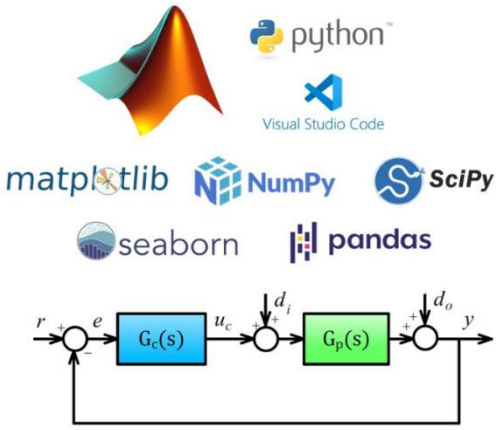


Python and MATLAB for Control Systems Simulation: A Side-by-Side Tutorial

Author: Dr Pouria Sarhadi
School of Physics, Engineering and Computer Science
University of Hertfordshire, UK

Abstract: This note presents a novel side-by-side tutorial on basic control systems simulation, analysis, and design using MATLAB and Python. While open-source software has advanced rapidly in areas such as artificial intelligence and machine learning, the adoption of Python for control system design has remained comparatively limited. This is despite the long-standing contributions of the control community to MATLAB toolboxes and workflows. Over the past decade, the emergence of the Python Control Systems Library [1,2] has provided a mature and capable alternative for control applications, warranting broader consideration. Many control engineers and students, accustomed to the intuitive simulation environment of MATLAB, may find the transition to Python challenging due to differences in syntax, libraries, and development practices. Although several resources exist, the learning process can often be fragmented and overwhelming. This tutorial addresses this gap by offering a structured, practical, and directly comparable presentation of MATLAB and Python implementations. The material covers various topics, from basic vector and matrix operations, transfer-function and state-space modelling, and classical control design, to selected topics in nonlinear and discrete-time systems, as well as data visualisation for technical reporting and research dissemination, presented in a practical format suitable for students and novice researchers. The document has been used in teaching contexts for students with basic backgrounds in control theory and programming, and has demonstrated effective learning impact through worked examples. It aims to serve as a practical reference for engineers and researchers seeking to implement control algorithms efficiently in both environments. While MATLAB remains a well-established and widely accepted tool with a proven track record in control engineering, this work aims to further facilitate the adoption of open-source Python tools. This is particularly valuable in environments where access to commercial software is limited and can help accelerate the dissemination of control methodologies within research and engineering communities that primarily operate in Python-based ecosystems.

V1.0- 2025/12/31

1- Introduction:

Nowadays, Python has established a strong presence in software and algorithm development for complex systems, particularly in communities such as artificial intelligence, computer vision, and data visualisation, which are prevalent in modern applications. While MATLAB has historically played a key role in developing engineering algorithms, and the control community has contributed significantly to MATLAB toolboxes and Simulink for system modelling and simulation, the use of Python in control system design has remained comparatively limited. Over the past decade, the efforts of Prof. Murray and collaborators have resulted in the Python Control Systems Library [1,2], which aims to provide a comparable control design experience in Python. A few articles and books have attempted to establish this connection [3–5]; however, students, particularly those with a basic background in programming, may sometimes find them not straightforward to follow. This tutorial presents a relatively straightforward and novel side-by-side approach for introducing Python and MATLAB simulations to students and novice researchers. Its novelty lies in presenting MATLAB code alongside equivalent Python scripts across various applied topics. While code translation has become increasingly feasible with modern AI tools (such as CharGPT[1]), this work provides a structured, step-by-step tutorial dedicated primarily to control engineering subject (while applicable to other topics), covering design, analysis, and simulation. The material spans fundamental vector and matrix operations, linear, nonlinear, and discrete-time control design, and practical data visualisation for technical reporting. Although not comprehensive, the tutorial enables readers to learn control system simulation systematically and apply their knowledge to more complex problems. It will be updated in the future to include additional advanced topics.

This note presents a step-by-step guide to control systems simulation and analysis using MATLAB and Python [1,2]. While some advanced topics are addressed, the focus is on providing standalone, simple examples suitable for beginners. Once a foundational level of knowledge is acquired, readers can progress to more complex problems. It is assumed that the reader has a basic understanding of control systems and programming; accordingly, the presentation is concise and omits theoretical material that can be found in conventional references [6,7]. The document is organised into several sections, each covering a specific topic. In each section, MATLAB and Python codes are presented side by side for direct comparison.

• Preparing the programming environment:

MATLAB does not require a specific setup to run functions. Hence, the codes explained within the sections can be written either in the command window or an m-file to run. It is recommended to program in an m-file since it provides debugging advantages and codes can be saved/reused. Especially, in writing functions, m-files should be employed.
Conversely, as explained in [8], running Python codes requires some libraries to be installed and included. Thus, to run the codes in Python, the following libraries should be imported into any Python code[2]:

MATLAB	Python
Adding libraries at the starting of codes	
NA	import matplotlib.pyplot as plt import control as ct import numpy as np

[1] We do not oppose the use of such tools and in fact support them for learning purposes, as they are inevitable in modern life.
[2] Reference [8] demonstrates how to create a MATLAB-like environment for Python coding in under 30 minutes.

Other examples in the Sections can be run by using the import lines above (unless explicitly stated otherwise). To install and run Python codes, Visual Studio Code IDE is suggested. A comprehensive document about setting up this environment is presented in [3].

Note1: In each table, the left-hand side includes MATLAB codes and results, and the right-hand side presents equivalent Python items.
Note2: Each topic starts with Greek numbers in green headers. Corresponding codes for that topic are written in grey background cells. Results are provided in white background cells. Codes between the green parts are assumed to be run consecutively.
Note3: Notes after per cent sign % in MATLAB and number sign # (Hash) in Python are comments and the compiler will not run them. Therefore, comments are only for extra explanations.

Important: This document is prepared for engineers and cannot be regarded as a professional programming and computer engineering guide. Therefore, one could find some terminology issues (and it is non-Pythonic!). For example, in the next Section, we call the array definition in MATLAB, as vector and matrix manipulations!

Note4: In each section, the essential and sufficient functions required to help readers begin programming are provided, along with applied examples. To gain a comprehensive understanding and effectively handle more intricate tasks, further study or instructor support may be required.

2- Basic vector and matrix manipulations

For control systems simulation, a minimal level of familiarity with arrays and matrices is necessary. In this section, some basic functions in those areas are presented.

MATLAB	Python
I. Defining vectors & matrices:	

A = [1 2 3 4 5]
%or:
A = [1, 2, 3, 4, 5]
	A = [1,2,3,4,5]
	#or:
	A = np.array([1,2,3,4,5])

A = 1 2 3 4 5	
NA	print(A)
NA	[1 2 3 4 5]
B = [1 2;3 4]	B= np.array([[1,2],[3,4]]) print(B)
B = 1 2 3 4	[[1 2] [3 4]]
C = zeros(3,2)	C=np.zeros((3,2)) print(C)
C = 0 0 0 0 0 0	[[0. 0.] [0. 0.] [0. 0.]]
D = ones(2,4)	D=np.ones((2,4)) print(D)
D = 1 1 1 1 1 1 1 1	[[1. 1. 1. 1.] [1. 1. 1. 1.]]
E = eye(3)	E=np.eye(3) print(E)
E = 1 0 0 0 1 0 0 0 1	[[1. 0. 0.] [0. 1. 0.] [0. 0. 1.]]
F = diag([-1,2,6])	F=np. diag([-1,2,6]) print(F)
F = -1 0 0 0 2 0 0 0 6	[[-1 0 0] [0 2 0] [0 0 6]]

a = -1:0.2:1 % a vector from -1 to 1 with 0.2 steps	a = np.arange(-1,1,0.2) # a vector from -1 to 1 with 0.2 steps
	print(a)

Results: a = Columns 1 through 6 -1.0000 -0.8000 -0.6000 -0.4000 -0.2000 0 Columns 7 through 11 0.2000 0.4000 0.6000 0.8000 1.0000	Results: [-1.00000000e+00 -8.00000000e-01 -6.00000000e-01 -4.00000000e-01 -2.00000000e-01 -2.22044605e-16 2.00000000e-01 4.00000000e-01 6.00000000e-01 8.00000000e-01]
b = linspace(-1,1,6) % a vector from -1 to 1 in 6 points	b = np.linspace(-1,1,6) print(b)
b = -1.0000 -0.6000 -0.2000 0.2000 0.6000 1.0000	[-1. -0.6 -0.2 0.2 0.6 1.]
II. Vector and matrix manipulations:	
% matrix multiply A = [1 2;3 4]; B = [0 1;0 1]; C = A*B	A = np.array([[1,2],[3,4]]) B = np.array([[0,1],[0,1]]) C = A@B print(C)

MATLAB	Python
C = 0 3 0 7	[[0 3] [0 7]]
% matrix element-wise multiply C = A.*B	C = A*B print(C)
C = 0 2 0 4	[[0 2] [0 4]]
% element-wise exponential A_power = A.^3	A_power = A**3 print(A_power)
A_power = 1 8 27 64	[[1 8] [27 64]]
D = A+B	D = A+B print(D)
D = 1 3 3 5	[[1 3] [3 5]]
% element-wise division E = A./B	E = A/B print(E)
E = Inf 2 Inf 4	[[inf 2.] [inf 4.]]
* Note: we should be careful in handing element-wise or normal manipulations, test the code before.	
B = [1 2;3 4]; B_transpose = B'	B = np.array([[1,2],[3,4]]) B_transponse = np.transpose(B) print(B_transponse)
B_transpose = 1 3 2 4	[[1 3] [2 4]]
B_size = size(B)	B_shape = np.shape(B) print(B_shape)
B_size = 2 2	(2, 2)
B_eigenvalues = eig(B)	B_eigenvalues = np.linalg.eig(B)
B_eigenvalues = -0.3723 5.3723	(array([-0.37228132, 5.37228132]), array([[-0.82456484, -0.41597356], [0.56576746, -0.90937671]]))
B_inverse = inv(B)	B_inverse = np.linalg.inv(B) print(B_inverse)
B_inverse = -2.0000 1.0000 1.5000 -0.5000	[[-2. 1.] [1.5 -0.5]]
A = [1 2 3 4] A_transpose = A'	A = np.array([1,2,3,4]) A_transponse = np.transpose(A) print(A_transponse)
A_transpose = 1 2 3 4	[1 2 3 4]
A_length = length(A)	A_length = len(A) print(A_length)
A_length = 4	4
III. Accessing vector and matrix elements:	
A = [4 5 6 7]; A(1)	A = np.array([4,5,6,7]) print(A[0])

MATLAB	Python
A(3)	print(A[2])
ans = 4 ans = 6	4 6
* Note that MATLAB indexing starts from 1, but Python one starts from 0.	
A(end)	print(A[-1])
7	7
A(end-1)	print(A[-2])
6	6
A(1:3)	print(A[0:3])
4 5 6	[4 5 6]
A(3:end)	print(A[2:])
6 7	[6 7]
B = [1 2 3;4 5 6;7 8 9]	B= np.array([[1,2,3],[4,5,6] ,[7,8,9]]) print(B)
B = 1 2 3 4 5 6 7 8 9	[[1 2 3] [4 5 6] [7 8 9]]
B(2,3)	print(B[1,2])
6	6
B(2:3,2:3)	print(B[1:3, 1:3])
ans = 5 6 8 9	[[5 6] [8 9]]

For getting familiar with more MATLAB-like functions in Python (*NumPy*), one can refer to [9].

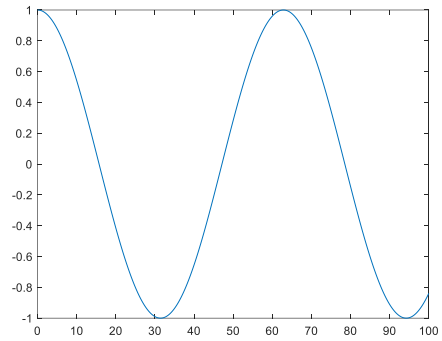
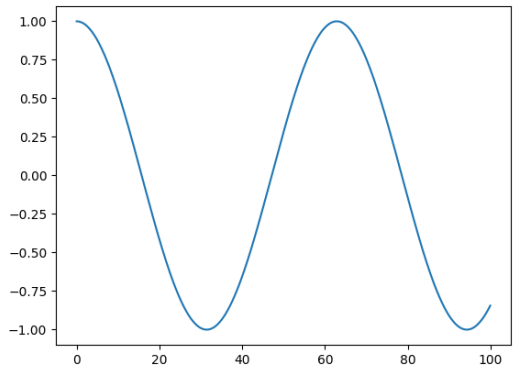
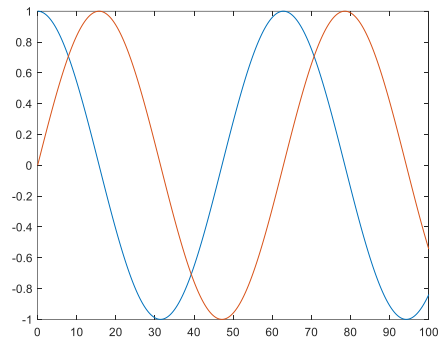
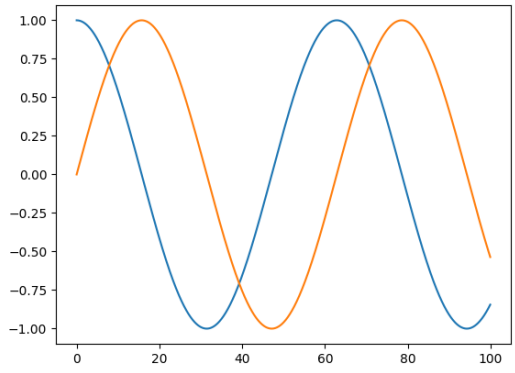
3- Basic plotting functions and visualisation

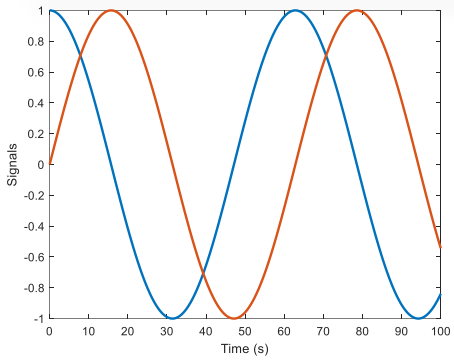
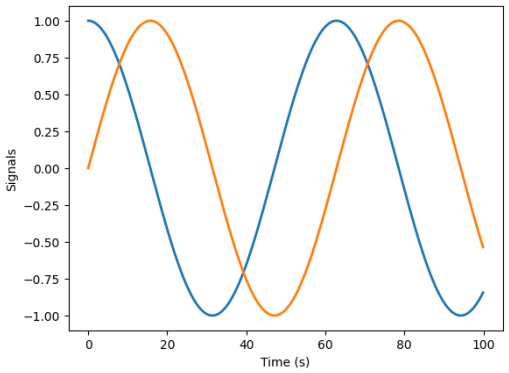
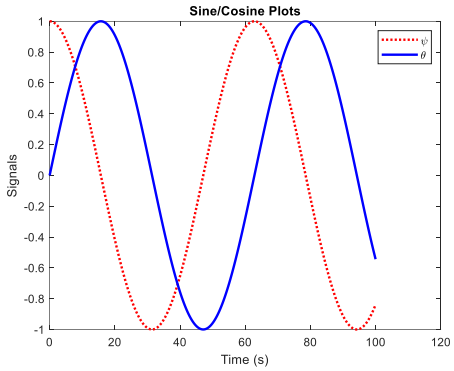
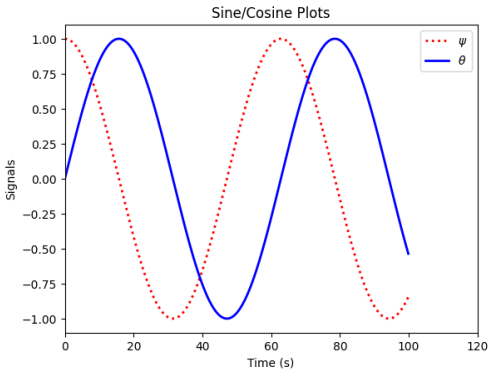
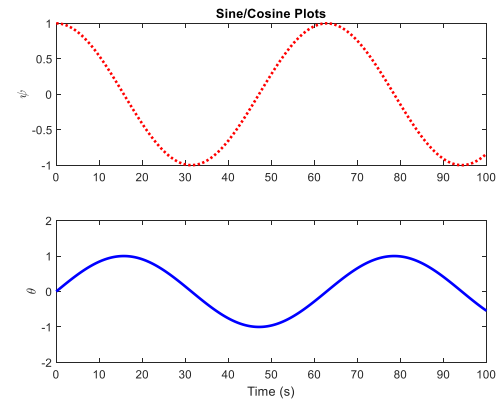
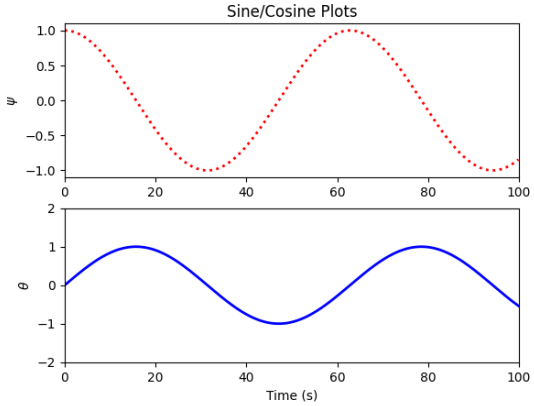
Plotting is an important tool to visualise and demonstrate engineering work. The following codes help us to prepare creative plots and figures in reports. It is evident that the Python code needs to utilise the *Matplotlib* package to generate figures. We generate sine and cosine of time and plot them as an example:

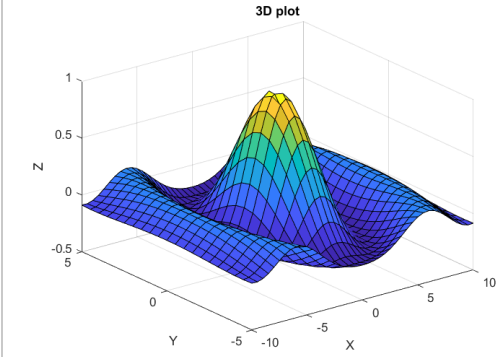
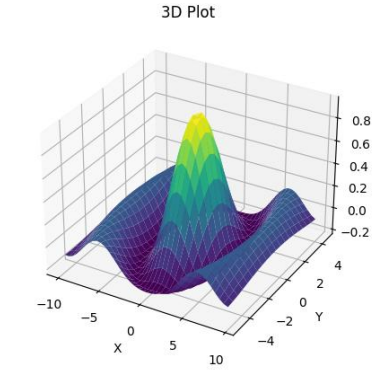
MATLAB	Python
Defining signals:	

t = 0:0.1:100;
x = cos(0.1*t);
y = sin(0.1*t);
	t = np.arange(0,100,0.1)
	x = np.cos(0.1*t)
	y = np.sin(0.1*t)

After the above definitions, we can plot our variables.

