CS65K Robotics

Modelling, Planning and Control

Appendix C: Feedback Loop

LECTURE 13: DYNAMIC MODELLING

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Introduction to Dynamic Modelling
- Solve differential equation by ODEINT
- Balance Equations
- Linearization
- First-Order Equations
- Time Delay
- FOPDT Graphical Fit
- FOPDT Optimization Fit

Objectives

- Laplace Transforms
- Transfer Functions
- State Space
- Second Order Equation
- SOPDT Graphical Fit
- SOPDT Optimization Fit
- Simulate TF, SS, ODE

Dynamic Model

SECTION 1

Dynamic Model

- Dynamic models are essential for understanding the system dynamics in open-loop (manual mode) or for closed-loop (automatic) control.
- These models are either derived from data (empirical) or from more fundamental relationships (first principles, physics-based) that rely on knowledge of the process.
- A combination of the two approaches is often used in practice where the form of the equations are developed from fundamental balance equations and unknown or uncertain parameters are adjusted to fit process data.

Dynamic Model

- In engineering, there are 4 common balance equations from conservation principles including mass, momentum, energy, and species (see **Balance Equations**).
- An alternative to physics-based models is to use input-output data to develop empirical dynamic models such as **first-order** or **second-order** systems.

Charge conservation	$\nabla \cdot \mathbf{E}_{ri} = \frac{z_i F C_i}{\varepsilon}$
Mass conservation	$\frac{\partial C_i}{\partial t} + \nabla \cdot (C_i \mathbf{u}_i) = \dot{m}_{ci}$
Momentum conservation	$\frac{\partial}{\partial t}(\rho_{mi} \mathbf{u}_i) + \nabla \cdot (\rho_{mi} \mathbf{u}_i \mathbf{u}_i) = \nabla \boldsymbol{\sigma}_i + \rho_{ci} \mathbf{E} + \rho_{mi} \mathbf{g}$
Concentration conservation	$\rho_m \left[\frac{\partial C_i}{\partial t} + \nabla \cdot (C_i \mathbf{u}_i)\right] = \nabla \cdot \left(\lambda_i \nabla C_i - \frac{z_i F \lambda_i}{RT} C_i \mathbf{E}\right) + \dot{m}_{ci}$
Current equation	$\mathbf{J}_i = -z_i F D_i \nabla C_i + \frac{{z_i}^2 F^2}{RT} D_i C_i \mathbf{E} + z_i F C_i \mathbf{u}_i$

Steps in Dynamic Modeling

The following are general guidelines for developing a dynamic model. The process is iterative as simulation results help inform modeling assumptions or correct errors in the dynamic balance equations.

1. Identify objective for the simulation
2. Draw a schematic diagram, labeling process variables
3. List all assumptions
4. Determine spatial dependence
    a. yes = Partial Differential Equation (PDE)
    b. no = Ordinary Differential Equation (ODE)
5. Write dynamic balances (mass, species, energy)
6. Other relations (thermo, reactions, geometry, etc.)

Steps in Dynamic Modeling

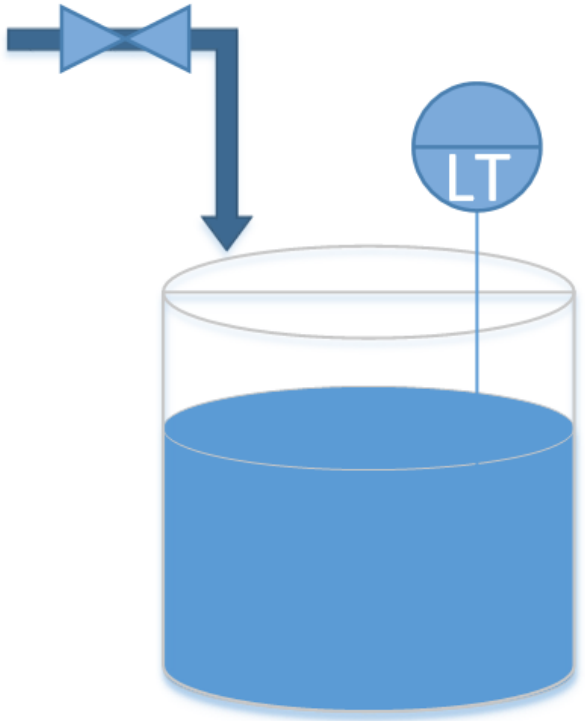
7. Degrees of freedom, does number of equations = number of unknowns?
8. Classify inputs as
    a. Fixed values
    b. Disturbances
    c. Manipulated variables
9. Classify outputs as
    a. States
    b. Controlled variables
10. Simplify balance equations based on assumptions
11. Simulate steady state conditions (if possible)
12. Simulate the output with an input step

Case Study

SECTION 2

A Beginning Example: Filling a Water Tank

- Consider a cylindrical tank with no outlet flow and an adjustable inlet flow. The inlet flow rate is not measured but there is a level measurement that shows how much fluid has been added to the tank.
- The objective of this exercise is to develop a model that can maintain a certain water level by automatically adjusting the inlet flow rate. See the subsequent section on **P-only control** for the tank level controller design.



A Beginning Example: Filling a Water Tank

- Diagram of a tank with an inlet and no outlet. The symbol **LT** is an abbreviation for *Level Transmitter*.
- A first step is to develop a dynamic model of how the inlet flow rate affects the level in the tank. A starting point for this model is a **balance equation**.

$$\frac{dm}{dt} = \dot{m}_{in} - \dot{m}_{out}$$

- The accumulation term is a differential variable such as *dm/dt* for mass. In this case, the accumulation of mass is equal to only an inlet flow and no outlet, generation, or consumption terms.

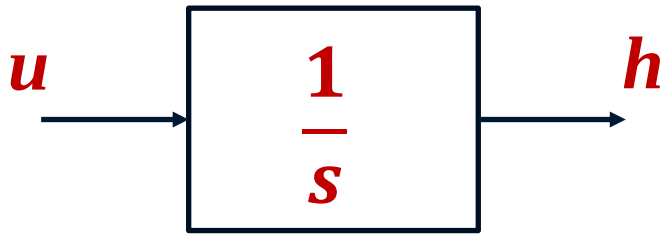
Assumptions

- The next objective is to simplify the expression and transform it into a relationship between height *h* and the valve opening *u (0-100%)*. For liquid water, density is nearly constant even over wide temperature ranges and the mass is equal to the density multiplied by the volume *V*. Assuming a constant cross-sectional area gives *V = h A* and a linear correlation between valve opening and inlet flow gives the following relationship.
  - $\rho A \frac{dh}{dt} = c\ u$, where $\dot{m_{in}} = c\ u$
- where *c* is a constant that relates valve opening to inlet flow.

Python Sample Program

Demo Program: water.py

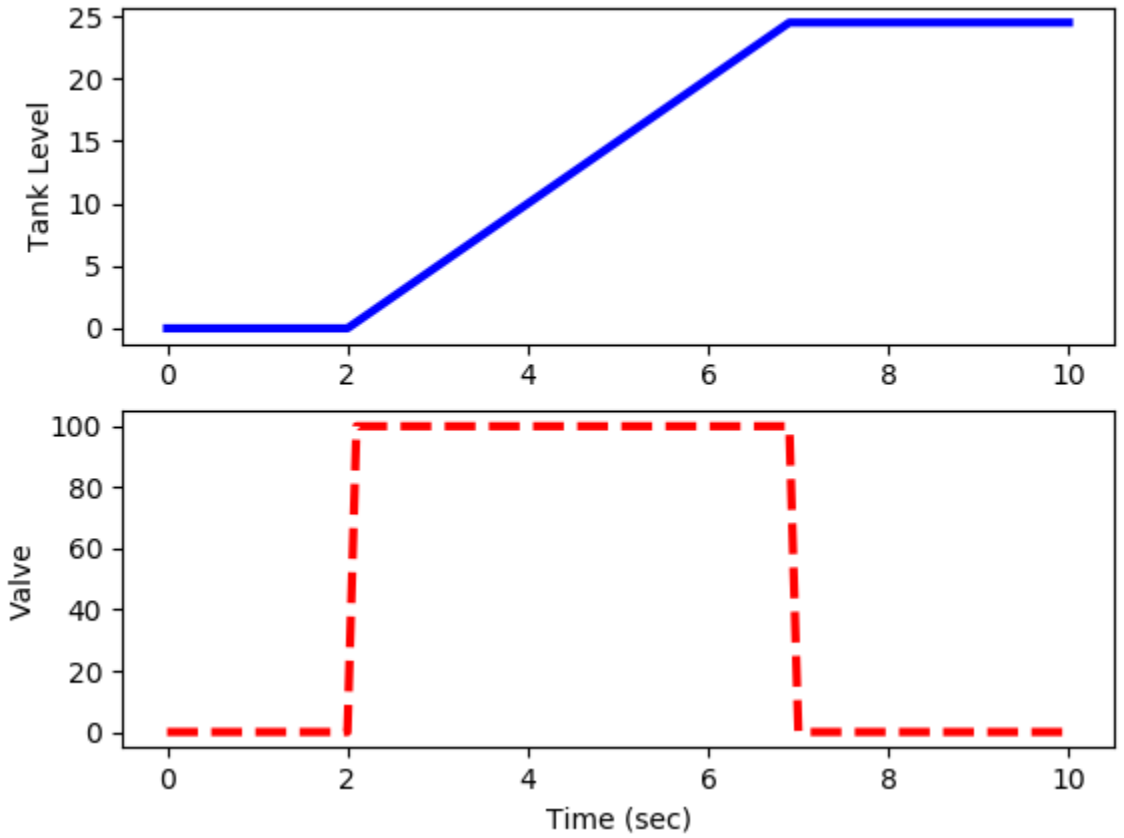
- Use Euler's method and ODE. $h' = h + \frac{dh}{dt} * \Delta t$
- u input
- tank function calculates the change of the tank level $\frac{dh}{dt}$
- [0, 0.1] is the step, 0.1 sec a point
- Open-loop. No feed back
- Level is h, time is t, and dLevel_dt is dh/dt
- y[-1] is the last point after integral



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

# define tank model
def tank(Level,time,c,valve):
    rho = 1000.0 # water density (kg/m^3)
    A = 1.0      # tank area (m^2)
    # calculate derivative of the Level
    dLevel_dt = (c/(rho*A)) * valve
    return dLevel_dt

# time span for the simulation for 10 sec, every 0.1 sec
ts = np.linspace(0,10,101)

# valve operation
c = 50.0          # valve coefficient (kg/s / %open)
u = np.zeros(101) # u = valve % open
u[21:70] = 100.0  # open valve between 2 and 7 seconds

# level initial condition
Level0 = 0

# for storing the results
z = np.zeros(101)
```

```
# simulate with ODEINT
for i in range(100):
    valve = u[i+1]
    y = odeint(tank,Level0,[0,0.1],args=(c,valve))
    Level0 = y[-1] # take the last point
    z[i+1] = Level0 # store the level for plotting

# plot results
plt.figure()
plt.subplot(2,1,1)
plt.plot(ts,z,'b-',linewidth=3)
plt.ylabel('Tank Level')
plt.subplot(2,1,2)
plt.plot(ts,u,'r--',linewidth=3)
plt.ylabel('Valve')
plt.xlabel('Time (sec)')
plt.show()
```

Output


Tank Level
25
20
15
10
5
0
0
2
4
6
8
10
Valve
100
80
60
40
20
0
0
2
4
6
8
10
Time (sec)

Solve with ODEINT

SECTION 3

odeint

from scipy.integrate import odeint

- Differential equations are solved in Python with the Scipy.integrate package using function **ODEINT**. Another Python package that solves differential equations is GEKKO. See this link for the same tutorial in GEKKO versus ODEINT. ODEINT requires three inputs:

```
y = odeint(model, y0, t)
```

- **model**: Function name that returns derivative values at requested y and t values as dydt = model(y,t)
- **y0**: Initial conditions of the differential states
- **t**: Time points at which the solution should be reported. Additional internal points are often calculated to maintain accuracy of the solution but are not reported.

Simple Differential Equation

Demo Program: odeint1.py

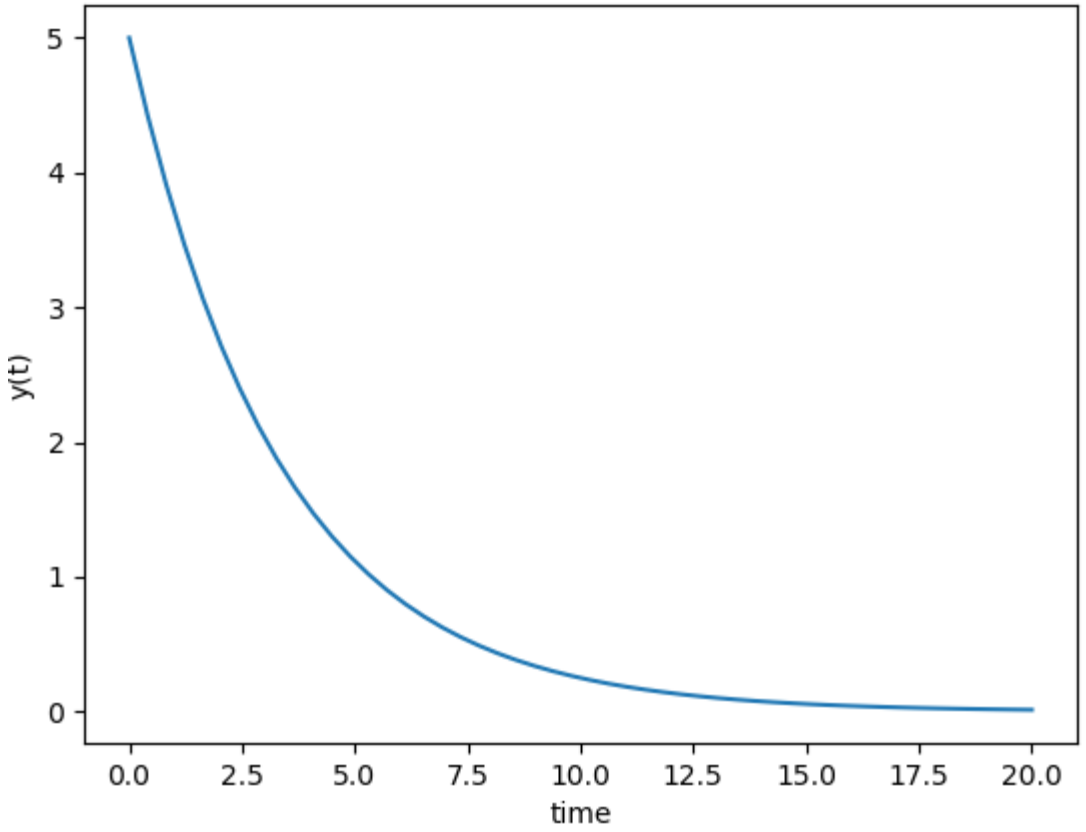
- An example of using *ODEINT* is with the following differential equation with parameter *k=0.3*, the initial condition $y_0=5$ and the following differential equation.

$$\frac{dy(t)}{dx} = -k\, y(t)$$

- The Python code first imports the needed Numpy, Scipy, and Matplotlib packages. The model, initial conditions, and time points are defined as inputs to *ODEINT* to numerically calculate *y(t)*.

