Numerical Analysis HW 14

Kevin Ho

April 2025

14.1. Verify that the overall truncation error for the FD scheme (14.50) is second-order in h.

Verification of Second-Order Accuracy

We define the leading truncation error from midpoint expansion as

$$K(x) = \frac{p(x)y^{(3)}(x)}{6}\left(\frac{h}{2}\right)^2 = O(h^2).$$

Using Taylor expansion at midpoints $x_{i\pm\frac{1}{2}}$, the midpoint approximations become:

$$\begin{aligned}(p\,y_x)^{\text{approx}}_{i+\frac{1}{2}} &= (p\,y_x)^{\text{exact}}_{i+\frac{1}{2}} - K(x_{i+\frac{1}{2}}) + O(h^4),\\ (p\,y_x)^{\text{approx}}_{i-\frac{1}{2}} &= (p\,y_x)^{\text{exact}}_{i-\frac{1}{2}} - K(x_{i-\frac{1}{2}}) + O(h^4).\end{aligned}$$

Now we consider the central FD approximation in (14.50):

$$\frac{(p\,y_x)^{\text{approx}}_{i+\frac{1}{2}} - (p\,y_x)^{\text{approx}}_{i-\frac{1}{2}}}{h} = \underbrace{\frac{(p\,y_x)^{\text{exact}}_{i+\frac{1}{2}} - (p\,y_x)^{\text{exact}}_{i-\frac{1}{2}}}{h}}_{(p\,y_x)'(x_i)+O(h^2)} - \frac{K(x_{i+\frac{1}{2}}) - K(x_{i-\frac{1}{2}})}{h} + O(h^2).$$

Since $K(x) = O(h^2)$, then its derivative $K'(x) = O(h^2)$ as well. Applying the Taylor expansion:

$$K(x_{i+\frac{1}{2}}) - K(x_{i-\frac{1}{2}}) = h\,K'(x_i) + O(h^3), \quad \Rightarrow \quad \frac{K(x_{i+\frac{1}{2}}) - K(x_{i-\frac{1}{2}})}{h} = K'(x_i) + O(h^2) = O(h^2).$$

Combining all terms:

$$O(h^2) - O(h^2) + O(h^2) = O(h^2),$$

which verifies that the FD scheme (14.50) has a second-order truncation error in h.

14.2. Consider a BVP

$$y'' = y' + 2y + \cos x, \quad 0 \le x \le \frac{\pi}{2}, \qquad y(0) = -0.3, \quad y\left(\frac{\pi}{2}\right) = -0.1.$$

Its exact solution is

$$y(x) = -\frac{\sin x + 3\cos x}{10}.$$

(a) Show that the BVP has a unique solution.
Consider the associated homogeneous BVP

$$y'' - y' - 2y = 0, \qquad y(0) = 0, \quad y\Big(\frac{\pi}{2}\Big) = 0.$$

Its characteristic equation is

$$r^2 - r - 2 = 0 \quad \Longrightarrow \quad r = 2,\ -1,$$

so the general solution is

$$y_h(x) = C_1 e^{2x} + C_2 e^{-x}.$$

Imposing $y_h(0) = 0$ gives

$$C_1 + C_2 = 0 \implies y_h(x) = C_1\big(e^{2x} - e^{-x}\big).$$

Then the condition at $x = \pi/2$ reads

$$y_h\Big(\tfrac{\pi}{2}\Big) = C_1\big(e^{\pi} - e^{-\pi/2}\big) = 0.$$

Since $e^{\pi} - e^{-\pi/2} > 0$, it follows that $C_1 = 0$, hence $C_2 = 0$. Thus the only solution of the homogeneous BVP is the trivial one,

$$y_h(x) \equiv 0.$$

By standard linear theory, the non-homogeneous BVP admits at most one solution.

(b) Use the shooting method (with `RK4SYSTEM`) to approximate the solution for

$$h = \frac{\pi}{10} \quad \text{and} \quad h = \frac{\pi}{20}.$$

(c) Measure the errors for the approximate solutions computed in part (b), and determine whether the numerical solutions demonstrate the expected accuracy.

Solution shown in the code

14.3. For numerical solution of the differential equation in (14.62)

(a) Implement a code, using the FD schemes studied in Section 14.3.

