CHAPTER 8

NUMERICAL METHODS

8.1 Second Order Runge-Kutta Algorithm (midpoint)

The second order Runge-Kutta (RK2) algorithm (also known as the **modified Euler's method**) uses an estimate for the derivative at the midpoint of the interval between t_0 and t_0+h, i.e. at $t_0+h/2$. This would result in a better approximation for the function at t_0+h, than using the derivative at t_0 (which is usually done in Euler's method, also known as *first order Runge-Kutta algorithm*).

Let us consider a first order ordinary differential equation defined by

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

with the initial condition $y(t_0) = y_0$ at $t = t_0$. For RK2 method, we'll start from this initial point and progess by one time step h, repetitively, using the following steps:

$$\begin{aligned} k_1 &= f(y_0,t_0) && \text{estimate of derivative at } t=t_0 \\ y_1(t_0+\frac{h}{2}) &= y_0+k_1\frac{h}{2} && \text{intermediate estimate of function at } t=t_0+\tfrac{h}{2} \\ k_2 &= f(y_1(t_0+\frac{h}{2}),t_0+\frac{h}{2}) && \text{estimate of slope at } t=t_0+\tfrac{h}{2} \\ y(t_0+h) &= y_0+k_2h && \text{estimate of } y(t_0+h) \end{aligned}$$

Exercise 8.1: Solve the first order linear differential equation

$$\frac{dy}{dt} + 2y = 0$$

with the initial condition $y(0) = 3$. Plot your result and compare that with its analytical solution $y(t) = 3e^{-2t}$.

```
def rk2( f, y0, t ):
    """
    Second-order Runge-Kutta method to solve dy/dx = f(y,t) with initial value y(t[0]) = y0.

    USAGE:
        y = rk2(f, y0, t)

    INPUT:
        f    - function of y and t equal to dy/dt. y may be a list or a
               NumPy array. In this case f must return a NumPy array with
               the same dimension as y.
        y0   - the initial condition(s). Specifies the value of y when
```

```
            t = t[0]. Can be either a scalar or a list or NumPy array
            if a system of equations is being solved.
      t     - list or NumPy array of t values to compute solution at.
            t[0] is the the initial condition point, and the difference
            h=t[i+1]-t[i] determines the step size h.

   OUTPUT:
      y     - NumPy array containing solution values corresponding to each
            entry in t array. If a system is being solved, y will be
            an array of arrays.
   """

   n = len( t )
   y = np.array( [ y0 ]*n )
   for i in range( n - 1 ):
      h = t[i+1] - t[i]
      k1 = f( y[i], t[i] )
      k2 = f( y[i] + k1*h/2.0, t[i] + h/2.0 )
      y[i+1] = y[i] + k2*h

   return y


""" DRIVER CODE """

# define your dy/dt here
def f( y, t ):
   return -2.*y

import numpy as np
import matplotlib.pyplot as plt

# Initial condition(s)
t0 = 0.0
y0 = 3.0

tf = 3.0 # final value of t, User-supplied
h = 0.1 # step-size, User-supplied

t = t0
ts = [t]

while t<tf:
   t += h
   ts.append(t)
ts = np.array(ts)

y = rk2(f, y0, ts) # calling the solver routine

plt.plot(ts, y, color = 'red', ls = 'solid', label='numerical (RK2)')

tp = np.linspace( t0, tf, 30 ) # For plotting the exact solution
yexact = 3.*np.exp(- 2.*tp) # Exact solution (in general we won't know this)
plt.plot(tp, yexact, 'g^', label='exact')
plt.xlabel('t')
plt.ylabel('y(t)')
plt.legend(loc='best')
plt.show()
```

