Problem definition: Write a program for fixed point continuation to generate both the stable and unstable branch of

$$\dot{x} = \mu - x^2 \tag{1}$$

Solution approach: We use sequential continuation for this problem. Each of the stable and unstable branches are tracked by starting from a location on them. It should be noted that due to the turning point at $\mu = 0$ we cannot get both branches using same initial point since we cannot got pass the turning point.

To track the stable branch, we choose our initial guess at $\mu_0 = 0.01$ and $x_0 = 0.1$. The x value is then updated for new value of μ_1 using the following equation.

$$x_1^{k+1} = x_1^k + r\Delta x^k \tag{2}$$

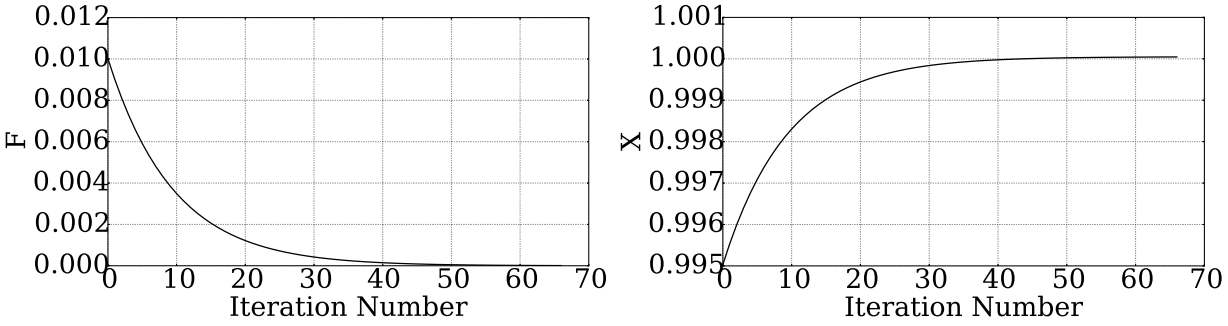
where we set x_1^0 equal to x_0. r is the relaxation parameter that defines how much we can go in Δx^k direction. For this problem, we chose r and 0.1. Δx^k is calculated using the following formula.

$$F_x\left(x_{j+1}^k, \mu_{j+1}\right)\Delta^k = -F\left(x_{j+1}^k, \mu_{j+1}\right) \tag{3}$$

where k is the iteration number and $j+1$ is the index of the control parameter we are at. We iterate on x until a convergence is satisfied. At the next fixed point, x_{j+1} we know that the right-hand-side of Equation (1) is equal to zero. Therefore, we define the convergence criteria as $F(x, \mu) < 10^{-5}$. The solution of this approach is verified using the analytical results for the stable and unstable branches. These are calculated as follows.

$$\begin{cases} x = \sqrt{\mu} & \text{stable branch} \\ x = -\sqrt{\mu} & \text{unstable branch} \end{cases} \tag{4}$$

The initial location for the stable branch is defined as $(x, \mu) = (0.01, 0.0001)$. The stable branch is followed for the 1000 points between $\mu = 0.0001$ and $\mu = 10$. The convergence plots for $\mu = 1$ is shown in Figure 1. As shown here, the value of F drops to zero when x converges to its value of 1 at $\mu = 1$.

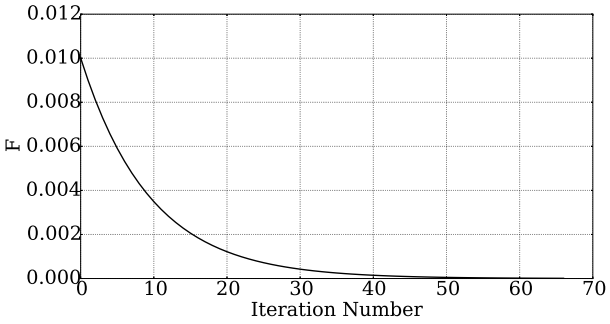


(a) Convergence of state equation, F (b) Convergence of x to its value.

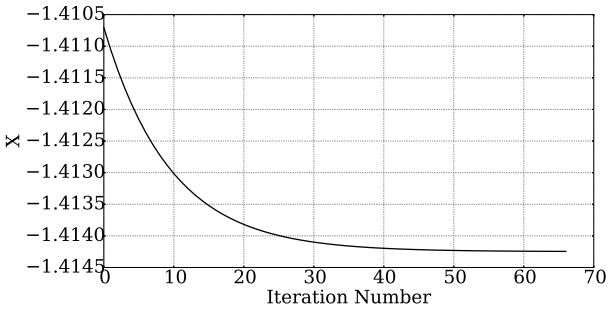
Figure 1: Convergence plots for x and F for $\mu = 1$.

The initial location for the unstable branch is defined as $(x, \mu) = (-0.01, 0.0001)$. The unstable branch is followed for the 1000 points between $\mu = 0.0001$ and $\mu = 10$. The convergence plots for $\mu = 2$ is shown in Figure 2. As shown here, the value of F drops to zero when x converges to its value of $-\sqrt{2}$ at $\mu = 2$.

The stable and unstable branch are shown in Figure (3). The analytical results are represented using circles and the *fixed point continuation* solution are shown with solid and dashed lines. The solid line represents the stable branch whereas the dashed line is used to show the unstable one. The `Python` code for this problem is included at the end of the documentation for this problem.



(a) Convergence of state equation, F



(b) Convergence of x to its value.

Figure 2: Convergence plots for x and F for $\mu = 2$.

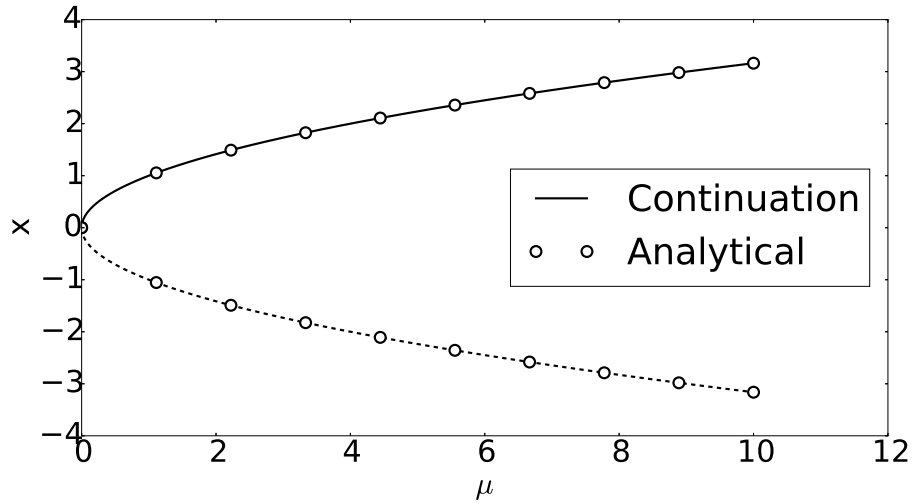


Figure 3: Comparison between fixed point continuation and analytical result for stable and unstable branches.

```
# Python code for Q1 of ME7160
__author__ = 'koorosh gobal'
import numpy as np
import matplotlib.pyplot as plt
# ------------------------------------
def FPC(mu0, r=0.1, N=10):
    def F(x, mu):
        return mu - x**2

    def Fx(x, mu):
        return -2 * x

    mu = np.linspace(mu0, 10+mu0, N)
    x1 = np.zeros(N)
    x1[0] = np.sqrt(mu0)
    x2 = np.zeros(N)
    x2[0] = -np.sqrt(mu0)

    for iMu in range(1, len(mu)):
        err = 1
        x1[iMu] = x1[iMu - 1]
        while err > 1e-5:
            dx = -F(x1[iMu], mu[iMu]) / Fx(x1[iMu], mu[iMu])
            x1[iMu] = x1[iMu] + r * dx
            err = F(x1[iMu], mu[iMu])

    for iMu in range(1, len(mu)):
        err = 1
        x2[iMu] = x2[iMu - 1]
        while err > 1e-5:
            dx = -F(x2[iMu], mu[iMu]) / Fx(x2[iMu], mu[iMu])
            x2[iMu] = x2[iMu] + r * dx
            err = F(x2[iMu], mu[iMu])

    return {'x1':x1, 'x2':x2, 'mu':mu}

sol = FPC(0.0001, N=1001)

mu = sol['mu']
x1 = sol['x1']
x2 = sol['x2']

plt.figure()
plt.plot(mu, x1, 'k',
         np.linspace(0, mu[-1], 10), np.sqrt(np.linspace(0, mu[-1], 10)),
                'wo',
         mu, x2, 'k--',
         np.linspace(0, mu[-1], 10), -np.sqrt(np.linspace(0, mu[-1], 10)),
                'wo')
plt.legend(['Continuation', 'Analytical'],loc='best')
plt.xlabel('$\mu$')
plt.ylabel('x')
plt.show()
```

Problem definition: Apply Harmonic Balance to obtain the response of a Duffing oscillator where the governing equation is defined as

$$\ddot{x} + \left(0.01 + \frac{\alpha}{100}\right)\dot{x} + x + \frac{\alpha}{10}x^3 = \sin 3t \qquad \text{where } \alpha = 12 \tag{5}$$

Solution approach: To solve Equation (5) using the Harmonic Balance (HB) method, we first define the residual function as

$$\mathcal{R} = \ddot{x} + \left(0.01 + \frac{12}{100}\right)\dot{x} + x + \frac{12}{10}x^3 - \sin 3t \tag{6}$$

we then assume a solution $\bar{x}$. If $\bar{x}$ is indeed the solution of Equation (5), when substituting it in the residual equation of (6) we should get zero. However, since it is an assumed solution, the residual won't be zero. By updating $\bar{x}$ is an optimization loop to minimize $\mathcal{R}^2$, we can get the solution of Equation (5). In this process, it is required to calculate $\ddot{x}$ and $\dot{x}$ for $\mathcal{R}$. This is done in frequency domain.

The variable x in the time domain can be approximated as follows in the frequency domain.

$$x(t) = \sum_{k=-\infty}^{\infty} X_k \cdot e^{i\omega_0 k t} \quad , \quad \omega_0 = \frac{2\pi}{T} \tag{7}$$

The time derivatives are calculated by differentiating above equation with respect to time.

$$\dot{x}(t) = \sum_{k=-\infty}^{\infty} i\omega_k X_k \cdot e^{i\omega_k t} \tag{8a}$$

$$\ddot{x}(t) = \sum_{k=-\infty}^{\infty} -\omega_k^2 X_k \cdot e^{i\omega_k t} \tag{8b}$$

By substituting Equations (7), (8a), and (8b) into Equation (6), we can minimize the residual based on a guessed displacement. The flow chart for this is shown in Figure 4.

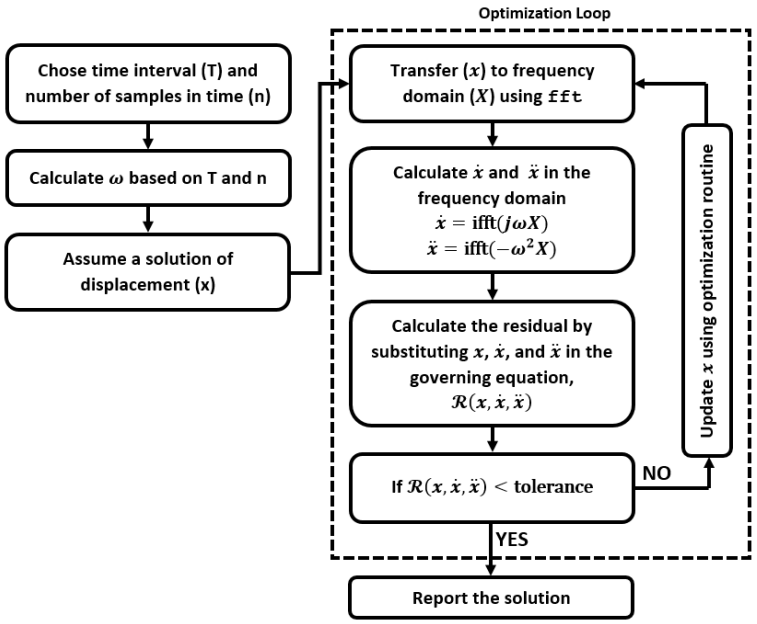


Figure 4: Flowchart for numerical approach.

To transfer the time domain data for frequency domain we use `numpy.fft.fft` function and for converting the frequency domain data back to time domain we used `numpy.fft.ifft`. We

sampled the frequency domain using 19 points. The initial guess for displacement is selected as a vector of ones, `numpy.ones`. It should be noted that this is a vector of displacement in time. The minimization of residual is done using python `scipy.optimize.minimize` function using `SLSQP` method. The convergence plot for the residual function is shown in Figure 5. The horizontal lines in this graph represent the steps used to calculate the finite difference derivatives of the residual with respect to each of design variables (number of x points in time). As can be seen here, the residual function is minimized at the end of optimization stage.

