CH E 663 Final Project

Exploring the Control of DC-DC Boost Converter

Amr Marey

December 15, 2023

Contents

1 System description

DC-DC converter electric circuits are power electronic circuits that are used to transform electrical power with minimum loss of efficiency [**boost**]. For example, consider we have an electrical load that requires 5V to operate and we only have a 3.3V power source. One can use a boost converter circuit, shown in Figure. 1, to aid in this problem [**boost**]. The input to this circuit would be the input voltage, denoted as V_i, which in this example is 3.3V. The electrical load would be connected to the output side of the circuit which has an output voltage v_o. The control objective is to keep the output voltage v_o as close as possible the desired voltage V_o^*, which is 5V in this example. This circuit consists of switches (assumed to be ideal in this report), and passive linear components having capacitance C, inductance L, and resistance R. The current flowing through the inductor is denoted as i_L and the voltage across the capacitor is identical to the output voltage v_o. The switch operates at a high frequency. The author of this report assumes the reader has a preliminary understanding of Kirchhoff's laws and electrical component system modeling as a discussion of these topics would detract from the main focus of the report. [**circuit˙review**] discusses Kirchhoff's laws and electrical component system modeling in detail.

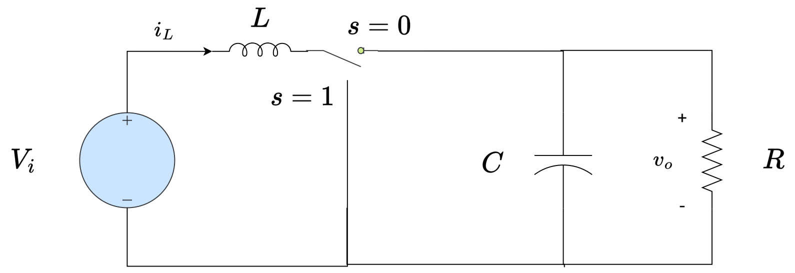


Figure 1: Topology of DC-DC boost converter circuit.

One can note the dimension of this system is two due to the presence of one inductor and one capacitor in the system. This is a set-point tracking problem as we are trying to follow a specific trajectory of an output voltage.

1.1 Deriving the State-Space Model

The circuit topology in Figure 1 has two states depending on whether the switch is in the $s = 0$ position or in the $s = 1$ position. The circuit topology for when $s = 0$ is shown in Figure 2. Similarly, the circuit topology for when $s = 1$ is shown in Figure 3.

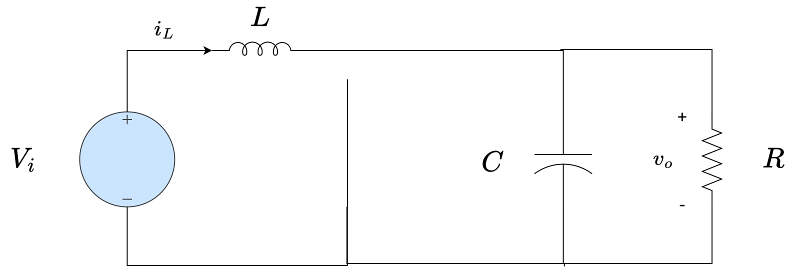


Figure 2: Topology of DC-DC boost converter circuit when $s = 0$.

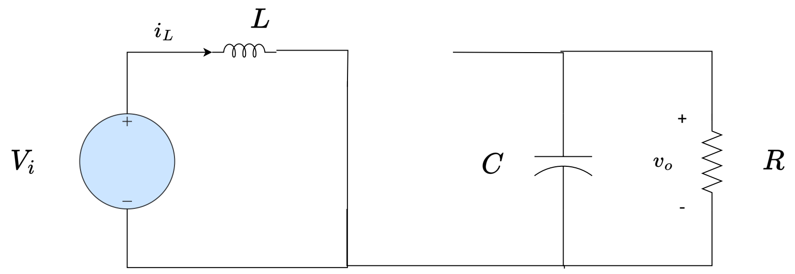


Figure 3: Topology of DC-DC boost converter circuit when $s = 1$.

From Figures 2 and 3, one can use Kirchhoff's laws to obtain the following system equations for each state

$$L\frac{di_L}{dt} = \begin{cases} V_i - v_o & s = 0 \\ V_i & s = 1 \end{cases} \tag{1}$$

$$C\frac{dv_o}{dt} = \begin{cases} i_L - \frac{v_o}{R} & s = 0 \\ \frac{-v_o}{R} & s = 1 \end{cases} \tag{2}$$

One can combine (1) and (2) into (3)

$$\begin{aligned} L\frac{di_L}{dt} &= V_i - (1-s)v_o \\ C\frac{dv_o}{dt} &= (1-s)i_L - \frac{v_o}{R} \end{aligned} . \tag{3}$$

Suppose the duty ratio of the circuit is denoted as D, where $0 \leq D \leq 1$. The duty ratio is the percentage of the time that the circuit operates at $s = 1$, one can use a high frequency transformation technique to average (3) and obtain (4) [**PE˙book**]. D is the control input that will regulate v_o.

$$\begin{aligned} L\frac{di_L}{dt} &= V_i - (1-D)v_o \\ C\frac{dv_o}{dt} &= (1-D)i_L - \frac{v_o}{R} \end{aligned} . \tag{4}$$

The steady-state equilibrium point of the converter can be obtained by setting the left hand side of (4) to 0. Hence, the steady state inductor current, the steady state duty ratio, and the steady-state output voltage are [**PE˙book**]:

$$\begin{aligned} i_L^* &= \frac{V_o^2}{V_{in}R} \\ v_o^* &= V_o \\ D^* &= 1 - \frac{V_{in}}{V_o} \end{aligned} . \tag{5}$$

Where V_o is the desired output voltage. We define $\alpha = \frac{1}{R}\sqrt{L/C}, u = D - D^*, \tau = \frac{t}{\sqrt{LC}}, k = V_o/V_i$. We define the state variables to be

$$\begin{aligned} x_1 &= \frac{1}{V_i}\sqrt{\frac{L}{C}}(i_L - i_L^*) \\ x_2 &= \frac{v_o - V_o}{V_i} \end{aligned} . \tag{6}$$

Hence, the state-space model of the boost-converter (obtained after some complex algebra) is

$$\begin{aligned} \dot{x}_1 &= \frac{-x_2}{k} + (x_2 + k)u \\ \dot{x}_2 &= \frac{x_1}{k} - \alpha x_2 - (x_1 + \alpha k^2)u \\ y &= x_2 \end{aligned} \cdot \tag{7}$$

This state-space model was particularly chosen such that the model can have an equilibrium point at $x_1 = x_2 = u = 0$. We take the output to be $y = x_2$ as that is the state that has the output voltage embedded in it [**PE˙book**].

1.2 Design Objectives and Circuit Parameter Selection

For a 3.3V to 5V boost converter, the input voltage will be centered around $3.3V$ and the output voltage will be centered around $5V$. Hence we will will design the controller such that the input voltage can range from 3V until 3.6V to show the robustness of the control algorithm. Similarly, we would like to have the steady-state desired output voltage to range from 4.5V to 5.5V to show robustness of our model. Assume the load has a resistance of $R = 100\Omega$.

We would like the output voltage ripple Δv_o to be 1% of the steady-state output voltage and the inductor current ripple Δi_L to be 10 % of the steady-state inductor current. We first calculate the steady-state values from (5):

$$\begin{aligned} i_L^* = \frac{V_o^2}{V_{in}R} = \frac{5^2}{3.3(100)} = 75.76\text{mA} \\ v_c^* = V_o = 5\text{V} \\ D = 1 - \frac{V_{in}}{V_o} = 0.34 \end{aligned} \tag{8}$$

Hence, the desired Δv_o and Δi_L are:

$$\begin{aligned} \Delta v_o = 0.01 v_C^* = 50\text{mV} \\ \Delta i_L = 0.1 i_L^* = 7.56\text{mA} \end{aligned} \tag{9}$$

Assume that switch is operating at a switching frequency of $f_s = 10$KHz. To select the suitable inductor and capacitor values for these design constraints, one can use the following equations obtained from [**PE˙book**]:

$$\begin{aligned} C = \frac{V_i D^*}{2Rf_s \Delta v_o} = \frac{3.3(0.34)}{2(100)(10\times10^3)(50\times10^{-3})} \approx 12\mu\text{F} \\ L = \frac{V_i D^*}{2f_s \Delta i_L} = \frac{3.3(0.34)}{2(10\times10^3)(7.56\times10^{-3})} \approx 7.5\text{mH} \end{aligned} \tag{10}$$

This allows us to determine the parameters of the state-space model presented in (7). Hence,

$$\begin{aligned} k = \frac{V_o}{V_i} = \frac{5}{3.3} \approx 1.52 \\ \alpha = \frac{1}{R}\sqrt{\frac{L}{C}} = \frac{1}{100}\sqrt{\frac{7.5\times10^{-3}}{12\times10^{-6}}} = 0.25 \end{aligned} \tag{11}$$

1.3 Cost Function

The chosen cost function for this desired output voltage tracking problem is:

$$J = \int_0^{t_f} x^T Q x + u^T R u \mathrm{d}t \tag{12}$$

Where $x = [x_1, x_2]^T$ and we define Q and R to be diagonal real matrices ($Q \in \mathcal{R}^{2\times 2}$). Hence,

$$Q = \begin{bmatrix} q_1 & 0 \\ 0 & q_2 \end{bmatrix}.$$

such that q_1 and q_2 are real numbers. The next task is to determine the weights q_1, q_2 and R. There are multiple ways to do this (neural networks, genetic search, etc). One method is to choose $R = 1$ as is this a single input system and choose $q_i (i = 1, 2)$ such that

$$q_i = \frac{1}{\max\{[x_i]^2\}_{SS}}.$$

Where $\max\{[x_i]^2\}_{SS}$ is the maximum of $x_i^2(t)$ in steady-state operation. This is referred to as Bryson's Method [**buck˙lqr˙thesis**].

The maximums of x_1^2 and x_2^2 will occur when i_L and v_o are at their maximum. These maximum values are:

$$\begin{gathered} \max\{i_L\}_{SS} = i_L^* + \Delta i_L = 83.32\text{mA} \\ \max\{v_o\}_{SS} = v_o^* + \Delta v_o = 5 + 0.05 = 5.05\text{V} \end{gathered}. \tag{13}$$

From (6) and (13), one can then obtain q_1 and q_2 as $q_1 = 304.57$ and $q_2 = 4.357 \times 10^3$.

1.4 Input and State Constraints

The input duty cycle to the ideal switch D is between 0 and 1 ($0 \le D \le 1$). Since $u = D - D*$, the input constraint to the state-space model is

$$-D^* \leq u \leq 1 - D^* \text{ or } -0.34 \leq u \leq 0.66. \tag{14}$$

The inductor current and capacitor voltage must be greater than 0 at all times to prevent DCM operation of the boost converter (DCM operation changes the state-space model drastically) [**PE˙book**]. Furthermore, it must be less than the maximum rated current that the inductor can handle and the capacitor can handle. We assume the maximum rated current for the inductor is $i_{L,max} = 150mA$ and the maximum rated voltage for the capacitor is $v_{o,max} = 7V$. Hence to obtain the minimum values of x_1 and x_2, we substitute $i_{L,min} = v_{o,min} = 0$ in (6). We substitute $i_{L,max}$ and $v_{o,max}$ in (6) to obtain the maximum possible values for x_1 and x_2. This yields the state-constraints: $-0.5739 \leq x_1 \leq 0.5624, -1.5151 \leq x_2 \leq 0.6061$.

1.5 Summarizing the Optimal Problem

The objective is to track the output voltage v_o to some desired output voltage V_o. Mathematically speaking this is equivalent to trying to have $x_1(t) \rightarrow 0$ and $x_1(t) \rightarrow 0$ as $t \rightarrow t_f$. Where t_f is the terminal time. Hence, the optimal control problem can be formulated as

$$u^*(t) = \underset{u(t)}{\operatorname{argmin}} \int_0^{t_f} x^T \underbrace{\begin{bmatrix} 304.57 & 0 \\ 0 & 4.357 \times 10^3 \end{bmatrix}}_{Q} x + u^T \underbrace{(1)}_{R} u \, \mathrm{d}t$$

$$\begin{gathered} \text{such that} \\ \dot{x}_1 = \frac{-x_2}{k} + (x_2 + k)u := f_1(x,u) \\ \dot{x}_2 = \frac{x_1}{k} - \alpha x_2 - (x_1 + \alpha k^2)u := f_2(x,u) \\ y = x_2 \\ -0.34 \leq u \leq 0.66 \\ -0.5739 \leq x_1 \leq 0.5624 \\ -1.5151 \leq x_2 \leq 0.6061 \\ \alpha = 0.25, k = 1.52 \end{gathered} \quad . \tag{15}$$

We also define $f(x,u) = [f_1(x,u), f_2(x,u)]^T$

2 Initial Steps

1

There exists a steady-state operating point at the point $x_1 = x_2 = u = 0$. Hence we define the equilibrium point to $x_{1s} = 0, x_{2s} = 0, u_s = 0$. As a matter of fact, the author intentionally designed the the state-variables in a way so that there can be an equilibrium point at the origin. Hence, the original states x can be identical to the deviation states $\tilde{x} = x - x_s = x$ and so that the deviation input $\tilde{u}$ can be identical to the original input u ($\tilde{u} = u - u_s = u$.)

