Homework 03

AA203: Optimal and learning-based Control

Ricardo Luiz Moreira Paschoeto

rp304154@stanford.edu

Problem 1: HJ Reachability

a) Method PlanarQuadrotor.optimal control,

Python code:

```python
    def optimal_control(self, state, grad_value):
        """Computes the optimal control realized by the HJ PDE Hamiltonian.

        Args:
            state: An unbatched (!) state vector, an array of shape `(4,)` containing `[y, v_y, phi, omega]`.
            grad_value: An array of shape `(4,)` containing the gradient of the value function at `state`.

        Returns:
            A vector of optimal controls, an array of shape `(2,)` containing `[T_1, T_2]`, that minimizes
            `grad_value @ self.dynamics(state, control)`.
        """
        # PART (a): WRITE YOUR CODE BELOW ################################################
        # You may find `jnp.where` to be useful; see corresponding numpy docstring:
        # https://numpy.org/doc/stable/reference/generated/numpy.where.html


        # If you find a way to use the jnp.where please teach me!
        result = []
        for T1 in (self.min_thrust_per_prop, self.max_thrust_per_prop):
            for T2 in (self.min_thrust_per_prop, self.max_thrust_per_prop):
                H_ = grad_value @ self.dynamics(state, jnp.array([T1, T2]))
                result.append(jnp.array([T1,T2,H_]))

        aresult = jnp.asarray(result)
        H = aresult[jnp.argmin(aresult[:,-1]), :]

        control = H[0:2]
        return control
        ################################################################
################
```

b) Function target set,

Python code:

```
@jax.jit
def target_set(state):
    """A real-valued function such that the zero-
sublevel set is the target set.

    Args:
        state: An unbatched (!) state vector, an array of shape `(4,)` co
ntaining `[y, v_y, phi, omega]`.

    Returns:
        A scalar, nonnegative iff the state is in the target set.
    """
    # PART (b): WRITE YOUR CODE BELOW ###################################
############
    target_min = jnp.array([3., -1., -np.pi/12, -1.])
    target_max = jnp.array([7.,  1.,  np.pi/12,  1.])

    ax = target_min - state
    bx = state - target_max

    hx = 5.*jnp.maximum(ax, bx)

    return jnp.max(hx)
    ####################################################################
############
```

c) Function envelope set,

Python code:

```
@jax.jit
def envelope_set(state):
    """A real-valued function such that the zero-
sublevel set is the operational envelope.

    Args:
        state: An unbatched (!) state vector, an array of shape `(4,)` co
ntaining `[y, v_y, phi, omega]`.

    Returns:
        A scalar, nonnegative iff the state is in the operational envelop
e.
    """
    # PART (c): WRITE YOUR CODE BELOW ###################################
############
    envelope_min = jnp.array([1., -6., -np.inf, -8.])
    envelope_max = jnp.array([9.,  6.,  np.inf,  8.])

    ax = envelope_min - state
    bx = state - envelope_max

    ex = jnp.maximum(ax, bx)

    return jnp.max(ex)
    ####################################################################
############
```

d) 3D plot of the zero isosurface,

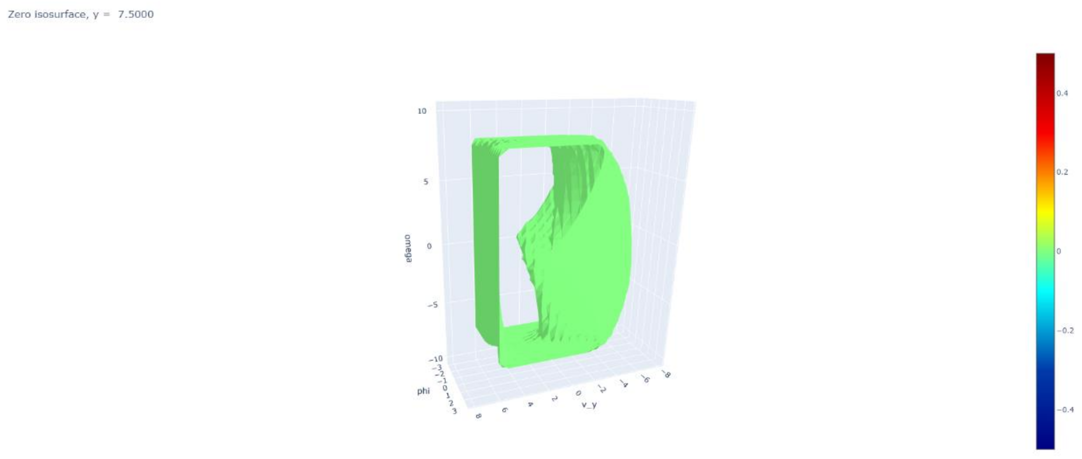

Figure 1- 3D isosurface

From the 3D isosurface, the bumps could be interpreted as when the states are outside of Envelope set, *e(x) > 0 and* $V(x,t) > 0$. The ridge where touch the plane at v_y = -6 is the state *x* bounded by operational envelope. Other ridges come from minimization $h(x(\tau))$regarding the t ∈[0, -5] and action *u,* where $V(x,t) \leq 0$.

e) Pros/cons of this approach,

- Cons
    - Regarding computational resources the time and memory, the computational complexity with respect to the numbers of states and discrete grid size is exponential. MPC is computationally more attractive.
    - Applied to small or simplified systems, MPC could be applied to more complex systems.

- Pros
    - Compatibility with nonlinear system dynamics, MPC for nonlinear systems requiring another approaching (e.g , nonlinear MPC).
    - Global Optimality guarantee by *h(x)* function, MPC gives a local optimality (finite horizon optimization).
    - Formal treatment of bounded disturbances - could be formulate inside the Value-function. MPC need to be included in the constraint set to be computed in receding horizon optimization-problem.

Problem 2: MPC Feasibility

a) Receding horizon control strategy – CVXPY.

Python Code:

```python
def recent_horizon(A,B, Q, R, P, x0, N, R2, uLB=-1, uUB=1, xLB0=-
10, xUB0=10, xLB1=-10, xUB1=10):
    n = Q.shape[0]
    m = R.shape[0]
    X = {}
    U = {}
    u = []
    x = []

    x_list_pred = []
    cost_terms = []
    constraints = []
    list_of_costs = []
    status = ""
    T = 20
    x_0 = x0
    for t in range(T):
        for k in range(N):

            X[k] = cvx.Variable(n)
            U[k] = cvx.Variable(m)

            cost_terms.append(cvx.quad_form(X[k], Q))
            cost_terms.append(cvx.quad_form(U[k], R))

            constraints.append(U[k] <= uUB)
            constraints.append(U[k] >= uLB)

            constraints.append(X[k][0] <= xUB0)
            constraints.append(X[k][0] >= xLB0)

            constraints.append(X[k][1] <= xUB1)
            constraints.append(X[k][1] >= xLB1)

            if k == 0:
                constraints.append(X[k] == x_0)

            if k > 0:
                constraints.append(A @ X[k - 1] + B @ U[k - 1] == X[k])

        X[k+1] = cvx.Variable(n)
        if not R2:
            constraints.append(X[k + 1] == np.zeros(2))
        else:
            constraints.append(A @ X[k] + B @ U[k] == X[k + 1])

        cost_terms.append(cvx.quad_form(X[k + 1], P))

        obj = cvx.Minimize(cvx.sum(cost_terms))
        problem = cvx.Problem(obj, constraints)
        problem.solve()

        status = problem.status
        if status in ["infeasible", "unbounded"]:
            break
        else:
            for k in range(N):
                u.append(U[k].value)
                x.append(X[k].value)

            list_of_costs.append(cvx.sum(cost_terms))
            x.append(X[k+1].value)
            x_0 = A @ X[0].value + B @ U[0].value

    return np.asarray(x), np.asarray(x_list_pred), list_of_costs
```

b) Results

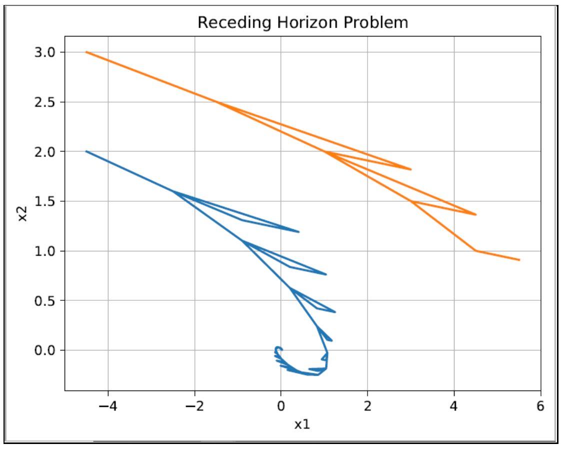

Figure 2 - x_0=[-4.5,2](blue) and x_0=[-4.5&3](orange).

