Homework 3

Sam Akinwande
AA 203 - Optimal Control

May 22 ,2021

Problem 1. HJ Reachability

Part (a)

The optimal control for this problem reduces to a form of bang bang control. The expression for the bang bang control is derived on the next page. Please note that it is also possible to derive this expression using gradient methods (I originally did). We arrive at the same solution because we find that the Hamiltonian is minimized when T1 and T2 have opposite signs. Because our control is constrained to be between 0 and 18, this reduces to bang bang control.

Part (b)

The functional form for this problem can be represented as a Lipschitz continuous polynomial. The form can also be represented with a series of linear constraints (as shown in my code). Because a Lipschitz constant was not provided, I opted for the sequence of linear constraints. The expressions are shown in the attached code

Part (c)

Same as above

Part (d)

The controlled trajectories look somewhat reasonable. The stability of the controller seems finicky but overall the trajectories look reasonable. To buck the trend, I decided to take a slice of the isosurface at $y = 6.25$. The isosurface describes the regions of stable control for the HJB controller. One of the major bumps on the isosurface (shown below) corresponds to a region of relatively slow descent ($v_y < 0$). In the region, the orientation of the drone is level ($\phi \ = 0$). This region is considered stable/marginally stable because it is still possible to return the drone to the target set without violating control constraints. Note that the central bump spans a range of ω values. This is because the drone can still be stabilized for large values of ω within the operational envelope

Reduced dynamics $\quad x = [y, v_y, \phi, \omega]^T \rightarrow$ state

Solve using dynamic programming $\quad \mathcal{T} = [3, 7] \times [-1, 1] \times [-\pi/12, \pi/12] \times [-1, 1]$ – Target

$\mathcal{E} = [1, 9] \times [-6, 6] \times [-\infty, \infty] \times [-8, 8]$ – Envelope

Want to solve the HJB P.D.E & minimize the hamiltonian

$$H(x, p) = \min_u p^T f(x, u)$$

ⓐ

To minimize the hamiltonian,

$$u^* = \underset{u}{\operatorname{argmin}}\, p^T f(x, u)$$

Given P

$$p^T f(x,u) = p_1 v_y + \frac{p_2 (T_1 + T_2)\cos\phi - C_D^v v_y - mg}{m} + p_3 \omega + \frac{p_4 (T_2 - T_1) l - C_D^\phi \omega}{I_{yy}}$$

Isolate expressions that depend on u

$$\downarrow$$

$$\frac{p_2 (T_1 + T_2)\cos\phi}{m} + \frac{p_4 (T_2 - T_1) l}{I_{yy}}$$

Expand and collect like terms

$$\frac{p_2 T_1 \cos\phi}{m} + \frac{p_2 T_2 \cos\phi}{m} + \frac{p_4 T_1 l}{I_{yy}} - \frac{p_4 T_2 l}{I_{yy}}$$

$$\underbrace{\left(\frac{p_2 \cos\phi}{m} + \frac{p_4 l}{I_{yy}}\right)}_{A} T_1 + \underbrace{\left(\frac{p_2 \cos\phi}{m} - \frac{p_4 l}{I_{yy}}\right)}_{B} T_2$$

u is T_1 & T_2. Find T_1 & T_2 that minimize $p^T f(x, u)$

Expression is minimized using bang bang control

If $A > 0$, $T_2 = \min$ else $T_2 = \text{Max}$

If $B > 0$, $T_1 = \min$ else $T_1 = \max$

The zero isosurface is shown below.

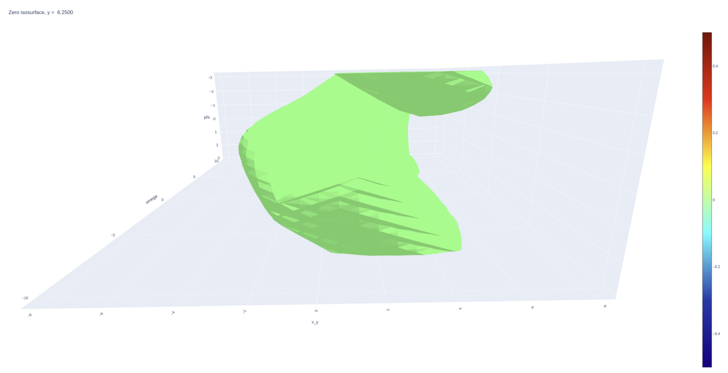

Figure 1: Zero Isosurface at $y = 6.25$

Part (e)

This approach results in a globally optimal control (because it solves the HJB equation over the entire state space). While global optimality is desirable, the approach is computationally expensive. In the (relatively) simple case of a drone with a 6-D state space, we had to reduce the dimensionality of the state space to make the solution tractable. Despite reducing the dimensionality of the problem, the solution still required 30 seconds of computation. This makes the HJB controller impractical for online operation. In short, the HJB controller is globally optimal but it suffers from the curse of dimensionality. MPC (other online methods) could arguably provide similar performance while requiring fewer computational resources
The code used in this problem is shown below

```
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

y, v_y, phi, omega = state

coeff1 = (((grad_value[1]*jnp.cos(phi))/self.m)
 + ((grad_value[3])*self.l/self.Iyy))
coeff2 = (((grad_value[1]*jnp.cos(phi))/self.m)
- ((grad_value[3])*self.l/self.Iyy))
T1 = -1
T2 = -1

#the goal is to minimize total control input (T1 + T2)
T2 = jnp.where(coeff1 > 0, self.min_thrust_per_prop,
               self.max_thrust_per_prop)
T1 = jnp.where(coeff2 > 0, self.min_thrust_per_prop,
               self.max_thrust_per_prop)
optControl = jnp.array([T1, T2])

return optControl

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, nonpositive iff the state is in the target set.
"""

stateInd1 = jnp.where(state[0] >= 5., -1., 1.)
#returns a negative if y is feasible
stateInd11 = jnp.where(state[0] <= 6., -1., 1.)
#returns a negative if y is feasible
stateInd2 = jnp.where(state[1] >= -1, -1., 1.)
#returns a negative is vY is feasible
stateInd22 = jnp.where(state[1]  <= 1, -1., 1.)
#returns a negative is vY is feasible
stateInd3 = jnp.where(state[2] >= -jnp.pi/12 , -1., 1.) #as above
stateInd33 = jnp.where(state[2] <= jnp.pi/12, -1., 1.) #as above
stateInd4 = jnp.where(state[3] >= -1, -1., 1.)
stateInd44 = jnp.where(state[3] <= 1, -1., 1.)

stateInd = jnp.amax(jnp.array([stateInd1,stateInd11,stateInd2,stateInd22,
        stateInd3,stateInd33, stateInd4, stateInd44]))

return stateInd

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, nonpositive iff the state is in the operational envelope.
"""
y, v_y, phi, omega = state
stateInd1 = jnp.where(state[0] >= 4., -1., 1.)
#returns a negative if y is feasible
stateInd11 = jnp.where(state[0] <= 7., -1., 1.)
#returns a negative if y is feasible
stateInd2 = jnp.where(state[1] >= -6, -1., 1.)
#returns a negative is vY is feasible
stateInd22 = jnp.where(state[1]  <= 6, -1., 1.)
#returns a negative is vY is feasible
stateInd3 = jnp.where(state[2] >= -jnp.inf , -1., 1.) #as above
stateInd33 = jnp.where(state[2] <= jnp.inf, -1., 1.) #as above
stateInd4 = jnp.where(state[3] >= -8, -1., 1.)
```

```
stateInd44 = jnp.where(state[3] <= 8, -1., 1.)

stateInd = jnp.amax(jnp.array([stateInd1,stateInd11,stateInd2,stateInd22,
            stateInd3,stateInd33, stateInd4, stateInd44]))
return stateInd
```

Problem 2. MPC Feasibility

Part (a)

The code used to implement MPC is appended to the end of this problem

Part (b)

The receding horizon controller implemented in part a was used to simulate the closed loop trajectories for the specified states. The resulting trajectories are shown below. Please note that the dashed lines represent the predicted trajectories at each timestep while the solid lines represent the realized trajectories. The blue line corresponds to $\mathbf{x}_0 = [-4.5, 2]$ while the orange line corresponds to $\mathbf{x}_0 = [-4.5, 3]$

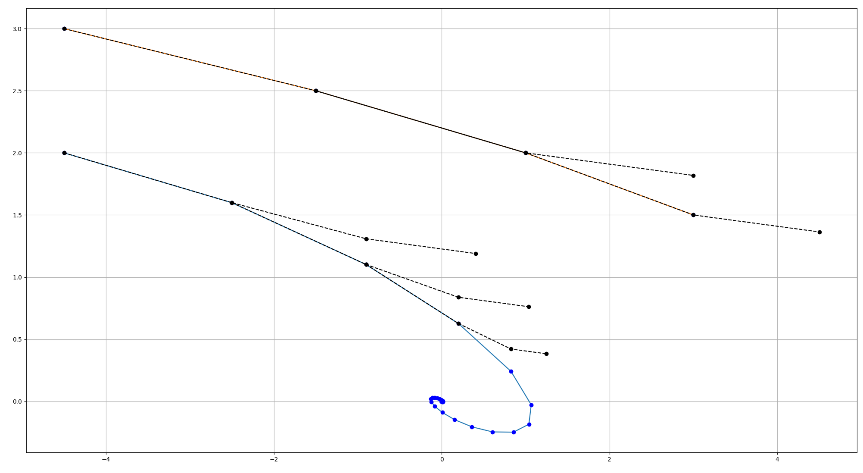

Figure 2: Simulating the Closed Loop Trajectories from the BBM Book

Part (c)

