7 Initial Value Problems

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

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 be transformed into n first-order equations. Using the notation

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

the equivalent first-order equations are

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

The solution now requires the knowledge of 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

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

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 because both auxiliary conditions imposed on the solution are given at $x = 0$. In contrast,

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

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. Boundary value problems, which are more difficult to solve, are discussed in the next chapter. We also make extensive use of vector notation, which allows us to 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 Euler's Method

Euler's method of solution is conceptually simple. Its basis is the truncated Taylor series of $\mathbf{y}$ about x:

$$\mathbf{y}(x + h) \approx \mathbf{y}(x) + \mathbf{y}'(x)h \tag{7.6}$$

Because Eq. (7.6) predicts $\mathbf{y}$ at $x + h$ from the information available at x, it can be used to move the solution forward in steps of h, starting with the given initial values of x and $\mathbf{y}$.

The error in Eq. (7.6) caused by truncation of the Taylor series is given by Eq. (A4):

$$\mathbf{E} = \frac{1}{2}\mathbf{y}''(\xi)h^2 = \mathcal{O}(h^2), \quad x < \xi < x + h \tag{7.7}$$

A rough idea of the accumulated error $\mathbf{E}_{\text{acc}}$ can be obtained by assuming that per-step error is constant over the period of integration. Then after n integration steps covering the interval x_0 to x_n we have

$$\mathbf{E}_{\text{acc}} = n\mathbf{E} = \frac{x_n - x_0}{h}\mathbf{E} = \mathcal{O}\left(h\right) \tag{7.8}$$

Hence the accumulated error is one order less than the per-step error.

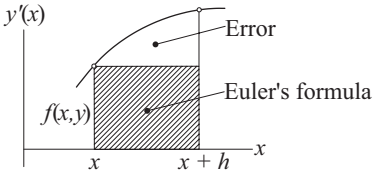


Figure 7.1. Graphical representation of Euler's formula.

Let us now take a look at the graphical interpretation of Euler's equation. 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 Figure 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 $(y')' = y''(x)$.

Euler's method is seldom used in practice because of its computational inefficiency. Suppressing the truncation error to an acceptable level requires a very small h, resulting in many integration steps accompanied by an increase in the roundoff error. The value of the method lies mainly in its simplicity, which facilitates the discussion of certain important topics, such as stability.

■ `euler`

This function implements Euler's method of integration. It can handle any number of first-order differential equations. The user is required to supply the function `F(x,y)` that specifies the differential equations in the form of the array

$$\mathbf{F}(x, \mathbf{y}) = \begin{bmatrix} y_0' \\ y_1' \\ y_2' \\ \vdots \end{bmatrix}$$

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

```
## module euler
''' X,Y = integrate(F,x,y,xStop,h).
    Euler's 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]}.
'''
import numpy as np
def integrate(F,x,y,xStop,h):
    X = []
    Y = []
    X.append(x)
    Y.append(y)
    while x < xStop:
        h = min(h,xStop - x)
        y = y + h*F(x,y)
        x = x + h
        X.append(x)
        Y.append(y)
    return np.array(X),np.array(Y)
```

■ 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 fifth 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 printout 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  ",end=" ")
        for i in range (n):
            print("      y[",i,"] ",end=" ")
        print()

    def printLine(x,y,n):
        print("{:13.4e}".format(x),end=" ")
        for i in range (n):
            print("{:13.4e}".format(y[i]),end=" ")
        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

Integrate the initial value problem

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

in steps of $h = 0.01$ from $x = 0$ to 0.03. Also compute the analytical solution

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

and the accumulated truncation error at each step.

Solution. It is convenient to use the notation

$$x_i = ih \qquad y_i = y(x_i)$$

so that Euler's formula takes the form

$$y_{i+1} = y_i + y_i' h$$

where

$$y_i' = x_i^2 - 4y_i$$

Step 1. ($x_0 = 0$ to $x_1 = 0.01$):

$$\begin{aligned} y_0 &= 0 \\ y_0' &= x_0^2 - 4y_0 = 0^2 - 4(1) = -4 \\ y_1 &= y_0 + y_0' h = 1 + (-4)(0.01) = 0.96 \\ (y_1)_{\text{exact}} &= \frac{31}{32}e^{-4(0.01)} + \frac{1}{4}0.01^2 - \frac{1}{8}0.01 + \frac{1}{32} = 0.9608 \\ E_{\text{acc}} &= 0.96 - 0.9608 = -0.0008 \end{aligned}$$

