**Homework 03**

**AA203: Optimal and learning-based Control**

Ricardo Luiz Moreira Paschoeto

[rp304154@stanford.edu](mailto:rp304154@stanford.edu)

**Problem 1:** HJ Reachability

a) Method PlanarQuadrotor.optimal control,

Python code:

    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,)` containing `[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,)` containing `[y, v\_y, phi, omega]`.

    Returns:

        A scalar, nonnegative iff the state is in the operational envelope.

    """

    # 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,

Uma imagem contendo Logotipo

Descrição gerada automaticamente

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 .* The ridge where touch the plane at v\_y = -6 is the state *x* bounded by operational envelope. Other ridges come from minimization regarding the t [0, -5] and action *u,* where .

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:

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

Gráfico, Gráfico de linhas

Descrição gerada automaticamente

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, xUB1, letter):

    result = []

    x0\_set = []

    for x0 in tqdm(grid):

        x, \_, status = recent\_horizon(A,B,Q,R,P,x0,N, uLB, uUB, xLB0, xUB0, 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:

Padrão do plano de fundo

Descrição gerada automaticamente

Figure 3 - Discrte State Space

c)

Gráfico, Gráfico de linhas

Descrição gerada automaticamente

Figure 4 - Trajectory and X0 set in black dots.

d)

Gráfico

Descrição gerada automaticamente

Figure 5 - Trajectory and X0 set in black dots.

e)

Gráfico

Descrição gerada automaticamente

Figure 6 - Trajectory and X0 set in black dots.

f)

Gráfico

Descrição gerada automaticamente

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:

Gráfico, Gráfico de linhas

Descrição gerada automaticamente

Figure 8 - Trajectories for N =2 to 6.

Gráfico, Gráfico de linhas

Descrição gerada automaticamente

Figure 9 - Costs N=2 to 6.

**Problem 3:** MPC Terminal Invariant and Stability

a),

Based on lecture\_12 slide 13. A is not asymptotically stable (eigenvalues greater than zero).

Python code:

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

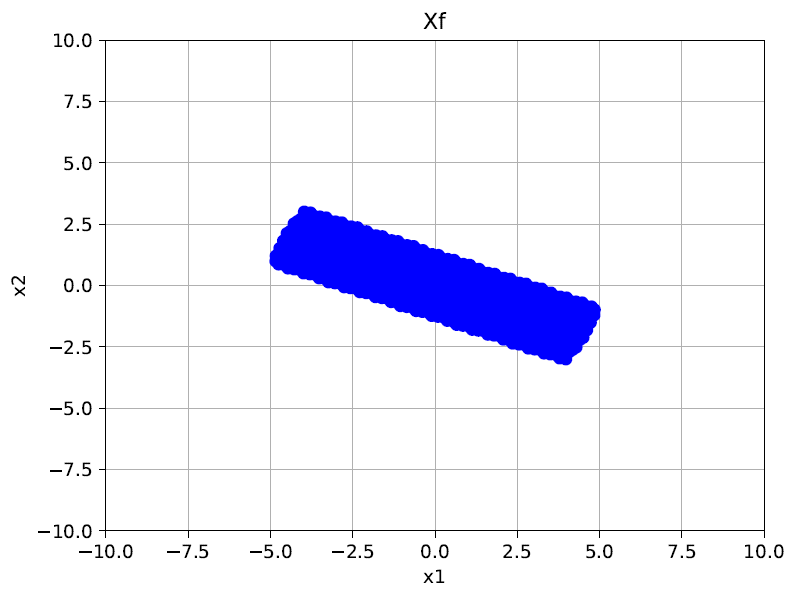


Figure 10 - Xf with x and u belongs to X and U constraints, respectively.

\* 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

The result,

Has all eigenvalues greater than or equals to zero,

\* Xf

Gráfico

Descrição gerada automaticamente

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)

Gráfico

Descrição gerada automaticamente

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

ii)

Gráfico

Descrição gerada automaticamente

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

iii)

Gráfico

Descrição gerada automaticamente

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

iv)

Gráfico

Descrição gerada automaticamente

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

**Problem 4:** Introduction to Reinforcement Learning

a) Python code:

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\_prev2 @ 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:

:

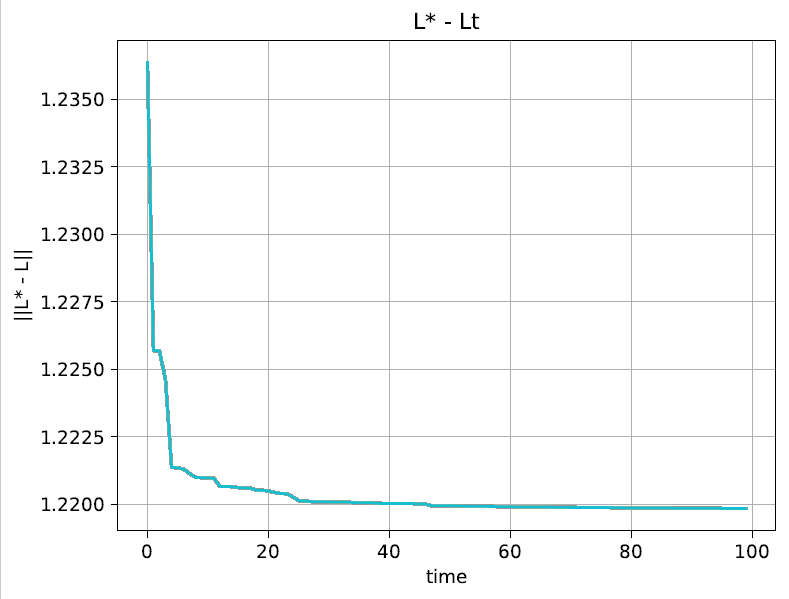


Figure - ||L\* - Lt||

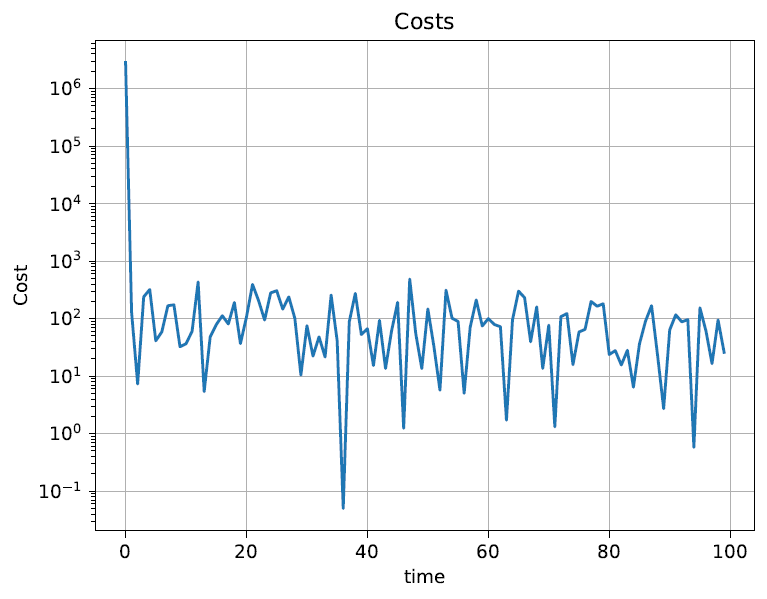


Figure 17- Costs x Time.

\*Plots Stochastic:

Gráfico, Gráfico de linhas

Descrição gerada automaticamente

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

Interface gráfica do usuário, Gráfico

Descrição gerada automaticamente

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\_prev2 @ 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:

Gráfico, Gráfico de linhas

Descrição gerada automaticamente

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

Gráfico, Histograma

Descrição gerada automaticamente

Figure 21 - Costs x Time Non-Stochastic

\* Plots Stochastic:

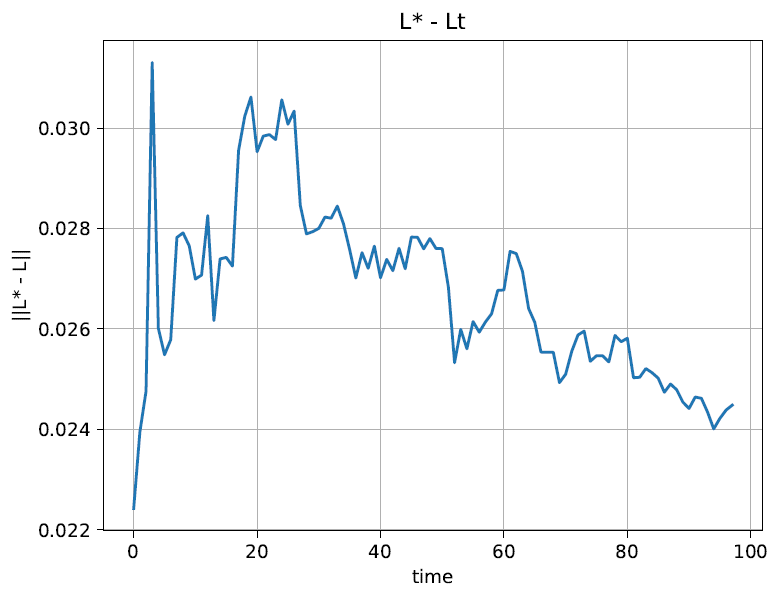


Figure 22 - ||L\* - L||. Stochastic

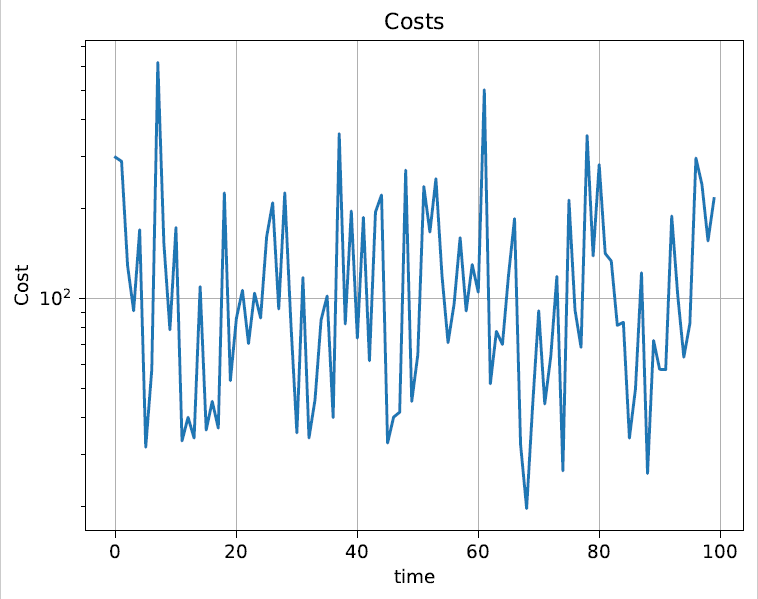


Figure 23 - Costs x Time Stochastic

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

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.matmul(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:

Gráfico, Histograma

Descrição gerada automaticamente

Figure - Loss x Iterations

Gráfico

Descrição gerada automaticamente

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.