The domain of attraction for MPC with $N = 2$ and $X_f = 0$ is shown below. A discretization step of 0.5 was used to generate the plot. Please note that the Riccati recursion was used to compute P_∞

Part (d)

The domain of attraction for MPC with $N = 6$ and $X_f = 0$ is shown below. A discretization step of 0.5 was used to generate the plot. Please note that the Riccati recursion was used to compute P_∞

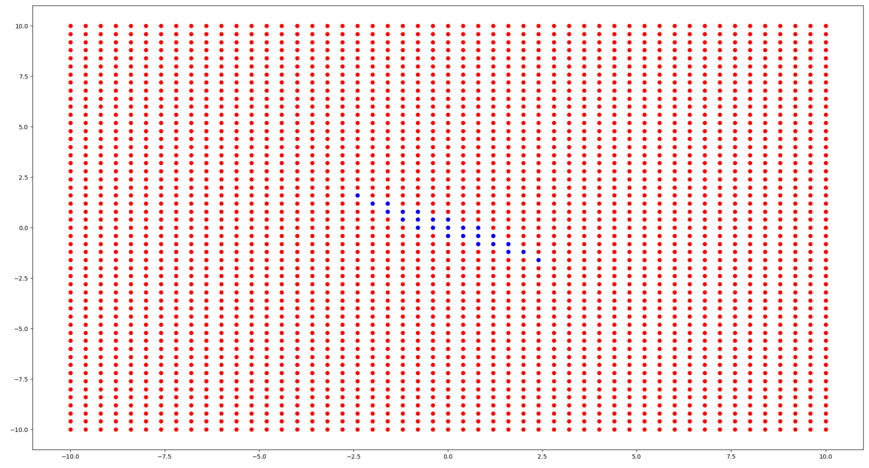

Figure 3: Domain of Attraction for $N = 2$ and $X_f = 0$

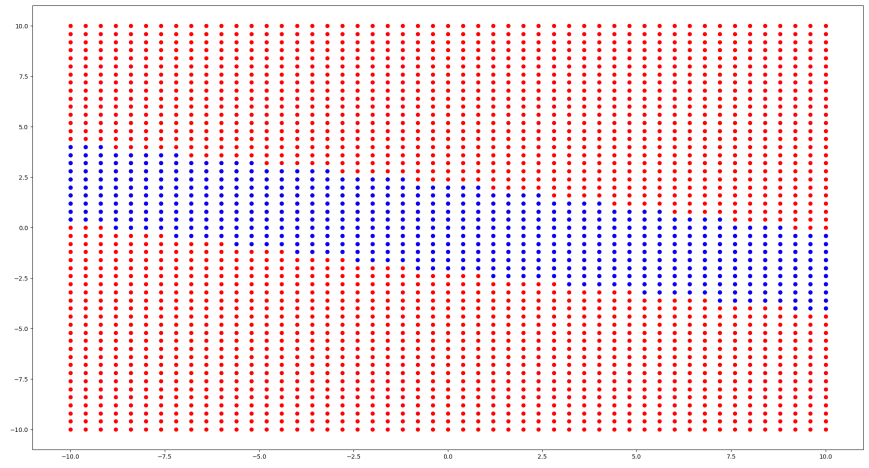

Figure 4: Domain of Attraction for $N = 6$ and $X_f = 0$

Part (e)

The domain of attraction for MPC with $N = 2$ and $X_f = \mathbb{R}^2$ is shown below. A discretization step of 0.5 was used to generate the plot. Please note that the Riccati recursion was used to compute P_∞

Part (f)

The domain of attraction for MPC with $N = 6$ and $X_f = \mathbb{R}^2$ is shown below. A discretization step of 0.5 was used to generate the plot. Please note that the Riccati recursion was used to compute P_∞

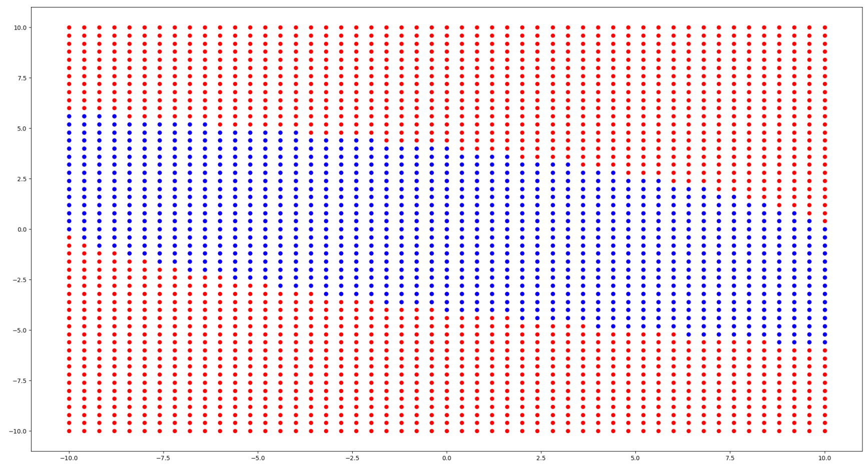

Figure 5: Domain of Attraction for $N = 2$ and $X_f = \mathbb{R}^2$

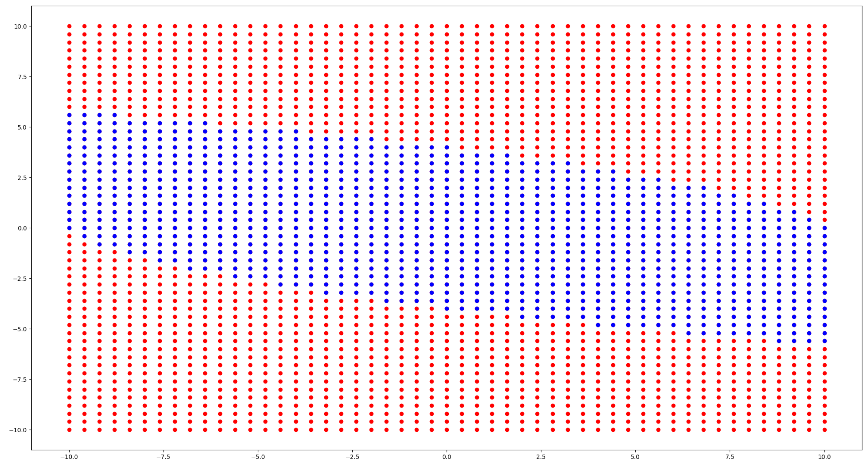

Figure 6: Domain of Attraction for $N = 6$ and $X_f = \mathbb{R}^2$

Part (g)

In part C with $N = 2$ and $X_f = 0$, the terminal constraint is really tight and the prediction horizon is (relatively) small. This results in a small feasible set of initial conditions as seen in Fig 3. In part D with $N = 6$ and $X_f = 0$, the terminal constraint is unchanged but the planning horizon is extended which results in larger set of feasible initial conditions as seen in Fig 4. This to be expected because the controller has more flexibility in computing a trajectory. In parts E and F, the terminal constraint is relaxed to $X_f = \mathbb{R}^2$. Because the terminal constraint is so loose, the prediction horizon has no effect on the feasibility of the initial condition. This is observed in Fig 5 and Fig 27

Part (h)

Using the intuition built from parts b-g, I selected $\mathbf{x}_0 = [-5, 2.5]$ and values of N spanning $[2, 4, 6, 8, 10]$. In this problem, $N = 2$ yields infeasible solutions but $N = 4, 6, 8, 10$ yield feasible solutions. Further, the prediction horizon barely affected the realized trajectory (for values of $N > 4$) but larger values of N resulted in marginally lower cost values. The simulated trajectory for different N values are shown below