```python
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
# function that returns dy/dt
def model(y,t):
    k = 0.3
    dydt = -k * y
    return dydt

# initial condition
y0 = 5
# time points
t = np.linspace(0,20)
# solve ODE
y = odeint(model,y0,t)

# plot results
plt.plot(t,y)
plt.xlabel('time')
plt.ylabel('y(t)')
plt.show()
```

Output



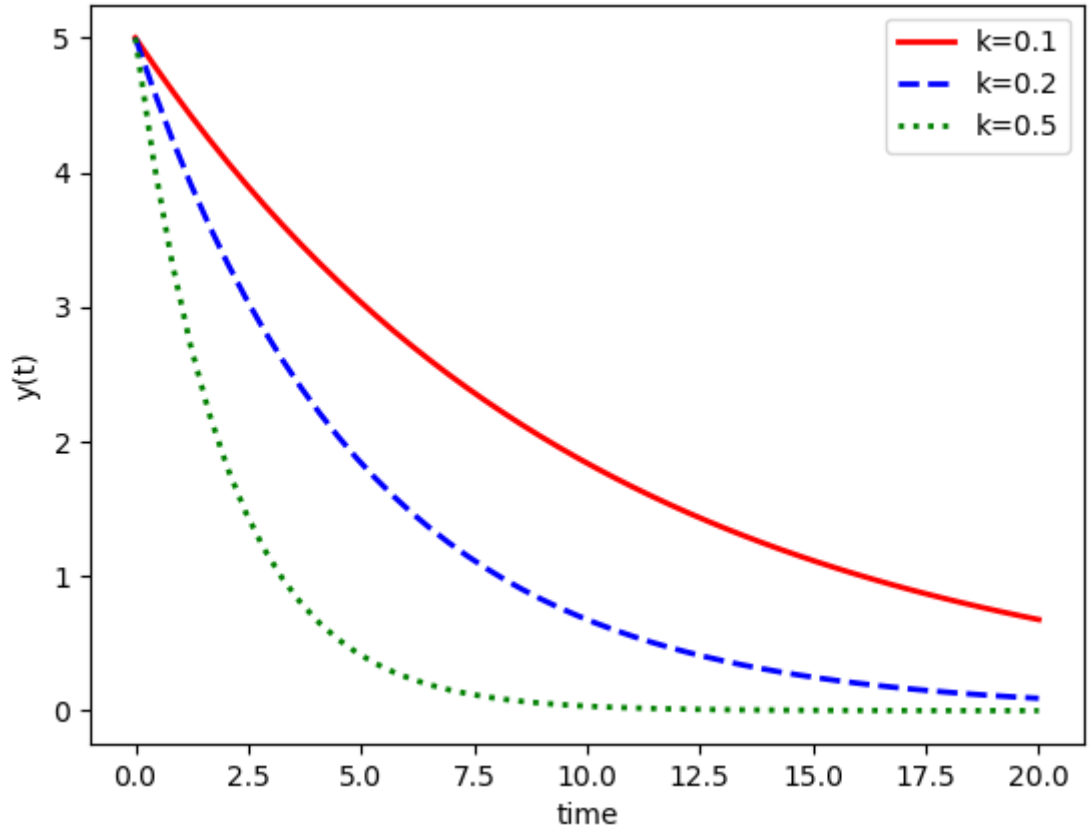
Passing Argument

Demo Program: odeint2.py

- An optional fourth input is args that allows additional information to be passed into the model function. The args input is a tuple sequence of values. The argument k is now an input to the model function by including an addition argument.

```python
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
# function that returns dy/dt
def model(y,t,k):
    dydt = -k * y
    return dydt
# initial condition
y0 = 5
# time points
t = np.linspace(0,20)
# solve ODEs
k = 0.1
y1 = odeint(model,y0,t,args=(k,))
k = 0.2
y2 = odeint(model,y0,t,args=(k,))
k = 0.5
y3 = odeint(model,y0,t,args=(k,))
# plot results
plt.plot(t,y1,'r-',linewidth=2,label='k=0.1')
plt.plot(t,y2,'b--',linewidth=2,label='k=0.2')
plt.plot(t,y3,'g:',linewidth=2,label='k=0.5')
plt.xlabel('time')
plt.ylabel('y(t)')
plt.legend()
plt.show()
```

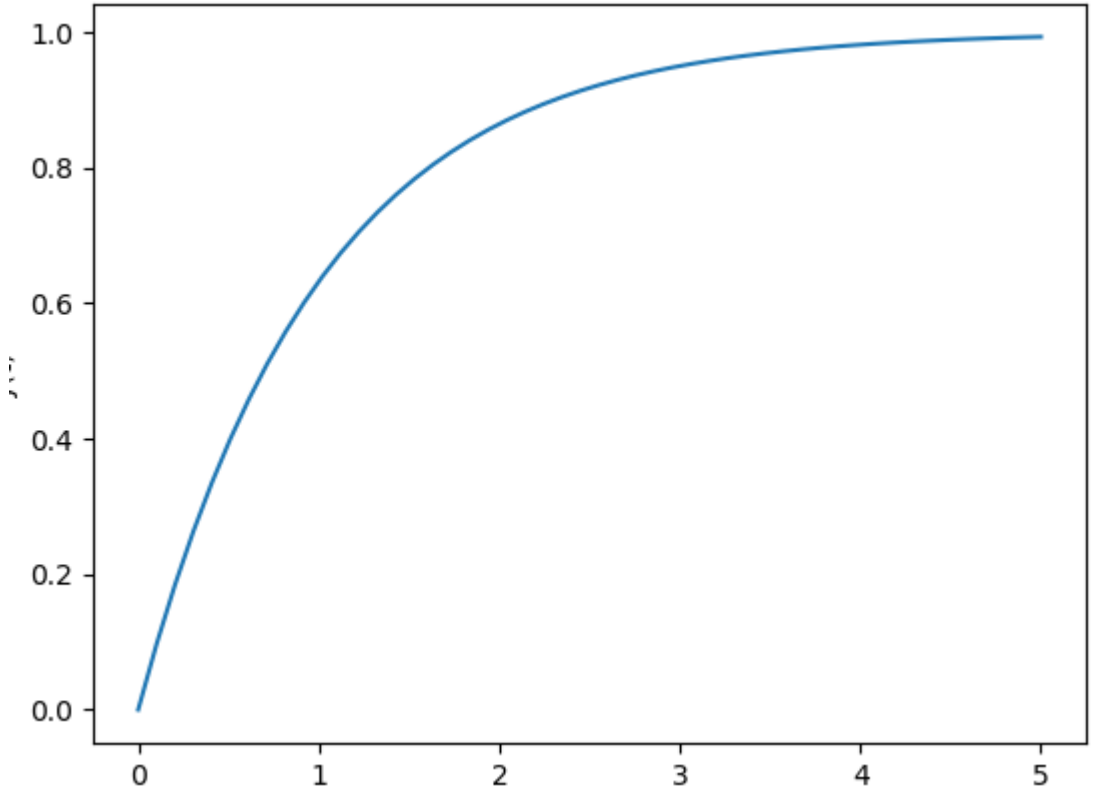
Output



Demo Program: odeint3.py

$$\frac{dy(t)}{dt} = -y(t) + 1$$

$$y(0) = 0$$



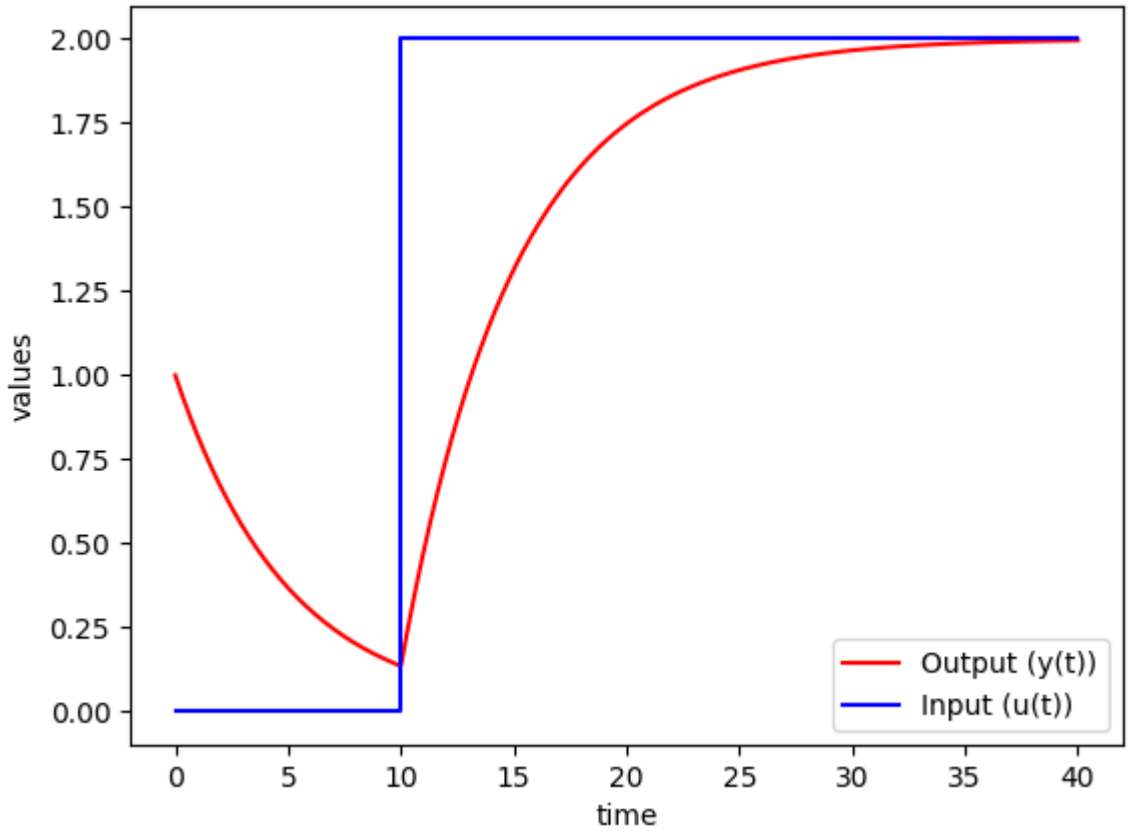
```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

# function that returns dy/dt
def model(y,t):
    dydt = -y + 1.0
    return dydt

# initial condition
y0 = 0
# time points
t = np.linspace(0,5)
# solve ODE
y = odeint(model,y0,t)

# plot results
plt.plot(t,y)
plt.xlabel('time')
plt.ylabel('y(t)')
plt.show()
```

Demo Program: odeint4.py

$$5\,\frac{dy(t)}{dt} = -y(t) + u(t)$$

$$y(0) = 1$$

u steps from *0* to *2* at $t = 10$



