HW #2 CS159

Zhuofang Li, Greg Stroot, Francesca-Zhoufan Li
2136043, 0002135380, 2136030

Due on April 15 2021

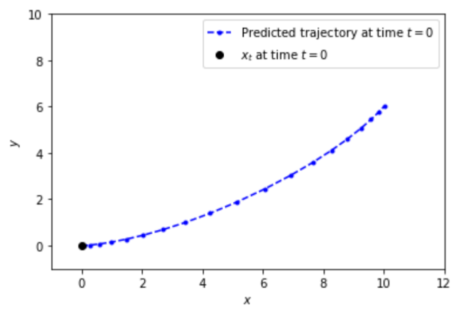
Instructions

Please LaTeX your solutions using the attached template. Fill in each section with your answers and please do not change the order of sections and subsection.

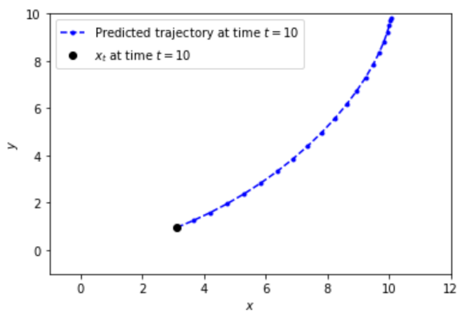
You need to submit both a PDF and your code.

1 Problem 1

1.1 Nonlinear MPC (1 points)



(a) t=0



(b) t=10

Figure 1: 1.1 Trajectory

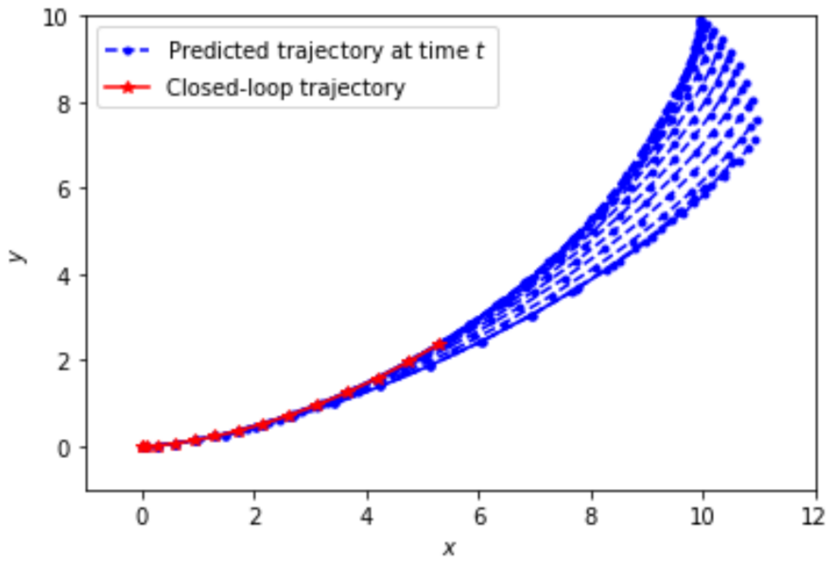
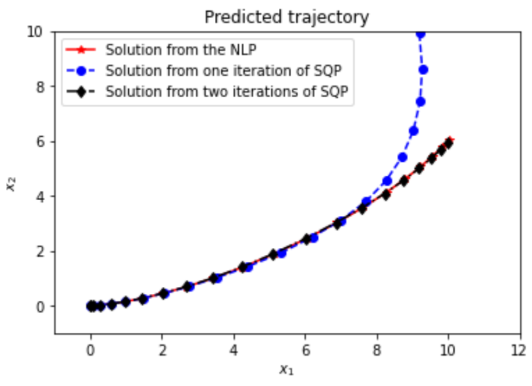


Figure 2: Trajectory 1.1

The closed-loop trajectory overlaps with the predicted trajectory.It overlaps more for larger t. It's expected since we're solving the original NLP problem and the closed-loop system is stable and accurate for non-linear systems. Also we keep updating and self-correcting so the match is better for larger t.