Step 2. ($x_1 = 0.01$ to $x_2 = 0.02$):

$$\begin{aligned} y_1' &= x_1^2 - 4y_1 = 0.01^2 - 4(0.96) = -3.840 \\ y_2 &= y_1 + y_1' h = 0.96 + (-3.840)(0.01) = 0.9216 \\ (y_2)_{\text{exact}} &= \frac{31}{32}e^{-4(0.02)} + \frac{1}{4}0.02^2 - \frac{1}{8}0.02 + \frac{1}{32} = 0.9231 \\ E_{\text{acc}} &= 0.9216 - 0.9231 = -0.0015 \end{aligned}$$

Step 3. ($x_2 = 0.02$ to $x_3 = 0.03$):

$$y_2' = x_2^2 - 4y_2 = 0.02^2 - 4(0.9216) = -3.686$$
$$y_3 = y_2 + y_2'h = 0.9216 + (-3.686)(0.01) = 0.8847$$
$$(y_3)_{\text{exact}} = \frac{31}{32}e^{-4(0.03)} + \frac{1}{4}0.03^2 - \frac{1}{8}0.03 + \frac{1}{32} = 0.8869$$
$$E_{\text{acc}} = 0.8847 - 0.8869 = -0.0022$$

We note that the magnitude of the per-step error is roughly constant at 0.008. Thus after 10 integration steps the accumulated error would be approximately 0.08, thereby reducing the solution to one significant figure accuracy. After 100 steps all significant figures would be lost.

EXAMPLE 7.2
Integrate the initial value problem

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

from $x = 0$ to 2 with Euler's method using $h = 0.05$. Plot the computed y together with the analytical solution,

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

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

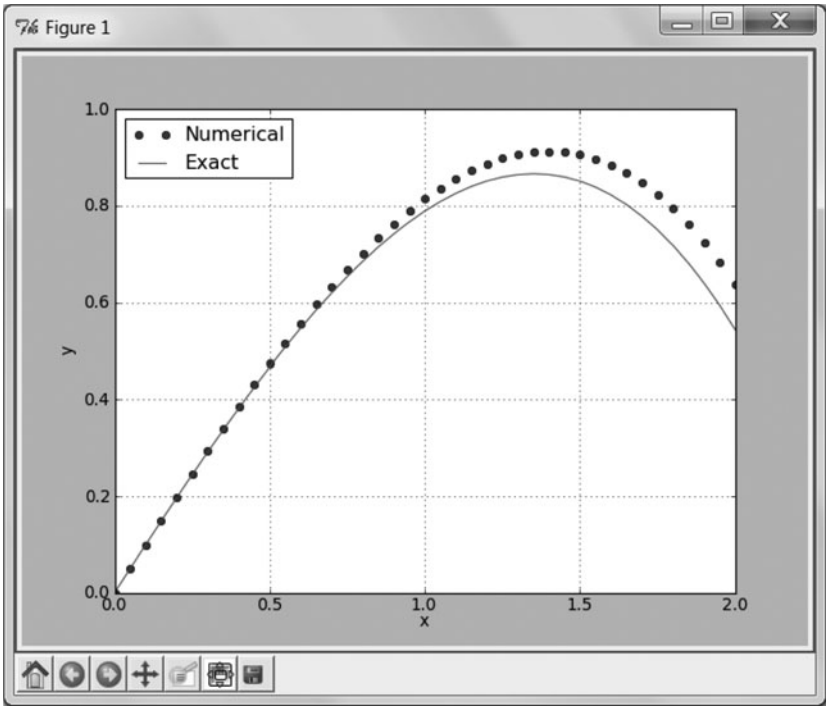
$$\mathbf{F}(x, \mathbf{y}) = \begin{bmatrix} y_0' \\ y_1' \end{bmatrix} = \begin{bmatrix} y_1 \\ -0.1y_1 - x \end{bmatrix} \qquad \mathbf{y}(0) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Here is a program that uses the function `euler`:

```
#!/usr/bin/python
## example7_2
import numpy as np
from euler import *
import matplotlib.pyplot as plt

def F(x,y):
    F = np.zeros(2)
    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 = np.array([0.0, 1.0])    # Initial values of {y}
h = 0.05                    # Step size
```

```
X,Y = integrate(F,x,y,xStop,h)
yExact = 100.0*X - 5.0*X**2 + 990.0*(np.exp(-0.1*X) - 1.0)
plt.plot(X,Y[:,0],'o',X,yExact,'-')
plt.grid(True)
plt.xlabel('x'); plt.ylabel('y')
plt.legend(('Numerical','Exact'),loc=0)
plt.show()
input("Press return to exit")
```

The resulting plot is



The initial portion of the plot is almost a straight line. Because the truncation error in the numerical solution is proportional to y'', the discrepancy between the two solutions is small. As the curvature of the plot increases, so does the truncation error.