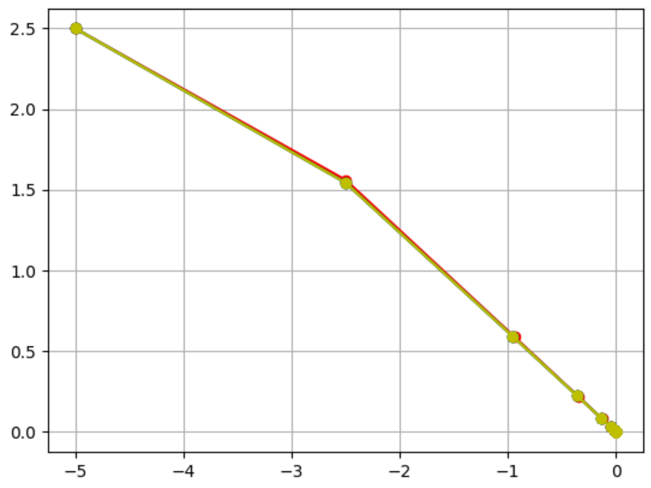

Figure 7: Simulated Trajectory for values of N ranging from 2 to 10

The code used for this problem is shown below

```
import numpy as np
import jax.numpy as jnp
import jax
import matplotlib.pyplot as plt
from tqdm import tqdm
import cvxpy as cvx


def MPC(Q,R,P,s0,sf, N,uB,sB):
#run for all time steps
u0 = 0
NMax = N
s = np.zeros((1,2)) #initialize s
u = np.zeros((1,1)) #initialize u
sCapture = np.zeros((1,N+1,2)) #initialize sCapture
A = np.array(([1,1],[0,1]))
B = np.array(([0], [1]))
eps = 1e-10
i = 0

while (np.linalg.norm(s0[0]) > sf[0] and np.linalg.norm(s0[1]) > sf[0]):
    sStep,uStep = convex_problem(Q,R,P,s0,u0,N,uB,sB,NMax)
    #solve the full convex problem at each timestep

    if np.any(uStep == None): #check feasibility
        print("Infeasible!")
        break

    if i == 0:
        s = s0 #initial condition
        u = uStep[0] #initial optimal control
        sCapture[i] = sStep #first layer
    else:
        sStep3d = np.expand_dims(sStep, axis=0)
        #to make it possible to concatenate
        sCapture = np.concatenate((sCapture,sStep3d)) #add a new layer
        u = np.vstack((u, uStep[0]))
        #define the action using the optimal action of the convex problem

    sNew = A@sStep[0] + B @uStep[0]
    #define the state using the solution of the convex problem
    s = np.vstack((s,sNew))


    s0 = s[i+1]
    i += 1




return s,u,sCapture

def convex_problem(Q,R,P,s0,u0,N,uB, xB, NMax):
cost_terms = []
constraints = []
s =[]
u = []

s = cvx.Variable((N+1, 2))
u = cvx.Variable((N, 1))

A = np.array(([1,1],[0,1]))
B = np.array(([0], [1]))

for t in range(N+1):
    if t < N:
        cost_terms.append(0.5*cvx.quad_form((s[t]), Q))
        #defining cost terms
        cost_terms.append(0.5*cvx.quad_form(u[t], R))
    else:
        cost_terms.append(0.5*cvx.quad_form((s[t]), P)) #terminal cost

    if t == 0:
        constraints.append(s[t] == s0)

    if N != NMax and t == 0: #excluding the first timestep
        constraints.append(u[0] == u0)
        #use the optimal control from the previous timestep in this step

    if t < N:
        constraints.append(u[t] >= -uB) #lower bound control constraint
        constraints.append(u[t] <= uB) #upper bound control constraint


    if t < N:
        constraints.append(s[t+1] == A @ s[t] + B @ u[t])
        #dynamics constraint
        constraints.append(s[t] >= -xB) #lower bound state constraint
        constraints.append(s[t] <= xB) #upper bound state constraint
```

```
costSum = cvx.sum(cost_terms)
objective = cvx.Minimize(costSum)
prob = cvx.Problem(objective, constraints)
prob.solve()
print(prob.status)
print(prob.value)

s = s.value
u = u.value
return s,u


 if __name__ == '__main__':
s0 = np.array([-4.5,2])
sf = np.array([1e-10, 1e-10])

# specify cost function
P = 1.*np.eye(2)                              # terminal state cost matrix
Q = 1*np.eye(2)                               # state cost matrix
R = 10*np.eye(1)                            # control cost matrix

#constraints
uB = 0.5                          # control effort lower bound
sB = 5.                            # control effort upper bound

# solve swing-up with scp
print('Computing MPC solution ... ', end='')

N = 3
s1,u1,sCapture1 = MPC(Q,R,P,s0,sf,N,uB,sB)
print('done!')

s0 = np.array([-4.5, 3])

s2,u2,sCapture2 = MPC(Q,R,P,s0,sf,N,uB,sB)

print('Simulating ...')

# Plot
fig, ax = plt.subplots()
ax.grid()    \caption{Simulated Trajectory for values of $N$ ranging from $2$
    \label{figure:2f}
ax.plot(s1[:,0], s1[:,1])
ax.plot(s1[:,0], s1[:,1], 'bo')
ax.plot(sCapture1[0,:,0], sCapture1[0,:,1], 'k--')
ax.plot(sCapture1[1,:,0], sCapture1[1,:,1], 'k--')
ax.plot(sCapture1[2,:,0], sCapture1[2,:,1], 'k--')
ax.plot(sCapture1[0,:,0], sCapture1[0,:,1],'ko')
ax.plot(sCapture1[1,:,0], sCapture1[1,:,1],'ko')
ax.plot(sCapture1[2,:,0], sCapture1[2,:,1],'ko')

ax.plot(s2[:,0], s2[:,1])
ax.plot(s2[:,0], s2[:,1],'bo')
ax.plot(sCapture2[0,:,0], sCapture2[0,:,1], 'k--')
ax.plot(sCapture2[1,:,0], sCapture2[1,:,1], 'k--')
ax.plot(sCapture2[0,:,0], sCapture2[0,:,1], 'ko')
ax.plot(sCapture2[1,:,0], sCapture2[1,:,1], 'ko')



plt.show()
```

Problem 3. MPC Terminal Invariant and Stability

Part (a)

As noted in the course notes (and in lecture), the choice of X_f and P is critical in ensuring persistent feasibility. For the given weights, I would define P as P_∞ (which is the solution to the finite horizon Riccati equation). I would define X_f as maximal positive invariant set. I would compute X_f using tools from computation geometry (mpt3) or by iterating from values of X that satisfy a constraint.

Part (b)

To prove that M is open loop invariant, we essential need to prove that $M - A^T M A$ is positive semi-definite. This can be verified analytically using Sylvester's criterion. Sylvester's criterion states that given a 2x2 matrix B, if

$$B(1,1) >= 0$$

$$Det(B) >= 0$$

$$B(2,2) >= 0$$

Then M is positive semi-definite. This is easily shown for $B = M - A^T M A$.

$$B(1,1) = 0.0039 >= 0$$

$$Det(B) = 3.065e - 06 >= 0$$

$$B(2,2) = 0.09335 >= 0$$

Hence, we can comfortably say that M is open-loop invariant

Part (c)

The simulated trajectory and optimal control action for the MPC problem with the terminal constraint and the terminal cost is shown below. Note that in my code, I tried to drive the system to zero.

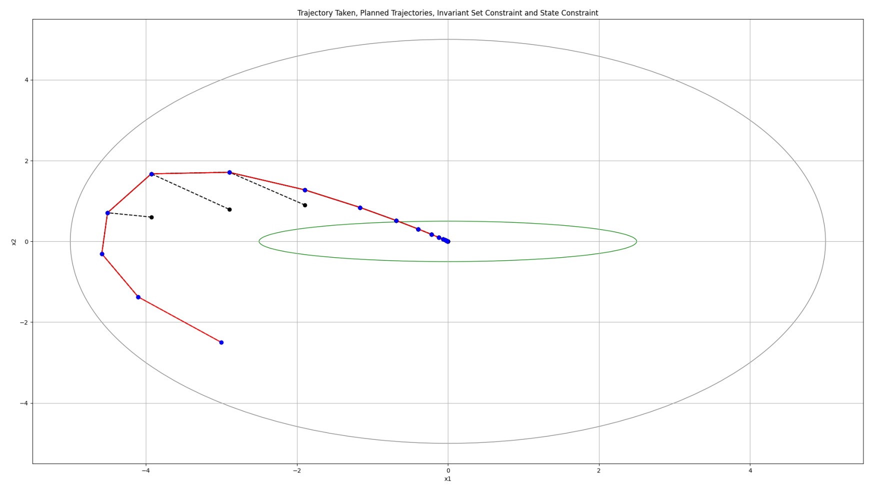

Figure 8: Simulated Trajectory for MPC Problem with Terminal Constraint and Cost

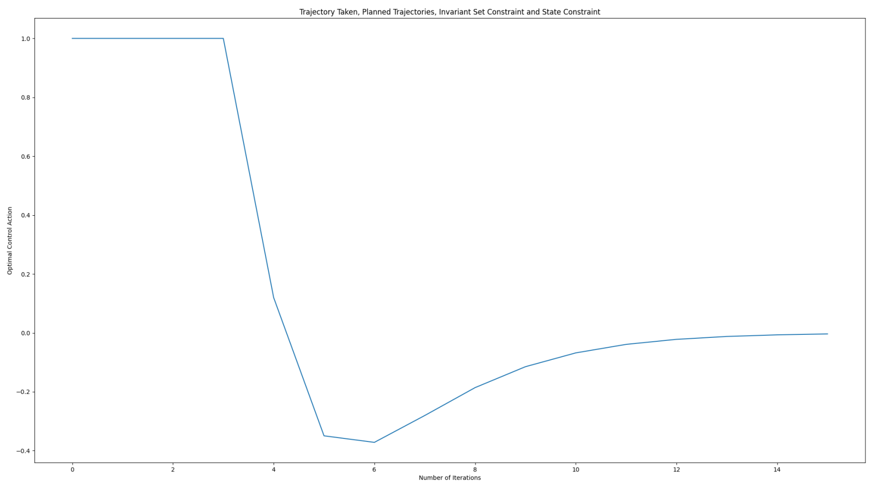

Figure 9: Optimal Control for MPC Problem with Terminal Constraint and Cost

The simulated trajectory and optimal control action for the MPC problem with only the terminal cost is shown below. Note that in my code, I tried to drive the system to zero. The optimal control is barely affected but the predicted trajectories are very different. Note that the realized trajectories are identical