2

We linearize the system around the operating point $(x_s, u_s) = (0, 0)$ to obtain the linear system

$$\dot{\tilde{x}} = A\tilde{x} + B\tilde{u}. \tag{16}$$

Where

$$\begin{aligned} A = \frac{\partial f}{\partial x}\Big|_{(0,0)} &= \begin{bmatrix} 0 & \frac{-1}{k} + u \\ \frac{1}{k} - u & -\alpha \end{bmatrix}\Bigg|_{(0,0)} = \begin{bmatrix} 0 & \frac{-1}{k} \\ \frac{1}{k} & -\alpha \end{bmatrix} = \begin{bmatrix} 0 & -0.6579 \\ 0.6579 & -0.25 \end{bmatrix} \\ B = \frac{\partial f}{\partial u}\Big|_{(0,0)} &= \begin{bmatrix} x_2 + k \\ -x_1 + \alpha k^2 \end{bmatrix}\Bigg|_{(0,0)} = \begin{bmatrix} k \\ \alpha k^2 \end{bmatrix} = \begin{bmatrix} 1.52 \\ 0.5776 \end{bmatrix} \end{aligned}. \tag{17}$$

Hence, the linearized model is

$$\begin{aligned} \dot{\tilde{x}} &= \begin{bmatrix} 0 & -0.6579 \\ 0.6579 & -0.25 \end{bmatrix}\tilde{x} + \begin{bmatrix} 1.52 \\ 0.5776 \end{bmatrix}\tilde{u} \\ y &= \tilde{x}_2 \end{aligned}. \tag{18}$$

3

We check the controllability of the system (A,B):

$$C_{ctrb} = [B, AB] = \begin{bmatrix} 1.52 & -0.38 \\ 0.5776 & 0.8556 \end{bmatrix}.$$

We note that C_{ctrb} is full-rank as $\det(C_{ctrb}) = 1.52 \neq 0$. Hence the system (A,B) is controllable. For the rest of this report we use x and u instead of $\tilde{x}$ and $\tilde{u}$ as the variables are identical for our system.

3 LQR

1

Our LQR performance measure is

$$J = \int_0^\infty x^T Q x + u^T R u \ \mathrm{d}t.$$

Although, we defined Q above using Bryson's method [**buck˙lqr˙thesis**]. The question states we should define Q to be $Q = C^T C$. In our example $C = [0, 1]$, hence

$$Q = C^T C + 0.001I = [0,1]^T[0,1] + 0.001I = \begin{bmatrix} 0.001 & 0 \\ 0 & 1.001 \end{bmatrix} \tag{19}$$

R will still be $R = 1$. We test the observability of (A, C) :

$$C_{obsv} = \begin{bmatrix} C \\ CA \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0.6579 & -0.25 \end{bmatrix}.$$

We note C_{obsv} is full-rank as $\det(C_{obsv}) = -0.6579 \neq 0$. Hence (A, C) is observable. Furthermore $Q \geq 0$. We choose $R = 1$ (as discussed in Section 1). Hence, $R > 0$.

2

The algebraic Riccati equation is $-PA - A^T P - Q + PBR^{-1}B^T P = 0$. We define P to be

$$P = \begin{bmatrix} p_1 & p_2 \\ p_2 & p_3 \end{bmatrix}.$$

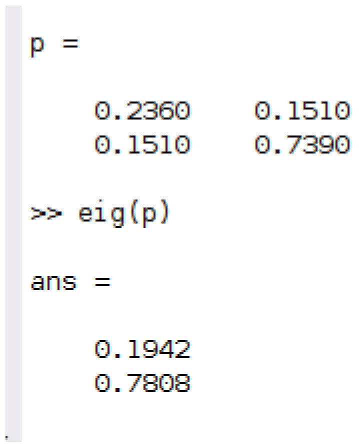
We compute the algebraic Riccati equation and solve the Riccati equation using the MATLAB code below:

```
clear; clc; close all
syms p1 p2 p3
P = [p1, p2; p2, p3];
A = [0, -0.6579; 0.6579, -0.25];
B = [1.52, 0.5776]';
Q = [0,0;0,1];
R = 1;

Riccati_mat = -P*A-A'*P-Q+P*B*inv(R)*B'*P;
eqn1 = Riccati_mat(1,1) == 0;
eqn2 = Riccati_mat(1,2) == 0;
eqn3 = Riccati_mat(2,2) == 0;
sol = solve(eqn1,eqn2,eqn3, p1, p2, p3);
vpa(sol.p1, 3)
vpa(sol.p2, 3)
vpa(sol.p3, 3)
```

A screenshot (see Figure 4) shows the matrix P computed by MATLAB and the eigenvalues of P:

We note that $P > 0$. Hence, we conclude the the solution to the Riccati equation for the



```
p =

    0.2360    0.1510
    0.1510    0.7390

>> eig(p)

ans =

    0.1942
    0.7808
```

Figure 4: The solution to the Riccati equation.

given system is

$$P = \begin{bmatrix} 0.2360 & 0.1510 \\ 0.1510 & 0.7390 \end{bmatrix}.$$

As discussed in class, the optimal control $u^*(t)$ would then be

$$u^*(t) = -Kx = -R^{-1}B^T Px(t) = -(1)^{-1}[1.52, 0.5776] \begin{bmatrix} 0.2360 & 0.1510 \\ 0.1510 & 0.7390 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = -0.4459x_1 - 0.6564x_2.$$

3

Putting the optimal control law into the system yields

$$\dot{x} = (A - BK)x = \begin{bmatrix} -0.6778 & -1.6556 \\ 0.4003 & -0.6291 \end{bmatrix} x$$

Assume $x(0) = [x_1(0), x_2(0)]^T = [0.25, 0.25]$. Hence the state trajectory is:

$$x(t) = e^{(A-BK)t}x(0).$$

We use MATLAB to compute the state-trajectory:

```
clear; clc; close all
A = [0, -0.6579; 0.6579, -0.25];
B = [1.52, 0.5776]';
K = [0.4459376, 0.6563664];
syms t
x0 = [0.25,0.25]'; %Initial condition
x = expm((A-B*K)*t)*x0;
x = vpa(simplify(vpa(simplify(rewrite(x, 'sincos')),16)),4);
```

The code yields:

$$x(t) = \begin{pmatrix} 0.5735\,\cos\left(0.8137\,t + 1.12\right)\,\mathrm{e}^{-0.6535\,t} \\ 0.282\,\mathrm{e}^{-0.6535\,t}\,\cos\left(0.8137\,t - 0.481\right) \end{pmatrix}.$$

Hence,

$$u^*(t) = -0.4459x_1 - 0.6564x_2$$

$$= -0.2557\cos\left(0.8137\,t + 1.12\right)\,\mathrm{e}^{-0.6535\,t} - 0.1851\mathrm{e}^{-0.6535\,t}\,\cos\left(0.8137\,t - 0.481\right).$$

The state-trajectories and input trajectory are presented in Figure 6. As expected the states and the input approach $(x_s, u_s) = (0, 0)$ over time.

4

Applying LQR to the original nonlinear system yields:

$$\dot{x} = f(x, -Kx)$$

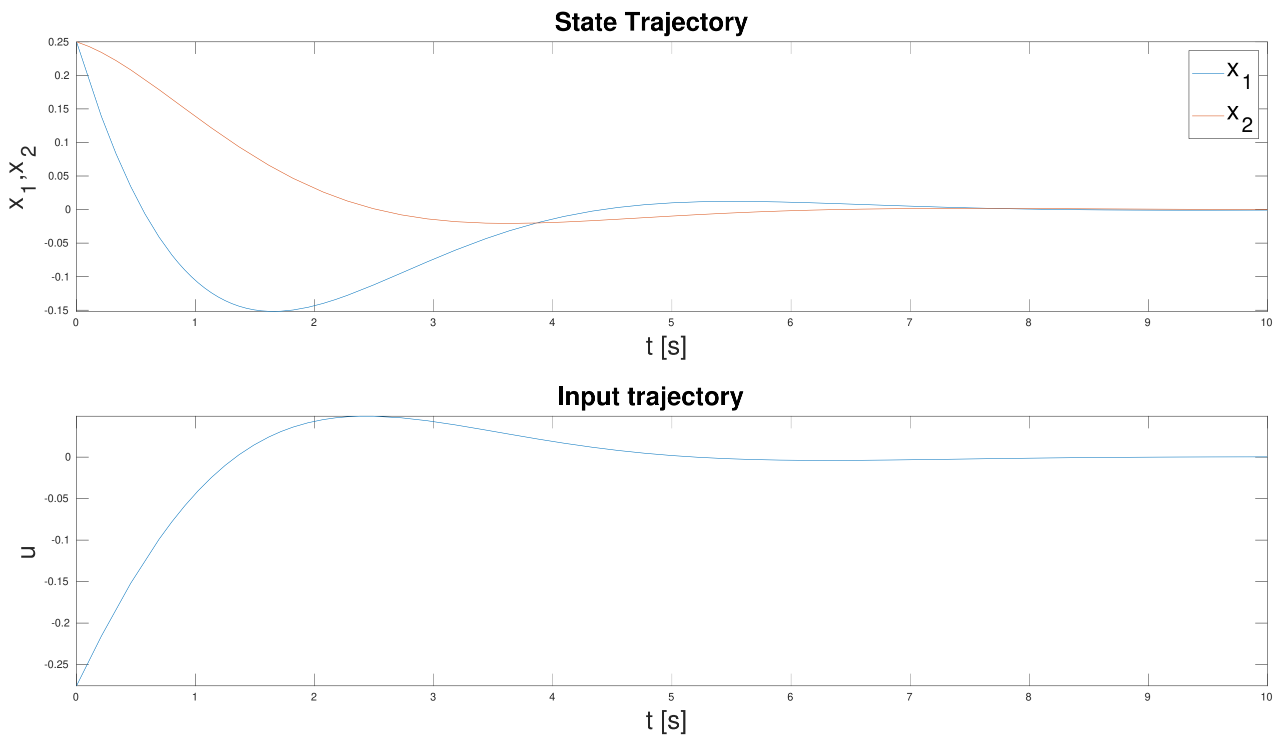


Figure 5: State trajectory and input trajectory of linearized system.

$$= \begin{bmatrix} \dfrac{-x_2}{k} + (x_2+k)(-0.2557\cos(0.8137\,t+1.12)\,\mathrm{e}^{-0.6535\,t} - 0.1851\mathrm{e}^{-0.6535\,t}\,\cos(0.8137\,t-0.481)) \\ \dfrac{x_1}{k} - \alpha x_2 - (x_1+\alpha k^2)(-0.2557\cos(0.8137\,t+1.12)\,\mathrm{e}^{-0.6535\,t} - 0.1851\mathrm{e}^{-0.6535\,t}\,\cos(0.8137\,t-0.481)) \end{bmatrix}$$

The MATLAB code to solve this system numerically is presented below:

```
function dydt = myODEs(t, y)
    % Define your system of ODEs here
    dydt = zeros(2, 1); % Preallocate the output vector
    k = 1.52; a = 0.25;
    K = [0.4459376, 0.6563664];
    % Example ODEs
    dydt(1) = -y(2)/k +(y(2)+k)*(-K*[y(1), y(2)]');
    dydt(2) = y(1)/k -a*y(2) -(y(1)+a*k^2)*(-K*[y(1), y(2)]');
end

% Set up time span
tspan = [0 20];

% Set initial conditions
y0 = [0.25; 0.25]; % Example initial condition

% Solve the system of ODEs
[t, y] = ode45(@myODEs, tspan, y0);

% Plot the results
figure;
plot(t, y(:, 1), 'b-', t, y(:, 2), 'r-');
xlabel('Time');
ylabel('Solution');
legend('y1', 'y2');
title('Solution of the System of ODEs');
```

The state-trajectories for the nonlinear system are presented in Figure .