You may want to see the effect of changing h by running the program with $h = 0.025$. Doing so should halve the truncation error.

7.3 Runge-Kutta Methods

Euler's method is classified as a first-order method because its cumulative truncation error behaves as $\mathcal{O}(h)$. Its basis was the truncated Taylor series

$$\mathbf{y}(x + h) = \mathbf{y}(x) + \mathbf{y}'(x)h$$

8.2 Shooting Method

Second-Order Differential Equation

The simplest two-point boundary value problem is a second-order differential equation with one condition specified at $x = a$ and another one at $x = b$. Here is an example of such a problem:

$$y'' = f(x, y, y'), \quad y(a) = \alpha, \quad y(b) = \beta \tag{8.1}$$

Let us now attempt to turn Eqs. (8.1) into the initial value problem

$$y'' = f(x, y, y'), \quad y(a) = \alpha, \quad y'(a) = u \tag{8.2}$$

The key to success is finding the correct value of u. This could be done by trial and error: Guess u and solve the initial value problem by marching from $x = a$ to b. If the solution agrees with the prescribed boundary condition $y(b) = \beta$, we are done; otherwise we have to adjust u and try again. Clearly, this procedure is very tedious.

More systematic methods become available to us if we realize that the determination of u is a root-finding problem. Because the solution of the initial value problem depends on u, the computed value of $y(b)$ is a function of u; that is

$$y(b) = \theta(u)$$

Hence u is a root of

$$r(u) = \theta(u) - \beta = 0 \tag{8.3}$$

where $r(u)$ is the *boundary residual* (the difference between the computed and specified boundary value at $x = b$). Equation (8.3) can be solved by one of the root-finding methods discussed in Chapter 4. We reject the method of bisection because it involves too many evaluations of $\theta(u)$. In the Newton-Raphson method we run into the problem of having to compute $d\theta/du$, which can be done but not easily. That leaves Ridder's algorithm as our method of choice.

Here is the procedure we use in solving nonlinear boundary value problems:

> Specify the starting values u_1 and u_2 that *must bracket the root* u of Eq. (8.3).
> Apply Ridder's method to solve Eq. (8.3) for u. Note that each iteration requires evaluation of $\theta(u)$ by solving the differential equation as an initial value problem.
> Having determined the value of u, solve the differential equations once more and record the results.

If the differential equation is linear, any root-finding method will need only one interpolation to determine u. Because Ridder's method uses three points (u_1, u_2, and u_3), it is wasteful compared with linear interpolation, which uses only two points

(u_1 and u_2). Therefore, we replace Ridder's method with linear interpolation whenever the differential equation is linear.

■ linInterp

Here is the algorithm we use for linear interpolation:

```
## module linInterp
''' root = linInterp(f,x1,x2).
    Finds the zero of the linear function f(x) by straight
    line interpolation based on x = x1 and x2.
'''
def linInterp(f,x1,x2):
    f1 = f(x1)
    f2 = f(x2)
    return = x2 - f2*(x2 - x1)/(f2 - f1)
```

EXAMPLE 8.1
Solve the boundary value problem

$$y'' + 3yy' = 0 \qquad y(0) = 0 \qquad y(2) = 1$$

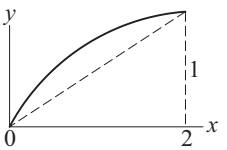
Solution. The equivalent first-order equations are

$$\mathbf{y}' = \begin{bmatrix} y_0' \\ y_1' \end{bmatrix} = \begin{bmatrix} y_1 \\ -3y_0y_1 \end{bmatrix}$$

with the boundary conditions

$$y_0(0) = 0 \qquad y_0(2) = 1$$

Now comes the daunting task of determining the trial values of $y'(0)$. We could always pick two numbers at random and hope for the best. However, it is possible to reduce the element of chance with a little detective work. We start by making the reasonable assumption that y is smooth (does not wiggle) in the interval $0 \le x \le 2$. Next we note that y has to increase from 0 to 1, which requires $y' > 0$. Because both y and y' are positive, we conclude that y'' must be negative to satisfy the differential equation. Now we are in a position to make a rough sketch of y:



Looking at the sketch it is clear that $y'(0) > 0.5$, so that $y'(0) = 1$ and 2 appear to be reasonable values for the brackets of $y'(0)$; if they are not, Ridder's method will display an error message.