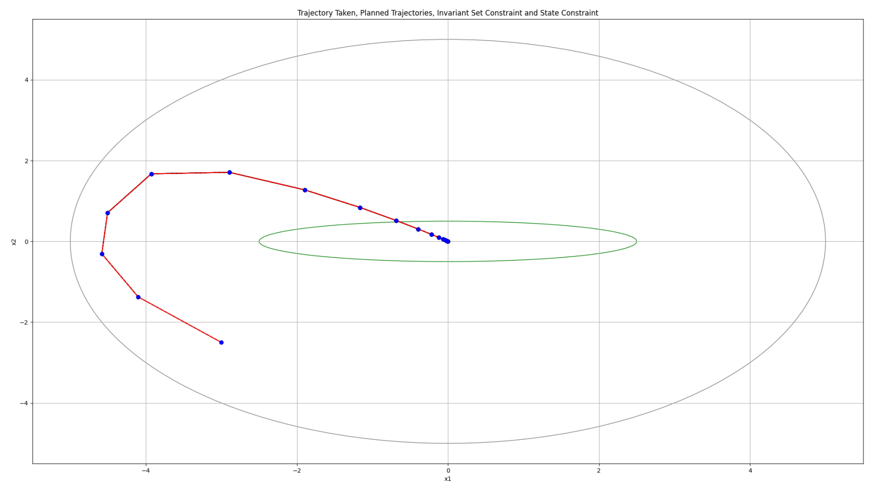

Figure 10: Simulated Trajectory for MPC Problem with Only Terminal Cost

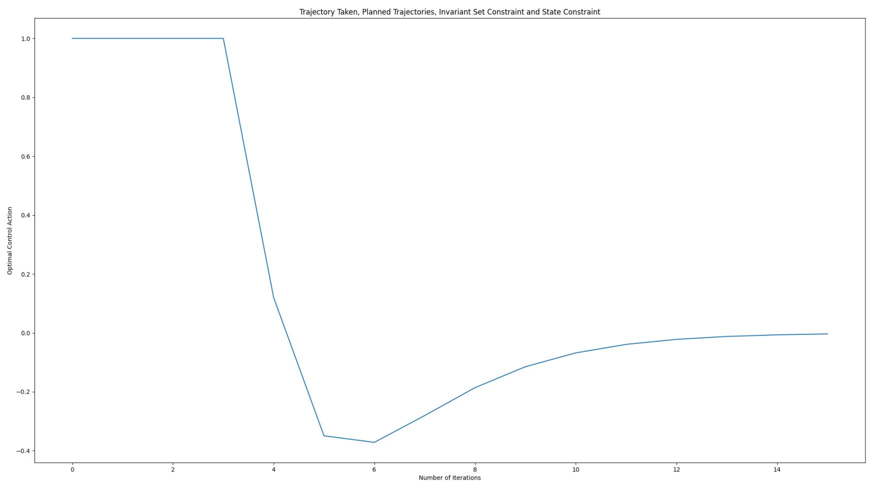

Figure 11: Optimal Control for MPC Problem with Only Terminal Cost

The simulated trajectory and optimal control action for the MPC problem with only the terminal constraint is shown below. Note that in my code, I tried to drive the system to zero. Note the change in the predicted trajectories and the slight change in the shape of the optima control. Note that the realized trajecctory is unchanged

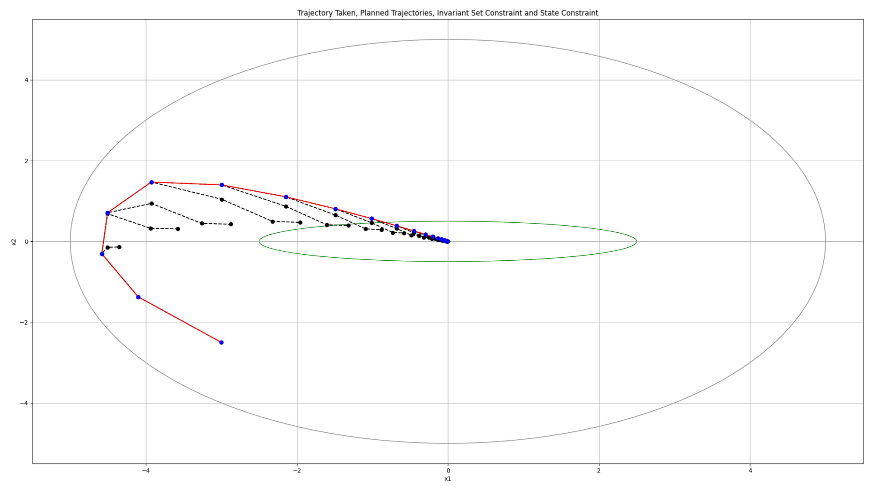

Figure 12: Simulated Trajectory for MPC Problem with Only Terminal Constraint

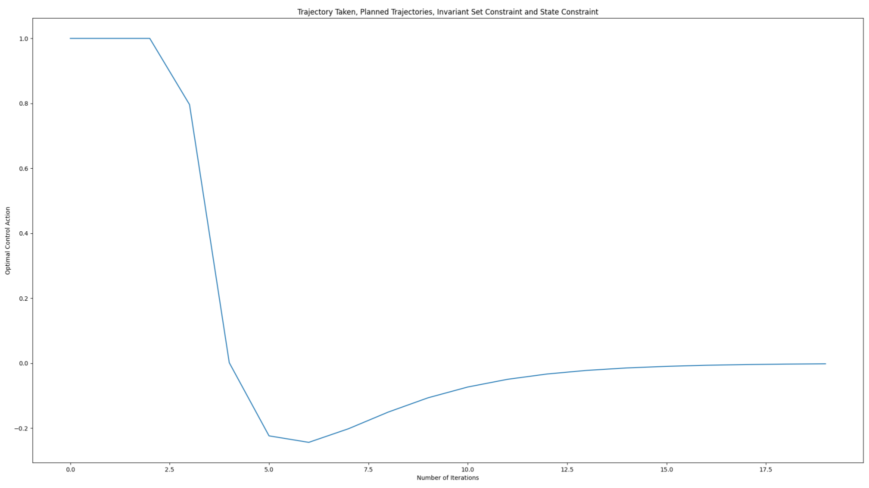

Figure 13: Optimal Control for MPC Problem with Only Terminal Constraint

The simulated trajectory and optimal control action for the MPC problem with only the terminal constraint is shown below. Note that in my code, I tried to drive the system to zero. Note the change in the predicted trajectories and the slight change in the shape of the optima control. Note that the realized trajecctory is unchanged

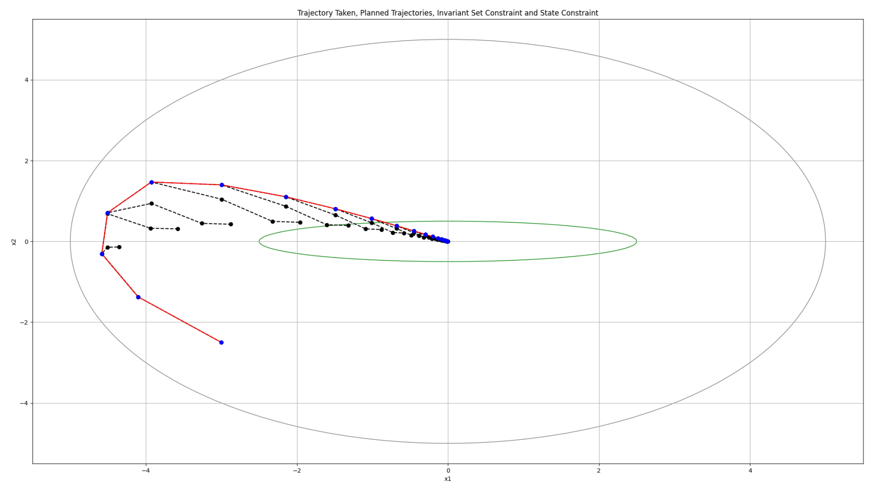

Figure 14: Simulated Trajectory for MPC Problem with No Terminal Constraints or Costs