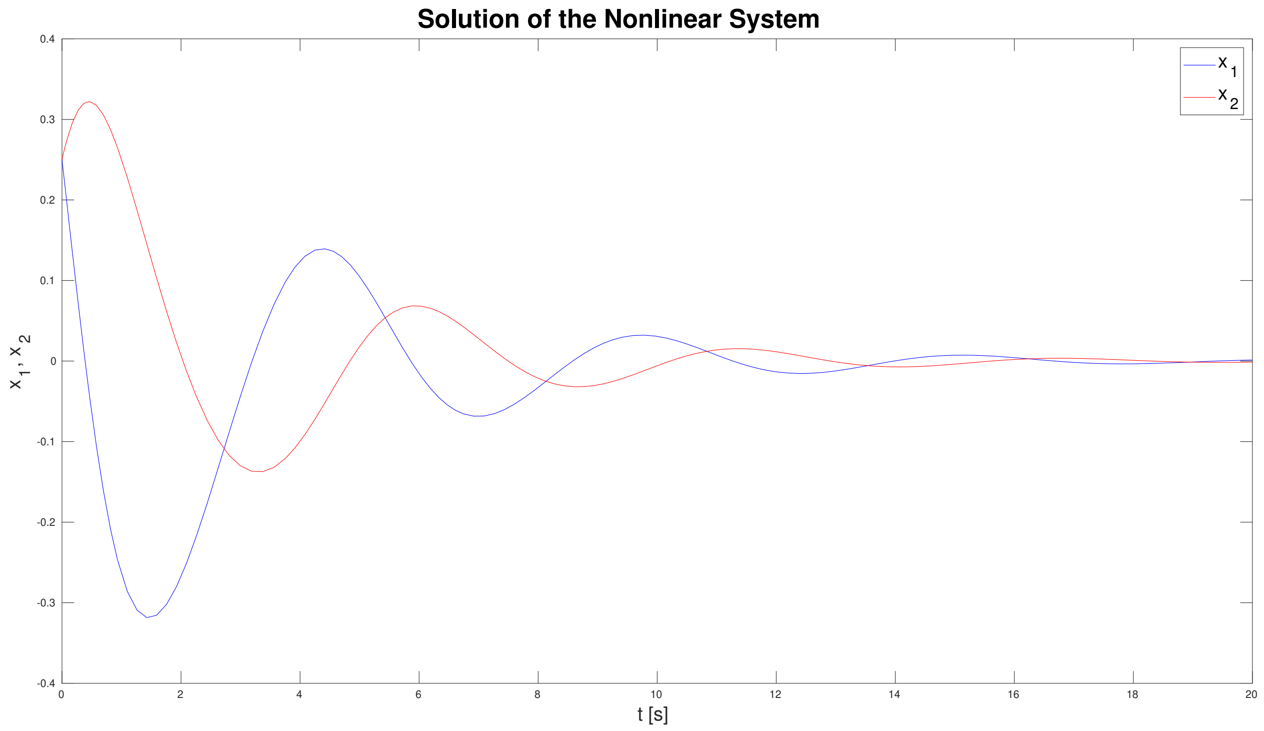


Figure 6: State trajectory and input trajectory of nonlinear system.

4 Model Predictive Control

We consider two separate model predictive control (MPC) approaches to this control problem (referred to as Approach 1 and Approach 2 in class). The input constraints are considered The state-constraints can be neglected as they are much more lenient compared to the input constraint (the system was intentionally designed this way in Section 1). As one can see shortly, all control approaches result in feasible state-trajectories and each state-trajectory doesn't come near the constraints. As mentioned in class, the MPC problem can be described as:

$$u^* = \underset{u}{\operatorname{argmin}} \sum_{k=0}^{\infty} [x_k^T Q x_k + u_k^T R u_k] \tag{20}$$

$$\begin{aligned} \text{s.t.} x_{k+1} &= f(x_k, u_k) \\ x_0 &= \bar{x} \\ x &\in X \\ u &\in U \end{aligned} .$$

Where $u*$ is the optimal input trajectory and X and U are the set of all feasible states and feasible input values. Due to its heavy computational requirements, this control method can't be implemented on a real computer. Hence, we approximate it using two different methods (referred to as Approach 1 and Approach 2 in class).

4.1 Linear MPC

We first obtain the discrete-time (DT) linear model of (16). This model is described as

$$x_{k+1} = A_d x_k + B_d u_k. \tag{21}$$

Where T denotes the sampling-time of the DT model and $A_d = e^{AT_s}, B_d = A^{-1}(e^{AT_s} - I)$. We choose $T = 0.1$, from this we evaluate the DT system matrices:

$$A_d = \begin{bmatrix} 0.99785453 & -0.06492757 \\ 0.06492757 & 0.97318225 \end{bmatrix}, B_d = \begin{bmatrix} 0.15000745 \\ 0.06195968 \end{bmatrix}.$$

The eigenvalues of A_d are $\lambda_{1,2} = 0.9855 \pm 0.0637j$, which both lie inside the unit circle $|\lambda_{1,2}| < 1$. This implies that the DT system is stable. We begin our MPC analysis with this approximated model.

4.1.1 Approach 1

We attempt to approximate (20) by selecting a long enough horizon. We define the cost function to be

$$J_{L1} = \sum_{k=0}^{N-1} [x_k^T Q x_k + u_k^T R u_k] + x_N^T Q x_N \tag{22}$$

.

Where N is the horizon-length that we need to figure out. To identify a suitable value for N, we do a brief search by exploring the effect of the cost J as the N increases. This relationship is presented in Figure 7.

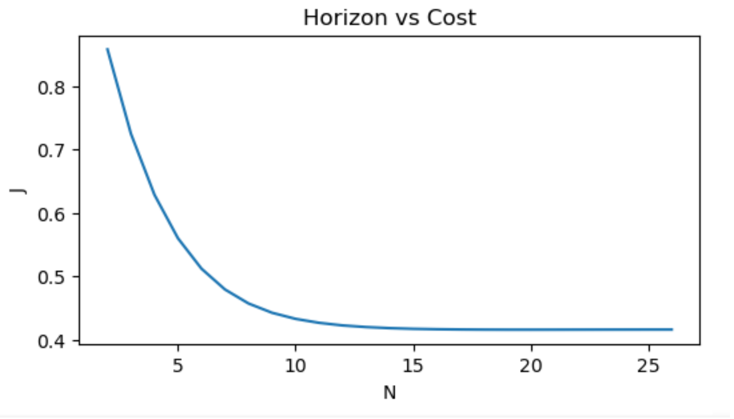


Figure 7: Comparing J for different values of N for the system modeled by (21) (Approach 1).

We see that J_{L1} is minimized when $N = 16.$ and above. However, if $N > 16$, the computational requirements increase without any benefit to the control performance (maybe very small improvements). We find $u*$ using the cost function (23). The state and input trajectories are presented in Figure 8.

We show that as N increases there is no visible improvement in the control performance (Figure 9) and when N decreases, we see visible degradation in the control performance

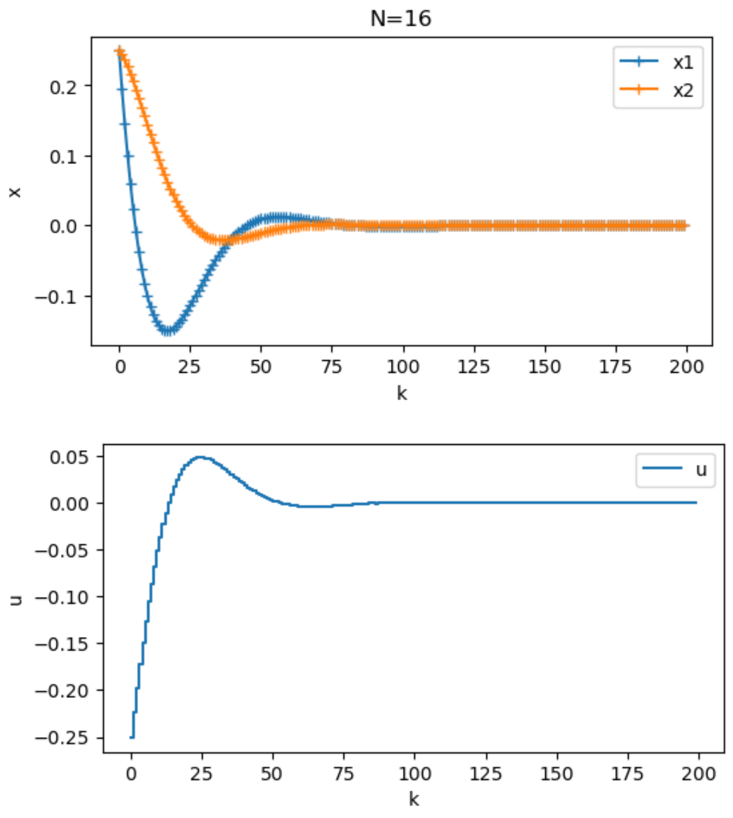


Figure 8: The input and state trajectories for Approach 1 MPC using N=16.

(Figure 10).

4.1.2 Approach 2

One can improve the performance of Approach 1 by requiring that for each horizon optimization that the end state reaches $x_s = 0$ (Approach 2 or terminal point constraint as discussed in class). This can summarized as

$$J_{L1} = \sum_{k=0}^{N-1} [x_k^T Q x_k + u_k^T R u_k] \tag{23}$$

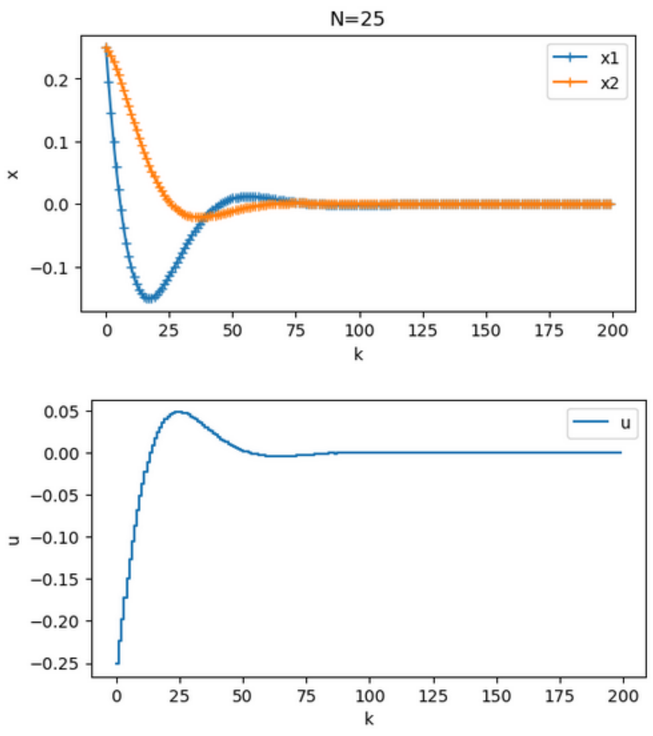


Figure 9: The input and state trajectories for Approach 1 MPC using $N = 25$. As one can see there is practically no difference between choosing $N = 25$ and $N = 16$ as the input and state-trajectories are nearly identical.

$$\text{s.t.} x_{k+1} = f(x_k, u_k)$$
$$x_0 = \bar{x}$$
$$x \in X \qquad .$$
$$u \in U$$
$$x(N) = 0$$

This would hypothetically allow for a quicker transient response. We first need to identify a suitable value for N. This can be done by exploring J vs N like we did in Approach 1 (Figure 11).