In the program listed next we chose the fourth-order Runge-Kutta method for integration. It can be replaced by the adaptive version by substituting `run_kut5` for `run_kut4` in the `import` statement. Note that three user-supplied functions are needed to describe the problem at hand. Apart from the function `F(x,y)` that defines the differential equations, we also need the functions `initCond(u)` to specify the initial conditions for integration, and `r(u)` to provide Ridder's method with the boundary condition residual. By changing a few statements in these functions, the program can be applied to any second-order boundary value problem. It also works for third-order equations if integration is started at the end where two of the three boundary conditions are specified.

```
#!/usr/bin/python
## example8_1
import numpy as np
from run_kut4 import *
from ridder import *
from printSoln import *

def initCond(u):  # Init. values of [y,y']; use 'u' if unknown
    return np.array([0.0, u])

def r(u):         # Boundary condition residual--see Eq. (8.3)
    X,Y = integrate(F,xStart,initCond(u),xStop,h)
    y = Y[len(Y) - 1]
    r = y[0] - 1.0
    return r

def F(x,y):       # First-order differential equations
    F = np.zeros(2)
    F[0] = y[1]
    F[1] = -3.0*y[0]*y[1]
    return F

xStart = 0.0        # Start of integration
xStop = 2.0         # End of integration
u1 = 1.0            # 1st trial value of unknown init. cond.
u2 = 2.0            # 2nd trial value of unknown init. cond.
h = 0.1             # Step size
freq = 2            # Printout frequency
u = ridder(r,u1,u2) # Compute the correct initial condition
X,Y = integrate(F,xStart,initCond(u),xStop,h)
printSoln(X,Y,freq)
input("\nPress return to exit")
```

Here is the solution:

```
      x          y[ 0 ]         y[ 1 ]
 0.0000e+00     0.0000e+00     1.5145e+00
 2.0000e-01     2.9404e-01     1.3848e+00
 4.0000e-01     5.4170e-01     1.0743e+00
 6.0000e-01     7.2187e-01     7.3287e-01
 8.0000e-01     8.3944e-01     4.5752e-01
 1.0000e+00     9.1082e-01     2.7013e-01
 1.2000e+00     9.5227e-01     1.5429e-01
 1.4000e+00     9.7572e-01     8.6471e-02
 1.6000e+00     9.8880e-01     4.7948e-02
 1.8000e+00     9.9602e-01     2.6430e-02
 2.0000e+00     1.0000e+00     1.4522e-02
```

Note that $y'(0) = 1.5145$, so that our starting values of 1.0 and 2.0 were on the mark.

EXAMPLE 8.2

Numerical integration of the initial value problem

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

yielded $y'(2) = 1.653\,64$. Use this information to determine the value of $y'(0)$ that would result in $y'(2) = 0$.

Solution. We use linear interpolation

$$u = u_2 - \theta(u_2)\frac{u_2 - u_1}{\theta(u_2) - \theta(u_1)}$$

where in our case $u = y'(0)$ and $\theta(u) = y'(2)$. So far we are given $u_1 = 0$ and $\theta(u_1) = 1.653\,64$. To obtain the second point, we need another solution of the initial value problem. An obvious solution is $y = x$, which gives us $y(0) = 0$ and $y'(0) = y'(2) = 1$. Thus the second point is $u_2 = 1$ and $\theta(u_2) = 1$. Linear interpolation now yields

$$y'(0) = u = 1 - (1)\frac{1 - 0}{1 - 1.653\,64} = 2.529\,89$$

EXAMPLE 8.3

Solve the third-order boundary value problem

$$y''' = 2y'' + 6xy \qquad y(0) = 2 \qquad y(5) = y'(5) = 0$$

and plot y and y' vs. x.