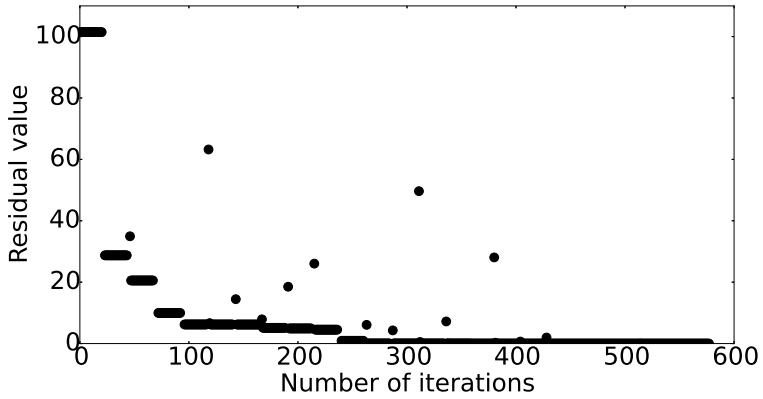


Figure 5: Convergence of residual.

To verify the result of HB method, we used numerical integration to calculate the solution of Equation (5). It should be noted that HB is capable of capturing the steady-state response of the system. Therefore, it is required to let the transient response of the time integration to die of before comparing the results. This comparison is shown in Figure 6.

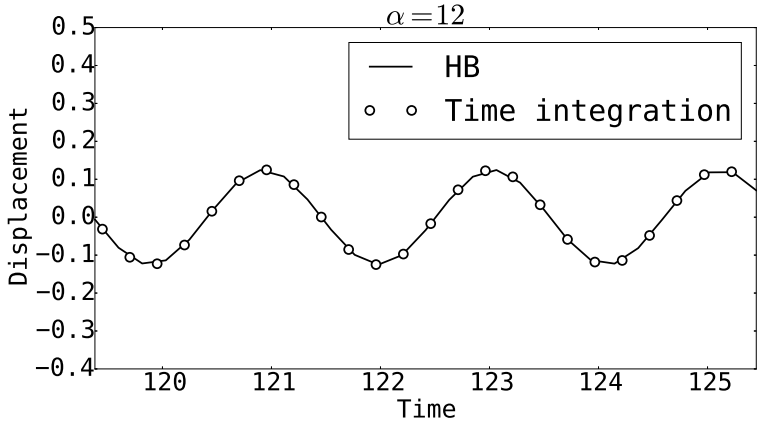


Figure 6: Comparison between HB and time integration results.

The numerical results of HB can be written in terms of trigonometric functions. This is done using fast Fourier transforms as well. The results of `numpy.fft.fft` is a vector in frequency domain where its values are the coefficients of trigonometric functions with appropriate frequencies. In particular the real portion of the `fft` results is the cosine transform and the imaginary portion is the sine transform of the vector in time domain. The frequency corresponding to the `fft` vector of length $N = 9$ can be written as:

$$\omega_k = \frac{2\pi}{T}\begin{bmatrix}0 & 1 & 2 & 3 & 4 & -4 & -3 & -2 & -1\end{bmatrix} \tag{9}$$

Using Equation (9) and the real and imaginary portion of `numpy.fft.fft` vector, the time domain results can be reconstructed upto n harmonics where $n \leq k$. k is the number of harmonics used

in calculating the response using HB method. Using this approach, the result of approximated using 3 harmonics is as follows

$$\begin{aligned} x(t) = &- 1.6074 \times 10^{-4} \\ &+ 1.3664 \times 10^{-5} \cos t - 1.4597 \times 10^{-5} \sin t \\ &+ 4.6340 \times 10^{-5} \cos 2t + 3.6900 \times 10^{-5} \sin 2t \\ &+ 6.0986 \times 10^{-3} \cos 3t + 1.2491 \times 10^{-1} \sin 3t \end{aligned}$$

Because most of the coefficients are small, above is approximated as

$$x(t) = 0.12491 \sin 3t \tag{10}$$

The comparison between approximating the solution of Equation (5) by (10), HB with 19 terms and numerical integration is shown in Figure 7. As can be seen here, the results match very well. The `python` code used for this problem is included in the following pages.