We note from this figure that $N = 16$ is a suitable value. The state and input trajectories

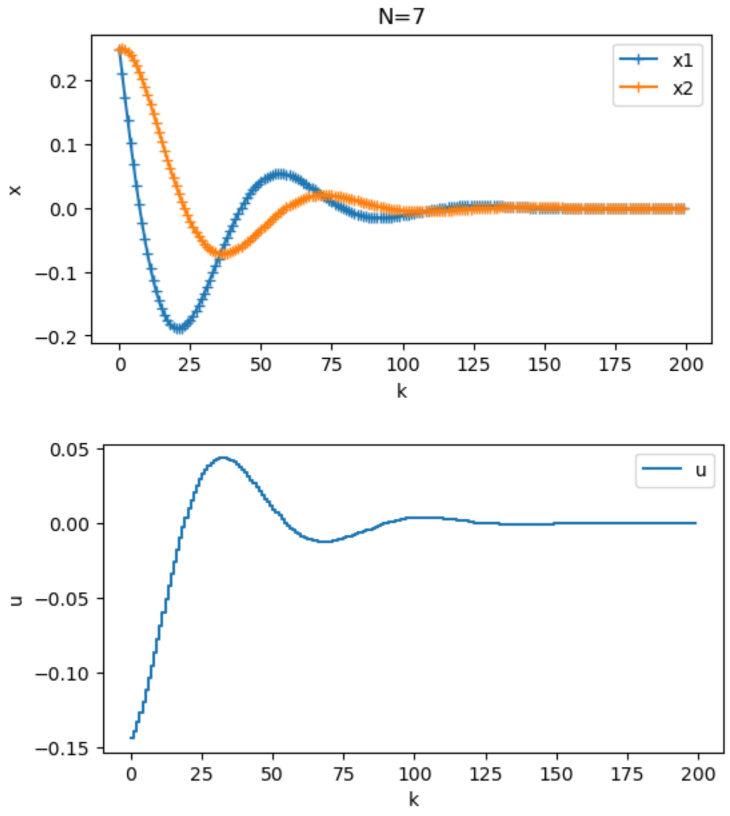


Figure 10: The input and state trajectories for Approach 1 MPC using $N = 7$. The control performance heavily degrades compared to when $N = 16$. One can see for example that the transient response of the state-trajectories takes much longer compared to the state-trajectories when $N = 16$.

are shown below (Figure 14):

One can clearly note there is a much better transient response as compared to Approach 1 (when $N = 16$). Specifically, the system reaches steady-state faster and there is less fluctuations and overshoot present in the transient response. This is especially noticeable in x_2. This is further explained in the **Comments and Future Work** subsection. We include the Approach 2 results for $N = 7$ and $N = 25$ (Figures x and y). As N increases from $N = 16$, there isn't much of an increase in the control performance and when N decreases

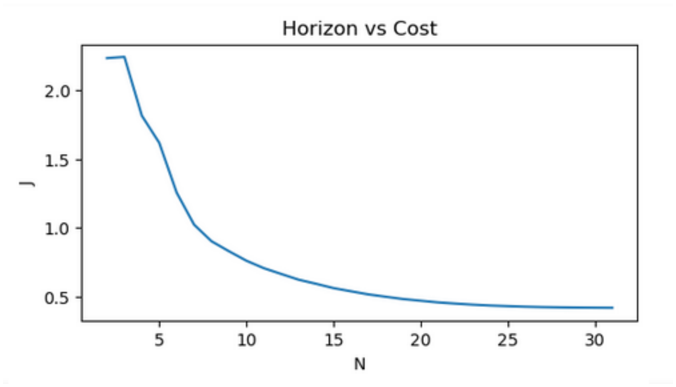


Figure 11: Comparing J for different values of N for the system modeled by (21) (Approach 2).

from $N = 7$, the control performance degrades (this mostly applies to x_1 (**see "Comments and Future Work"** for an explanation on why)).

4.1.3 Comments on Constraints

The initial condition considered $x_0 = [0.25, 0.25]^T$ is far away from any state or input constraint. To see if any feasibility issues may potentially ever arise, we consider Approach 1 and change the initial conditions.

We consider the first extreme initial condition case to be $x_0 = [0.5624, 0.6061]^T$ (maximum allowable values for x_1 and x_2 respectively). Using Approach 1 $N = 16$ yields the following state and input trajectories (Figure 15).

Similarly, we consider the initial condition case $x_0 = [-0.5739, -1.5151]^T$ (minimum allowable values for x_1 and x_2 respectively). Using Approach 1 $N = 16$ yields the following state and input trajectories (Figure 16).

We see that we don't run anywhere near the state or input constraints. This was done intentionally by design in Section 1.

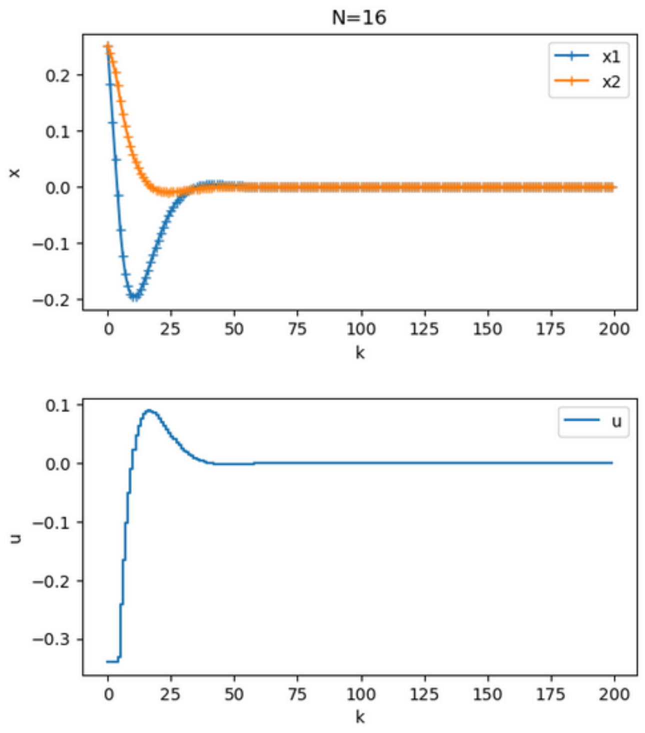


Figure 12: The input and state trajectories for Approach 2 MPC using N=16.

4.2 Nonlinear MPC

Nonlinear MPC uses all the same foundations as the linear MPC above. However, the system function utilized is now $x_{k+1} = f_d(x_k, u_k)$, which is the DT version of the original continuous nonlinear function (7). We present the results of the two approaches. The integration step size is taken to be $\bar{T} = 0.005$

4.2.1 Approach 1

We present the results for the nonlinear Approach 1 method when $N = 2$ in Figure 17. One can see that the fluctuations and overshoot has been eliminated compared to the linear MPC methods in only a horizon length of $N = 2$! This completely outperforms the performance of

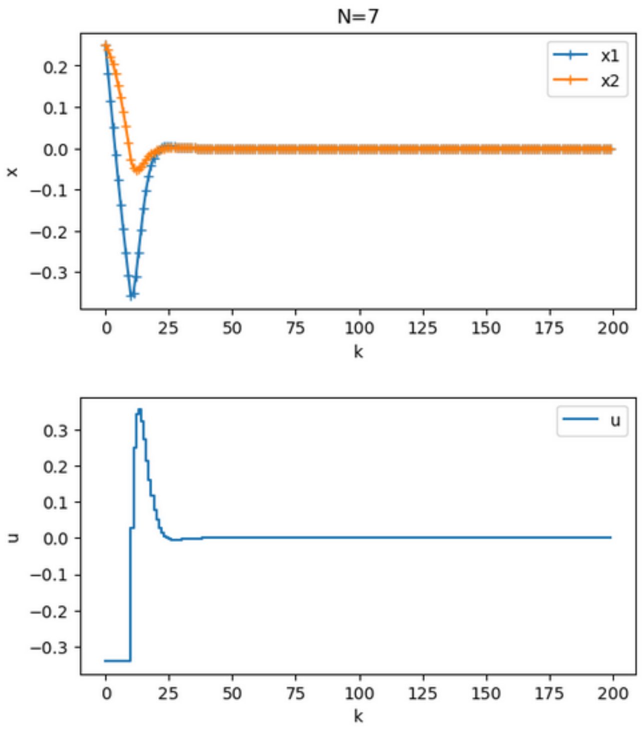


Figure 13: The input and state trajectories for Approach 1 MPC using N=7. x_1 has high transient overshoot, while x_2 has low transient overshoot.

linear MPC methods. The author experimented with higher order horizons, however there was no difference in performance (as expected).

4.2.2 Approach 2

We present the results for the nonlinear Approach 2 method when $N = 2$ in Figure 20. One can see that the fluctuations and overshoot has been eliminated compared to the linear MPC methods in only a horizon length of $N = 2$! This completely outperforms the performance of linear MPC methods. The nonlinear Approach 2 reaches steady-state at the same time as the nonlinear Approach 1 method so there is little to no improvement. The author experimented with higher order horizons, however there was no difference in performance (as expected).

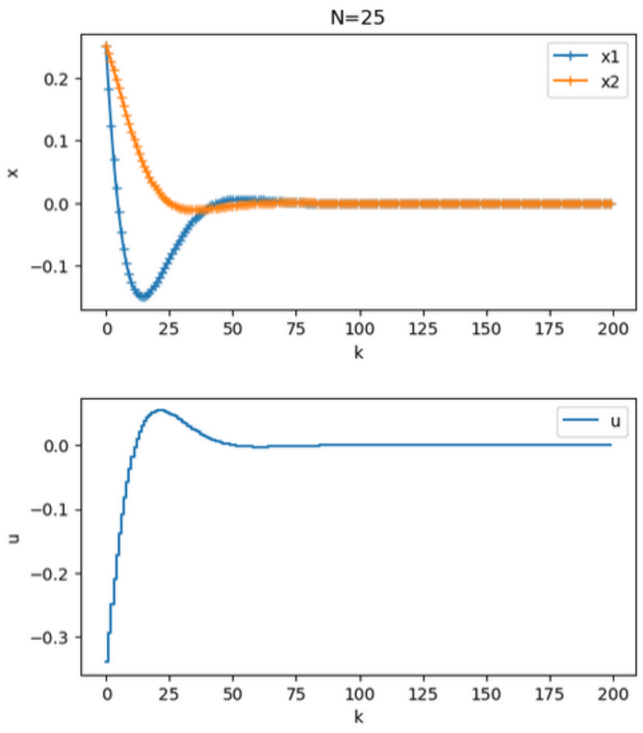


Figure 14: The input and state trajectories for Approach 1 MPC using N=25. x_1 has a reduced transient overshoot compared to $N = 7$.

4.3 Comments and Future Work

In the Linear MPC methods, one can notice that there is a very large overshoot in the x_1 state trajectories as opposed to the x_2 state-trajectory. The author hypothesizes it has to do with the choice of Q chosen

$$Q = \begin{bmatrix} 0.001 & 0 \\ 0 & 1.001 \end{bmatrix}.$$

As one can see q_{22} is much larger than q_{11}. This means that the cost function is "more interested" in optimizing x_2 (even if it comes at the risk of degrading the performance of

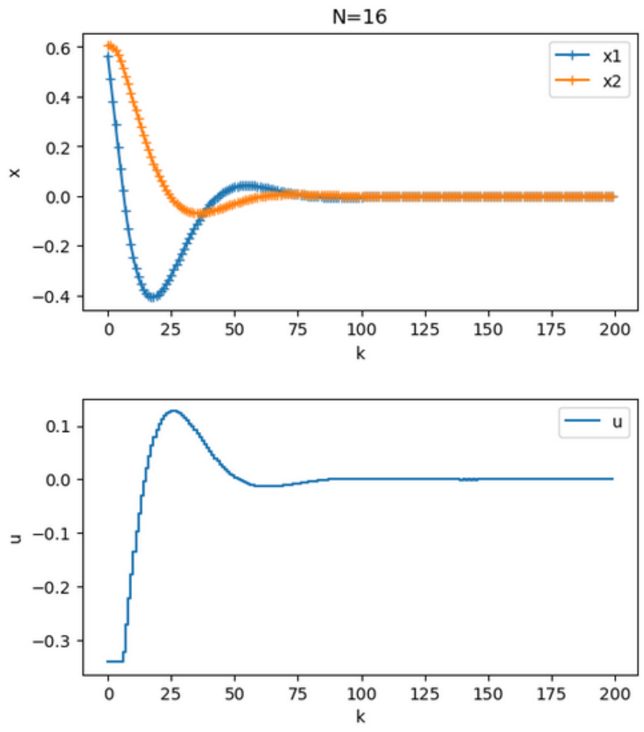


Figure 15: The input and state trajectories for Approach 1 MPC using N=16. $x_0 = [0.5624, 0.6061]^T$.

x_1). In Approach 1 linear MPC, the cost J associated with $N = 7$ was relatively high compared to higher order horizons (see Figure 7.) However, when one takes a look at the actual state-trajectories for linear Approach 1 $N = 7$ (Figure 10), one can see that the cost J is only so high because of the bad state trajectory of x_1. x_2 has a relatively smooth transient response with low overshoot.

The author hypothesizes that this problem can be potentially be fixed by adjusting the coefficients of Q. This would mean the cost function J would to pay more attention to the first state and "optimize it better". One method of this is utilizing Bryson's method or doing a genetic algorithm search [**buck˙lqr˙thesis**]. However, this is beyond the scope of what was

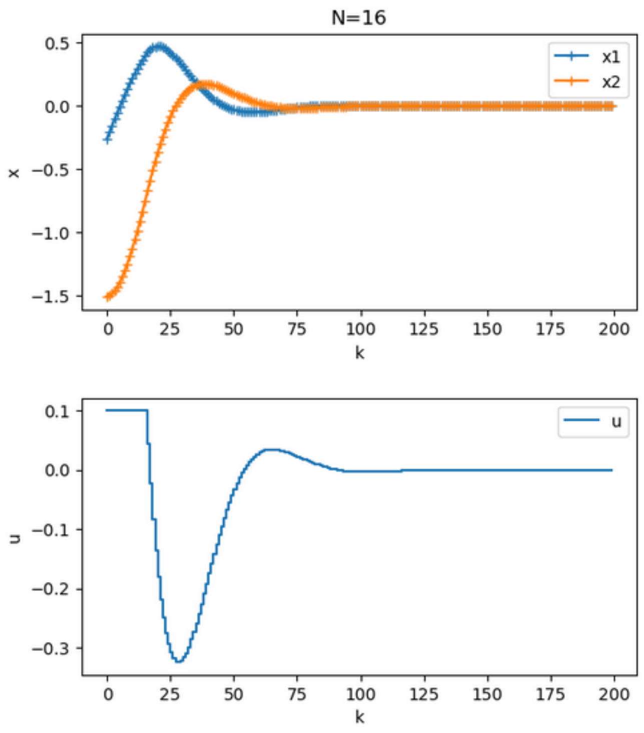


Figure 16: The input and state trajectories for Approach 1 MPC using N=16. $x_0 = [-0.5739, -1.5151]^T$.

discussed in the report (I plan on trying to do this after finals!).

