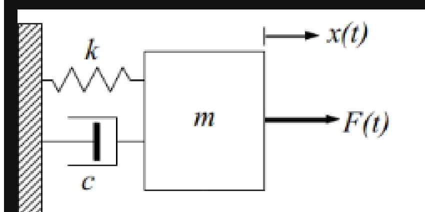
In [83]:

```python
#################################################################################
#                               BEN WISH LE                                     #
#                                ENGR 3703                                      #
#                             DUE: 05/05/2022                                   #
# You are an essential ingredient in our ongoing effort to reduce Security Risk. #
#################################################################################

from IPython.display import Image
img = Image(url="1.PNG", width = 1500, height = 6000)
img
```

Out[83]:

# ENGR 3703 Project - What you have to do

The project involves a very common problem in engineering - the spring-mass-damper problem. It has applications in electrical and mechanical engineering and beyond. The basic problem is below:




In the figure the variables and parameters have the following meaning:

- $m$ = mass (kg)
- $c$ = damping constant (kg/s) - proportional to the speed of $m$
- $k$ = spring constant (N/s/s) - proportional to the distance $m$ is from it's
- $x(t)$ = position of $m$ as a function of time
- $F(t)$ = A force applied to $m$ as a function of time

In the background notebook, we determined that:

$$\ddot{x} + 2\zeta\omega_n\dot{x} + \omega_n^2 x = \frac{F(t)}{m}$$

1. Create a github repository to host your project files. If you do not have a github account you will need to create one. I will need a link to the repo. If the repo is not public you will need to make me a collaborator so I can access the repo. a. You can have as many files as you want in the repo, but there should be a single jupyter notebook called: yourname_engr3703_fall2021_project.ipynb. This will serve as your report for the project.
2. Write python code that uses RK 4th order to solve the ODE above. For the first part of the project, you will assume F(t) is zero. Later you will be using F(t), so plan accordingly.
3. Test your code with all three unforced cases - overdamping, critically damped, and underdamping. You will choose the values of $\omega_n$ and $\zeta$. You will need to demonstrate your code works over a range of time-step sizes for all cases. Code and graphics are required. You should calculate and verify that both $x(t)$ and $v(t)$ can reliably be calculated using Runge-Kutta by numerically and graphically comparing the analytical solutions and the calculated values from your RK program.
4. Test your code using values of F(t)/m. You must choose at least three values two of which may not be constant values of F(t) - i.e. there has to be some time variation of F. You should analyze the results of these tests. If there are known solutions for a given F(t) compare to those. In the end you need to convice the reader (Dr. Lemley) that your code is accurately calculating both x(t) and v(t) for each of these cases. Again, numerical and graphical evidence is required.
5. Finally test your code with $F(t)/m = Acos\omega_f t$ where you choose the value of $A$. You should vary $\omega_d$ over a range of values such that you can see instances where the oscillations (x(t)) are growing out of control over time (i.e. resonance). Graphically show how this occuring by displaying cases with different $\omega_f$ change the velocity and position over time.