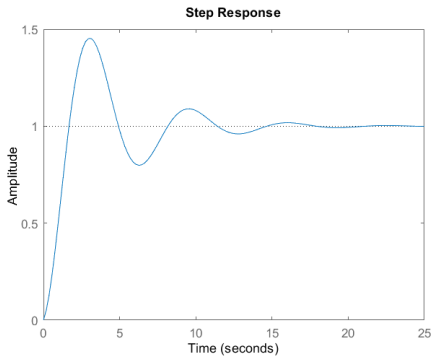
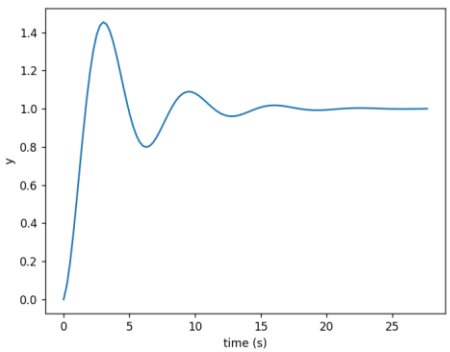
MATLAB	Python
I. Plot signals:	
plot(t,x)	plt.plot(t,x) plt.show()
	
plot(t,x,t,y)	plt.plot(t,x,t,y) plt.show()
	
plot(t,x,t,y,'linewidth',2) xlabel('Time (s)') ylabel('Signals')	plt.plot(t,x,t,y,linewidth=2) plt.xlabel('Time (s)') plt.ylabel('Signals') plt.show()

MATLAB	Python
	
plot(t,x,':r','linewidth',2), hold on plot(t,y,'b',linewidth=2) legend('\psi','\theta') title("Sine/Cosine Plots") xlabel('Time (s)') ylabel('Signals') xlim([0,120])	plt.plot(t,x,':r', label=r'ψ',linewidth=2) plt.plot(t,y,'b', label=r'θ',linewidth=2) plt.legend() plt.title('Sine/Cosine Plots') plt.xlabel('Time (s)') plt.ylabel('Signals') plt.xlim([0,120]) plt.show()
	
subplot(211) plot(t,x,':r','linewidth',2), title("Sine/Cosine Plots") ylabel('\psi') xlim([0,100]) subplot(212) plot(t,y,'b',linewidth=2) ylabel('\theta') xlabel('Time (s)') xlim([0,100]) ylim([-2 2])	plt.subplot(2,1,1) plt.plot(t,x,':r',linewidth=2) plt.title('Sine/Cosine Plots') plt.ylabel('ψ') plt.xlim([0,100]) plt.subplot(2,1,2) plt.plot(t,y,'b',linewidth=2) plt.ylabel('$\\theta$') plt.xlabel('Time (s)') plt.xlim([0,100]) plt.ylim([-2,2]) plt.show()
	

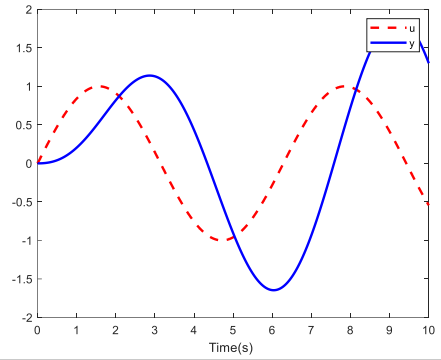
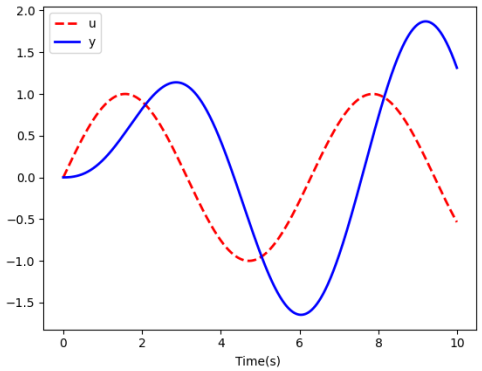
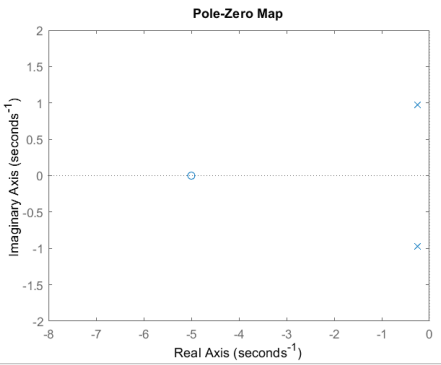
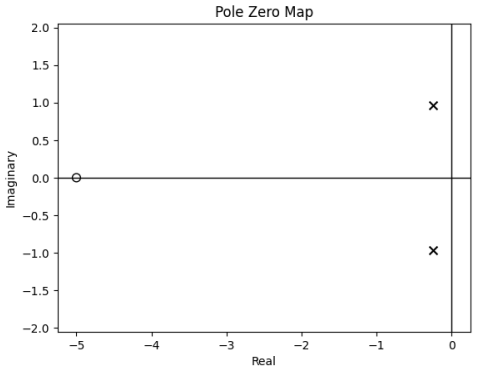
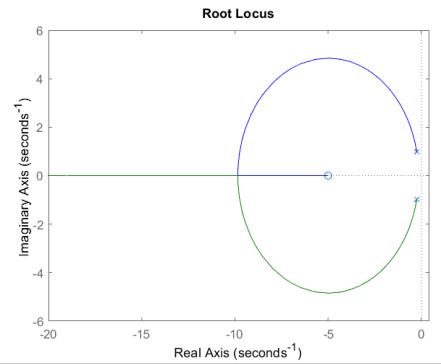
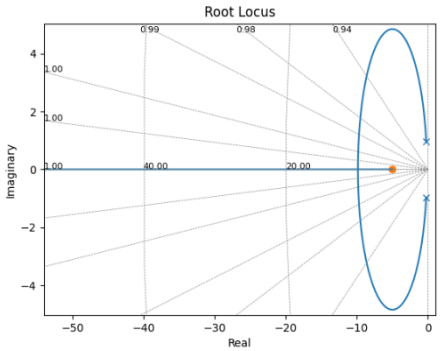
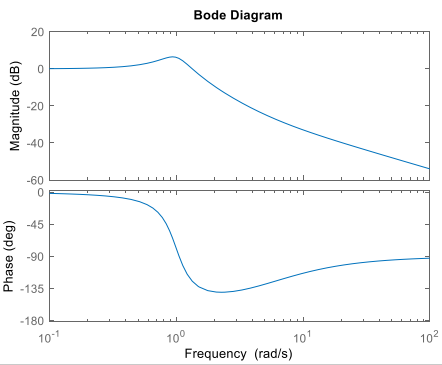
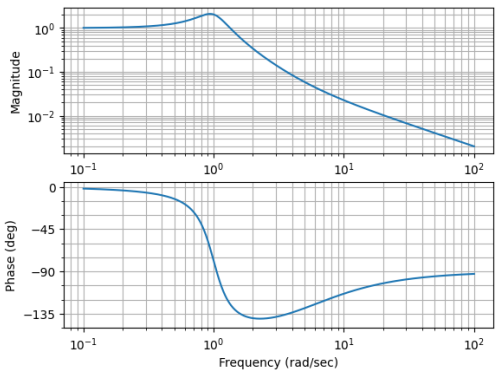
MATLAB	Python
II. 3D plots:	
X = -10:0.5:10; Y = -5:0.5:5; [X,Y] = meshgrid(X,Y); % converts a 2D vector to 3D R = sqrt(X.^2 + Y.^2) ; Z = sin(R)./R; surf(X,Y,Z) % surface plot title('3D plot') xlabel('X') ylabel('Y') zlabel('Z')	X = np.arange(-10, 10, 0.5) Y = np.arange(-5, 5, 0.5) X, Y = np.meshgrid(X, Y) R = np.sqrt(X**2 + Y**2) Z = np.sin(R)/R plt.close('all') ax = plt.axes(projection ='3d') ax.plot_surface(X, Y, Z, cmap='viridis') ax.set_title('3D Plot') ax.set_xlabel('X') ax.set_ylabel('Y') ax.set_zlabel('Z') plt.show()
	
* Note x and y axes difference in MATLAB and Python	

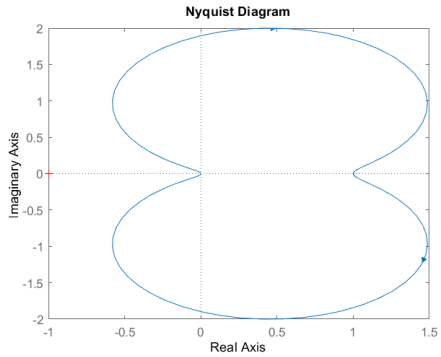
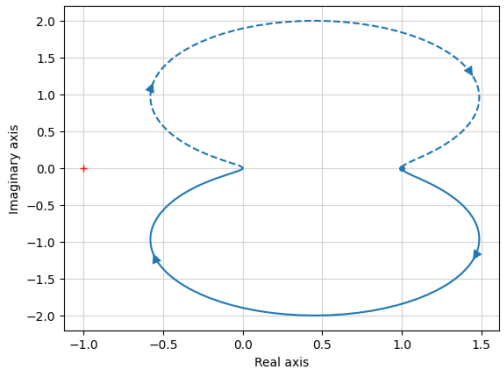
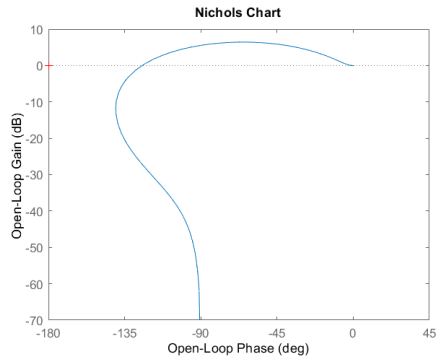
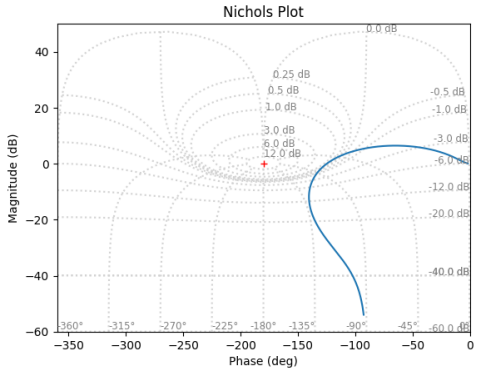
4- Classic linear control system analysis (transfer function)

Python Control [1, 2] provides a handy tool to analyse and design control systems, with functions that are very similar to MATLAB. In this section, the most useful functions are reviewed.

MATLAB	Python
I. Defining a system and finding its poles and zeros:	
sys = tf([0.2 1],[1 0.5 1])	sys = ct.tf([0.2,1],[1,0.5,1]) print(sys)
sys = 0.2 s + 1 --------------- s^2 + 0.5 s + 1 Continuous-time transfer function.	0.2 s + 1 --------------- s^2 + 0.5 s + 1
* or alternatively:	
s = tf('s') sys = (0.2*s+1)/(s^2+0.5*s+1)	s = ct.tf('s') sys = (0.2*s+1)/(s**2+0.5*s+1) print(sys)
sys = 0.2 s + 1 --------------- s^2 + 0.5 s + 1 Continuous-time transfer function.	0.2 s + 1 --------------- s^2 + 0.5 s + 1
P=pole(sys)	P = ct.pole(sys)
P = -0.2500 + 0.9682i -0.2500 - 0.9682i	[-0.25+0.96824584j -0.25-0.96824584j]
Z = zero(sys)	Z = ct.zeros(sys)
P = -5	[-5.+0.j]
II. System simulation and visualisation (time and frequency domain plots)	
sys = tf([0.2 1],[1 0.5 1]); step(sys)	time,y = ct.step_response(sys) plt.figure(1) plt.plot(time,y) plt.xlabel("time (s)") plt.ylabel("y") plt.show()
	

% response to arbitrary inputs like sinus
t = 0:0.01:10;
u = sin(t);
sys = tf([0.2 1],[1 0.5 1]);
y = lsim(sys,u,t);
plot(t,u,'r--',t,y,'b','linewidth',2)
xlabel('Time(s) ')
legend('u','y')
	t = np.arange(0,10,0.01)
	u = np.sin(t)
	sys = ct.tf([0.2,1],[1,0.5,1])
	(t,y) = ct.forced_response(sys,t,u)
	
	plt.figure(1)
	plt.plot(t,u,'r--',label='u',linewidth=2)
	plt.plot(t,y,'b',label='y',linewidth=2)
	
	plt.xlabel('Time(s) ')

MATLAB	Python
	plt.legend() plt.show()
	
figure, pzmap(sys)	ct.pzmap(sys) plt.show()
	
figure, rlocus(sys)	ct.rlocus(sys) plt.show()
	
figure, bode(sys)	
	
* note the difference in x axis scale of both plots	
figure, nyquist(sys)	

MATLAB	Python
	
figure, nichols(sys)	
	

III. System manipulation

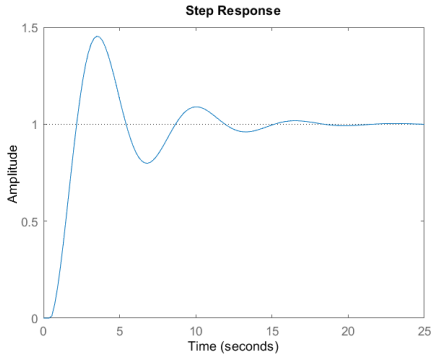
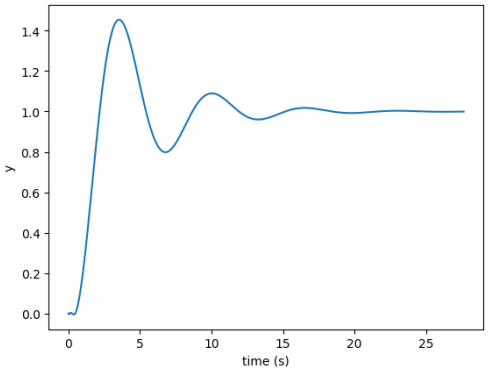
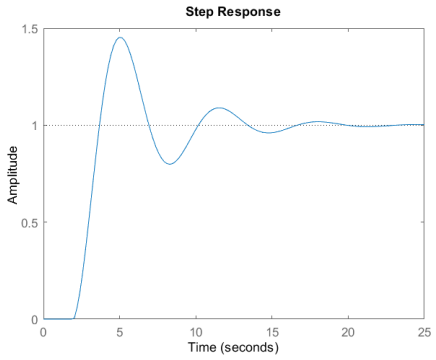
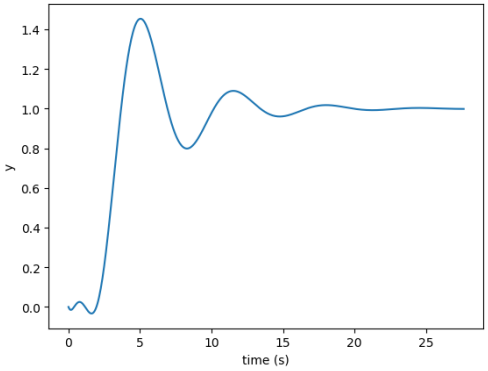
* Both MATLAB and Python Control allow for arithmetic manipulations of systems; however, there are built-in functions, for example, to create feedback or series of two systems.

s = tf('s'); Gc=1/(s+1); Gp = (0.2*s+1)/(s^2+0.5*s+1); Gol = Gc * Gp	s = ct.tf('s') Gc=1/(s+1) Gp = (0.2*s+1)/(s**2+0.5*s+1) Gol = Gc + Gp print(Gol)
Gol = 0.2 s + 1 ---------- s^3 + 1.5 s^2 + 1.5 s + 1	0.2 s + 1 ---------- s^3 + 1.5 s^2 + 1.5 s + 1
Gol2 = series(Gc,Gp)	Gol2 = ct.series(Gc,Gp) print(Gol2)
0.2 s + 1 ---------- s^3 + 1.5 s^2 + 1.5 s + 1	0.2 s + 1 ---------- s^3 + 1.5 s^2 + 1.5 s + 1
Gcl = feedback(Gol,1)	Gcl = ct.feedback(Gol,1) print(Gcl)
0.2 s + 1 ---------- s^3 + 1.5 s^2 + 1.7 s + 2	0.2 s + 1 ---------- s^3 + 1.5 s^2 + 1.7 s + 2
Gparallel = parallel(Gc,Gp)	Gparallel = ct.parallel(Gc,Gp) print(Gparallel)
1.2 s^2 + 1.7 s + 2 ---------- s^3 + 1.5 s^2 + 1.5 s + 1	1.2 s^2 + 1.7 s + 2 ---------- s^3 + 1.5 s^2 + 1.5 s + 1

IV. Delay modelling

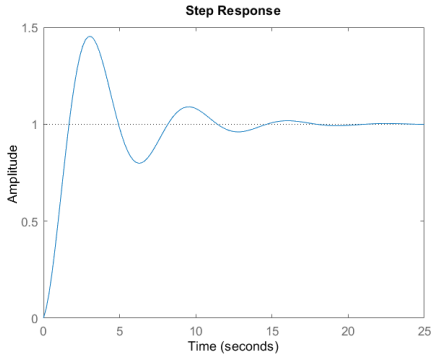
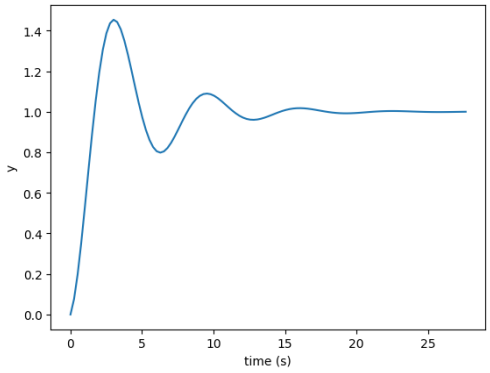
* Currently, Python Control does not support Delay. To model delay, Pade approximation is provided. To increase its precision, its order should be increased; nevertheless, it is an approximation!

s=tf('s'); sys=exp(-0.5*s)*(0.2*s+1)/(s^2+0.5*s+1) step(sys)	s = ct.tf('s') (num,den) = ct.pade(0.5,3) #0.5 delay, 3 Pade order sys = ct.tf(num,den)*(0.2*s+1)/(s**2+0.5*s+1)

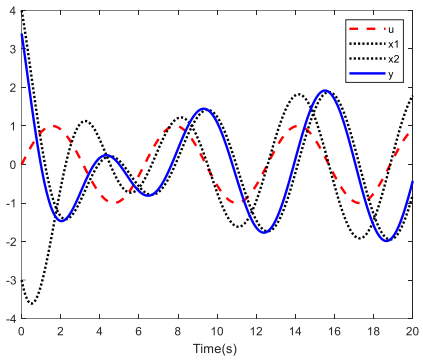
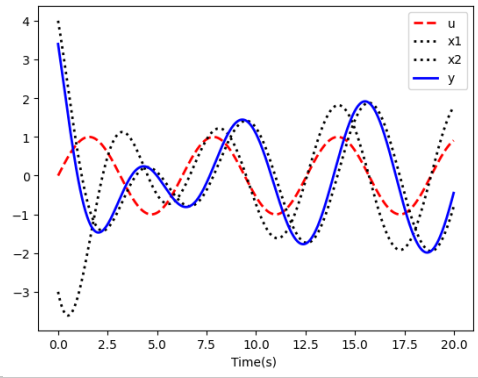
MATLAB	Python
	time,y = ct.step_response(sys) plt.figure(1) plt.plot(time,y) plt.xlabel("time (s)") plt.ylabel("y") plt.show()
sys = 0.2 s + 1 exp(-0.5*s) * ---------- s^2 + 0.5 s + 1 	-0.2 s^4 + 3.8 s^3 - 24 s^2 - 48 s + 960 ---------- s^5 + 24.5 s^4 + 253 s^3 + 1104 s^2 + 720 s + 960 
Only change the delay to 2s in the previous code	Only change the delay to 2s in the previous code
	

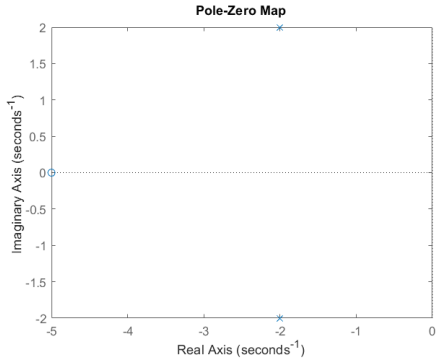
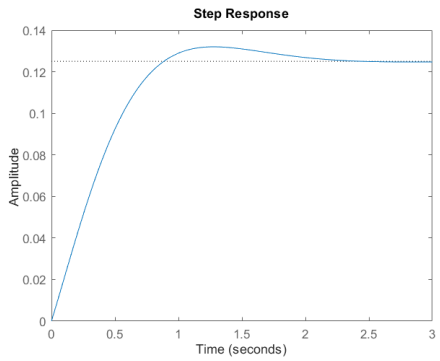
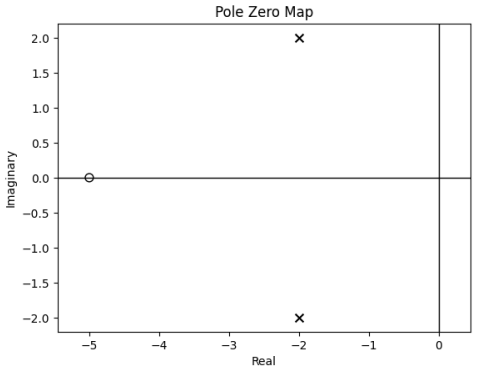
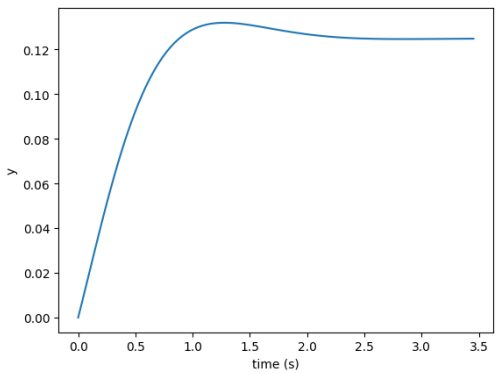
5- State space control system design and analysis

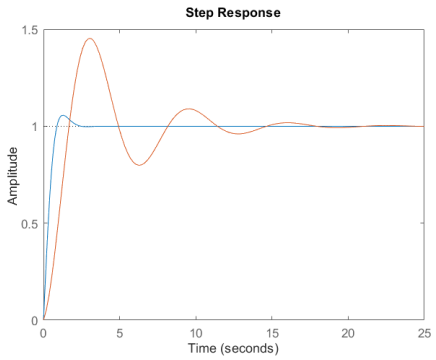
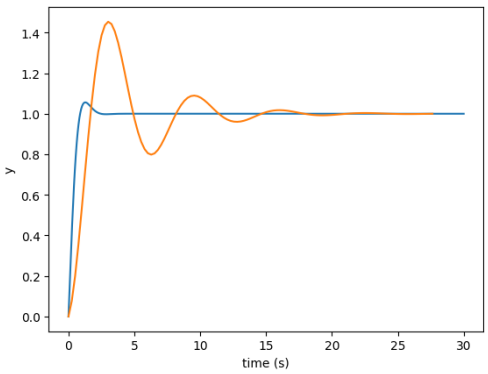
This section presents basic functions for handling state-space simulations in MATLAB and Python.

MATLAB	Python
I. Defining and manipulating a system in state-space form:	
Ap = [0 1;-1 -0.5]; Bp = [0;1]; Cp = [1 0.2]; Dp = 0; sys = ss(Ap,Bp,Cp,Dp)	Ap = np.array([[0,1],[-1,-0.5]]) Bp = np.array([[0],[1]]) Cp = np.array([[1,0.2]]) Dp = 0 sys = ct.ss(Ap,Bp,Cp,Dp)
A = x1 x2 x1 0 1 x2 -1 -0.5 B = u1 x1 0 x2 1 C = x1 x2 y1 1 0.2 D = u1 y1 0 Continuous-time state-space model.	<LinearIOSystem>: sys[2] Inputs (1): ['u[0]'] Outputs (1): ['y[0]'] States (2): ['x[0]', 'x[1]'] A = [[0. 1.] [-1. -0.5]] B = [[0.] [1.]] C = [[1. 0.2]] D = [[0.]]
step(sys)	plt.figure(1) plt.plot(time,y) plt.xlabel("time (s)") plt.ylabel("y") plt.show() print(sys)
	

% response to arbitrary inputs and initial conditions
t = 0:0.01:20;
u = sin(t);
x0 = [4 -3];
[y,tout,x] = lsim(sys,u,t,x0);
plot(t,u,'r--',t,x,'k:',t,y,'b','linewidth',2)
xlabel('Time(s) ')
legend('u','x1','x2','y')
	t = np.arange(0,20,0.01)
	u = np.sin(t)
	x0 = np.array([[4],[-3]])
	(t,y,x) = ct.forced_response(sys,t,u,X0=x0,return_x=True)
	
	plt.figure(1)
	plt.plot(t,u,'r--',label='u',linewidth=2)
	plt.plot(t,x[0,:],'k:',label='x1',linewidth=2)
	plt.plot(t,x[1,:],'k:',label='x2',linewidth=2)
	plt.plot(t,y,'b',label='y',linewidth=2)
	
	plt.legend()
	plt.xlabel('Time(s) ')
	plt.show()

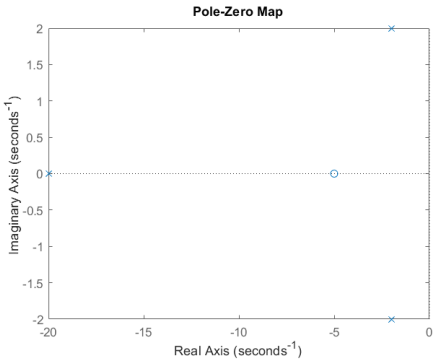
MATLAB	Python
	
* almost all other tools like step, bode, nyquist, etc. are available for the state space form like TF.	
sys2 = tf(sys)	sys2 = ct.tf(sys)
0.2 s + 1 --------------- s^2 + 0.5 s + 1 Continuous-time transfer function.	0.2 s + 1 --------------- s^2 + 0.5 s + 1
Eig_A = eig(Ap)	Eig_A = np.linalg.eig(Ap)
Eig_A = -0.2500 + 0.9682i -0.2500 - 0.9682i	(array([-0.25+0.96824584j, -0.25-0.96824584j]), array([[0.70710678+0.j , 0.70710678-0.j], [-0.1767767 +0.6846532j, -0.1767767 -0.6846532j]]))
* controllability and observability check of the system	
Ct = ctrb(Ap,Bp)	Ct = ct.ctrb(Ap,Bp) print(Ct)
ans = 0 1.0000 1.0000 -0.5000	[[0. 1.] [1. -0.5]]
rank(Ct)	print(np.linalg.matrix_rank(Ct))
ans = 2	2
Ob = obsv(A,C) rank(Ob)	Ob = ct.obsv(A,C) print(np.linalg.matrix_rank(Ob))
ans = 2	2
II. Feedback controller design and tracking using system's inverse DC gain	
Ap = [0 1;-1 -0.5]; Bp = [0;1]; Cp = [1 0.2]; Dp = 0; sys = ss(Ap,Bp,Cp,Dp); P = [-2+2*i,-2-2*i]; % new pole places selected by designer K = place(Ap,Bp,P) %Gains to place the poles sys_new = ss(Ap-Bp*K,Bp,Cp,Dp) %forming the new system figure,pzmap(sys_new) figure,step(sys_new)	Ap = np.array([[0,1],[-1,-0.5]]) Bp = np.array([[0],[1]]) Cp = np.array([[1,0.2]]) Dp = 0 sys = ct.ss(Ap,Bp,Cp,Dp) P = np.array([[-2+2j,-2-2j]]) K = ct.place(Ap,Bp,P) print(K) sys_new = ct.ss(Ap-Bp*K,Bp,Cp,Dp) print(sys_new) time,y = ct.step_response(sys) Kdc = 1/ct.dcgain(sys_new) sys_new_2 = Kdc*ct.ss(Ap-Bp*K,Bp,Cp,Dp) time2,y2 = ct.step_response(sys_new_2,30) plt.figure(1)

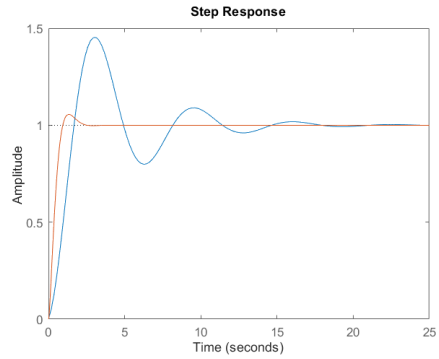
MATLAB	Python
	ct.pzmap(sys_new) plt.show() plt.figure(2) plt.plot(time2,y2,time,y) plt.xlabel("time (s)") plt.ylabel("y") plt.show() print(sys)
K = 7.0000 3.5000 sys_new = A = x1 x2 x1 0 1 x2 -8 -4 B = u1 x1 0 x2 1 C = x1 x2 y1 1 0.2 D = u1 y1 0  	[[7. 3.5]] <LinearIOSystem>: sys[3] Inputs (1): ['u[0]'] Outputs (1): ['y[0]'] States (2): ['x[0]', 'x[1]'] A = [[0. 1.] [-8. -4.]] B = [[0.] [1.]] C = [[1. 0.2]] D = [[0.]]  
Kdc = 1/dcgain(sys_new) %compensation using system inverse gain sys_new_2 = Kdc*ss(Ap-Bp*K,Bp,Cp,Dp) figure,step(sys_new_2,sys)	Kdc = 1/ct.dcgain(sys_new) sys_new_2 = Kdc*ct.ss(Ap-Bp*K,Bp,Cp,Dp) time,y = ct.step_response(sys) time2,y2 = ct.step_response(sys_new_2,30) plt.figure(2) plt.plot(time2,y2,time,y) plt.xlabel("time (s)")

MATLAB	Python
	plt.ylabel("y") plt.show() print(sys)
	
III. Feedback controller design for tracking with augmented integral state	
Ap = [0 1;-1 -0.5]; Bp = [0;1]; Cp = [1 0.2]; Dp = 0; sys = ss(Ap,Bp,Cp,Dp) A = [0 1 0; zeros(2,1) Ap]; B = [0;Bp]; C = [0 Cp]; D = []; P = [-2+2*i,-2-2*i,-20]; K = place(A,B,P) Br = -[1;0;0]; sys_new = ss(A-B*K,Br,C,D) figure,pzmap(sys_new) figure,step(sys,sys_new)	Ap = np.array([[0,1],[-1,-0.5]]) Bp = np.array([[0],[1]]) Cp = np.array([[1,0.2]]) Dp = 0 sys = ct.ss(Ap,Bp,Cp,Dp) P = np.array([[-2+2j,-2-2j,-20]]) A = np.array([[0,1,0], [0,0,1], [0,-1,-0.5]]) B = np.array([[0],[0],[1]]) C = np.array([[0,1,0.2]]) D = 0 K = ct.place(A,B,P) print(K) Br = -np.array([[1],[0],[0]]) sys_new = ct.ss(A-B*K,Br,C,D) print(sys_new) time,y = ct.step_response(sys) time2,y2 = ct.step_response(sys_new,30) plt.figure(1) ct.pzmap(sys_new) plt.show() plt.figure(2) plt.plot(time,y,time2,y2) plt.xlabel("time (s)") plt.ylabel("y") plt.show() print(sys)
K = 160.0000 87.0000 23.5000 sys_new = A = x1 x2 x3 x1 0 1 0	[[160. 87. 23.5]] A = [[0. 1. 0.] [0. 0. 1.] [-160. -88. -24.]] B = [[-1.] [0.]

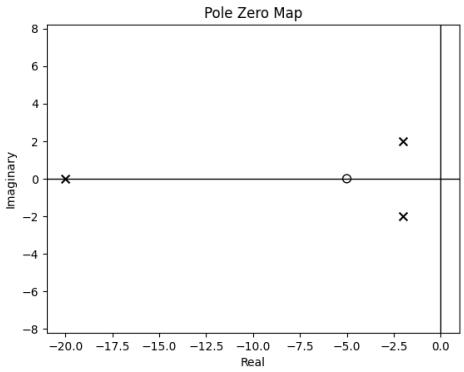
MATLAB	Python

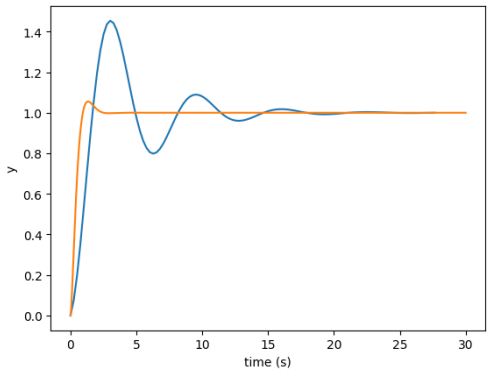
```
 x2    0    0    1
 x3 -160  -88  -24
B =
    u1
 x1  -1
 x2   0
 x3   0
C =
     x1  x2  x3
 y1   0   1 0.2
D =
    u1
 y1  0
```






```
    [ 0.]]
C = [[0.  1.  0.2]]
D = [[0.]]
```

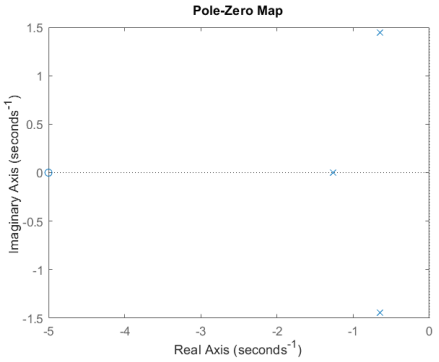
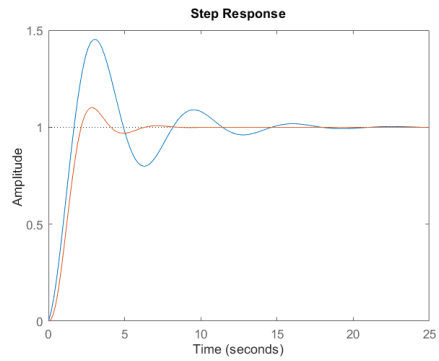
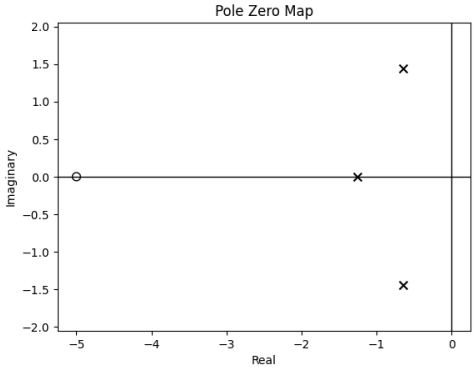
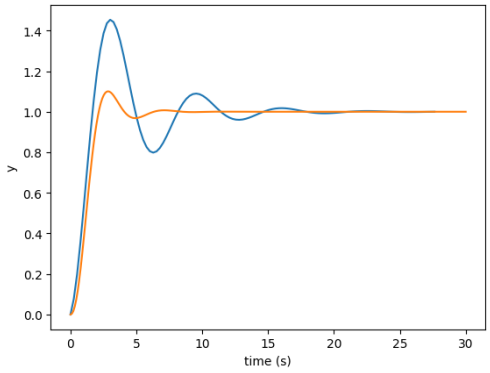




IV. Linear Quadratic Controller with Augmented Error Integral State Feedback (LQISF)

```
Ap = [0 1;-1 -0.5];
Bp = [0;1];
Cp = [1 0.2];
Dp = 0;

sys = ss(Ap,Bp,Cp,Dp)

A = [0 1 0;
  zeros(2,1) Ap];
B = [0;Bp];
C = [0 Cp];
D = [];

% LQR Settings
QLQR=[10 0 0;0 0 0;0 0 0];
RLQR=1;

KLQR = lqr(A,B,QLQR,RLQR)

Br = -[1;0;0];
```

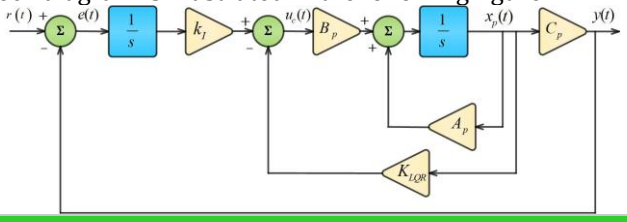
```
Ap = np.array([[0,1],[-1,-0.5]])
Bp = np.array([[0],[1]])
Cp = np.array([[1,0.2]])
Dp = 0

sys = ct.ss(Ap,Bp,Cp,Dp)

P = np.array([[-2+2j,-2-2j,-20]])

A = np.array([[0,1,0],
  [0,0,1],
  [0,-1,-0.5]])
B = np.array([[0],[0],[1]])
C = np.array([[0,1,0.2]])
D = 0

# LQR Settings
QLQR = np.array([[10,0,0],
          [0,0,0],
          [0,0,0]])
RLQR=1
```

MATLAB	Python
sys_new = ss(A-B*KLQR,Br,C,D) figure,step(sys,sys_new)	KLQR = ct.lqr(A,B,QLQR,RLQR) print(KLQR[0]) Br = -np.array([[1],[0],[0]]) sys_new = ct.ss(A-B*KLQR[0],Br,C,D) print(sys_new) time,y = ct.step_response(sys) time2,y2 = ct.step_response(sys_new,30) plt.figure(2) plt.plot(time,y,time2,y2) plt.xlabel("time (s)") plt.ylabel("y") plt.show() print(sys)
 	 
KLQR = 3.1623 3.1438 2.0569 sys_new = A = x1 x2 x3 x1 0 1 0 x2 0 0 1 x3 -3.162 -4.144 -2.557 B = u1 x1 -1 x2 0 x3 0 C = x1 x2 x3 y1 0 1 0.2 D = u1	[[3.16227766 3.14380118 2.05687355]] A = [[0. 1. 0.] [0. 0. 1.] [-3.16227766 -4.14380118 -2.55687355]] B = [[-1.] [0.] [0.]] C = [[0. 1. 0.2]] D = [[0.]]

MATLAB	Python
y1 0	

Note: Compare gains (K) for pole placement (PP) and LQR-based controllers. The PP method does not consider the control signal, thus generating large gains to relocate the poles. In contrast, the LQR-based controller, because of its consideration of the control effort weight, generates a smoother control effort and assigns the poles to closer positions.

Note: The augmented integral controllers in 7.III and 7.IV are employed to design the controller. In practice, they are implemented as shown in the following form; therefore, the integral part should be separated and embedded in the controller, not in the plant. Its block diagram is illustrated in the following figure.



V. State observer design

TBC

6- Control flow statements (loops and conditional statements)

Control flow statements, including loops and conditional statements, govern the execution order of a program. Conditional statements, like "if," allow the program to make decisions based on specified conditions, enabling different code paths. Loops, such as "for" and "while," facilitate repetitive execution of code, iterating over a sequence of elements or until a specified condition is met. These control flow structures enhance the flexibility and efficiency of programming by enabling dynamic and context-dependent behavior in code execution.

It is noteworthy that, in using control flow statements, we often need to employ comparison operators. These operators are crucial for evaluating conditions in both MATLAB and Python, helping to establish relationships between values. Comparison Operators include the following items:

<	Less than	>	Greater than
<=	Less than or equal to	>=	Greater than or equal to
==	Equal to	~=	Not equal to (in Python "!=")

Moreover, logical operators are utilised to combine or negate conditions. For instance, in MATLAB, the 'AND' operator is represented by "&", and an "if" statement might be written as if a > 0 & a < 1 to execute a task if the variable "a" is between 0 and 1. Logical operators are also summarised below:

Operator	MATLAB	Python	Operator	MATLAB	Python
AND	&	and	**Or**	|	or
NOT	~	not	**Exclusive OR (XOR)**	xor	^

Some examples of control flow statements are provided below:

MATLAB	Python
I. A simple "for" loop to print numbers 1 to 3:	

for i=1:3
  fprintf('%i\n',i)
end
	for i in range(1, 4):
	  print(i)

1 2 3	1 2 3
II. Using while	

i = 1;
 	
while i<4
  fprintf('%i\n',i)
  i = i+1;
end
	i = 1
	while i<4:
	  print(i)
	  i += 1 # or i = i+1

1 2 3	1 2 3
III. If else example to check a conditional statement	

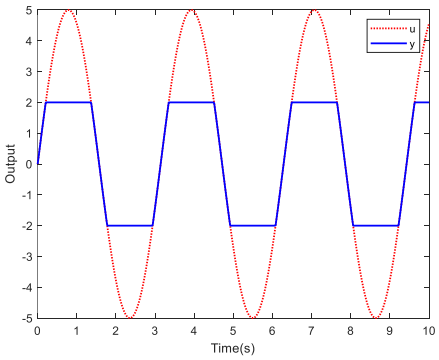
a=4
if a==4
  fprintf('a = 4\n')
elseif a==5
  fprintf('a = 5\n')
end
	a = 4
	if a == 4:
	  print('a = 4')
	elif a == 5:
	  print('a = 5')

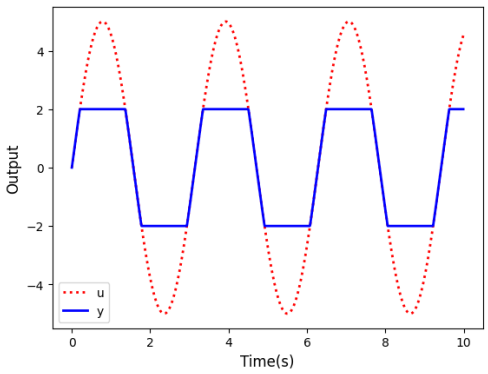
a = 4	a = 4
IV. Simulating a saturation function using a for loop and if	

clear all, close all, clc
 	
time = 0:0.01:10; %defining time
 	
Sat = 2; %Defining Saturation Level
 	
u = 5*sin(2*time); % generating the input
	time = np.arange(0,10,0.01) # Defining time
	
	Sat = 2 # Defining Saturation Level
	
	# generating the input
	u = 5*np.sin(2*time)

```
for t=1:length(time)  % lopp to simulate the time

   if u(t)>Sat % if then rules to mimic saturation
      y(t)=Sat;
   elseif u(t)<-Sat
      y(t)=-Sat;
   else
      y(t)=u(t);
   end
end

% plot setpoint and output
figure,
plot(time,u,'r:',time,y,'b','linewidth',1.5)
xlabel('Time(s)'), ylabel('Output')
legend('u','y')
```

```
# initial conditions
y = np.zeros(len(time))

for t in range(1,len(time)):  # loop to simulate the time

   if u[t]>Sat: # if then rules to mimic saturation
      y[t] = Sat
   elif u[t]<-Sat:
      y[t]=-Sat
   else:
      y[t]=u[t]

# plot input and output
plt.close("all")
plt.figure()
plt.plot(time, u,':r', label='u', linewidth=2)
plt.plot(time, y,'b', label='y', linewidth=2)
plt.xlabel('Time(s)', fontsize=12)
plt.ylabel('Output', fontsize=12)
plt.legend(loc='best')
plt.show()
```

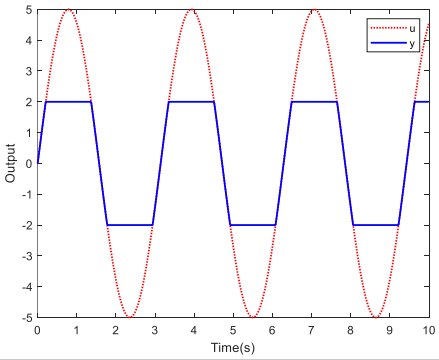
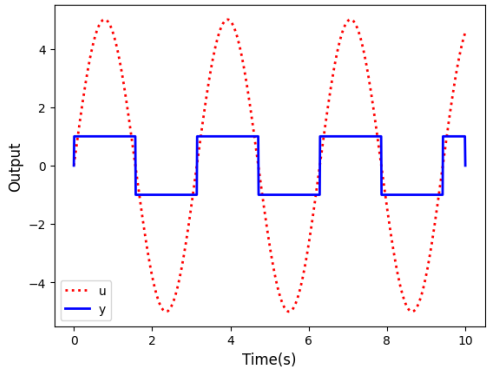




V. Simulating a relay function using a while loop and if

```
dt = 0.01; % simulation sampling time
Tf = 10; % final time
time = 0:dt:Tf; % Defining time

Rel = 2; % Defining Relay Level

u = 5*sin(2*time); % generating the input

t = 1;

while Tf>time(t)

   if u(t)>Rel % if then rules to mimic saturation
      y(t)=Rel;
   elseif u(t)<-Rel
      y(t)=-Rel;
   else
      y(t)=u(t);
   end

   t = t+1;
end

% plot input and output
figure,
plot(time,u,'r:',time(1:end-1),y,'b','linewidth',1.5)
```

```
dt = 0.01 # sampling time [s]
Tf = 10.0 # simulation time [s]

time = np.arange(0,Tf+dt,dt) # Defining time

Rel = 1 # Defining Relay Level

# generating the input
u = 5*np.sin(2*time)

# initial conditions
y = np.zeros(len(time))
t = 0

while Tf >=time[t]+dt: # loop to simulate the time

   if u[t]>0: # if then rules to mimic saturation
      y[t] = Rel
   elif u[t]<-0:
      y[t] = -Rel
   else:
      y[t]=u[t]

   t += 1       # adding sample

# plot input and output
```

xlabel('Time(s)'), ylabel('Output') legend('u','y')	plt.close("all") plt.figure() plt.plot(time, u,':r', label='u', linewidth=2) plt.plot(time, y,'b', label='y', linewidth=2) plt.xlabel('Time(s)', fontsize=12) plt.ylabel('Output', fontsize=12) plt.legend(loc='best') plt.show()
	

7- Defining functions

A function is a reusable block of code designed to perform a specific task when called, enhancing code organisation and structure. They are like black-boxes that receive some inputs and provide outputs without access to their internal mechanisms(unless defined). Functions render several advantages including but not limited to:

- Reusability: Functions can be called multiple times, reducing redundancy and promoting efficient code reuse.
- Modularity: Functions break down code into smaller, manageable units, promoting modular development.
- Abstraction: Functions hide internal implementation details, allowing users to interact at a higher level.
- Readability: Well-named functions improve code clarity, making it easier to understand and maintain.
- Scoping: Functions provide a controlled scope for variables, preventing naming conflicts and enhancing code reliability.

We frequently employ functions in control system design and simulation. In fact, functions such as "plot()" and "step()" that we used previously are pre-programmed to conduct tasks like plotting. In this section, we learn how to program and use customised functions. We need functions in control system implementation and simulations.

MATLAB	Python
VI. Defining and calling a function:	

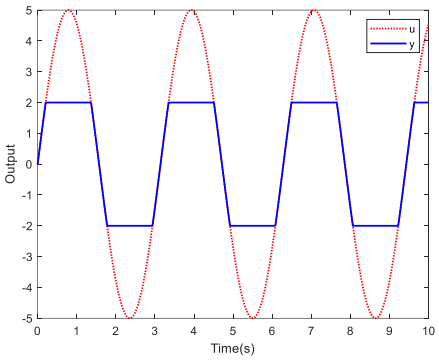
x = 3;
y = 5;
sum_result = add_numbers(x, y);
 	
disp(['The sum is: ' num2str(sum_result)]);
 	
function result = add_numbers(a, b)
  result = a + b;
end
	def main():
	  x = 3
	  y = 5
	  sum_result = add_numbers(x, y)
	  print(f'The sum is: {sum_result}')
	
	def add_numbers(a, b):
	  result = a + b
	  return result
	
	if __name__ == '__main__':
	  main()

The sum is: 8	The sum is: 8
* Python functions can be called without a main() function, however, functions should be defined before calling them. * Python functions need “return” to return a value.	
VII. Simulating Saturation by a function	

clear all, close all, clc,
 	
time = 0:0.01:10; % defining time
Sat = 2; % Defining Saturation Level
u = 5 * sin(2 * time); % Generating the input
 	
% Initialize y
y = zeros(size(u));
 	
for t = 1:length(time) % loop to simulate the time
  % Call the function for saturation
  y(t) = saturation(u(t), Sat);
end
 	
% Plot setpoint and output
figure;
	def main():
	  # Defining time
	  time = np.arange(0, 10, 0.01)
	
	  # Defining Saturation Level
	  Sat = 2
	
	  # Generating the input
	  u = 5 * np.sin(2 * time)
	
	  # Initializing output
	  y = np.zeros(len(time))
	
	  # Loop to simulate the time
	  for t in range(1, len(time)):
	  # Call the function for saturation

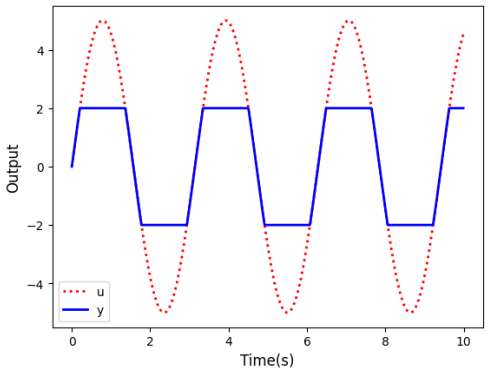
MATLAB

```
plot(time, u, 'r:', time, y, 'b', 'linewidth', 1.5);
xlabel('Time(s)'), ylabel('Output');
legend('u', 'y');

% Function definition for Saturation
function result = saturation(input, Sat)
   if input > Sat
      result = Sat;
   elseif input < -Sat
      result = -Sat;
   else
      result = input;
   end
end
```



Python

```
    y[t] = saturation(u[t], Sat)

   # Plot input and output
   plt.close("all")
   plt.figure()
   plt.plot(time, u, ':r', label='u', linewidth=2)
   plt.plot(time, y, 'b', label='y', linewidth=2)
   plt.xlabel('Time(s)', fontsize=12)
   plt.ylabel('Output', fontsize=12)
   plt.legend(loc='best')
   plt.show()

# Function definition for Saturation
def saturation(input_val, Sat):
   if input_val > Sat:
      return Sat
   elif input_val < -Sat:
      return -Sat
   else:
      return input_val

if __name__ == '__main__':
   main()
```



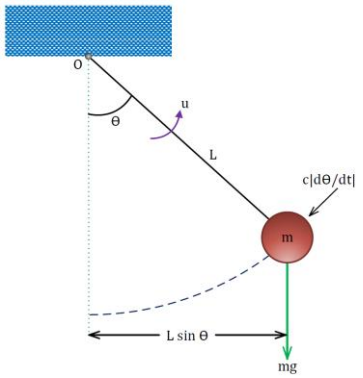
* Saturation can be implemented with better approaches, this is an example for learning.

8- Differential equation (nonlinear system) simulation

We became familiarwith simulating linear systems (transfer function) in Section 3, which is achieved using MATLAB or Python's built-in functions. However, systems are mostly nonlinear, and in our context, we need to simulate systems that are not in transfer function or linear state space forms. Now that we understand how to use functions and loops in programming, it is feasible to simulate Ordinary Differential Equations (ODEs) that may include linear and nonlinear models. Actually, in the backend of transfer function simulation, similar approaches are adopted.

To simulate a differential equation, we use integrators that are vital components of any simulation. There are various approaches to integration, ranging from first-order Euler integration to more complex ones like higher-order Runge-Kutta methods. The common approach is to consider systems' first-order differential equations and numerically integrate them. If these equations are of a higher degree, we convert them to first-order equations and integrate them. For example, consider the differential equations of a simple pendulum, which is a second order ODE:

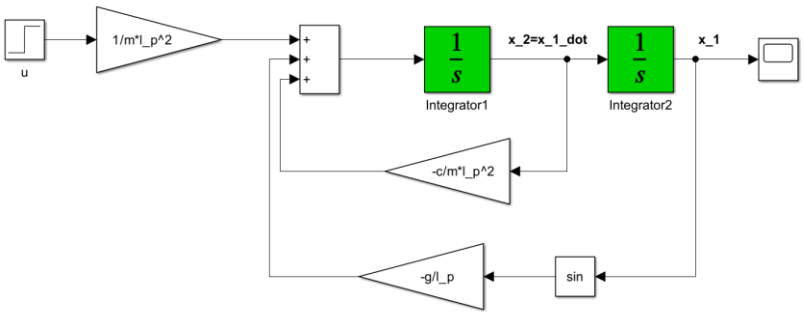
$$\ddot{\theta} = \frac{-c}{ml_p^2}\dot{\theta} - \frac{g}{l_p}\sin(\theta) + \frac{1}{ml_p^2}T$$



where θ is the angle of the pendulum with respect to the vertical axis, which can be considered as the system output. T is the input moment to control the shaft. Symbols m and l_p are the pendulum mass and length, respectively. Variable c denotes the friction coefficient at the hinge, and g symbolises the gravity constant. This model is a nonlinear representation arising from the mechanical relationships of the system. Assuming $x_1 = \theta$ and $x_1 = \dot{\theta}$ as well as $u = T$, the above equation can be rewritten in following form which is expressed by first order ODEs:

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = \dfrac{-c}{ml_p^2}x_2 - \dfrac{g}{l_p}x_1 + \dfrac{1}{ml_p^2}u \end{cases}$$

Using block diagrams or Simulink, we can implement and simulate this system (or any similar system) using integrators. For the pendulum system, the corresponding blocks could simulate its behaviour.



In a code, we need to use integrator functions available in MATLAB and Python's SciPy library. In this section, we will first simulate this system with zero input. Then, we will simulate it in a loop with online control actions. Finally, we introduce a PID controller. As an advanced topic, custom integrator functions will be developed to replace available functions in MATLAB and Python.

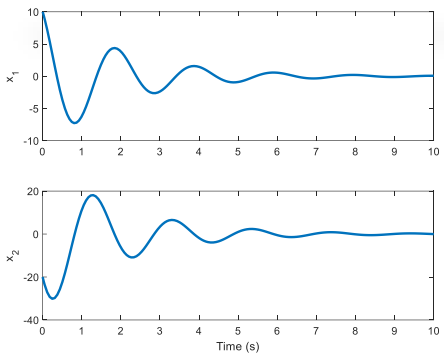
I. One shot simulation of the pendulum model without an input

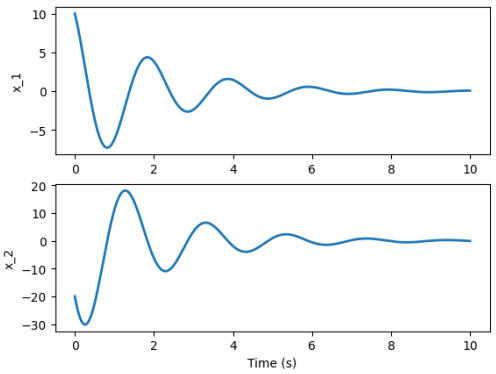
MATLAB

```
clear all, close all, clc

%% Simulation parameters
T0 = 0.0; % Initial time
Tf = 10;  % Final simulation time (duration)

%% Initial Conditions
x_1_0 = 10*pi/180; % theta
x_2_0 = -20*pi/180; % theta_dot
x_0 = [x_1_0, x_2_0]; % Transpose to make it a row vector

% Call the ode solver
[t,x] = ode45(@PendulumDynamics, [T0, Tf], x_0);

% Extract state variables
x1 = x(:, 1);
x2 = x(:, 2);

% Plot results
figure;
subplot(211);
plot(t, x1*180/pi, 'linewidth', 2);
ylabel('x_1');

subplot(212);
plot(t, x2*180/pi, 'linewidth', 2);
ylabel('x_2'); % Include LaTeX interpreter
xlabel('Time (s)');

% a function to define pendulum model
function dxdt = PendulumDynamics(t, x)
   l_p = 1; % Length of the pendulum
   m = 1;   % Mass of the pendulum bob
   c = 1;   % Damping coefficient
   g = 9.81; % Acceleration due to gravity

   % Input is considered zero
   u = 0;

   dxdt = zeros(2, 1);
   dxdt(1) = x(2);
   dxdt(2) = -(c / (m * l_p^2)) * x(2) - (g / l_p) * sin(x(1)) +
(1 / (m * l_p^2)) * u;
end
```

Python

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def main():
   # Simulation parameters
   T0 = 0.0
   Tf = 10

   # Initial Conditions
   x_1_0 = 10 * np.pi / 180  # theta
   x_2_0 = -20 * np.pi / 180  # theta_dot
   x_0 = [x_1_0, x_2_0]

   # Call the ode solver
   t = np.arange(T0, Tf, 0.01) # odeint requires time defined
unlike MATLAB ode
   x = odeint(PendulumDynamics, x_0, t)   # Transpose the
result

   # Extract state variables
   x1 = x[:,0]
   x2 = x[:,1]

   # Plot results
   plt.figure()

   plt.subplot(2, 1, 1)
   plt.plot(t, x1 * 180 / np.pi, linewidth=2)
   plt.ylabel('x_1')

   plt.subplot(2, 1, 2)
   plt.plot(t, x2 * 180 / np.pi, linewidth=2)
   plt.ylabel('x_2')
   plt.xlabel('Time (s)')

   plt.show()

# Function definition for Pendulum Dynamics
def PendulumDynamics(x, t):
   l_p = 1  # Length of the pendulum
   m = 1    # Mass of the pendulum bob
   c = 1    # Damping coefficient
   g = 9.81  # Acceleration due to gravity

   # Input is considered zero
   u = 0

   dxdt = np.zeros(2)
   dxdt[0] = x[1]
   dxdt[1] = -(c / (m * l_p ** 2)) * x[1] - (g / l_p) * np.sin(x[0])
+ (1 / (m * l_p ** 2)) * u

   return dxdt

if __name__ == '__main__':
   main()
```



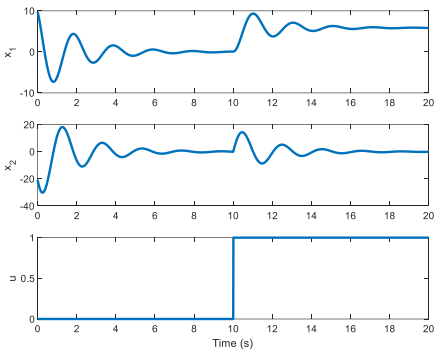


II. Simulation of the pendulum model (open-loop) with variable (step response test)

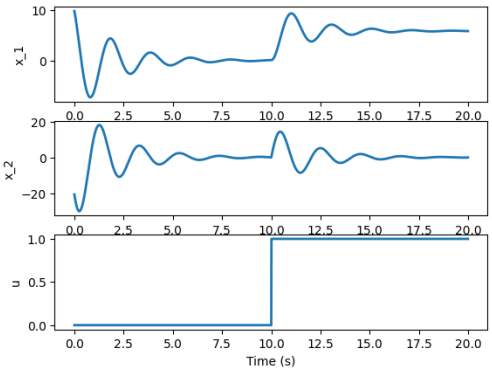
```
clear all, close all, clc

%% Simulation parameters
T0 = 0.0; % Initial time
Tf = 20;  % Final simulation time (duration)
dt = 0.01;  % Time step

%% Initial Conditions
x_1_0 = 10*pi/180; % theta
x_2_0 = -20*pi/180; % theta_dot
x_0 = [x_1_0, x_2_0]; % Transpose to make it a row vector

% Initialize time and u
time = T0:dt:Tf;
u = zeros(size(time));

% Using a loop to simulate the time
for i = 1:length(time)

   % Changing input
   u(i) = 0;

   if time(i) > 10
      u(i) = 1;
   end

   % Call the ode solver for each time step using an
anonymous function
   x_temp = ode45(@(t, x) PendulumDynamics(t, x, u(i)),
[time(i), time(i) + dt], x_0);

   % Update initial conditions for the next step
   x_0 = x_temp.y(:, end)';

   % Store state variables for plotting
   x1(i) = x_0(1);
   x2(i) = x_0(2);
end

% Plot results
figure;
subplot(311);
plot(time, x1*180/pi, 'linewidth', 2);
ylabel('x_1');

subplot(312);
plot(time, x2*180/pi, 'linewidth', 2);
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def main():
   # Simulation parameters
   T0 = 0.0
   Tf = 20
   dt = 0.01

   # Initial Conditions
   x_1_0 = 10 * np.pi / 180  # theta
   x_2_0 = -20 * np.pi / 180  # theta_dot
   x_0 = [x_1_0, x_2_0]

   # Initialize time and u
   time = np.arange(T0, Tf, dt)
   u = np.zeros(len(time))

   # Loop to simulate the time
   x1 = np.zeros(len(time))
   x2 = np.zeros(len(time))

   for i in range(len(time)):
      # Change input u at time > 10
      if time[i] > 10:
         u[i] = 1
      else:
         u[i] = 0

      # Call the ode solver for each time step
      x_temp = odeint(PendulumDynamics, x_0, [time[i],
time[i] + dt], args=(u[i],))

      # Update initial conditions for the next step
      x_0 = x_temp[-1, :]

      # Store state variables for plotting
      x1[i] = x_0[0]
      x2[i] = x_0[1]

   # Plot results
   plt.figure()

   plt.subplot(3, 1, 1)
   plt.plot(time, x1 * 180 / np.pi, linewidth=2)
   plt.ylabel('x_1')
```

```
ylabel('x_2');

subplot(313);
plot(time, u, 'linewidth', 2);
ylabel('u');
xlabel('Time (s)');

% Function to define pendulum model with input u
function dxdt = PendulumDynamics(t, x, u)
   l_p = 1; % Length of the pendulum
   m = 1;   % Mass of the pendulum bob
   c = 1;   % Damping coefficient
   g = 9.81; % Acceleration due to gravity

   dxdt = zeros(2, 1);
   dxdt(1) = x(2);
   dxdt(2) = -(c / (m * l_p^2)) * x(2) - (g / l_p) * sin(x(1)) +
(1 / (m * l_p^2)) * u;
end
```




```
    plt.subplot(3, 1, 2)
    plt.plot(time, x2 * 180 / np.pi, linewidth=2)
    plt.ylabel('x_2')

    plt.subplot(3, 1, 3)
    plt.plot(time, u, linewidth=2)
    plt.ylabel('u')
    plt.xlabel('Time (s)')

    plt.show()

# Function definition for Pendulum Dynamics with input u
def PendulumDynamics(x, t, u):
    l_p = 1  # Length of the pendulum
    m = 1    # Mass of the pendulum bob
    c = 1    # Damping coefficient
    g = 9.81  # Acceleration due to gravity

    dxdt = np.zeros(2)
    dxdt[0] = x[1]
    dxdt[1] = -(c / (m * l_p ** 2)) * x[1] - (g / l_p) * np.sin(x[0])
+ (1 / (m * l_p ** 2)) * u

    return dxdt

if __name__ == '__main__':
    main()
```



* by changing sin(x_1) to x_1 we can covert the system to its linear form. We can see for this system the nonlinearity is not dominant and becomes more effective for ($\boldsymbol{\theta} > \mathbf{0}$). The linear system code with one line change is below.

III. A PID controller for the pendulum model

```
clear all, close all, clc

%% Simulation parameters
T0 = 0.0; % Initial time
Tf = 20;  % Final simulation time (duration)
dt = 0.01;  % Time step

%% Initial Conditions
x_1_0 = 10*pi/180; % theta
x_2_0 = -20*pi/180; % theta_dot
x_0 = [x_1_0, x_2_0]; % Transpose to make it a row vector

% Initialisation
time = T0:dt:Tf;
u = zeros(size(time));
```

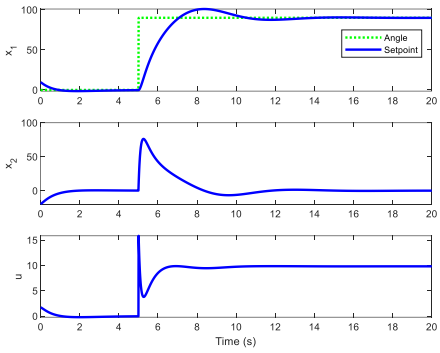
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def main():
    # Simulation parameters
    T0 = 0.0
    Tf = 20
    dt = 0.01

    # Initial Conditions
    x_1_0 = 10 * np.pi / 180  # theta
    x_2_0 = -20 * np.pi / 180  # theta_dot
    x_0 = np.array([x_1_0, x_2_0])
```

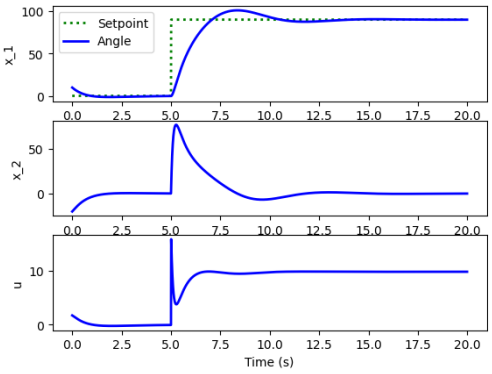
```
r = zeros(size(time));
e_i1 = 0;

% Using a loop to simulate the time
for i = 1:length(time)

   % Changing input
   if time(i) > 5
      r(i) = 90*pi/180;
   end

   % Call PID Controller
   [u(i), e_i1] = PID_Controller(r(i), x_0(1), x_0(2), e_i1, dt);

   % Call the ode solver for each time step using an
anonymous function
   [t, x_temp] = ode45(@(t, x) PendulumDynamics(t, x, u(i)),
[time(i), time(i) + dt], x_0);

   % Update initial conditions for the next step
   x_0 = x_temp(end, :);

   % Store state variables for plotting
   x_1(i) = x_0(1);
   x_2(i) = x_0(2);
end

% Plot results
figure;
subplot(311);

plot(time,r*180/pi, 'g:', 'linewidth', 2); % Setpoint
hold on;
plot(time,x_1*180/pi,'b', 'linewidth', 2);

ylabel('x_1');
legend('Angle','Setpoint');

subplot(312);
plot(time, x_2*180/pi,'b','linewidth',2);
ylabel('x_2');

subplot(313);
plot(time,u,'b','linewidth',2);
ylabel('u');
xlabel('Time (s)');

% Function to define pendulum model with input u
function dxdt = PendulumDynamics(t, x, u)
   l_p = 1; % Length of the pendulum
   m = 1;   % Mass of the pendulum bob
   c = 1;   % Damping coefficient
   g = 9.81; % Acceleration due to gravity

   dxdt = zeros(2, 1);
   dxdt(1) = x(2);
   dxdt(2) = -(c / (m * l_p^2)) * x(2) - (g / l_p) * sin(x(1)) +
(1 / (m * l_p^2)) * u;
end
```

```
   # Initialize time, u, and setpoint r
   time = np.arange(T0, Tf, dt)
   u = np.zeros(len(time))
   r = np.zeros(len(time))
   e_i1 = 0

   # Loop to simulate the time
   x_1 = np.zeros(len(time))
   x_2 = np.zeros(len(time))

   for i in range(len(time)):
      # Change input u at time > 5
      if time[i] > 5:
         r[i] = 90 * np.pi / 180

      # Call PID Controller
      u[i], e_i1 = PID_Controller(r[i], x_0[0], x_0[1], e_i1, dt)

      # Call the ode solver for each time step
      x_temp = odeint(PendulumDynamics, x_0, [time[i],
time[i] + dt], args=(u[i],))

      # Update initial conditions for the next step
      x_0 = x_temp[-1, :]

      # Store state variables for plotting
      x_1[i] = x_0[0]
      x_2[i] = x_0[1]

   # Plot results
   plt.figure()

   plt.subplot(3, 1, 1)
   plt.plot(time, r * 180 / np.pi, 'g:', linewidth=2)  # Setpoint
   plt.plot(time, x_1 * 180 / np.pi, 'b', linewidth=2)
   plt.ylabel('x_1')
   plt.legend(['Setpoint', 'Angle'])

   plt.subplot(3, 1, 2)
   plt.plot(time, x_2 * 180 / np.pi, 'b', linewidth=2)
   plt.ylabel('x_2')

   plt.subplot(3, 1, 3)
   plt.plot(time, u, 'b', linewidth=2)
   plt.ylabel('u')
   plt.xlabel('Time (s)')

   plt.show()

# Function definition for Pendulum Dynamics with input u
def PendulumDynamics(x, t, u):
   l_p = 1  # Length of the pendulum
   m = 1    # Mass of the pendulum bob
   c = 1    # Damping coefficient
   g = 9.81  # Acceleration due to gravity

   dxdt = np.zeros(2)
   dxdt[0] = x[1]
   dxdt[1] = -(c / (m * l_p ** 2)) * x[1] - (g / l_p) * np.sin(x[0])
+ (1 / (m * l_p ** 2)) * u
```

```
% Function for PID Controller
function [u_c, e_i1] = PID_Controller(r, x_1, x_2, e_i1, Ts)
   Kp = 10.0;
   Ki = 10.0;
   Kd = 10.0;

   e = r - x_1;
   % Integration
   e_i = e_i1 + Ts*e;
   e_i1 = e_i;

   up = Kp*e; % calculation of proprtional branch
   ui = Ki*e_i; % calculation of integral branch
   ud = Kd*x_2; % derivative branch from output derivative

   u_c = up + ui - ud;
end
```

```
   return dxdt

# Function definition for PID Controller
def PID_Controller(r, x_1, x_2, e_i1, Ts):
   Kp = 10.0
   Ki = 10.0
   Kd = 10.0

   e = r - x_1

   # Integration
   e_i = e_i1 + Ts * e
   e_i1 = e_i

   up = Kp * e # calculation of proprtional branch
   ui = Ki * e_i # calculation of integral branch
   ud = Kd * x_2 # derivative branch from output derivative

   u_c = up + ui - ud

   return u_c, e_i1

if __name__ == '__main__':
   main()
```

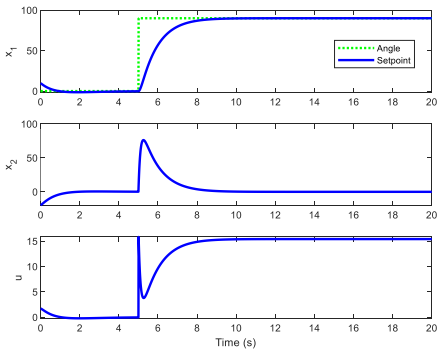


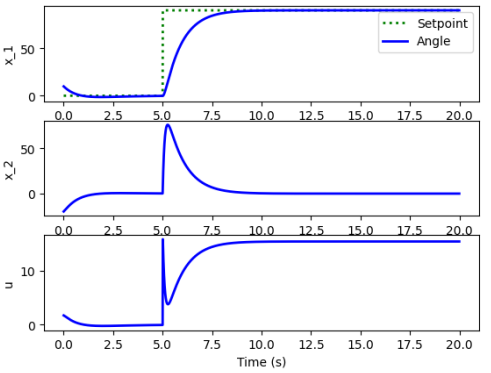


* by changing sin(x_1) to x_1 we can covert the system to its linear form. We can see for this system the nonlinearity is not dominant and becomes more effective for ($\boldsymbol{\theta} > \mathbf{0}$). The linear system code with one line change is below.

```
dxdt(2) = -(c / (m * l_p^2)) * x(2) - (g / l_p) * x(1) + (1 / (m * l_p^2)) * u; %linear model
```

```
dxdt[1] = -(c / (m * l_p ** 2)) * x[1] - (g / l_p) * x[0] + (1 / (m * l_p ** 2)) * u # linear model
```





An advanced topic- simulation with custom integrator function

In some applications like real-time implementation and to reduce computational burdens, we may prefer not to use built-in ode functions in MATLAB and Python.
Euler's method is a simple numerical technique for solving ordinary differential equations. It uses the first-order Taylor expansion to approximate the next value based on the current value and the derivative. The formula is given by:

$$x_{n+1} = x_n + \Delta t f(x_n, t_n)$$

Where x_{n+1} is the next value of states, x_n the current value, Δt is the time step, $f(x_n, t_n)$ is the deivatives function (state-space equations), and t_n is current time.
The Runge-Kutta method is a higher-order numerical technique that improves accuracy compared to Euler's method. The fourth order Runge-Kutta formula is widely used and is given by:

$$k_1 = x_n + \Delta t f(x_n, u_n)$$
$$k_2 = f(x_n + 0.5\Delta t k_1, u_{n+0.5})$$
$$k_3 = f(x_n + 0.5\Delta t k_2, u_{n+0.5})$$
$$k_4 = h\ f(x_n + 0.5\Delta t k_3, u_{n+1})$$
$$x_{n+1} = x_n + (1/6)\ (k_1 + 2k_2 + 2k_3 + k_4)$$

here k_1, k_2, k_3, k_3 are intermediate slopes, x_{n+1} and x_n are the next and current values of states, Δt is the time step, $f(x_n, u_n)$ is the derivative function. At first glance, it might seem complicated, but a simplified version of this algorithm can be recursively implemented. We can observe that the following implementation provides similar results to that obtained from the MATLAB "ode()" function. The following simulation performs the same example of a pendulum with a custom "Integration()" function that provides Euler or Runge-Kutta options.

I. Pendulum simulation with a custom integrator:

MATLAB

```
clear all, close all, clc

%% Simulation parameters
T0 = 0.0; % Initial time
Tf = 20;  % Final simulation time (duration)
dt = 0.01;  % Time step

%% Initial Conditions
x_1_0 = 10*pi/180; % theta
x_2_0 = -20*pi/180; % theta_dot
x_0 = [x_1_0, x_2_0]; % initial states

% Initialisation
time = T0:dt:Tf;
u = zeros(size(time));
e_i1 = 0;

% Using a loop to simulate the system
for i = 1:length(time)

   r(i) = 0;
   % Changing setpoint
   if time(i) > 5
      r(i) = 90*pi/180;
   end

   % Call PID Controller
   [u(i), e_i1] = PID_Controller(r(i), x_0(1), x_0(2), e_i1, dt);
```

Python

```
import numpy as np
import matplotlib.pyplot as plt

def main():
   # Simulation parameters
   T0 = 0.0
   Tf = 20
   dt = 0.01

   # Initial Conditions
   x_1_0 = 10 * np.pi / 180  # theta
   x_2_0 = -20 * np.pi / 180  # theta_dot
   x_0 = np.array([x_1_0, x_2_0])

   # Initialize time, u, and setpoint r
   time = np.arange(T0, Tf, dt)
   u = np.zeros(len(time))
   r = np.zeros(len(time))
   e_i1 = 0

   # Loop to simulate the system
   x_1 = np.zeros(len(time))
   x_2 = np.zeros(len(time))

   for i in range(len(time)):
      # Changing setpoint
      if time[i] > 5:
         r[i] = 90 * np.pi / 180
```

MATLAB

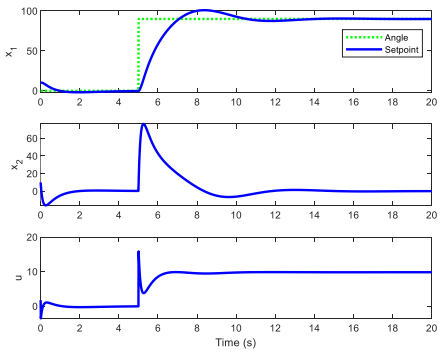
```
    % Custom integration using Runge-Kutta or Euler methods
    x_temp = Integration(@PendulumDynamics, x_0, u(i), dt, 'RungeKutta');

    % Update initial conditions for the next step
    x_0 = x_temp;

    % Store state variables for plotting
    x_1(i) = x_0(1);
    x_2(i) = x_0(2);
end

% Plot results
figure;
subplot(311);
plot(time, r*180/pi, 'g:', 'linewidth', 2); % Setpoint
hold on;
plot(time, x_1*180/pi, 'b', 'linewidth', 2);
ylabel('x_1');
legend('Angle','Setpoint');

subplot(312);
plot(time, x_2*180/pi, 'b', 'linewidth', 2);
ylabel('x_2');

subplot(313);
plot(time, u, 'b', 'linewidth', 2);
ylabel('u');
xlabel('Time (s)');

% Function to define pendulum model with input u
function dxdt = PendulumDynamics(x, u)
    l_p = 1; % Length of the pendulum
    m = 1;   % Mass of the pendulum bob
    c = 1;   % Damping coefficient
    g = 9.81; % Acceleration due to gravity

    dxdt = zeros(2, 1);
    dxdt(1) = x(2);
    dxdt(2) = -(c / (m * l_p^2)) * x(2) - (g / l_p) * sin(x(1)) + (1 / (m * l_p^2)) * u;
end

% Function for PID Controller
function [u_c, e_i1] = PID_Controller(r, x_1, x_2, e_i1, Ts)
    Kp = 10.0;
    Ki = 10.0;
    Kd = 10.0;

    e = r - x_1;
    % Integration
    e_i = e_i1 + Ts * e;
    e_i1 = e_i;

    up = Kp*e; % calculation of proportional branch
    ui = Ki*e_i; % calculation of integral branch
    ud = Kd*x_2; % derivative branch from output derivative
```

Python

```
    # Call PID Controller
    u[i], e_i1 = PID_Controller(r[i], x_0[0], x_0[1], e_i1, dt)

    # Custom integration using RungeKutta or Euler methods
    x_temp = Integration(PendulumDynamics, x_0, u[i], dt, 'RungeKutta')

    # Update initial conditions for the next step
    x_0 = x_temp

    # Store state variables for plotting
    x_1[i] = x_0[0]
    x_2[i] = x_0[1]

# Plot results
plt.figure()

plt.subplot(3, 1, 1)
plt.plot(time, r * 180 / np.pi, 'g:', linewidth=2)  # Setpoint
plt.plot(time, x_1 * 180 / np.pi, 'b', linewidth=2)
plt.ylabel('x_1')
plt.legend(['Setpoint', 'Angle'])

plt.subplot(3, 1, 2)
plt.plot(time, x_2 * 180 / np.pi, 'b', linewidth=2)
plt.ylabel('x_2')

plt.subplot(3, 1, 3)
plt.plot(time, u, 'b', linewidth=2)
plt.ylabel('u')
plt.xlabel('Time (s)')

plt.show()

# Function to define pendulum model with input u
def PendulumDynamics(x, u):
    l_p = 1  # Length of the pendulum
    m = 1    # Mass of the pendulum bob
    c = 1    # Damping coefficient
    g = 9.81  # Acceleration due to gravity

    dxdt = np.zeros(2)
    dxdt[0] = x[1]
    dxdt[1] = -(c / (m * l_p ** 2)) * x[1] - (g / l_p) * np.sin(x[0]) + (1 / (m * l_p ** 2)) * u

    return dxdt

# Function for PID Controller
def PID_Controller(r, x_1, x_2, e_i1, Ts):
    Kp = 10.0
    Ki = 10.0
    Kd = 10.0

    e = r - x_1

    # Integration
```

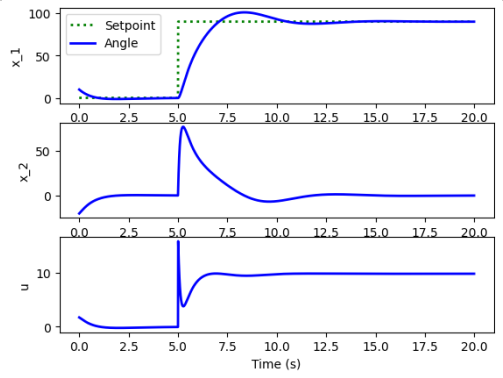
MATLAB

```
    u_c = up + ui - ud;
end

function x_out = Integration(dx, state, inputs, dt, method)
    % select between Runge Kutta or Euler methods
    if strcmp(method, 'RungeKutta') % To compare strings
        f1 = dx(state, inputs);
        x1 = state + (dt/2.0)*f1;

        f2 = dx(x1, inputs);
        x2 = state + (dt/2.0)*f2;

        f3 = dx(x2, inputs);
        x3 = state + dt*f3;

        f4 = dx(x3, inputs);
        x_out = state + (dt/6.0)*(f1 + 2*f2 + 2*f3 + f4);

    elseif strcmp(method, 'Euler')
        f1 = dx(state, inputs);
        x_out = state + dt*f1;
    else
        error('Invalid integration method. Use "RungeKutta" or "Euler" method.');
    end
end
```



Python

```
    e_i = e_i1 + Ts * e
    e_i1 = e_i

    up = Kp * e # calculation of proprtional branch
    ui = Ki * e_i # calculation of integral branch
    ud = Kd * x_2 # derivative branch from output derivative

    u_c = up + ui - ud

    return u_c, e_i1

# Function for custom integration using Runge-Kutta or Euler methods
def Integration(dx, state, inputs, dt, method):
    if method == 'RungeKutta':
        f1 = dx(state, inputs)
        x1 = state + (dt / 2.0) * f1

        f2 = dx(x1, inputs)
        x2 = state + (dt / 2.0) * f2

        f3 = dx(x2, inputs)
        x3 = state + dt * f3

        f4 = dx(x3, inputs)
        x_out = state + (dt / 6.0) * (f1 + 2 * f2 + 2 * f3 + f4)

    elif method == 'Euler':
        f1 = dx(state, inputs)
        x_out = state + dt * f1
    else:
        raise ValueError('Invalid integration method. Use "RungeKutta" or "Euler" method.')

    return x_out

if __name__ == '__main__':
    main()
```



9- Discrete-time control system analysis

This Section presents some examples to simulate digital systems and control loops.

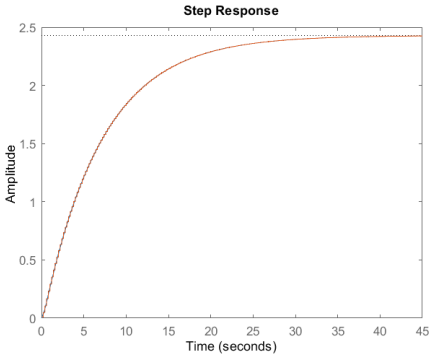
MATLAB | **Python**

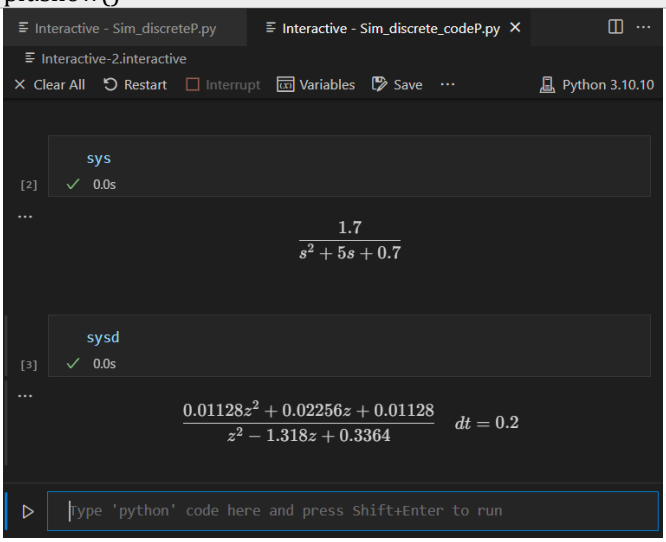
I. Discrete time transfer function extraction from continuous time TF: ($G(s) = 1.7/(s^2 + 5s + 0.7), T_s = 0.2$)

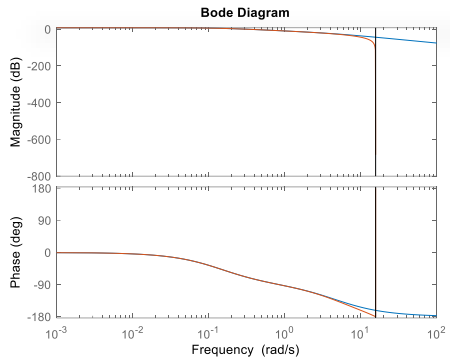
```
% system TF definition
sys=tf(1.7,[1 5 0.7])

Ts = 0.2; % sampling time selection
sysd = c2d(sys,Ts,'tustin') % system discretization

% comparison between continous and discrete model
figure, step(sys,sysd)

figure, bode(sys,sysd)
```

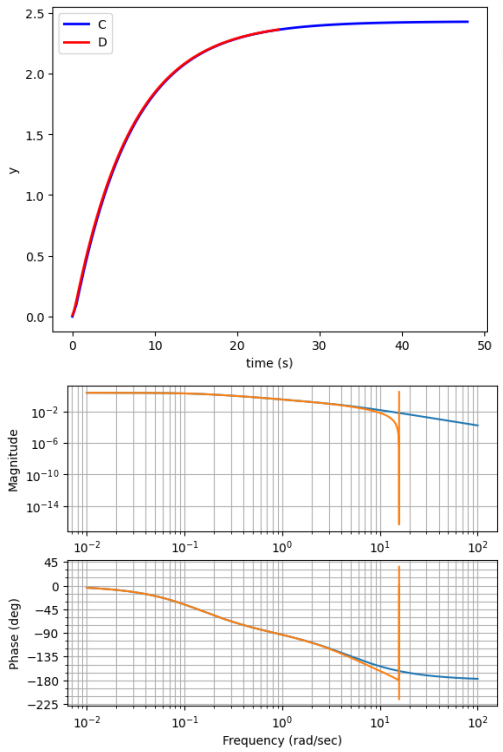
```
# system TF definition
sys = ct.tf([1.7],[1,5,0.7])

Ts = 0.2 # sampling time selection
sysd = ct.c2d(sys,Ts, method='bilinear') # system discretization

# comparison between continuous and discrete model
# step response calculation for both transfer functions
time,y = ct.step_response(sys)
timed,yd = ct.step_response(sysd)

# step response plot for both systems
plt.figure(1)
plt.plot(time,y,'b', label='C', linewidth=2)
plt.plot(timed,yd,'r', label='D', linewidth=2)
plt.xlabel("time (s)")
plt.ylabel("y")
plt.legend(loc='best')
plt.show()

# Bode plot for both systems
plt.figure(2)
ct.bode_plot(sys)
ct.bode_plot(sysd)
plt.show()
```

```
sys =

      1.7
 ---------------
 s^2 + 5 s + 0.7

Continuous-time transfer function.

sysd =

 0.01128 z^2 + 0.02256 z + 0.01128
 ---------------------------------
    z^2 - 1.318 z + 0.3364

Sample time: 0.2 seconds
Discrete-time transfer function.
```