5 Minimum Principle

We first approximate Q as

$$Q = \begin{bmatrix} 0.001 & 0 \\ 0 & 1.001 \end{bmatrix} \approx \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

This is so the Hamiltonian can be simplified. Based on the analysis above, it seems that by $t_f = 16$, the system should have reached steady-state. Hence we define the two boundary

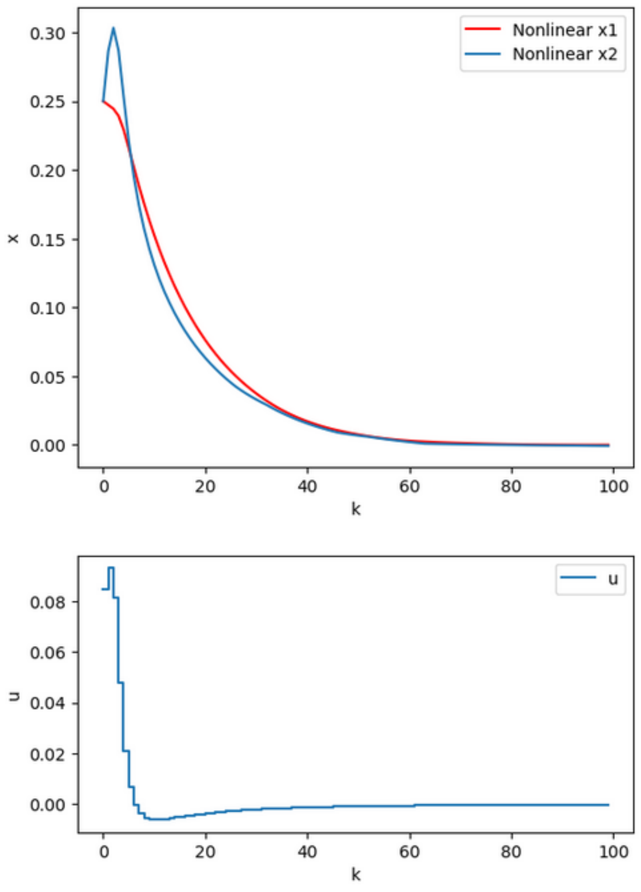


Figure 17: The input and state trajectories for Nonlinear Approach 1 MPC using N=2.

conditions as $x(0) = [0.25, 0.25]^T$ and $x(16) = [0, 0]^T$. The Hamiltonian is

$$\mathcal{H} = x_2^2 + u^2 + p_1(\frac{-x_2}{k} + (x_2 + k)u) + p_2(\frac{x_1}{k} - \alpha x_2 - (x_1 + \alpha k^2)u).$$

The necessary conditions for optimality are:

$$\begin{aligned} \dot{x}^* &= \frac{\partial \mathcal{H}}{\partial p}(x^*, u^*, p^*, t) \\ \dot{p}^* &= -\frac{\partial \mathcal{H}}{\partial x}(x^*, u^*, p^*, t) \quad \forall t \in [0, 16]. \\ 0 &= \frac{\partial \mathcal{H}}{\partial u}(x^*, u^*, p^*, t) \end{aligned}$$

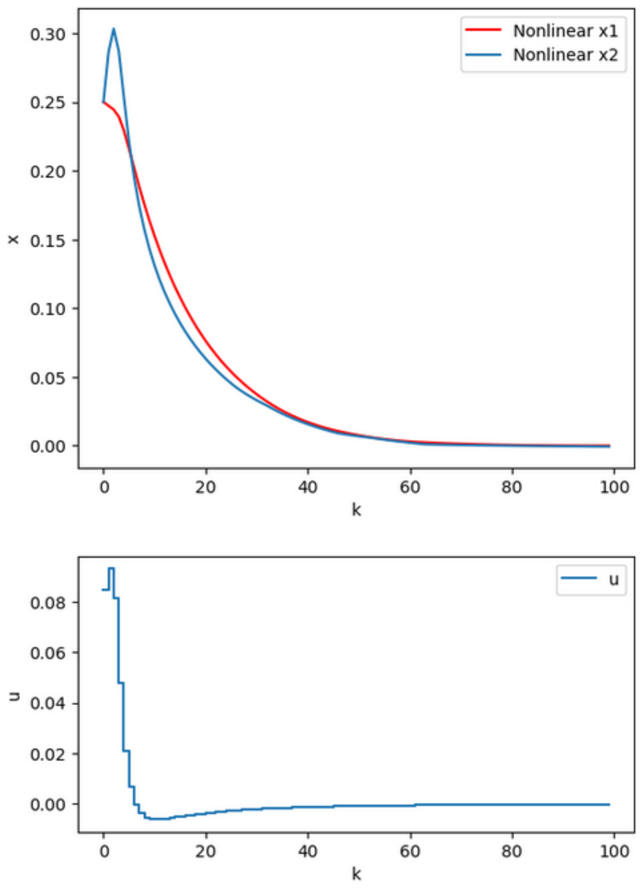


Figure 18: The input and state trajectories for Nonlinear Approach 2 MPC using N=2.

Computing these necessary conditions yields:

$$\begin{aligned}
\dot{x}_1^* &= \frac{-x_2^*}{k} + (x_2^* + k)u^* \\
\dot{x}_2^* &= \frac{x_1^*}{k} - \alpha x_2^* - (x_1^* + \alpha k^2)u^* \\
\dot{p}_1^* &= p_2^* u^* - \frac{p_2^*}{k} \qquad\qquad \forall t \in [0, 16]. \\
\dot{p}_2^* &= \frac{p_1^*}{k} + \alpha p_2^* + 2x_2^* - p_1^* u^* \\
u^* &= \frac{p_2^*(x_1^* + \alpha k^2) - p_1^*((x_2^* + k))}{2}
\end{aligned}$$

We solve the these necessary conditions alongside the boundary conditions using collocation. The trajectories for x_1^*, x_2^*, p_1^*, and p_2^* are presented in Figure x. The input trajectory is presented in presented in Figure y.

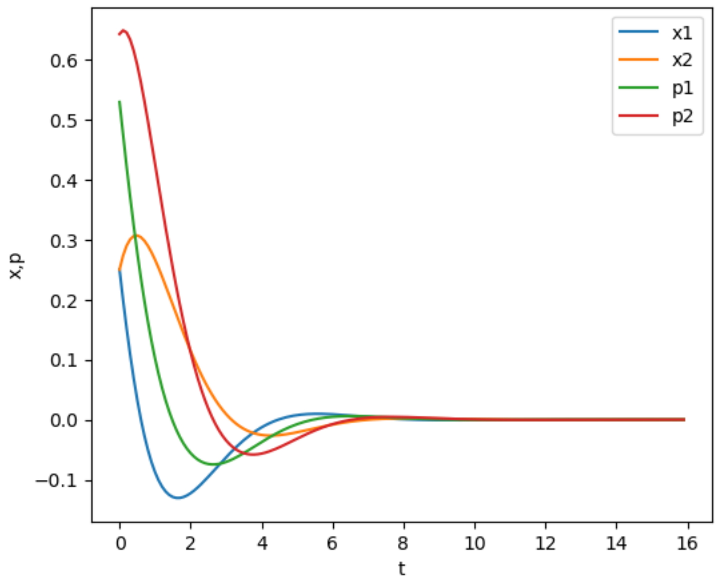


Figure 19: State-trajectories of the TPBVP.

Appendix

This appendix contains all the code utilized in this report. The LQR section was done through MATLAB, while the rest was done in python

MATLAB

Riccati_derivation.m

```
clear; clc; close all
syms p1 p2 p3
P = [p1, p2; p2, p3];
```

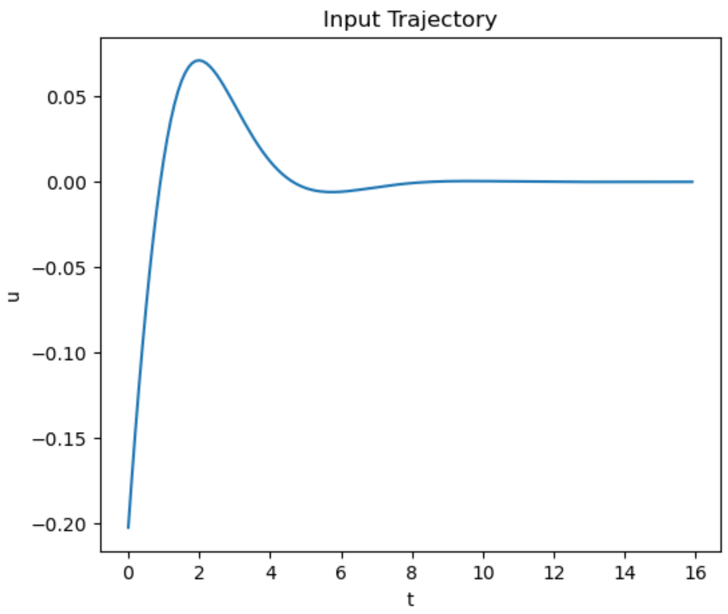


Figure 20: Input trajectory of the TPBVP.

```
A = [0, -0.6579; 0.6579, -0.25];
B = [1.52, 0.5776]';
Q = [0,0;0,1];
R = 1;


Riccati_mat = -P*A-A'*P-Q+P*B*inv(R)*B'*P;
eqn1 = Riccati_mat(1,1) == 0;
eqn2 = Riccati_mat(1,2) == 0;
eqn3 = Riccati_mat(2,2) == 0;
sol = solve(eqn1,eqn2,eqn3, p1, p2, p3);
vpa(sol.p1, 3)
vpa(sol.p2, 3)
vpa(sol.p3, 3)
```

LQR_state_trajectory.m

```
clear; clc; close all
A = [0, -0.6579; 0.6579, -0.25];
B = [1.52, 0.5776]';
K = [0.4459376, 0.6563664];
syms t
x0 = [0.25,0.25]'; %Initial condition
x = expm((A-B*K)*t)*x0;
u = -K*x;
x = vpa(simplify(vpa(simplify(rewrite(x, 'sincos')),16)),4);
figure
fplot(x)
figure
fplot(u)
```

myODEs.m

```
function dydt = myODEs(t, y)
    % Define your system of ODEs here
    dydt = zeros(2, 1); % Preallocate the output vector
    k = 1.52; a = 0.25;
    K = [0.4459376, 0.6563664];
    % Example ODEs
    dydt(1) = -y(2)/k +(y(2)+k)*(-K*[y(1), y(2)]');
    dydt(2) = y(1)/k -a*y(2) -(y(1)+a*k^2)*(-K*[y(1), y(2)]');
end
```

LQR_ODE.m

```
% Set up time span
tspan = [0 20];

% Set initial conditions
y0 = [0.25; 0.25]; % Example initial condition

% Solve the system of ODEs
[t, y] = ode45(@myODEs, tspan, y0);

% Plot the results
figure;
plot(t, y(:, 1), 'b-', t, y(:, 2), 'r-');
xlabel('Time');
ylabel('Solution');
legend('y1', 'y2');
title('Solution of the System of ODEs');
```

Python

MPC Linear - Approach 1

We first define the nonlinear and linearized system. The non-linear system is

$$\begin{aligned}\dot{x}_1 &= \frac{-x_2}{k} + (x_2 + k)u \\ \dot{x}_2 &= \frac{x_1}{k} - \alpha x_2 - (x_1 + \alpha k^2)u\end{aligned}.$$

As mentioned in the report, it has a equilibrium point at $(x_s, u_s) = (0, 0)$. Hence, the linearized system around this equilibrium point is:

$$\dot{x} = \underbrace{\begin{bmatrix} 0 & -0.6579 \\ 0.6579 & -0.25 \end{bmatrix}}_{A} x + \underbrace{\begin{bmatrix} 1.52 \\ 0.5776 \end{bmatrix}}_{B} u \; .$$
$$y = x_2$$

We begin our MPC exploration with the linearized system.