```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
# function that returns dy/dt
def model(y,t):
    # u steps from 0 to 2 at t=10
    if t<10.0: u = 0
    else: u = 2
    dydt = (-y + u)/5.0
    return dydt
# initial condition
y0 = 1
# time points
t = np.linspace(0,40,1000)
# solve ODE
y = odeint(model,y0,t)
# plot results
plt.plot(t,y,'r-',label='Output (y(t))')
plt.plot([0,10,10,40],[0,0,2,2],'b-',label='Input (u(t))')
plt.ylabel('values')
plt.xlabel('time')
plt.legend(loc='best')
plt.show()
```

Balance Equations

SECTION 4

Balance Equations

- In engineering, there are 4 common balance equations from conservation principles including mass, momentum, energy, and species. General forms of each equation are shown below with the accumulation term on the left and inlet (in), outlet (out), generation (gen), and consumption (cons) terms on the right side of the equation.

$$\textbf{Accumulation} = \textbf{In} - \textbf{Out} + \textbf{Generation} - \textbf{Consumption}$$

Mass Balance

$$\frac{dm}{dt} = \frac{d(\rho V)}{dt} = \sum \dot{m}_{in} - \sum \dot{m}_{out}$$

Species Balance

- A species balance tracks the number of moles n of species A in a control volume. The accumulation of A, d(nA)/dt, in a control volume is calculated by inlet, outlet, reaction generation, and reaction consumption rates.

$$\frac{dn_A}{dt} = \sum \dot{n}_{A_{in}} - \sum \dot{n}_{A_{out}} + \sum \dot{n}_{A_{gen}} - \sum \dot{n}_{A_{cons}}$$

Species Balance

- The molar amount, n_A is often measured as a concentration, c_A and reaction rates are often expressed in terms of a specific reaction rate, r_A, as a molar rate of generation per volume.

$$\frac{dc_A V}{dt} = \sum c_{A_{in}} \dot{V}_{in} - \sum c_{A_{out}} \dot{V}_{out} + r_A V$$

Momentum Balance

- A momentum balance is the accumulation of momentum for a control volume equal to the sum of forces *F* acting on that control volume.

$$\frac{d(m\,v)}{dt} = \sum F$$

- with *m* as the mass in the control volume and *v* as the velocity of the control volume.

Energy Balance

$$\frac{dE}{dt} = \frac{d(U+K+P)}{dt} = \sum \dot{m}_{in}\left(\hat{h}_{in} + \frac{v_{in}^2}{2g_c} + \frac{z_{in}g_{in}}{g_c}\right)$$

$$-\sum \dot{m}_{out}\left(\hat{h}_{out} + \frac{v_{out}^2}{2g_c} + \frac{z_{out}g_{out}}{g_c}\right) + Q + W_s$$

- Kinetic (K) and potential (P) energy terms are omitted because the internal energy (due to temperature) is typically a much larger contribution than any elevation (z) or velocity (v) changes of a fluid for most chemical processes.

Energy Balance

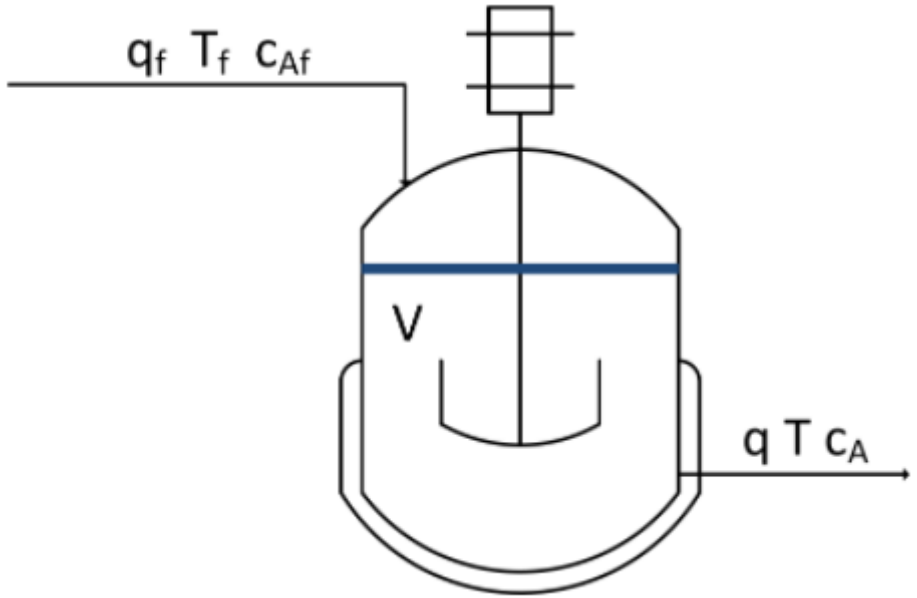
$$\frac{dh}{dt} = \sum \dot{m}_{in}\hat{h}_{in} - \sum \dot{m}_{in}\hat{h}_{out} + Q + W_s$$

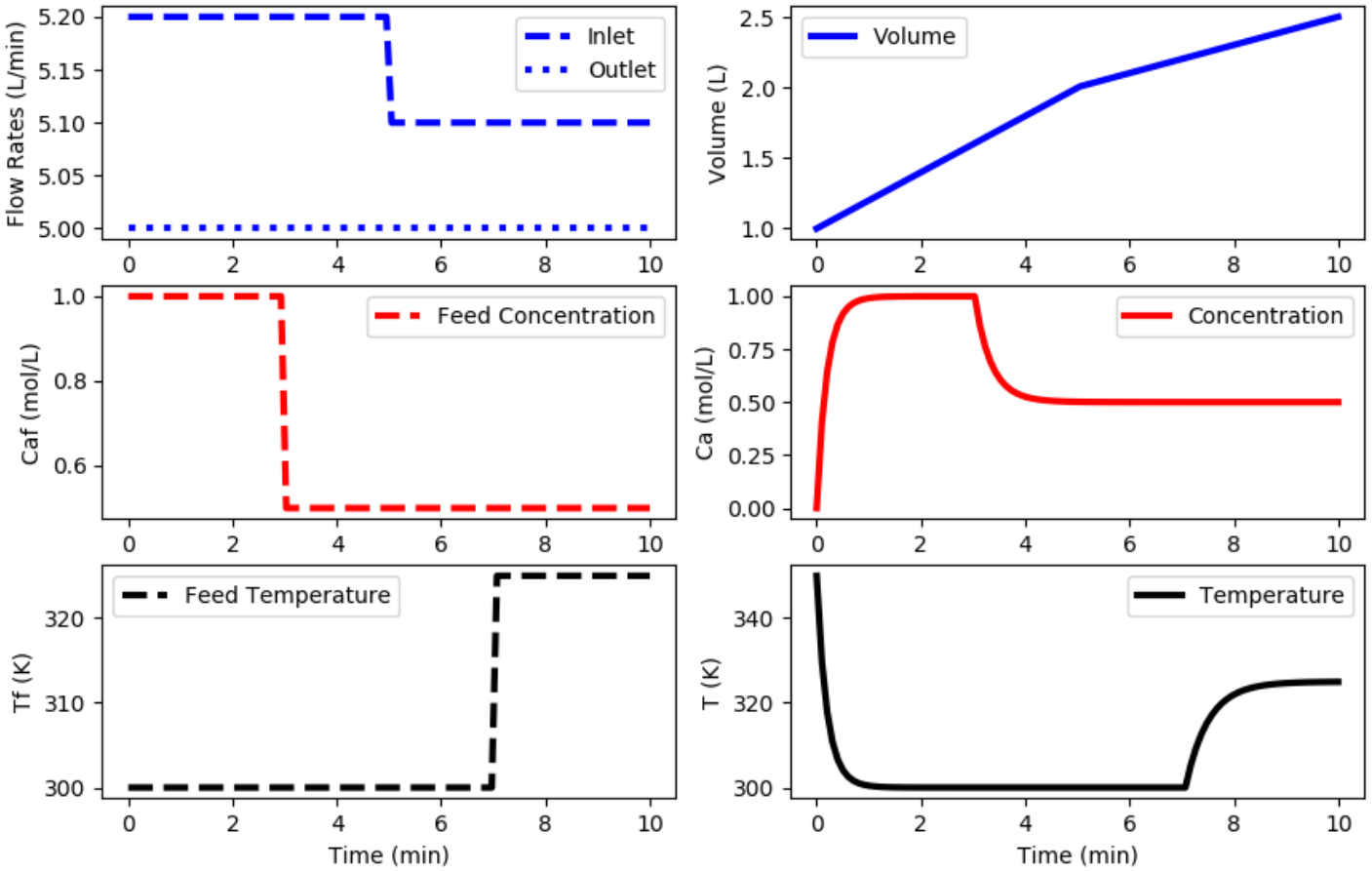
The enthalpy, *h*, is related to temperature as $m\ c_p\ (T\text{-}T_{ref})$ where c_p is the heat capacity. With a constant reference temperature (T_{ref}), this reduces to the following.

$$m\,c_p\frac{dT}{dt} = \sum \dot{m}_{in}c_p\left(T_{in} - T_{ref}\right) - \sum \dot{m}_{out}c_p\left(T_{out} - T_{ref}\right) + Q + W_s$$

Balance Equations

- Use a mass, species, and energy balance to describe the dynamic response in volume, concentration, and temperature of a well-mixed vessel.
- The inlet (q_f) and outlet (q) volumetric flowrates, feed concentration (C_{af}), and inlet temperature (T_f) can be adjusted. Initial conditions for the vessel are V= 1.0 L, Ca = 0.0 mol/L, and T=350 K. There is no reaction and no significant heat added by the mixer. There is a cooling jacket that can be used to adjust the outlet temperature. Show step changes in the process inputs.





Balance Equations

Demo Program: balance.py

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
# define mixing model
def vessel(x,t,q,qf,Caf,Tf):
    # Inputs (4):
    # qf  = Inlet Volumetric Flowrate (L/min)
    # q   = Outlet Volumetric Flowrate (L/min)
    # Caf = Feed Concentration (mol/L)
    # Tf  = Feed Temperature (K)
    # States (3):
    # Volume (L)
    V = x[0]
    # Concentration of A (mol/L)
    Ca = x[1]
    # Temperature (K)
    T = x[2]
    # Parameters:
    # Reaction
    rA = 0.0
    # Mass balance: volume derivative
    dVdt = qf - q
    # Species balance: concentration derivative
    # Chain rule: d(V*Ca)/dt = Ca * dV/dt + V * dCa/dt
    dCadt = (qf*Caf - q*Ca)/V - rA - (Ca*dVdt/V)
    # Energy balance: temperature derivative
    # Chain rule: d(V*T)/dt = T * dV/dt + V * dT/dt
    dTdt = (qf*Tf - q*T)/V - (T*dVdt/V)
    # Return derivatives
    return [dVdt,dCadt,dTdt]
```

```
# Initial Conditions for the States
V0 = 1.0
Ca0 = 0.0
T0 = 350.0
y0 = [V0,Ca0,T0]
# Time Interval (min)
t = np.linspace(0,10,100)
# Inlet Volumetric Flowrate (L/min)
qf = np.ones(len(t))* 5.2
qf[50:] = 5.1
# Outlet Volumetric Flowrate (L/min)
q = np.ones(len(t))*5.0
# Feed Concentration (mol/L)
Caf = np.ones(len(t))*1.0
Caf[30:] = 0.5
# Feed Temperature (K)
Tf = np.ones(len(t))*300.0
Tf[70:] = 325.0
# Storage for results
V  = np.ones(len(t))*V0
Ca = np.ones(len(t))*Ca0
T  = np.ones(len(t))*T0
```

```
# Loop through each time step
for i in range(len(t)-1):
    # Simulate
    inputs = (q[i],qf[i],Caf[i],Tf[i])
    ts = [t[i],t[i+1]]
    y = odeint(vessel,y0,ts,args=inputs)
    # Store results
    V[i+1]  = y[-1][0]
    Ca[i+1] = y[-1][1]
    T[i+1]  = y[-1][2]
    # Adjust initial condition for next loop
    y0 = y[-1]
# Construct results and save data file
data = np.vstack((t,qf,q,Tf,Caf,V,Ca,T)) # vertical
stack
data = data.T             # transpose data
np.savetxt('data.txt',data,delimiter=',')
```

```
# Plot the inputs and results
plt.figure()

plt.subplot(3,2,1)
plt.plot(t,qf,'b--',linewidth=3)
plt.plot(t,q,'b:',linewidth=3)
plt.ylabel('Flow Rates (L/min)')
plt.legend(['Inlet','Outlet'],loc='best')

plt.subplot(3,2,3)
plt.plot(t,Caf,'r--',linewidth=3)
plt.ylabel('Caf (mol/L)')
plt.legend(['Feed Concentration'],loc='best')

plt.subplot(3,2,5)
plt.plot(t,Tf,'k--',linewidth=3)
plt.ylabel('Tf (K)')
plt.legend(['Feed Temperature'],loc='best')
plt.xlabel('Time (min)')

plt.subplot(3,2,2)
plt.plot(t,V,'b-',linewidth=3)
plt.ylabel('Volume (L)')
plt.legend(['Volume'],loc='best')

plt.subplot(3,2,4)
plt.plot(t,Ca,'r-',linewidth=3)
plt.ylabel('Ca (mol/L)')
plt.legend(['Concentration'],loc='best')

plt.subplot(3,2,6)
plt.plot(t,T,'k-',linewidth=3)
plt.ylabel('T (K)')
plt.legend(['Temperature'],loc='best')
plt.xlabel('Time (min)')

plt.show()
```

Linearization of Differential Equations

SECTION 5

Linearization of Differential Equations

- Linearization is the process of taking the gradient of a nonlinear function with respect to all variables and creating a linear representation at that point. It is required for certain types of analysis such as stability analysis, solution with a Laplace transform, and to put the model into linear state-space form. Consider a nonlinear differential equation model that is derived from **balance equations** with input *u* and output *y*.

$$\frac{dy}{dt} = f(y, u)$$

Linearization of Differential Equations

- The right hand side of the equation is linearized by a Taylor series expansion, using only the first two terms.

$$\frac{dy}{dt} = f(y,u) \approx f(\bar{y},\bar{u}) + \left.\frac{\partial f}{\partial y}\right|_{\bar{y},\bar{u}} (y-\bar{y}) + \left.\frac{\partial f}{\partial u}\right|_{\bar{y},\bar{u}} (u-\bar{u})$$

Linearization of Differential Equations

If the values of $\bar{u}$ and $\bar{y}$ are chosen at steady state conditions then $f(\bar{y}, \bar{u}) = 0$ because the derivative term $\frac{dy}{du} = 0$ at steady state. To simplify the final linearized expression, deviation variables are defined as $y' = y - \bar{y}$ and $u' = u - \bar{u}$. A deviation variable is a change from the nominal steady state conditions. The derivatives of the deviation variable is defined as $\frac{dy'}{dt} = \frac{dy}{dt}$ because $\frac{d\bar{y}}{dt} = 0$ in $\frac{dy'}{dt} = \frac{d(y - \bar{y})}{dt} = \frac{dy}{dt} - \cancel{\frac{d\bar{y}}{dt}}$. If there are additional variables such as a disturbance variable d then it is added as another term in deviation variable form $d' = d - \bar{d}$.

$$\frac{dy'}{dt} = \alpha y' + \beta u' + \gamma d'$$

The values of the constants α, β, and γ are the partial derivatives of $f(y, u, d)$ evaluated at steady state conditions.

$$\alpha = \left.\frac{\partial f}{\partial y}\right|_{\bar{y},\bar{u},\bar{d}} \qquad \beta = \left.\frac{\partial f}{\partial u}\right|_{\bar{y},\bar{u},\bar{d}} \qquad \gamma = \left.\frac{\partial f}{\partial d}\right|_{\bar{y},\bar{u},\bar{d}}$$

Demo Program:

Part A: Linearize the following differential equation with an input value of *u*=16.

$$\frac{dx}{dt} = -x^2 + \sqrt{u}$$

Part B: Determine the steady state value of *x* from the input value and simplify the linearized differential equation.

Part C: Simulate a doublet test with the nonlinear and linear models and comment on the suitability of the linear model to represent the original nonlinear equation solution.

t A Solution: The equation is linearized by taking the partial derivative of
t hand side of the equation for both x and u.

$$\frac{\partial\left(-x^2+\sqrt{u}\right)}{\partial x}=\alpha=-2\,x$$

$$\frac{\partial\left(-x^2+\sqrt{u}\right)}{\partial u}=\beta=\frac{1}{2}\frac{1}{\sqrt{u}}$$

linearized differential equation that approximates $\frac{dx}{dt}=f(x,u)$ is the foll

$$\frac{dx}{dt}=f\left(x_{ss},u_{ss}\right)+\left.\frac{\partial f}{\partial x}\right|_{x_{ss},u_{ss}}\left(x-x_{ss}\right)+\left.\frac{\partial f}{\partial u}\right|_{x_{ss},u_{ss}}\left(u-u_{ss}\right)$$

stituting in the partial derivatives results in the following differential equat

$$\frac{dx}{dt}=0+\left(-2x_{ss}\right)\left(x-x_{ss}\right)+\left(\frac{1}{2}\frac{1}{\sqrt{u_{ss}}}\right)\left(u-u_{ss}\right)$$

is further simplified by defining new deviation variables as $x'=x-x_{ss}$ a
$=u-u_{ss}$.

$$\frac{dx'}{dt}=\alpha x'+\beta u'$$

Part A Solution:

Part B Solution: The steady state values are determined by setting $\frac{dx}{dt} = 0$ and solving for x.

$$0 = -x_{ss}^2 + \sqrt{u_{ss}}$$

$$x_{ss}^2 = \sqrt{16}$$

$$x_{ss} = 2$$

At steady state conditions, $\frac{dx}{dt} = 0$ so $f(x_{ss}, u_{ss}) = 0$ as well. Plugging in numeric values gives the simplified linear differential equation:

$$\frac{dx}{dt} = -4\,(x - 2) + \frac{1}{8}(u - 16)$$

The partial derivatives can also be obtained from Python, either symbolically with SymPy or else numerically with SciPy.