Python Code for c-f:

- Riccati Equation:

```
def compute_riccati(A, B, Q, R):
    return linalg.solve_discrete_are(A,B,Q,R)
```

- Discrete State Space:

```
def discrete_ss(dt=0.33, xc=10):
    space = []
    l = int(xc/dt)
    for x1 in np.linspace(-xc, xc, int(xc/dt)):
        for x2 in np.linspace(-xc, xc, int(xc/dt)):
            x = np.array([x1,x2])
            space.append(x)
    space.append(np.array([0.,0.]))

    space = np.asarray(space)
    plt.plot(space[:,0],space[:,1], 'bo')
    plt.savefig('grid.pdf', bbox_inches='tight')
    plt.show()

    return space
```

- Compute the graphics of the Attraction:

```
def get_attraction_set(grid,A,B,Q,R,P,R2,N, uLB, uUB, xLB0, xUB0, xLB1, x
UB1, letter):
    result = []
    x0_set = []
    for x0 in tqdm(grid):
        x, _, status = recent_horizon(A,B,Q,R,P,x0,N, uLB, uUB, xLB0, xUB
0, xLB1, xUB1)
        if letter != 'b':
            if status not in ["infeasible", "unbounded"]:
                if R2:
                    result.append(x)
                    x0_set.append(x0)
                elif linalg.norm(x[-1]) <= 1e-4:
                    result.append(x)
                    x0_set.append(x0)
        else:
            result.append(x)
            x0_set.append(x0)

    return np.asarray(result), np.asarray(x0_set)
```

Result with Attraction and trajectories:

* Discrete Grid:

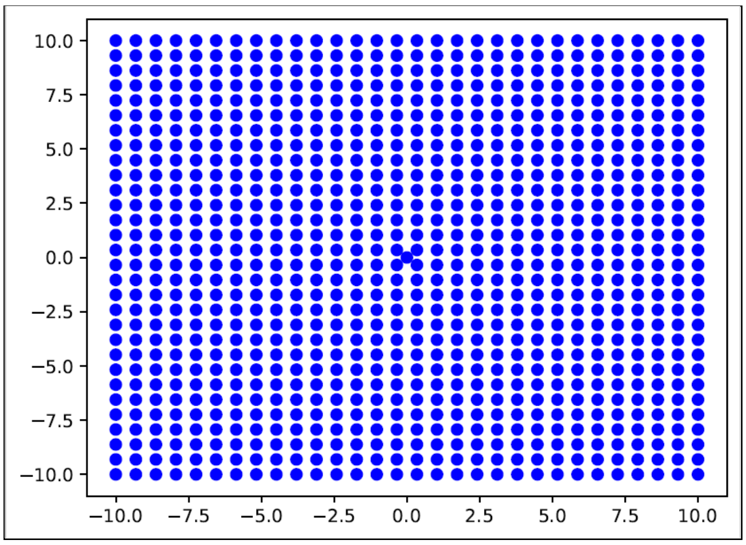

Figure 3 - Discrte State Space

c)

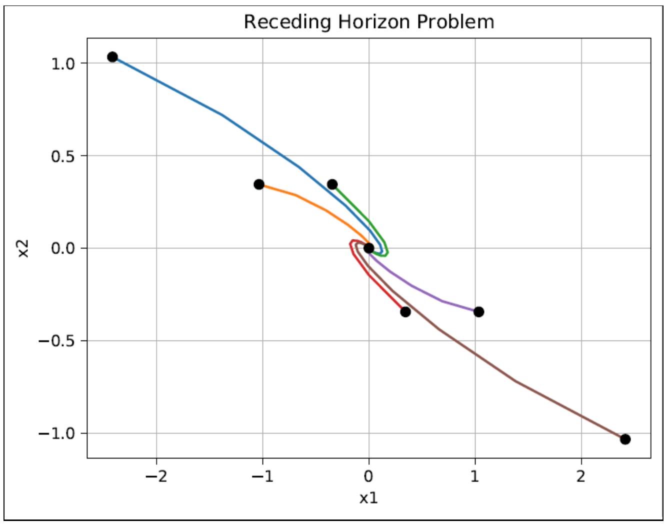

Figure 4 - Trajectory and X0 set in black dots.

d)

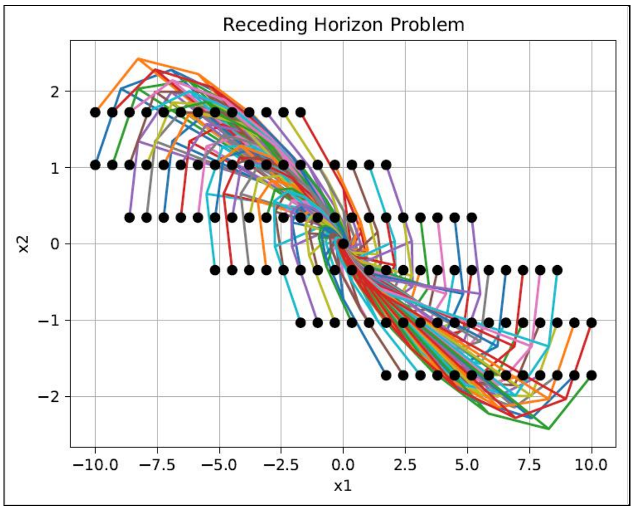

Figure 5 - Trajectory and X0 set in black dots.

e)

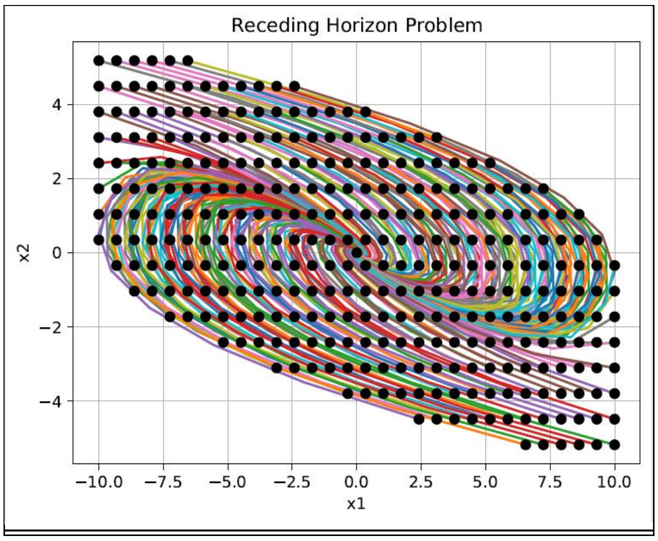

Figure 6 - Trajectory and X0 set in black dots.

f)