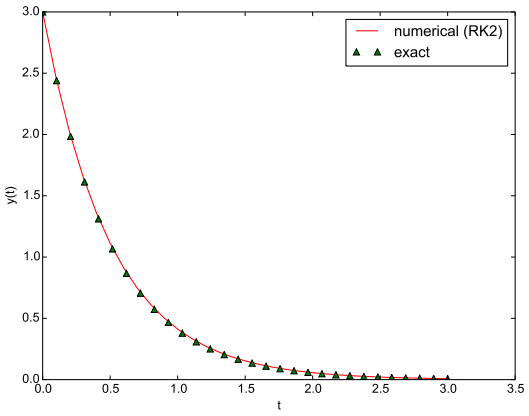

Figure 8.1

Exercise 8.2: Solve the first order nonlinear differential equation

$$\frac{dy}{dt} - y(1-2t) = 0$$

with the initial condition $y(0) = 1$. Plot your result and compare that with its analytical solution $y(t) = e^{t-t^2}$.

```
def rk2( f, y0, t ):
   """
   Second-order Runge-Kutta method to solve dy/dx = f(y,t) with initial value y(t[0]) = y0.

   USAGE:
      y = rk2(f, y0, t)

   INPUT:
      f    - function of y and t equal to dy/dt. y may be a list or a
             NumPy array. In this case f must return a NumPy array with
             the same dimension as y.
      y0   - the initial condition(s). Specifies the value of y when
             t = t[0]. Can be either a scalar or a list or NumPy array
             if a system of equations is being solved.
      t    - list or NumPy array of t values to compute solution at.
             t[0] is the the initial condition point, and the difference
             h=t[i+1]-t[i] determines the step size h.

   OUTPUT:
      y    - NumPy array containing solution values corresponding to each
             entry in t array. If a system is being solved, y will be
             an array of arrays.
   """

   n = len( t )
   y = np.array( [ y0 ]*n )
   for i in range( n - 1 ):
      h = t[i+1] - t[i]
      k1 = f( y[i], t[i] )
      k2 = f( y[i] + k1*h/2.0, t[i] + h/2.0 )
      y[i+1] = y[i] + k2*h

   return y
```

```
""" DRIVER CODE """

# define your dy/dt here
def f( y, t ):
   return y*(1-2*t)

import numpy as np
import matplotlib.pyplot as plt

# Initial condition(s)
t0 = 0.0
y0 = 1.0

tf = 5.0 # final value of t, User-supplied
h = 0.1 # step-size, User-supplied

t = t0
ts = [t]

while t<tf:
   t += h
   ts.append(t)
ts = np.array(ts)

y = rk2(f, y0, ts) # calling the solver routine

plt.plot(ts, y, color = 'red', ls = 'solid', label='numerical (RK2)')

tp = np.linspace( t0, tf, 30 ) # For plotting the exact solution
yexact = np.exp(tp - tp**2.) # Exact solution (in general we won't know this)
plt.plot(tp, yexact, 'g^', label='exact')
plt.xlabel('t')
plt.ylabel('y(t)')
plt.legend(loc='best')
plt.show()
```

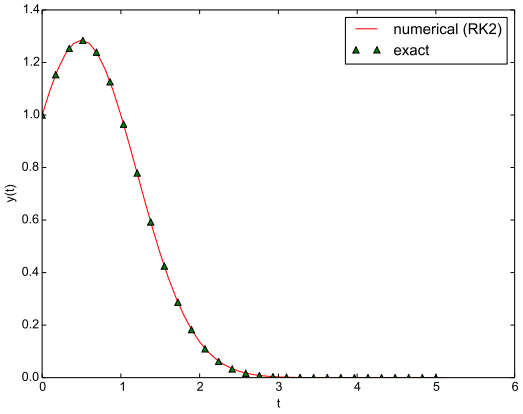

Figure 8.2

8.2 Fourth Order Runge-Kutta Algorithm

In the last section we've seen that, using two estimates of the slope (i.e., Second Order Runge Kutta; using slopes at the beginning and midpoint of the time step) gave an approximation with greater accuracy than using just a single slope (i.e., First Order Runge Kutta; using only the slope at the beginning of the interval). Thus, it seems reasonable to assume that using even more estimates of the slope would result in even more accuracy. In fact, it turns out to be true, and leads to higher-order Runge-Kutta methods. We shall restrict ourselves to the most commonly used Fourth Order Runge-Kutta (RK4) technique, which uses four approximations to the slope.