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        from scipy.optimize import fsolve
        from scipy import linalg

        def f(A, B, xk, uk):
            x = np.zeros(shape=(xk.size,1))
            x = A.dot(xk) + B*uk # actual state at the next step
            return x
```

```
In [ ]: A_c = np.array([[0, -0.6579], [0.6579, -0.25]]) #A_c means A matrix in conti
        B_c = np.array([[1.52],[0.5776]]) ##B_c means B matrix in continous domain
        C = np.array([[0, 1]])
        Q = C.T.dot(C)+0.001*np.identity(2)
        T1 = 0.1
        A = linalg.expm(A_c*T1)
        B = np.matmul(np.matmul(linalg.inv(A_c), (linalg.expm(A_c*T1)-np.identity(2)
        eig_Q = np.linalg.eig(Q)
        eig_A = np.linalg.eig(A)
        print(f'The eigenvalues of the Q matrix are: {eig_Q}')
        print(f'The eigenvalues of the A matrix are: {eig_A}')
        R = 1
        umin = -0.34
        umax = 0.66
        print(f'The Ad matrix is: {A}')
        print(f'The Bd matrix is: {B}')
```

```
In [ ]: ## MPC
        from scipy.optimize import minimize
```

```
# the function mpc_obj below defines how the cost function is calculated
def mpc_obj(U, A, B, Q, R, xk, N):
    J = 0.5*np.matmul(np.matmul(xk.T, Q), xk)
    for k in range(0,N):
        uk = U[k] # this U is the optimized input trajectory by 'minimize' u
        xk = f(A,B,xk,uk) # the state at the next step
        J += 0.5*uk*R*uk + 0.5*np.matmul(np.matmul(xk.T, Q), xk)
    return J.flatten()

def mpc(A, B, Q, R, xk, N, umin, umax):
    u_init = np.zeros(N) # initial guess for input trajectory
    u_bounds = ((umin, umax),)*N # bounds on the input trajectory
    # call scipy.optimize.minimize to solve the MPC optimization problem
    # mpc_obj - defines the objective function
    # args - parameters will be passed to mpc_obj
    # there are different methods for solving the optimization problem. Here
    sol = minimize(mpc_obj, u_init, args=(A,B,Q,R,xk,N), method='SLSQP', bou
    # the optimized input trajectory is stored in sol.x
    U = sol.x
    # only the first input value is implemented according to receding horizo
    uk = U[0]
    return uk
```

In [ ]:

```
J = np.zeros(25)
for N in range(2,27):
    ## Closed-loop simulation & predictions
    tk = np.arange(0,200) # simulate for 200 steps
    # each state vector is saved as a 2 by 1 vector - corresponds to the las
    x = np.zeros(shape=(tk.size, 2, 1))
    y = np.zeros(shape=(tk.size,1,1))
    u = np.zeros(shape=(tk.size-1)) # store actually implemented control inp
    x[0,:,:] = np.array([[0.25], [0.25]])  # initial condition

    for k in range(0,tk.size-1):
        # only the first u is actually implemented
        xk = x[k,:,:] # current x
        # uk = 0 # set u = 0 to generate open-loop trajectory
        uk = mpc(A,B,Q,R,xk,N,umin, umax) # calls MPC function to calculate
        u[k] = uk
        x[k+1,:,:] = f(A, B, x[k,:,:], uk) # the state at the next step
        J[N-2] += 0.5*uk*R*uk + 0.5*np.matmul(np.matmul(x[k,:,:].T, Q), x[k,


from matplotlib.pyplot import *
figure(figsize=(6,3))
plot(np.arange(2,27),J)
xlabel('N')
ylabel('J')
title("Horizon vs Cost")
#axis([0, 35, -1.5, 0.5])
```

In [ ]:

```
N = 16

## Closed-loop simulation & predictions
tk = np.arange(0,200) # simulate for 200 steps
```

```
# each state vector is saved as a 2 by 1 vector - corresponds to the last tw
x = np.zeros(shape=(tk.size, 2, 1))
y = np.zeros(shape=(tk.size,1,1))
u = np.zeros(shape=(tk.size-1)) # store actually implemented control inputs
x[0,:,:] = np.array([[-0.5739], [-1.5151]])  # initial condition
J = 0
for k in range(0,tk.size-1):
    # only the first u is actually implemented
    xk = x[k,:,:] # current x
    # uk = 0 # set u = 0 to generate open-loop trajectory
    uk = mpc(A,B,Q,R,xk,N,umin, umax) # calls MPC function to calculate the
    u[k] = uk
    x[k+1,:,:] = f(A, B, x[k,:,:], uk) # the state at the next step
    J += 0.5*uk*R*uk + 0.5*np.matmul(np.matmul(x[k,:,:].T, Q), x[k,:,:])
print(J)
```

```
In [ ]: ## plot results
from matplotlib.pyplot import *
figure(figsize=(6,3))
plot(tk,x[:,0,:]/2.2, '-+', tk, x[:,1,:], '-+')
xlabel('k')
ylabel('x')
legend(['x1', 'x2'])
title("N=16")
#axis([0, 35, -1.5, 0.5])

figure(figsize=(6,3))
uforplotting = np.zeros(shape=tk.size)
uforplotting[0] = u[0]
for k in range(1,uforplotting.size):
    uforplotting[k] = u[k-1]
step(tk,uforplotting)
xlabel('k')
ylabel('u')
legend(['u'])
#axis([0, 35, -0.12, 0.12])
```

```
In [ ]: ## plot results
from matplotlib.pyplot import *
figure(figsize=(6,3))
plot(tk,x[:,0,:], '-+', tk, x[:,1,:], '-+')
xlabel('k')
ylabel('x')
legend(['x1', 'x2'])
title("N=16")
plt.xlim(125, 150)  # Set x-axis limits
plt.ylim(-0.05, 0.05)  # Set y-axis limits
#axis([0, 35, -1.5, 0.5])

figure(figsize=(6,3))
uforplotting = np.zeros(shape=tk.size)
uforplotting[0] = u[0]
for k in range(1,uforplotting.size):
    uforplotting[k] = u[k-1]
```

```
step(tk,uforplotting)
xlabel('k')
ylabel('u')
legend(['u'])
#axis([0, 35, -0.12, 0.12])
```

In [ ]:
```
N = 7

## Closed-loop simulation & predictions
tk = np.arange(0,200) # simulate for 200 steps
# each state vector is saved as a 2 by 1 vector - corresponds to the last tw
x = np.zeros(shape=(tk.size, 2, 1))
y = np.zeros(shape=(tk.size,1,1))
u = np.zeros(shape=(tk.size-1)) # store actually implemented control inputs
x[0,:,:] = np.array([[0.25], [0.25]])  # initial condition

for k in range(0,tk.size-1):
    # only the first u is actually implemented
    xk = x[k,:,:] # current x
    # uk = 0 # set u = 0 to generate open-loop trajectory
    uk = mpc(A,B,Q,R,xk,N,umin, umax) # calls MPC function to calculate the
    u[k] = uk
    x[k+1,:,:] = f(A, B, x[k,:,:], uk) # the state at the next step
```

In [ ]:
```
## plot results
from matplotlib.pyplot import *
figure(figsize=(6,3))
plot(tk,x[:,0,:], '-+', tk, x[:,1,:], '-+')
xlabel('k')
ylabel('x')
legend(['x1', 'x2'])
title("N=7")
#axis([0, 35, -1.5, 0.5])

figure(figsize=(6,3))
uforplotting = np.zeros(shape=tk.size)
uforplotting[0] = u[0]
for k in range(1,uforplotting.size):
    uforplotting[k] = u[k-1]
step(tk,uforplotting)
xlabel('k')
ylabel('u')
legend(['u'])
#axis([0, 35, -0.12, 0.12])
```

In [ ]:
```
N = 12

## Closed-loop simulation & predictions
tk = np.arange(0,200) # simulate for 200 steps
# each state vector is saved as a 2 by 1 vector - corresponds to the last tw
x = np.zeros(shape=(tk.size, 2, 1))
y = np.zeros(shape=(tk.size,1,1))
u = np.zeros(shape=(tk.size-1)) # store actually implemented control inputs
x[0,:,:] = np.array([[0.25], [0.25]])  # initial condition
```

```
for k in range(0,tk.size-1):
    # only the first u is actually implemented
    xk = x[k,:,:] # current x
    # uk = 0 # set u = 0 to generate open-loop trajectory
    uk = mpc(A,B,Q,R,xk,N,umin, umax) # calls MPC function to calculate the
    u[k] = uk
    x[k+1,:,:] = f(A, B, x[k,:,:], uk) # the state at the next step
```

In [ ]:
```
## plot results
from matplotlib.pyplot import *
figure(figsize=(6,3))
plot(tk,x[:,0,:], '-+', tk, x[:,1,:], '-+')
xlabel('k')
ylabel('x')
title("N=12")

legend(['x1', 'x2'])
#axis([0, 35, -1.5, 0.5])

figure(figsize=(6,3))
uforplotting = np.zeros(shape=tk.size)
uforplotting[0] = u[0]
for k in range(1,uforplotting.size):
    uforplotting[k] = u[k-1]
step(tk,uforplotting)
xlabel('k')
ylabel('u')
legend(['u'])
#axis([0, 35, -0.12, 0.12])
```

In [ ]:
```
N = 2

## Closed-loop simulation & predictions
tk = np.arange(0,400) # simulate for 200 steps
# each state vector is saved as a 2 by 1 vector - corresponds to the last tw
x = np.zeros(shape=(tk.size, 2, 1))
y = np.zeros(shape=(tk.size,1,1))
u = np.zeros(shape=(tk.size-1)) # store actually implemented control inputs
x[0,:,:] = np.array([[0.25], [0.25]])  # initial condition


for k in range(0,tk.size-1):
    # only the first u is actually implemented
    xk = x[k,:,:] # current x
    # uk = 0 # set u = 0 to generate open-loop trajectory
    uk = mpc(A,B,Q,R,xk,N,umin, umax) # calls MPC function to calculate the
    u[k] = uk
    x[k+1,:,:] = f(A, B, x[k,:,:], uk) # the state at the next step
```

In [ ]:
```
## plot results
from matplotlib.pyplot import *
figure(figsize=(6,3))
plot(tk,x[:,0,:], '-+', tk, x[:,1,:], '-+')
xlabel('k')
```

```
ylabel('x')
legend(['x1', 'x2'])
title("N=2")
#axis([0, 35, -1.5, 0.5])

figure(figsize=(6,3))
uforplotting = np.zeros(shape=tk.size)
uforplotting[0] = u[0]
for k in range(1,uforplotting.size):
    uforplotting[k] = u[k-1]
step(tk,uforplotting)
xlabel('k')
ylabel('u')
legend(['u'])
#axis([0, 35, -0.12, 0.12])
```

In [ ]:
```
N = 25

## Closed-loop simulation & predictions
tk = np.arange(0,400) # simulate for 200 steps
# each state vector is saved as a 2 by 1 vector - corresponds to the last tw
x = np.zeros(shape=(tk.size, 2, 1))
y = np.zeros(shape=(tk.size,1,1))
u = np.zeros(shape=(tk.size-1)) # store actually implemented control inputs
x[0,:,:] = np.array([[0.25], [0.25]])  # initial condition


for k in range(0,tk.size-1):
    # only the first u is actually implemented
    xk = x[k,:,:] # current x
    # uk = 0 # set u = 0 to generate open-loop trajectory
    uk = mpc(A,B,Q,R,xk,N,umin, umax) # calls MPC function to calculate the
    u[k] = uk
    x[k+1,:,:] = f(A, B, x[k,:,:], uk) # the state at the next step
```

In [ ]:
```
## plot results
from matplotlib.pyplot import *
figure(figsize=(6,3))
plot(tk,x[:,0,:], '-+', tk, x[:,1,:], '-+')
xlabel('k')
ylabel('x')
legend(['x1', 'x2'])
title("N=25")
#axis([0, 35, -1.5, 0.5])