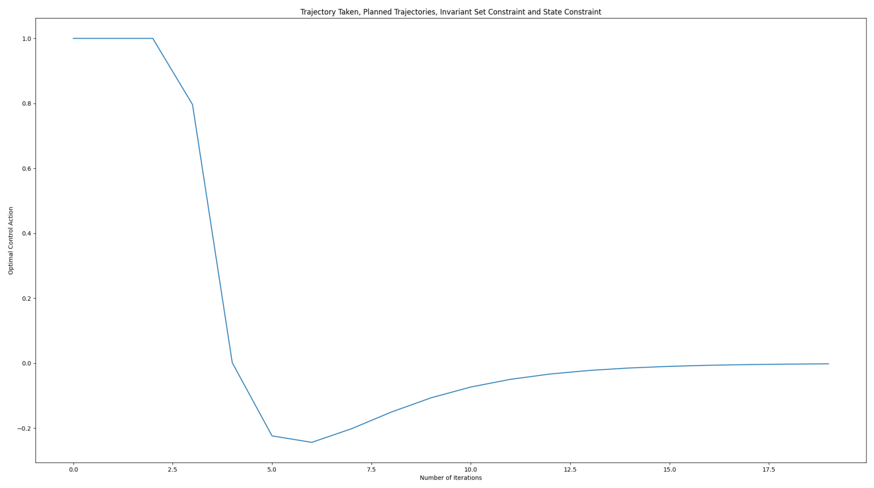

Figure 15: Optimal Control for MPC Problem with No Terminal Constraints or Costs

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patch
import cvxpy as cvx
from numpy.linalg import inv
from numpy import linalg as LA
```

```
def MPC(Q,M,R,P,s0,sf, N,uB,sB):
#run for all time steps
u0 = 0
NMax = N
s = np.zeros((1,2)) #initialize s
u = np.zeros((1,1)) #initialize u
sCapture = np.zeros((1,N+1,2)) #initialize sCapture
A = np.array(([0.95,0.5],[0,0.95]))
B = np.array(([0], [1]))
i = 0

while (np.linalg.norm(s0[0]) > 0.01 or np.linalg.norm(s0[1]) > 0.01):
sStep,uStep = convex_problem(Q,M,R,P,s0,sf,u0,N,uB,sB,NMax)
#solve the full convex problem at each timestep

if np.any(uStep == None): #check feasibility
    print("Infeasible!")
    break

if i == 0:
    s = s0 #initial condition
    u = uStep[0] #initial optimal control
    sCapture[i] = sStep #first layer
else:
    sStep3d = np.expand_dims(sStep, axis=0)
    #to make it possible to concatenate
    sCapture = np.concatenate((sCapture,sStep3d)) #add a new layer
    u = np.vstack((u, uStep[0]))
    #define the action using the optimal action of the convex problem

sNew = A@sStep[0] + B @uStep[0]
#define the state using the solution of the convex problem
s = np.vstack((s,sNew))



s0 = s[i+1]
i += 1





return s,u,sCapture

def convex_problem(Q,M,R,P,s0,sf,u0,N,uB, xB, NMax):
cost_terms = []
```

```
constraints = []
s =[]
u = []

s = cvx.Variable((N+1, 2))
u = cvx.Variable((N, 1))

A = np.array(([0.95,0.5],[0,0.95]))
B = np.array(([0], [1]))

for t in range(N+1):
    if t < N:
        cost_terms.append(0.5*cvx.quad_form((s[t]), Q))
        #defining cost terms
        cost_terms.append(0.5*cvx.quad_form(u[t], R))
    # else:
    #     cost_terms.append(0.5*cvx.quad_form((s[t]), P)) #terminal cost

    if t == 0:
        constraints.append(s[t] == s0)

    if N != NMax and t == 0: #excluding the first timestep
        constraints.append(u[0] == u0)
        #use the optimal control from the previous timestep in this step

    if t < N:
        constraints.append(u[t] >= -uB) #lower bound control constraint
        constraints.append(u[t] <= uB) #upper bound control constraint


    if t < N:
        constraints.append(s[t+1] == A @ s[t] + B @ u[t])
        #dynamics constraint
        constraints.append(s[t] >= -xB) #lower bound state constraint
        constraints.append(s[t] <= xB) #upper bound state constraint
    # else:
    #     constraints.append(cvx.quad_form(s[t],M) <= 1)




costSum = cvx.sum(cost_terms)
objective = cvx.Minimize(costSum)
prob = cvx.Problem(objective, constraints)
```

```
prob.solve()
print(prob.status)
print(prob.value)

s = s.value
u = u.value

#############################################################################################
return s,u


def Riccati(R, A, B, Q, PkOld):
Kk = -inv(R + B.T@PkOld@B)@B.T@PkOld@A
Pk = Q + A.T@PkOld@(A + B@Kk)

while LA.norm(Pk - PkOld) > 1e-4:
PkOld = Pk
Kk = -inv(R + B.T@PkOld@B)@B.T@PkOld@A
Pk = Q + A.T@PkOld@(A + B@Kk)

return Pk



if __name__ == '__main__':
sf = np.array([5, 1])
s0 = np.array([-3.0, -2.5])

A = np.array(([0.95,0.5],[0,0.95]))
B = np.array(([0], [1]))

M = np.array(([0.04,0],[0,1.06]))
#constraints
uB = 1.                    # control effort lower bound
sB = 5.                       # control effort upper bound

Q = 1*np.eye(2)                          # state cost matrix
R = 1*np.eye(1)                       # control cost matrix

initP = np.zeros((2,2))
P = Riccati(R,A,B,Q,initP)                      # terminal state cost matrix

# solve swing-up with scp
print('Computing MPC solution ... ', end='')

N = 4
```

```
s1,u1,sCapture1 = MPC(Q,M,R,P,s0,sf,N,uB,sB)
fig, ax = plt.subplots()
ax.grid()

ell1 = patch.Ellipse((0.,0.), 5, 1, color='green', fill=False)

circ1 = plt.Circle((0,0), 5, color='grey', fill=False)
ax.add_patch(circ1)
ax.add_patch(ell1)



for i in range(sCapture1.shape[0]):
#ax.plot(sCapture1[i], 'k--')
ax.plot(sCapture1[i,:,0], sCapture1[i,:,1], 'k--')
ax.plot(sCapture1[i,:,0], sCapture1[i,:,1],'ko')

ax.plot(s1[:,0], s1[:,1], 'r')
ax.plot(s1[:,0], s1[:,1], 'bo')
ax.set_title(
"Trajectory Taken, Planned Trajectories ,
Invariant Set Constraint and State Constraint")
ax.set_xlabel("x1")
ax.set_ylabel("x2")

fig2, ax2 = plt.subplots()
ax2.plot(u1)
ax2.set_title("Trajectory Taken, Planned Trajectories ,
Invariant Set Constraint and State Constraint")
ax2.set_xlabel("Number of Iterations")
ax2.set_ylabel("Optimal Control Action")