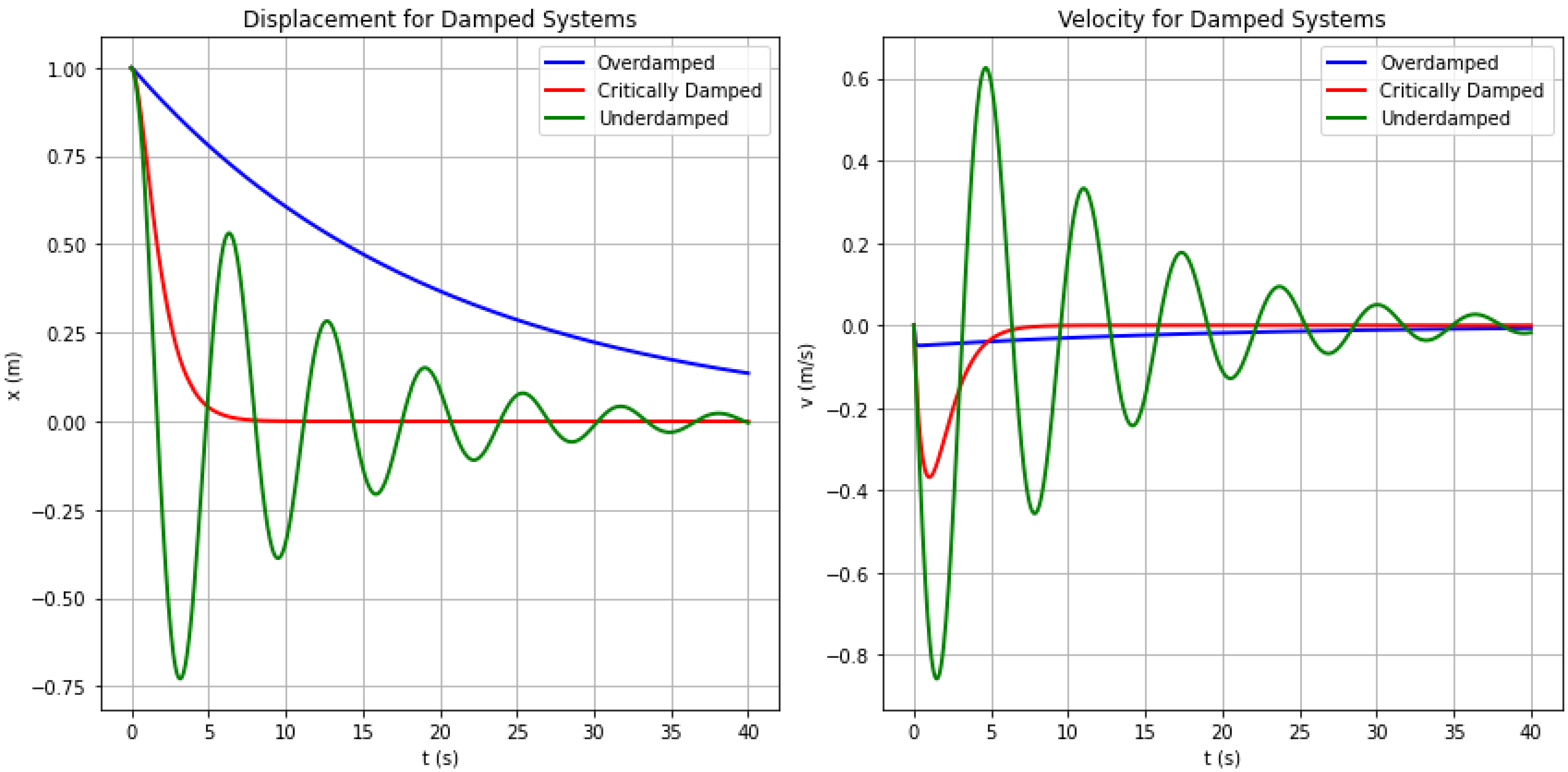
In [129…

```python
# 2 and 3. Using value of F(t)/m = 0
# Test the code with all three unforced cases - overdamping, critically damped, and underdamping
#

%matplotlib inline
from math import *
import numpy as np
import matplotlib.pyplot as plt

# Test case: f(t)/m = 0
zeta_od = 10
zeta_cd = 1.
zeta_ud = 0.1

omega_n = 1.0
omega_d = omega_n * sqrt(1-zeta_ud**2)

v0 = 0
x0 = 1
t0 = 0
t_end = 40
dt = (t_end - t0) / 499

t_1 = t0
t_2 = t0
t_3 = t0

v_1 = v0
v_2 = v0
v_3 = v0

x_1 = x0
x_2 = x0
x_3 = x0

t_list_1 = [t0]
t_list_2 = [t0]
t_list_3 = [t0]

v_list_1 = [v0]
v_list_2 = [v0]
v_list_3 = [v0]

x_list_1 = [x0]
x_list_2 = [x0]
```

```python
x_list_3 = [x0]


# Over damped
def dxdt_1(t, v, x):
    return v
def dvdt_1(t, v, x):
    return  - 2*zeta_od*omega_n*v - (omega_n**2)*x

# Critically damped
def dxdt_2(t, v, x):
    return v
def dvdt_2(t, v, x):
    return  - 2*zeta_cd*omega_n*v - (omega_n**2)*x

# Under damped
def dxdt_3(t, v, x):
    return v
def dvdt_3(t, v, x):
    return  - 2*zeta_ud*omega_d*v - (omega_d**2)*x

def RungeKuttaCoupled(x, y, z, dx, dydx, dzdx):
    k1 = dx*dydx(x, y, z)
    h1 = dx*dzdx(x, y, z)

    k2 = dx*dydx(x+dx/2., y+k1/2., z+h1/2.)
    h2 = dx*dzdx(x+dx/2., y+k1/2., z+h1/2.)

    k3 = dx*dydx(x+dx/2., y+k2/2., z+h2/2.)
    h3 = dx*dzdx(x+dx/2., y+k2/2., z+h2/2.)

    k4 = dx*dydx(x+dx, y+k3, z+h3)
    h4 = dx*dzdx(x+dx, y+k3, z+h3)

    y = y + 1./6.*(k1+2*k2+2*k3+k4)
    z = z + 1./6.*(h1+2*h2+2*h3+h4)
    x = x + dx
    return x, y, z

while t_1 <= t_end:
    t_1, v_1, x_1 = RungeKuttaCoupled(t_1, v_1, x_1, dt, dvdt_1, dxdt_1)
    t_2, v_2, x_2 = RungeKuttaCoupled(t_2, v_2, x_2, dt, dvdt_2, dxdt_2)
    t_3, v_3, x_3 = RungeKuttaCoupled(t_3, v_3, x_3, dt, dvdt_3, dxdt_3)
```

```python
    t_list_1.append(t_1)
    t_list_2.append(t_2)
    t_list_3.append(t_3)

    v_list_1.append(v_1)
    v_list_2.append(v_2)
    v_list_3.append(v_3)

    x_list_1.append(x_1)
    x_list_2.append(x_2)
    x_list_3.append(x_3)

print ("\nUnforced test cases: f(t)/m = 0\n\n")

fig = plt.figure(figsize=(12,6))

# Plotting displacement
ax1 = plt.subplot(121)
plt.plot(t_list_1, x_list_1, label="Overdamped"       ,color="b",linewidth="2.0")
plt.plot(t_list_2, x_list_2, label="Critically Damped",color="r",linewidth="2.0")
plt.plot(t_list_3, x_list_3, label="Underdamped"      ,color="g",linewidth="2.0")
plt.title("Displacement for Damped Systems")
plt.xlabel("t (s)")
plt.ylabel("x (m)")
plt.legend()
plt.tight_layout()
plt.grid()

# Plotting velocity
ax2 = plt.subplot(122)
plt.plot(t_list_1, v_list_1, label="Overdamped"       ,color="b",linewidth="2.0")
plt.plot(t_list_2, v_list_2, label="Critically Damped",color="r",linewidth="2.0")
plt.plot(t_list_3, v_list_3, label="Underdamped"      ,color="g",linewidth="2.0")
plt.title("Velocity for Damped Systems")
plt.xlabel("t (s)")
plt.ylabel("v (m/s)")
plt.legend()
plt.tight_layout()
plt.grid()

plt.show()
```

```
Unforced test cases: f(t)/m = 0
```




In [19]:

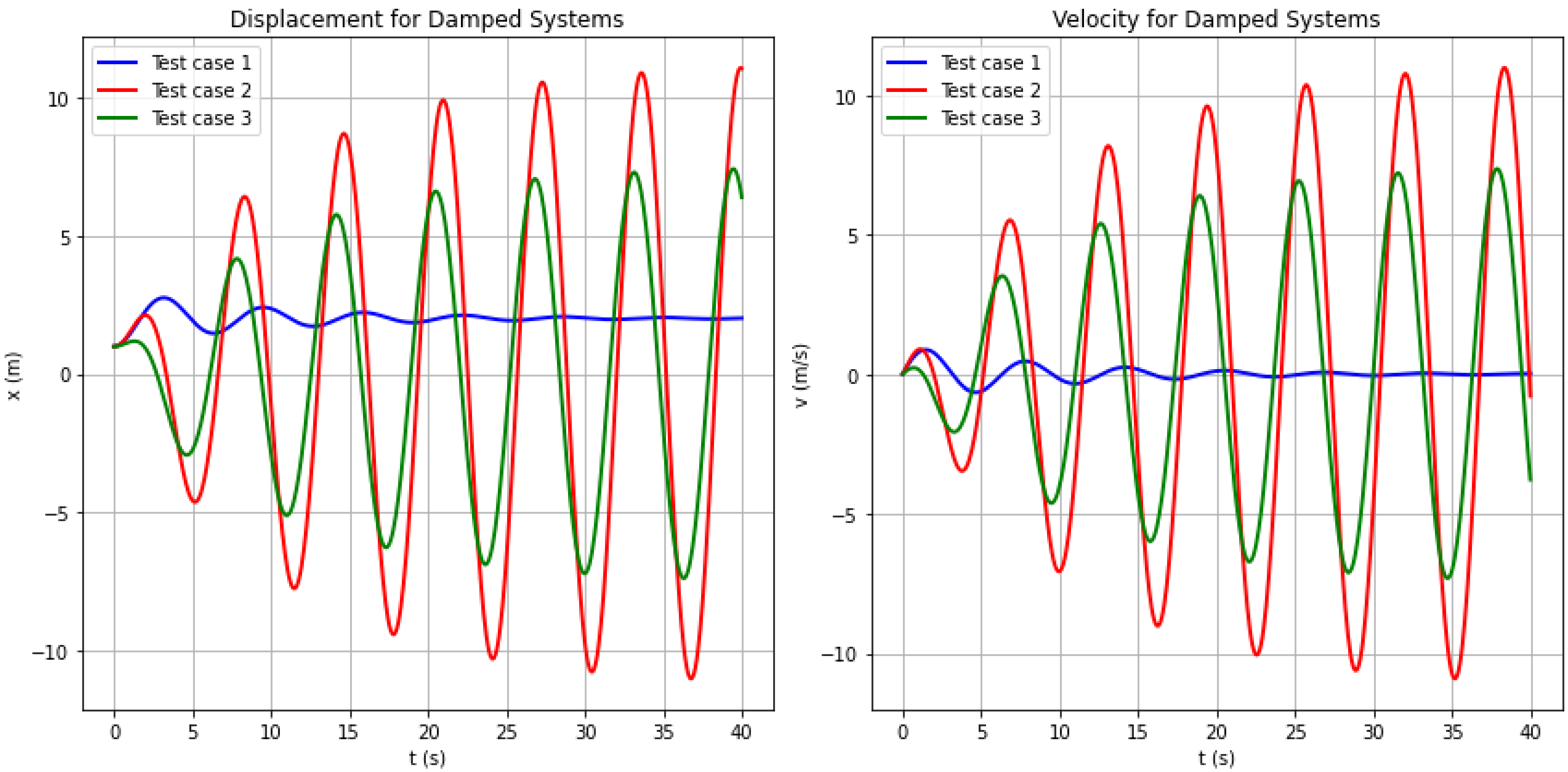
```
# 4. Test code using values of F(t)/m (FOR ALL UNDER DAMPED CASES)

%matplotlib inline
from math import *
import numpy as np
import matplotlib.pyplot as plt

zeta_ud = 0.1
omega_n = 1.0
omega_d = omega_n * sqrt(1-zeta_ud**2)

# Test case 1: f(t)/m = 2
def dvdt_1(t, v, x):
    return 2 - 2*zeta_ud*omega_d*v - (omega_d**2)*x
def dxdt_1(t, v, x):
    return v
```

```python
# Test case 2: f(t)/m = 2*cos (omega_d*t + 0.5) + sin(omega_d*t)
def dvdt_2(t, v, x):
    return 2*cos(omega_d*t) + sin(omega_d*t) - 2*zeta_ud*omega_d*v - (omega_d**2)*x
def dxdt_2(t, v, x):
    return v

# Test case 3: f(t)/m = 1.5*cos(omega_d*t)
def dvdt_3(t, v, x):
    return 1.5*cos(omega_d*t) - 2*zeta_ud*omega_d*v - (omega_d**2)*x
def dxdt_3(t, v, x):
    return v

v0 = 0
x0 = 1
t0 = 0
t_end = 40
dt = (t_end- t0)/499

t_1 = t0
t_2 = t0
t_3 = t0

v_1 = v0
v_2 = v0
v_3 = v0

x_1 = x0
x_2 = x0
x_3 = x0

t_list_1 = [t0]
t_list_2 = [t0]
t_list_3 = [t0]

v_list_1 = [v0]
v_list_2 = [v0]
v_list_3 = [v0]

x_list_1 = [x0]
x_list_2 = [x0]
x_list_3 = [x0]

def RungeKuttaCoupled(x, y, z, dx, dydx, dzdx):
    k1 = dx*dydx(x, y, z)
    h1 = dx*dzdx(x, y, z)
```

```
    k2 = dx*dydx(x+dx/2., y+k1/2., z+h1/2.)
    h2 = dx*dzdx(x+dx/2., y+k1/2., z+h1/2.)

    k3 = dx*dydx(x+dx/2., y+k2/2., z+h2/2.)
    h3 = dx*dzdx(x+dx/2., y+k2/2., z+h2/2.)

    k4 = dx*dydx(x+dx, y+k3, z+h3)
    h4 = dx*dzdx(x+dx, y+k3, z+h3)

    y = y + 1./6.*(k1+2*k2+2*k3+k4)
    z = z + 1./6.*(h1+2*h2+2*h3+h4)
    x = x + dx
    return x, y, z


while t_1 <= t_end or t_2 <= t_end or t_3 <= t_end:
    t_1, v_1, x_1 = RungeKuttaCoupled(t_1, v_1, x_1, dt, dvdt_1, dxdt_1)
    t_list_1.append(t_1)
    v_list_1.append(v_1)
    x_list_1.append(x_1)

    t_2, v_2, x_2 = RungeKuttaCoupled(t_2, v_2, x_2, dt, dvdt_2, dxdt_2)
    t_list_2.append(t_2)
    v_list_2.append(v_2)
    x_list_2.append(x_2)

