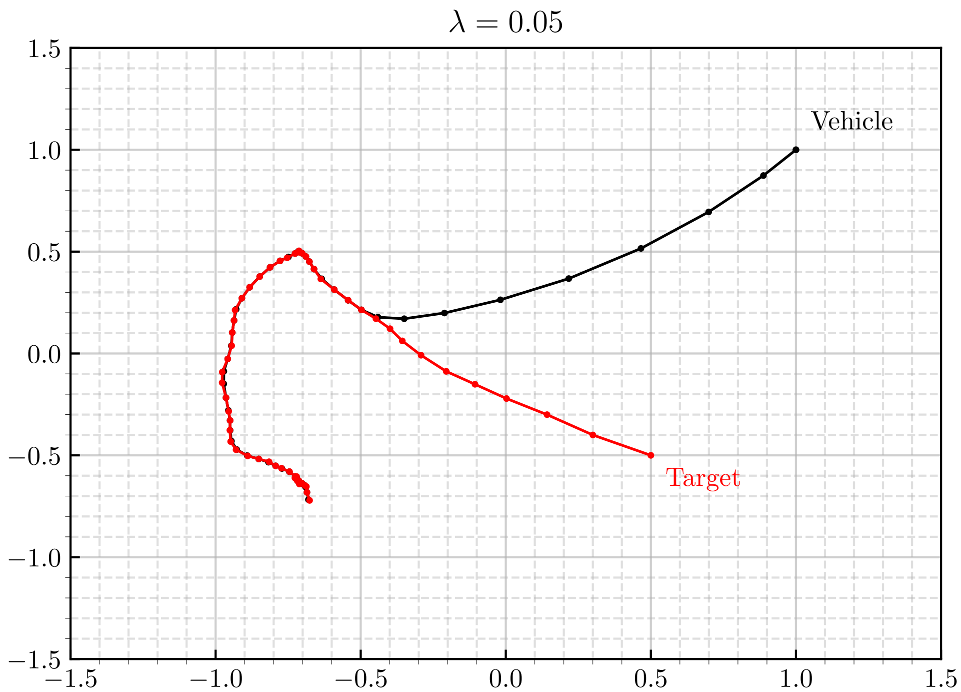


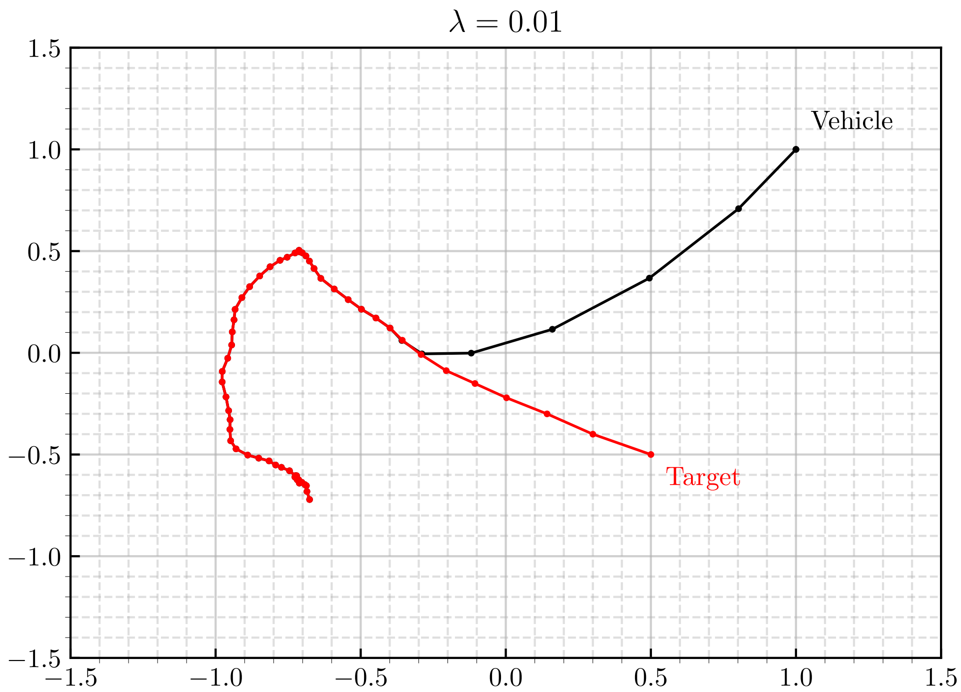
$\lambda = 1$
Vehicle
Target
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

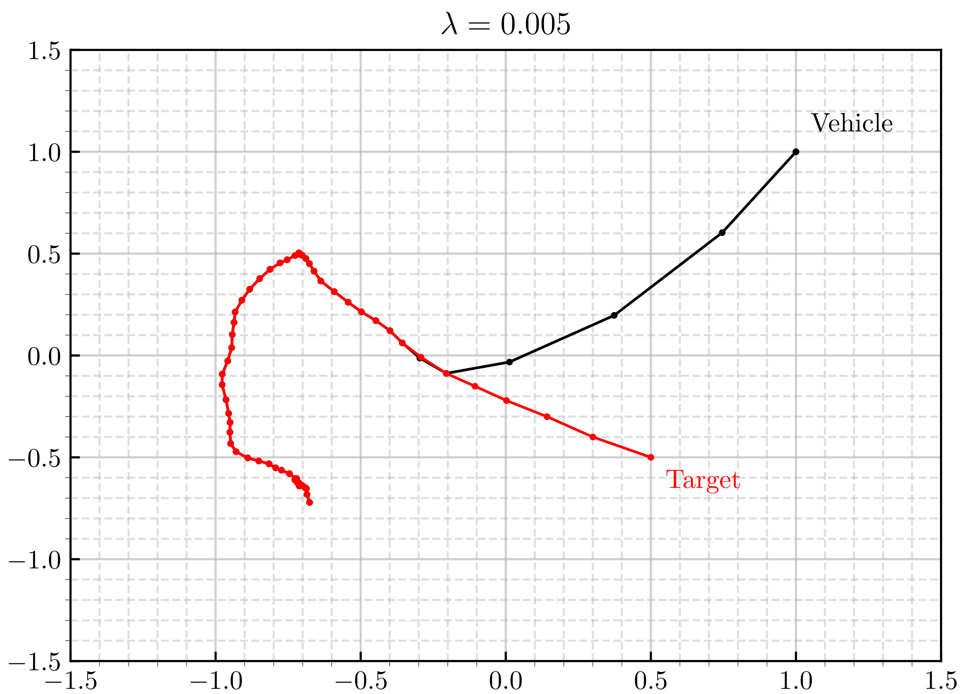


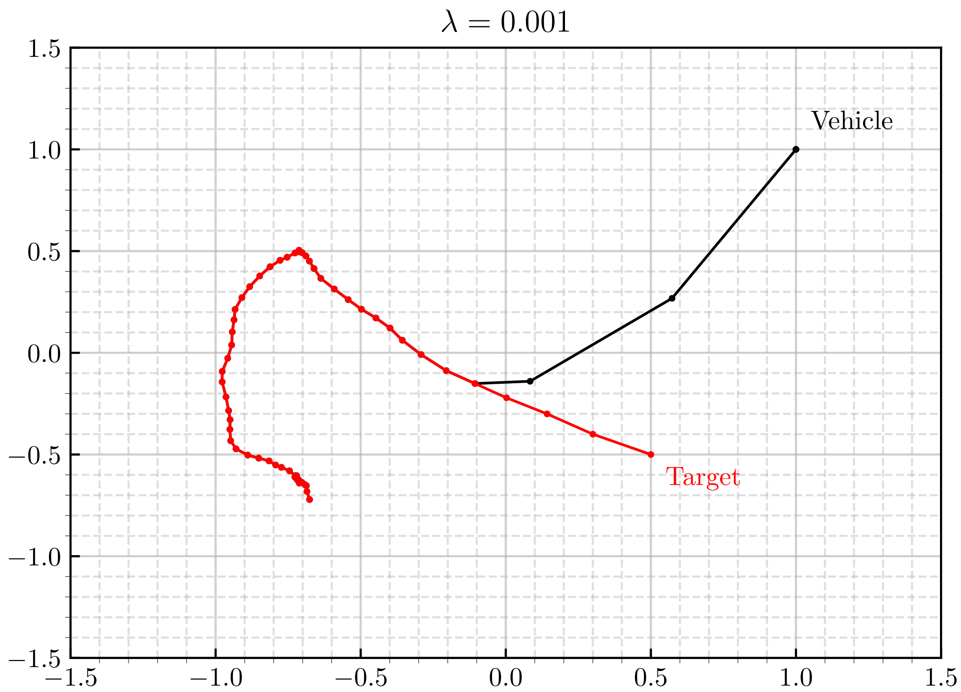
$\lambda = 0.5$
Vehicle
Target
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