Part B Solution:

Symbolic Solution

Demo Program: linearB.py

```
# analytic solution with Python
import sympy as sp
sp.init_printing()
# define symbols
x,u = sp.symbols(['x','u'])
# define equation
dxdt = -x**2 + sp.sqrt(u)

print(sp.diff(dxdt,x))
print(sp.diff(dxdt,u))
```

```
-2*x
1/(2*sqrt(u))
```

Numerical Solution

Demo Program: linearB.py

```
# numeric solution with Python
import numpy as np
from scipy.misc import derivative
u = 16.0
x = 2.0
def pd_x(x):
    dxdt = -x**2 + np.sqrt(u)
    return dxdt
def pd_u(u):
    dxdt = -x**2 + np.sqrt(u)
    return dxdt

print('Approximate Partial Derivatives')
print(derivative(pd_x,x,dx=1e-4))
print(derivative(pd_u,u,dx=1e-4))

print('Exact Partial Derivatives')
print(-2.0*x) # exact d(f(x,u))/dx
print(0.5 / np.sqrt(u)) # exact d(f(x,u))/du
```

```
Approximate Partial Derivatives
-4.000000000004
0.12499999999970868
Exact Partial Derivatives
-4.0
0.125
```

Visualization

Demo Program: linearB2.py

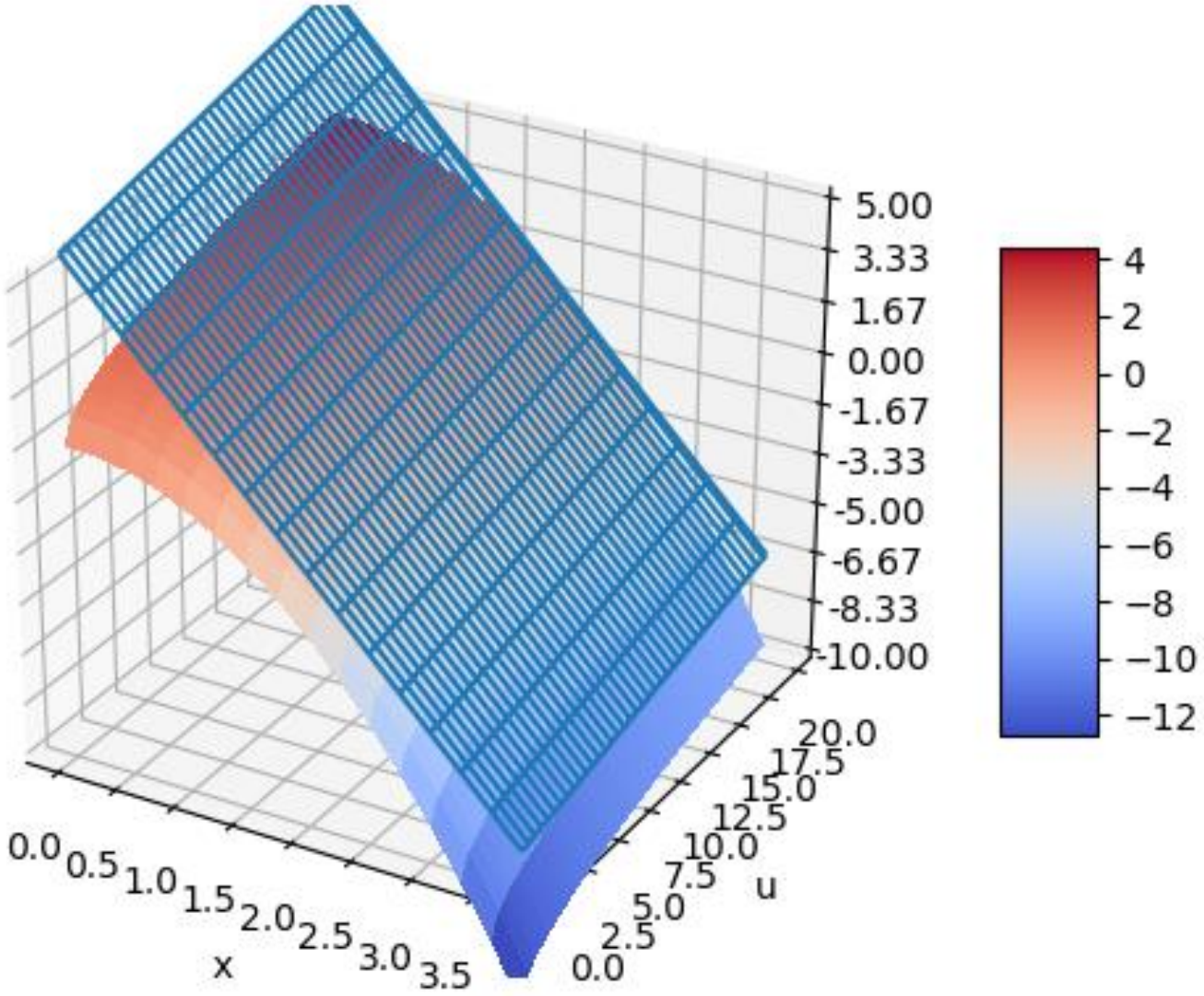
- The nonlinear function for $\frac{dx}{dt}$ can also be visualized with a 3D contour map. The choice of steady state conditions x_{ss} and u_{ss} produces a planar linear model that represents the nonlinear model only at a certain point.
- The linear model can deviate from the nonlinear model if used further away from the conditions at which the linear model is derived.

Visualization

Demo Program: linearB2.py

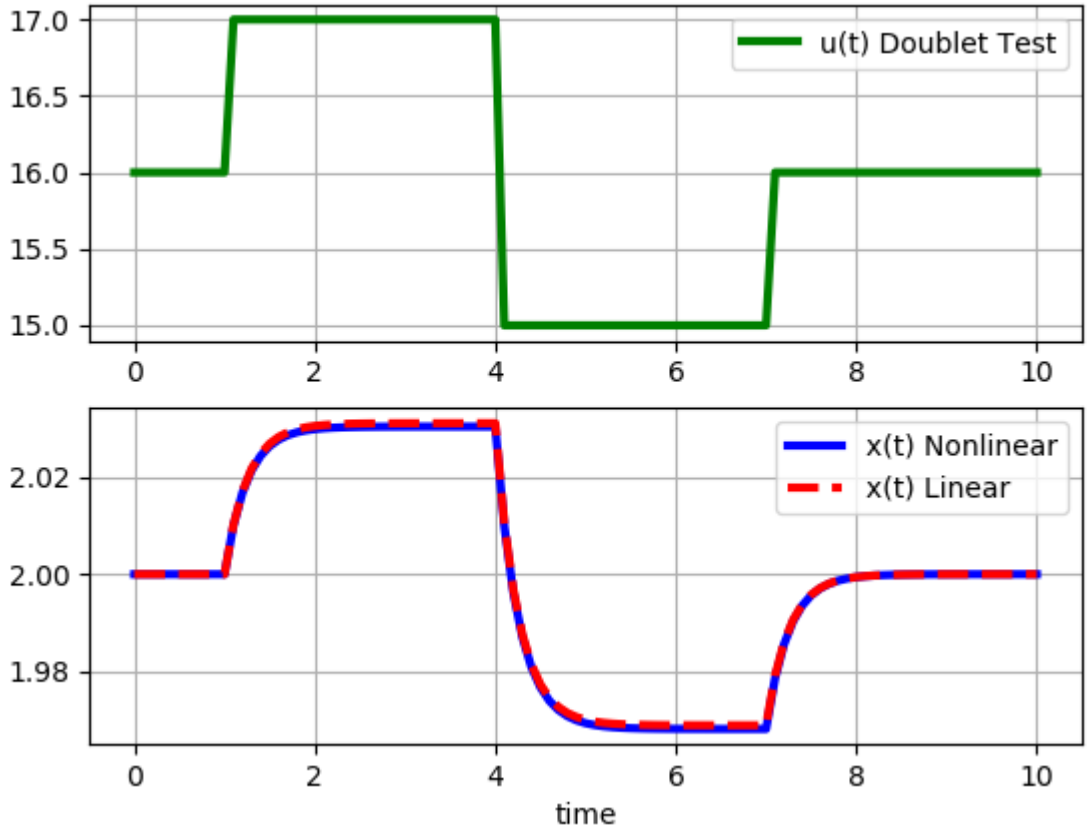
```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import numpy as np

fig = plt.figure()
ax = fig.gca(projection='3d')

# Make data.
X = np.arange(0, 4, 0.25)
U = np.arange(0, 20, 0.25)
X, U = np.meshgrid(X, U)
DXDT = -X**2 + np.sqrt(U)
LIN = -4.0 * (X-2.0) + 1.0/8.0 * (U-16.0)
```

```
# Plot the surface.
surf = ax.plot_wireframe(X, U, LIN)
surf = ax.plot_surface(X, U, DXDT, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)

# Customize the z axis.
ax.set_zlim(-10.0, 5.0)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))

# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)

# Add labels
plt.xlabel('x')
plt.ylabel('u')

plt.show()
```


5.00
3.33
1.67
0.00
-1.67
-3.33
-5.00
-6.67
-8.33
-10.00
4
2
0
−2
−4
−6
−8
−10
−12
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
x
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
u

Part C Solution: The final step is to simulate a doublet test with the nonlinear and linear models.

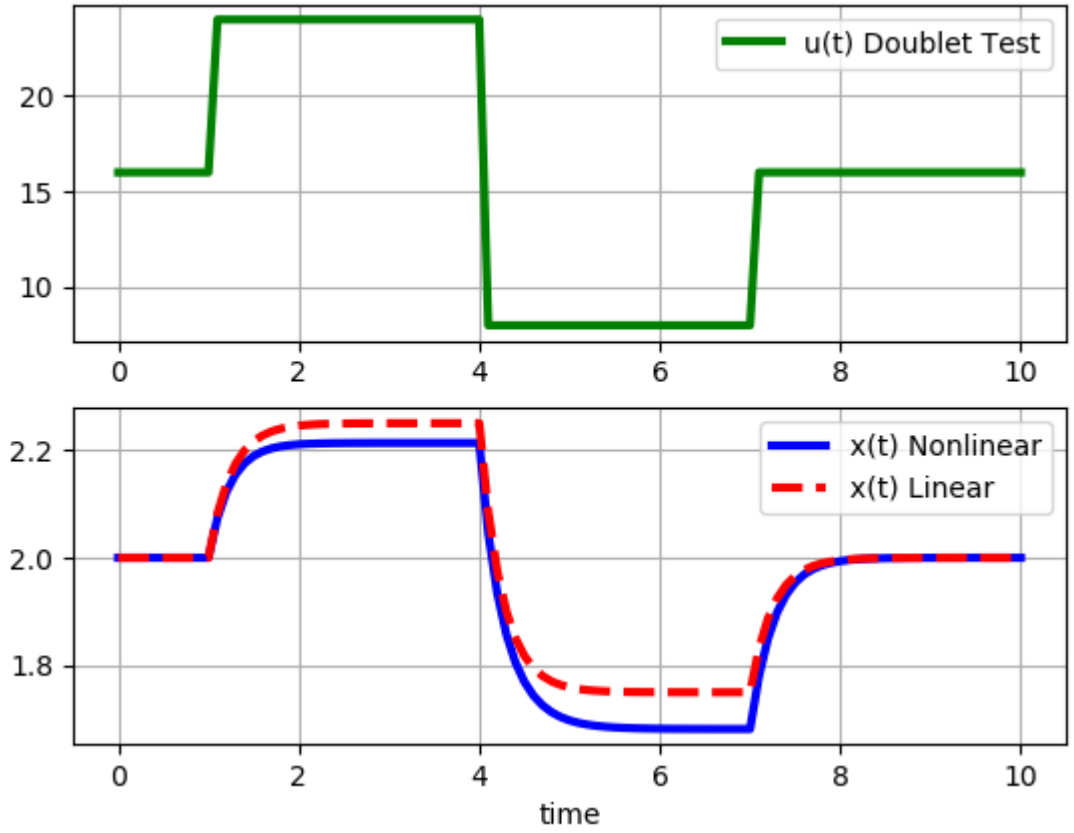
- ***Small step changes (+/-1)*:** Small step changes in *u* lead to nearly identical responses for the linear and nonlinear solutions. The linearized model is *locally* accurate.



Part C Solution:

Part C Solution: The final step is to simulate a doublet test with the nonlinear and linear models.

- **Large step changes (+/-8):** As the magnitude of the doublet steps increase, the linear model deviates further from the original nonlinear equation solution.



Part C Solution:

First Order Plus Dead Time (FOPDT)

SECTION 6

The First Order Plus Dead Time (FOPDT) model

- A first-order linear system with time delay is a common empirical description of many stable dynamic processes.
- The First Order Plus Dead Time (FOPDT) model is used to obtain initial controller tuning constants. An interactive FOPDT IPython Widget demonstrates the effect of the three adjustable parameters in the FOPDT equation. Downloading and running the iPython notebook locally reveals three slider bars that update the plot based on changes to the gain K_p, time constant τ_p, and dead time θ_p.

The First Order Plus Dead Time (FOPDT) model

- After gaining an intuitive understanding of how the parameters influence the step response, it is important to understand the mathematical FOPDT equation. The equation

$$\tau_p \frac{dy(t)}{dt} = -y(t) + K_p u(t - \theta_p)$$

has variables *y(t)* and *u(t)* and three unknown parameters.

K_p = Process gain

τ_p = Process time constant

θ_p = Process dead time

Process Gain, K_p

Given a change in $u(t) = \Delta u$, the solution to the linear first-order differential (without time delay) becomes:

$$y(t) = \left(e^{-t/\tau_p}\right) y(0) + \left(1 - e^{-t/\tau_p}\right) K_p \Delta u$$

If the initial condition $y(0)=0$ and at $t=\tau_p$, the solution is simplified to the following.

$$y(\tau_p) = \left(1 - e^{-\tau_p/\tau_p}\right) K_p \Delta u = \left(1 - e^{-1}\right) K_p \Delta u = 0.632 K_p \Delta u$$

Process Time Constant, τ_p

Given a change in $u(t) = \Delta u$, the solution to the linear first-order differential (without time delay) becomes:

$$y(t) = \left(e^{-t/\tau_p}\right) y(0) + \left(1 - e^{-t/\tau_p}\right) K_p \Delta u$$

If the initial condition $y(0)=0$ and at $t=\tau_p$, the solution is simplified to the following.

$$y\left(\tau_p\right) = \left(1 - e^{-\tau_p/\tau_p}\right) K_p \Delta u = \left(1 - e^{-1}\right) K_p \Delta u = 0.632 K_p \Delta u$$

Analytic Solution Derivation with Laplace Transforms

Start with the linear differential equation with time delay:

$$\tau_p \frac{dy(t)}{dt} = -y(t) + K_p u(t)$$

Perform a **Laplace transform from the tables** on each part of the equation:

$$\mathcal{L}\left(\tau_p \frac{dy(t)}{dt}\right) = \tau_p \left(sY(s) - y(0)\right)$$

$$\mathcal{L}\left(-y(t)\right) = -Y(s)$$

$$\mathcal{L}\left(K_p u(t)\right) = K_p U(s)$$

If the input $U(s)$ is a step function of size Δu then:

$$U(s) = \frac{\Delta u}{s}$$

Analytic Solution Derivation with Laplace Transforms

If the input $U(s)$ is a step function of size Δu then:

$$U(s) = \frac{\Delta u}{s}$$

Combining all of the individual Laplace transforms, the equation in Laplace domain with zero initial condition *y(0)=0* is then:

$$\tau_p \, s \, Y(s) = -Y(s) + K_p \frac{\Delta u}{s}$$

At this point, the equation is algebraic and can be solved for *Y(s)* by combining terms on each side:

$$\tau_p \, s \, Y(s) + Y(s) = K_p \frac{\Delta u}{s}$$

and factoring out the *Y(s)* term:

$$Y(s) \, (\tau_p \, s + 1) = K_p \frac{\Delta u}{s}$$

Analytic Solution Derivation with Laplace Transforms

A final steps are to isolate *Y(s)* on the left side of the equation and perform an inverse Laplace transform to return to the time domain:

$$Y(s) = K_p \frac{\Delta u}{s\ (\tau_p\, s + 1)}$$

$$\mathcal{L}^{-1}\left(Y(s)\right) = y(t) = K_p \left(1 - \exp\left(1 - \frac{(t)}{\tau_p}\right)\right) \Delta u$$

The process time constant is therefore the amount of time needed for the output to reach *(1-exp(-1))* or *63.2%* of the way to steady state conditions. The process time constant affects the speed of response.