Let us again consider a first order ordinary differential equation defined by

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

with the initial condition $y(t_0) = y_0$ at $t = t_0$. For RK4 method, we'll start from this initial point and progess by one time step h, repetitively, using the following steps:

$$\begin{aligned}
k_1 &= f(y_0,t_0) && \text{estimate of derivative at } t = t_0 \\
y_1(t_0+\frac{h}{2}) &= y_0 + k_1\frac{h}{2} && \text{intermediate estimate of function at } t = t_0+\tfrac{h}{2} \\
k_2 &= f(y_1(t_0+\frac{h}{2}),t_0+\frac{h}{2}) && \text{estimate of slope at } t = t_0+\tfrac{h}{2} \\
y_2(t_0+\frac{h}{2}) &= y_0 + k_2\frac{h}{2} && \text{another intermediate estimate of function at } t = t_0+\tfrac{h}{2} \\
k_3 &= f(y_2(t_0+\frac{h}{2}),t_0+\frac{h}{2}) && \text{another estimate of slope at } t = t_0+\tfrac{h}{2} \\
y_3(t_0+h) &= y_0 + k_3 h && \text{an estimate of function at } t = t_0+h \\
k_4 &= f(y_3(t_0+h),t_0+h) && \text{estimate of slope at } t = t_0+h \\
y(t_0+h) &= y_0 + \frac{k_1+2k_2+2k_3+k_4}{6}h && \text{estimate of } y(t_0+h)
\end{aligned}$$

Exercise 8.3: Solve the first order differential equation using fourth order Runge-Kutta method

$$\frac{dy}{dx} = e^{-2x} - 3y$$

with the initial condition $y(0) = 5$. Plot your result and compare that with its analytical solution $y(t) = e^{-2x} + 4e^{-3x}$. Also compare your result with that obtained using `scipy.integrate.odeint()`.

```
def rk4( f, y0, t ):
    """
    Fourth-order Runge-Kutta method to solve dy/dx = f(y,t) with initial value y(t[0]) = y0.

    USAGE:
        y = rk4(f, y0, t)

    INPUT:
        f    - function of y and t equal to dy/dt. y may be a list or a
               NumPy array. In this case f must return a NumPy array with
               the same dimension as y.
        y0   - the initial condition(s). Specifies the value of y when
               t = t[0]. Can be either a scalar or a list or NumPy array
               if a system of equations is being solved.
        t    - list or NumPy array of t values to compute solution at.
               t[0] is the the initial condition point, and the difference
               h=t[i+1]-t[i] determines the step size h.

    OUTPUT:
        y    - NumPy array containing solution values corresponding to each
```

```
            entry in t array. If a system is being solved, y will be
            an array of arrays.
    """

    n = len( t )
    y = np.array( [ y0 ] * n )
    for i in range( n - 1 ):
        h = t[i+1] - t[i]
        k1 = f( y[i], t[i] )
        k2 = f( y[i] + 0.5 * k1 * h, t[i] + 0.5 * h )
        k3 = f( y[i] + 0.5 * k2 * h, t[i] + 0.5 * h )
        k4 = f( y[i] + k3 * h, t[i+1] )
        y[i+1] = y[i] + ( k1 + 2.0 * ( k2 + k3 ) + k4 ) / 6.0 * h

    return y


""" DRIVER CODE """

# define your dy/dt here
def f( y, t ):
    return np.exp(-2.*t)-3.*y

import numpy as np
import matplotlib.pyplot as plt

from scipy.integrate import odeint

# Initial condition(s)
t0 = 0.0
y0 = 5.0

tf = 3.0 # final value of t, User-supplied
h = 0.1 # step-size, User-supplied

t = t0
ts = [t]

while t<tf:
    t += h
    ts.append(t)
ts = np.array(ts)

y = rk4(f, y0, ts) # calling the solver routine

plt.plot(ts, y, color = 'red', ls = 'solid', label='numerical (RK4)')

#################
# From scipy
yp = odeint(f, y0, ts)
plt.plot(ts, yp, color='black', ls = 'dashed', label='numerical (scipy)')
#################

tp = np.linspace( t0, tf, 30 ) # For plotting the exact solution
yexact = np.exp(- 2.*tp) + 4.*np.exp(- 3.*tp) # Exact solution (in general we won't know this)
plt.plot(tp, yexact, 'g^', label='exact')
plt.xlabel('t')
plt.ylabel('y(t)')
plt.legend(loc='best')
plt.show()
```