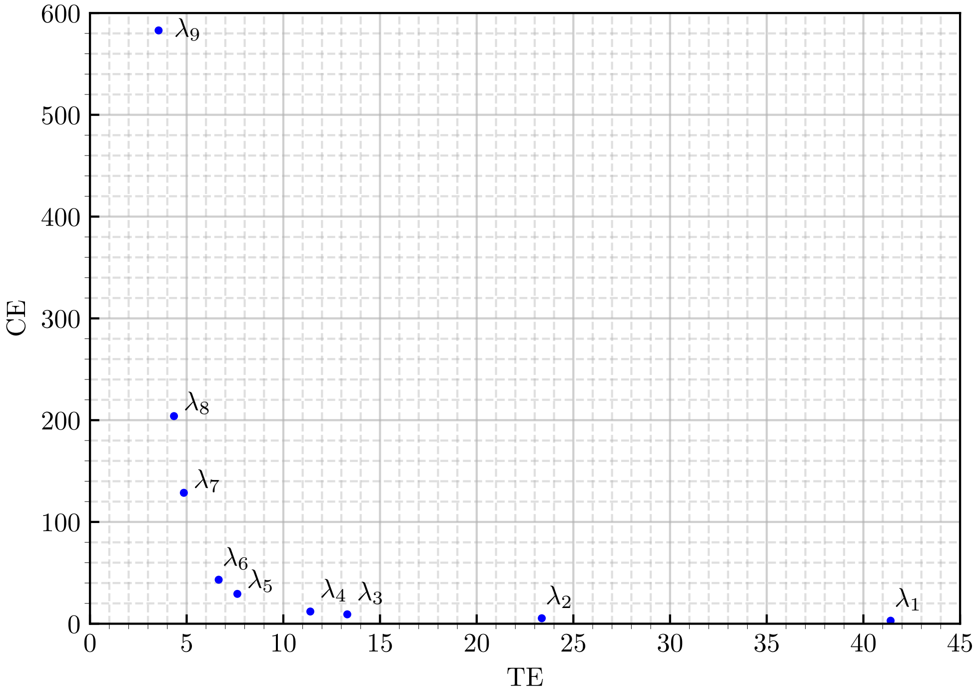

$\lambda = 0.1$
Vehicle
Target


$\lambda = 0.05$
Vehicle
Target


$\lambda = 0.01$
Vehicle
Target


$\lambda = 0.005$
Vehicle
Target





Upon analyzing the plots above, one concludes that a decrease in the parameter λ of the optimization problem yields a solution x^* to the corresponding optimization problem such that:

1. the resulting vehicle trajectory is overall less dissimilar to that of the target (when it comes to proximity between positions in the two trajectories associated with the same time instant).

2. the value of the resulting Tracking Effort (TE) decreases, while the corresponding Control Error (CE) registers an increase.

In order to give a rationale for these observations, notice that, for a given value of $(x, u) = (x(1), ..., x(T), u(1), ..., u(T-1))$, the cost function of the optimization problem can be written as:

$$f(x, u) = \text{TE}(x, u) + \lambda\text{CE}(x, u)$$

As such, we can argue that:

– a decrease in λ diminishes the importance of the parcel $\lambda\text{CE}(x, u)$ relative to that of $\text{TE}(x, u)$ in the objective function. Hence, bigger decreases in the cost function are more easily attained by lowering the TE instead of the CE, explaining the lower value for the TE that is obtained at an optimal solution. Furthermore, since the parcel TE pertains to the overall proximity of the positions of the target and the vehicle's trajectories at each instant t, it also follows that these trajectories tend to become similar with the shrinkage of λ.

– Conversely, an increase in λ diminishes the importance of the parcel $\text{TE}(x, u)$ relative to that of $\lambda\text{CE}(x, u)$ in the objective function, resulting in a lower value for the CE at an optimal solution, following an analogous reasoning to the one above. Moreover, we have that lower values for the CE result in lower norms for u and, consequently, for $u(t)$ $(t = 1, ..., T-1)$. This, together with the relation

$$x(t+1) = Ax(t) + Bu(t)$$

present in the constraints, explains the fact that changes in the state x also tend to be smaller in norm. Therefore, a lesser degree of similarity between the trajectories is attained, given that the target and the vehicle start at different positions and that the process of reaching the same position at a given instant in time is hindered.

2 Task 2

Let (x_a, u_a) and (x_b, u_b) denote some minimizers obtained after solving the given optimization problem for $\lambda = \lambda_a$ and $\lambda = \lambda_b$, respectively. Let $\text{TE}(x, u)$ and $\text{CE}(x, u)$ denote the Tracking Error and Control Effort for a given value of $(x, u) = (x(1), ..., x(T), u(1), ..., u(T-1))$, respectively. Then, suppose that $\text{TE}(x_a, u_a) \leq \text{TE}(x_b, u_b)$.

Since (x_b, u_b) minimizes the cost function for $\lambda = \lambda_b$, it follows that:

$$\begin{aligned}\text{TE}(x_b, u_b) + \lambda_b\text{CE}(x_b, u_b) &\leq \text{TE}(x_a, u_a) + \lambda_b\text{CE}(x_a, u_a) && (1)\\ &\leq \text{TE}(x_b, u_b) + \lambda_b\text{CE}(x_a, u_a) && (2)\end{aligned}$$

where we used the hypothesis that $\text{TE}(x_a, u_a) \leq \text{TE}(x_b, u_b)$ do derive (2) from (1). In particular, we have that:

$$\begin{aligned}
\mathrm{TE}(x_b, u_b) + \lambda_b \mathrm{CE}(x_b, u_b) &\leq \mathrm{TE}(x_b, u_b) + \lambda_b \mathrm{CE}(x_a, u_a) \\
\Leftrightarrow \lambda_b \mathrm{CE}(x_b, u_b) &\leq \lambda_b \mathrm{CE}(x_a, u_a) \\
\Leftrightarrow \mathrm{CE}(x_b, u_b) &\leq \mathrm{CE}(x_a, u_a)
\end{aligned}$$

with the last inequality coming from the fact that $\lambda_b > 0$. We have thus proven the desired result. $\square$

3 Task 3

To prove that the problem has a unique solution we begin by converting it to an unconstrained optimization problem. By solving the recurrence relation in the constraints we get:

$$\begin{aligned}
x(t) &= Ax(t-1) + Bu(t-1) \Leftrightarrow \\
\Leftrightarrow x(t) &= A^2 x(t-2) + ABu(t-2) + Bu(t-1) \Leftrightarrow \\
&\vdots \\
\Leftrightarrow x(t) &= A^{(t-1)} x_{\text{initial}} + \sum_{i=1}^{t-1} A^{(i-1)} Bu(i)
\end{aligned}$$

This closed form can be confirmed using induction in t. The induction basis ($t = 1$) is trivially verified (considering that a summation whose lower limit exceeds its upper limit evaluates to zero):

$$\begin{aligned}
x(1) = A^0 x(1) &= A^0 x_{\text{initial}} + \sum_{i=1}^{0} A^{(i-1)} Bu(i) \\
&= A^{(1-1)} x_{\text{initial}} + \sum_{i=1}^{1-1} A^{(i-1)} Bu(i)
\end{aligned}$$

For the induction step, we have:

$$\begin{aligned}
x(t+1) &= Ax(t) + Bu(t) \\
&= A \left(A^{(t-1)} x_{\text{initial}} + \sum_{i=1}^{t-1} A^{(i-1)} Bu(i) \right) + Bu(t) \\
&= A^{((t+1)-1)} x_{\text{initial}} + \sum_{i=1}^{t-1} A^{(i+1-1)} Bu(i) + Bu(t) \\
&= A^{((t+1)-1)} x_{\text{initial}} + \sum_{i=2}^{(t+1)-1} A^{(i-1)} Bu(i) + A^{(1-1)} Bu(t) \\
&= A^{((t+1)-1)} x_{\text{initial}} + \sum_{i=1}^{(t+1)-1} A^{(i-1)} Bu(i)
\end{aligned}$$

Plugging this into the objective function we now get an unconstrained version of the original problem:

$$\underset{u}{\text{minimize}} \quad \underbrace{\sum_{t=1}^{T} ||E(A^{(t-1)}x_{\text{initial}} + \sum_{i=1}^{t-1} A^{(i-1)}Bu(i)) - q(t)||_\infty + \lambda \sum_{t=1}^{T-1} ||u(t)||_2^2}_{\phi(u)}$$

Consider now the following decomposition for the function ϕ:

$$\phi(u) = \sum_{t=1}^{T} g_t(u) + h(u)$$

where

$$\begin{aligned} g_t(u) &= ||E(A^{(t-1)}x_{\text{initial}} + \sum_{i=1}^{t-1} A^{(i-1)}Bu(i)) - q(t)||_\infty \\ h(u) &= \lambda \sum_{t=1}^{T-1} \|u(t)\|_2^2 = \lambda \left(u_1^2(1) + u_2^2(1) + ... + u_1^2(T-1) + u_2^2(T-1))\right) \\ &= \lambda \|u\|_2^2 = u^T(\lambda I_2)u \end{aligned}$$

Clearly, h is a strongly convex function, since it is a quadratic function where the matrix associated with the quadratic form present in its expression (λI_2) is positive definite, since all its eigenvalues are λ and $\lambda > 0$.