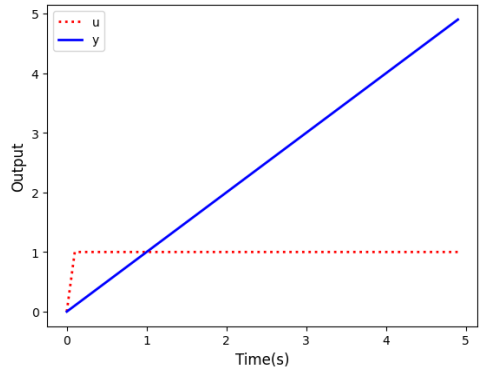


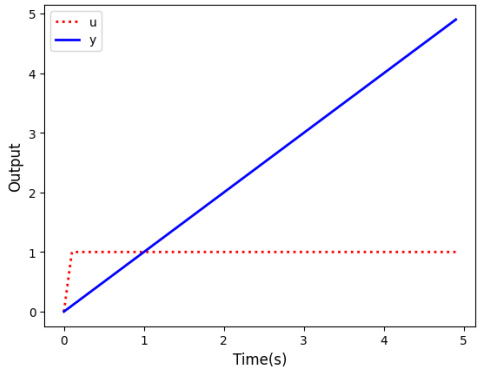


II. Generating a step signal and integrating it in discrete time ($H(z) = T_s/(1 - z^{-1})$)

```
Ts = 0.1; %sampling time
Tf = 5; %final simulation time
time = 0:0.1:Tf; %defining time sequence

% initial conditions, e.g: for y, i.e., y0
u(1) = 0;
y(1) = 0;

for t=2:length(time)  % loop to simulate the time
   u(t) = 1; % generating the step input
   y(t) = y(t-1) + Ts*u(t); % Integrating the step input
end

% plot setpoint and output
figure,
plot(time,u,'r:',time,y,'b','linewidth',1.5)
xlabel('Time(s)'), ylabel('Output')
legend('u','y')
```

```
Ts = 0.1 #sampling time
Tf = 5 #final simulation time
time = np.arange(0,Tf,Ts)  #defining time sequence

# initial conditions, e.g: for ys, i.e., ys0
u = np.zeros(len(time))
y = np.zeros(len(time))

for t in range(1,len(time)):  # loop to simulate the time
   u[t] = 1.0 # generating the step input
   y[t] = y[t-1] + Ts*u[t] # Integrating the step input

# end of for loop

# plot setpoint and output
plt.close("all")
plt.figure()
plt.plot(time, u,':r', label='u', linewidth=2)
plt.plot(time, y,'b', label='y', linewidth=2)
plt.xlabel('Time(s)', fontsize=12)
plt.ylabel('Output', fontsize=12)
plt.legend(loc='best')
plt.show()
```

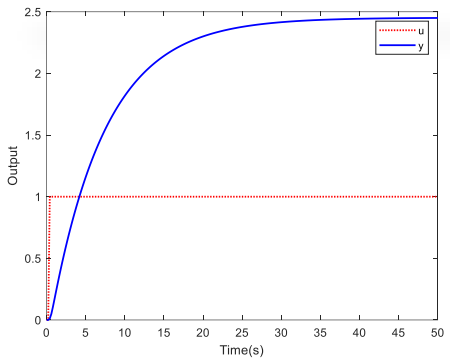


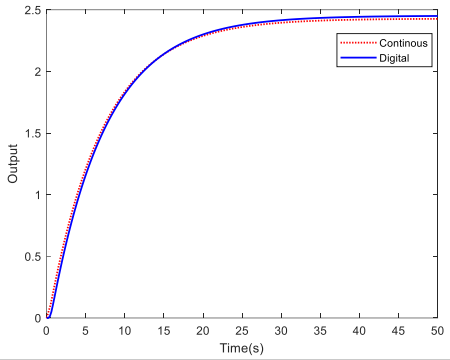


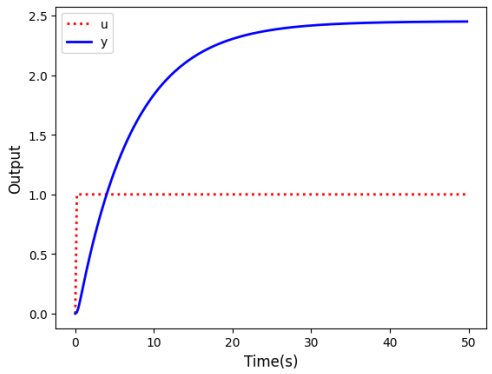
III. Discrete time (recursive) implementation of transfer functions by code:

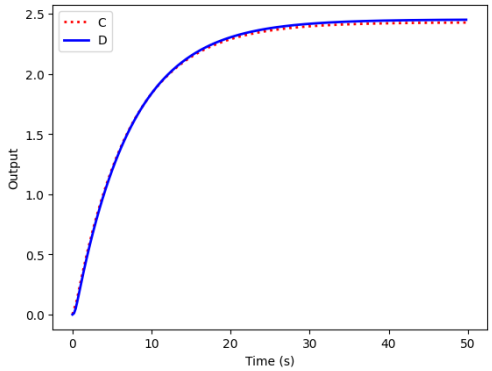
```
% system definition
sys=tf(1.7,[1 5 0.7])

Ts = 0.2; % sampling time selection
sysd = c2d(sys,Ts,'tustin') % system discretization

Tf = 50; % final simulation time
time = 0:Ts:Tf; %defining time

% system digital parameters
b0 = 0.01128;
b1 = 0.02256;
b2 = 0.01128;
a0 = 1;
a1 = -1.318;
a2 = 0.3364;

% initialisation
u(1) = 0;
u(2) = 0;
y(1) = 0;
y(2) = 0;

for t=3:1:length(time)  % lopp to simulate the time
   u(t) = 1; % generating the step input
   % discrete time system implementation
   y(t) = 1/a0*(-a1*y(t-1) - a2*y(t-2) + b0*u(t) + b1*u(t-1)
+ b2*u(t-2));
end

% plot discrete-time setpoint and output
figure,
plot(time,u,'r:',time,y,'b','linewidth',1.5)
xlabel('Time(s)'), ylabel('Output')
legend('u','y')

% step response comparison between implemented and
MATLAB
[yc,tc] = step(sys,50);
figure,
plot(tc,yc,'r:',time,y,'b','linewidth',1.5)
xlabel('Time(s)'), ylabel('Output')
legend('Continous','Digital')
```

```
# system TF definition
sys = ct.tf([1.7],[1,5,0.7])

Ts = 0.2 # sampling time selection
sysd = ct.c2d(sys,Ts, method='bilinear') # system
discretization

Tf = 50 #final simulation time
time = np.arange(0,Tf,Ts)  #defining time sequence

# system digital parameters
b0 = 0.01128
b1 = 0.02256
b2 = 0.01128
a0 = 1
a1 = -1.318
a2 = 0.3364

# initial conditions, e.g: for ys, i.e., ys0
u = np.zeros(len(time))
y = np.zeros(len(time))

for t in range(1,len(time)):  # loop to simulate the time
   u[t] = 1.0 # generating the step input
   #discrete time system implementation
   y[t] = 1/a0*(-a1*y[t-1] - a2*y[t-2] + b0*u[t] + b1*u[t-1] +
b2*u[t-2])

# end of the for loop

# plot setpoint and output
plt.close("all")
plt.figure()
plt.plot(time, u,':r', label='u', linewidth=2)
plt.plot(time, y,'b', label='y', linewidth=2)
plt.xlabel('Time(s)', fontsize=12)
plt.ylabel('Output', fontsize=12)
plt.legend(loc='best')
plt.show()

# step response calculation from Python control
timec,yc = ct.step_response(sys,Tf)

# step response comparison between implemented and
control
plt.figure(1)
plt.plot(timec,yc,':r', label='C', linewidth=2)
plt.plot(time,y,'b', label='D', linewidth=2)
plt.xlabel("Time (s)")
plt.ylabel("Output")
plt.legend(loc='best')
plt.show()
```









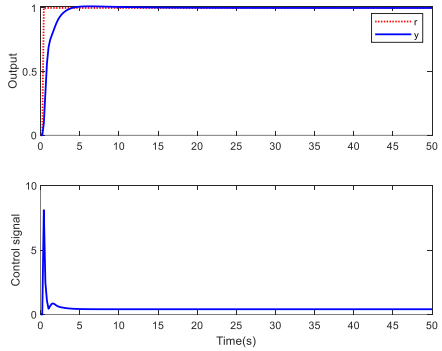
IV. Discrete time (recursive) PID controller implementation:

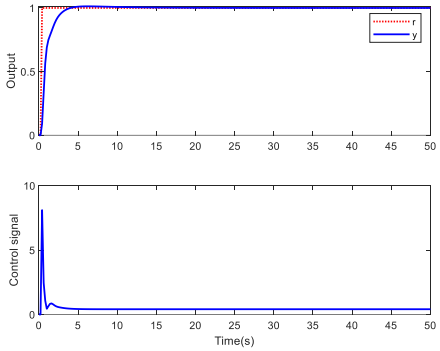
```
clear all, close all, clc

% system definition
sys=tf(1.7,[1 5 0.7]);

Ts = 0.2; % sampling time selection
sysd = c2d(sys,Ts,'tustin'); % system discretization

Tf = 50; % final simulation time
time = 0:Ts:Tf; %defining time

% system digital parameters
b0 = 0.01128;
b1 = 0.02256;
b2 = 0.01128;
a0 = 1;
a1 = -1.318;
a2 = 0.3364;

% controller gains
Kp = 3;
Ki = 0.5;
Kd = 1;

% initialisation
u_c(1) = 0;
u_c(2) = 0;
y(1) = 0;
y(2) = 0;
ei(1) = 0;
ei(2) = 0;

for t=3:1:length(time)  % loop to simulate the time
   r(t) = 1; % generating the step input
   e(t) = r(t)-y(t-1);
```

```
# system TF definition
sys = ct.tf([1.7],[1,5,0.7])

Ts = 0.2 # sampling time selection
sysd = ct.c2d(sys,Ts, method='bilinear') # system discretization

Tf = 50 #final simulation time
time = np.arange(0,Tf,Ts)  #defining time sequence

# system digital parameters
b0 = 0.01128
b1 = 0.02256
b2 = 0.01128
a0 = 1
a1 = -1.318
a2 = 0.3364

# controller gains
Kp = 3
Ki = 0.5
Kd = 1

# initial conditions, e.g: for ys, i.e., ys0
u = np.zeros(len(time))
y = np.zeros(len(time))
r = np.zeros(len(time))
ei = np.zeros(len(time))
e = np.zeros(len(time))
ed = np.zeros(len(time))
up = np.zeros(len(time))
ui = np.zeros(len(time))
ud = np.zeros(len(time))
u_c = np.zeros(len(time))
```

```
    up(t) = Kp*e(t); % P term of PID

    ei(t) = ei(t-1) + Ts*e(t); % error integration
    ui(t) = Ki*ei(t); % I term pf PID

    ed(t) = 1/Ts*(e(t)-e(t-1)); % error differentiation
    ud(t) = Kd*ed(t); % D term of PID

    u_c(t) = up(t) + ui(t) + ud(t); % control signal calculation
    % discrete time system implementation
    y(t) = 1/a0*(-a1*y(t-1) - a2*y(t-2) + b0*u_c(t) + b1*u_c(t-
1) + b2*u_c(t-2));
end

% plot discrete-time setpoint and output
figure,
subplot(211)
plot(time,r,'r:',time,y,'b','linewidth',1.5)
ylabel('Output')
legend('r','y')

% plot the control signal
subplot(212)
plot(time,u_c,'b','linewidth',1.5)
xlabel('Time(s)'), ylabel('Control signal')
```

```
for t in range(1,len(time)):  # loop to simulate the time
    r[t] = 1.0 # generating the step input
    e[t] = r[t]-y[t-1] # error signal

    up[t] = Kp*e[t] # P term of PID

    ei[t] = ei[t-1] + Ts*e[t] # error integration
    ui[t] = Ki*ei[t] # I term pf PID

    ed[t] = 1/Ts*(e[t]-e[t-1]) # error differentiation
    ud[t] = Kd*ed[t] # D term of PID

    u_c[t] = up[t] + ui[t] + ud[t] # control signal calculation


    #discrete time system implementation
    y[t] = 1/a0*(-a1*y[t-1] - a2*y[t-2] + b0*u_c[t] + b1*u_c[t-
1] + b2*u_c[t-2]);

# end of for loop

# plot setpoint and output
plt.close("all")
plt.figure()
plt.subplot(211)
plt.plot(time, r,':r', label='r', linewidth=2)
plt.plot(time, y,'b', label='y', linewidth=2)
plt.ylabel('Output', fontsize=12)
plt.legend(loc='best')

# plot setpoint and output
plt.subplot(212)
plt.plot(time, u_c,'b', label='y', linewidth=2)
plt.xlabel('Time(s)', fontsize=12)
plt.ylabel('Control signal', fontsize=12)
plt.legend(loc='best')
plt.show()
```





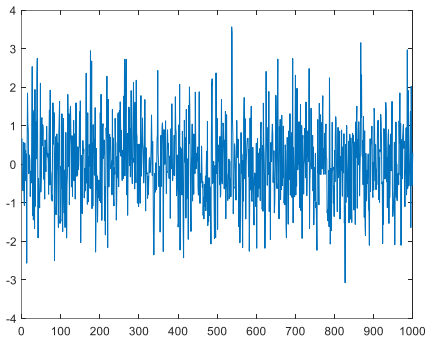
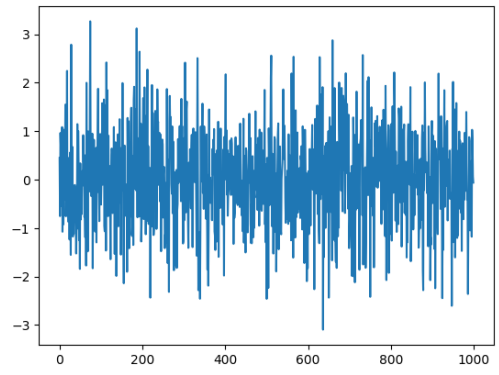
10- Data analysis and visualisation

Depending on the requirement, we need to conduct data analysis and subsequent visualisation. This is a crucial step to extract information from data. The initial stage in any data analysis involves descriptive analysis and visualisation. We have already acquainted ourselves with plotting functions; now we are to enhance our skills further. Descriptive data analysis entails several steps:

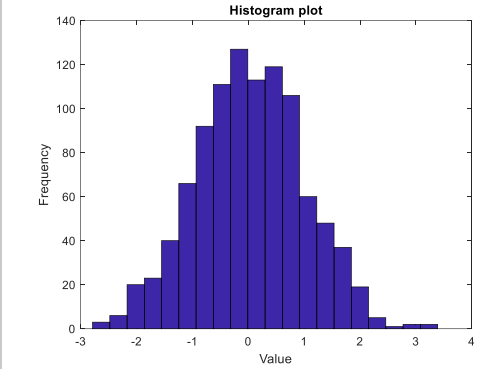
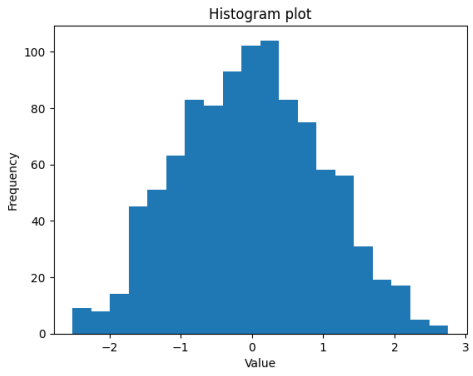
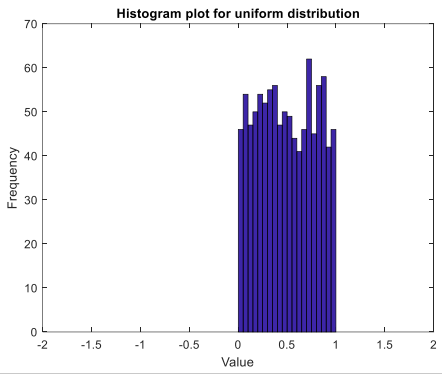
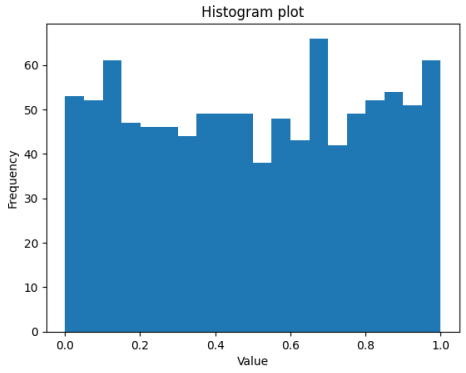
- Data collection (or simulation if data is unavailable)
- Data preparation including cleaning, merging, etc.
- Determining data dispersion
- Extracting key data features (maximum, minimum, mean value, standard deviation, etc.)
- Data visualisation
- Data analysis (correlation analysis, forecasting, etc.)