    t_3, v_3, x_3 = RungeKuttaCoupled(t_3, v_3, x_3, dt, dvdt_3, dxdt_3)
    t_list_3.append(t_3)
    v_list_3.append(v_3)
    x_list_3.append(x_3)

print ()
print ("I want to observe the forced vibrations and see the resonance results!!")
print ("UNDER DAMPED TEST CASE 1: f(t)/m = 2")
print ("UNDER DAMPED TEST CASE 2: f(t)/m = 2*cos (omega_d*t + 0.5) + sin(omega_d*t)")
print ("UNDER DAMPED TEST CASE 3: f(t)/m = 1.5*cos (omega_d*t)")
print ("Please look at the attached pdf file for numerical evidence ")
print ("And explaination for x(t) increases by transient solution or stay in steady state solution")
print ()

fig = plt.figure(figsize=(12,6))
# Plotting displacement
ax1 = plt.subplot(121)
plt.plot(t_list_1, x_list_1, label="Test case 1",color="b",linewidth="2.0")
```

```python
plt.plot(t_list_2, x_list_2, label="Test case 2",color="r",linewidth="2.0")
plt.plot(t_list_3, x_list_3, label="Test case 3",color="g",linewidth="2.0")
plt.title("Displacement for Damped Systems")
plt.xlabel("t (s)")
plt.ylabel("x (m)")
plt.legend()
plt.tight_layout()
plt.grid()


# Plotting velocity
ax2 = plt.subplot(122)
plt.plot(t_list_1, v_list_1, label="Test case 1",color="b",linewidth="2.0")
plt.plot(t_list_2, v_list_2, label="Test case 2",color="r",linewidth="2.0")
plt.plot(t_list_3, v_list_3, label="Test case 3",color="g",linewidth="2.0")
plt.title("Velocity for Damped Systems")
plt.xlabel("t (s)")
plt.ylabel("v (m/s)")
plt.legend()
plt.tight_layout()
plt.grid()

plt.show()
```

```
I want to observe the forced vibrations and see the resonance results!!
UNDER DAMPED TEST CASE 1: f(t)/m = 2
UNDER DAMPED TEST CASE 2: f(t)/m = 2*cos (omega_d*t + 0.5) + sin(omega_d*t)
UNDER DAMPED TEST CASE 3: f(t)/m = 1.5*cos (omega_d*t)
Please look at the attached pdf file for numerical evidence
And explaination for x(t) increases by transient solution or stay in steady state solution
```



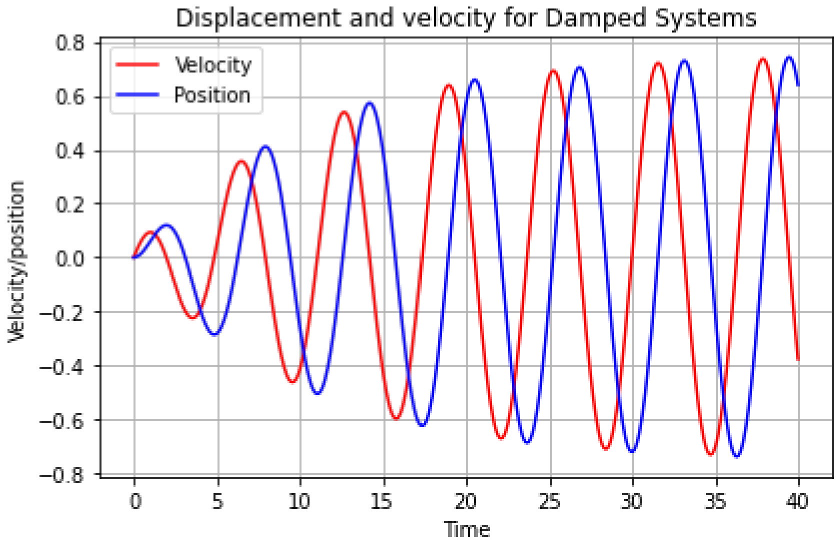

In [18]:

```
# 5. Test code using values of F(t)/m = A*cos(omega_n*t) (FOR ALL UNDER DAMPED CASES)

%matplotlib inline
from math import *
import numpy as np
import matplotlib.pyplot as plt

zeta_ud = 0.1
omega_n = 1.0
omega_d = omega_n * sqrt(1-zeta_ud**2)
A = 0.15

# Test case: f(t)/m = A*cos(omega_d*t)
def dvdt_1(t, v, x):
    return A*cos(omega_d*t) - 2*zeta_ud*omega_d*v - (omega_d**2)*x
def dxdt_1(t, v, x):
    return v
```

```
v0 = 0
x0 = 0
t0 = 0
t_end = 40
dt = (t_end- t0) / 499

t_1 = t0
v_1 = v0
x_1 = x0

t_list_1 = [t0]
v_list_1 = [v0]
x_list_1 = [x0]

def RungeKuttaCoupled(x, y, z, dx, dydx, dzdx):
    k1 = dx*dydx(x, y, z)
    h1 = dx*dzdx(x, y, z)

    k2 = dx*dydx(x+dx/2., y+k1/2., z+h1/2.)
    h2 = dx*dzdx(x+dx/2., y+k1/2., z+h1/2.)

    k3 = dx*dydx(x+dx/2., y+k2/2., z+h2/2.)
    h3 = dx*dzdx(x+dx/2., y+k2/2., z+h2/2.)

    k4 = dx*dydx(x+dx, y+k3, z+h3)
    h4 = dx*dzdx(x+dx, y+k3, z+h3)

    y = y + 1./6.*(k1+2*k2+2*k3+k4)
    z = z + 1./6.*(h1+2*h2+2*h3+h4)
    x = x + dx
    return x, y, z

while t_1 <= t_end:
    t_1, v_1, x_1 = RungeKuttaCoupled(t_1, v_1, x_1, dt, dvdt_1, dxdt_1)
    t_list_1.append(t_1)
    v_list_1.append(v_1)
    x_list_1.append(x_1)

print ("\nUNDER DAMPED TEST CASE: f(t)/m = A*cos(omega_d*t)\n\n")
print ("Please look at the attached pdf file for numerical evidence ")
print ("And explaination for x(t) increases by transient solution or stay in steady state solution")

plt.title("Displacement and velocity for Damped Systems")
plt.plot(t_list_1, v_list_1, label="Velocity", color="red")
```

```
plt.plot(t_list_1, x_list_1, label="Position", color="blue")
plt.xlabel("Time")
plt.ylabel("Velocity/position")
plt.legend(loc="best")
plt.legend()
plt.tight_layout()
plt.grid()
plt.show()
```

```
UNDER DAMPED TEST CASE: f(t)/m = A*cos(omega_d*t)


Please look at the attached pdf file for numerical evidence
And explaination for x(t) increases by transient solution or stay in steady state solution
```




```
In [ ]:
```