plt.show()
print('done!')
```

Problem 4. Introduction to Reinforcement Learning

Part (a)

I implemented a discounted form of the Riccati equation. After multiple simulations, I noticed that the average cost hovers around 100/120.

Part (b)

I implemented least-squares regression for the model and the cost function. I had a lot of trouble getting my Ricatti equation to converge but I fixed this by introducing a parameter to exit the recursion after 5000 iterations. The norm of the difference between the resulting policy and the optimal policy is shown below. The resulting cost per episode is also shown below. The plots for both the stochastic dynamics problem and the deterministic dynamics problem are attached.

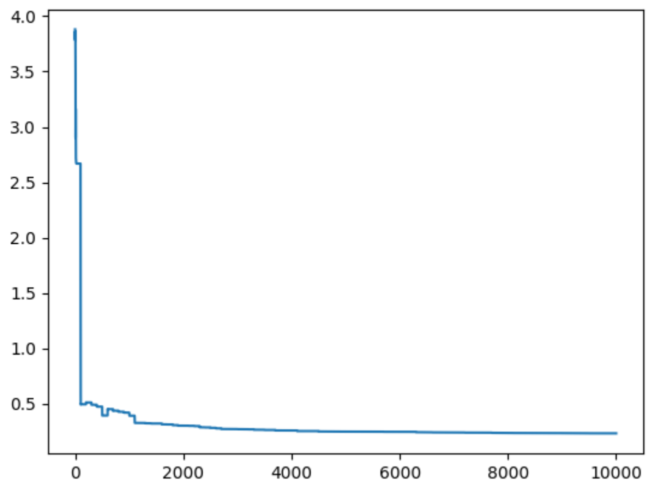

Figure 16: Norm of the Difference Between Optimal Policy and Model-Based Learning Policy with Deterministic Dynamics

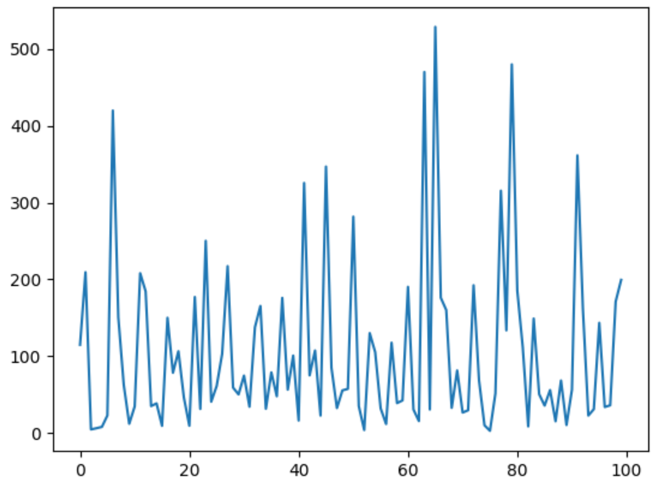

Figure 17: Cost of Model-Based Learning Policy with Deterministic Dynamics

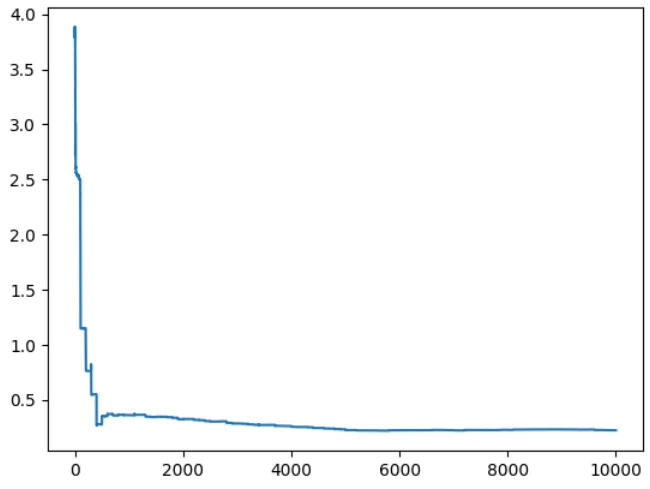

Figure 18: Norm of the Difference Between Optimal Policy and Model-Based Learning Policy with Stochastic Dynamics

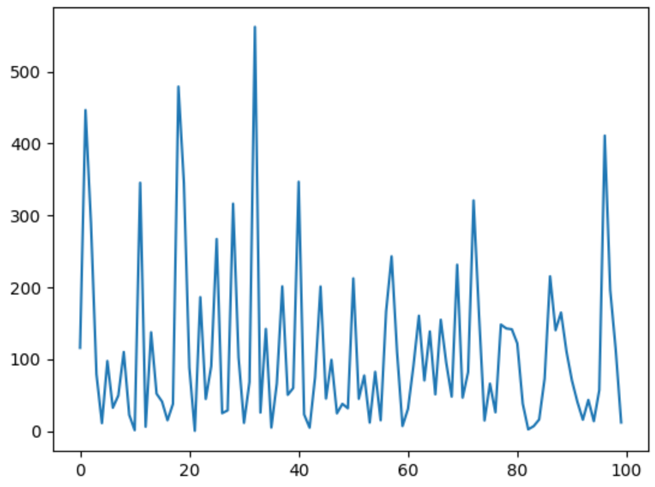

Figure 19: Cost of Model-Based Learning Policy with Stochastic Dynamics

Part (c)

I implemented the policy iteration scheme described in the linked paper. I had to set my $P_0 = 1e8$ to observe a semblance of convergence. I also had a lot of trouble observing convergence for this problem The norm of the difference between the resulting policy and the optimal policy is shown below. The resulting cost per episode is also shown below. The plots for both the stochastic dynamics problem and the deterministic dynamics problem are attached.

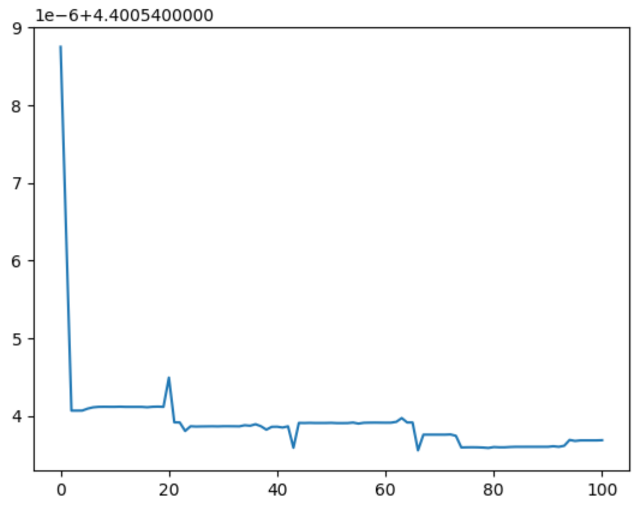

Figure 20: Norm of the Difference Between Optimal Policy and Model-Free Policy Iteration with Deterministic Dynamics

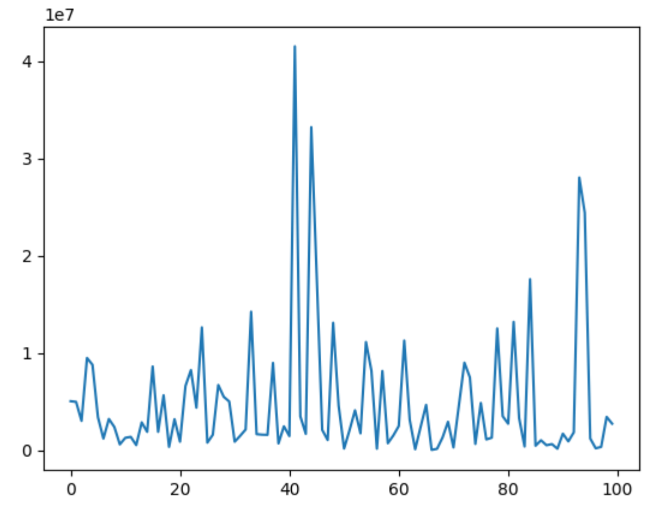

Figure 21: Cost of Model-Free Policy Iteration with Deterministic Dynamics

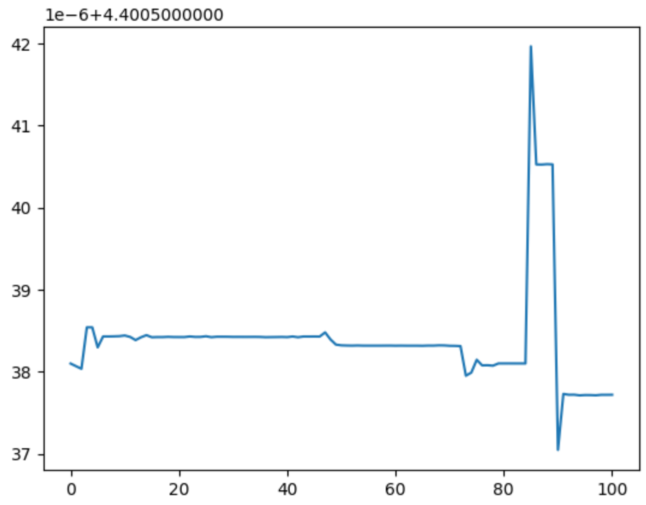

Figure 22: Norm of the Difference Between Optimal Policy and Model-Free Policy Iteration with Stochastic Dynamics

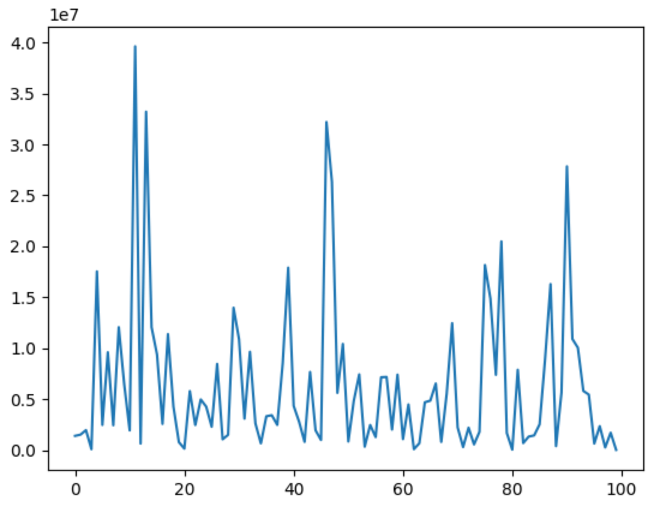

Figure 23: Cost of Model-Free Policy Iteration with Stochastic Dynamics

Part (d)

I implemented the Monte-Carlo policy gradient scheme described in the lecture. I maintained a running average of the average control. I also had to use the functional form of the multivariate Gaussian normal to compute the gradient of the policy. The expression of the gradient took the form of the hint provided by Spencer (on Piazza). I had a lot of trouble observing convergence for the problem with stochastic dynamics. The norm of the difference between the resulting policy and the optimal policy is shown below. The resulting cost per episode is also shown below. The plots for both the stochastic dynamics problem and the deterministic dynamics problem are attached.

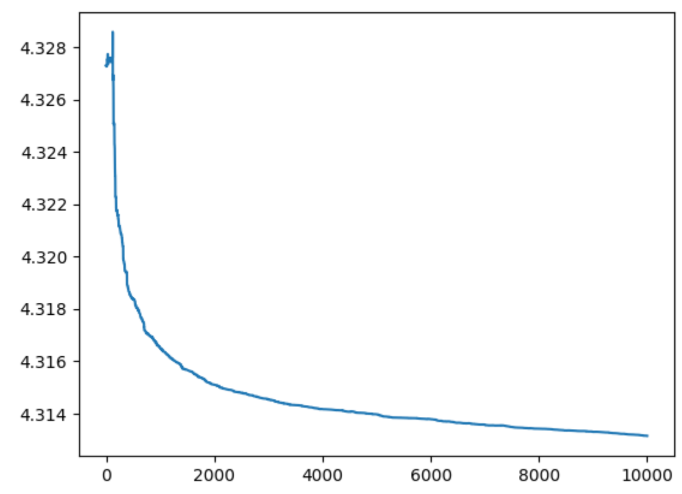

Figure 24: Norm of the Difference Between Optimal Policy and Model-Free Monte-Carlo Policy Gradient with Deterministic Dynamics

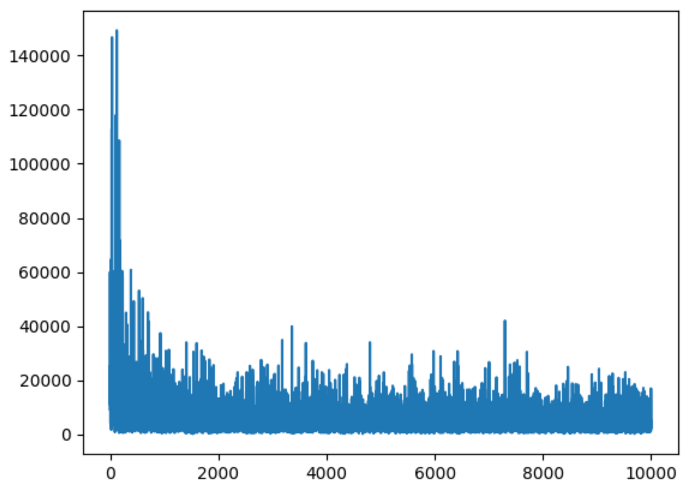

Figure 25: Cost of Model-Free Monte-Carlo Policy Gradient with Deterministic Dynamics

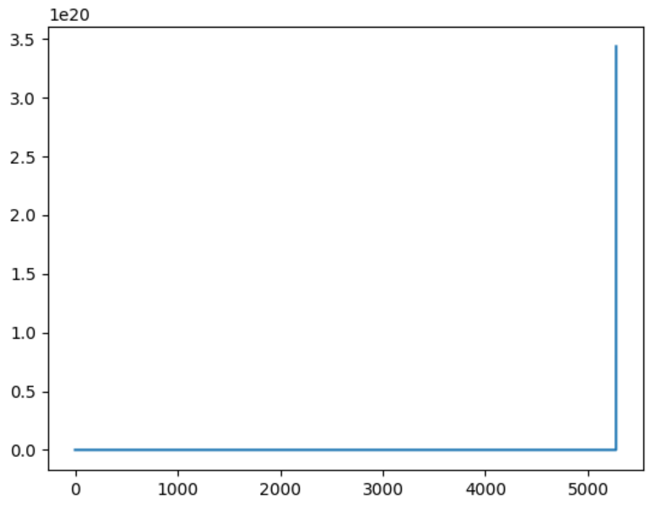

Figure 26: Norm of the Difference Between Optimal Policy and Model-Free Policy Iteration with Stochastic Dynamics

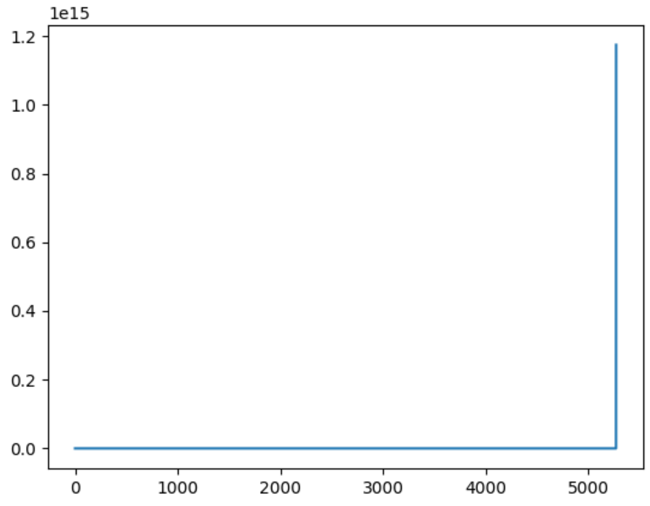

Figure 27: Cost of Model-Free Monte-Carlo Policy Gradient with Stochastic Dynamics

Part (e)

The baseline method (Riccati's equation) required perfect knowledge of the problem's dynamics the cost function. While the baseline results in optimal performance, the amount of problem-specific information required makes it somewhat unrealistic. The method in Part (b) (Model Based Regression) requires knowledge of the state and the cost at each timestep. With only state and cost information, it constructs a model of the dynamics and the cost function. This method provides the best performance of the learning-based methods. The method in part (c) (Model Free Policy Iteration) also requires knowledge of the state and the cost at each timestep.I avoids creating a model of the dynamics and cost but it requires a feasible initial controller to ensure convergence. While the zero matrix is a trivial feasible controller, we need problem specific information to generate nontrivial feasible initial controller. The performance of this method is worse than that of the model based method. The method in part (d) (Monte-Carlo Policy Gradient) requires the least information of all the learning based method. It performs surprisingly well but it struggles when used on systems with stochastic dynamics