On another hand, for $t = 1, ..., T$, g_t can be re-written as:

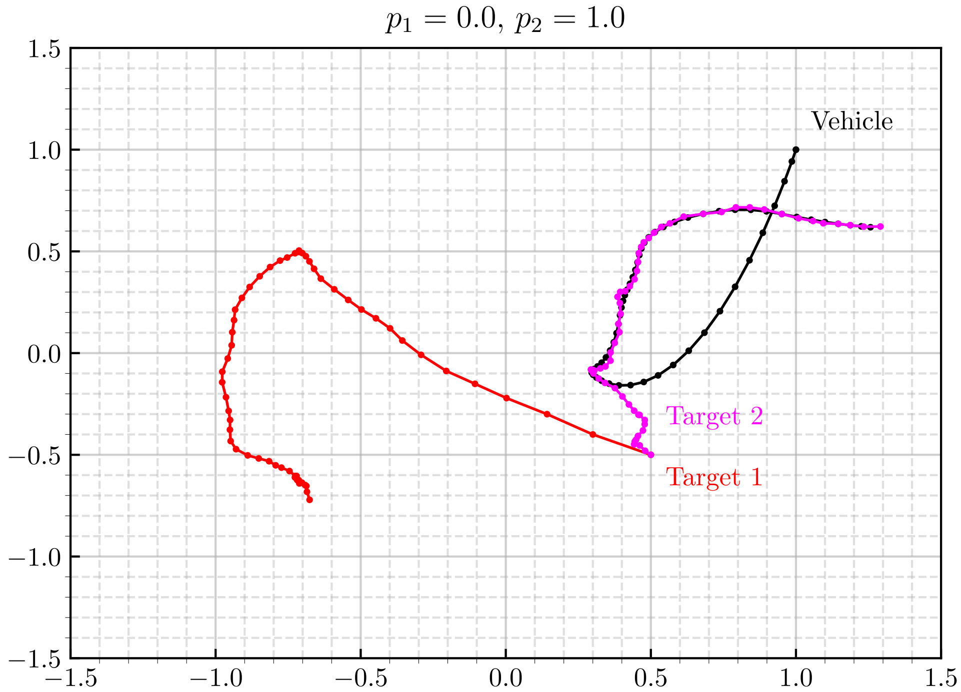
$$\begin{aligned} g_t(u) &= \left\| \sum_{i=1}^{t-1} EA^{(i-1)}Bu(i) + EA^{(t-1)}x_{\text{initial}} - q(t) \right\|_\infty \\ &= \left\| \underbrace{\begin{bmatrix} EB & EAB & EA^2B & ... & EA^{(t-2)}B \end{bmatrix}}_{\mathcal{A}} \begin{bmatrix} u(1) \\ u(2) \\ u(3) \\ ... \\ u(t-1) \end{bmatrix} + \underbrace{EA^{(t-1)}x_{\text{initial}} - q(t)}_{\beta} \right\|_\infty \\ &= \|\mathcal{A}u_{t-1} + \beta\|_\infty = (N \circ T)(u_{t-1}) \end{aligned}$$

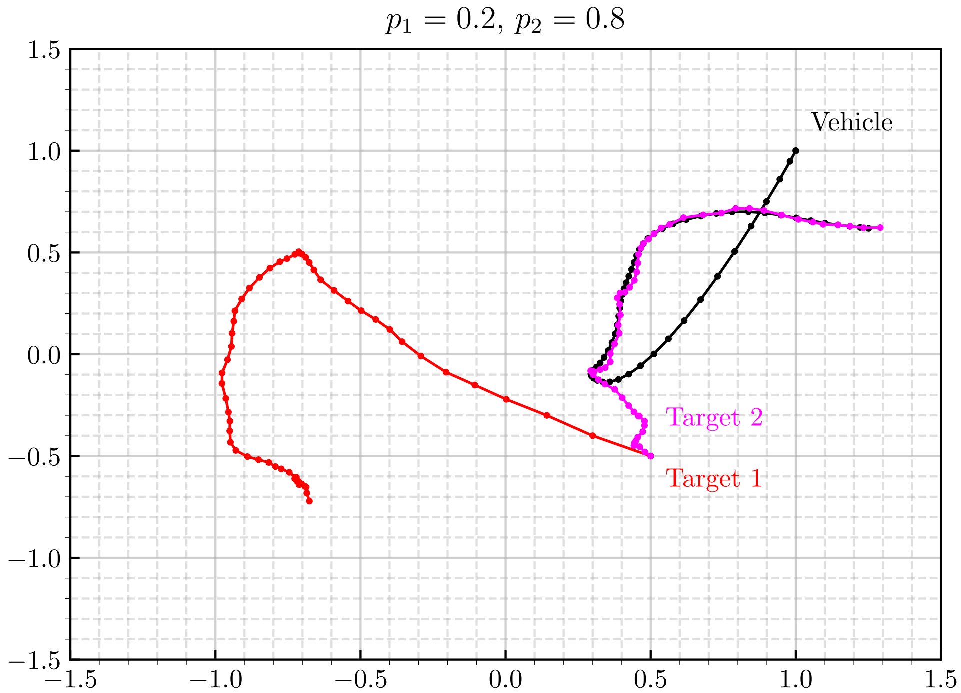
where $T(u_{t-1}) = \mathcal{A}u_{t-1} + \beta$ and $N(z) = \|z\|_\infty$.

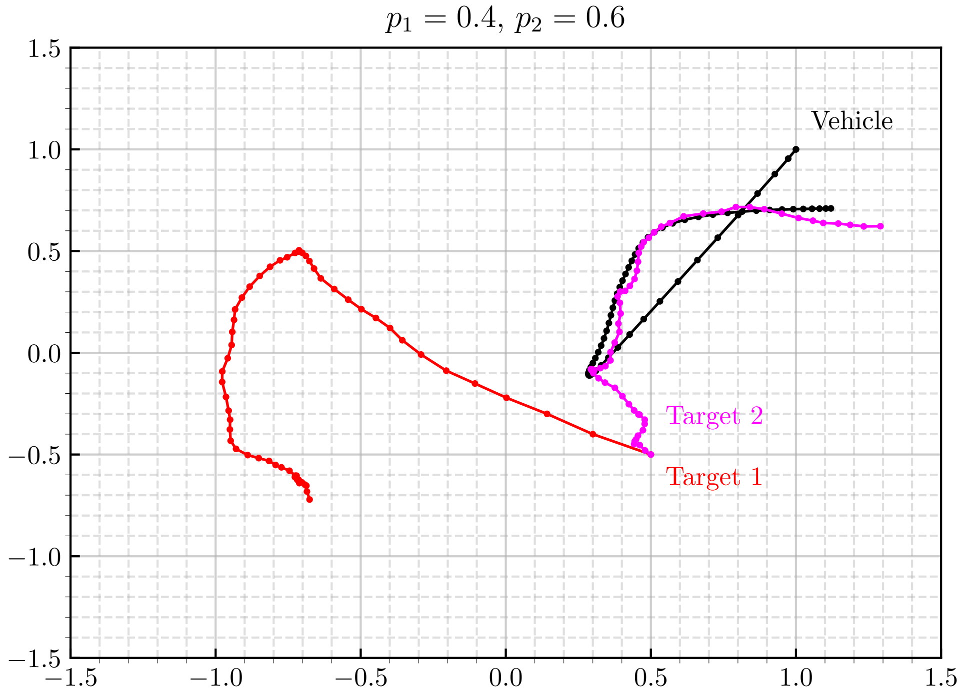
Since N is a convex function (it is a norm) and T is an affine map, it follows that $g_t(u)$ is a convex function. Moreover, $\sum_{t=1}^{T} g_t(u)$ is also a convex function, since it is a conic combination of convex functions.

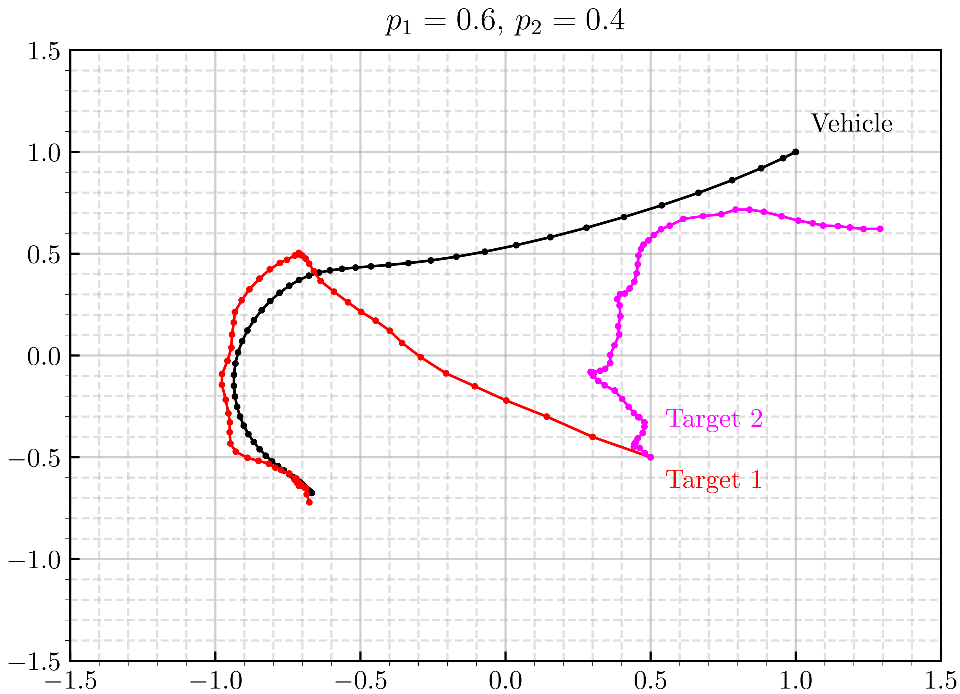
In conclusion, the cost function is the sum of a convex function ($\sum_{t=1}^{T} g_t(u)$) with a strongly convex function ($h(u)$), so it's itself a strongly convex function, thereby proving it has a unique global minimizer and so the optimization problem has a unique solution. □