Solution. The first-order equations and the boundary conditions are

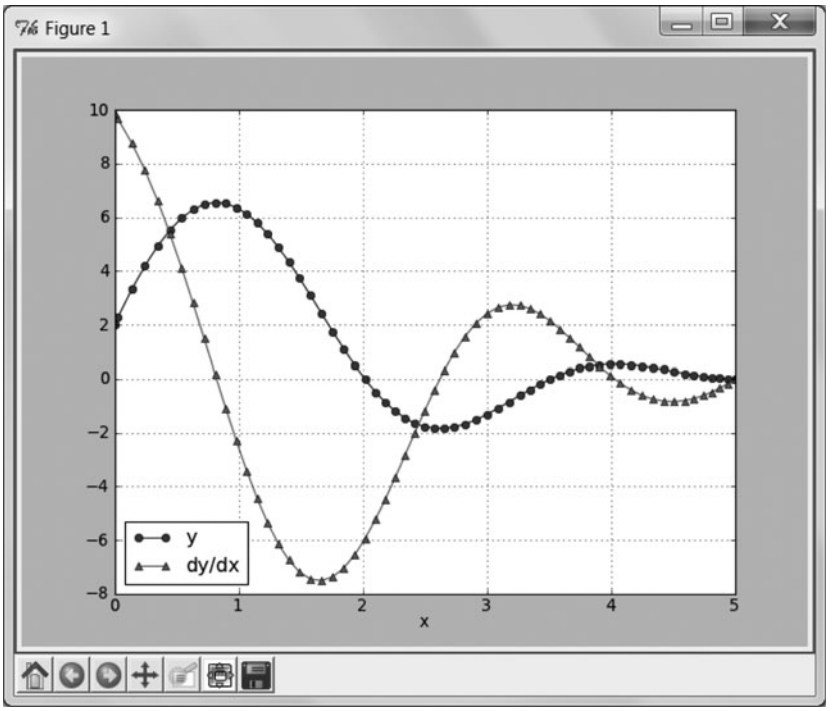
$$\mathbf{y}' = \begin{bmatrix} y_0' \\ y_1' \\ y_2' \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ 2y_2 + 6xy_0 \end{bmatrix}$$

$$y_0(0) = 2 \qquad y_0(5) = y_1(5) = 0$$

The program listed next is based on `example8_1`. Because two of the three boundary conditions are specified at the right end, we start the integration at $x = 5$ and proceed with negative h toward $x = 0$. Two of the three initial conditions are prescribed— $y_0(5) = y_1(5) = 0$—whereas the third condition $y_2(5)$ is unknown. Because the differential equation is linear, we replaced `ridder` with `linInterp`. In linear interpolation the two guesses for $y_2(5)$ (u_1 and u_2) are not important, so we left them as they were in Example 8.1. The adaptive Runge-Kutta method (`run_kut5`) was chosen for the integration.

```
#!/usr/bin/python
## example8_3
import matplotlib.pyplot as plt
import numpy as np
from run_kut5 import *
from linInterp import *

def initCond(u):  # Initial values of [y,y',y"];
                  # use 'u' if unknown
    return np.array([0.0, 0.0, u])

def r(u):  # Boundary condition residual--see Eq. (8.3)
    X,Y = integrate(F,xStart,initCond(u),xStop,h)
    y = Y[len(Y) - 1]
    r = y[0] - 2.0
    return r

def F(x,y):  # First-order differential equations
    F = np.zeros(3)
    F[0] = y[1]
    F[1] = y[2]
    F[2] = 2.0*y[2] + 6.0*x*y[0]
    return F

xStart = 5.0        # Start of integration
xStop = 0.0         # End of integration
u1 = 1.0            # 1st trial value of unknown init. cond.
u2 = 2.0            # 2nd trial value of unknown init. cond.
h = -0.1            # initial step size
freq = 2            # printout frequency
u = linInterp(r,u1,u2)
X,Y = integrate(F,xStart,initCond(u),xStop,h)

plt.plot(X,Y[:,0],'o-',X,Y[:,1],'^-')
plt.xlabel('x')
```

```
plt.legend(('y','dy/dx'),loc = 3)
plt.grid(True)
plt.show()
input("\nPress return to exit")
```



Higher Order Equations

Let us consider the fourth-order differential equation

$$y^{(4)} = f(x, y, y', y'', y''') \tag{8.4a}$$

with the boundary conditions

$$y(a) = \alpha_1 \qquad y''(a) = \alpha_2 \qquad y(b) = \beta_1 \qquad y''(b) = \beta_2 \tag{8.4b}$$

