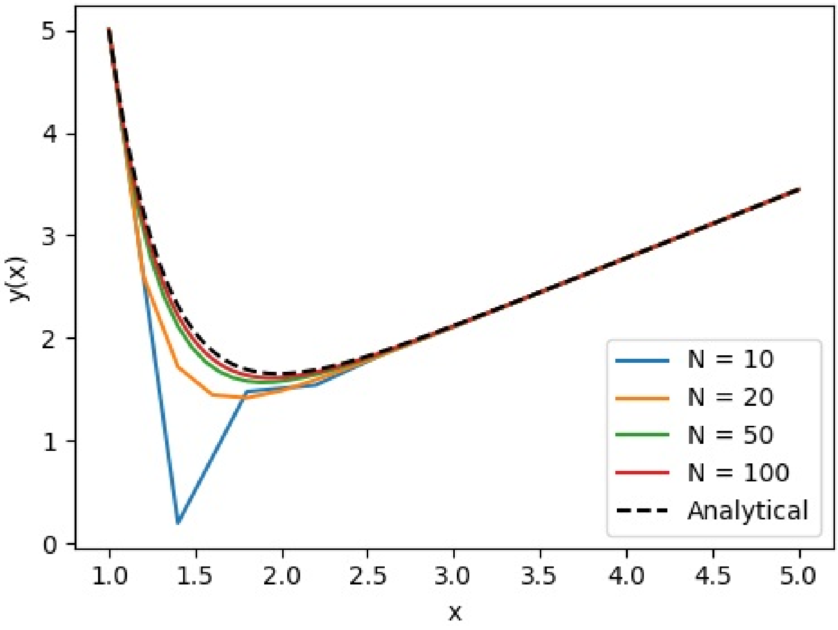
PHYS2160 Introductory Computational Physics
2021/22 Solutions to Exercise 6

1.
```
# euler.py
# This program solves the ODE dy/dx = 2x - 3y + 1 with y(1) = 5.0 using
# Euler's method and plots the results.
# Last update on 1 Feb 2021 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

def f(y, x):
    """ Function to compute the function f(y, x) = 2x - 3y + 1 """
    return 2*x - 3*y + 1

a, b = 1.0, 5.0     # Start and end of the interval

# Use Euler's method to solve the ODE for different number of steps N
# and plot the numerical and analytical solutions on the same graph
fig, ax = plt.subplots()
for N in [10, 20, 50, 100]:
    h = (b - a)/N   # Size of a single step
    y = 5.0         # Initial condition
    xpts = np.linspace(a, b, N+1)
    ynpts = [y]
    for x in xpts[:-1]:
        y += h*f(y, x)
        ynpts.append(y)
    lbl = "N = {}".format(N)
    ax.plot(xpts, ynpts, label=lbl)
yapts = 1/9.0 + 2*xpts/3.0 + 38*np.exp(-3*(xpts-1))/9.0
ax.plot(xpts, yapts, "k-", label="Analytical")
ax.set_xlabel("x")
ax.set_ylabel("y(x)")
ax.legend()
plt.show()
```