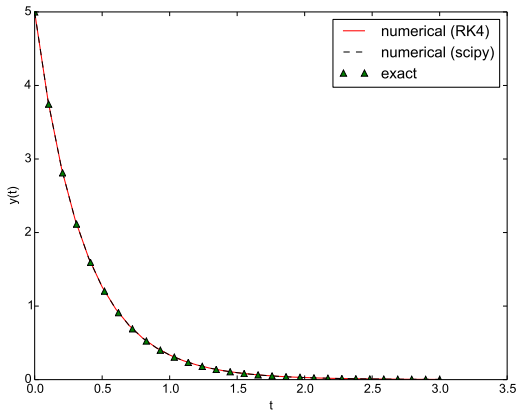

Figure 8.3

8.2.1 RK4 to solve 2nd order differential equation

Note that, in order to solve higher order differential equation, we must first convert it to a system of first order equations and then apply the same numerical method as we did earlier for solving first oder differential equation.

Exercise 8.4: Solve the second order differential equation for the damped harmonic oscillator using fourth order Runge-Kutta method

$$\frac{d^2y}{dt^2} + \lambda \frac{dy}{dt} + ky = 0$$

with the initial condition $y(0) = 0$ and $\frac{dy}{dt}(0) = 1$. Choose the parameters as $\lambda = 0.5$ and $k = 2$. Plot your result as y vs. t. Also compare your result with that obtained using `scipy.integrate.odeint()`.

```
def rk4( f, y0, t ):
    """
    Fourth-order Runge-Kutta method to solve dy/dx = f(y,t) with initial value y(t[0]) = y0.

    USAGE:
        y = rk4(f, y0, t)

    INPUT:
        f    - function of y and t equal to dy/dt. y may be a list or a
               NumPy array. In this case f must return a NumPy array with
               the same dimension as y.
        y0   - the initial condition(s). Specifies the value of y when
               t = t[0]. Can be either a scalar or a list or NumPy array
               if a system of equations is being solved.
        t    - list or NumPy array of t values to compute solution at.
               t[0] is the the initial condition point, and the difference
               h=t[i+1]-t[i] determines the step size h.

    OUTPUT:
        y    - NumPy array containing solution values corresponding to each
               entry in t array. If a system is being solved, y will be
               an array of arrays.
    """

    n = len( t )
    y = np.array( [ y0 ] * n )
    for i in range( n - 1 ):
```

```
      h = t[i+1] - t[i]
      k1 = f( y[i], t[i] )
      k2 = f( y[i] + 0.5 * k1 * h, t[i] + 0.5 * h )
      k3 = f( y[i] + 0.5 * k2 * h, t[i] + 0.5 * h )
      k4 = f( y[i] + k3 * h, t[i+1] )
      y[i+1] = y[i] + ( k1 + 2.0 * ( k2 + k3 ) + k4 ) / 6.0 * h

   return y


""" DRIVER CODE """
# Solve d2x/dt2 + lam*dx/dt + k*x = 0
# Two first order coupled equations
# dx/dt = y
# dy/dt = yprime = - lam*y - k*x

# define your f(y, t) here
# Note that, now y is an array with y[0] = x and y[1] = y
# This function should return a 1x2 array with the first entry
# equal to y and the second entry equal to yprime.


def f( y, t ):
   lam, k = 0.5, 2.0 # Parameters
   yprime = - lam*y[1] - k*y[0]
   return np.array( [y[1], yprime] )

import numpy as np
import matplotlib.pyplot as plt

from scipy.integrate import odeint

# Initial condition(s)
t0 = 0.0
y0 = [0.0, 1.0] # # In this case y0 will be an array or list with y[0] = y_0 and y[1] = yprime_0

tf = 20.0 # final value of t, User-supplied
h = 0.1 # step-size, User-supplied

t = t0
ts = [t]

while t<tf:
   t += h
   ts.append(t)
ts = np.array(ts)

y = rk4(f, y0, ts) # calling the solver routine
# In this case y will be an array of arrays with y[:,0] = y and y[:,1] = yprime

plt.plot(ts, y[:,0], color='red', ls = 'solid', label='numerical (RK4)')

#################
# From scipy
yp = odeint(f, y0, ts)
plt.plot(ts, yp[:,0], color='black', ls = 'dashed', label='numerical (scipy)')
#################

plt.axhline()
plt.xlabel('t')
```

```
plt.ylabel('y(t)')
plt.legend(loc='best')
plt.show()
```