figure(figsize=(6,3))
uforplotting = np.zeros(shape=tk.size)
uforplotting[0] = u[0]
for k in range(1,uforplotting.size):
    uforplotting[k] = u[k-1]
step(tk,uforplotting)
xlabel('k')
ylabel('u')
legend(['u'])
#axis([0, 35, -0.12, 0.12])
```

MPC Linear - Approach 2

We first define the nonlinear and linearized system. The non-linear system is

$$\dot{x}_1 = \frac{-x_2}{k} + (x_2 + k)u$$
$$\dot{x}_2 = \frac{x_1}{k} - \alpha x_2 - (x_1 + \alpha k^2)u \, .$$

As mentioned in the report, it has a equilibrium point at $(x_s, u_s) = (0, 0)$. Hence, the linearized system around this equilibrium point is:

$$\dot{x} = \underbrace{\begin{bmatrix} 0 & -0.6579 \\ 0.6579 & -0.25 \end{bmatrix}}_{A} x + \underbrace{\begin{bmatrix} 1.52 \\ 0.5776 \end{bmatrix}}_{B} u \, .$$
$$y = x_2$$

We begin our MPC exploration with the linearized system.

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        from scipy.optimize import fsolve
        from scipy import linalg

        def f(A, B, xk, uk):
            x = np.zeros(shape=(xk.size,1))
            x = A.dot(xk) + B*uk # actual state at the next step
            return x
```

```
In [ ]: A_c = np.array([[0, -0.6579], [0.6579, -0.25]]) #A_c means A matrix in conti
        B_c = np.array([[1.52],[0.5776]]) ##B_c means B matrix in continous domain
        C = np.array([[0, 1]])
        Q = C.T.dot(C)+0.001*np.identity(2)
        T1 = 0.1
        A = linalg.expm(A_c*T1)
        B = np.matmul(np.matmul(linalg.inv(A_c), (linalg.expm(A_c*T1)-np.identity(2)
        eig_Q = np.linalg.eig(Q)
        eig_A = np.linalg.eig(A)
        print(f'The eigenvalues of the Q matrix are: {eig_Q}')
        print(f'The eigenvalues of the A matrix are: {eig_A}')
        R = 1
        umin = -0.34
        umax = 0.66
        print(f'The Ad matrix is: {A}')
        print(f'The Bd matrix is: {B}')
```

```
In [ ]: ## MPC
        from scipy.optimize import minimize, NonlinearConstraint
```

```
# the function mpc_obj below defines how the cost function is calculated
def mpc_obj(U, A, B, Q, R, xk, N):
    J = 0.5*np.matmul(np.matmul(xk.T, Q), xk)
    for k in range(0,N):
        uk = U[k] # this U is the optimized input trajectory by 'minimize' u
        xk = f(A,B,xk,uk) # the state at the next step
        J += 0.5*uk*R*uk + 0.5*np.matmul(np.matmul(xk.T, Q), xk)
    return J.flatten()

# def more complicated constraints for mpc
def mpc_con(U, A, B, xk, N, index):
    # index is used to indicate the specific constraint
    for k in range(0,N):
        uk = U[k] # this U is the optimized input trajectory by 'minimize' u
        xk = f(A,B,xk,uk) # the state at the next step
    return xk[index]

def mpc(A, B, Q, R, xk, N, umin, umax):
    u_init = np.zeros(N) # initial guess for input trajectory
    u_bounds = ((umin, umax),)*N # bounds on the input trajectory
    # set up constraints for scipy.minimize
    con = [{'type': 'eq', 'fun': mpc_con, 'args': (A, B, xk, N, 0)}, # index
           {'type': 'eq', 'fun': mpc_con, 'args': (A, B, xk, N, 1)}]  # inde
    # call scipy.optimize.minimize to solve the MPC optimization problem
    sol = minimize(mpc_obj, u_init, args=(A,B,Q,R,xk,N), method='SLSQP', bou
    # the optimized input trajectory is stored in sol.x
    U = sol.x
    # only the first input value is implemented according to receding horizo
    uk = U[0]
    return uk
```

In [ ]:

```
J = np.zeros(30)
for N in range(2,32):
    ## Closed-loop simulation & predictions
    tk = np.arange(0,200) # simulate for 200 steps
    # each state vector is saved as a 2 by 1 vector - corresponds to the las
    x = np.zeros(shape=(tk.size, 2, 1))
    y = np.zeros(shape=(tk.size,1,1))
    u = np.zeros(shape=(tk.size-1)) # store actually implemented control inp
    x[0,:,:] = np.array([[0.25], [0.25]])  # initial condition

    for k in range(0,tk.size-1):
        # only the first u is actually implemented
        xk = x[k,:,:] # current x
        # uk = 0 # set u = 0 to generate open-loop trajectory
        uk = mpc(A,B,Q,R,xk,N,umin, umax) # calls MPC function to calculate
        u[k] = uk
        x[k+1,:,:] = f(A, B, x[k,:,:], uk) # the state at the next step
        J[N-2] += 0.5*uk*R*uk + 0.5*np.matmul(np.matmul(x[k,:,:].T, Q), x[k,


from matplotlib.pyplot import *
figure(figsize=(6,3))
plot(np.arange(2,32),J)
xlabel('N')
ylabel('J')
```

```
title("Horizon vs Cost")
#axis([0, 35, -1.5, 0.5])
```

In [ ]:
```
N = 7

## Closed-loop simulation & predictions
tk = np.arange(0,200) # simulate for 200 steps
# each state vector is saved as a 2 by 1 vector - corresponds to the last tw
x = np.zeros(shape=(tk.size, 2, 1))
y = np.zeros(shape=(tk.size,1,1))
u = np.zeros(shape=(tk.size-1)) # store actually implemented control inputs
x[0,:,:] = np.array([[0.25], [0.25]])  # initial condition

J = 0
for k in range(0,tk.size-1):
    # only the first u is actually implemented
    xk = x[k,:,:] # current x
    # uk = 0 # set u = 0 to generate open-loop trajectory
    uk = mpc(A,B,Q,R,xk,N,umin, umax) # calls MPC function to calculate the
    u[k] = uk
    x[k+1,:,:] = f(A, B, x[k,:,:], uk) # the state at the next step
    J += 0.5*uk*R*uk + 0.5*np.matmul(np.matmul(x[k,:,:].T, Q), x[k,:,:])
print(J)
```

In [ ]:
```
## plot results
from matplotlib.pyplot import *
figure(figsize=(6,3))
plot(tk,x[:,0,:], '-+', tk, x[:,1,:], '-+')
xlabel('k')
ylabel('x')
legend(['x1', 'x2'])
title("N=7")
#axis([0, 35, -1.5, 0.5])

figure(figsize=(6,3))
uforplotting = np.zeros(shape=tk.size)
uforplotting[0] = u[0]
for k in range(1,uforplotting.size):
    uforplotting[k] = u[k-1]
step(tk,uforplotting)
xlabel('k')
ylabel('u')
legend(['u'])
#axis([0, 35, -0.12, 0.12])
```

In [ ]:
```
N = 2

## Closed-loop simulation & predictions
tk = np.arange(0,200) # simulate for 200 steps
# each state vector is saved as a 2 by 1 vector - corresponds to the last tw
x = np.zeros(shape=(tk.size, 2, 1))
y = np.zeros(shape=(tk.size,1,1))
u = np.zeros(shape=(tk.size-1)) # store actually implemented control inputs
x[0,:,:] = np.array([[0.25], [0.25]])  # initial condition
```

```python


for k in range(0,tk.size-1):
    # only the first u is actually implemented
    xk = x[k,:,:] # current x
    # uk = 0 # set u = 0 to generate open-loop trajectory
    uk = mpc(A,B,Q,R,xk,N,umin, umax) # calls MPC function to calculate the
    u[k] = uk
    x[k+1,:,:] = f(A, B, x[k,:,:], uk) # the state at the next step
```

In [ ]:
```python
## plot results
from matplotlib.pyplot import *
figure(figsize=(6,3))
plot(tk,x[:,0,:], '-+', tk, x[:,1,:], '-+')
xlabel('k')
ylabel('x')
title("N=2")

legend(['x1', 'x2'])
#axis([0, 35, -1.5, 0.5])

figure(figsize=(6,3))
uforplotting = np.zeros(shape=tk.size)
uforplotting[0] = u[0]
for k in range(1,uforplotting.size):
    uforplotting[k] = u[k-1]
step(tk,uforplotting)
xlabel('k')
ylabel('u')
legend(['u'])
#axis([0, 35, -0.12, 0.12])
```

In [ ]:
```python
N = 16

## Closed-loop simulation & predictions
tk = np.arange(0,200) # simulate for 200 steps
# each state vector is saved as a 2 by 1 vector - corresponds to the last tw
x = np.zeros(shape=(tk.size, 2, 1))
y = np.zeros(shape=(tk.size,1,1))
u = np.zeros(shape=(tk.size-1)) # store actually implemented control inputs
x[0,:,:] = np.array([[0.25], [0.25]])  # initial condition


for k in range(0,tk.size-1):
    # only the first u is actually implemented
    xk = x[k,:,:] # current x
    # uk = 0 # set u = 0 to generate open-loop trajectory
    uk = mpc(A,B,Q,R,xk,N,umin, umax) # calls MPC function to calculate the
    u[k] = uk
    x[k+1,:,:] = f(A, B, x[k,:,:], uk) # the state at the next step
```

In [ ]:
```python
## plot results
from matplotlib.pyplot import *
figure(figsize=(6,3))
plot(tk,x[:,0,:], '-+', tk, x[:,1,:], '-+')
```

```
xlabel('k')
ylabel('x')
legend(['x1', 'x2'])
title("N=16")
#axis([0, 35, -1.5, 0.5])

figure(figsize=(6,3))
uforplotting = np.zeros(shape=tk.size)
uforplotting[0] = u[0]
for k in range(1,uforplotting.size):
    uforplotting[k] = u[k-1]
step(tk,uforplotting)
xlabel('k')
ylabel('u')
legend(['u'])
#axis([0, 35, -0.12, 0.12])
```

MPC NonLinear - Approach 1

We first define the nonlinear and linearized system. The non-linear system is

$$\dot{x}_1 = \frac{-x_2}{k} + (x_2 + k)u$$
$$\dot{x}_2 = \frac{x_1}{k} - \alpha x_2 - (x_1 + \alpha k^2)u.$$

As mentioned in the report, it has a equilibrium point at $(x_s, u_s) = (0, 0)$. Hence, the linearized system around this equilibrium point is:

$$\dot{x} = \underbrace{\begin{bmatrix} 0 & -0.6579 \\ 0.6579 & -0.25 \end{bmatrix}}_{A} x + \underbrace{\begin{bmatrix} 1.52 \\ 0.5776 \end{bmatrix}}_{B} u.$$
$$y = x_2$$

We begin our MPC exploration with the linearized system.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
from scipy import linalg
from scipy.optimize import minimize

def ode(xk, uk):
    x1k = xk[0]
    x2k = xk[1]
    k_c = 1.52
    a = 0.25
    dx = np.zeros(shape=(xk.size,1))
    dx[0] = -x2k/k_c +(x2k+k_c)*uk
    dx[1] = x1k/k_c -a*x2k-(x1k+a*(k**2))*uk
    return dx
```

In [ ]:
```
A = np.array([[0, -0.6579], [0.6579, -0.25]]) #A means A matrix in continous
B = np.array([[1.52],[0.5776]]) ##B means B matrix in continous domain
print(f'A matrix =: {A}')
print(f'B matrix =: {B}')
```

In [ ]:
```
# discretize the linearized system to find Ad, Bd
T = 0.5 # sampling time
Ad = linalg.expm(A*T)
Bd = np.matmul(np.matmul(linalg.inv(A), (linalg.expm(A*T)-np.identity(2))),B
print(f'Ad matrix =: {Ad}')
print(f'Bd matrix =: {Bd}')
xs = np.array([[0],[0]])
us = np.array([0])
```

In [ ]:
```
# open-loop simulations to investigate the linearized system
dt = 0.1
t = np.arange(0,200,dt)
x_c = np.zeros(shape=(t.size, 2, 1)) # x_c saves states of the original CT n
x_l = np.zeros(shape=(t.size, 2, 1)) # x_l saves states of the linearized CT
x_c[0,:,:] = 0.25
x_l[0,:,:] = 0.25

for k in range(1,x_c.shape[0]):  # evaluate the solution using a loop
    xk = x_c[k-1,:,:]
    xk2 = x_l[k-1,:,:] - xs # linearized model is based on deviation variabl
    uk = us
    uk2 = us - us # linearized model is based on deviation variables
    x_c[k,:,:] = xk + ode(xk,uk)*dt
    x_l[k,:,:] = xk2 + (A.dot(xk2) + B*uk2)*dt + xs # linearized model is ba

plt.figure(figsize=(6,5))
plt.plot(t,x_c[:,0,:],'r', t, x_l[:,0,:], t,x_c[:,1,:], t, x_l[:,1,:])
plt.xlabel('t')
plt.ylabel('x')
plt.legend(['Nonlinear x1', 'Linearized x1', 'Nonlinear x2', 'Linearized x2'
print(np.max(np.absolute(x_c[:,0,:]-x_l[:,0,:]))) # We note the nonlinear mo
```

In [ ]:
```
# MPC designed based on nonlinear model
from scipy.optimize import minimize

# define the model used in MPC
def f(A, B, xk, uk):
    x = np.zeros(shape=(xk.size,1))
    #x = np.matmul(A, xk) + B*uk # actual state at the next step
    x1k = xk[0]
    x2k = xk[1]
    k_c = 1.52
    a = 0.25
    x[0] = -x2k/k_c +(x2k+k_c)*uk
    x[1] = x1k/k_c -a*x2k-(x1k+a*(k**2))*uk
    return x