```
from model import dynamics, cost
import numpy as np
from numpy.linalg import inv
from numpy import linalg as LA

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

# Riccati recursion
def Riccati(A,B,Q,R):
    PkOld = np.zeros((len(A), len(A)))

    L = -inv(R + B.T@PkOld@B)@B.T@PkOld@A
    Pk = Q + A.T@PkOld@(A + B@L)

    while LA.norm(Pk - PkOld) > 1e-8:
        PkOld = Pk
        L = -inv(R + B.T@PkOld@B)@B.T@PkOld@A
        Pk = Q + A.T@PkOld@(A + B@L)
        #maybe update L again after


    return L,Pk


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

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

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))

from numpy.core.shape_base import block
from model import dynamics, cost
import numpy as np
from numpy.linalg import inv
from numpy import linalg as LA
from scipy.linalg import block_diag
import matplotlib.pyplot as plt




stochastic_dynamics = True # 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

def Riccati(A,B,Q,R,eps):
PkOld = np.zeros((len(A), len(A)))
```

```
ricCount = 0

L = -LA.inv(R + B.T@PkOld@B)@B.T@PkOld@A
Pk = gamma*(Q + A.T@PkOld@(A + B@L))

#while LA.norm(Pk - PkOld) > eps:
while (not np.allclose(Pk, PkOld)) and (ricCount < 5000):
print(LA.norm(Pk - PkOld))
PkOld = Pk
L = -LA.inv(R + B.T@PkOld@B)@B.T@PkOld@A
Pk = gamma*(Q + A.T@PkOld@(A + B@L))
ricCount += 1

print("done!")
return L,Pk


total_costs = []

Aknown = dynfun.A
Bknown = dynfun.B
Qknown = costfun.Q
Rknown = costfun.R

eps = 1e-6

L_star, P_star = Riccati(Aknown,Bknown,Qknown,Rknown,eps)

timeCount = 0

np.random.seed(0)
A = np.random.rand(4,4) #maybe use a seed?
B = np.random.rand(4,2)
Q = np.eye(4) #defining state cost
R = np.eye(2) #defining control cost