MATLAB	Python
I. Descriptive generation and plotting:	

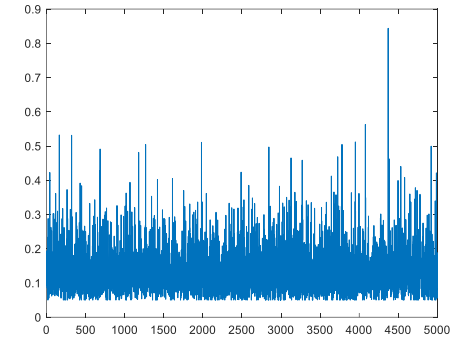
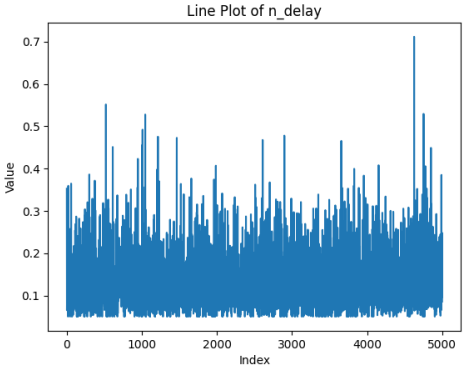
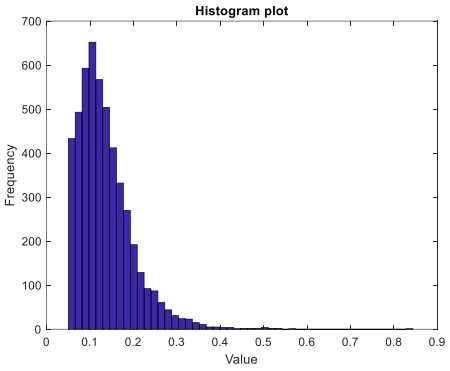
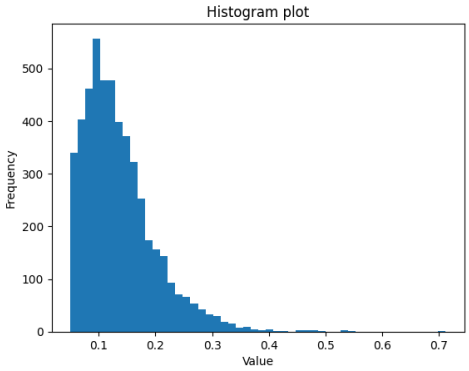
N=1000;
n = randn(1,1000); % generating a Gaussian noise/data
plot(n)
	N = 1000
	n = np.random.randn(1, N) # generating Gaussian
	plt.plot(n[0])
	plt.show()

	
max_val = max(n) %finding maximum value	max_val = np.max(n) print(max_val)
3.5784	3.267429572353682
[max_val, max_index] = max(n) %findinf the index of max	max_index = np.argmax(n) print(max_index)
max_val = 3.5699 max_index = 537	
min_val = min(n)	min_val = np.min(n) print(min_val)
-3.2320	-3.0977798184918863
mean_val = mean(n)	mean_val = np.mean(n) print(mean_val)
-0.0326	0.001079240110400903
std_val = std(n) % standard deviation	std_val = np.std(n) print(std_val)
std_val = 0.9990	0.9960657516198083
II. Histogram plot and dispersion analysis	

N=1000;
n = randn(1,1000); % generating a Gaussian noise/data
 	
hist(n, 20); % 20 bins for the histogram
 	
title('Histogram plot');
xlabel('Value');
	N = 1000
	n = np.random.randn(1, N) # generating Gaussian noise/data
	
	n = np.random.randn(1,1000)
	plt.hist(n[0], bins=20)
	plt.title('Histogram plot')

MATLAB	Python
ylabel('Frequency');	plt.xlabel('Value') plt.ylabel('Frequency') plt.show()
	
n2 = rand(1,N); %uniform noise plot…	n2 = np.random.rand(1,1000) plot…
	
* A more complex random number, like delay in communication which adheres to lognormal distribution (below)	

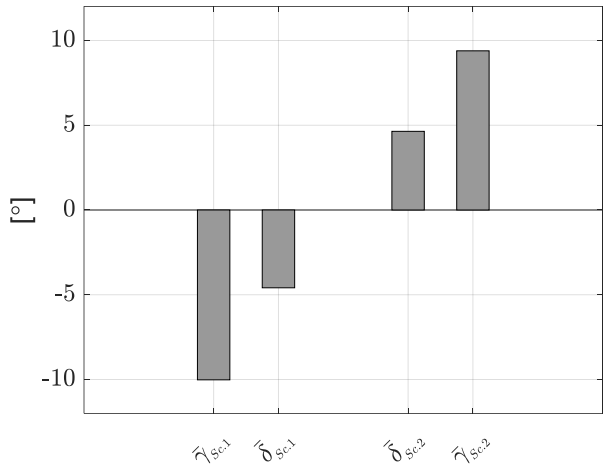
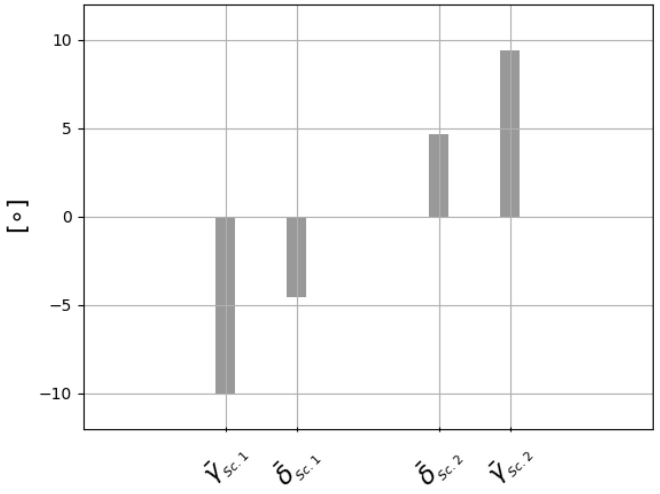
N = 5000;
 	
mu = -2.1; % mean value around av = exp(mu+sig^2/10)
sig = 0.45;
 	
n_delay = lognrnd(mu,sig,1,N); % lognormal to model network delay
 	
n_delay(n_delay < 0.05) = 0.05; % replave delay<0.05 with 0.05
 	
figure, plot(n_delay)
 	
figure,hist(n_delay, 50); % 50 bins for the histogram
 	
title('Histogram plot');
xlabel('Value');
ylabel('Frequency');
	import numpy as np
	import matplotlib.pyplot as plt
	
	N = 5000
	
	mu = -2.1 # mean value around av = exp(mu+sig^2/10)
	sig = 0.45
	
	# Simulating lognormal distribution to model network delay
	n_delay = np.random.lognormal(mu, sig, N)
	
	# Replacing values less than 0.05 with 0.05
	n_delay[n_delay < 0.05] = 0.05
	
	# Plotting line plot of n_delay
	plt.figure()
	plt.plot(n_delay)
	plt.title('Line Plot of n_delay')
	plt.xlabel('Index')
	plt.ylabel('Value')
	
	# Plotting histogram of n_delay with 50 bins
	plt.figure()
	plt.hist(n_delay, bins=50)
	plt.title('Histogram plot')
	plt.xlabel('Value')
	plt.ylabel('Frequency')

MATLAB	Python
	plt.show()
	
	

III. Bar plots and Latex interpreter (Example to plot/illustrate Greek variables using Latex interpreter)

```
clear all, close all, clc

x=[-2,-1,1,2]*3;

temp_high = [-10.02,-4.59,4.64,9.39]; %%%% amounts of hitch and delta
w1 = 0.5;
bar(x,temp_high,w1,'FaceColor',[0.6 0.6 0.6]) %%% plot the bar
grid on
axis([-12,12,-12,12])
ylabel('[\circ]')

%%% set the labels
ax = gca;
ax.XTick = x;
set(ax,'TickLabelInterpreter','latex','fontsize',16)
ax.XTickLabels = {'$\bar{\gamma}_{_{Sc.1}}$','$$\bar{\delta}_{_{Sc.1}}$$','$$\bar{\delta}_{_{Sc.2}}$$','$$\bar{\gamma}_{_{Sc.2}}$$'};
ax.XTickLabelRotation = 45;
```

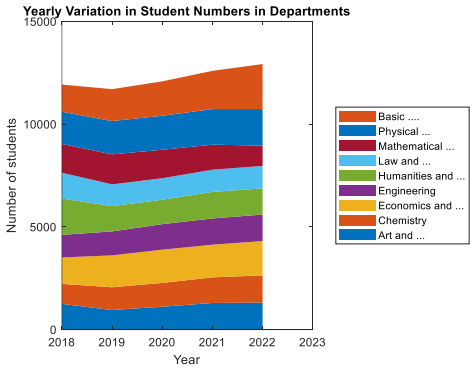
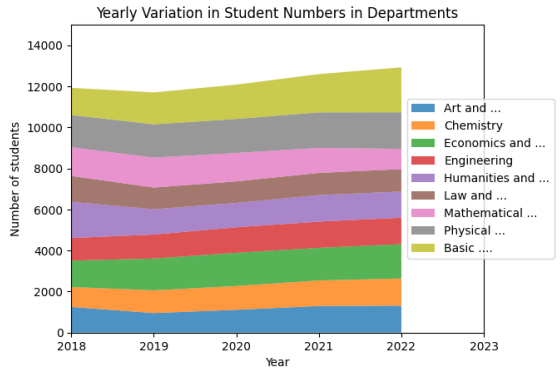
```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([-2, -1, 1, 2]) * 3

temp_high = [-10.02, -4.59, 4.64, 9.39]  # amounts of gamma and delta
w1 = 0.8  # Adjust the width of the bars
plt.close('all')
plt.bar(x, temp_high, w1, color=[0.6, 0.6, 0.6])  # plot the bar
plt.grid(True)
plt.axis([-12, 12, -12, 12])
plt.ylabel(r'$[\circ]$', fontsize=16)  # Use r to indicate raw string for LaTeX rendering

# Set the labels
ax = plt.gca()
ax.set_xticks(x)
plt.tick_params(axis='x', which='both', direction='inout')
plt.xticks(x, [r'$\bar{\gamma}_{_{Sc.1}}$', r'$\bar{\delta}_{_{Sc.1}}$', r'$\bar{\delta}_{_{Sc.2}}$', r'$\bar{\gamma}_{_{Sc.2}}$'],
           rotation=45, fontsize=16)

# Enable LaTeX rendering for the labels
plt.rc('text', usetex=True)

plt.show()
```

MATLAB	Python
	

IV. Plotting the streamgraph illustrating students number per year

```
clear all, close all, clc

% Inputs years, departments, and their corresponding student numbers
year = [2018, 2019, 2020, 2021, 2022];
population_by_continent = [
    1250, 952, 1112, 1295, 1313;
    973, 1108, 1159, 1247, 1325;
    1286, 1551, 1618, 1589, 1672;
    1101, 1167, 1243, 1277, 1289;
    1765, 1233, 1189, 1293, 1276;
    1260, 1057, 1046, 1080, 1092;
    1399, 1459, 1387, 1220, 987;
    1570, 1622, 1656, 1731, 1781;
    1321, 1554, 1672, 1862, 2189
];

% Plotting
figure;
colormap(parula(size(population_by_continent, 1)));

area(year, population_by_continent', 'EdgeColor', 'none');
legend({'Art and ...', 'Chemistry', 'Economics and ...', 'Engineering', 'Humanities and ...', 'Law and ...', 'Mathematical ...', 'Physical ...', 'Basic ....'}, 'Location', 'eastoutside');
xlim([2018 2023]);
ylim([0 15000]);
title('Yearly Variation in Student Numbers in Departments');
xlabel('Year');
ylabel('Number of students');
```

```
import matplotlib.pyplot as plt
import numpy as np

# Inputs years, departments and their corresponding student numbers
year = [2018, 2019, 2020, 2021, 2022]
population_by_continent = {
    'Art and ...': [1250, 952, 1112, 1295, 1313],
    'Chemistry': [973, 1108, 1159, 1247,      1325],
    'Economics and ...': [1286, 1551, 1618, 1589, 1672],
    'Engineering': [1101, 1167, 1243,          1277, 1289],
    'Humanities and ...': [1765, 1233, 1189, 1293, 1276],
    'Law and ...': [1260, 1057, 1046, 1080, 1092],
    'Mathematical ...': [1399, 1459, 1387, 1220, 987],
    'Physical ...': [1570, 1622, 1656, 1731, 1781],
    'Basic ....': [1321, 1554, 1672, 1862, 2189],
}

# Plotting the streamgraph illustrating students number per year
fig, ax = plt.subplots()
ax.stackplot(year, population_by_continent.values(),
             labels=population_by_continent.keys(), alpha=0.8)
ax.legend(loc='center left', bbox_to_anchor=(0.8, 0.5))
ax.set_xlim(xmin=2018, xmax=2023)
ax.set_ylim(ymin=0, ymax=15000)
ax.set_title('Yearly Variation in Student Numbers in Departments')
ax.set_xlabel('Year')
ax.set_ylabel('Number of students')

plt.show()
```

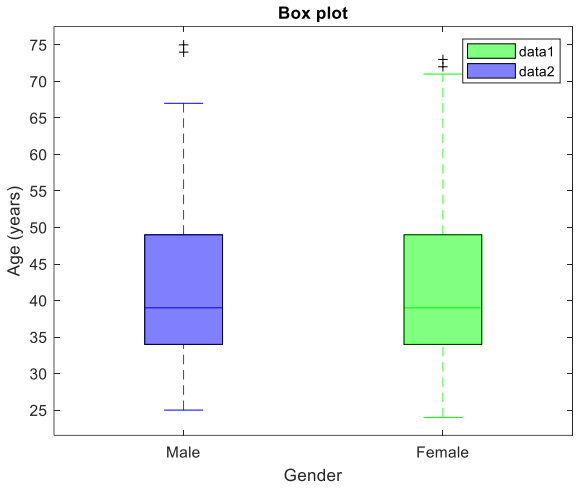
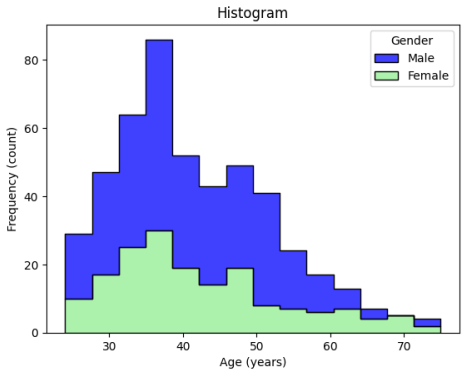
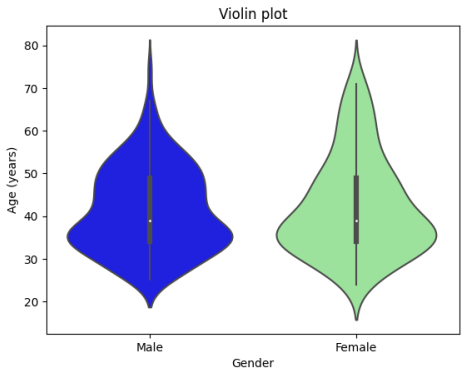
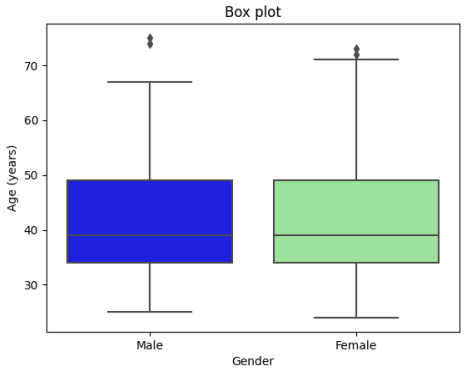
MATLAB	Python
	

V. Appealing violin and box plots in data visualisation

To the best of my knowledge, it is possible in MATLAB to generate these figures, but not easily or with a similar quality. Box plot example is provided that needs colour allocation for boxes:

```
clear all, close all, clc

% Read the population data
data = readtable('Population.txt', 'Delimiter', '\t', 'HeaderLines', 0, 'ReadVariableNames', false);
data.Properties.VariableNames = {'No', 'Age', 'Gender'};

% Extract data for Male and Female
age_male = data.Age(strcmp(data.Gender, 'Male'));
age_female = data.Age(strcmp(data.Gender, 'Female'));

% Box plot
figure;
boxplot(age_male, 'Positions', 1, 'Widths', 0.3, 'Colors', 'b', 'Symbol', 'k+');
hold on;
boxplot(age_female, 'Positions', 2, 'Widths', 0.3, 'Colors', 'g', 'Symbol', 'k+');
hold off;

% Customize the plot
title('Box plot');
xlabel('Gender');
ylabel('Age (years)');
set(gca, 'XTick', [1, 2], 'XTickLabels', {'Male', 'Female'});

% Add legend
legend('Male', 'Female');

% Fill box plots with colors
h = findobj(gca, 'Tag', 'Box');
for j = 1:length(h)
   patch(get(h(j), 'XData'), get(h(j), 'YData'), get(h(j), 'Color'), 'FaceAlpha', 0.5);
end
```

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# Read the population data
df = pd.read_csv('Population.txt', sep='\t', header=0, names=['No', 'Age', 'Gender'])

# Define custom color palette
custom_palette = {'Male': 'blue', 'Female': 'lightgreen'}

# Histogram
plt.figure()
sns.histplot(df, x='Age', hue='Gender', element='step', stat='count', common_norm=False, multiple='stack', palette=custom_palette, hue_order=['Male', 'Female'])
plt.title("Histogram")
plt.xlabel("Age (years)")
plt.ylabel("Frequency (count)")
plt.show()

# Violin plot
plt.figure()
sns.violinplot(x='Gender', y='Age', data=df, order=['Male', 'Female'], palette=custom_palette)
plt.title("Violin plot")
plt.ylabel("Age (years)")
plt.show()

# Box plot
plt.figure()
sns.boxplot(x='Gender', y='Age', data=df, order=['Male', 'Female'], palette=custom_palette)
plt.title("Box plot")
plt.ylabel("Age (years)")
plt.show()
```

MATLAB	Python
	  

VI. Manipulating files

TBC

References:

[1] Python Control Systems Library, https://python-control.org, https://github.com/python-control/Slycot.

[2] Fuller, S., Greiner, B., Moore, J., Murray, R., van Paassen, R. and Yorke, R., 2021, December. The python control systems library (python-control). In 2021 60th IEEE Conference on Decision and Control (CDC) (pp. 4875-4881). IEEE.

[3] Kim, J., Dynamic System Modelling and Analysis with MATLAB and Python: For Control Engineers, John Wiley & Sons, 2022.

[4] Bucher, R., "Python for control purposes," Scuola Universitaria Professionale della Svizzera Italiana Dipartimento Tecnologie Innovative, no. 102, Sep. 2019.

[5] Gordon, S. I., and Guilfoos, B., Introduction to Modeling and Simulation with MATLAB® and Python, Chapman and Hall/CRC, 2017.

[6] Franklin, G. F., Powell, J. D., and Emami-Naeini, A., Feedback Control of Dynamic Systems, 8th ed., Pearson, 2019.

[7] Astrom, K. J., and Murray, R. M., Feedback Systems: An Introduction for Scientists and Engineers, Princeton University Press, 2008.

[8] Sarhadi, P., "Simple-Python-Control repository", GitHub, [Online]. Available: https://github.com/Psarhadi/Simple-Python-Control.

[9] NumPy for MATLAB users, https://numpy.org/doc/stable/user/numpy-for-matlab-users.html