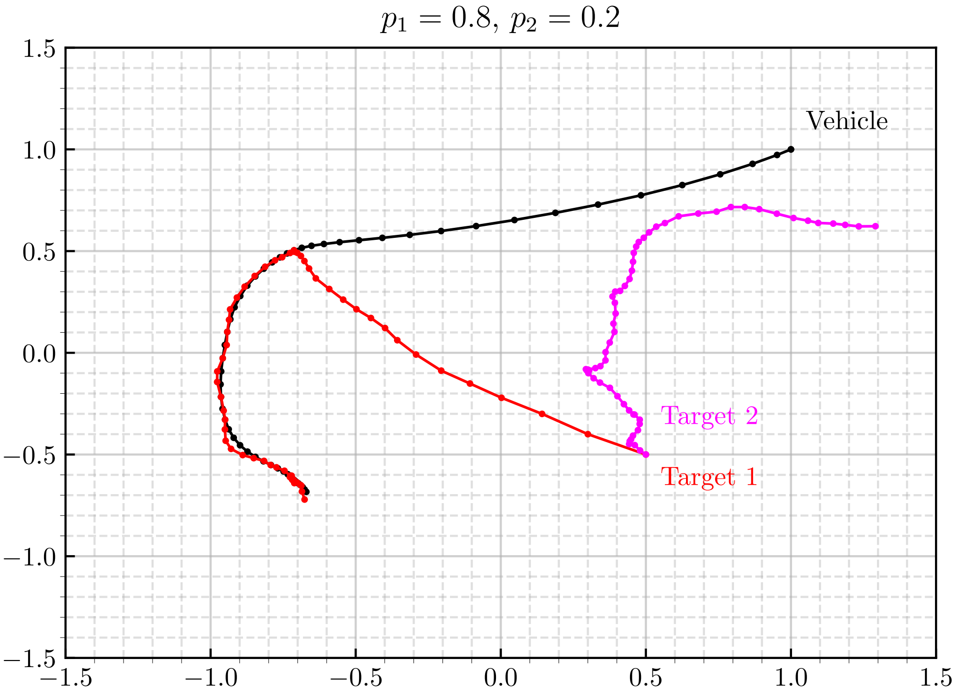
4 Task 4

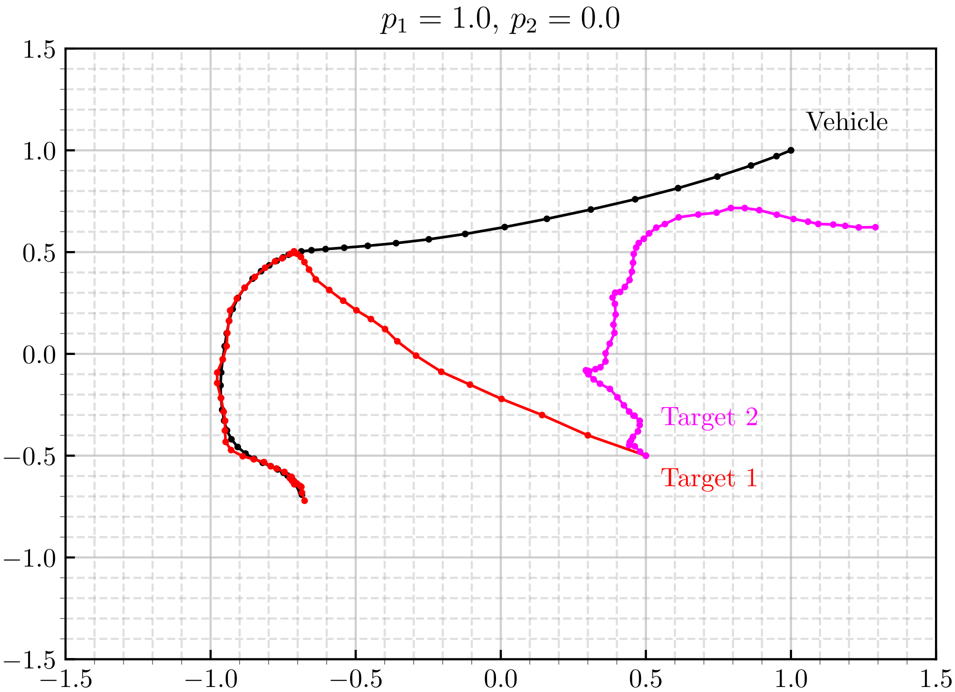

$p_1 = 0.0$, $p_2 = 1.0$
Vehicle
Target 2
Target 1
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


$p_1 = 0.2$, $p_2 = 0.8$
Vehicle
Target 2
Target 1
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


$p_1 = 0.4$, $p_2 = 0.6$
Vehicle
Target 2
Target 1


$p_1 = 0.6$, $p_2 = 0.4$
Vehicle
Target 2
Target 1





From the results, we can see that an increase in p_i yields an increase in proximity to the trajectory of the vehicle i ($i \in \{1, 2\}$) in the sense that, for each time instant, the corresponding position becomes less distant (in this case, according to l_∞ distance) to the position of the trajectory of vehicle i in the same instant. Moreover, we observe that this overall proximity is more substantial when it is relative to the trajectory of the vehicle associated with the highest prior probability.

Noting that the cost function can be written as:

$$f(x,u) = p_1 \mathrm{TE}_1(x,u) + p_2 \mathrm{TE}_2(x,u) + \lambda \mathrm{CE}(x,u)$$

we see that, for a fixed lambda, if p_i increases, the parcel $p_i \mathrm{TE}_i(x,u)$ gains more relative importance in the cost function. As such, bigger decreases in the cost function can be more easily attained by lowering $\mathrm{TE}_i(x,u)$. Consequently, the solution (x^*, u^*) yields a lower value of $\mathrm{TE}_i(x^*, u^*)$, and so the resulting trajectory will tend to become overall closer to the one of target i. In particular, if $p_i > p_j$, then $\mathrm{TE}_i(x^*, u^*) >= \mathrm{TE}_j(x^*, u^*)$ and the vehicles trajectory will be more similar to that of vehicle i.

5 Task 5

Given the current problem formulation, we have that the cost function can be written as:

$$f(x_1, u_1, x_2, u_2) = f_1(x_1, u_1) + f_2(x_2, u_2)$$

where

$$f_k(x_k, u_k) = p_k \left(\sum_{t=1}^{T} ||Ex_k(t) - q_k(t)||_\infty + \lambda \sum_{t=1}^{T-1} ||u_k(t)||_2^2 \right)$$

with $k \in \{1, 2\}$. Given that the set of constraints can be broken into two sets, each one pertaining to disjoint sets of variables (namely $\{x_1, u_1\}$ and $\{x_2, u_2\}$) and that $p_k \geq 0$ we conclude that (x_1, u_1) and (x_2, u_2) can be obtained by solving two independent optimization problems, each one of the form:

$$\begin{aligned} \underset{u}{\text{minimize}} \quad & \sum_{t=1}^{T} ||Ex_k(t) - q_k(t)||_\infty + \lambda \sum_{t=1}^{T-1} ||u_k(t)||_2^2 \\ \text{s.t.} \quad & x_k(1) = x_{\text{initial}} \\ & x_k(t+1) = Ax_k(t) + Bu_k(t) \end{aligned}$$

where $k \in \{1, 2\}$. With that being said, we would be obtaining two independent solutions for the problem discussed in Tasks 1, 2 and 3. In turn, these two trajectories would only coincide at the initial instant but would tendentially become closer to the target associated with the corresponding problem. Hence, this formulation is completely agnostic to the fact that, until the target is made known at $t = 35$, the trajectories must be exactly the same.