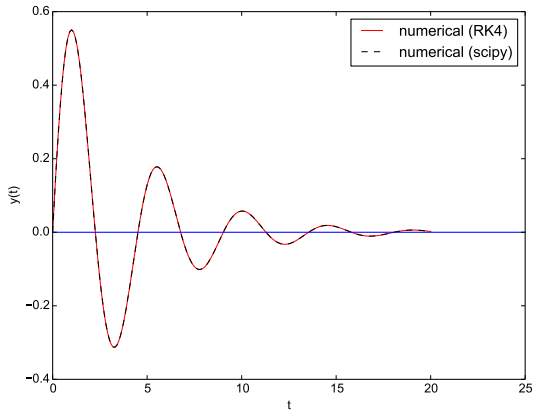

Figure 8.4

Exercise 8.5: Solve the second order differential equation for the periodically undamped ($\lambda = 0$) forced harmonic oscillator using fourth order Runge-Kutta method

$$\frac{d^2y}{dt^2} + ky = F cos(\omega t)$$

with the initial condition $y(0) = 0$ and $\frac{dy}{dt}(0) = 0$. Choose the parameters as $k = \omega_0^2 = 9$, $F = 1$. Plot your result as y vs. t for different $\omega = 3.5$, 3.3, 3.1 and 3 to see the beats and resonance. Also compare your results with those obtained using `scipy.integrate.odeint()`.

```
def rk4( f, y0, t ):
    """
    Fourth-order Runge-Kutta method to solve dy/dx = f(y,t) with initial value y(t[0]) = y0.

    USAGE:
        y = rk4(f, y0, t)

    INPUT:
        f    - function of y and t equal to dy/dt. y may be a list or a
               NumPy array. In this case f must return a NumPy array with
               the same dimension as y.
        y0   - the initial condition(s). Specifies the value of y when
               t = t[0]. Can be either a scalar or a list or NumPy array
               if a system of equations is being solved.
        t    - list or NumPy array of t values to compute solution at.
               t[0] is the the initial condition point, and the difference
               h=t[i+1]-t[i] determines the step size h.

    OUTPUT:
        y    - NumPy array containing solution values corresponding to each
               entry in t array. If a system is being solved, y will be
               an array of arrays.
    """

    n = len( t )
    y = np.array( [ y0 ] * n )
    for i in range( n - 1 ):
        h = t[i+1] - t[i]
```

```
      k1 = f( y[i], t[i] )
      k2 = f( y[i] + 0.5 * k1 * h, t[i] + 0.5 * h )
      k3 = f( y[i] + 0.5 * k2 * h, t[i] + 0.5 * h )
      k4 = f( y[i] + k3 * h, t[i+1] )
      y[i+1] = y[i] + ( k1 + 2.0 * ( k2 + k3 ) + k4 ) / 6.0 * h

   return y


""" DRIVER CODE """
# Solve d2x/dt2 + lam*dx/dt + k*x = F*cos(w*t)
# Two first order coupled equations
# dx/dt = y
# dy/dt = yprime = F*cos(w*t) - lam*y - k*x

# define your f(y, t) here
# Note that, now y is an array with y[0] = x and y[1] = y
# This function should return a 1x2 array with the first entry
# equal to y and the second entry equal to yprime.


def f( y, t ):
   yprime = F*np.cos(w*t) - lam*y[1] - k*y[0]
   return np.array( [y[1], yprime] )

import numpy as np
import matplotlib.pyplot as plt

from scipy.integrate import odeint

lam, k, F, w = 0.0, 9.0, 1.0, 3.5 # Parameters (compare with Fig 1 of
     http://math.colgate.edu/~wweckesser/math308Fall02/handouts/ForcedHarmonicOsc.pdf)

# Initial condition(s)
t0 = 0.0
y0 = [0.0, 0.0] # # In this case y0 will be an array or list with y[0] = y_0 and y[1] = yprime_0

tf = 50.0 # final value of t, User-supplied
h = 0.1 # step-size, User-supplied

t = t0
ts = [t]

while t<tf:
   t += h
   ts.append(t)
ts = np.array(ts)

y = rk4(f, y0, ts) # calling the solver routine
# In this case y will be an array of arrays with y[:,0] = y and y[:,1] = yprime

plt.title('$F=1$, $\omega_0=\sqrt{k}=3$, $\omega=3.5$')
plt.plot(ts, y[:,0], color = 'red', ls = 'solid', label='numerical (RK4)')

#################
# From scipy
yp = odeint(f, y0, ts)
plt.plot(ts, yp[:,0], color='black', ls = 'dashed', label='numerical (scipy)')
#################
```

```
plt.axhline()
plt.xlabel('t')
plt.ylabel('y(t)')
plt.legend(loc='best')
plt.show()
```