# the function mpc_obj below defines how the cost function is calculated
def mpc_obj(U, A, B, Q, R, xk, N):
    J = 0.5*np.matmul(np.matmul(xk.T, Q), xk)
    for k in range(0,N):
        uk = U[k] # this U is the optimized input trajectory by 'minimize' u
        xk = f(A,B,xk,uk) # the state at the next step
        J += 0.5*uk*R*uk + 0.5*np.matmul(np.matmul(xk.T, Q), xk)
    return J.flatten()

def mpc(A, B, Q, R, xk, N, umax, umin):
    u_init = np.zeros(N) # initial guess for input trajectory
    u_bounds = ((umin, umax),)*N # bounds on the input trajectory
    # call scipy.optimize.minimize to solve the MPC optimization problem
    sol = minimize(mpc_obj, u_init, args=(A,B,Q,R,xk,N), method='SLSQP', bou
    # the optimized input trajectory is stored in sol.x
    U = sol.x
    # only the first input value is implemented according to receding horizo
    uk = U[0]
    return uk
```

In [ ]:
```
## MPC parameters
C = np.array([[0, 1]])
Q = C.T.dot(C)+0.001*np.identity(2)
R = np.array([1])
umax = 0.66
umin = -0.34
N = 2
dt = 0.005
## Closed-loop simulation & predictions
tk = np.arange(0,100) # simulate for 200 steps
x = np.zeros(shape=(tk.size, 2, 1))
u = np.zeros(shape=(tk.size-1)) # store actually implemented control inputs
x[0,:,:] = np.array([[0.25], [0.25]])  # initial condition
for k in range(0,tk.shape[0]-1):
    # only the first u is actually implemented
    xk = x[k,:,:] - xs # current x for the linearized model
    uk = mpc(Ad,Bd,Q,R,xk,N,umax-us,umin-us) # calls MPC function to calcula
    u[k] = uk + us[0] # uk is for the linearized system; convert it back to

    # use Euler's method to integrate the nonlinear system for one sampling
    xck = x[k,:,:]
    for i in range(0, int(T/dt)):
        xck1 = xck + ode(xck,u[k])*dt
        print(xck1)
        xck = xck1
    x[k+1,:,:] = xck1
    print(xck1)
```

In [ ]:
```
plt.figure(figsize=(6,5))
plt.plot(tk,x[:,0,:],'r', tk,x[:,1,:])
plt.xlabel('k')
plt.ylabel('x')
plt.legend(['Nonlinear x1', 'Nonlinear x2'])
plt.figure(figsize=(6,3))
uforplotting = np.zeros(shape=tk.size)
uforplotting[0] = u[0]
for k in range(1,uforplotting.size):
    uforplotting[k] = u[k-1]
plt.step(tk,uforplotting)
plt.xlabel('k')
plt.ylabel('u')
plt.legend(['u'])
```

In [ ]:
```
import numpy as np

def colsim_tpbvp(pinit,m,n,ffun,T):
    pinit = np.reshape(pinit,(3*m,n))
    p = np.zeros(shape=(3*m,n))
    for k in range(0,3*m):
        p[k,:] = pinit[k,:]
    pfun = np.zeros(shape=(3*m,n))
    for k in range(0,m):
        if k == 0:
            ## --------------------------------------------------------------
            ## BOUNDARY CONDITIONS ARE ENTERED HERE DIRECTLY
            ## p[0,:] denotes the state at the initial time
            ## p[3*m-1,:] denotes the state at the final time
            ## --------------------------------------------------------------
            pfun[k*3,0] = p[0,0] - 0.25   # x1(0) = 1
            pfun[k*3,1] = p[0,1] - 0.25 # x1(pi/2) = 1
            pfun[k*3,2] = p[3*m-1,0]
            pfun[k*3,3] = p[3*m-1,1]
        else:
            pfun[k*3,:] = p[k*3,:] - p[k*3-1,:]
        pfun[k*3+1,:] = -2.003*p[k*3,:] + 1.504*p[k*3+1,:] + 0.499*p[k*3+2,:
        pfun[k*3+2,:] = 2.003*p[k*3,:] - 4.502*p[k*3+1,:] + 2.499*p[k*3+2,:]
    return np.reshape(pfun,(3*m*n,1)).squeeze()
```

In [ ]:
```
from scipy import optimize

# ---------------------------------------------------------------------
# define ffun that will be used in defining the collocation conditions
# ---------------------------------------------------------------------
def f(x0):
    x = np.zeros(shape=(x0.size))
    x1k = x0[0]
    x2k = x0[1]
    p1 = x0[2]
    p2 = x0[3]
    k_c = 1.52
    a = 0.25
    uk = (p2*(x1k+a*(k_c**2))-p1*(x2k+k_c))/2
    x[0] = -x2k/k_c +(x2k+k_c)*uk
    x[1] = x1k/k_c -a*x2k-(x1k+a*(k_c**2))*uk
    x[2] = p2*uk-p2/k_c
    x[3] = p1/k_c + a*p2 - p1*uk - 2*x2k
    return np.squeeze(x)

# ---------------------------------------------------------------------
# settings
# ---------------------------------------------------------------------
T = 0.1 # step size
#tf = 3.14/2+T
tf = 16
t = np.arange(0.0,tf,T)
n = 4
m = int(tf/T)
```

```
# ----------------------------------------------------------------------
# Simultaneous Approach
# ----------------------------------------------------------------------
x = np.zeros(shape=(t.size,n))  # create x for saving results
# use zero for the initial guesses for all the variables
# this may be changed
xinit = np.zeros(shape=(3*(t.size),n))
# call fsolve to solve the collocation conditions simultaneously
temp = optimize.fsolve(colsim_tpbvp, xinit, args=(m, n, f, T))
temp = np.reshape(temp, (3*m,n))
for k in range(0, m):
    x[k,:] = temp[3*k,:]

print(t)
```

In [ ]:

```
from matplotlib.pyplot import *
import math

x1 = x[:,0]
x2 = x[:,1]
p1 = x[:,2]
p2 = x[:,3]

# plot results
figure(figsize=(6,5))
plot(t,x1, t, x2, t, p1, t, p2)
xlabel('t')
ylabel('x,p')
legend(['x1', 'x2', 'p1', 'p2'])

a = 0.25
k_c = 1.52
uk = (p2*(x1+a*(k_c**2))-p1*(x2+k_c))/2
figure(figsize=(6,5))
plot(t,uk)
title("Input Trajectory")
xlabel('t')
ylabel('u')
```

In [ ]:

In [ ]:

MPC NonLinear - Approach 2

We first define the nonlinear and linearized system. The non-linear system is

$$\begin{aligned}\dot{x}_1 &= \frac{-x_2}{k} + (x_2 + k)u \\ \dot{x}_2 &= \frac{x_1}{k} - \alpha x_2 - (x_1 + \alpha k^2)u\end{aligned}.$$

As mentioned in the report, it has a equilibrium point at $(x_s, u_s) = (0, 0)$. Hence, the linearized system around this equilibrium point is:

$$\dot{x} = \underbrace{\begin{bmatrix} 0 & -0.6579 \\ 0.6579 & -0.25 \end{bmatrix}}_{A} x + \underbrace{\begin{bmatrix} 1.52 \\ 0.5776 \end{bmatrix}}_{B} u .$$

$$y = x_2$$

We begin our MPC exploration with the linearized system.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
from scipy import linalg
from scipy.optimize import minimize

def ode(xk, uk):
    x1k = xk[0]
    x2k = xk[1]
    k_c = 1.52
    a = 0.25
    dx = np.zeros(shape=(xk.size,1))
    dx[0] = -x2k/k_c +(x2k+k_c)*uk
    dx[1] = x1k/k_c -a*x2k-(x1k+a*(k**2))*uk
    return dx
```

In [ ]:
```
A = np.array([[0, -0.6579], [0.6579, -0.25]]) #A means A matrix in continous
B = np.array([[1.52],[0.5776]]) ##B means B matrix in continous domain
print(f'A matrix =: {A}')
print(f'B matrix =: {B}')
```

In [ ]:
```
# discretize the linearized system to find Ad, Bd
T = 0.5 # sampling time
Ad = linalg.expm(A*T)
Bd = np.matmul(np.matmul(linalg.inv(A), (linalg.expm(A*T)-np.identity(2))),B
print(f'Ad matrix =: {Ad}')
print(f'Bd matrix =: {Bd}')
xs = np.array([[0],[0]])
us = np.array([0])
```

In [ ]:
```
# open-loop simulations to investigate the linearized system
dt = 0.1
t = np.arange(0,200,dt)
x_c = np.zeros(shape=(t.size, 2, 1)) # x_c saves states of the original CT n
x_l = np.zeros(shape=(t.size, 2, 1)) # x_l saves states of the linearized CT
x_c[0,:,:] = 0.25
x_l[0,:,:] = 0.25

for k in range(1,x_c.shape[0]):  # evaluate the solution using a loop
    xk = x_c[k-1,:,:]
    xk2 = x_l[k-1,:,:] - xs # linearized model is based on deviation variabl
    uk = us
    uk2 = us - us # linearized model is based on deviation variables
    x_c[k,:,:] = xk + ode(xk,uk)*dt
    x_l[k,:,:] = xk2 + (A.dot(xk2) + B*uk2)*dt + xs # linearized model is ba

plt.figure(figsize=(6,5))
plt.plot(t,x_c[:,0,:],'r', t, x_l[:,0,:], t,x_c[:,1,:], t, x_l[:,1,:])
plt.xlabel('t')
plt.ylabel('x')
plt.legend(['Nonlinear x1', 'Linearized x1', 'Nonlinear x2', 'Linearized x2'
print(np.max(np.absolute(x_c[:,0,:]-x_l[:,0,:]))) # We note the nonlinear mo
```

In [ ]:
```
# MPC designed based on nonlinear DT model
from scipy.optimize import minimize

# define the model used in MPC
def f(A, B, xk, uk):
    x = np.zeros(shape=(xk.size,1))
    #x = np.matmul(A, xk) + B*uk # actual state at the next step
    x1k = xk[0]
    x2k = xk[1]
    k_c = 1.52
    a = 0.25
    x[0] = -x2k/k_c +(x2k+k_c)*uk
    x[1] = x1k/k_c -a*x2k-(x1k+a*(k**2))*uk
    return x

# the function mpc_obj below defines how the cost function is calculated
def mpc_obj(U, A, B, Q, R, xk, N):
    J = 0.5*np.matmul(np.matmul(xk.T, Q), xk)
    for k in range(0,N):
        uk = U[k] # this U is the optimized input trajectory by 'minimize' u
        xk = f(A,B,xk,uk) # the state at the next step
        J += 0.5*uk*R*uk + 0.5*np.matmul(np.matmul(xk.T, Q), xk)
    return J.flatten()

def mpc_con(U, A, B, xk, N, index):
    # index is used to indicate the specific constraint
    for k in range(0,N):
        uk = U[k] # this U is the optimized input trajectory by 'minimize' u
        xk = f(A,B,xk,uk) # the state at the next step
    return xk[index]

def mpc(A, B, Q, R, xk, N, umax, umin):
    u_init = np.zeros(N) # initial guess for input trajectory
    u_bounds = ((umin, umax),)*N # bounds on the input trajectory
    # call scipy.optimize.minimize to solve the MPC optimization problem
    sol = minimize(mpc_obj, u_init, args=(A,B,Q,R,xk,N), method='SLSQP', bou
    # the optimized input trajectory is stored in sol.x
    U = sol.x
    # only the first input value is implemented according to receding horizo
    uk = U[0]
    return uk
```

In [ ]:

```
## MPC parameters
C = np.array([[0, 1]])
#Q = C.T.dot(C)+0.01*np.identity(2)
Q = np.array([[1, 0], [0, 1]])
R = np.array([1])
umax = 0.66
umin = -0.34
N = 2
dt = 0.005
## Closed-loop simulation & predictions
tk = np.arange(0,100) # simulate for 200 steps
x = np.zeros(shape=(tk.size, 2, 1))
u = np.zeros(shape=(tk.size-1)) # store actually implemented control inputs
x[0,:,:] = np.array([[0.25], [0.25]])  # initial condition
for k in range(0,tk.shape[0]-1):
    # only the first u is actually implemented
    xk = x[k,:,:] - xs # current x for the linearized model
    uk = mpc(Ad,Bd,Q,R,xk,N,umax-us,umin-us) # calls MPC function to calcula
    u[k] = uk + us[0] # uk is for the linearized system; convert it back to

    # use Euler's method to integrate the nonlinear system for one sampling
    xck = x[k,:,:]
    for i in range(0, int(T/dt)):
        xck1 = xck + ode(xck,u[k])*dt
        xck = xck1
    x[k+1,:,:] = xck1
```

In [ ]:

```
plt.plot(tk,x[:,0,:],'r', tk,x[:,1,:])
plt.xlabel('k')
plt.ylabel('x')
plt.legend(['Nonlinear x1', 'Nonlinear x2'])
plt.figure(figsize=(6,3))
uforplotting = np.zeros(shape=tk.size)
uforplotting[0] = u[0]
for k in range(1,uforplotting.size):
    uforplotting[k] = u[k-1]
plt.step(tk,uforplotting)
plt.xlabel('k')
plt.ylabel('u')
plt.legend(['u'])
```