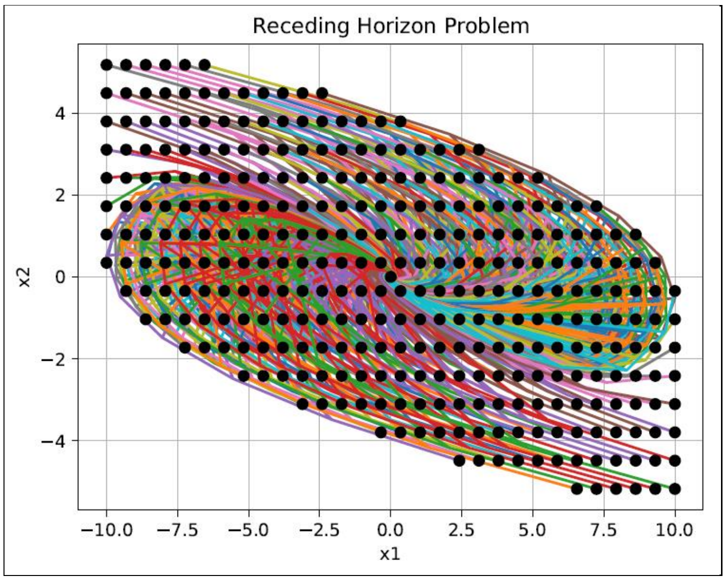

Figure 7 - Trajectory and X0 set in black dots.

g) The results follow the expectations regarding the numbers of steps and the distance of initial states from the origin. With less N the initial states need to be close to origin to get feasibility and reach the Xf set. The questions *e* and *f* have more points and trajectories compared to c and d, these results was influenced also by the restrictions of Xf's, for c and d we facing the restrictions that Xf = 0.

h) Python code:

```
def iterate_N(A,B,Q,R,P, uLB, uUB, xLB0, xUB0, xLB1, xUB1):
    N = range(2, 7)
    x0 = np.array([-2.4137931,  1.03448276])
    trajectory = []
    list_costs = []
    for n in tqdm(N):
        x, costs, _ = recent_horizon(A,B,Q,R,P,x0,n, uLB, uUB, xLB0, xUB0
, xLB1, xUB1)
        if linalg.norm(x[-1]) <= 1e-4:
            trajectory.append(x)
            list_costs.append(costs)

    plot_traj(np.asarray(trajectory), len(N))
    plot_cost(list_costs)
```

Considering the initial state: x0 = [-2.4137931, 1.03448276], it's feasible and reached for Xf = 0 to N = 2 to 6. We can verified that when N increase, the number of planned trajectory increase and more trajectories reached the Xf, the costs are same for all N's because we departure from the same initial point for all N's and reached the Xf. Plots below:

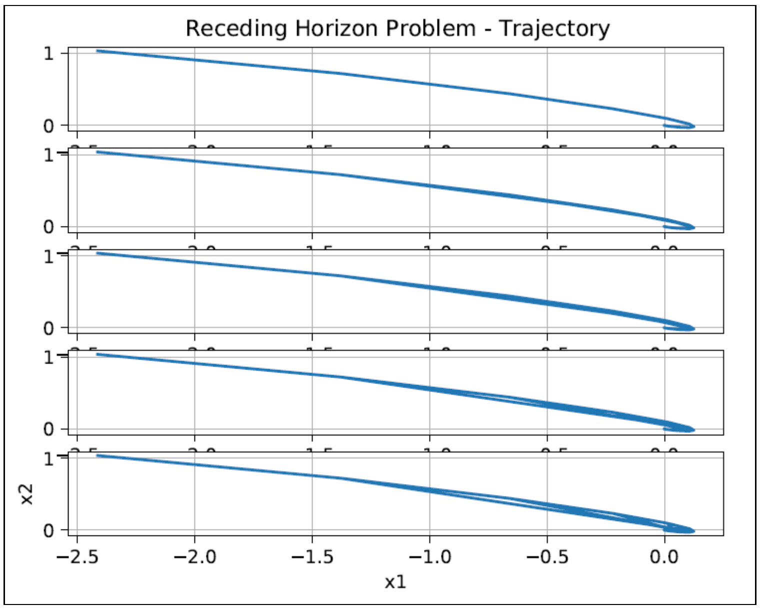

Figure 8 - Trajectories for N =2 to 6.

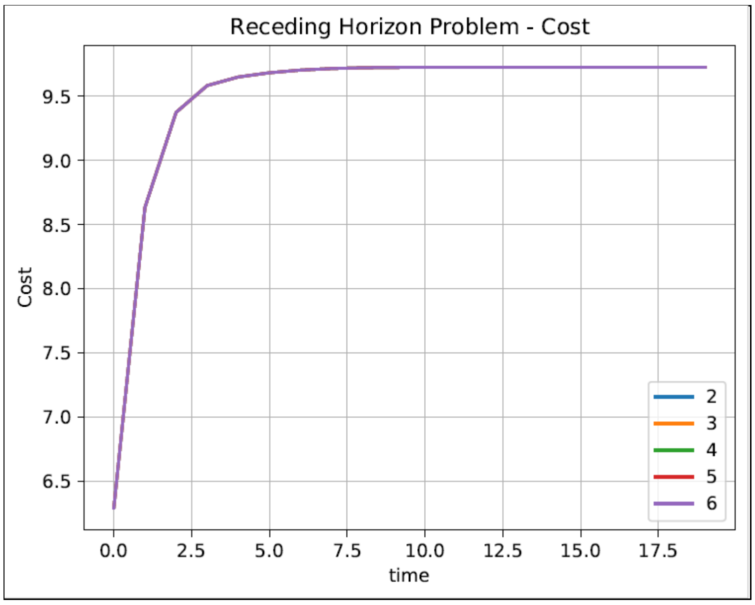

Figure 9 - Costs N=2 to 6.

Problem 3: MPC Terminal Invariant and Stability

a)X_f *and* P,

Based on lecture_12 slide 13. A is not asymptotically stable (eigenvalues greater than zero).

Python code:

```python
def compute_riccati_gain(A, B, Q, R):
    Pinf = linalg.solve_discrete_are(A,B,Q,R)
    Finf = - linalg.inv(R + np.transpose(B) @ Pinf @ B) @ (np.transpose(B
) @ Pinf @ A)

    return Pinf, Finf

# letter(a)
def compute_Xf(A, B, Finf):
    Xf = []
    for x1 in np.linspace(-10, 10, int(10/0.1)):
        for x2 in np.linspace(-10, 10, int(10/0.1)):
            # (A + B*Finf)x(t) - x(t) E X and Finf*x(t) E U
            x = (A + B @ Finf) @ np.array([x1,x2])
            u = Finf @ np.array([x1,x2])
            x_norm = linalg.norm(x, 2)
            u_norm = linalg.norm(u, 2)
            if x_norm <= 5 and u_norm <= 1:
                Xf.append(x)

    _, axes = plt.subplots(1)
    axes.set_title('Xf')
    axes.grid(True)
    axes.set_xlabel('x1')
    axes.set_ylabel('x2')
    axes.set_ylim([-10,10])
    axes.set_xlim([-10,10])
    plt.plot(Xf[:,0],Xf[:,1], 'bo')
    plt.savefig('problem_03_Xf.pdf', bbox_inches='tight')
    plt.show()

    return Xf
```

* Xf

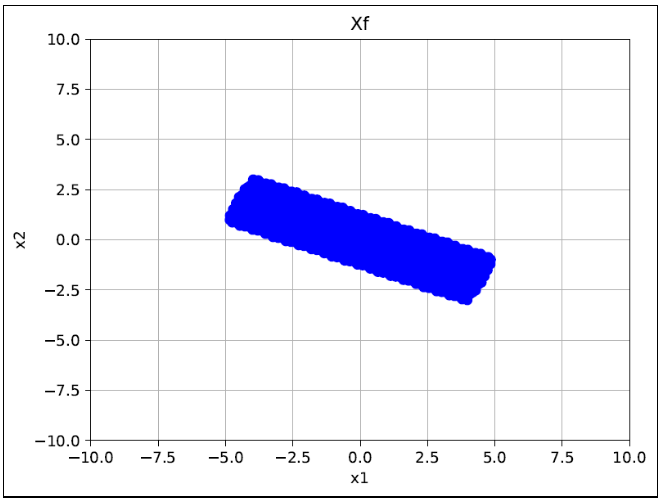

Figure 10 - Xf with x and u belongs to X and U constraints, respectively.