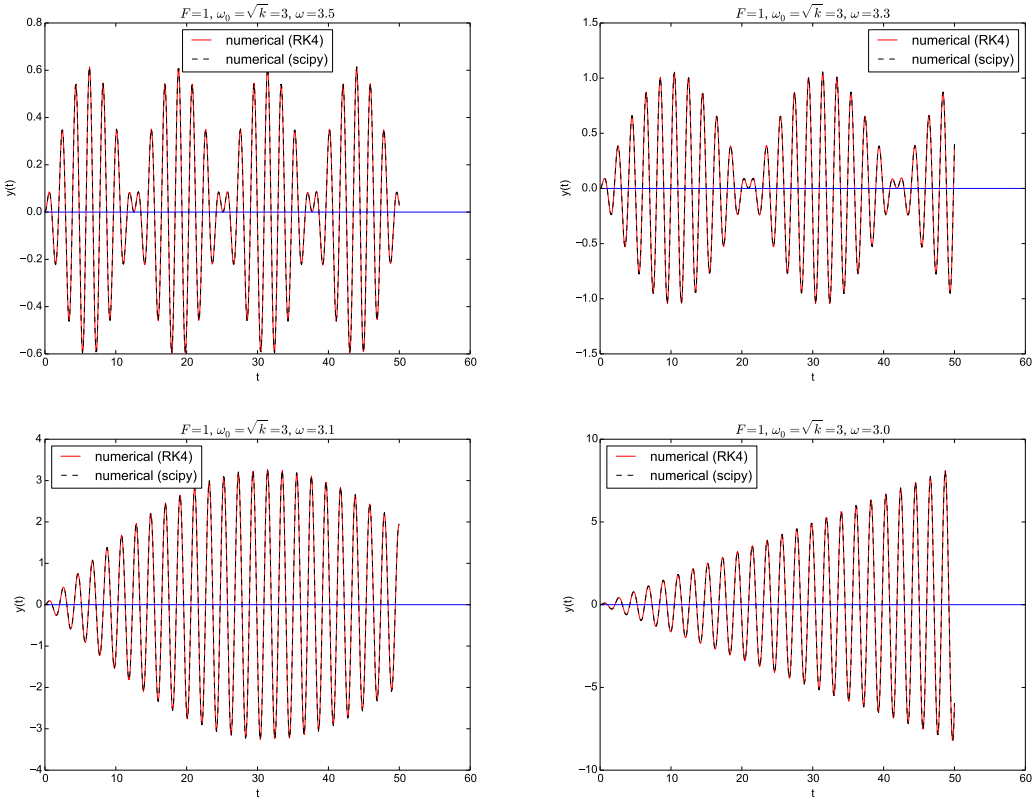

Figure 8.5

Exercise 8.6: Consider an LCR circuit where an EMF (an applied voltage V_a) is connected across the series combination of a switch, a resistor R, an inductor L, and a capacitor C. For this circuit, we can obtain a single second order differential equation as:

$$L\frac{d^2I}{dt^2}+R\frac{dI}{dt}+\frac{I}{C}=\frac{dV_a}{dt}$$

which can be decomposed into two first order differential equations as:

$$\frac{dI}{dt}=\frac{1}{L}\left[V_a-IR-q/C\right]$$
$$\frac{dq}{dt}=I$$

where I is the current accross the circuit and q is the charge on the capacitor. Numerically solve the equation assuming the initial conditions, $I=0$ and $q=1$ at time $t=0$ for two special cases:
1. Free oscillations of the LCR circuit when $V_a=0$ (parameters: $R=0.2$ ohm, $L=2$ henry, $C=2$ farad)
2. Forced oscillations of the LCR circuit with $V_a=V_0\sin\omega t$ (parameters: $R=1$ ohm, $L=8$ henry, $C=8$ farad, $V_0=0.7$, $\omega=0.25$).
Also compare your results with those obtained using `scipy.integrate.odeint()`.

For details of this exercise, see - `http://www.physnet.org/modules/pdf_modules/m351.pdf`

```
def rk4( f, y0, t ):
    """
```

```
   Fourth-order Runge-Kutta method to solve dy/dx = f(y,t) with initial value y(t[0]) = y0.

   USAGE:
      y = rk4(f, y0, t)

   INPUT:
      f    - function of y and t equal to dy/dt. y may be a list or a
             NumPy array. In this case f must return a NumPy array with
             the same dimension as y.
      y0   - the initial condition(s). Specifies the value of y when
             t = t[0]. Can be either a scalar or a list or NumPy array
             if a system of equations is being solved.
      t    - list or NumPy array of t values to compute solution at.
             t[0] is the the initial condition point, and the difference
             h=t[i+1]-t[i] determines the step size h.

   OUTPUT:
      y    - NumPy array containing solution values corresponding to each
             entry in t array. If a system is being solved, y will be
             an array of arrays.
   """

   n = len( t )
   y = np.array( [ y0 ] * n )
   for i in range( n - 1 ):
      h = t[i+1] - t[i]
      k1 = f( y[i], t[i] )
      k2 = f( y[i] + 0.5 * k1 * h, t[i] + 0.5 * h )
      k3 = f( y[i] + 0.5 * k2 * h, t[i] + 0.5 * h )
      k4 = f( y[i] + k3 * h, t[i+1] )
      y[i+1] = y[i] + ( k1 + 2.0 * ( k2 + k3 ) + k4 ) / 6.0 * h

   return y


""" DRIVER CODE """
# Solve L*d2I/dt2 + R*dI/dt + I/C = dVa/dt
# Two first order coupled equations
# dq/dt = I or dx/dt = y
# dI/dt = 1/L * [Va - IR - q/C] or dy/dt = yprime = 1/L * [Va - yR - x/C]

# define your f(y, t) here
# Note that, now y is an array with y[0] = x and y[1] = y
# This function should return a 1x2 array with the first entry
# equal to y and the second entry equal to yprime.


def f( y, t ):
   Va = V0*np.sin(omega*t)
   yprime = 1.0/L * (Va - R*y[1] - y[0]/C)
   return np.array( [y[1], yprime] )

import numpy as np
import matplotlib.pyplot as plt

from scipy.integrate import odeint

R, L, C, V0, omega = 1.0, 8.0, 8.0, 0.7, 0.25 # Parameters (compare with
     http://www.physnet.org/modules/pdf_modules/m351.pdf)
```

```
# Initial condition(s)
t0 = 0.0
y0 = [1.0, 0.0] # # In this case y0 will be an array or list with y[0] = q(0) = 1 and y[1] = y(0) = 0 at t = 0

tf = 55.0 # final value of t, User-supplied
h = 0.1 # step-size, User-supplied

t = t0
ts = [t]

while t<tf:
   t += h
   ts.append(t)
ts = np.array(ts)

y = rk4(f, y0, ts) # calling the solver routine
# In this case y will be an array of arrays with y[:,0] = y and y[:,1] = yprime

plt.subplots(2,2,figsize=(10,4))

plt.subplot(1,2,1)
plt.title('current vs. time (forced oscillations)')
plt.plot(ts, y[:,1], color = 'red', ls = 'solid', label='numerical (RK4)')
################
# From scipy
yp = odeint(f, y0, ts)
plt.plot(ts, yp[:,1], color='black', ls = 'dashed', label='numerical (scipy)')
################
plt.axhline()
plt.xlabel('$t$')
plt.ylabel('$I(t)$')
plt.legend(loc='best')

plt.subplot(1,2,2)
plt.title('charge vs. time (forced oscillations)')
plt.plot(ts, y[:,0], color = 'red', ls = 'solid', label='numerical (RK4)')
################
# From scipy
yp = odeint(f, y0, ts)
plt.plot(ts, yp[:,0], color='black', ls = 'dashed', label='numerical (scipy)')
################
#damping_factor = np.exp(-(R/(2.0*L)*ts))
#plt.plot(ts, damping_factor, color = 'green', ls = 'dotted', label='damping factor $= e^{-(R/2L)t}$ (exact)')
plt.axhline()
plt.xlabel('$t$')
plt.ylabel('$q(t)$')
plt.legend(loc='best')
plt.show()
```