Laplace Transforms

SECTION 7

Laplace Transform

Laplace transforms convert a function *f(t)* in the time domain into function in the Laplace domain *F(s)*.

$$F(s) = \mathcal{L}\left(f(t)\right) = \int_0^\infty f(t)e^{-s\,t}dt$$

As an example of the Laplace transform, consider a constant *c*. The function *f(t)* = *c* and the following expression is integrated.

$$\mathcal{L}(c) = \int_0^\infty c\,e^{-s\,t}dt = -\frac{c}{s}e^{-s\,t}\Big|_0^\infty = 0 - \left(-\frac{c}{s}\right) = \frac{c}{s}$$

Mathematicians have developed tables of commonly used Laplace transforms. Below is a summary table with a few of the entries that will be most common for analysis of linear differential equations in this course. Notice that the derived value for a constant *c* is the unit step function with *c=1* where a signal output changes from 0 to 1 at time=0.

f(t) in Time Domain	*F(s)* in Laplace Domain
$\delta(t)$ unit impulse	1
$S(t)$ unit step	$\frac{1}{s}$
t ramp with slope $= 1$	$\frac{1}{s^2}$
t^{n-1}	$\frac{(n-1)!}{s^n}$
e^{-bt}	$\frac{1}{s+b}$
$1-e^{-t/\tau}$	$\frac{1}{s(\tau s+1)}$
$\sin(\omega t)$	$\frac{\omega}{s^2+\omega^2}$
$\cos(\omega t)$	$\frac{s}{s^2+\omega^2}$
$\frac{1}{\tau_1-\tau_2}\left(\exp\left(-t/\tau_1\right)-\exp\left(-t/\tau_2\right)\right)$	$\frac{1}{(\tau_1 s+1)(\tau_2 s+1)}$

f(t) in Time Domain	*F(s)* in Laplace Domain
$\frac{1}{\tau^n(n-1)!}t^{n-1}\exp\left(-\frac{t}{\tau}\right)$	$\frac{1}{(\tau s+1)^n}$
$\frac{1}{\tau\sqrt{1-\zeta^2}}\exp\left(-\frac{\zeta t}{\tau}\right)\sin\left(\sqrt{1-\zeta^2}\frac{t}{\tau}\right)$	$\frac{1}{\tau^2 s^2+2\zeta\tau s+1}$
See **2nd Order Systems**	$\frac{1}{s(\tau^2 s^2+2\zeta\tau s+1)}$
$\frac{df}{dt}$	$sF(s)-f(0)$
$\frac{d^n f}{dt^n}$	$s^nF(s)-s^{n-1}f(0)-s^{n-2}f^{(1)}(0)-\ldots$ $-sf^{(n-2)}(0)-f^{(n-1)}(0)$
$\int f(t)$	$\frac{F(s)}{s}$
$f(t-t_0)\,S(t-t_0)$	$e^{-t_0 s}F(s)$

Final Value Theorem

The Final Value Theorem (FVT) gives the steady state signal value y_∞ for a stable system. Note that the Laplace variable s is multiplied by the signal $Y(s)$ before the limit is taken.

$$y_\infty = \lim_{s \to 0} s\, Y(s)$$

The FVT may give misleading results if applied to an unstable system. It is only applicable to stable systems or signals. There is more information on **how to determine system stability**.

Initial Value Theorem

The Initial Value Theorem (IVT) gives an initial condition of a signal by taking the limit as $s \to \infty$. Like the FVT, the Laplace variable s is multiplied by the signal $Y(s)$ before the limit is taken.

$$y_0 = \lim_{s \to \infty} s\,Y(s)$$

Laplace Transforms with Python

- Python Sympy is a package that has symbolic math functions.
- A few of the notable ones that are useful for this material are the Laplace transform (laplace_transform), inverse Laplace transform (inverse_laplace_transform), partial fraction expansion (apart), polynomial expansion (expand), and polynomial roots (roots).

```
import sympy as sym
from sympy.abc import s,t,x,y,z
from sympy.integrals import laplace_transform
from sympy.integrals import inverse_laplace_transform

# Laplace transform (t->s)
U = laplace_transform(5*t, t, s)
print('U')
print(U[0])
# Result: 5/s**2

# Inverse Laplace transform (s->t)
X = inverse_laplace_transform(U[0],s,t)
print('X')
print(X)
# Result: 5*t*Heaviside(t)

# Function
F = 5*(s+1)/(s+3)**2
print('F')
print(F)
# Result: (5*s + 5)/(s + 3)**2
```

```
# Partial fraction decomposition
G = sym.apart(F)
print('G')
print(G)
# Result: 5/(s + 3) - 10/(s + 3)**2

# denominator of transfer function
d1 = (s+1)*(s+3)*(s**2+3*s+1)

# expand polynomial
d2 = sym.expand(d1)
print('d2')
print(d2)
# Result: s**4 + 7*s**3 + 16*s**2 + 13*s + 3

# find roots
print(sym.roots(d2))
# Result: {-1: 1, -3: 1, -3/2 - sqrt(5)/2: 1, -3/2 + sqrt(5)/2: 1}
```

U

5/s**2

X

5*t*Heaviside(t)

F

(5*s + 5)/(s + 3)**2

G

5/(s + 3) - 10/(s + 3)**2

d2

s**4 + 7*s**3 + 16*s**2 + 13*s + 3

{-1: 1, -3: 1, -3/2 - sqrt(5)/2: 1, -3/2 + sqrt(5)/2: 1}

Process Time Delay

SECTION 8

Process Time Delay, θ_p

The time delay is expressed as a time shift in the input variable *u(t)*.

$$u\left(t-\theta_p\right)$$

Suppose that that input signal is a step function that normally changes from 0 to 1 at time=0 but this shift is delayed by *5 sec*. The input function *u(t)* and output function *y(t)* are time-shifted by *5 sec*. The solution to the first-order differential equation with time delay is obtained by replacing all variables t with $t-\theta_p$ and applying the conditional result based on the time in relation to the time delay, θ_p.

$$y(t<\theta_p)=y(0)$$

$$y(t\geq\theta_p)=\left(e^{-(t-\theta_p)/\tau_p}\right)y(0)+\left(1-e^{-(t-\theta_p)/\tau_p}\right)K_p\Delta u$$

Time Delay in Dynamic Systems

Time delay is a shift in the effect of an input on an output dynamic response. A first-order linear system with time delay is:

$$\tau_p \frac{dy(t)}{dt} = -y(t) + K_p u\left(t - \theta_p\right)$$

has variables *y(t)* and *u(t)* and three unknown parameters with

$$K_p \quad = \text{Process gain}$$

$$\tau_p \quad = \text{Process time constant}$$

$$\theta_p \quad = \text{Process dead time}$$

The time delay θ_p is expressed as a time shift in the input variable *u(t)*.

$$u\left(t - \theta_p\right)$$

$$y(t < \theta_p) = y(0)$$

$$y(t \geq \theta_p) = \left(e^{-(t-\theta_p)/\tau_p}\right) y(0) + \left(1 - e^{-(t-\theta_p)/\tau_p}\right) K_p \Delta u$$

Time Delay in Dynamic Systems

For a step change Δu, the analytical solution for a first-order linear system without time delay ($x(t) = y(t)$ with $\theta_p = 0$)

$$\tau_p \frac{dx(t)}{dt} = -y(t) + K_p u(t)$$

is

$$x(t) = K_p \left(1 - \exp\left(\frac{-t}{\tau_p}\right)\right) \Delta u$$

With dead-time, the solution becomes:

$$y(t) = x(t - \theta_p) S(t - \theta_p)$$

$$= K_p \left(1 - \exp\left(\frac{t - \theta_p}{\tau_p}\right)\right) \Delta u \, S(t - \theta_p)$$

where

$$S(t - \theta_p)$$

is a step function that changes from zero to one at $t = \theta_p$.

Analytic Solution Derivation with Laplace Transforms

Start with the linear differential equation with time delay:

$$\tau_p \frac{dy(t)}{dt} = -y(t) + K_p u\left(t - \theta_p\right)$$

Perform a **Laplace transform from the tables** on each part of the equation:

$$\mathcal{L}\left(\tau_p \frac{dy(t)}{dt}\right) = \tau_p \left(sY(s) - y(0)\right)$$

$$\mathcal{L}\left(-y(t)\right) = -Y(s)$$

$$\mathcal{L}\left(K_p u\left(t - \theta_p\right)\right) = K_p U(s) e^{-\theta_p s}$$

If the input $U(s)$ is a step function of size Δu then:

$$U(s) = \frac{\Delta u}{s}$$

Analytic Solution Derivation with Laplace Transforms

Combining all of the individual Laplace transforms, the equation in Laplace domain with zero initial condition *y(0)=0* is then:

$$\tau_p \, s \, Y(s) = -Y(s) + K_p \frac{\Delta u}{s} e^{-\theta_p s}$$

At this point, the equation is algebraic and can be solved for *Y(s)* by combining terms on each side:

$$\tau_p \, s \, Y(s) + Y(s) = K_p \frac{\Delta u}{s} e^{-\theta_p s}$$

and factoring out the *Y(s)* term:

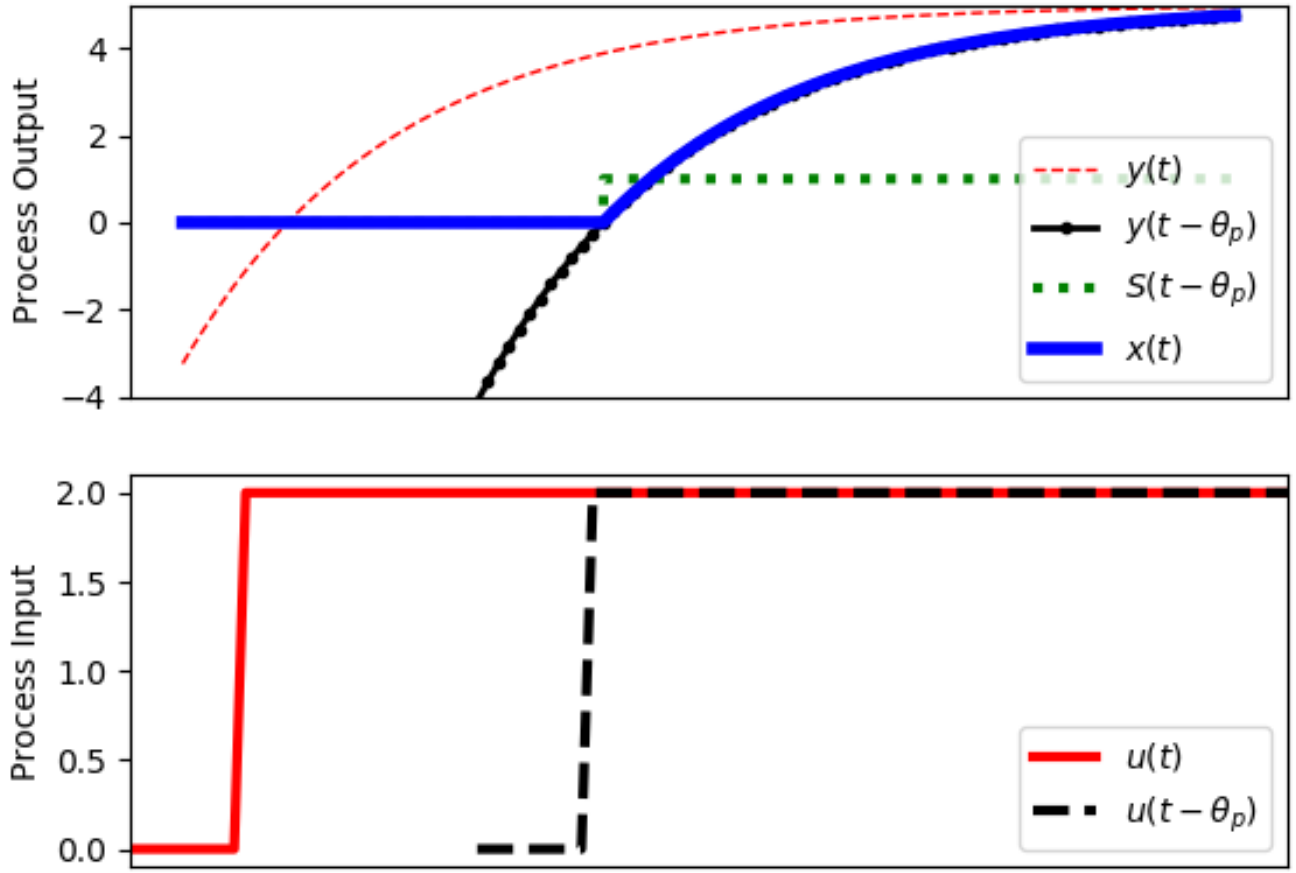
$$Y(s) \left(\tau_p \, s + 1 \right) = K_p \frac{\Delta u}{s} e^{-\theta_p s}$$

A final steps are to isolate *Y(s)* on the left side of the equation and perform an inverse Laplace transform to return to the time domain:

$$Y(s) = K_p \frac{\Delta u}{s \, \left(\tau_p \, s + 1 \right)} e^{-\theta_p s}$$

$$\mathcal{L}^{-1} \left(Y(s) \right) = y(t) = K_p \left(1 - \exp\left(1 - \frac{(t - \theta_p)}{\tau_p} \right) \right) \, \Delta u \, S \left(t - \theta_p \right)$$