* P = P_riccati

P = [[4.15625333 ,3.4069349]

[3.4069349 ,7.61283467]]

b)Python code:

```
def compute_Xf_ellipsoid(A, M):
    Xf = []
    for x1 in np.linspace(-10, 10, 100):
        for x2 in np.linspace(-10, 10, 100):
            x = np.array([x1,x2])
            x_norm = linalg.norm(x)
            if x_norm <= 5:
                k = x @ (A.T @ M) @ A @ x
                if k <= 1:
                    Xf.append(x)

    _, axes = plt.subplots(1)
    axes.set_title('Xf')
    axes.grid(True)
    axes.set_xlabel('x1')
    axes.set_ylabel('x2')
    axes.set_ylim([-10,10])
    axes.set_xlim([-10,10])
    plt.plot(Xf[:,0],Xf[:,1], 'bo')
    plt.savefig('problem_03_b_Xf.pdf', bbox_inches='tight')
    plt.show()

    return Xf
```

* Calculating

$$M - A^T M A \succcurlyeq 0,$$

$$\begin{bmatrix} 0.04 & 0 \\ 0 & 1.06 \end{bmatrix} - \begin{bmatrix} 0.95 & 0.5 \\ 0 & 0.95 \end{bmatrix}^T \begin{bmatrix} 0.04 & 0 \\ 0 & 1.06 \end{bmatrix} \begin{bmatrix} 0.95 & 0.5 \\ 0 & 0.95 \end{bmatrix} \succcurlyeq 0,$$

The result,

$$\begin{bmatrix} 0.0039 & -0.0190 \\ -0.0190 & 0.0934 \end{bmatrix}$$

Has all eigenvalues greater than or equals to zero,

$$\begin{bmatrix} 0 \\ 0.0972 \end{bmatrix}$$

* Xf

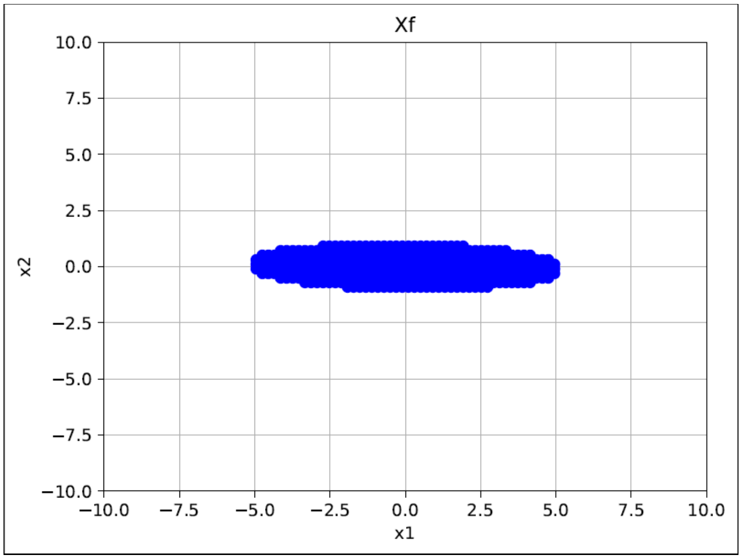

Figure 11 - Invariant Set Xf

c)Python Code:

```
def recent_horizon(A,B, Q, R, P, Xf, item,x0=np.array([-3.,-
2.5]), N=4, uUB=1, xUB=5):
    n = Q.shape[0]
    m = R.shape[0]
    X = {}
    U = {}
    u = []
    x = []

    cost_terms = []
    constraints = []
    status = ""
    T = 10
    x_0 = x0
    for t in range(T):
        for k in range(N):
            X[k] = cvx.Variable(n)
            U[k] = cvx.Variable(m)

            if item == 'i' or item == 'iii':
                cost_terms.append(cvx.quad_form(X[k] - Xf, Q))
            elif item == 'ii' or item == 'iv':
                cost_terms.append(cvx.quad_form(X[k], Q))

            cost_terms.append(cvx.quad_form(U[k], R))

            constraints.append(cvx.norm(U[k]) <= uUB)
            constraints.append(cvx.norm(X[k]) <= xUB)

            if k == 0:
                constraints.append(X[k] == x_0)

            if k > 0:
                constraints.append(A @ X[k - 1] + B @ U[k - 1] == X[k])

        X[k+1] = cvx.Variable(n)
        if item == 'i':
            ##################################################
            constraints.append(A @ X[k] + B @ U[k] == X[k + 1])
            cost_terms.append(cvx.quad_form(X[k + 1] - Xf, P))
            ##################################################
        elif item == 'ii':
            ##################################################
            constraints.append(A @ X[k] + B @ U[k] == X[k + 1])
            cost_terms.append(cvx.quad_form(X[k + 1] - Xf, P))
            ##################################################
        elif item == 'iii' or item == 'iv':
            ##################################################
            constraints.append(A @ X[k] + B @ U[k] == X[k + 1])
            cost_terms.append(cvx.quad_form(X[k + 1], P))
            ##################################################

        obj = cvx.Minimize(cvx.sum(cost_terms))
        problem = cvx.Problem(obj, constraints)
        problem.solve()

        status = problem.status
        if status in ["infeasible", "unbounded"]:
            break
        else:
            for k in range(N):
                u.append(U[k].value)
                x.append(X[k].value)
            x_0 = A @ X[0].value + B @ U[0].value
            x.append(X[k+1].value)
    return x, u

def problem_3(Xf, item):
    trajectories = []
    controls = []
    for xf in tqdm(Xf):
        x, u = recent_horizon(A,B,Q,R, P, xf, item)
        trajectories.append(x)
        controls.append(u)

    plot_cases(item,trajectories,controls, Xf)


def plot_cases(letter, traj, ctrls, Xf):
    _, axes = plt.subplots(2)
    for x in traj:
        x = np.asarray(x)
        axes[0].plot(x[:,0], x[:,1])
    axes[0].grid(True)
    axes[0].plot(Xf[:,0], Xf[:,1], 'ko')
    axes[1].grid(True)
    for u in ctrls:
        axes[1].plot(u)
    plt.savefig('problem_03_c_' + letter + '_v2_.pdf', bbox_inches='tight
')
    plt.show()
```

i)

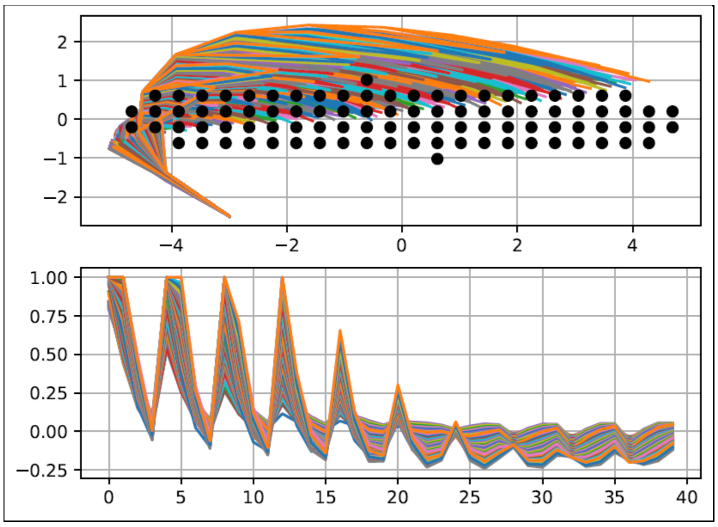

Figure 12 - Trajectories and Xf set (black dots).

ii)

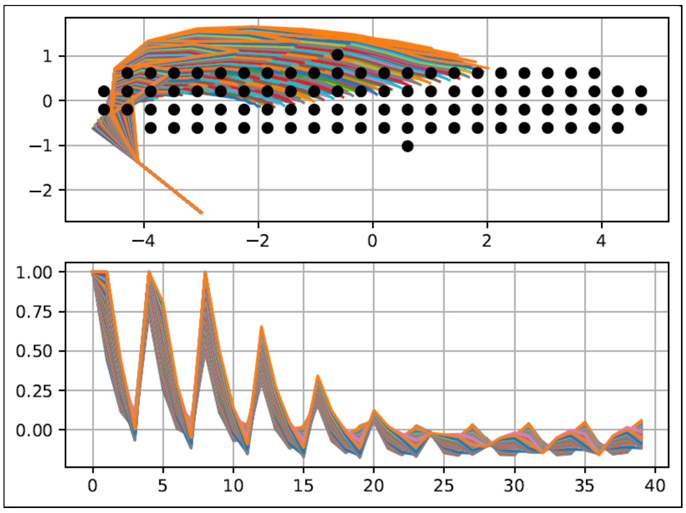

Figure 13 - Trajectories and Xf set (black dots).

iii)

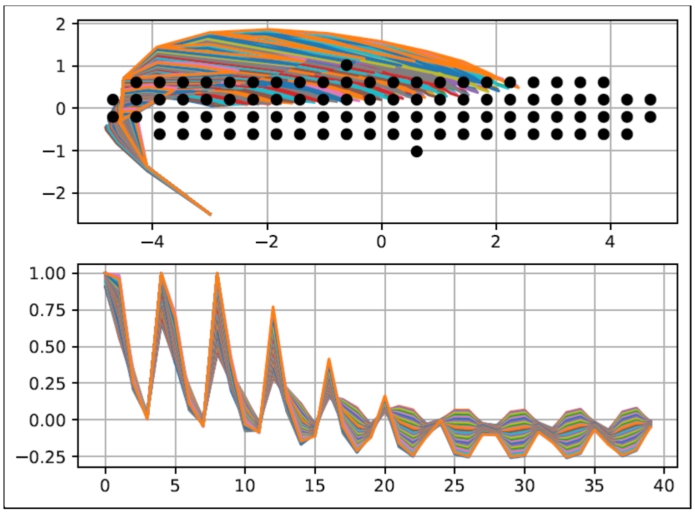

Figure 14 - Trajectories and Xf set (black dots).

iv)

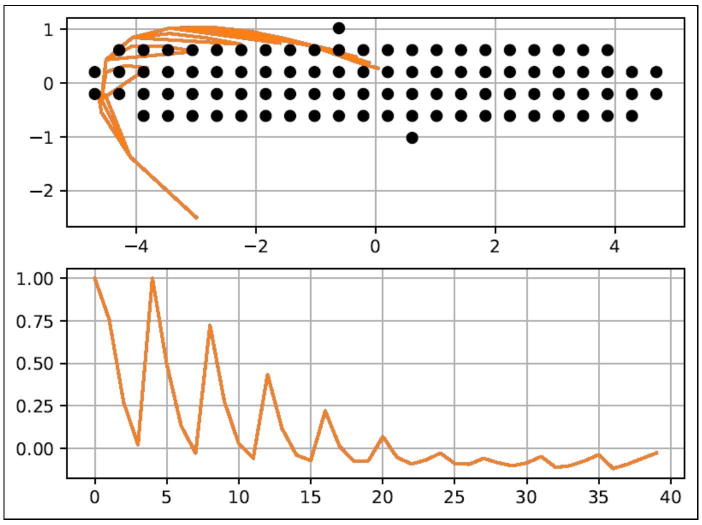

Figure 15 - Trajectories and Xf set (black dots).

Problem 4: Introduction to Reinforcement Learning

a) Python code:

```python
from model import dynamics, cost
import numpy as np
from scipy import linalg
import control as ctrl

dynfun = dynamics(stochastic=False)
# dynfun = dynamics(stochastic=True) # uncomment for stochastic dynamics

costfun = cost()


T = 100 # episode length
N = 100 # number of episodes
gamma = 0.95 # discount factor
TOLERANCE = 1e-12

total_costs = []

# Riccati recursion
def Riccati(A,B,Q,R):

    # TODO implement infinite horizon riccati recursion
    P = np.zeros((4,4))

    for i in range(N):
        L_next = (np.linalg.pinv(R + B.T @ P @ B) @ B.T @ P @ A)
        P_next = Q + A.T @ P @ (A - B @ L_next)

        if np.max(np.abs(P_next - P)) < TOLERANCE:
           break

        P = P_next

    L = L_next
    P = P_next

    return L,P


A = dynfun.A
B = dynfun.B
Q = costfun.Q
R = costfun.R

L,P = Riccati(A,B,Q,R)
print(L)
```

```
total_costs = []

for n in range(N):
    costs = []

    x = dynfun.reset()
    for t in range(T):

        # policy
        u = (-L @ x)

        # get reward
        c = costfun.evaluate(x,u)
        costs.append((gamma**t)*c)

        # dynamics step
        x = dynfun.step(u)

    total_costs.append(sum(costs))

print(np.mean(total_costs))
```

Simulated Average Cost : 120.15394345797391

b)Python code:

```
from model import dynamics, cost
import numpy as np
import scipy
from scipy import linalg
import matplotlib.pyplot as plt
from tqdm import tqdm


stochastic_dynamics = False # set to True for stochastic dynamics
dynfun = dynamics(stochastic=stochastic_dynamics)
costfun = cost()

T = 100 # episode length
N = 100 # number of episodes
gamma = 0.95 # discount factor
TOLERANCE = 1e-12

total_costs = []

# Riccati recursion
def Riccati(A,B,Q,R):

    # TODO implement infinite horizon riccati recursion
    P = np.zeros((4,4))

    for i in range(N):
        L_next = (np.linalg.pinv(R + B.T @ P @ B) @ B.T @ P @ A)
        P_next = Q + A.T @ P @ (A - B @ L_next)

        if np.max(np.abs(P_next - P)) < TOLERANCE:
           break

        P = P_next

    L = L_next
    P = P_next

    return L,P

def Riccati2(A,B,Q,R):
    P = scipy.linalg.solve_discrete_are(A, B, Q, R)
    L = np.linalg.pinv(R + B.T @ P @ B) @ B.T @ P @ A

    return L, P

A = np.random.rand(4, 4)
B = np.random.rand(4, 2)
Q = np.eye(4)
R = np.eye(2)
L_star = np.array([[2.51210992,-
1.03523418, 3.10840684,0.11485763], [0.12845042,0.95608089,0.07756693, 1.
17061578]])
L_iter = np.zeros((N,T))

for n in tqdm(range(N)):
    costs = []

    if n == 0:

       Q_hat = Q
       R_hat = R
       A_hat = A
       B_hat = B

       C = np.concatenate([A,B], axis = 1)
       P_prev = np.eye(6)
       C_prev = np.concatenate([A,B], axis = 1)

       P_prev2 = np.eye(20)
       F_prev = np.random.rand(20,1)
    x = dynfun.reset()

    for t in range(T):

        # TODO compute policy
        L,P_Ricatti = Riccati(A_hat, B_hat, Q_hat, R_hat)

        L_iter[n] = linalg.norm(L_star - L)

        # compute action
        u = (-L @ x)

        z = np.concatenate((x.T, u.T))
        z = z.T
        z = z.reshape((6,1))

        # get reward
        c = costfun.evaluate(x,u)
        costs.append((gamma**t)*c)

        # dynamics step
        xp = dynfun.step(u)

        P = P_prev - (P_prev @ z @ z.T @ P_prev) / (1 + z.T @ P_prev @ z)
        C = C_prev + ((P_prev @ z) @ (xp.T - z.T @ C_prev.T) / (1 + z.T @
 P_prev @ z)).T
        P_prev = P
        C_prev = C

        A_hat = C[0:4, 0:4]
        B_hat = C[0:4, 4:6]

        u = u.reshape((2,1))

        x2 = np.outer(x, x)
        u2 = np.outer(u, u)

        z2 = np.concatenate((x2.flatten(), u2.flatten()))
        z2 = z2.reshape(z2.shape[0],1)

        P2 = P_prev2 - (P_prev2 @ z2 @ z2.T @ P_prev2) / (1 + z2.T @ P_pr
ev2 @ z2)
        F = F_prev + (P_prev2 @ z2) @ (c - z2.T @ F_prev) / (1 + z2.T @ P
_prev2 @ z2)
        P_prev2 = P2
        F_prev = F

        Q_hat = F[0:16].reshape((4,4))
        R_hat = F[16:20].reshape((2,2))
        x = xp.copy()

    total_costs.append(sum(costs))
print(np.mean(total_costs))


_, axes = plt.subplots(1)
axes.set_title('Costs')
axes.set_xlabel('time')
axes.set_ylabel('Cost')
axes.grid(True)
axes.grid(True)
axes.semilogy(np.arange(0, N), total_costs)
plt.savefig('problem_04_b_costs.pdf', bbox_inches='tight')
plt.show()

_, axes = plt.subplots(1)
axes.set_title('L* - Lt')
axes.set_xlabel('time')
axes.set_ylabel('||L* - L||')
axes.grid(True)
axes.grid(True)
axes.plot(L_iter[:,:])
plt.savefig('problem_04_b_norm.pdf', bbox_inches='tight')
plt.show()
```