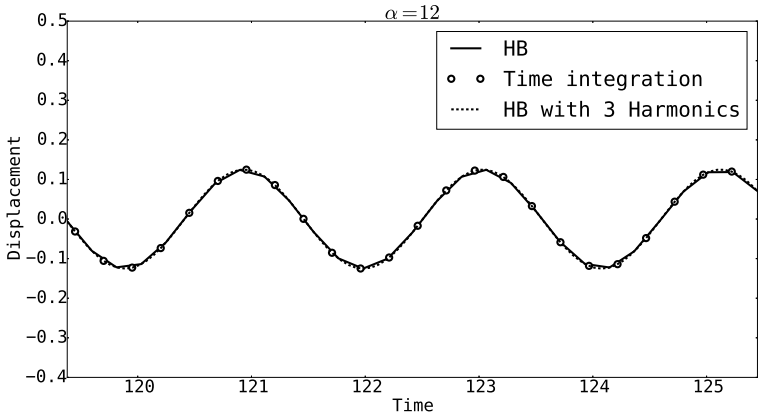


Figure 7: Comparison between HB and time integration results.

```
# Python code for Q2 of ME7160
__author__ = 'koorosh gobal'
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize
from scipy.integrate import odeint
# # ----------------------------------------
font = {'family' : 'monospace',
        'weight' : 'normal',
        'size'   : 22}
plt.rc('font', **font)
linewidth = 5.0
markersize = 15
skip = 10
# # ----------------------------------------
# Harmonic Balance (HB) method
N = 19 # Number of harmonics for HB
T = 1 * 2 * np.pi # Time interval
t = np.linspace(0, T, N+1)
t = t[0:-1]
f = np.sin(3*t) # Forcing function
Omega = 2*np.pi / T * np.fft.fftfreq(N, 1/N) # \omega
x0 = np.ones(N) # Initial guess for the solution
alpha = 12 # \alpha for this problem

# defining residual function of HB method
def residual(x):
    X = np.fft.fft(x)
    dx = np.fft.ifft(np.multiply(1j * Omega, X))
    ddx = np.fft.ifft(np.multiply(-Omega**2, X))
    Residual = ddx + (0.01 + alpha / 100) * dx + x + alpha / 10 * x**3 - f
    Residual = np.sum(np.abs((Residual**2)))
    return Residual

res = minimize(residual, x0, method='SLSQP') # Minimizing the residual
xSol = res.x # Solution in time domain

# Converting result to sine and cosines
n = 3 # Number of harmonics used
X = np.fft.fft(xSol)
X = np.append(X[:n+1], X[-n:]) / N
OMEGA = np.append(Omega[:n+1], Omega[-n:]).reshape(1, -1)
tReconst = np.linspace(0, 2*np.pi, 100).reshape(-1, 1)
OMEGAt = tReconst.dot(OMEGA).T
xReconst = X[0] # Reconstracuted results
for i in range(1, OMEGA.shape[1]):
    xReconst = xReconst + np.real(X[i]) * np.cos(OMEGAt[i, :]) -
                          np.imag(X[i]) * np.sin(OMEGAt[i, :])
np.savetxt('X.txt', X*2)
np.savetxt('OMEGA.txt', OMEGA.T)

# Numerical solution using odeint
def RHS(X, t=0.0):
    x1, x2 = X
```

```
    x1dot = x2
    x2dot = -(0.01 + alpha / 100) * x2 - x1 - alpha / 10 * x1**3
            + np.sin(3*t)
    return [x1dot, x2dot]

ta = np.linspace(0.0, 19*2*np.pi + t[-1], 5000)
sol = odeint(RHS, [0, 0], ta)

# Ploting the comparing the results
# We have to shift the HB result to region where the transient
# Solution is died out and the particular solution is the dominent
# one. This is what '19*2*np.pi' does.
plt.figure()
plt.plot(19*2*np.pi + t, res.x, 'k',
         ta[0:-1:skip], sol[0:-1:skip, 0], 'wo',
         19*2*np.pi + tReconst, xReconst, 'k--',
         lw=linewidth, ms=markersize, mew=linewidth)
plt.legend(['HB', 'Time integration', 'HB with 3 Harmonics'], loc='best')
plt.xlabel('Time')
plt.ylabel('Displacement')
plt.xlim([19*2*np.pi + t[0], 19*2*np.pi + t[-1]])
plt.title(r'$\alpha = 12$')
plt.show()
```

Problem definition: Apply Floquet theory to your solution of the preceding problem to determine whether the solution is stable or not.