Demo Program: timedelay.py



Transfer Functions

SECTION 9

Transfer Function

- Transfer functions are input to output representations of dynamic systems. One advantage of working in the Laplace domain (versus the time domain) is that differential equations become algebraic equations. These algebraic equations can be rearranged and transformed back into the time domain to obtain a solution or further combined with other transfer functions to create more complicated systems.
- The first step in creating a transfer function is to convert each term of a differential equation with a Laplace transform as shown in the **table of Laplace transforms**. A transfer function, *G(s)*, relates an input, *U(s)*, to an output, *Y(s)*.

$$G(s) = \frac{Y(s)}{U(s)}$$

First-order Transfer Function

- A **first-order linear differential equation** is shown as a function of time.

$$\tau_p \frac{dy(t)}{dt} = -y(t) + K_p u(t - \theta_p)$$

- The first step is to apply the **Laplace transform** to each of the terms in the differential equation.

$$L(\tau_p \frac{dy(t)}{dt}) = L(-y(t)) + L(K_p u(t - \theta_p))$$

First-order Transfer Function

- Because the Laplace transform is a linear operator, each term can be transformed separately. With a zero initial condition the value of *y* is zero at the initial time or *y(0)=0*.

$$L(\tau_p \frac{dy(t)}{dt}) = \tau_p\big(sY(s) - y(0)\big) = \tau_p sY(s)$$

$$L\big(-y(t)\big) = -Y(s)$$

$$L\left(K_p u\big(t - \theta_p\big)\right) = \ K_p U(s) e^{-\theta_p s}$$

- Putting these terms together gives the first-order differential equation in the Laplace domain.

$$\tau_p sY(s) = -Y(s) + K_p U(s) e^{-\theta_p s}$$

First-order Transfer Function

- For the first-order linear system, the transfer function is created by isolating terms with *Y(s)* on the left side of the equation and the term with *U(s)* on the right side of the equation.

$$\tau_p s Y(s) + Y(s) = K_p U(s) e^{-\theta_p s}$$

- Factoring out the *Y(s)* and dividing through gives the final transfer function.

$$G(s) = \frac{Y(s)}{U(s)} = \frac{K_p e^{-\theta_p s}}{\tau_p s + 1}$$

Transfer Function Gain

The Final Value Theorem (FVT) gives the steady state gain K_p of a transfer function G(s) by taking the limit as s→0

$$K_p = \lim_{s \to 0} G(s)$$

- This is related to the final value theorem by considering the output response Y(s) when the input is a unit step $U(s) = \frac{1}{s}$.

$$K_p = \frac{\Delta y}{\Delta u} = \lim_{s \to 0} \frac{Y(s)}{U(s)} = \lim_{s \to 0} \frac{Y(s)}{\frac{1}{s}} = \lim_{s \to 0} sY(s)$$

Transfer Function Gain

- In deviation variable form, the initial condition for u and y is zero.

$$\frac{y_\infty - 0}{1 - 0} = y_\infty = \lim_{s \to 0} \frac{Y(s)}{\frac{1}{s}} = \lim_{s \to 0} sY(s)$$

- The FVT also determines the final signal value y∞ for a stable system with output Y(s). The Laplace variable s is multiplied by the signal Y(s) before the limit is taken, unlike when calculating the gain.

$$y_\infty = \lim_{s \to 0} sY(s)$$

- The FVT may give misleading results if applied to an unstable system. It is only applicable to stable systems. There is additional information on how to **determine if a system is stable**.

PID Equation in Laplace Domain

- The **PID equation** is a block in an overall control loop diagram.

$$u(t) = u_{bias} + K_c e(t) + \frac{K_c}{\tau_I}\int_0^t e(t)dt - K_c\tau_D\frac{d(PV)}{dt}$$

- The PID equation can be converted to a transfer function by performing a Laplace transform on each of the elements. The controller output $u(t)$ is combined with the u_{bias} to create a deviation variable $u'(t) = u(t)\text{-}u_{bias}$.

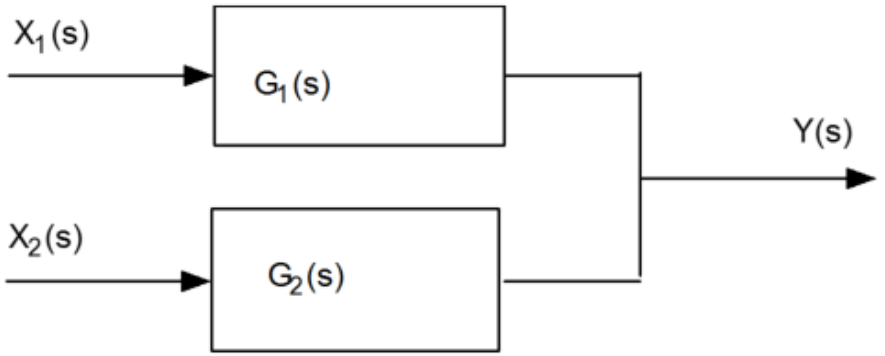
$$U(s) = K_c E(s) + \frac{K_c}{\tau_I\ s}E(s) - K_c\tau_D PV(s)$$

- If τ_D=0 for a PI controller, the transfer function for the controller is simplified.

$$G(s) = \frac{U(s)}{E(s)} = K_c + \frac{K_c}{\tau_I\ s} = K_c\frac{(\tau_I\ s\ + 1)}{\tau_I\ s}$$

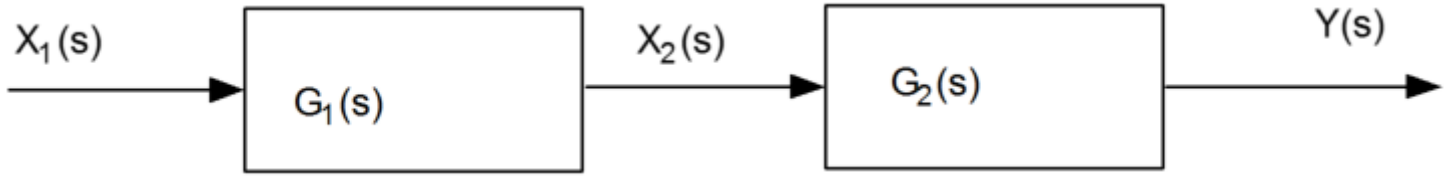
Combining Transfer Functions

- The additive property is used for transfer functions in parallel. The input signal $X_1(s)$ becomes $Y_1(s)$ when it is transformed by $G_1(s)$. Likewise, $X_2(s)$ becomes $Y_2(s)$ when it is transformed by $G_2(s)$. The two signals $Y_1(s)$ and $Y_2(s)$ are added to create the final output signal $Y(s)=Y_1(s)+Y_2(s)$. This gives a final output expression of $Y(s)=G_1(s)X_1(s)+G_2(s)X_2(s)$



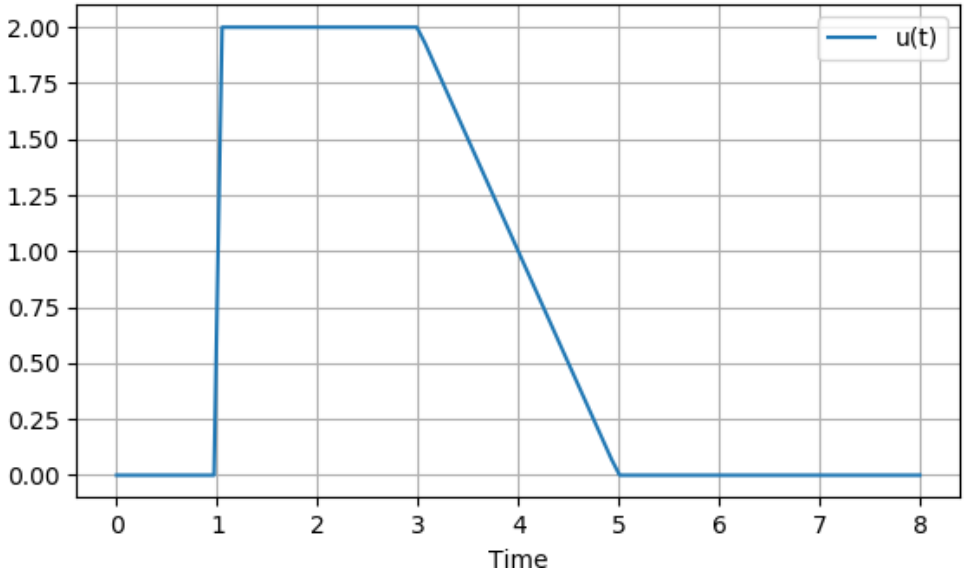
Combining Transfer Functions

- The multiplicative property is used for transfer functions in series. The input signal $X_1(s)$ becomes $X_2(s)$ when it is transformed by $G_1(s)$. The intermediate signal $X_2(s)$ becomes the input for the second transfer function $G_2(s)$ to produce $Y(s)$. The final output signal is $Y(s)=G_2(s)X_2(s)=G_1(s)G_2(s)X_1(s)$.



Transfer Functions with Python

- Python SymPy computes symbolic solutions to many mathematical problems including Laplace transforms. A symbolic and numeric solution is created with the following example problem.
- Compute the analytic and numeric system response to an input that includes a step and ramp function.



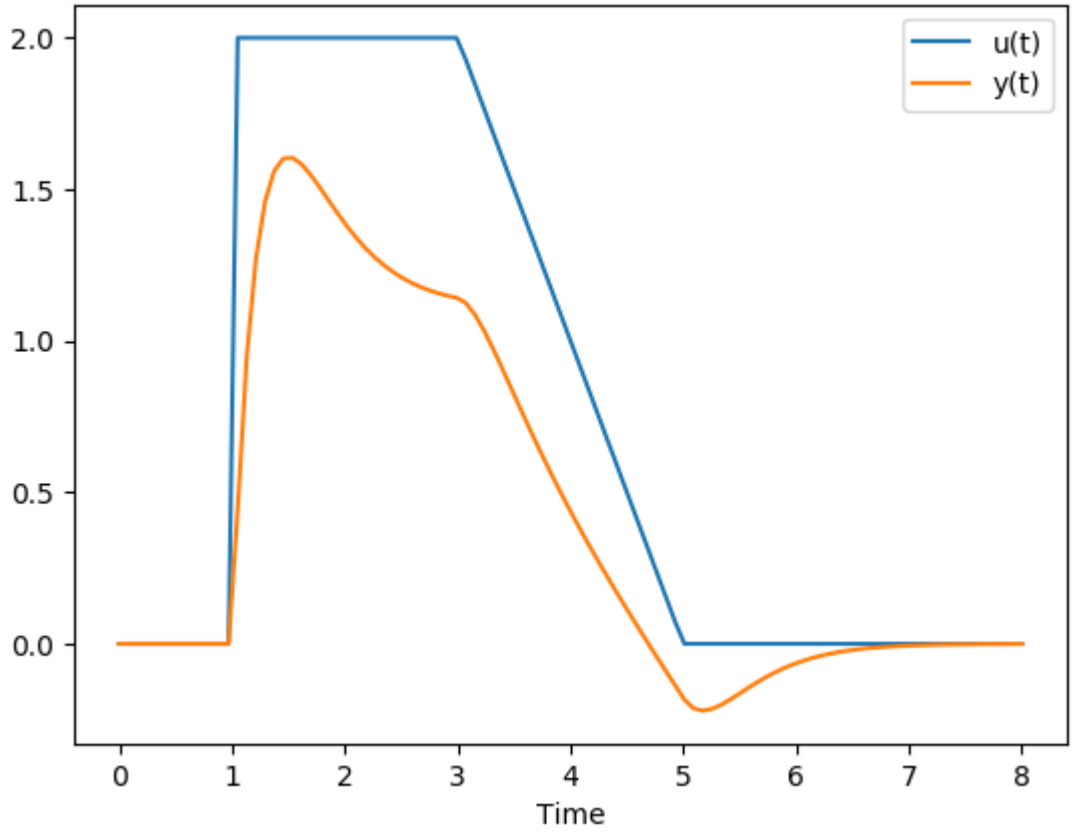
Transfer Functions with Python

- The system transfer function is a stable system with two poles (denominator roots) and one zero (numerator root):

$$G(s) = \frac{5\,(s+1)}{(s+3)^2}$$

Symbolic Solution (Python Sympy)

Demo Program: gain.py



```
# Define inputs
# First step (up) starts at 1 sec
U1 = 2/s*sym.exp(-s)
# Ramp (down) starts at 3 sec
U2 = -1/s**2*sym.exp(-3*s)
# Ramp completes at 5 sec
U3 = 1/s**2*sym.exp(-5*s)

# Transfer function
G = 5*(s+1)/(s+3)**2

# Calculate responses
Y1 = G * U1
Y2 = G * U2
Y3 = G * U3

# Inverse Laplace Transform
u1 = inverse_laplace_transform(U1,s,t)
u2 = inverse_laplace_transform(U2,s,t)
u3 = inverse_laplace_transform(U3,s,t)
y1 = inverse_laplace_transform(Y1,s,t)
y2 = inverse_laplace_transform(Y2,s,t)
y3 = inverse_laplace_transform(Y3,s,t)
print('y1')
print(y1)
```

```
# generate data for plot
tm = np.linspace(0,8,100)
us = np.zeros(len(tm))
ys = np.zeros(len(tm))

# substitute numeric values for u and y
for u in [u1,u2,u3]:
    for i in range(len(tm)):
        us[i] += u.subs(t,tm[i])
for y in [y1,y2,y3]:
    for i in range(len(tm)):
        ys[i] += y.subs(t,tm[i])

# plot results
plt.figure()
plt.plot(tm,us,label='u(t)')
plt.plot(tm,ys,label='y(t)')
plt.legend()
plt.xlabel('Time')
plt.show()
```

```
y1
-10*(-exp(3*t) + 6*exp(3)*log(exp(-t)) + 7*exp(3))*exp(-
3*t)*Heaviside(t - 1)/9
```

Numeric Solution (Python GEKKO)

- An alternative to a symbolic solution is to numerically compute the response in the time domain. The transfer function must first be translated into a differential equation.

$$G(s) = \frac{Y(s)}{U(s)} = \frac{5(s+1)}{(s+3)^2}$$

$$Y(s)s(s+3)^2 = 5(s+1)U(s)$$

$$Y(s)s(s^2+6s+9) = (5s+5)U(s)$$

$$\frac{dy^2(t)}{dt^2} + 6\frac{dy(t)}{dt} + 9\,y(t) = 5\frac{du(t)}{dt} + 5\,u(t)$$