To solve Eqs. (8.4) with the shooting method, we need four initial conditions at $x = a$, only two of which are specified. Denoting the unknown initial values by u_1 and u_2, the set of initial conditions is

$$y(a) = \alpha_1 \qquad y'(a) = u_1 \qquad y''(a) = \alpha_2 \qquad y'''(a) = u_2 \tag{8.5}$$

If Eq. (8.4a) is solved with the shooting method using the initial conditions in Eq. (8.5), the computed boundary values at $x = b$ depend on the choice of u_1 and u_2. We denote this dependence as

$$y(b) = \theta_1(u_1, u_2) \qquad y''(b) = \theta_2(u_1, u_2) \tag{8.6}$$

The correct values u_1 and u_2 satisfy the given boundary conditions at $x = b$,

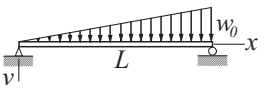
$$\theta_1(u_1, u_2) = \beta_1 \qquad \theta_2(u_1, u_2) = \beta_2$$

or, using vector notation

$$\boldsymbol{\theta}(\mathbf{u}) = \boldsymbol{\beta} \tag{8.7}$$

These are simultaneous, (generally nonlinear) equations that can be solved by the Newton-Raphson method discussed in Section 4.6. It must be pointed out again that intelligent estimates of u_1 and u_2 are needed if the differential equation is not linear.

EXAMPLE 8.4



The displacement v of the simply supported beam can be obtained by solving the boundary value problem

$$\frac{d^4v}{dx^4} = \frac{w_0}{EI}\frac{x}{L} \qquad v = \frac{d^2v}{dx^2} = 0 \text{ at } x = 0 \text{ and } x = L$$

where EI is the bending rigidity. Determine by numerical integration the slopes at the two ends and the displacement at mid-span.

Solution. Introducing the dimensionless variables

$$\xi = \frac{x}{L} \qquad y = \frac{EI}{w_0 L^4}v$$

the problem is transformed to

$$\frac{d^4y}{d\xi^4} = \xi \qquad y = \frac{d^2y}{d\xi^2} = 0 \text{ at } \xi = 0 \text{ and } 1$$

The equivalent first-order equations and the boundary conditions are (the prime denotes $d/d\xi$)

$$\mathbf{y}' = \begin{bmatrix} y_0' \\ y_1' \\ y_2' \\ y_3' \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \xi \end{bmatrix}$$

$$y_0(0) = y_2(0) = y_0(1) = y_2(1) = 0$$

The program listed next is similar to the one in Example 8.1. With appropriate changes in functions `F(x,y)`, `initCond(u)`, and `r(u)` the program can solve boundary value problems of any order greater than two. For the problem at hand we

chose the Bulirsch-Stoer algorithm to do the integration because it gives us control over the printout (we need y precisely at mid-span). The nonadaptive Runge-Kutta method could also be used here, but we would have to guess a suitable step size h.

Because the differential equation is linear, the solution requires only one iteration with the Newton-Raphson method. In this case the initial values $u_1 = dy/d\xi|_{x=0}$ and $u_2 = d^3y/d\xi^3|_{x=0}$ are irrelevant; convergence always occurs in one iteration.

```
#!/usr/bin/python
## example8_4
import numpy as np
from bulStoer import *
from newtonRaphson2 import *
from printSoln import *

def initCond(u):  # Initial values of [y,y',y",y"'];
                  # use 'u' if unknown
    return np.array([0.0, u[0], 0.0, u[1]])

def r(u):  # Boundary condition residuals--see Eq. (8.7)
    r = np.zeros(len(u))
    X,Y = bulStoer(F,xStart,initCond(u),xStop,H)
    y = Y[len(Y) - 1]
    r[0] = y[0]
    r[1] = y[2]
    return r

def F(x,y):  # First-order differential equations
    F = np.zeros(4)
    F[0] = y[1]
    F[1] = y[2]
    F[2] = y[3]
    F[3] = x
    return F

xStart = 0.0               # Start of integration
xStop = 1.0                # End of integration
u = np.array([0.0, 1.0])   # Initial guess for {u}
H = 0.5                    # Printout increment
freq = 1                   # Printout frequency
u = newtonRaphson2(r,u,1.0e-4)
X,Y = bulStoer(F,xStart,initCond(u),xStop,H)
printSoln(X,Y,freq)
input("\nPress return to exit")
```