(b) Perform an error analysis comparing with results in Example 14.17.

Solution in the code

Code for Homework 14

14.2 Code

```
import numpy as np

# Reuse the shooting method solver from before
def f(x, u):
    y, yp = u
    return np.array([yp, yp + 2*y + np.cos(x)])

def rk4_step(f, x, u, h):
    k1 = f(x, u)
    k2 = f(x + 0.5*h, u + 0.5*h*k1)
    k3 = f(x + 0.5*h, u + 0.5*h*k2)
    k4 = f(x + h, u + h*k3)
    return u + (h/6)*(k1 + 2*k2 + 2*k3 + k4)

def integrate_and_endvalue(s, h, x_end=np.pi/2):
    n_steps = int(np.round(x_end/h))
    x = 0.0
    u = np.array([-0.3, s])
    for _ in range(n_steps):
        u = rk4_step(f, x, u, h)
        x += h
    return u[0]

def find_slope(h, target, tol=1e-10, maxit=50):
    s0, s1 = 0.0, 1.0
    F0 = integrate_and_endvalue(s0, h) - target
    F1 = integrate_and_endvalue(s1, h) - target
    for _ in range(maxit):
        s2 = s1 - F1*(s1 - s0)/(F1 - F0)
        F2 = integrate_and_endvalue(s2, h) - target
        if abs(F2) < tol:
            return s2
        s0, F0, s1, F1 = s1, F1, s2, F2
    raise RuntimeError("Secant failed")

def solve_bvp(h):
    x_end = np.pi/2
    target = -0.1
    s = find_slope(h, target)
    n = int(round(x_end/h))
    xs = np.linspace(0, x_end, n+1)
    us = np.zeros((n+1, 2))
    us[0] = np.array([-0.3, s])
    x = 0.0
    for i in range(n):
        us[i+1] = rk4_step(f, x, us[i], h)
        x += h
    return xs, us[:,0]

def y_exact(x):
```

```
    return -(np.sin(x) + 3*np.cos(x)) / 10

# Grids and solutions
h1, h2 = np.pi/10, np.pi/20
x1, y1 = solve_bvp(h1)
x2, y2 = solve_bvp(h2)
y_ex1 = y_exact(x1)

# Sample points: i = 1..5 on coarse grid
indices = range(1, 6)
rows = []
for i in indices:
    xv = x1[i]
    u1 = y1[i]
    # find index in fine grid
    idx2 = int(round(xv/h2))
    u2 = y2[idx2]
    ye = y_ex1[i]
    e1 = abs(u1 - ye)
    e2 = abs(u2 - ye)
    # Richardson extrapolation
    uR = u2 + (u2 - u1)/3
    eR = abs(uR - ye)
    rows.append((xv, ye, u1, e1, u2, e2, uR, eR))

# Print formatted table
header = f"{'x':>7} {'exact_y':>10} | {'u(h=pi/10)':>12} {'error':>10} |
{'u(h=pi/20)':>12} {'error':>10} | {'Rich u':>10} {'Rich err':>12}"
print(header)
print("-"*len(header))
for xv, ye, u1, e1, u2, e2, uR, eR in rows:
    print(f"x={xv:5.3f} {ye:10.5f} | {u1:12.5f} {e1:10.5f} | {u2:12.5f}
{e2:10.5f} | {uR:10.5f} {eR:12.3e}")
```

Output

```
      x     exact_y |    u(h=pi/10)       error |    u(h=pi/20)       error |      Rich u     Rich err
-------------------------------------------------------------------------------------------------------
x=0.314   -0.31622 |     -0.31621    0.00001 |     -0.31622    0.00000 |   -0.31622    1.571e-06
x=0.628   -0.30148 |     -0.30148    0.00001 |     -0.30148    0.00000 |   -0.30149    1.929e-06
x=0.942   -0.25724 |     -0.25723    0.00000 |     -0.25724    0.00000 |   -0.25724    1.454e-06
x=1.257   -0.18781 |     -0.18781    0.00000 |     -0.18781    0.00000 |   -0.18781    6.103e-07
x=1.571   -0.10000 |     -0.10000    0.00000 |     -0.10000    0.00000 |   -0.10000    3.053e-16
```