Solution approach: Equation (5) is a nonautonomous system. To apply Floquet theory, we need to convert this system to an autonomous system by defining a new state for time. This is done by setting $3t$ equal to x_3. The state equations are written as

$$\begin{cases} \dot{x}_1 = x_2 \\ \dot{x}_2 = -\left(0.01 + \dfrac{\alpha}{100}\right) x_2 - x_1 - \dfrac{\alpha}{10} x_1^3 + \sin x_3 \\ \dot{x}_3 = 3 \end{cases} \tag{11}$$

We assume the solution of Equation (5) is X_0. A disturbance, $Y(t)$ is imposed on X_0 as follows

$$X(t) = X_0 + Y(t) \tag{12}$$

It should be noted that we are only putting a disturbance on x_1 and x_2. Therefore, $y_3(t)$ is equal to zero. Substituting Equation (12) into (11) we get the following

$$\begin{cases} \dot{x}_{01} + \dot{y}_1 = x_{02} + y_2 \\ \dot{x}_{02} + \dot{y}_2 = -\left(0.01 + \dfrac{\alpha}{100}\right)(x_{02} + y_2) - x_{01} - y_1 - \dfrac{\alpha}{10}(x_{01} + y_1)^3 + \sin(x_{03} + y_3) \\ \dot{x}_{03} + \dot{y}_3 = 3 \end{cases}$$

Since X_0 is the solution of Equation (5), above can be rewritten as

$$\begin{cases} \dot{y}_1 = y_2 \\ \dot{y}_2 = -\left(0.01 + \dfrac{\alpha}{100}\right) y_2 - y_1 - \dfrac{\alpha}{10}\left(3y_1 x_{01}^2 + 3y_1^2 x_{01} + y_1^3\right) \\ \dot{y}_3 = 0 \end{cases} \tag{13}$$

We linearise Equation (13) at its stationary point at $[0, 0, 0]$. The linearised equation is written in the matrix form as follows.

$$\begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \\ \dot{y}_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -\left(1 + \dfrac{3\alpha}{10} x_{01}^2\right) & -\left(0.01 + \dfrac{\alpha}{100}\right) & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \tag{14}$$

where x_{01} is the displacement result calculated by solving Equation (5) as shown in Equation (10). Equation (14) is solved numerically using `scipy.integrate.odeint` from 0 and 2π. The state equations are required to be solved three times (since we have three states) using different boundary conditions. The boundary conditions for each of these cases are defined by perturbing one of the states by unity and setting the rest equal to zero. The solution of the last time step for each is saved and put in the rows of the matrix Φ. The Φ matrix for this problem is calculated as

$$\Phi = \begin{bmatrix} 0.666 & -0.050 & 0 \\ 0.048 & 0.659 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{15}$$

The eigenvalues of this matrix, define the characteristics of the solution. These eigenvalues are shown in the following equation.

$$\begin{aligned} \lambda_1 &= 0.662 + 0.049i \\ \lambda_2 &= 0.662 - 0.049i \\ \lambda_3 &= 1.000 \end{aligned}$$

For the solution to be stable, all the eigenvalues except one need to be inside the unit circle in the complex plain. It is easier to look at the *norm* of the eigenvalues. If the norm is less than zero it means that the solution is stable. In other words, the disturbance dies out when the time expands. The norm of eigenvalues are shown in the following equation.

$$|\lambda_1| = 0.664$$
$$|\lambda_2| = 0.664$$
$$|\lambda_3| = 1.000$$

As shown here, all the eigenvalues except than one of them is less than unity. Therefore, we can conclude than the solution is stable. This can verified by looking the the numerical solution of the system as well. The `python` code used to this problem is available in the following pages.

```
# Python code for Q3 of ME7160
__author__ = 'koorosh gobal'
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

alpha = 12
t = np.linspace(0.0, 2*np.pi, 100)

# Linearized State Matrix
def A(Y, t=0.0):
    y1, y2, y3 = Y
    x10 = 0.12494 * np.sin(3 * t)
    y1dot = y2
    y2dot = -(0.01 + alpha / 100) * y2 - (1 + 0.3 * alpha * x10**2) * y1
    y3dot = 0
    return [y1dot, y2dot, y3dot]

# Solving the perturbed equation using different initial conditions
sol1 = odeint(A, [1, 0, 0], t)
sol2 = odeint(A, [0, 1, 0], t)
sol3 = odeint(A, [0, 0, 1], t)

# Assembling \Phi matrix
PHI = np.array([sol1[-1, :], sol2[-1, :], sol3[-1, :]])
eigenValue = np.linalg.eigvals(PHI)

print(eigenValue)
print(np.abs(eigenValue))
```