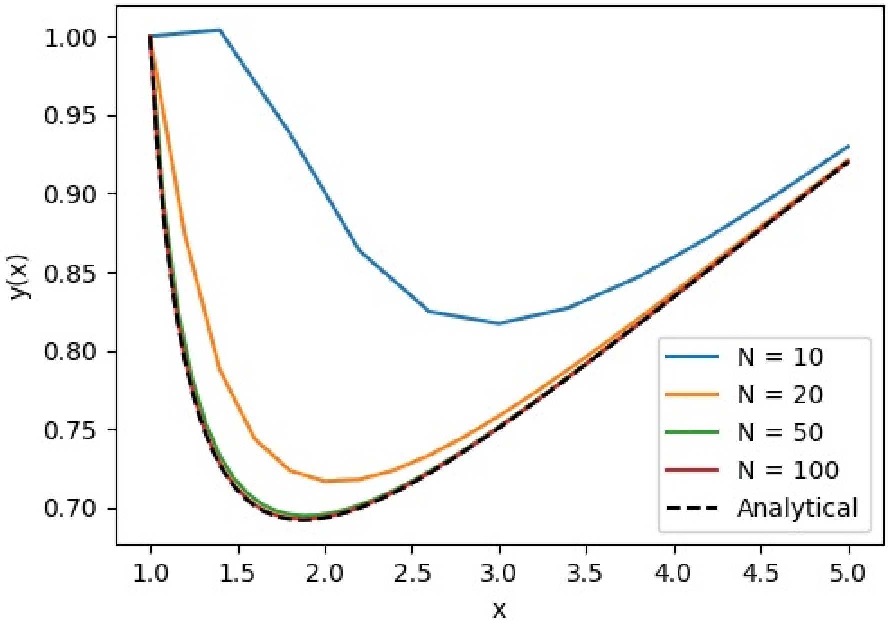
The figure on the top of next page is the output of this program.



2.
```
# rk2.py
# This program solves the ODE dy/dx = (x+1)y/(2x) - 3y^3/x with y(1)
# = 1.0 using the second-order Runge-Kutta method and plots the
# results.
# Last update on 1 Feb 2021 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

def f(y, x):
    """ Function to compute the function f(y, x) = (x+1)y/(2x) -
        3y^3/x """
    return 0.5*(x+1)*y/x - 3*y**3/x

a, b = 1.0, 5.0      # Start and end of the interval

# Use the second-order Runge-Kutta method to solve the ODE for
# different number of steps N and plot the numerical and analytical
# solutions on the same graph
fig, ax = plt.subplots()
for N in [10, 20, 50, 100]:
    h = (b - a)/N    # Size of a single step
    y = 1.0          # Initial condition
    xpts = np.linspace(a, b, N+1)
    ynpts = [y]
    for x in xpts[:-1]:
        k1 = h*f(y, x)
        k2 = h*f(y+0.5*k1, x+0.5*h)
        y += k2
```

```
        ynpts.append(y)
    lbl = "N = {}".format(N)
    ax.plot(xpts, ynpts, label=lbl)
yapts = np.sqrt(xpts/(6 - 5*np.exp(1-xpts)))
ax.plot(xpts, yapts, "k-", label="Analytical")
ax.set_xlabel("x")
ax.set_ylabel("y(x)")
ax.legend()
plt.show()
```

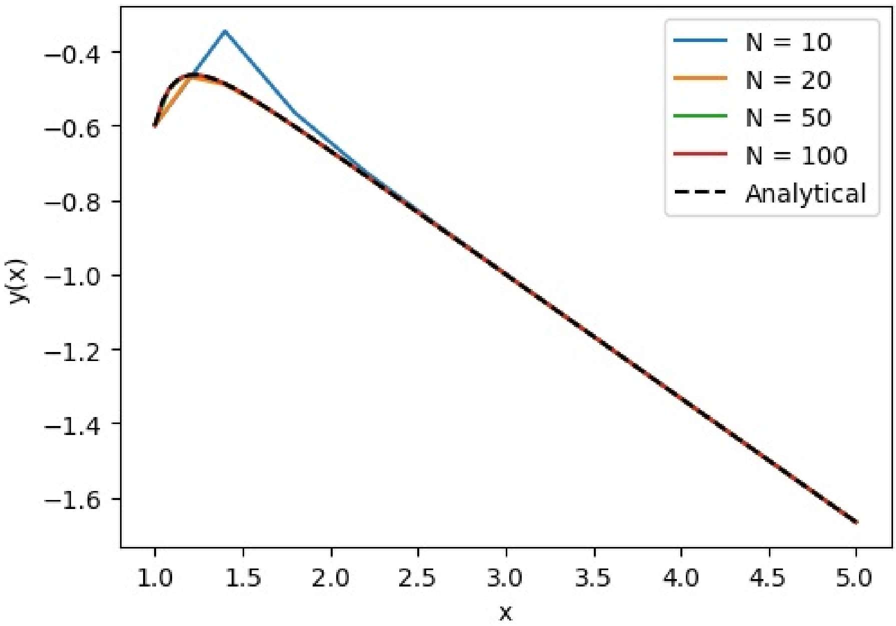
Here is the output of this program:



3.
```
# rk4.py
# This program solves the ODE dy/dx = 6y^2/x^2 - 3y/x - 2 with y(1) =
# -0.6 using the fourth-order Runge-Kutta method and plots the results.
# Last update on 1 Feb 2021 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

def f(y, x):
    """ Function to compute the function f(y, x) = 6y^2/x^2 - 3y/x -
        2 """
    return 6*y**2/x**2 - 3*y/x - 2

a, b = 1.0, 5.0     # Start and end of the interval

# Use the fourth-order Runge-Kutta method to solve the ODE for
# different number of steps N and plot the numerical and analytical
# solutions on the same graph
fig, ax = plt.subplots()
```

```
for N in [10, 20, 50, 100]:
    h = (b - a)/N    # Size of a single step
    y = -0.6         # Initial condition
    xpts = np.linspace(a, b, N+1)
    ynpts = [y]
    for x in xpts[:-1]:
        k1 = h*f(y, x)
        k2 = h*f(y+0.5*k1, x+0.5*h)
        k3 = h*f(y+0.5*k2, x+0.5*h)
        k4 = h*f(y+k3, x+h)
        y += (k1 + 2*k2 + 2*k3 + k4)/6.0
        ynpts.append(y)
    lbl = "N = {}".format(N)
    ax.plot(xpts, ynpts, label=lbl)
yapts = (1 + 2*xpts**8)*xpts/(1 - 6*xpts**8)
ax.plot(xpts, yapts, "k-", label="Analytical")
ax.set_xlabel("x")
ax.set_ylabel("y(x)")
ax.legend()
plt.show()
```

Here is the output of this program:



4.
```
# lorenzrk4.py
# This program solves the Lorenz equations dx/dt = sigma(y - x),
# dy/dt = x(rho - z) - y, and dz/dt = xy - beta z with (x(0), y(0),
# z(0)) = (0, 1, 0) from t = 0 to t = 5s for sigma = 10, rho = 28,
# beta = 8/3 using the fourth-order Runge-Kutta method and then plots
# the results.
# Last update on 4 Apr 2022 by F K Chow
```

```python
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

sigma, rho, beta = 10, 28, 8.0/3.0

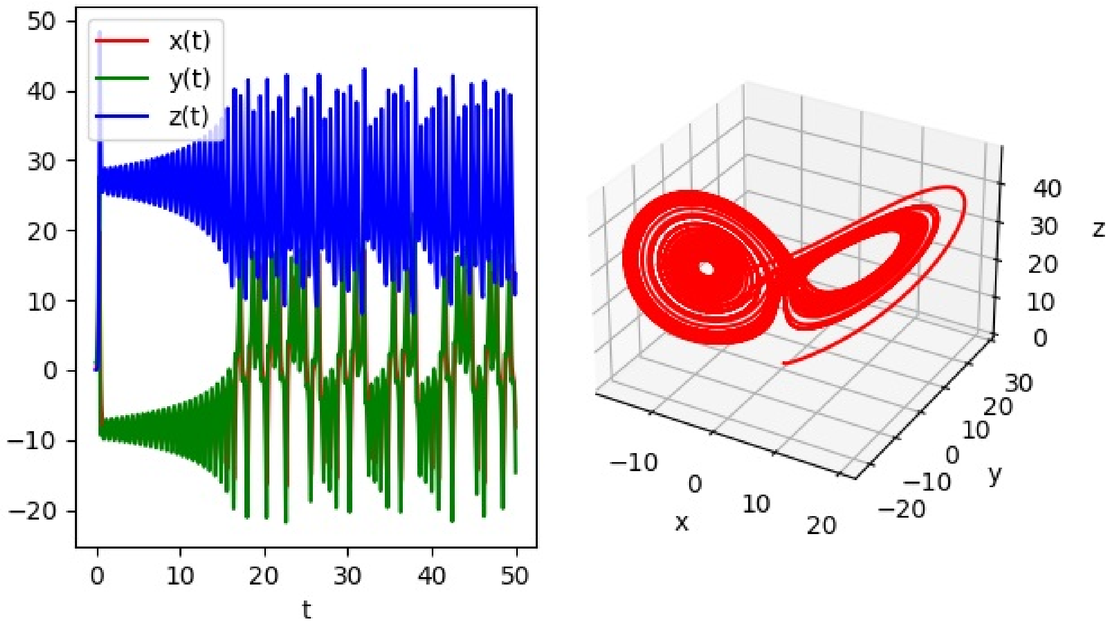
def f(r, t):
    """ Function to compute the function f(r, t) = (fx(r, t),
        fy(r, t), fz(r, t)) where x = r[0], y = r[1], z = r[2] """
    fx = sigma*(r[1] - r[0])
    fy = r[0]*(rho - r[2]) - r[1]
    fz = r[0]*r[1] - beta*r[2]
    return np.array([fx, fy, fz])

a, b = 0.0, 50.0              # Start and end of the interval
h = 0.001                     # Size of a single step
N = (b - a)/h                 # Number of steps
r = np.array([0, 1.0, 0])     # Initial conditions

# Use the fourth-order Runge-Kutta method to solve the Lorenz equations
tpts = np.arange(a, b+h, h)
xpts, ypts, zpts = [r[0]], [r[1]], [r[2]]
for t in tpts[:-1]:
    k1 = h*f(r, t)
    k2 = h*f(r+0.5*k1, t+0.5*h)
    k3 = h*f(r+0.5*k2, t+0.5*h)
    k4 = h*f(r+k3, t+h)
    r += (k1 + 2*k2 + 2*k3 + k4)/6
    xpts.append(r[0])
    ypts.append(r[1])
    zpts.append(r[2])

# Make a plot of x, y, and z as a function of time t as well as a plot
# of z against x and y
fig = plt.figure()
ax1 = fig.add_subplot(121)
ax1.plot(tpts, xpts, "r-", label="x(t)")
ax1.plot(tpts, ypts, "g-", label="y(t)")
ax1.plot(tpts, zpts, "b-", label="z(t)")
ax1.set_xlabel("t")
ax1.legend()
ax2 = fig.add_subplot(122, projection="3d")
ax2.plot(xpts, ypts, zpts, "r-")
ax2.set_xlabel("x")
ax2.set_ylabel("y")
ax2.set_zlabel("z")
fig.tight_layout()
plt.show()
```

Here is the output of this program:




5. ```
fallingrk4.py
This program solves the 2nd order ODE md^2 x/dt^2 = mg - k(dx/dt)^2
with (x(0), x'(0)) = (0, 0) from t = 0 to t = 5s for m = 2kg and k =
0.1kg/m using the fourth-order Runge-Kutta method and then plots the
results.
Last update on 1 Feb 2021 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

m, k, g = 2, 0.1, 9.81

def f(r, t):
    """ Function to compute the function f(r, t) = (fx(r, t),
        fv(r, t)) where x = r[0] and v = x' = r[1] """
    fx = r[1]
    fv = g - (k/m)*r[1]**2
    return np.array([fx, fv])