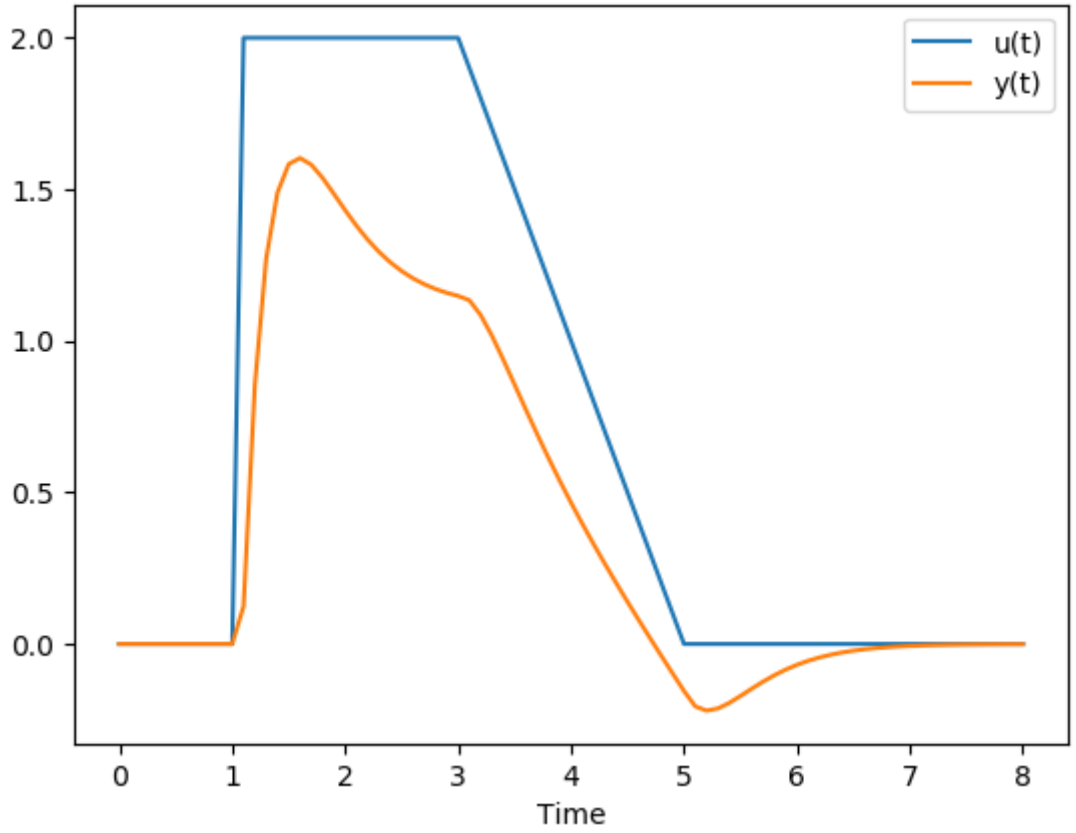
- There is additional information on solving differential equations with **Python GEKKO** or with **Python Scipy ODEINT**.

Installation of Gekko

pip install gekko

Numeric Solution (Python GEKKO)

Demo Program: gain_numerical.py



```
from gekko import GEKKO
import numpy as np
import matplotlib.pyplot as plt

# Create GEKKO model
m = GEKKO()

# Time points for simulation
nt = 81
m.time = np.linspace(0,8,nt)

# Define input
# First step (up) starts at 1 sec
# Ramp (down) starts at 3 sec
# Ramp completes at 5 sec
ut = np.zeros(nt)
ut[11:31] = 2.0
for i in range(31,51):
    ut[i] = ut[i-1] - 0.1

# Define model
u = m.Param(value=ut)
ud = m.Var()
y = m.Var()
dydt = m.Var()
m.Equation(ud==u)
m.Equation(dydt==y.dt())
m.Equation(dydt.dt() + 6*y.dt() + 9*y==5*ud.dt()+5*u)

# Simulation options
m.options.IMODE=7
m.options.NODES=4
m.solve(disp=False)

# plot results
plt.figure()
plt.plot(m.time,u.value,label='u(t)')
plt.plot(m.time,y.value,label='y(t)')
plt.legend()
plt.xlabel('Time')
plt.show()
```

Solution (Symbolic and Numeric)

- The advantage of an symbolic (analytic) solution is that it is highly accurate and does not rely on numerical methods to approximate the solution. Also, the solution is in a compact form that can be used for further analysis. Symbolic solutions are limited to cases where the input function and system transfer function can be expressed in Laplace form.
- This may not be the case for inputs that come from data sources where there the input function has random variation. A symbolic solution with Laplace transforms is also not possible for systems that are nonlinear or complex while numeric solvers can handle many thousands or millions of equations with nonlinear relationships.

Solution (Symbolic and Numeric)

- The disadvantage of a numeric solution is that it is an approximation of the true solution with possible inaccuracies. Another disadvantage is that solvers may fail to converge although this is not typical on problems with an analytic solution.

Stability Analysis

SECTION 10

Controller Stability Analysis

- The purpose of controller stability analysis is to determine the range of controller gains between lower K_{cL} and upper K_{cU} limits that lead to a stable controller.

$$K_{cL} \leq K_c \leq K_{cU}$$

- The principles of stability analysis presented here are general for any linear time-invariant system whether it is for controller design or for analysis of system dynamics. Several characteristics of a system in the Laplace domain can be deduced without transforming a system signal or transfer function back into the time domain. Some of the analysis relies on the roots of the transfer function denominator, also known as poles. The roots of the numerator, also known as zeros, do not affect the stability directly but can potentially cancel an unstable pole to create an overall stable system.

Converge or Diverge

- A first point of analysis is whether the system converges or diverges. This is determined by analyzing the roots of the denominator of the transfer function. If any of the real parts of the roots of the denominator are positive then the system is unstable.
- A simple rule to determine whether there are positive real roots is to examine the signs of the polynomial. If there are mixed signs (+ or -) then the system will be unstable because there is at least one positive real root.

Converge or Diverge

- Before modern computational methods, there were several methods devised to determine the stability of a system. One such approach is the **Routh-Hurwitz stability criterion**. The leading left edge of a table determines whether the system is stable for or any *n*th-degree polynomial

$$a_n s^n + a_{n-1} s^{n-1} + \cdots + a_1 s + a_0$$

Converge or Diverge

The coefficients of the polynomial are placed into tabular form and additional coefficients b and c are computed from higher rows.

a_n	a_{n-2}	a_{n-4}	$\cdots$
a_{n-1}	a_{n-3}	a_{n-5}	$\cdots$
$b_1 = \frac{a_{n-1}a_{n-2} - a_n a_{n-3}}{a_{n-1}}$	$b_2 = \frac{a_{n-1}a_{n-4} - a_n a_{n-5}}{a_{n-1}}$	b_3	$\cdots$
$c_1 = \frac{b_1 a_{n-3} - a_{n-1} b_2}{b_1}$	$c_2 = \frac{b_1 a_{n-5} - a_{n-1} b_3}{b_1}$	c_3	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$

Converge or Diverge

- The terms b and c are:

$$b_i = \frac{a_{n-1} \times a_{n-2i} - a_n \times a_{n-2i-1}}{a_{n-1}}$$

$$c_i = \frac{b_1 \times a_{n-2i-1} - a_{n-1} \times b_{i+1}}{b_1}$$

- A changing sign (+ or -) of the leading left edge a_n, a_{n-1}, b_1, c_1 indicates that the system is unstable.

Converge or Diverge

- Several additional methods can be used to determine stability as summarized below.

Method	Stability Condition
Routh Array	Leading left edge terms do not change sign
Root Locus Plot	All poles (roots of denominator) have negative real parts
Bode Plot	The response magnitude at -180 deg phase is less than one
Simulation	Simulate a closed loop response with increasing or decreasing controller gains until instability occurs

Converge or Diverge

- In addition to analysis in the Laplace domain, stability can be determined from a model in **state space form**.

$$\dot{x} = A\,x + Bu$$

$$y = C\,x + Du$$

- A state space model is stable when the eigenvalues of the A matrix have negative real parts.

Oscillatory or Smooth

- A second point of analysis is whether the system exhibits oscillatory or smooth behavior. If any of the roots of the denominator have an imaginary component then the system has oscillations. Imaginary roots always come in pairs with the same positive and negative imaginary values.

Final Value Theorem

- The Final Value Theorem (FVT) gives the steady state gain K_p of a transfer function $G(s)$ by taking the limit as $s \rightarrow 0$

$$K_p = \lim_{s \to 0} G(s)$$

- The FVT also determines the final signal value y_∞ for a stable system with output $Y(s)$. Note that the Laplace variable *s* is multiplied by the signal $Y(s)$ before the limit is taken.

$$y_\infty = \lim_{s \to 0} sY(s)$$

- The FVT may give misleading results if applied to an unstable system. It is only applicable to stable systems or signals.

Initial Value Theorem

- The Initial Value Theorem (IVT) gives an initial condition of a signal by taking the limit as $s \rightarrow \infty$. Like the FVT, the Laplace variable s is multiplied by the signal $Y(s)$ before the limit is taken.

$$y_0 = \lim_{s \to \infty} sY(s)$$

Controller Stability

- Controller stability analysis is finding the range of controller gains that lead to a stabilizing controller. There are multiple methods to compute this range between a lower limit K_{cL} and an upper limit K_{cU}.

$$K_{cL} \leq K_c \leq K_{cU}$$

- This range is important to know the range of tuning values that will not lead to a destabilizing controller. With modern computational tools and powerful computers, the simulation based option is frequently used for complex systems.

Cast Study: Stability Analysis

SECTION 11

Case Study: Open Loop Transfer Function

- Consider a feedback control system that has the following open loop transfer function.

$$G(s) = \frac{4\ K_c}{(s+1)(s+2)(s+3)}$$

- Determine the values of K_c that keep the closed loop system response stable.

Routh Array

$a_n = 1$	$a_{n-2} = 11$
$a_{n-1} = 6$	$a_{n-3} = 6 + 4K_c$
$b_1 = \dfrac{66 - 6 - 4K_c}{6}$	$b_2 = 0$
$c_1 = 6 + 4K_c$	0

Routh Array

- The leading edge cannot change signs for the system to be stable. Therefore, the following conditions must be met:

$$a_n = 1 > 0$$

$$a_n - 1 = 6 > 0$$

$$b_1 = \frac{66-6-4\,K_c}{6} > 0$$

$$c_1 = 6 + 4\,K_c > 0$$

Routh Array

- The positive constraint on b_1 leads to $K_c<15$. The positive constraint on c1c1 means that $K_c > -1.5$. Therefore, the following ranges are acceptable for the controller stability.

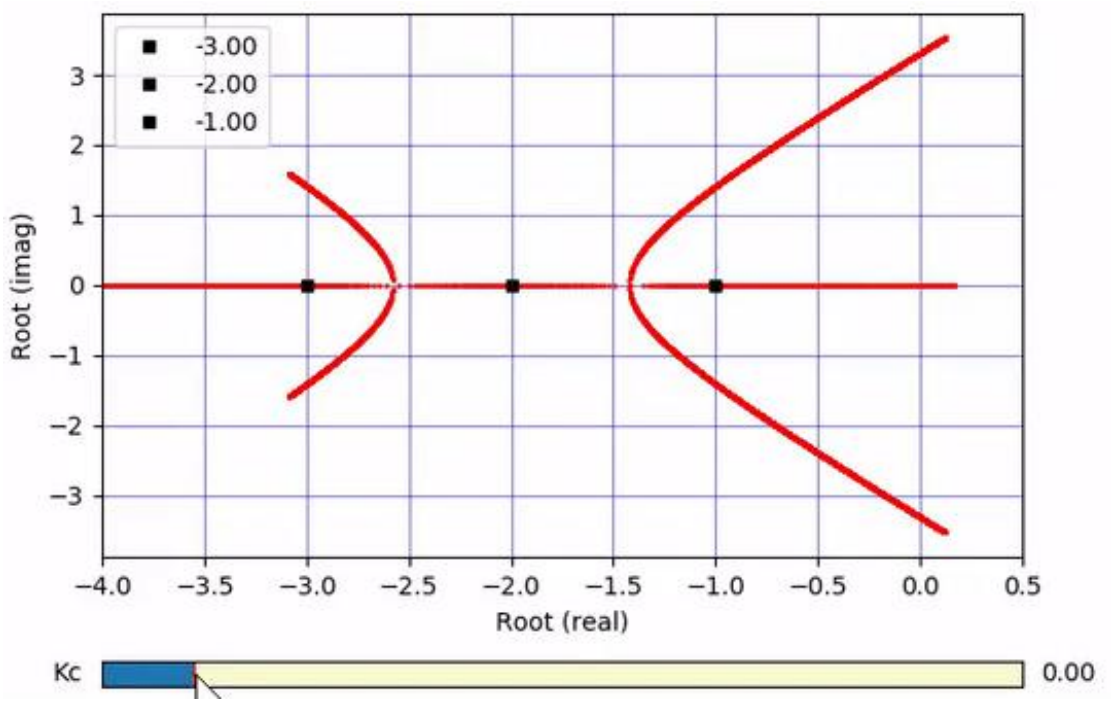
$$-1.5<K_c<15$$

- This is a more comprehensive solution than the other methods shown below because it also includes a lower bound on the controller stability limit (if a direct acting controller were inadvertently used).