1.2 Sequential Quadratic Programming (3 points)

```
    def buildCost(self):
        listQ = [self.Q] * (self.N-1)
        barQ = linalg.block_diag(linalg.block_diag(*listQ), self.Qf)

        listTotR = [self.R] * (self.N)
        barR = linalg.block_diag(*listTotR)

        H = linalg.block_diag(barQ, barR)

        goal = self.goal
        # Hint: First construct a vector z_{goal} using the goal state and then leverage the matrix H

#        q= np.matmul(np.tile(goal,self.N),barQ).T
#        q=-2*np.concatenate([q, np.zeros(self.d*self.N)])
        q= -2*np.matmul(np.concatenate([np.tile(goal,self.N),np.zeros(self.d*self.N)]),H).T

        if self.printLevel >= 2:
            print("H: ")
            print(H)
            print("q: ", q)

        self.q = q
        self.H = sparse.csc_matrix(2 * H)  #  Need to multiply by two because CVX considers 1/2 in front of quadratic cost

    def buildEqConstr(self):
        # Hint 1: The equality constraint is: [Gx, Gu]*z = E * x(t) + C
        # Hint 2: Write on paper the matrices Gx and Gu to see how these matrices are constructed
        Gx = np.eye(self.n * self.N )
        Gu = np.zeros((self.n * self.N, self.d*self.N) )

        self.C = []
        E_eq = np.zeros((Gx.shape[0], self.n))
        for k in range(0, self.N):
            A, B, C = self.buildLinearizedMatrices(self.xGuess[k], self.uGuess[k])

            if k == 0:
                E_eq[0:self.n, :] = A
            else:
                Gx[((k)*self.n):(self.n*(k+1)), (k-1)*self.n:((k)*self.n)] = -A
            Gu[k*self.n:self.n*(k+1), (k)*self.d:(1+k)*self.d]= -B
            self.C = np.append(self.C, C)

        G_eq = np.hstack((Gx, Gu))
        C_eq = self.C

        if self.printLevel >= 2:
            print("G_eq: ")
            print(G_eq)
            print("E_eq: ")
            print(E_eq)
            print("C_eq: ", C_eq)

        self.C_eq = C_eq
        self.G_eq = sparse.csc_matrix(G_eq)
        self.E_eq = E_eq
```



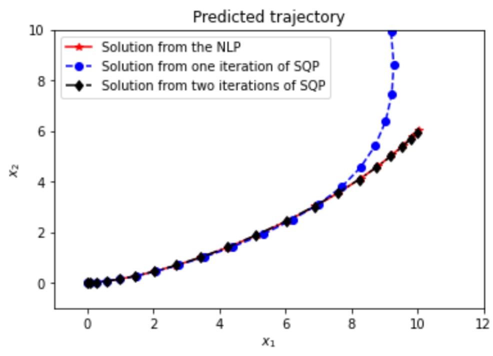


Figure 3: 1.2 Trajectory

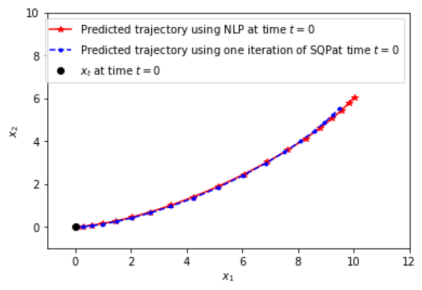
The SQP algorithm well-approximates the solution from NLP after 2 iterations.The two curves overlap pretty well for the second iteration. And SQP is faster. The first iteration takes 0.011 seconds to linearize, and it takes 0.004 seconds to solve. In total it's about 0.015 seconds. The second iteration takes 0.013 in total. In comparison, the median time the NLP solver takes to solve one solution is about 0.04. So SQP is about three times faster.

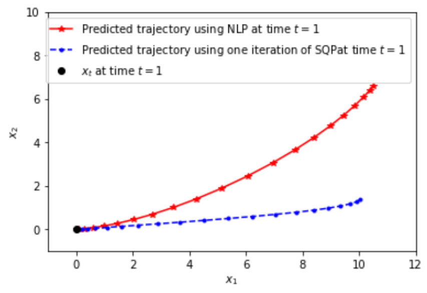
1.3 Nonlinear MPC using an SQP Approach (2 points)

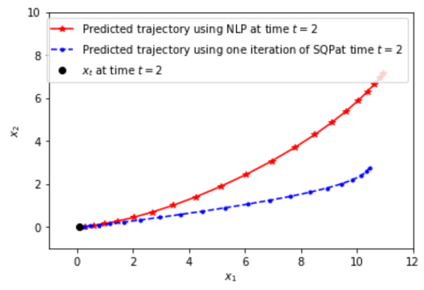
Observation: the closed-loop trajectories using the two methods both matches with NLP trajectory overall, while approach 2 matches even better(almost the same). The predicted trajectory at some time does not exactly match with the trajectory of NLP, though. The difference in predicted trajectories is bigger for approach 1 and smaller for approach 2. It's probably because we're using a bad uGuess in approach 1.