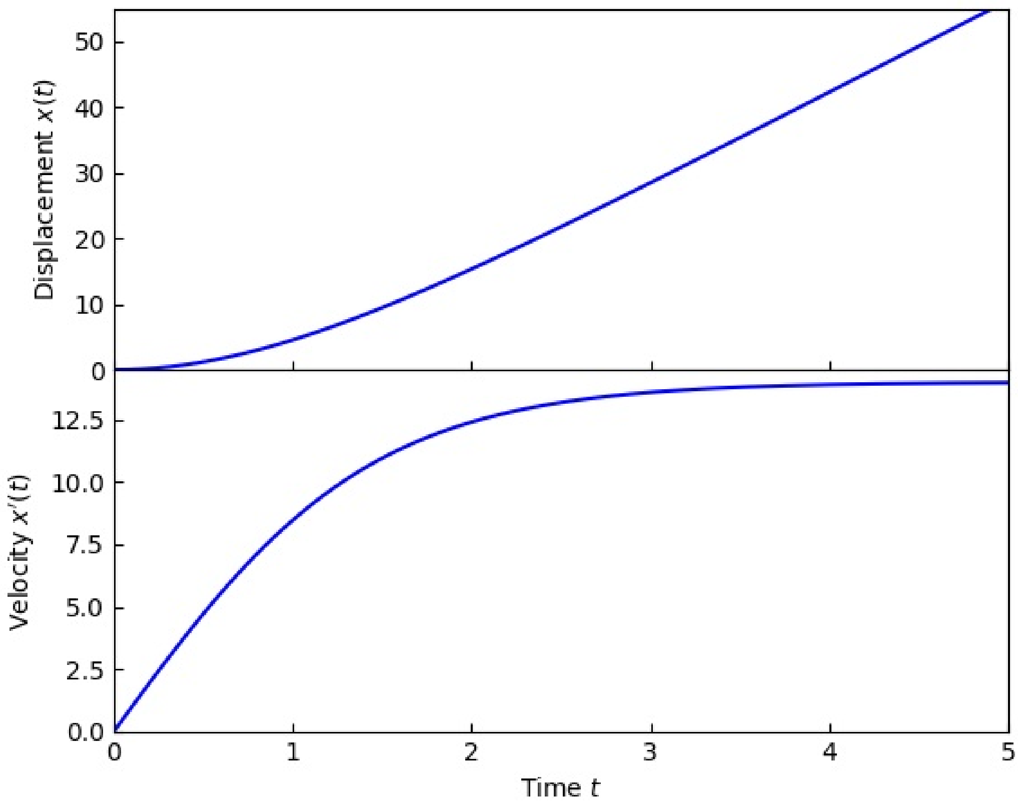
a, b = 0.0, 50.0   # Start and end of the interval
h = 0.001   # Size of a single step
N = (b - a)/h   # Number of steps
r = np.array([0.0, 0.0])   # Initial conditions

# Use the fourth-order Runge-Kutta method to solve the 2nd order ODE
tpts = np.arange(a, b+h, h)
xpts, vpts = [r[0]], [r[1]]
for t in tpts[:-1]:
    k1 = h*f(r, t)
```
```

```
    k2 = h*f(r+0.5*k1, t+0.5*h)
    k3 = h*f(r+0.5*k2, t+0.5*h)
    k4 = h*f(r+k3, t+h)
    r += (k1 + 2*k2 + 2*k3 + k4)/6
    xpts.append(r[0])
    vpts.append(r[1])

# Make a plot of x and x' as a function of time t
fig, ax = plt.subplots(nrows=2, ncols=1)
fig.subplots_adjust(hspace=0)
ax[0].plot(tpts, xpts, "b-")
ax[0].set_ylabel(r"Displacement $x(t)$")
ax[0].set_xlim(0, 5)
ax[0].set_ylim(0, 55)
ax[0].set_xticks(np.linspace(0, 5, 6))
ax[0].set_xticklabels("")
ax[0].tick_params(direction="in")
ax[1].plot(tpts, vpts, "b-")
ax[1].set_xlabel(r"Time $t$")
ax[1].set_ylabel(r"Velocity $x'(t)$")
ax[1].set_xlim(0, 5)
ax[1].set_ylim(0, 14.5)
ax[1].tick_params(direction="in")
plt.show()
```