14.3

```
import numpy as np

def solve_fd(h):
    """Solve - (p y')' + q y = f on [1,2] with y(1)=0, y'(2)=-2 using FD from
§14.3."""
    a, b = 1.0, 2.0
    alpha, beta = 0.0, -2.0
    # grid
    N = int(round((b - a) / h))
    x = np.linspace(a, b, N+1)
    # allocate matrix A and rhs
    A = np.zeros((N, N))
    rhs = np.zeros(N)
    # coefficient functions
    p = lambda z: 1.0 / z
    q = lambda z: 3.0 / z**3
    f = lambda z: 4.0 * z

    for i in range(1, N+1):
        xi = x[i]
        qi = q(xi)
        fi = f(xi)
        # midpoints
        p_imh = p((x[i] + x[i-1]) / 2)
        p_iph = p((x[i] + (x[i+1] if i < N else b + h)) / 2)
        row = i-1
        if i == 1:
            A[row, row]   = p_imh + p_iph + h**2 * qi
            A[row, row+1] = -p_iph
            rhs[row]      = h**2 * fi + p_imh * alpha
        elif i == N:
            A[row, row-1] = -(p_imh + p_iph)
            A[row, row]   =  (p_imh + p_iph + h**2 * qi)
            rhs[row]      = h**2 * fi + 2*h*beta * p_iph
        else:
            A[row, row-1] = -p_imh
            A[row, row]   =  p_imh + p_iph + h**2 * qi
            A[row, row+1] = -p_iph
            rhs[row]      = h**2 * fi

    y_int = np.linalg.solve(A, rhs)
    y = np.empty(N+1)
    y[0] = alpha
    y[1:] = y_int
    return x, y

def y_exact(x):
    """Exact solution."""
    return -8.0/(7.0*x) + (68.0/35.0)*x**3 - (4.0/5.0)*x**4

# compute solutions
h1, h2 = 1/5, 1/10
```

```
x1, y1 = solve_fd(h1)
x2, y2 = solve_fd(h2)
y_ex1 = y_exact(x1)
y_ex2 = y_exact(x2)

# x-values for comparison
x_vals = [1.2, 1.4, 1.6, 1.8, 2.0]

rows = []
for xv in x_vals:
    i1 = np.where(np.isclose(x1, xv))[0][0]
    i2 = np.where(np.isclose(x2, xv))[0][0]
    ye = y_ex1[i1]
    u1 = y1[i1]
    u2 = y2[i2]
    err1 = abs(u1 - ye)
    err2 = abs(u2 - ye)
    # Richardson extrapolation (second-order)
    uR = u2 + (u2 - u1)/3
    errR = abs(uR - ye)
    rows.append((xv, ye, u1, err1, u2, err2, uR, errR))

# print table
header = ("{:>6} {:>10} | {:>10} {:>10} | {:>10} {:>10} | {:>10} {:>12}"
          .format("x", "exact_y", "u(h=1/5)", "error", "u(h=1/10)", "error",
"Rich u", "Rich err"))
print(header)
print("-" * len(header))
for xv, ye, u1, e1, u2, e2, uR, eR in rows:
    print(f"x={xv:4.2f} {ye:10.5f} | {u1:10.5f} {e1:10.5f} | {u2:10.5f}
{e2:10.5f} | {uR:10.5f} {eR:12.3e}")
```

Output

```
     x    exact_y |   u(h=1/5)      error |  u(h=1/10)      error |      Rich u     Rich err
-------------------------------------------------------------------------------------------
x=1.20    0.74600 |    0.75386    0.00787 |    0.74795    0.00195 |    0.74597    2.317e-05
x=1.40    1.44159 |    1.46325    0.02166 |    1.44700    0.00540 |    1.44158    1.456e-05
x=1.60    2.00078 |    2.04176    0.04098 |    2.01103    0.01025 |    2.00078    4.150e-06
x=1.80    2.29774 |    2.36390    0.06615 |    2.31430    0.01656 |    2.29777    2.774e-05
x=2.00    2.17143 |    2.26914    0.09772 |    2.19590    0.02447 |    2.17148    5.516e-05
```