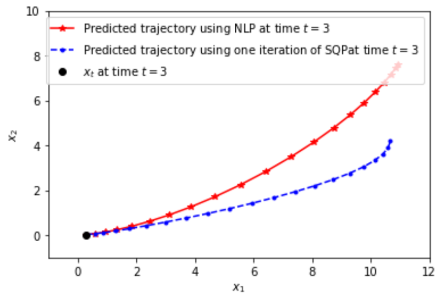
Computation time: approach 1 takes no more than 0.01 to perfrom linearization, and the solver time ranges from 0.002-0.005. So together it less than 0.015s. Approach 2 takes no more than 0.01 to linearize, and no more than 0.01 to solve. So together it takes less than 0.02s. In comparison, NLP always takes more than 0.03 and some times more than 0.05. So SQP approaches are much faster.

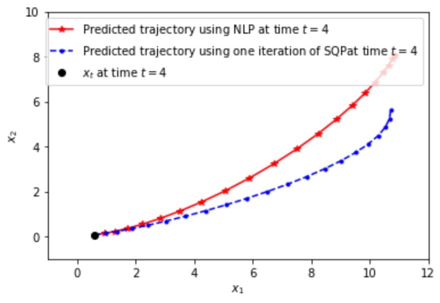
Approach 1:

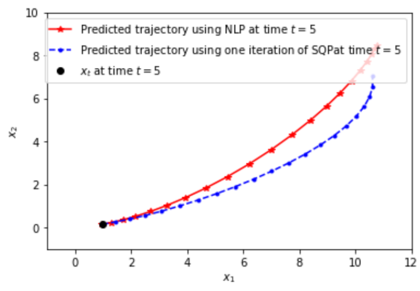

Predicted trajectory using NLP at time $t = 0$
Predicted trajectory using one iteration of SQPat time $t = 0$
x_t at time $t = 0$
x_1
x_2

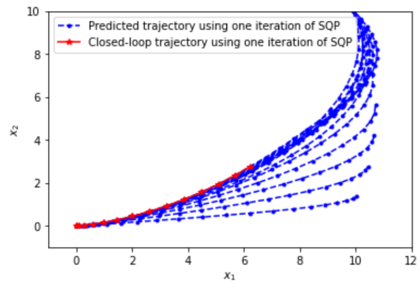

Predicted trajectory using NLP at time $t = 1$
Predicted trajectory using one iteration of SQPat time $t = 1$
x_t at time $t = 1$
x_1
x_2

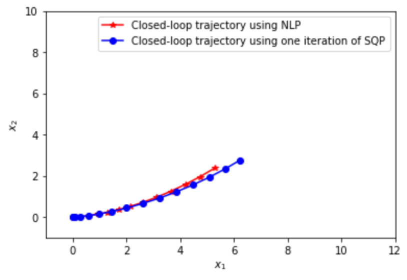

Predicted trajectory using NLP at time $t = 2$
Predicted trajectory using one iteration of SQPat time $t = 2$
x_t at time $t = 2$
x_1
x_2


Predicted trajectory using NLP at time $t = 3$
Predicted trajectory using one iteration of SQPat time $t = 3$
x_t at time $t = 3$
x_1
x_2

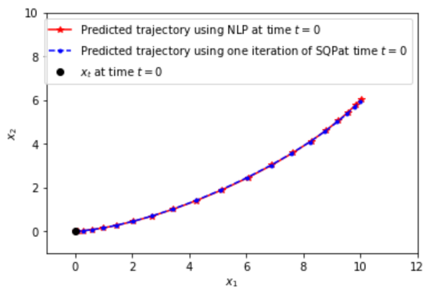

Predicted trajectory using NLP at time $t = 4$
Predicted trajectory using one iteration of SQPat time $t = 4$
x_t at time $t = 4$
x_1
x_2


Predicted trajectory using NLP at time $t = 5$
Predicted trajectory using one iteration of SQPat time $t = 5$
x_t at time $t = 5$
x_1
x_2

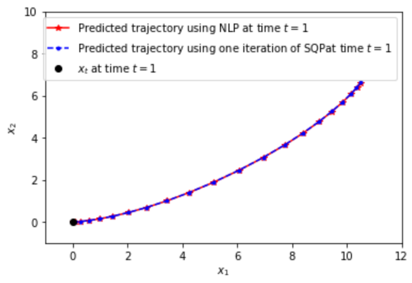

Predicted trajectory using one iteration of SQP
Closed-loop trajectory using one iteration of SQP
x_1
x_2


Closed-loop trajectory using NLP
Closed-loop trajectory using one iteration of SQP
x_1
x_2

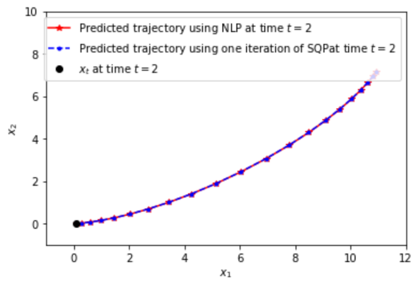
Approach 2:



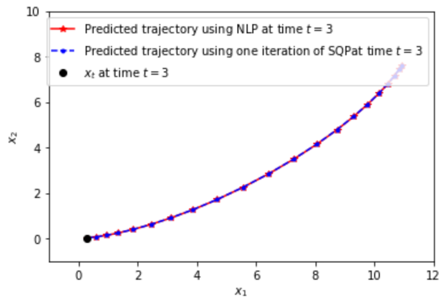
Predicted trajectory using NLP at time $t = 0$
Predicted trajectory using one iteration of SQPat time $t = 0$
x_t at time $t = 0$
x_1
x_2



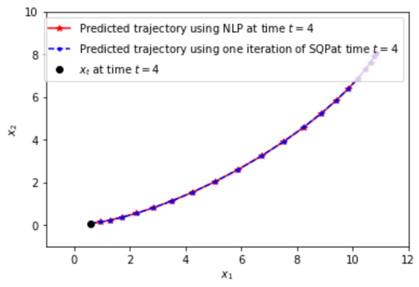
Predicted trajectory using NLP at time $t = 1$
Predicted trajectory using one iteration of SQPat time $t = 1$
x_t at time $t = 1$
x_1
x_2



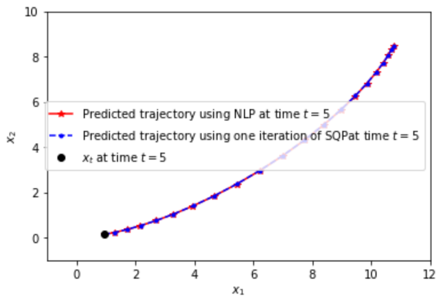
Predicted trajectory using NLP at time $t = 2$
Predicted trajectory using one iteration of SQPat time $t = 2$
x_t at time $t = 2$
x_1
x_2



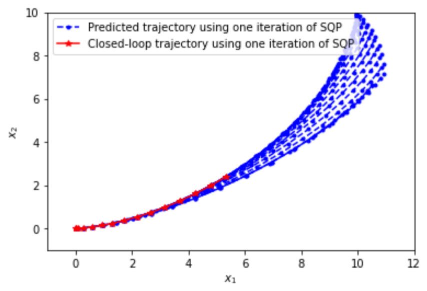
Predicted trajectory using NLP at time $t = 3$
Predicted trajectory using one iteration of SQPat time $t = 3$
x_t at time $t = 3$
x_1
x_2



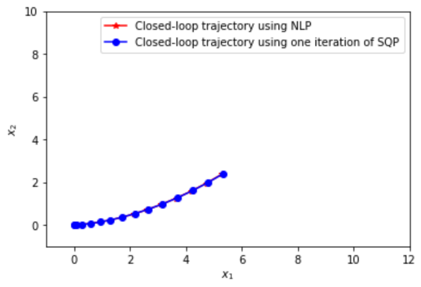
Predicted trajectory using NLP at time $t = 4$
Predicted trajectory using one iteration of SQPat time $t = 4$
x_t at time $t = 4$
x_1
x_2



Predicted trajectory using NLP at time $t = 5$
Predicted trajectory using one iteration of SQPat time $t = 5$
x_t at time $t = 5$
x_1
x_2



Predicted trajectory using one iteration of SQP
Closed-loop trajectory using one iteration of SQP
x_1
x_2



Closed-loop trajectory using NLP
Closed-loop trajectory using one iteration of SQP
x_1
x_2

2 Problem 2

2.1 MPC Design (6 points)

Approach 1:

As seen below in Figure 4, the MPC converges to the origin for N=3 and greater. When increasing N, we see that the predict trajectories converge to that of the closed loop trajectory. Marginal change is seen in increase N past 8, which makes sense since the number of points to converge is less than 8. Moreover, we note that the model did not converge when $N = 2$

Approach 2:

We chose our terminal set to be the same as the control invariant $O_i nf$ matrix F define the set, want to use the same matrice matrix define the set O_inf a set if F is identiy, b O all neg value use the similar to

Hint: the terminal set is X_f ={x — F_f x ¡= b_f} Ff = F filled in here bf = b filled in here

As seen below in Figure 7, when we go from $N = 2$ to $N = 7$, the predicted trajectory converges to the closed loop trajectory better until no further changes.

Compared to approach 1, $N = 2$ is a valid solution and its convergence with lower N is better than that generated from approach 1.

Also note from the lecture that the cost $h(x, u) \geq 0$ meaning that it will be 0 once we actually reach the equilibrium goal but it will be positive if we are not as the cost is quadratic.