* Plots Non-Stochastic:

:

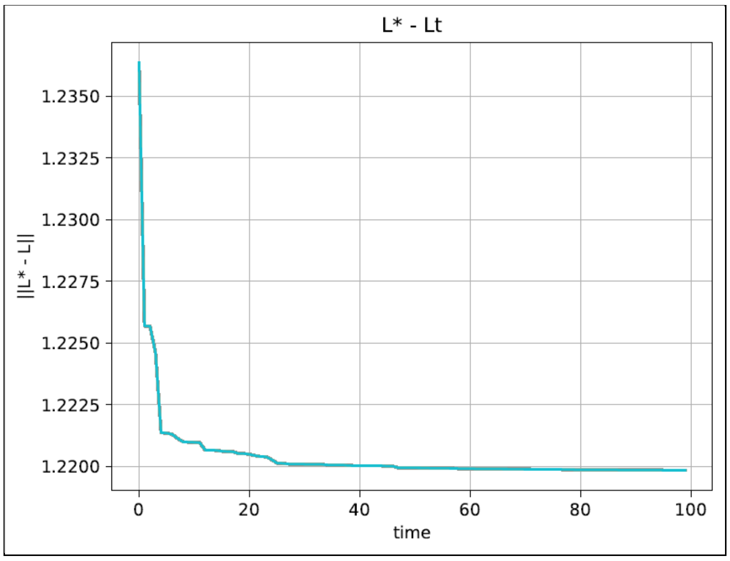

Figure 16 - ||L - Lt||*

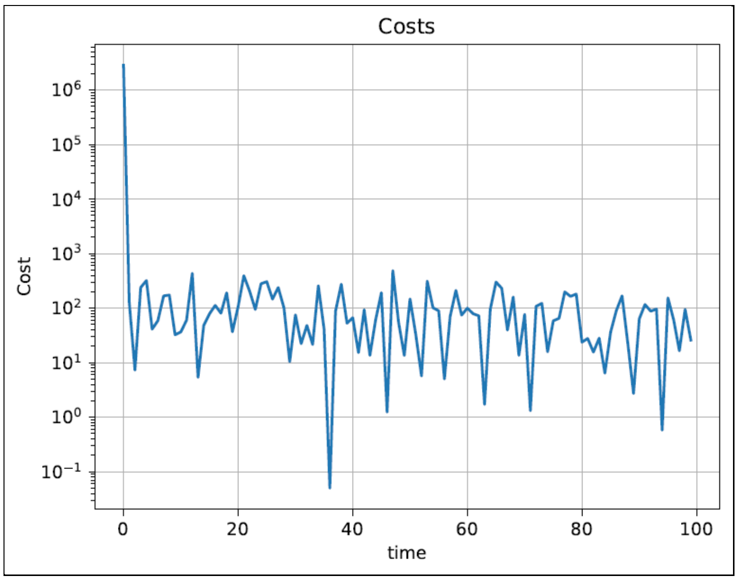

Figure 17- Costs x Time.

*Plots Stochastic:

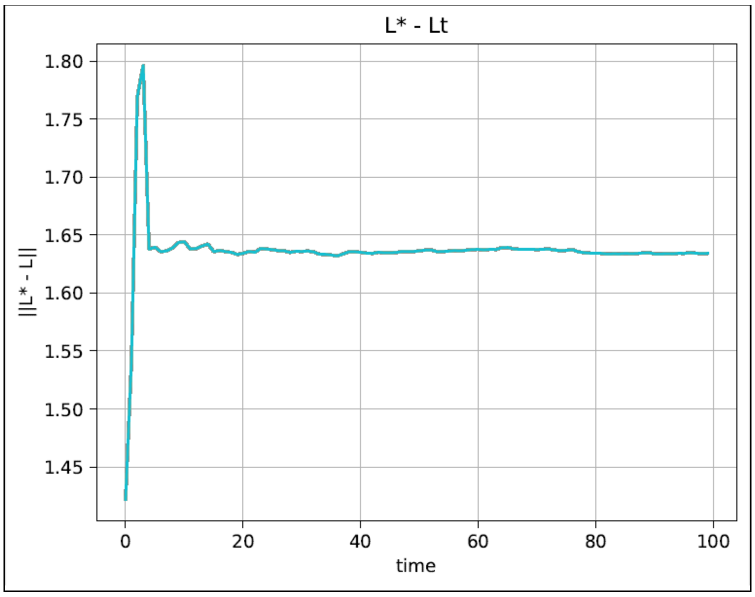

Figure 18 - ||L - Lt|| Stochastic.*

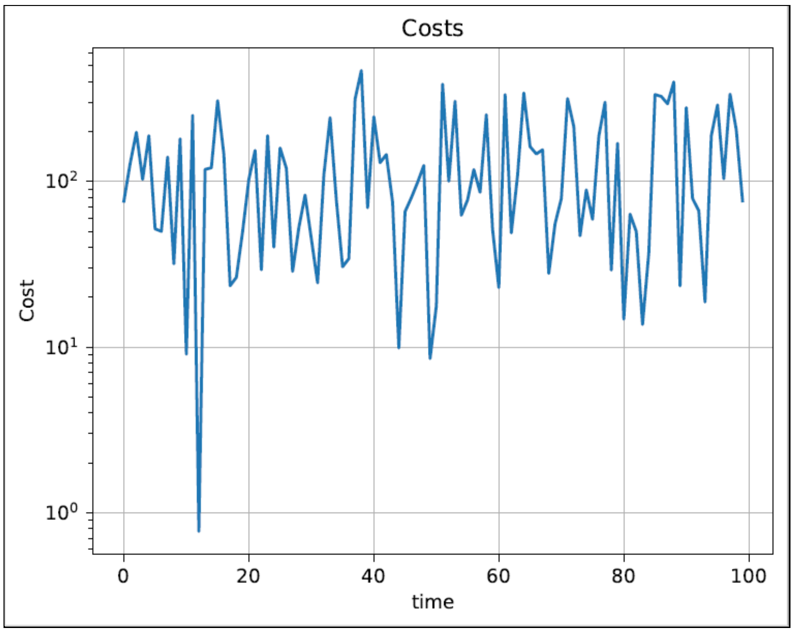

Figure 19 - - Costs x Time Stochastic

c)Python Code:

```
from model import dynamics, cost
import numpy as np
import scipy
from scipy import linalg
from tqdm import tqdm
import matplotlib.pyplot as plt


stochastic_dynamics = False # set to True for stochastic dynamics
dynfun = dynamics(stochastic=stochastic_dynamics)
costfun = cost()

T = 100 # episode length
N = 100 # number of episodes
gamma = 0.95 # discount factor
TOLERANCE = 1e-12

total_costs = []

# Riccati recursion
def Riccati(A,B,Q,R):

    # TODO implement infinite horizon riccati recursion

    P = np.zeros((4,4))

    for i in range(N):
        L_next = gamma * (np.linalg.pinv(R + gamma * B.T @ P @ B) @ B.T @
 P @ A)
```

```
        P_next = Q + A.T @ P @ (A - B @ L_next)

        if np.max(np.abs(P_next - P)) < TOLERANCE:
            break

        P = P_next

    L = L_next
    P = P_next

    return L,P

def Riccati2(A,B,Q,R):
    P = scipy.linalg.solve_discrete_are(A, B, Q, R)
    L = gamma * np.linalg.pinv(R + gamma * B.T @ P @ B) @ B.T @ P @ A

    return L, P

A = np.random.rand(4, 4)
B = np.random.rand(4, 2)
Q = np.eye(4)
R = np.eye(2)
L_iter = np.zeros(T)

L_star = np.array([[2.51210992,-
1.03523418, 3.10840684,0.11485763], [0.12845042,0.95608089,0.07756693, 1.
17061578]])

Q_hat = Q
R_hat = R
A_hat = A
B_hat = B

C = np.concatenate([A,B], axis = 1)
P_prev = 200*np.eye(6)
C_prev = C

P_prev2 = 200*np.eye(20)
F_prev = np.random.rand(20,1)

norms = []
L, _ = Riccati(A_hat, B_hat, Q_hat, R_hat)
for n in tqdm(range(N)):
    costs = []
    x = dynfun.reset()

    if n > 1:
        norms.append(np.linalg.norm(L_star - L, 2))
    for t in range(T):

        # compute action
        u = np.random.multivariate_normal(-L @ x, np.eye(2))

        z = np.concatenate((x.T, u.T))
        z = z.T
        z = z.reshape((6,1))

        # get reward
        c = costfun.evaluate(x,u)
        costs.append((gamma**t)*c)

        # dynamics step
        xp = dynfun.step(u)

        # dynamics recursive least squares update
        P = P_prev - (P_prev @ z @ z.T @ P_prev) / (1 + z.T @ P_prev @ z)
        C = C_prev + ((P_prev @ z) @ (xp.T - z.T @ C_prev.T) / (1 + z.T @
 P_prev @ z)).T
        P_prev = P
        C_prev = C

        A_hat = C[0:4, 0:4]
        B_hat = C[0:4, 4:6]

        u = u.reshape((2,1))

        # Cost recursive least squares update
        x2 = np.outer(x, x)
        u2 = np.outer(u, u)

        z2 = np.concatenate((x2.flatten(), u2.flatten()))
        z2 = z2.reshape(z2.shape[0],1)

        P2 = P_prev2 - (P_prev2 @ z2 @ z2.T @ P_prev2) / (1 + z2.T @ P_pr
ev2 @ z2)
        F = F_prev + (P_prev2 @ z2) @ (c - z2.T @ F_prev) / (1 + z2.T @ P
_prev2 @ z2)
        P_prev2 = P2
        F_prev = F

        Q_hat = F[0:16].reshape((4,4))
        R_hat = F[16:20].reshape((2,2))

        Q_hat = 0.5 * (Q_hat + Q_hat.T)
        R_hat = 0.5 * (R_hat + R_hat.T)

        x = xp.copy()
    # TODO policy improvement step
    L,_ = Riccati(A_hat, B_hat, Q_hat, R_hat) # Uk+1
    total_costs.append(sum(costs))

print(np.mean(total_costs))

_, axes = plt.subplots(1)
axes.set_title('Costs')
axes.set_xlabel('time')
axes.set_ylabel('Cost')
axes.grid(True)
axes.semilogy(np.arange(0, N), total_costs)
plt.savefig('problem_04_c_cost.pdf', bbox_inches='tight')
plt.show()

_, axes = plt.subplots(1)
axes.set_title('L* - Lt')
axes.set_xlabel('time')
axes.set_ylabel('||L* - L||')
axes.grid(True)
axes.plot(norms)
plt.savefig('problem_04_c_norm.pdf', bbox_inches='tight')
plt.show()
```

* Plots Non - Stochastic:

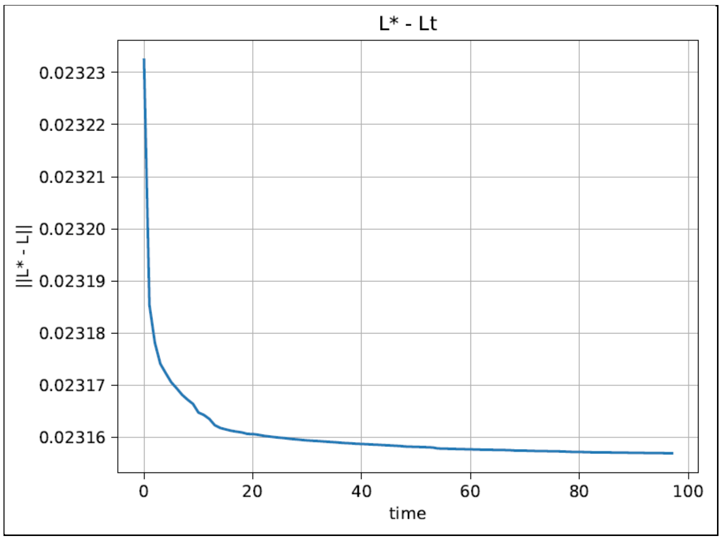

Figure 20 - ||L - L||. Non -Stochastic*

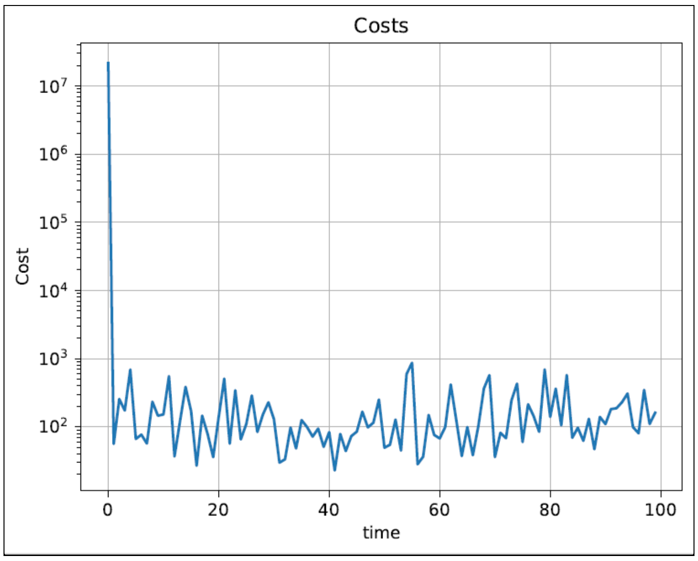

Figure 21 - Costs x Time Non-Stochastic

* Plots Stochastic:

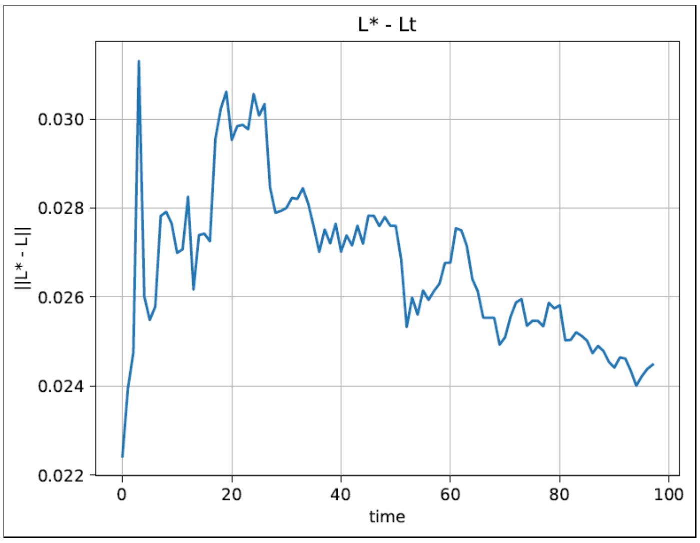

Figure 22 - ||L - L||. Stochastic*

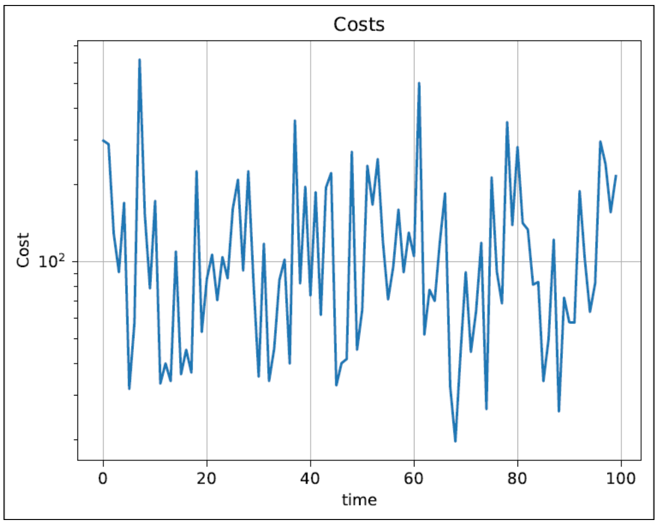

Figure 23 - Costs x Time Stochastic

d)Python code:

This code was done with another approach with the group partner, using pytorch and neural networks, the plots presented were related to Loos and Mean Reward.

```
from model import dynamics, cost
import numpy as np
import torch
from torch.autograd import Variable
import torch.nn.utils as utils
import gym
from matplotlib import pyplot as plt
from scipy.stats import multivariate_normal
import torch.optim as optim
from torch.distributions import Categorical
from torch.distributions import MultivariateNormal
from tqdm import tqdm



stochastic_dynamics = False # set to True for stochastic dynamics
dynfun = dynamics(stochastic=stochastic_dynamics)
costfun = cost()

T = 100
#N = 10000
N = 1000
gamma = 0.95 # discount factor

total_costs = []

# Define the policy Network
state_space_size = 4
output_size = 5 # 2-D Mu vector, 3 paramters for covariance matrix
```

```python
num_hidden_layer = 32
model = torch.nn.Sequential(torch.nn.Linear(state_space_size, num_hidden_
layer),
                            torch.nn.ReLU(),
                            torch.nn.Linear(num_hidden_layer, output_size
),
                            torch.nn.Softmax())

optimizer = optim.Adam(model.parameters(), lr=1e-3)
model.train()

losses = []
mean_rewards = []



for n in tqdm(range(N)):
    costs = []
    log_probs = []
    rewards = []
    entropies = []

    x = dynfun.reset()
    for t in range(T):

        # TODO compute action
        outputs = model(Variable(torch.FloatTensor(x)))
        mu_vector = outputs[0:2]

        L = torch.eye(2)
        L[0][0] = outputs[4]
        L[1][1] = outputs[3]
        L[1][1] = outputs[2]

        epsilon = 0.01
        cov_matrix = L @ L.T + epsilon * torch.eye(2)
        cov_matrix *= 0.1

        #print("mu_vector = " + str(mu_vector))
        #print("mu_vector shape = " + str(mu_vector.shape))
        #print("cov_matrix = " + str(cov_matrix))
        #print("cov_matrix shape = " + str(cov_matrix.shape))

        dist = MultivariateNormal(mu_vector, cov_matrix)
        u = dist.sample()
        entropies.append(dist.entropy())
        #print("u = " + str(u))

        log_prob = dist.log_prob(u)
        #print("log probability = " + str(log_prob))
        log_probs.append(log_prob)

        # get reward
        #c = costfun.evaluate(x,u)

        x1 = Variable(torch.FloatTensor(x))

        Q = torch.eye(4)
        Q[2,2] *= 0.1
        Q[3,3] *= 0.1
        R = 0.01 * torch.eye(2)
        c = torch.matmul(torch.matmul(x1, Q), x1) + torch.matmul(torch.ma
tmul(u, R), u)
        rewards.append(-c)

        # dynamics step
        xp = dynfun.step(u.detach().numpy())

        x = xp.copy()

    # TODO update policy
    R = torch.zeros(1, 1)

    loss = 0

    for i in reversed(range(len(rewards))):
        R = gamma * R + rewards[i]
        #print("rewards[i] = " + str(rewards[i]))
        loss = loss - log_probs[i] * Variable(R) - 0.0001 * entropies[i]

    loss = loss / len(rewards)
    #print("Mean reward = " + str(np.mean(rewards)))
    #print("Loss = " + str(loss))
    losses.append(loss.item())
    mean_rewards.append(np.mean(rewards))

    optimizer.zero_grad()

    loss.backward()
    utils.clip_grad_norm(model.parameters(), 40)
    optimizer.step()

    total_costs.append(sum(costs))
```

Plots:

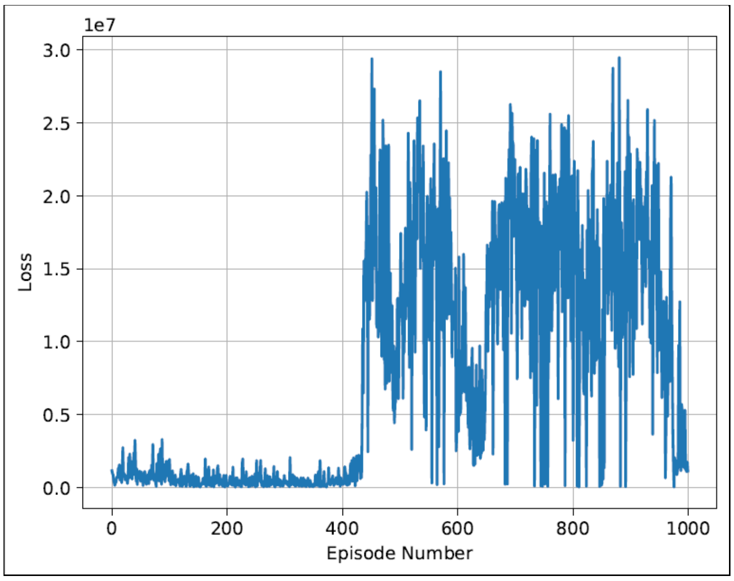

Figure 24 - Loss x Iterations

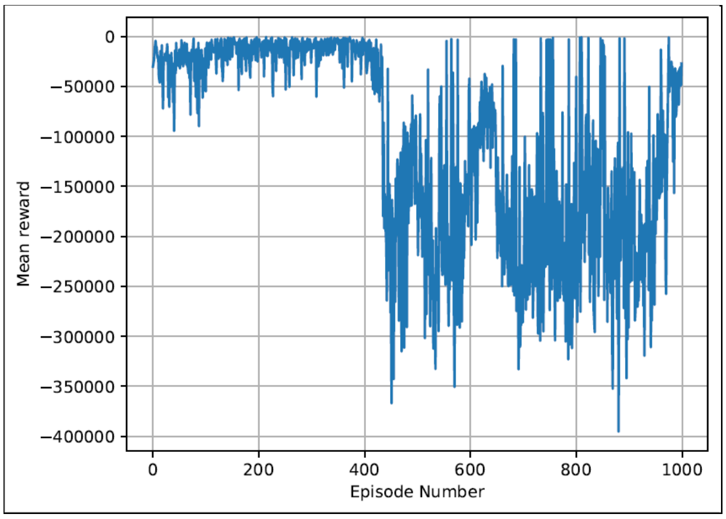

Figure 25 - Reward x Iterations

e) The problem (a) we have knowledge about the system dynamics and cost, the Riccati equation is calculated direct from the model, each step and cost as well-known from the model, the performance of this method is better and used as a benchmark.

The problem (b) we don't have known about the system and cost and we will estimate the dynamics and cost parameters, our control is computed by the Riccati Gain (L) every iteration accordingly with our updated parameters of the model using linear regression estimation, the performance in this way is worst compared to the previous. Q-function is estimated at each iteration.

The problem (c) we have the same process as the previous but now the control contains a random normal distribution component. The policy improvement now is realized at the end of episode. convergence related with controllability of matrix (A,B). In comparison with our benchmark in my simulation the results presented worst performance against the previous (a), (b).

The problem (d) using neural network has time consuming worst (a, b, and c), in the case of state vector increase (number of states) the performance tends to became worst.