Root Locus Plot

Demo Program: stable.py

- Determine where the real portion of the roots crosses to the right hand side of the plane. In this case, the real part of two roots becomes positive at $K_c = 15$.



```
# open loop
num = [4.0]
den = [1.0,6.0,11.0,6.0]
def dcl(K):
    return [1.0,6.0,11.0,4.0*K+6.0]

# root locus plot
n = 10000 # number of points to plot
nr = len(den)-1 # number of roots
rs = np.zeros((n,2*nr))   # store results
Kc1 = -2.0
Kc2 = 18.0
Kc = np.linspace(Kc1,Kc2,n)  # Kc values
for i in range(n):          # cycle through n times
    roots = np.roots(dcl(Kc[i]))
    for j in range(nr):   # store roots
        rs[i,j] = roots[j].real # store real
        rs[i,j+nr] = roots[j].imag # store imaginary
```

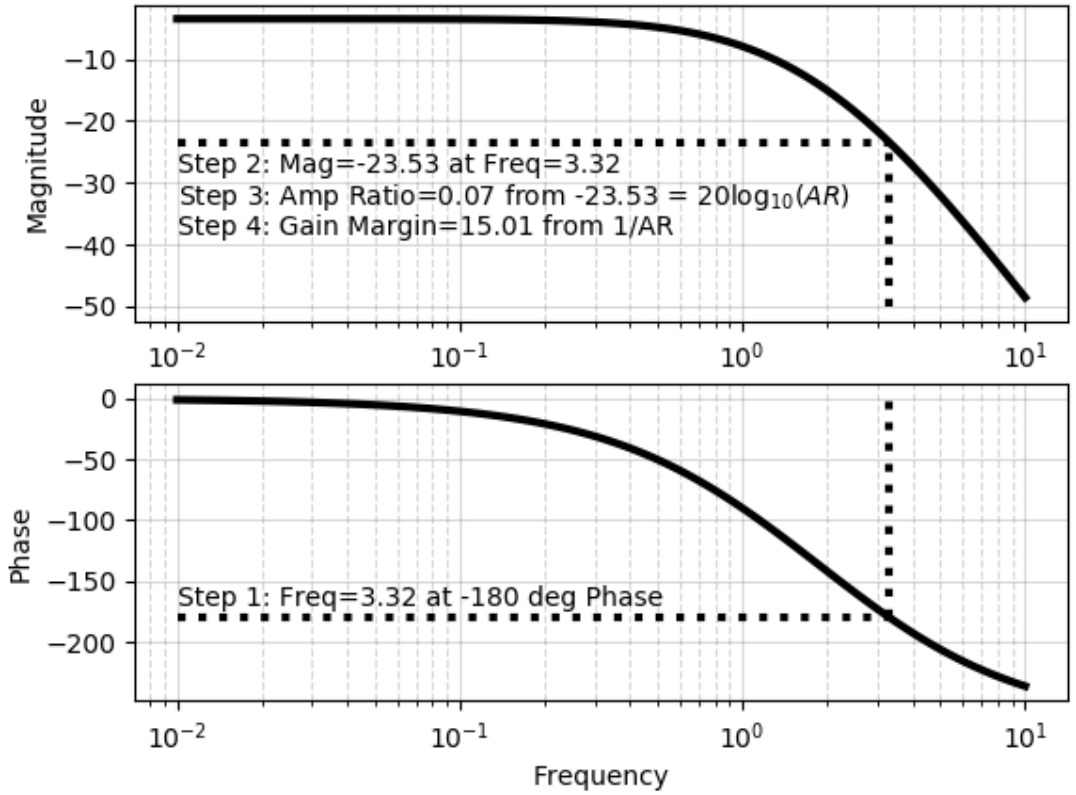
```
fig, ax = plt.subplots()
plt.subplots_adjust(left=0.10, bottom=0.25)
ls = []
for i in range(nr):
    plt.plot(rs[:,i],rs[:,i+nr],'r.',markersize=2)
    # this handle is required to update the plot
    if math.isclose(rs[0,i+nr],0.0):
        lbl = f'{rs[0,i]:0.2f}'
    else:
        lbl = f'{rs[0,i]:0.2f}, {rs[0,i+nr]:0.2f}i'
    l, = plt.plot(rs[0,i],rs[0,i+nr], 'ks', markersize=5,label=lbl)
    ls.append(l)
leg = plt.legend(loc='best')
plt.xlabel('Root (real)')
plt.ylabel('Root (imag)')
plt.grid(b=True, which='major', color='b', linestyle='-',alpha=0.5)
plt.grid(b=True, which='minor', color='r', linestyle='--',alpha=0.5)
plt.xlim([-4.0,0.5])
#plt.ylim([-4.0,4.0])

# slider creation
axcolor = 'lightgoldenrodyellow'
axKc = plt.axes([0.10, 0.1, 0.80, 0.03], facecolor=axcolor)
sKc = Slider(axKc, 'Kc', Kc1, Kc2, valinit=0, valstep=0.1)
```

```
def update(val):
    Kc_val= sKc.val
    indx = (np.abs(Kc-Kc_val)).argmin()
    for i in range(nr):
        ls[i].set_ydata(rs[indx,i+nr])
        ls[i].set_xdata(rs[indx,i])
        if math.isclose(rs[indx,i+nr],0.0):
            lbl = f'{rs[indx,i]:0.2f}'
        else:
            lbl = f'{rs[indx,i]:0.2f}, {rs[indx,i+nr]:0.2f}i'
        leg.texts[i].set_text(lbl)
    fig.canvas.draw_idle()
sKc.on_changed(update)

resetax = plt.axes([0.8, 0.025, 0.1, 0.04])
button = Button(resetax, 'Reset', color=axcolor, hovercolor='0.975')

def reset(event):
    sKc.reset()
button.on_clicked(reset)
```

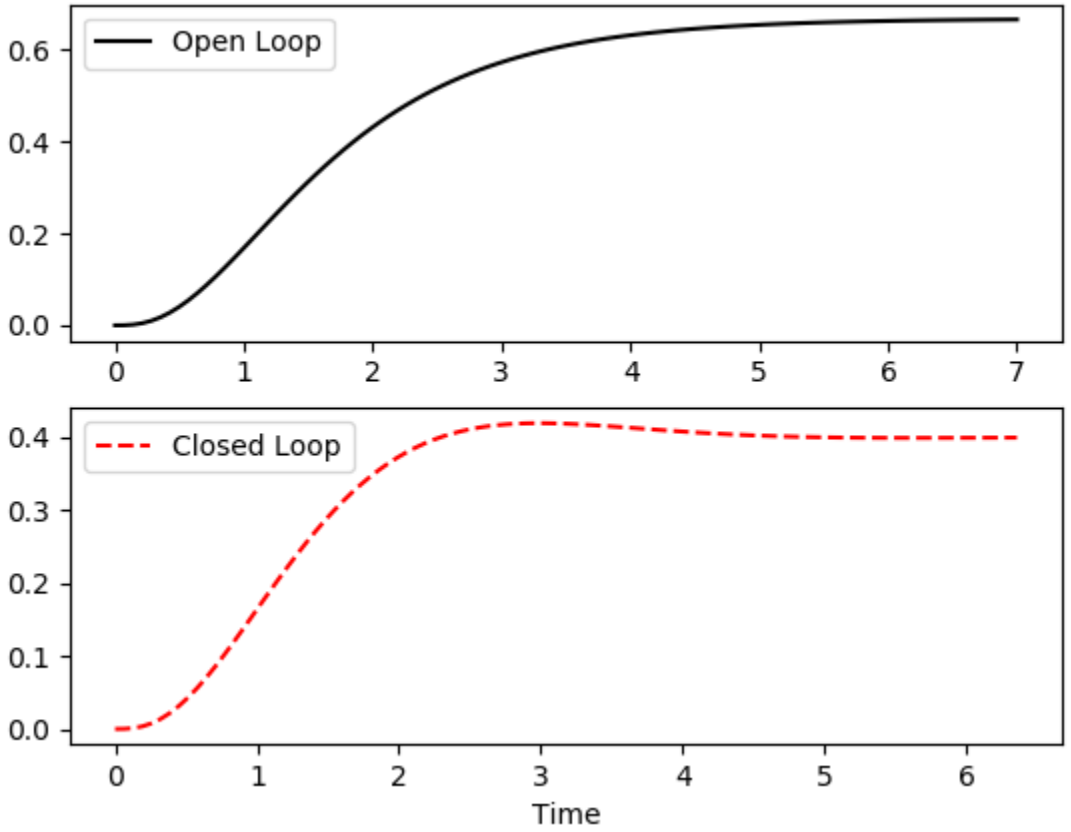
Bode Plot

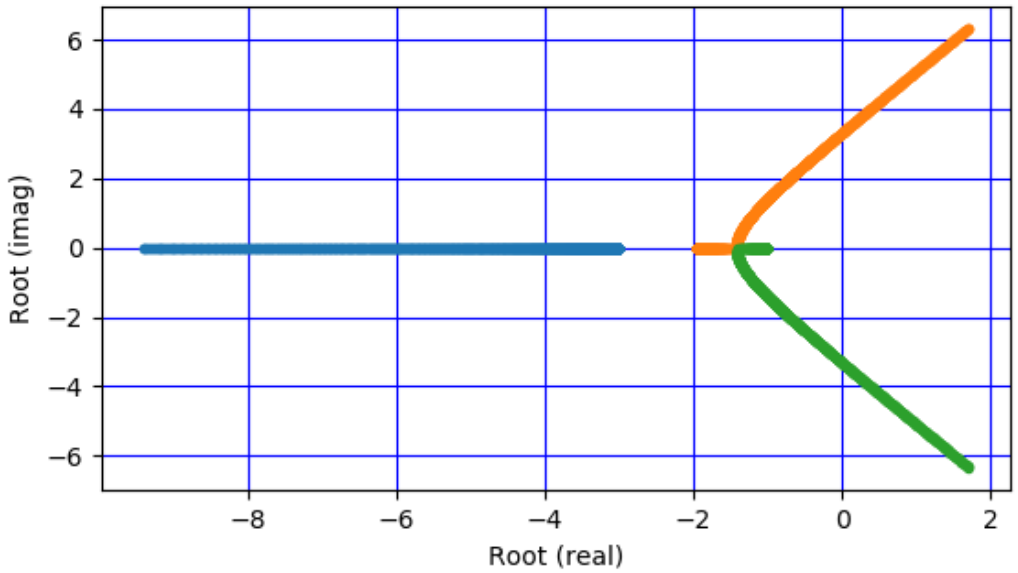
Demo Program: bodeplot.py

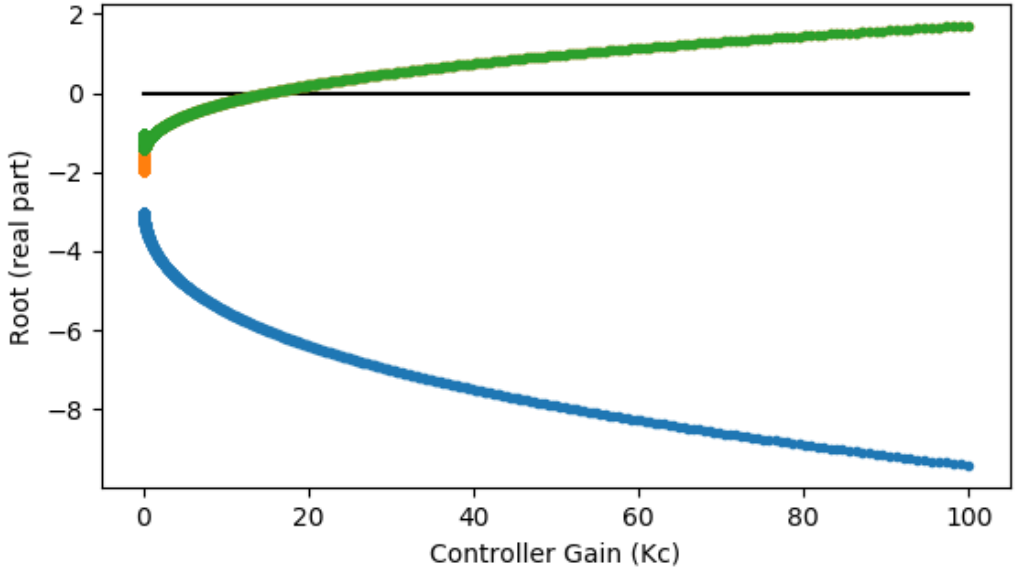
- Determine the gain margin at -180° phase. The magnitude at -180° phase is about -23 dB. With −23 = 20 log10(AR), the gain margin is $\frac{1}{AR}$ and approximately equal to 15.
- This is the upper bound on the controller gain to keep the system stable. This answer agrees with the root locus plot solution.



Bode Plot

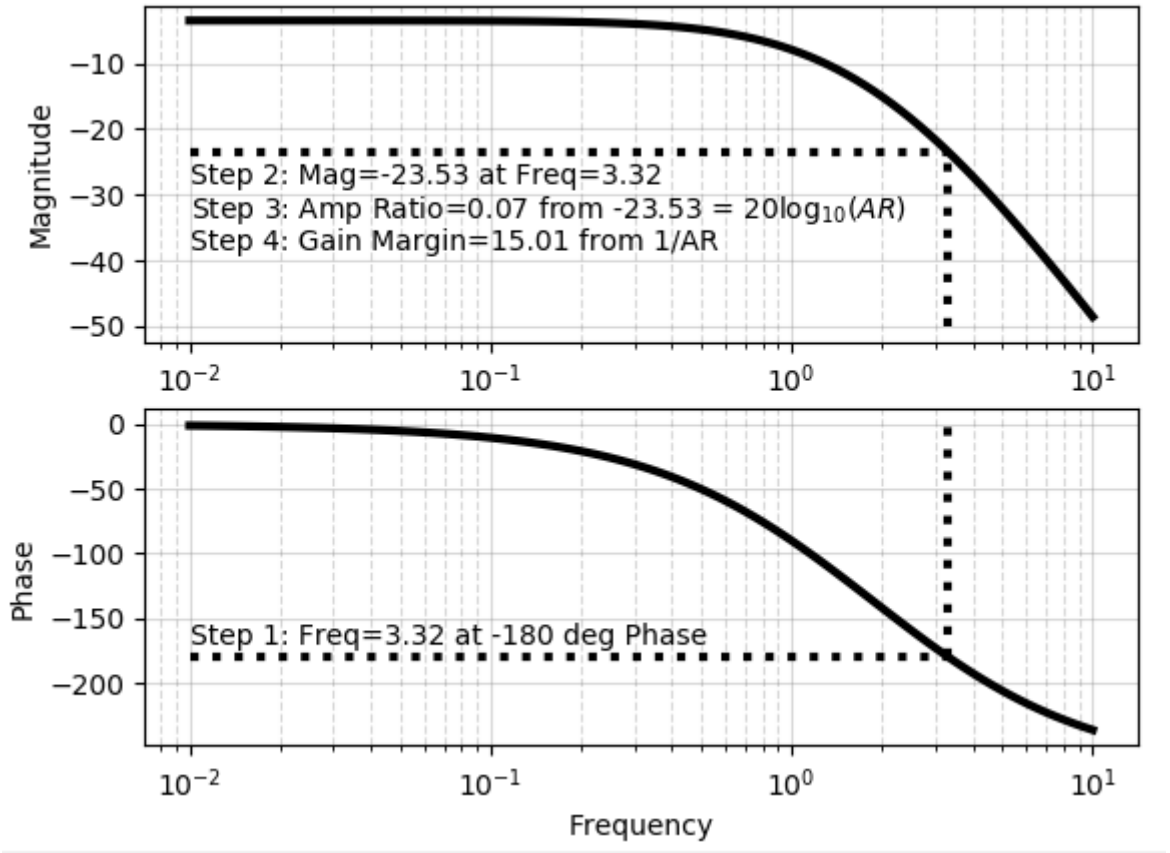

Open Loop
0.6
0.4
0.2
0.0
0
1
2
3
4
5
6
7
Closed Loop
0.4
0.3
0.2
0.1
0.0
0
1
2
3
4
5
6
Time


Root (imag)
Root (real)
6
4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2


Root (real part)
Controller Gain (Kc)
2
0
−2
−4
−6
−8
0
20
40
60
80
100

Bode Plot

Bode Plot


Magnitude
−10
−20
−30
−40
−50
Step 2: Mag=-23.53 at Freq=3.32
Step 3: Amp Ratio=0.07 from -23.53 = 20log10(AR)
Step 4: Gain Margin=15.01 from 1/AR
10−2
10−1
100
101
Phase
0
−50
−100
−150
−200
Step 1: Freq=3.32 at -180 deg Phase
10−2
10−1
100
101
Frequency