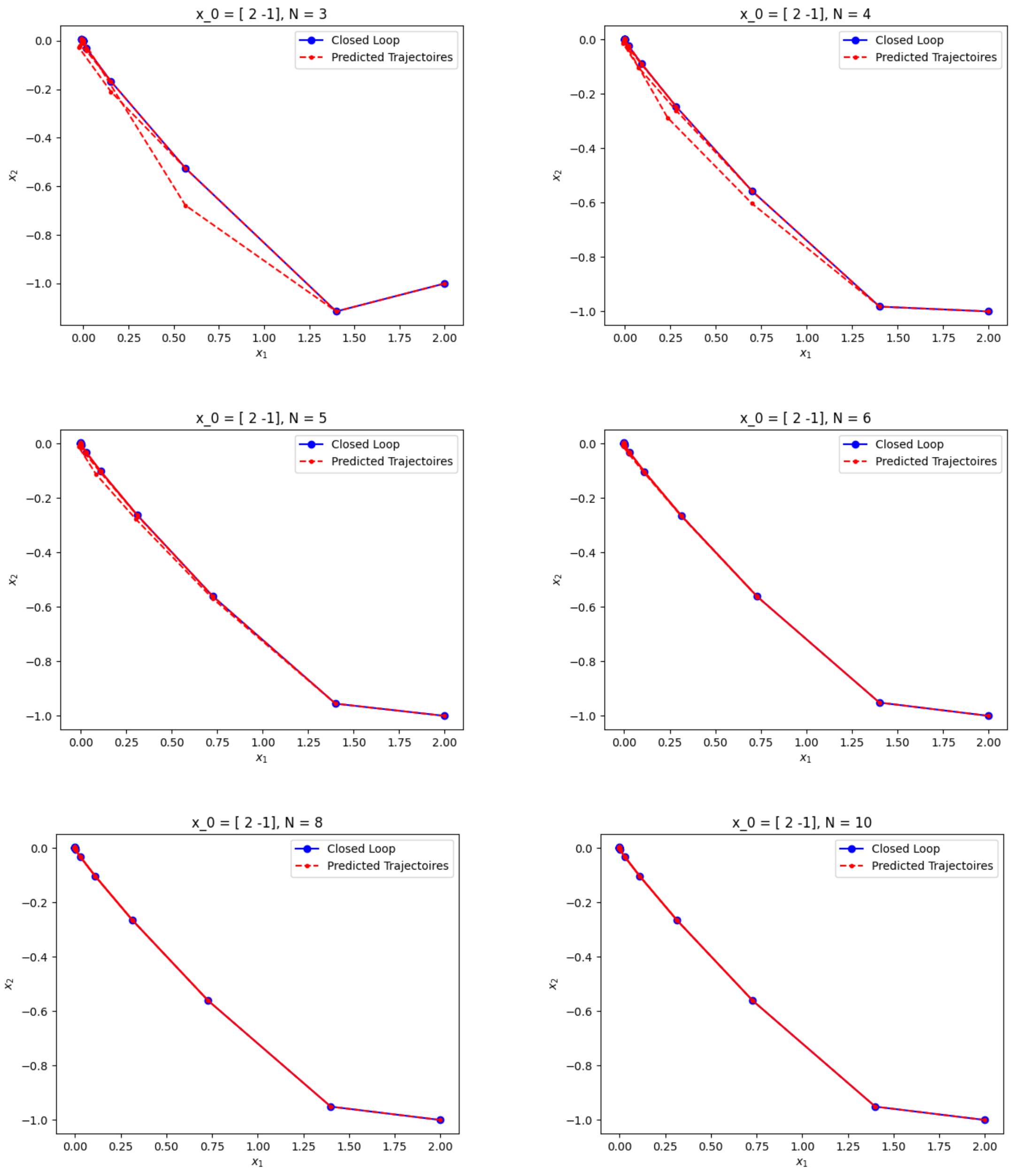


Figure 4: Close vs open-loop with different N using Approach 1

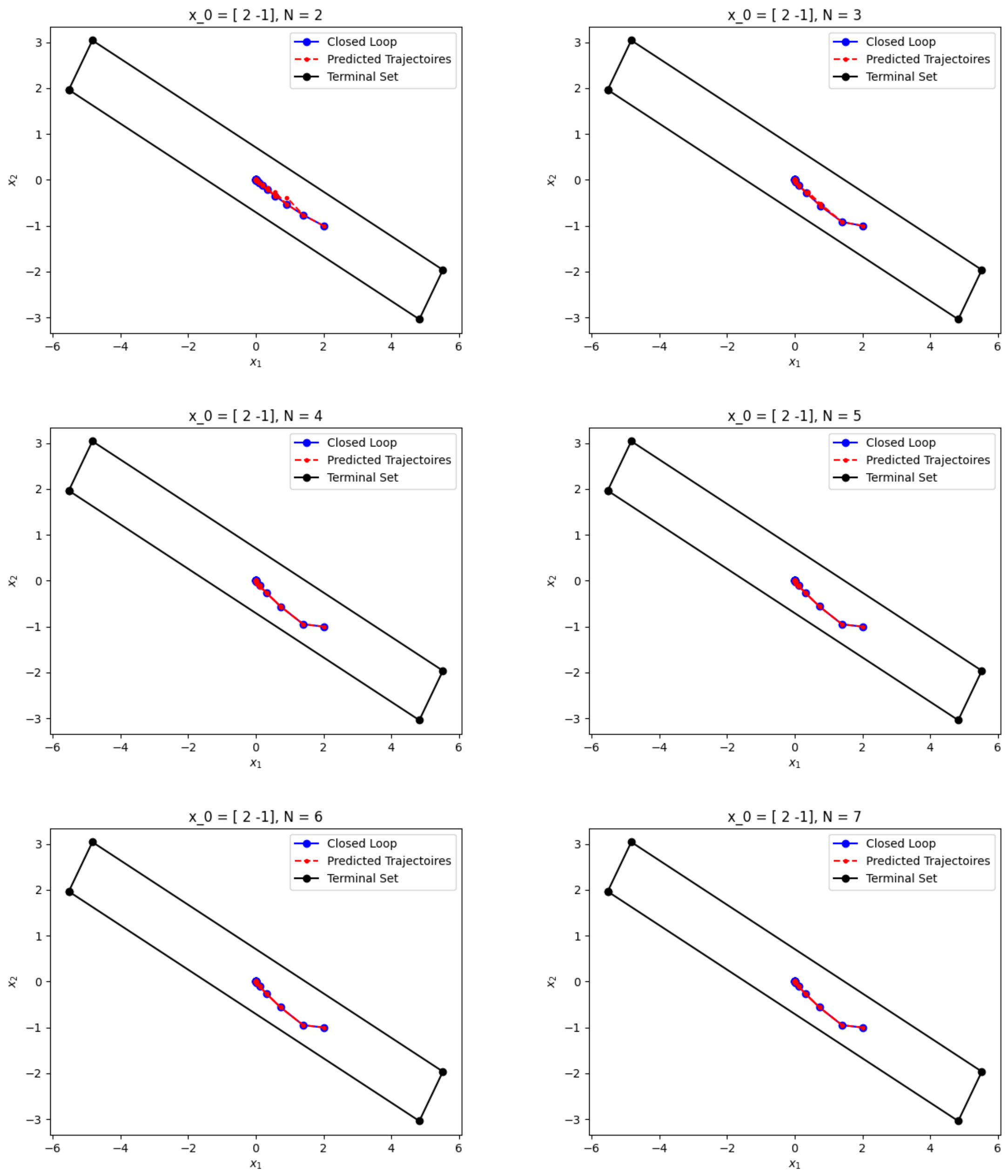


Figure 5: Close vs open-loop with different N using Approach 2

2.2 MPC Region of attraction (4 points)

Approach 1: We find that the MPC control problem is not feasible for Approach one. This can be seen by looking at Figure 7. The new initial condition for this problem is well outside the region of attraction for Approach one.

Approach 2: In contrast to approach one, we see that the FTOPC is feasible for Approach 2, which can be seen by looking at Figure 7. We see that the initial condition for this problem is near the border of the region of attraction, but it is still within the bounds. The trajectory may be seen in Figure 6.