6 Task 6

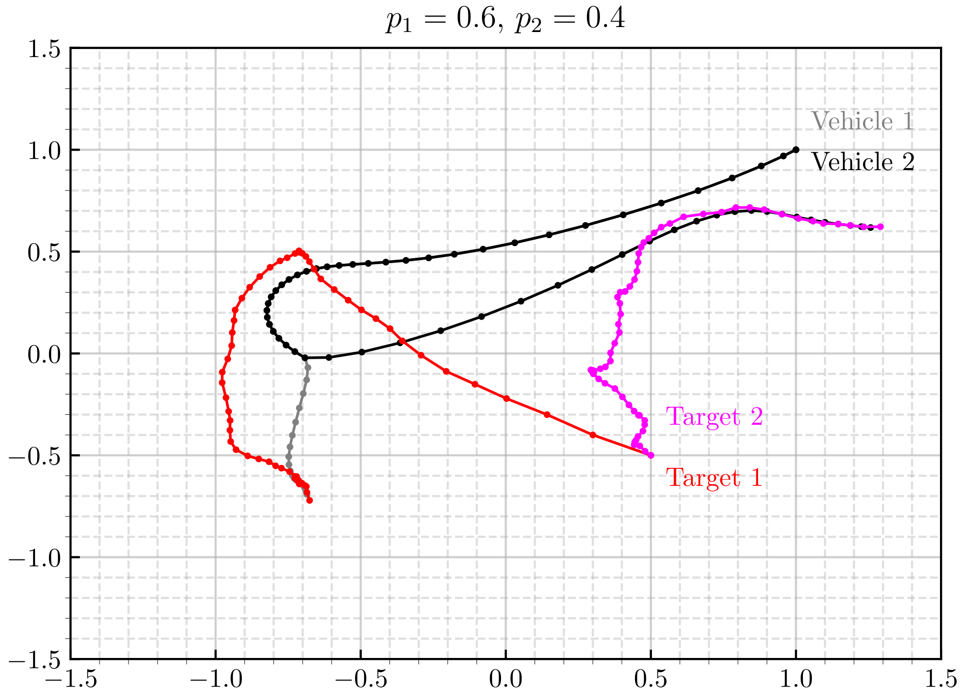
The following constraint solves the questions raised in the previous task:

$$u_1(t) = u_2(t), \quad 1 \leq t \leq 34 \tag{3}$$

Note that x_1 and x_2 become equal (for $1 \leq t \leq 34$), given these constraints together with (**??**):

$$x_1(t+1) = Ax_1(t) + Bu_1(t), \quad 1 \leq t \leq T-1$$
$$x_2(t+1) = Ax_2(t) + Bu_2(t), \quad 1 \leq t \leq T-1$$

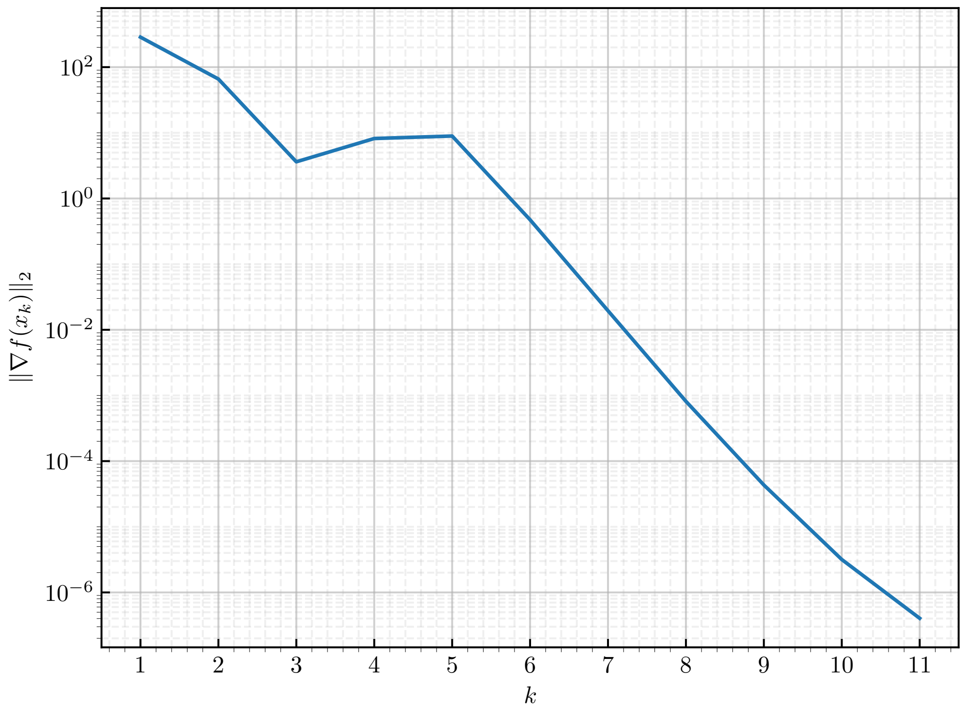
7 Task 7

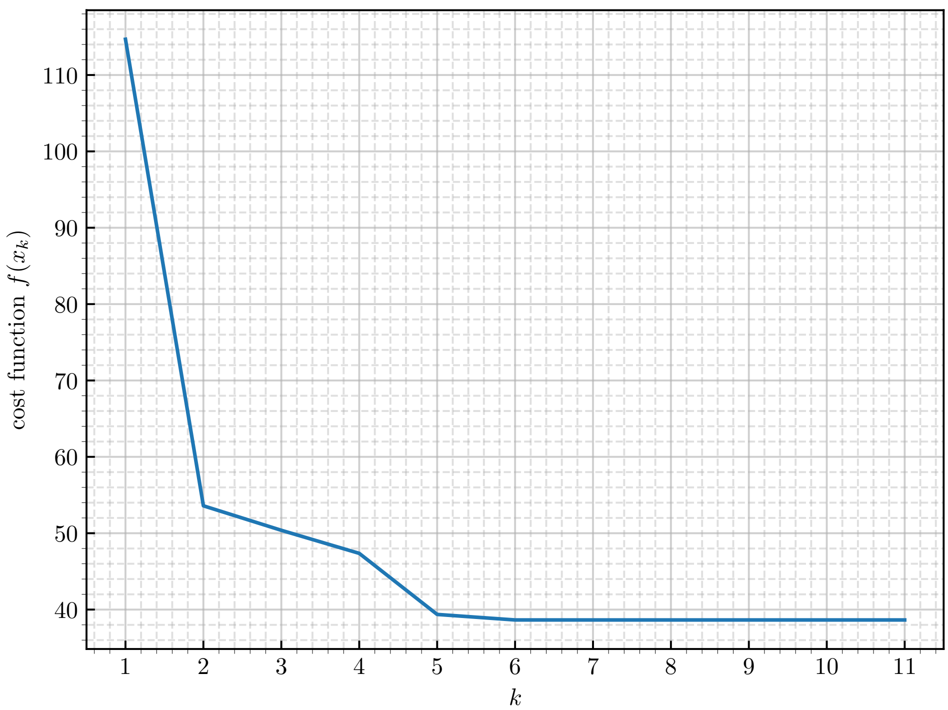


8 Task 8

Final estimates of p and v:

$$p^* = \begin{bmatrix} 1.77002952 \\ -0.92169453 \end{bmatrix} \qquad v^* = \begin{bmatrix} -0.95187589 \\ 1.49529891 \end{bmatrix}$$


10^2
10^0
10^{-2}
10^{-4}
10^{-6}
$\|\nabla f(x_k)\|_2$
1 2 3 4 5 6 7 8 9 10 11
k


110
100
90
80
70
60
50
40
cost function $f(x_k)$
1 2 3 4 5 6 7 8 9 10 11
k

Derivation of the gradients

In order to apply the Levenberg-Marquardt method, we must write the cost function as a sum of squares of some other functions. To do that, we note that:

$$f(p,v) = \sum_{t\in\mathcal{T}} \sum_{i=1}^{2} (g_{t,i}(p,v))^2$$

where $g_{t,i}(p,v) = \|(p+tv) - s_i\|_2 - r_i(t)$. Since in iteration $(k+1)$ we solve:

$$\underset{p,v}{\text{minimize}} \left\| A \begin{bmatrix} p \\ v \end{bmatrix} - b \right\|^2$$

where

$$A = \begin{bmatrix} \nabla_{(p,v)} g_{t_1,1}(p_k,v_k)^T \\ \nabla_{(p,v)} g_{t_1,2}(p_k,v_k)^T \\ \cdots \\ \nabla_{(p,v)} g_{t_{|\mathcal{T}|},1}(p_k,v_k)^T \\ \nabla_{(p,v)} g_{t_{|\mathcal{T}|},2}(p_k,v_k)^T \\ \sqrt{\lambda_k} I_4 \end{bmatrix} \qquad b = \begin{bmatrix} \nabla_{(p,v)} g_{t_1,1}(p_k,v_k)^T \begin{bmatrix} p_k, v_k \end{bmatrix}^T - g_{t_1,1}(p_k,v_k) \\ \nabla_{(p,v)} g_{t_1,2}(p_k,v_k)^T \begin{bmatrix} p_k, v_k \end{bmatrix}^T - g_{t_1,2}(p_k,v_k) \\ \cdots \\ \nabla_{(p,v)} g_{t_{|\mathcal{T}|},1}(p_k,v_k)^T \begin{bmatrix} p_k, v_k \end{bmatrix}^T - g_{t_{|\mathcal{T}|},1}(p_k,v_k) \\ \nabla_{(p,v)} g_{t_{|\mathcal{T}|},2}(p_k,v_k)^T \begin{bmatrix} p_k, v_k \end{bmatrix}^T - g_{t_{|\mathcal{T}|},2}(p_k,v_k) \\ \sqrt{\lambda_k} [p_k, v_k]^T \end{bmatrix}$$

we must compute the gradients $\nabla_{(p,v)} g_{t,i}(p,v)$, for $t \in \mathcal{T}$ and $i \in 1,2$.
For that purpose, we write $g_{t,i}$ as a composition of functions:

$$g_{t,i}(p,v) = h_{t,i}^{(3)}(h_{t,i}^{(2)}(h_{t,i}^{(1)}(p,v)))$$

where

$$\begin{aligned} h_{t,i}^{(1)}(p,v) &= (p+tv) - s_i, \quad p,v \in \mathbb{R}^2 \\ h_{t,i}^{(2)}(z) &= \|z\|_2, \quad z \in \mathbb{R}^2 \\ h_{t,i}^{(3)}(u) &= u - r_i(t), \quad u \in \mathbb{R} \end{aligned}$$

and proceed to apply the chain rule:

$$\begin{aligned} \nabla_{(p,v)} g_{t,i}(p,v) &= \nabla_{(p,v)} h_{t,i}^{(1)}(p,v)\, \nabla_z h_{t,i}^{(2)}(h_{t,i}^{(1)}(p,v))\, \nabla_u h_{t,i}^{(3)}(h_{t,i}^{(2)}(h_{t,i}^{(1)}(p,v))) \\ &= \nabla_{(p,v)}((p+tv) - s_i)\, \nabla_z(\|z\|_2)\Big|_{z=(p+tv)-s_i} \nabla_u(u - r_i(t))\Big|_{u=\|(p+tv)-s_i\|_2} \\ &= \begin{bmatrix} I_2 \\ tI_2 \end{bmatrix} \left(\frac{z}{\|z\|_2} \right)\Bigg|_{z=(p+tv)-s_i} \cdot 1 \\ &= \begin{bmatrix} I_2 \\ tI_2 \end{bmatrix} \frac{(p+tv) - s_i}{\|(p+tv) - s_i\|_2} \end{aligned}$$

where I_2 denotes the 2×2 identity matrix. Furthermore, we also have to calculate $\nabla_{(p,v)} f(p,v)$ to evaluate the stopping condition $\|\nabla_{(p,v)} f(p,v)\| < \epsilon$:

$$
\begin{aligned}
\nabla_{(p,v)} f(p,v) &= \sum_{t\in\mathcal{T}} \sum_{i=1}^{2} \nabla_{(p,v)} (g_{t,i}(p,v))^2 \\
&= \sum_{t\in\mathcal{T}} \sum_{i=1}^{2} 2 g_{t,i}(p,v) \nabla_{(p,v)} (g_{t,i}(p,v)) \\
&= \sum_{t\in\mathcal{T}} \sum_{i=1}^{2} 2(\|(p+tv)-s_i\|_2 - r_i(t)) \begin{bmatrix} I_2 \\ tI_2 \end{bmatrix} \frac{(p+tv)-s_i}{\|(p+tv)-s_i\|_2}
\end{aligned}
$$

Appendices

A Task 1 Code

```python
#%%
import cvxpy as cp
import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt
import matplotlib

# use tex fonts
plt.rcParams.update({
    "text.usetex": True,
    "font.family": "serif",
    "font.serif": "Computer Modern",
})

# Optimization problem constants
q1 = loadmat('../inputs/target_1.mat')["target"].T
T = q1.shape[0]
A = np.array([ [1, 0, 0.2, 0], [0, 1, 0, 0.2], [0, 0, 0.8, 0], [0, 0, 0, 0.8] ])
B = np.array([ [0, 0], [0, 0], [0.2, 0], [0, 0.2] ])
x_0 = np.array([1, 1, 0, 0])
E = np.array([ [1, 0, 0, 0], [0, 1, 0, 0] ])
lambs = [10, 5, 1, 0.5, 0.1, 0.05, 0.01, 0.005, 0.001]

# Optimization problem variables
x = cp.Variable((T, 4))
u = cp.Variable((T - 1, 2))

# Optimization problem constraints
TE = 0
CE = 0
constr = [x[0] == x_0]

for t in range(T - 1):
    TE += cp.norm(E @ x[t] - q1[t], 'inf')
    CE += cp.sum_squares(u[t])
    constr += [x[t + 1] == A @ x[t] + B @ u[t]]

TE += cp.norm(E @ x[T - 1] - q1[T - 1], "inf")

# Record values for TE and CE for each lambda parameter value
TEs = []
CEs = []

# Solve problem for each lambda value and plot obtained trajectory
for i in range(len(lambs)):
    objective = cp.Minimize(TE + lambs[i] * CE)
    prob = cp.Problem(objective, constr)
    result = prob.solve()
    TEs.append(TE.value)
```

```
    CEs.append(CE.value)

    # Magic image size line
    plt.figure(figsize=(46.82 * .5**(.5 * 6), 33.11 * .5**(.5 * 6)))

    plt.plot(x[:, 0].value, x[:, 1].value, label = "Vehicle", color = "black",
        marker = 'o',linewidth = 1, markersize = 1.5)
    plt.plot(q1[:, 0], q1[:, 1], label = "Target 1", color = "red", marker='o',
        linewidth = 1, markersize = 1.5)
    plt.xlim(-1.5,1.5)
    plt.ylim(-1.5,1.5)
    plt.minorticks_on()
    plt.grid(which = "major", linestyle = "-", alpha = 0.6)
    plt.grid(which = "minor", linestyle = "--", alpha = 0.4)
    plt.tick_params(which = "minor", width = 0)
    plt.tick_params(which = "major", direction = "in")
    plt.text(x_0[0] + .05, x_0[1] + .1, "Vehicle", {"color": "black"})
    plt.text(q1[0][0] + .05, q1[0][1] - .15, "Target", {"color": "red"})
    plt.title(f"$\lambda = {lambs[i]}$")
    plt.savefig(f"./output/ex_1_i={i + 1}.pdf")
    plt.cla()

plt.plot(TEs, CEs, 'ro', label = range(i, len(TEs) + 1), color = "blue",
    markersize = 2)
plt.axis([0, 45, 0, 600])
plt.xlabel("TE")
plt.ylabel("CE")

# fine tune where labels should be placed for each (TE, CE) pair
offsets = [[0.2, 10], # 1
           [0.2, 10], # 2
           [0.5, 10], # 3
           [0.5, 10], # 4
           [0.5, 2],  # 5
           [0.2, 10], # 6
           [0.5, 1],  # 7
           [0.5, 2],  # 8
           [0.8, -10]] # 9

for i in range(len(TEs)):
    plt.text(TEs[i] + offsets[i][0], CEs[i] + offsets[i][1], f"$\lambda_{i + 1}$
        ")

plt.minorticks_on()
plt.grid(which = "major", linestyle = "-", alpha = 0.6)
plt.grid(which = "minor", linestyle = "--", alpha = 0.4)
plt.tick_params(which = "minor", width = 0)
plt.tick_params(which = "major", direction = "in")
plt.savefig("./output/TEvsCE.pdf")

# %%
```