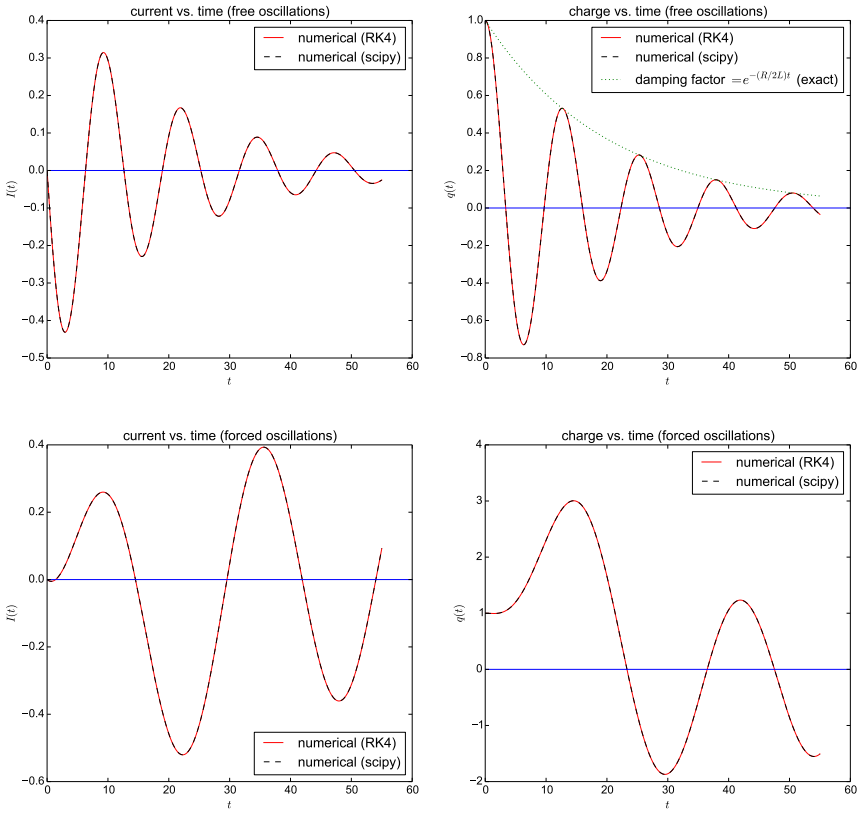
current vs. time (free oscillations)
numerical (RK4)
numerical (scipy)
$I(t)$
t
charge vs. time (free oscillations)
numerical (RK4)
numerical (scipy)
damping factor $=e^{-(R/2L)t}$ (exact)
$q(t)$
t
current vs. time (forced oscillations)
numerical (RK4)
numerical (scipy)
$I(t)$
t
charge vs. time (forced oscillations)
numerical (RK4)
numerical (scipy)
$q(t)$
t

Figure 8.6