We can have a better direct comparison with the overlay as shown in Figure 8

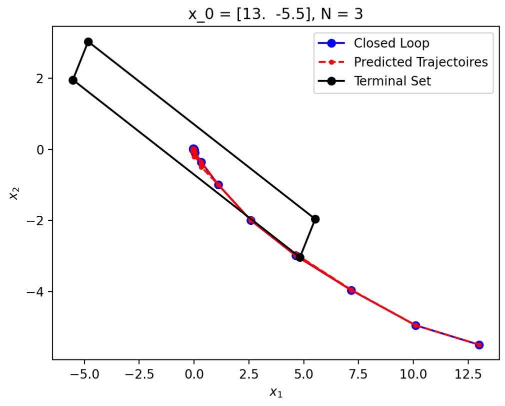


Figure 6: Trajectory for $x_0 = [13, -5.5]$ for Approach 2 with $N = 3$

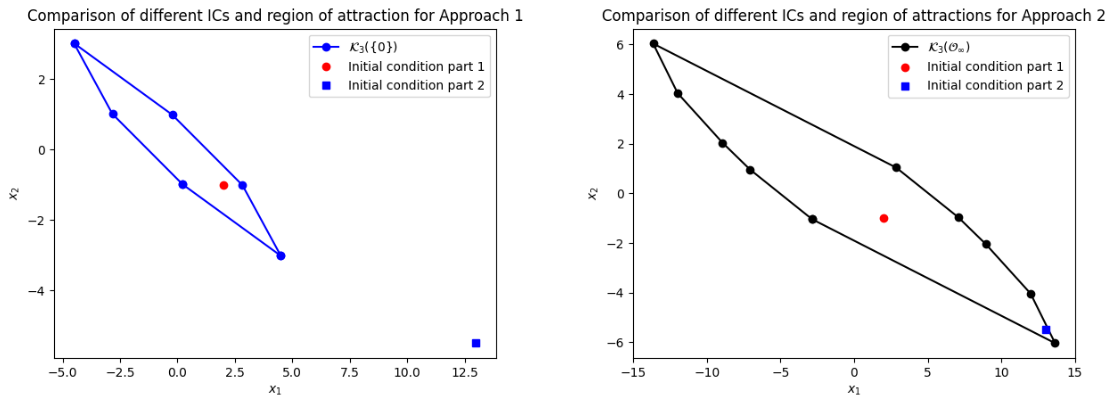


Figure 7: Compassion of different ICs and region of attractions for Approach 1 and Approach 2