B Task 4 Code

```
#%%
import cvxpy as cp
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.io import loadmat

plt.rcParams.update({
    "text.usetex": True,
    "font.family": "serif",
    "font.serif": "Computer Modern",
})

# Read input
input_t1 = loadmat('../inputs/target_1.mat')
t1 = [[element for element in upperElement] for upperElement in input_t1['target
    ']]
t1_data = list(zip(t1[0], t1[1]))
columns = ['x', 'y']
df1 = pd.DataFrame(t1_data, columns=columns)

input_t2 = loadmat('../inputs/target_2.mat')
t2 = [[element for element in upperElement] for upperElement in input_t2['target
    ']]
t2_data = list(zip(t2[0], t2[1]))
df2 = pd.DataFrame(t2_data, columns=columns)

# Problem data.
i = 6-1   # choose (i=0 -> instance=1)
T = 60
x_init = np.array([1, 1, 0, 0])
A = np.array([[1, 0, 0.2, 0], [0, 1, 0, 0.2], [0, 0, 0.8, 0], [0, 0, 0, 0.8]])
B = np.array([[0, 0], [0, 0], [0.2, 0], [0, 0.2]])
E = np.array([[1, 0, 0, 0], [0, 1, 0, 0]])

p1 = np.array([0, 0.2, 0.4, 0.6, 0.8, 1])
p2 = np.array([1, 0.8, 0.6, 0.4, 0.2, 0])
q1 = np.array(t1_data)
q2 = np.array(t2_data)

# Construct the problem.
X = cp.Variable((T,4)) # Matrix Tx4
U = cp.Variable((T-1,2))

te1 = 0
te2 = 0
ce = 0
constraints = [X[0] == x_init]
for t in range(T-1):
    te1 += cp.norm(E @ X[t] - q1[t], "inf")
    te2 += cp.norm(E @ X[t] - q2[t], "inf")
    ce += cp.sum_squares(U[t])
    constraints += [X[t+1] == A @ X[t] + B @ U[t]]

te1 += cp.norm(E @ X[T-1] - q1[T-1], "inf")
```

```
te2 += cp.norm(E @ X[T-1] - q2[T-1], "inf")


# Plots

# Solve problem for each p1 and p2 values and plot obtained trajectory
for i in range(len(p1)):
    objective = cp.Minimize(p1[i]*te1 + p2[i]*te2 + 0.5*ce)
    prob = cp.Problem(objective, constraints)
    result = prob.solve()

    # Magic image size line
    plt.figure(figsize=(46.82 * .5**(.5 * 6), 33.11 * .5**(.5 * 6)))

    plt.plot(X[:, 0].value, X[:, 1].value, label = "Vehicle", color = "black",
        marker = 'o',linewidth = 1, markersize = 1.5)
    plt.plot(q1[:, 0], q1[:, 1], label = "Target 1", color = "red", marker='o',
        linewidth = 1, markersize = 1.5)
    plt.plot(q2[:, 0], q2[:, 1], label = "Target 2", color = "magenta", marker='
        o', linewidth = 1, markersize = 1.5)
    plt.xlim(-1.5,1.5)
    plt.ylim(-1.5,1.5)
    plt.minorticks_on()
    plt.grid(which = "major", linestyle = "-", alpha = 0.6)
    plt.grid(which = "minor", linestyle = "--", alpha = 0.4)
    plt.tick_params(which = "minor", width = 0)
    plt.tick_params(which = "major", direction = "in")
    plt.text(x_init[0] + .05, x_init[1] + .1, "Vehicle", {"color": "black"})
    plt.text(q1[0][0] + .05, q1[0][1] - .15, "Target 1", {"color": "red"})
    plt.text(q2[0][0] + .05, q2[0][1] + .15, "Target 2", {"color": "magenta"})
    plt.title(f"$p_1 = {p1[i]}$, $p_2 = {p2[i]}$")
    plt.savefig(f"./output/ex_4_i={i + 1}.pdf")
    plt.cla()

# %%
```

C Task 7 Code

```
#%%
import cvxpy as cp
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.io import loadmat

plt.rcParams.update({
    "text.usetex": True,
    "font.family": "serif",
    "font.serif": "Computer Modern",
})

# Read input
input_t1 = loadmat('../inputs/target_1.mat')
t1 = [[element for element in upperElement] for upperElement in input_t1['target
```

```
    ']]
t1_data = list(zip(t1[0], t1[1]))
columns = ['x', 'y']
df1 = pd.DataFrame(t1_data, columns=columns)

input_t2 = loadmat('../inputs/target_2.mat')
t2 = [[element for element in upperElement] for upperElement in input_t2['target
    ']]
t2_data = list(zip(t2[0], t2[1]))
df2 = pd.DataFrame(t2_data, columns=columns)

# Problem data.
i = 4-1    # choose (i=0 -> instance=1)
T = 60
x_init = np.array([1, 1, 0, 0])
A = np.array([[1, 0, 0.2, 0], [0, 1, 0, 0.2], [0, 0, 0.8, 0], [0, 0, 0, 0.8]])
B = np.array([[0, 0], [0, 0], [0.2, 0], [0, 0.2]])
E = np.array([[1, 0, 0, 0], [0, 1, 0, 0]])

p1 = np.array([0, 0.2, 0.4, 0.6, 0.8, 1])
p2 = np.array([1, 0.8, 0.6, 0.4, 0.2, 0])
q1 = np.array(t1_data)
q2 = np.array(t2_data)

# Construct the problem.
X1 = cp.Variable((T,4)) # Matrix Tx4
X2 = cp.Variable((T,4))
U1 = cp.Variable((T-1,2))
U2 = cp.Variable((T-1,2))

te1 = 0
te2 = 0
ce1 = 0
ce2 = 0

constraints = [X1[0] == x_init, X2[0] == x_init]
for t in range(34):
    constraints += [U1[t] == U2[t]]

for t in range(T-1):
    te1 += cp.norm(E @ X1[t] - q1[t], "inf")
    te2 += cp.norm(E @ X2[t] - q2[t], "inf")
    ce1 += cp.sum_squares(U1[t])
    ce2 += cp.sum_squares(U2[t])
    constraints += [X1[t+1] == A @ X1[t] + B @ U1[t]]
    constraints += [X2[t+1] == A @ X2[t] + B @ U2[t]]

te1 += cp.norm(E @ X1[T-1] - q1[T-1], "inf")
te2 += cp.norm(E @ X2[T-1] - q2[T-1], "inf")

# Solve problem for each p1 and p2 values and plot obtained trajectory
objective = cp.Minimize(p1[i]*(te1 + 0.5*ce1) + p2[i]*(te2 + 0.5*ce2))
prob = cp.Problem(objective, constraints)
result = prob.solve()
```

```
# Magic image size line
plt.figure(figsize=(46.82 * .5**(.5 * 6), 33.11 * .5**(.5 * 6)))

plt.plot(X1[:, 0].value, X1[:, 1].value, label = "Vehicle 1", color = "gray",
    marker = 'o',linewidth = 1, markersize = 1.5)
plt.plot(X2[:, 0].value, X2[:, 1].value, label = "Vehicle 2", color = "black",
    marker = 'o',linewidth = 1, markersize = 1.5)
plt.plot(q1[:, 0], q1[:, 1], label = "Target 1", color = "red", marker='o',
    linewidth = 1, markersize = 1.5)
plt.plot(q2[:, 0], q2[:, 1], label = "Target 2", color = "magenta", marker='o',
    linewidth = 1, markersize = 1.5)
plt.xlim(-1.5,1.5)
plt.ylim(-1.5,1.5)
plt.minorticks_on()
plt.grid(which = "major", linestyle = "-", alpha = 0.6)
plt.grid(which = "minor", linestyle = "--", alpha = 0.4)
plt.tick_params(which = "minor", width = 0)
plt.tick_params(which = "major", direction = "in")
plt.text(x_init[0] + .05, x_init[1] + .1, "Vehicle 1", {"color": "gray"})
plt.text(x_init[0] + .05, x_init[1] - .1, "Vehicle 2", {"color": "black"})
plt.text(q1[0][0] + .05, q1[0][1] - .15, "Target 1", {"color": "red"})
plt.text(q2[0][0] + .05, q2[0][1] + .15, "Target 2", {"color": "magenta"})
plt.title(f"$p_1 = {p1[3]}$, $p_2 = {p2[3]}$")
plt.savefig(f"./output/ex_7.pdf")
plt.cla()

# %%
```

D Task 8 Code

```
import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
import matplotlib

# column vector of instants of measurements
t_i = 0
t_f = 5
no_divisions = 10
T = np.array([np.linspace(t_i, t_f, (no_divisions) * (t_f - t_i) + 1)]).T
print(T)

# sensor positions
s = np.array([[0, -1],
              [1, 5]])

# Levenberg-Marquardt algorithm parameters
# initial guess
p = np.array([-1, 0])
v = np.array([0, 1])

# trust parameter lambda
lamb = 1
```

```
# tolerance parameter
epsilon = 1e-6

def get_displacement_from_sensor(p, v, t, sensor_index):
    """ Given the position and velocity of the target, get estimate of
        distance to target number sensor_index

    Args:
        p: 2D NumPy array.
        v: 2D NumPy array.
        t: time instance (scalar)
        sensor_index: 1 or 2

    Returns:
        A scalar representing the displacement from the sensor
    """
    return (p + t * v) - s[sensor_index - 1]

def compute_gradient_g_t_i(p, v, t, sensor_index):
    """ Given the initial position and velocity of the target, get the gradient
       of the
        function g_t_i such that f = sum(g_t_1 ** 2 + g_t_2 ** 2)
    Args:
        p: 2D NumPy array representing the current estimate of target's initial
            position
        v: 2D NumPy array representing the current estimate of target's velocity
        t: time instant (scalar)
        sensor_index: 1 or 2

    Returns:
        A 1D NumPy array representing the gradient
    """
    displacement = get_displacement_from_sensor(p, v, t, sensor_index)
    identities = np.concatenate((np.identity(2), t * np.identity(2)), axis = 0)

    return identities @ (displacement / np.linalg.norm(displacement))

def compute_g_t_i(p, v, t, measurement, sensor_index):
    """ Given the initial position and velocity of the target, get the value of
       the
        function g_t_i such that f = sum(g_t_1 ** 2 + g_t_2 ** 2)
    Args:
        p: 2D NumPy array representing the current estimate of target's initial
            position
        v: 2D NumPy array representing the current estimate of target's velocity
        t: time instant (scalar)
        sensor_index: 1 or 2

    Returns:
        A 1D NumPy array representing the gradient
    """
    displacement = get_displacement_from_sensor(p, v, t, sensor_index)
    return np.linalg.norm(displacement) - measurement
```

```
def compute_gradient_f_t(p, v, t, measurement_1, measurement_2):
    """ Given the initial position and velocity of the target, get the gradient
       of the
        function representing the sum of the squared errors of the estimated
            distances
        to each sensor relative to the actual measured distances by each sensor
            (i.e., nabla(f_t(p_k, v_k)))
    Args:
        p: 2D NumPy array representing the current estimate of target's initial
            position
        v: 2D NumPy array representing the current estimate of target's velocity
        t: time instant (scalar)
        measurement_1: distance measured by sensor 1
        measurement_2: distance measured by sensor 2

    Returns:
        A 1D NumPy array representing the gradient
    """

    return 2 * compute_g_t_i(p, v, t, measurement_1, 1) * compute_gradient_g_t_i
       (p, v, t, 1) + \
           2 * compute_g_t_i(p, v, t, measurement_2, 2) * compute_gradient_g_t_i
               (p, v, t, 2)

def compute_f_t(p, v, t, measurement_1, measurement_2):
    """ Given the position and velocity of the target, get the sum of the
        squared errors of the estimated distances to each sensor relative
        to the actual measured distances by each sensor (i.e., f_t)
    Args:
        p: 2D NumPy array representing the current estimate of target's initial
            position
        v: 2D NumPy array representing the current estimate of target's velocity
        t: time instance (scalar)
        measurement_1: distance measured by sensor 1
        measurement_2: distance measured by sensor 2

    Returns:
        A scalar representing
    """
    return compute_g_t_i(p, v, t, measurement_1, 1) ** 2 + compute_g_t_i(p, v, t
       , measurement_2, 2) ** 2

def compute_gradient_f(p, v, r1, r2):
    """ Given the position and velocity of the target, get the gradient of the
        function representing the sum of the squared errors of the estimated
            distances
        to each sensor relative to the actual measured distances by each sensor
            at all instants
        (i.e., sum(nabla(f_t(p_k, v_k))))
     Args:
        p: 2D NumPy array representing the current estimate of target's initial
            position
        v: 2D NumPy array representing the current estimate of target's velocity
        r1: 1D NumPy array representing measurements of sensor 1
```

```
        r2: 1D NumPy array representing measurements of sensor 2

    Returns:
        A 1D NumPy array repreesenting grad(f)
    """
    # matrix where the entries of row i are (t_i, r1[t_i], r2[t_i])
    t_m_triples = np.hstack((T, r1.reshape(r1.size, 1), r2.reshape(r2.size, 1)))

    return np.apply_along_axis(lambda entry: compute_gradient_f_t(p, v, entry
       [0], entry[1], entry[2]),
                               arr = t_m_triples, axis = 1).sum(axis = 0)

def compute_f(p, v, r1, r2):
    """ Given the position and velocity of the target, get the sum of the
        squared errors of the estimated distances to each sensor relative
        to the actual measured distances by each sensor at all instants (i.e.,
           sum(f_t))
    Args:
        p: 2D NumPy array representing the current estimate of target's initial
           position
        v: 2D NumPy array representing the current estimate of target's velocity
        r1: 1D NumPy array representing measurements of sensor 1
        r2: 1D NumPy array representing measurements of sensor 2

    Returns:
        A scalar representing f
    """
    # matrix where the entries of row i are (t_i, r1[t_i], r2[t_i])
    t_m_triples = np.hstack((T, r1.reshape(r1.size, 1), r2.reshape(r2.size, 1)))

    return np.apply_along_axis(lambda entry: compute_f_t(p, v, entry[0], entry
       [1], entry[2]),
                               arr = t_m_triples, axis = 1).sum(axis = 0)

def get_stacked_g_gradients(p, v):
    """ Get a matrix where the (2k + 1)-th and (2k + 2)-th row have nabla(g_k_1)
         transpose
        and nabla(g_k_2) transpose, respectively, for k = 0, ..., T
    Args:
        p: 2D NumPy array representing the current estimate of target's initial
           position
        v: 2D NumPy array representing the current estimate of target's velocity

    Returns:
        The matrix with the stacked transposed gradients
    """

    # matrix where the first and the second entries of row i have (t_i, 1) and (
       t_i, 2), respectively
    t_i_pairs = np.array([[t_i, i] for t_i in T.flatten() for i in range(1, 3)])

    return np.apply_along_axis(lambda entry: compute_gradient_g_t_i(p, v, entry
       [0], int(entry[1])),
                                           arr = t_i_pairs, axis = 1)
```

```
def get_stacked_g_values(p, v, measurements_1, measurements_2):
    """ Get a column vector where the (2k + 1)-th and (2k + 2)-th entry have
       g_k_1
        and  g_k_2, respectively, for k = 0, ..., T
    Args:
        p: 2D NumPy array representing the current estimate of target's initial
            position
        v: 2D NumPy array representing the current estimate of target's velocity

    Returns:
        The matrix with the stacked transposed gradients
    """

    # matrix where the first and the second entries of row i have (t_i, 1) and (
       t_i, 2), respectively
    t_i_pairs = np.array([[t_i, i] for t_i in T.flatten() for i in range(1, 3)])

    # array where measurements from the first and second sensors are intertwined
    measurements = np.empty((measurements_1.size + measurements_2.size), dtype =
         measurements_1.dtype)
    measurements[0::2] = measurements_1
    measurements[1::2] = measurements_2

     # matrix where the first and the second entries of row i have (t_i, 1,
         measurement_1)
     # and (t_i, 2, measurement_2), respectively
    t_i_m_triples = np.hstack((t_i_pairs, measurements.reshape(measurements.size
       , 1)))

    return np.apply_along_axis(lambda entry: compute_g_t_i(p, v, entry[0], entry
       [2], int(entry[1])),
                                                      arr = t_i_m_triples, axis = 1)

# Read the data
measurements = loadmat('../inputs/measurements.mat')
r1 = measurements["r1"].flatten()
r2 = measurements["r2"].flatten()

function_values = [compute_f(p, v, r1, r2)]
gradient_norms = []

# Run Levenberg-Marquardt Algorithm for non-linear Least-Squares
while (True):

    # check stopping condition
    curr_grad = compute_gradient_f(p, v, r1, r2)
    gradient_norms.append(np.linalg.norm(curr_grad))

    if np.linalg.norm(curr_grad) < epsilon:
        break

    # Solve least squares problem
    grads_g = get_stacked_g_gradients(p, v)
    g_values = get_stacked_g_values(p, v, r1, r2)
    x = np.concatenate((p, v), axis = 0)
```

```python
    while (True):

        # Construct A matrix and b vector of the corresponding least-squares
          problem
        diagonal_matrix = np.sqrt(lamb) * np.identity(4)
        A = np.concatenate((grads_g, diagonal_matrix), axis = 0)
        b = grads_g @ x - g_values
        b = np.concatenate((b, diagonal_matrix @ x), axis = 0)

        # Solve least squares problem
        solution = np.linalg.lstsq(A, b, rcond = None)[0]
        tentative_p = solution[ : 2]
        tentative_v = solution[2 : ]

        function_values.append(compute_f(tentative_p, tentative_v, r1, r2))

        # check if the step was valid
        if function_values[-1] < function_values[-2]:
            p = tentative_p
            v = tentative_v
            lamb = lamb * 0.7
            break
        else:
            lamb = lamb * 2
            gradient_norms.append(np.linalg.norm(curr_grad))

print(f"p = {p}  v = {v}")
print(f"f = {compute_f(p, v, r1, r2)}")

plt.rcParams['text.usetex'] = True
plt.rc('font', family='serif')

# Plot cost function values across iterations
k_range = [i for i in range(1, len(function_values) + 1)]

fig = plt.figure()
ax = fig.add_subplot(111)

plt.plot(k_range, function_values)

ax.set_xlabel('$k$')
ax.set_ylabel('cost function $f(x_k)$')

ax.set_xticks(k_range)
ax.minorticks_on()
ax.grid(which = "major", linestyle = "-", alpha = 0.6)
ax.grid(which = "minor", linestyle = "--", alpha = 0.4)
ax.tick_params(which = "minor", width = 0)
ax.tick_params(which = "major", direction = "in")

plt.savefig("./output/cost_function_values.pdf")

# Plot gradient norm values across iterations
fig = plt.figure()
```

```python
ax = fig.add_subplot(111)

plt.plot(k_range, gradient_norms)

ax.set_xlabel('$k$')
ax.set_ylabel('$\|\\nabla f(x_k)\|_{2}$')

ax.set_xticks(k_range)

# Only show even powers of 10
log_range = [10 ** i for i in range(int(np.floor(np.log10(np.min(gradient_norms)))),
                                    int(np.ceil(np.log10(np.max(gradient_norms)))) if i % 2 == 0]

ax.set_yscale('log')
ax.minorticks_on()
ax.set_yticks(log_range)
ax.grid(which = "major", linestyle = "-", alpha = 0.6)
ax.grid(which = "minor", linestyle = "--", alpha = 0.2)
ax.tick_params(which = "minor", width = 0)
ax.tick_params(which = "major", direction = "in")

locmin = mticker.LogLocator(base=10, subs=np.arange(0.1, 1, 0.1),numticks=10)
ax.yaxis.set_minor_locator(locmin)
ax.yaxis.set_minor_formatter(mticker.NullFormatter())

plt.savefig("./output/gradient_norms.pdf")
```