Here is the output of this program:



6.

```
# nlpenddrk4.py
# This program solves the 2nd order ODE d^2 theta/dt^2 = -(g/l) sin
# (theta) + Ccos(theta)cos(Omega*t) with (theta(0), theta'(0)) = (0, 0)
# from t = 0 to t = 100s for l = 0.1m, C = 2s^(-2), Omega = 5s^(-1)
# using the fourth-order Runge-Kutta method and then plots the results.
# Last update on 1 Feb 2021 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

l, C, Omega, g = 0.1, 2, 5, 9.81

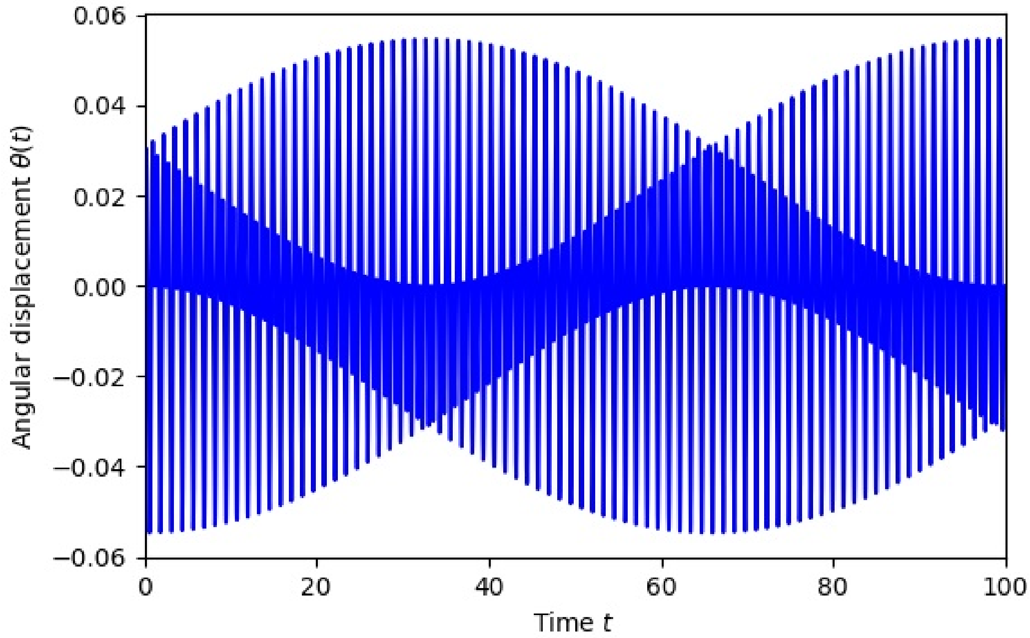
def f(r,t):
    """ Function to compute the function f(r, t) = (ftheta(r, t),
        fomega(r, t)) where theta = r[0] and omega = r[1] """
    ftheta = r[1]
    fomega = -(g/l)*np.sin(r[0]) + C*np.cos(r[0])*np.cos(Omega*t)
    return np.array([ftheta,fomega])

a, b = 0.0, 100.0            # Start and end of the interval
h = 0.001                    # Size of a single step
N = (b - a)/h                # Number of steps
r = np.array([0.0, 0.0])     # Initial conditions

# Use the fourth-order Runge-Kutta method to solve the 2nd order ODE
tpts = np.arange(a, b+h, h)
thetapts = [r[0]]
for t in tpts[:-1]:
    k1 = h*f(r, t)
    k2 = h*f(r+0.5*k1, t+0.5*h)
    k3 = h*f(r+0.5*k2, t+0.5*h)
    k4 = h*f(r+k3, t+h)
    r += (k1 + 2*k2 + 2*k3 + k4)/6
    thetapts.append(r[0])

# Make a plot of theta as a function of time t
fig, ax = plt.subplots()
ax.plot(tpts, thetapts, "b-")
ax.set_xlabel(r"Time $t$")
ax.set_ylabel(r"Angular displacement $\theta(t)$")
ax.set_xlim(0, 100)
plt.show()
```

The figure on the top of next page is the output of this program.