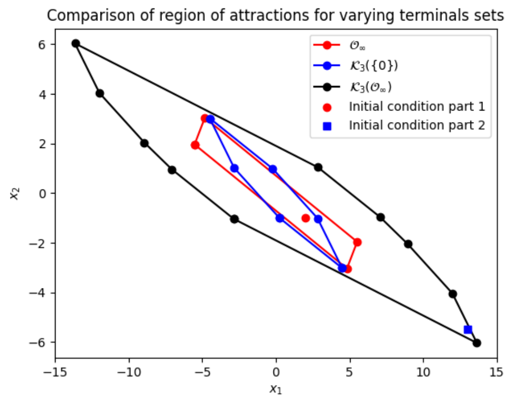


Figure 8: Compassion of region of attractions for varying terminal sets

3 Appendix

Code see Github repo

3.1 Code for Problem 1

See Github repo and above sections

3.2 Code for Problem 2

```python
import numpy as np
from utils import dlqr, system
import pdb
import matplotlib.pyplot as plt
from matplotlib import rc
import pypoman
from polytope import polytope
from ftocp import FTOCP
# =============================
# Initialize system parameters
A = np.array([[1.2, 1],
                  [0,   1]]);
B = np.array([[0],
                          [1]]);
n = 2; d = 1;
x0      = np.array([2, -1]).T   # initial condition
# x0      = np.array([13, -5.5]).T
sys     = system(A, B, x0)
maxTime = 25
N       = 3

for N in range(2,11):
        Q       = np.eye(n)
        R       = np.eye(d)

        # State constraint set X = \{ x : F_x x \leq b_x \}
        Fx = np.vstack((np.eye(n), -np.eye(n)))
        bx = np.array([15,15]*(2))

        # Input constraint set U = \{ u : F_u u \leq b_u \}
        Fu = np.vstack((np.eye(d), -np.eye(d)))
        bu = np.array([1]*2)

        # ==========================================================================================
        # ============== Approach 1 ================================================================
        # ==========================================================================================
        # Hint: the terminal set is X_f =\{x | F_f x <= b_f\}
        # These are derived by thinking about the number of minimal intersecting polytopes
        # extract only the origin
        Ff = np.array( [ [1,  -1],
                                [-1,  1],
                                [1,   1] ] )
        bf = np.array([[0], [0], [0]])
        Qf = np.array([ [1, 1],
                                [1, 1] ] ) #This doesn't matter since \mathscr{X}_f = {0}

        printLevel = 1
        mpcApproach1 = FTOCP(N, A, B, Q, R, Qf, Fx, bx, Fu, bu, Ff, bf, printLevel)
        # Run a closed-loop simulation
```

```
sys.reset_IC() # Reset initial conditions
xPredApp1 = []
for t in range(0,maxTime): # Time loop
        xt = sys.x[-1]
        ut = mpcApproach1.solve(xt)
        if mpcApproach1.feasible == 0:
                print("============ The MPC problem is not feasible")
                break
        xPredApp1.append(mpcApproach1.xPred)
        sys.applyInput(ut)

x_cl_1 = np.array(sys.x)

# Plot the results if the MPC problem was feasible
if mpcApproach1.feasible == 1:
        plt.figure()
        plt.plot(x_cl_1[:,0], x_cl_1[:,1], '-ob', label = "Closed Loop")
        for i in range(0, maxTime):
                if i == 0:
                        plt.plot(xPredApp1[i][:,0], xPredApp1[i][:,1], '--.r', label="Predicted Trajectoires")
                else:
                        plt.plot(xPredApp1[i][:,0], xPredApp1[i][:,1], '--.r')

        plt.xlabel('$x_1$')
        plt.ylabel('$x_2$')
        plt.legend()
        plt.title("x_0 = {0}, N = {1}".format(str(x0), str(N)))
        plt.savefig("problem_2/approach1_figs/HW2_pb2_1_approach1_N" + str(N) + ".png")

plt.show()

# =======================================================================================
# ============== Approach 2 =============================================================
# =======================================================================================
# Hint: the dlqr function return: i) P which is the solution to the DARE,
# ii) the optimal feedback gain K and iii) the closed-loop system matrix Acl = (A-BK)
P, K, Acl = dlqr(A, B, Q, R)
Ftot = np.vstack((Fx, np.dot(Fu, -K)))
btot = np.hstack((bx, bu ))
Qf = np.eye(n) # filled in here

poli = polytope(Ftot, btot)
F, b = poli.computeO_inf(Acl)
# Hint: this function returns F and b so that compute O_inf = \{ x | Fx <= b\}
# matrix F define the set, want to use the same matrice
# matrix define the set
# O_inf a set if F is identiy, b O all neg value use the similar to

# Hint: the terminal set is X_f =\{x | F_f x <= b_f\}
Ff = F # filled in here
bf = b # filled in here

terminalSetApproach2 = polytope(Ff, bf)

mpcApproach2 = FTOCP(N, A, B, Q, R, Qf, Fx, bx, Fu, bu, Ff, bf, printLevel)

# Simulate the closed-loop system
sys.reset_IC() # Reset initial conditions
xPredApp2 = []
for t in range(0,maxTime): # Time loop
```

```
                xt = sys.x[-1]
                ut = mpcApproach2.solve(xt)
                if mpcApproach2.feasible == 0:
                        print("============ The MPC problem is not feasible")
                        break

                xPredApp2.append(mpcApproach2.xPred)
                sys.applyInput(ut)

        x_cl_2 = np.array(sys.x)

        # Plot the results if the MPC problem was feasible
        if mpcApproach2.feasible == 1:
                plt.figure()
                plt.plot(x_cl_2[:,0], x_cl_2[:,1], '-ob', label = "Closed Loop")
                for i in range(0, maxTime):
                        if i == 0:
                                plt.plot(xPredApp2[i][:,0], xPredApp2[i][:,1], '--.r', label="Predicted Trajectoires")
                        else:
                                plt.plot(xPredApp2[i][:,0], xPredApp2[i][:,1], '--.r')

                terminalSetApproach2.plot2DPolytope('k','Terminal Set')
                plt.xlabel('$x_1$')
                plt.ylabel('$x_2$')
                plt.legend()
                plt.title("x_0 = {0}, N = {1}".format(str(x0), str(N)))
                plt.savefig("problem_2/approach2_figs/HW2_pb2_1_approach2_N" + str(N) + ".png")

        plt.show()

# =======================================================================================
# ============== Compute the Region of Attraction =======================================
# =======================================================================================

# First we over-approximate the terminal set used for approach 1
# (We do so to have a set which is full dimension)
Ff = np.vstack((np.eye(n), -np.eye(n)))
bf = np.hstack((np.ones(n), np.ones(n)))*0.01
terminalSetApproach1 = polytope(Ff, bf)

# Compute the N-Step Controllable sets for approach 1 and approach 2
NStepControllable = []
for terminalSet in [terminalSetApproach1, terminalSetApproach2]:
        F, b = terminalSet.NStepPreAB(A, B, Fu, bu, N)
        NStepControllable.append(polytope(F, b))

# Plot the results
plt.figure()
terminalSetApproach2.plot2DPolytope('r', '$\mathcal{O}_\infty$')
NStepControllable[0].plot2DPolytope('b', '$\mathcal{K}_3(\{0\})$')
NStepControllable[1].plot2DPolytope('k', '$\mathcal{K}_3(\mathcal{O}_\infty)$')
plt.plot([2], [-1], 'or', label='Initial condition part 1')
plt.plot([13], [-5.5], 'sb', label='Initial condition part 2')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.legend()

plt.title('Comparison of different ICs and region of attractions for Approach 2')
plt.savefig('HW2_pb2_2_approach2.png')
plt.show()
```