cDyn0 = np.vstack((A.T,B.T)) #make psd? 6x4
cCost0 = np.zeros((20,1)) #20x1


pC0 = np.eye(20) #initialize P for cost regression
pX0 = np.eye(6) #initialize P for dynamics regression
timeCount = 0
```

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

x = dynfun.reset() #new initial x for next training episode

#qSmall = Q.reshape(16,1)
qSmall = Q.ravel() #review
#rSmall = R.reshape(4,1)
rSmall = R.ravel()

cCost0 = np.concatenate((qSmall, rSmall)).reshape((20,1))
#wrap it into a 20x1


for t in range(T): #run the episode

# for i in range(len(x)): #put x in form requested by cost problem
#     for j in range(len(x)):
#         if i == 0 and j == 0:
#             xr = x[i]*x[j]
#         else:
#             xComp = x[i]*x[j]
#             xr = np.vstack((xr, xComp))

xr = np.outer(x,x).ravel()


# TODO compute policy
eps = 1e-3
L,P = Riccati(A,B,Q,R,eps)

# TODO compute action
u = L@x
count = 0

# for i in range(len(u)): #put u in form requested by cost problem
#     for j in range(len(u)):
#         if i == 0 and j == 0:
#             up = u[i]*u[j]
#         else:
#             uComp = u[i]*u[j]
#             up = np.vstack((up, uComp))

up = np.outer(u,u).ravel()
```

```
# get reward
zC = np.concatenate((xr,up)) #20x0
c = costfun.evaluate(x,u)
print("Cost at timestep: ", c)
costs.append((gamma**t)*c)

pC = pC0 - ((pC0 @ np.outer(zC, zC) @ pC0)/(1 + zC.T @ pC0 @ zC))

cCost = cCost0 + np.outer(pC0 @ zC,c
  - cCost0.T @ zC)/(1 + zC.T @ pC0 @ zC) #should return a 20x1

pC0 = np.copy(pC) #update
cCost0 = np.copy(cCost) #update


# print("Old Q:", Q)
# print("Old R:", R)


qSmallNew = cCost[0:16]
rSmallNew = cCost[16:20]

Q = qSmallNew.reshape(4,4)
R = rSmallNew.reshape(2,2)

# print("New Q:", Q)
# print("New R:", R)

#updated Q and R
#now update dynamics

# dynamics step
xp = dynfun.step(u) #x_t+1


#xp = np.reshape(xp, (1,-1))
# x = np.reshape(x, (-1,1)) #make 2d
# u = np.reshape(u, (-1,1)) #ditto

zX = np.concatenate((x,u)) #a 6x1


#C.T is a 4x6 for dynamics regression so CDyn is a 6x4
#CDyn0 is a 6x4
pX = pX0 - (pX0 @ np.outer(zX,zX) @ pX0)/(1 + zX.T @ pX0 @ zX) #6x6
```

```
cDyn = cDyn0 + np.outer(pX0 @ zX,xp -cDyn0.T @ zX)/(1 + zX.T @ pX0 @zX)
#should return a 6x4

cDyn0 = np.copy(cDyn) #cDyn0 has to be 4x6
pX0 = np.copy(pX) #update pX0

# print("Old A:", A)
# print("Old B:", B)

#Update A and B
A = cDyn[:4 ,:].T
B = cDyn[4: ,:].T #has to be 4x2

# print("New A:", A)
# print("New B:", B)



x = xp.copy() #update x


if t == 0 and n == 0:
LnormVal = LA.norm(L_star - L)
normTime = np.array([timeCount])
timeCount += 1
else:
LnormVal = np.vstack((LnormVal, (LA.norm(L_star - L))))
timeCount += 1
normTime = np.vstack((normTime, np.array([timeCount])))



total_costs.append(sum(costs))
print("Total␣Cost:␣", sum(costs))

if n == 0:
costHold = np.array([sum(costs)])
else:
costHold = np.vstack((costHold, np.array([sum(costs)])))

episodes = np.arange(N)

fig1 , ax1 = plt.subplots()
ax1.plot(normTime, LnormVal)
```

```
fig2 , ax2 = plt.subplots()
ax2.plot(episodes , costHold)

plt.show()

from model import dynamics , cost
import numpy as np
from numpy import linalg as LA
import matplotlib.pyplot as plt

stochastic_dynamics = True # set to True for stochastic dynamics
dynfun = dynamics(stochastic=stochastic_dynamics)
costfun = cost()
gamma = 1

def Riccati(A,B,Q,R,eps):
PkOld = np.zeros((len(A), len(A)))
ricCount = 0

L = -LA.inv(R + B.T@PkOld@B)@B.T@PkOld@A
Pk = gamma*(Q + A.T@PkOld@(A + B@L))

#while LA.norm(Pk - PkOld) > eps:
while (not np.allclose(Pk, PkOld)) and (ricCount < 5000):
print(LA.norm(Pk - PkOld))
PkOld = Pk
L = -LA.inv(R + B.T@PkOld@B)@B.T@PkOld@A
Pk = gamma*(Q + A.T@PkOld@(A + B@L))
ricCount += 1

print("done!")
return L,Pk


Aknown = dynfun.A
Bknown = dynfun.B
Qknown = costfun.Q
Rknown = costfun.R

eps = 1e-6

L_star , P_star = Riccati(Aknown,Bknown,Qknown,Rknown,eps)


T = 100
```

```
N = 100

total_costs = []

#intialize the value function
np.random.seed(0)
Q = np.random.rand(6,6) #maybe use a seed?
Q = 0.5*(Q + Q.T) #make symmetric

pQ0 = 1e8*np.eye(36) #trying to learn 36 parameters
Q22 = Q[4:,4:] #a P.D 2x2
Q21 = Q[4:,0:4]

L = np.zeros((2,4))
timeCount = 0

cQ0 = Q.ravel().reshape(36,1) #check shape should be 36x1 c matrix

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

pQ0 = 1e8*np.eye(36) #trying to learn 36 parameters

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

# TODO compute action
u = -L@x + np.random.normal(0,1,size=(2))

xu = np.concatenate((x,u))
xu_bar = np.outer(xu,xu).ravel() #36x1 z matrix

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

# dynamics step
xp = dynfun.step(u)

# TODO recursive least squares polict evaluation step
pQ = pQ0 - ((pQ0 @ np.outer(xu_bar, xu_bar)
@ pQ0)/(1 + xu_bar.T @ pQ0 @ xu_bar))

cQ = cQ0 + np.outer(pQ0 @ xu_bar,c - cQ0.T @
          xu_bar)/(1 + xu_bar.T @ pQ0 @ xu_bar) #should return a 36x1
```

```
pQ0 = np.copy(pQ)
cQ0 = np.copy(cQ)


x = xp.copy()


# TODO policy improvement step
Q = cQ.reshape(6,6)
Q22 = Q[4:,4:]
Q21 = Q[4:,0:4]
L = -LA.inv(Q22) @ Q21

total_costs.append(sum(costs))

if n == 0:
LnormVal = LA.norm(L_star + L)
normTime = np.array([timeCount])
costHold = np.array([sum(costs)])
timeCount += 1
else:
LnormVal = np.vstack((LnormVal, (LA.norm(L_star + L))))
timeCount += 1
normTime = np.vstack((normTime, np.array([timeCount])))
costHold = np.vstack((costHold, np.array([sum(costs)])))

episodes = np.arange(N)
fig1, ax1 = plt.subplots()
ax1.plot(normTime, LnormVal)

fig2, ax2 = plt.subplots()
ax2.plot(episodes, costHold)

plt.show()

from model import dynamics, cost
import numpy as np
from numpy import linalg as LA
import matplotlib.pyplot as plt

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

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

total_costs = []

def Riccati(A,B,Q,R,eps):
PkOld = np.zeros((len(A), len(A)))
ricCount = 0

L = -LA.inv(R + B.T@PkOld@B)@B.T@PkOld@A
Pk = gamma*(Q + A.T@PkOld@(A + B@L))

while (not np.allclose(Pk, PkOld)) and (ricCount < 5000):
print(LA.norm(Pk - PkOld))
PkOld = Pk
L = -LA.inv(R + B.T@PkOld@B)@B.T@PkOld@A
Pk = gamma*(Q + A.T@PkOld@(A + B@L))
ricCount += 1

print("done!")
return L,Pk



Aknown = dynfun.A
Bknown = dynfun.B
Qknown = costfun.Q
Rknown = costfun.R

eps = 1e-6

L_star, P_star = Riccati(Aknown,Bknown,Qknown,Rknown,eps)

L = np.zeros((2,4)) #initialize policy
mu = np.zeros((2)) #initialize mu
sigma = 0.1*np.eye(2) #initialize sigma using the identity matrix
alpha = 1e-13
G = 0 #initialize the reward counter
timeCount = 0

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

G = 0 #maybe reset G for every episode
```

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

u = np.random.multivariate_normal(L@x, sigma)

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

G += c #update G. Maybe move outside loop
costs.append((gamma**t)*c)

# dynamics step
xp = dynfun.step(u)


x = xp.copy()
mu = 0.5*(mu + u)

# TODO update policy
gradLog = ((LA.inv(sigma) + LA.inv(sigma).T)@(L@x
    - mu)).reshape(2,1)@(x.reshape(4,1)).T #check dimensions of x.T

L += alpha*G*gradLog
#mu = 0.5*(mu + L@x) #update mu


total_costs.append(sum(costs))

if n == 0:
LnormVal = LA.norm(L_star - L)
normTime = np.array([timeCount])
costHold = np.array([sum(costs)])
timeCount += 1
else:
LnormVal = np.vstack((LnormVal, (LA.norm(L_star - L))))
timeCount += 1
normTime = np.vstack((normTime, np.array([timeCount])))
costHold = np.vstack((costHold, np.array([sum(costs)])))

episodes = np.arange(N)
fig1, ax1 = plt.subplots()
ax1.plot(normTime, LnormVal)

fig2, ax2 = plt.subplots()
ax2.plot(episodes, costHold)
```

```
plt.show()
```