7.
```
# ddoscrk4.py
# This program solves the 2nd order ODE d^2 x/dt^2 + gamma*(dx/dt) +
# omega0^2*x = F0*cos(omega*t) with (x(0), x'(0)) = (10m, 0) from t = 0
# to t = 10s for gamma = 1s^(-1), omega0 = 10rad/s, F0 = 50m/s^2, omega
# = 12rad/s using the fourth-order Runge-Kutta method with adaptive
# step sizes and then plots the results.
# Last update on 11 Apr 2022 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

gamma, omega0, F0, omega = 1, 10, 50, 12

def f(r,t):
    """ Function to compute the function f(r, t) = (fx(r, t),
        fv(r, t)) where x = r[0] and v = x' = r[1] """
    fx = r[1]
    fv = F0*np.cos(omega*t) - gamma*r[1] - omega0**2*r[0]
    return np.array([fx,fv])

def rn(r,t,h):
    """ Function to compute the new value of r using the fourth-order
        Runge-Kutta method with step size h """
    k1 = h*f(r, t)
    k2 = h*f(r+0.5*k1, t+0.5*h)
    k3 = h*f(r+0.5*k2, t+0.5*h)
    k4 = h*f(r+k3, t+h)
    return r + (k1 + 2*k2 + 2*k3 + k4)/6

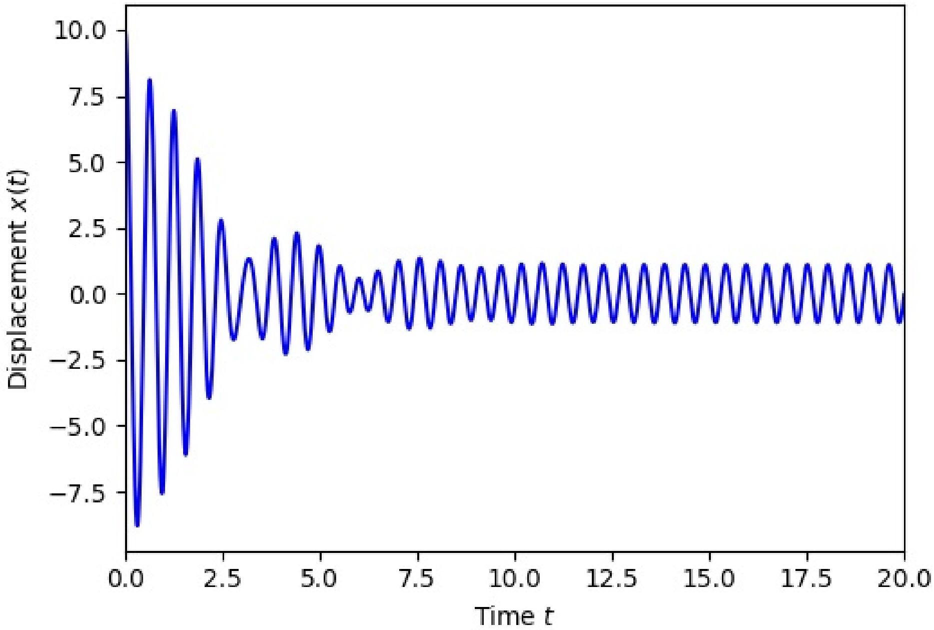
a, b = 0.0, 20.0             # Start and end of the interval
```

```
N = 1000                      # Number of steps
h = (b - a)/N                 # Size of a single step
r = np.array([10.0, 0.0])     # Initial conditions
delta = 1e-6                  # Target accuracy for each step

# Use the fourth-order Runge-Kutta method with adaptive step sizes to
# solve the 2nd order ODE
t = a
tpts, xpts = [a], [r[0]]
while t < b:
    while True:
        r0 = rn(r, t, h)
        r1 = rn(r0, t+h, h)
        r2 = rn(r, t, 2*h)
        if np.isclose(r1[0], r2[0], rtol=0) == 0:
            rho = 30*h*delta/abs(r1[0] - r2[0])
        else:
            rho = 10000
        if rho >= 1:
            tpts.extend([t+h, t+2*h])
            xpts.extend([r0[0], r1[0]])
            t += 2*h
            r = r1
            h *= min(rho**0.25, 2.0)
            break
        h *= rho**0.25

# Make a plot of x as a function of time t
fig, ax = plt.subplots()
ax.plot(tpts, xpts, "b-")
ax.set_xlabel(r"Time $t$")
ax.set_ylabel(r"Displacement $x(t)$")
ax.set_xlim(0, 10)
plt.show()
```

The figure on the top of next page is the output of this program.



8.
```
# shellrk4.py
# This program solves the system of 2nd order ODEs md^2 x/dt^2 =
# -kv(dx/dt) and md^2 y/dt^2 = mg - kv(dy/dt) until y(t) = 0 for
# different values of the launch angle theta using the fourth-order
# Runge-Kutta method and then plots all these trajectories on the same
# graph. Here we assume k/m = 4e-5m^(-1), (x(0), y(0)) = (0, 0), and
# initial speed v0 = 700m/s.  All quantities are expressed in Si units.
# Last update on 7 Jan 2022 by F K Chow

import matplotlib.pyplot as plt
import numpy as np

koverm, g = 4e-5, 9.81

def f(r, t):
    """ Function to compute the function f(r, t) = (fx(r, t),
        fvx(r, t), fy(r, t), fvy(r, t)) where x = r[0], vx = x' = r[1],
        y = r[2], vy = y' = r[3] """
    fx = r[1]
    fvx = -koverm*np.sqrt(r[1]**2+r[3]**2)*r[1]
    fy = r[3]
    fvy = -g - koverm*np.sqrt(r[1]**2+r[3]**2)*r[3]
    return np.array([fx, fvx, fy, fvy])

t0 = 0.0    # Start of the interval
h = 0.01    # Size of a single step
v0 = 700    # Initial speed

# Use the fourth-order Runge-Kutta method to solve the system of 2nd order
```

```
# ODEs for different values of the launch angle theta and plot all these
# trajectories on the same graph
fig, ax = plt.subplots()
for theta in np.arange(30, 60, 5):
    r = np.array([0.0, v0*np.cos(np.deg2rad(theta)), 0.0,
                  v0*np.sin(np.deg2rad(theta))])    # Initial conditions
    t = t0
    xpts, ypts = [r[0]], [r[2]]
    while True:
        k1 = h*f(r, t)
        k2 = h*f(r+0.5*k1, t+0.5*h)
        k3 = h*f(r+0.5*k2, t+0.5*h)
        k4 = h*f(r+k3, t+h)
        r += (k1 + 2*k2 + 2*k3 + k4)/6
        if r[2] <= 0:
            break
        t += h
        xpts.append(r[0])
        ypts.append(r[2])
    lbl = r"$\theta$ = {:d}".format(theta) + chr(176)
    ax.plot(np.array(xpts)/1000, np.array(ypts)/1000, label=lbl)
ax.set_xlabel(r"$x(t)$/km")
ax.set_ylabel(r"$y(t)$/km")
ax.set_xlim(0, 22.5)
ax.set_ylim(0, 10)
ax.legend()
plt.show()
```

Here is the output of this program: