Solutions Manual to Applied Partial Differential Equations

Stewart Nash

Email address, S. Nash: pervigilent@gmail.com
URL: https://github.com/Pervigilent/applied_pde_solutions

Contents

Preface

This document consists of computer-based solutions to problems in "Applied Partial Differential Equations" by Paul DuChateau and David Zachmann.

CHAPTER 1

Mathematical Modeling and Partial Differential Equations

1. Mathematical Modeling of Physical Systems

2. Equation of Heat Conduction

1. Consider an infinitely long rod for which the parameters K, ϵ, σ, C are such that $\beta = 0.1$. Then equation (1.2.8*) becomes

$$u_n^{j+1} = 0.1u_{n+1}^j + 0.8u_n^j + 0.1u_{n-1}^j \tag{2.1}$$

Suppose

$$u_n^0 \begin{cases} 1 & \text{for } n = 4, 5, 6 \\ 0 & \text{for all other } n \end{cases} \tag{2.2}$$

Then use (1.2.8*) and this initial condition to compose u_n^j for $n = -5, \ldots, 5$ for $j = 1, \ldots, 5$. For each value of j, for how many n is u_n^j different from zero?

2. Repeat Exercise 1 for the situation in which the rod is of finite length L with $10\epsilon = L$. Suppose

$$u_0^j = 1 \text{ and } \quad u_{10}^j = -1 \quad \text{for all } j > 0 \tag{2.3}$$

and

$$u_n^0 = 0 \quad \text{for all } n \tag{2.4}$$

Then use (1.2.8*) to compute u_n^j for $n = 1, \ldots, 9$ and $j = 1, \ldots, 5$.

```
import numpy as np
import matplotlib.pyplot as plt

N = 10 # Length of rod
T = 5 # Duration of simulation

def matrix_power(x, n):
    y = x.copy()
    if n > 1:
        for i in np.arange(n - 1):
            y = np.matmul(y, x)
    return y

coefficients = np.zeros((N, N))
```

```
for i in np.arange(N):
    if i == 0:
        coefficients[i, 0]  = 1
    elif i == N - 1:
        coefficients[i, N - 1] = 1
    else:
        coefficients[i, i - 1] = 0.1
        coefficients[i, i] = 0.8
        coefficients[i, i + 1] = 0.1

initial_conditions = np.zeros((N, 1))
initial_conditions[0, 0] = 1
initial_conditions[N - 1, 0] = -1
y = [np.squeeze(np.transpose(np.matmul(matrix_power(coefficients, i),\
 initial_conditions))) for i in np.arange(T)]

length_intervals = np.arange(N)
plt.figure()
for i in np.arange(T):
    plt.plot(length_intervals, y[i])
plt.show()
print("The number of non-zero elements is {0}.".format(\
 np.count_nonzero(y[T - 1] != 0)))
```

3. Steady-State Conduction of Heat

4. Transmission Line Equations

5. Well-Posed Problems

6. Classification of Equations

CHAPTER 2

Fourier Series and Eigenfunction Expansions

1. Fourier Series

2. Generalized Fourier Series

3. Sturm-Liouville Problems

1. Compute all the eigenvalues and corresponding eigenfunctions for the following Sturm-Liouville problems:
(a) $-u''(x) = \mu u(x)$, $u(0) = u'(1) = 0$
(b) $-v''(x) = \mu v(x)$, $v(0) = v(2) = 0$
(c) $-w''(x) = \mu w(x)$, $w'(0) = w(1) = 0$
(d) $-z''(x) = \mu z(x)$, $z'(0) = z'(2) = 0$

(a) $\mu_n = [(n - \frac{1}{2})\pi]^2$, $u_n(x) = \sin(n - \frac{1}{2})\pi x$, $n = 1, 2, \ldots$
(b) $\mu_n = [n\pi/2]^2$, $u_n(x) = \sin(n\pi/2)x$, $n = 1, 2, \ldots$
(c) $\mu_n = [(n - \frac{1}{2})\pi]^2$, $u_n(x) = \cos(n - \frac{1}{2})\pi x$, $n = 0, 1, 2, \ldots$
(d) $\mu_0 = 0$, $\mu_n = [n\pi/2]^2$, $u_0(x) = \sqrt{\frac{1}{2}}$, $u_n(x) = \cos(n\pi/2)x$, $n = 1, 2, \ldots$

4. Discrete Fourier Series

5. Function Space L^2

6. Multiple Fourier Series

7. Summary

CHAPTER 3

Boundary-Value Problems and Initial-Boundary-Value Problems on Spatially Bounded Domains

1. Boundary-Value Problems for Laplace and Poisson Equations

2. Evolution Equations: Initial-Boundary-Value Problems for Heat Equation

3. Evolution Equations: Initial-Boundary-Value Problems for Wave Equation

CHAPTER 4

Integral Transforms

1. Function Space $L^2(a, b)$ when (a, b) is Unbounded

2. The Fourier Transform

1. Compute the Fourier transform of the following functions:

(a)

$$f(x) = \begin{cases} 4 & \text{if } 3 < x < 7 \\ 0 & \text{otherwise} \end{cases}$$

(b)

$$g(x) = xI_2(x) = \begin{cases} x & |x| < 2 \\ 0 & |x| > 2 \end{cases}$$

(c)

$$h(x) = \begin{cases} x - 1 & \text{if} 0 < x < 2 \\ 0 & \text{otherwise} \end{cases}$$

Compute the transform in two ways: first, using the definition and, second, using the result of Example 4.2.1(a) together with the shifting properties of Proposition 4.2.3. For f, g, and h compute

$$\|f\|^2 = \int_{-\infty}^{\infty} |f(x)|^2 \, dx$$

(a)

$$F(\omega) = \overline{\pi\omega} - \overline{\pi\omega}$$

(b)

$$F(\omega) = \; -$$

(c)

$$F(\omega) = \; -$$

3. The Laplace Transform

CHAPTER 5

Boundary-Value Problems and Initial-Boundary Value Problems on Unbounded Domains

1. Elementary Examples on $(-\infty, \infty)$

2. Examples on Semibounded Regions

3. Inhomogeneous Equations

4. Duhamel's Principle

CHAPTER 6

Uniqueness and Continuous Dependence on Data

1. Well-Posed Problems in Partial Differential Equations

2. Green's Identities and Energy Inequalities

3. Maximum-Minimum Principles

CHAPTER 7

First-Order Equations

1. Constant-Coefficient Advection Equation

2. Linear and Quasi-linear Equations

3. Conservation Law Equations

4. Generalized Solutions

5. Applications of Scalar Conservation Laws

6. Systems of First-Order Equations

CHAPTER 8

Finite Difference Methods for Parabolic Equations

1. Difference Formulas

2. Finite-Difference Equations for $u_t - a^2 u_{xx} = S$

3. Computational Methods

4. Use Algorithm 8.1 to approximate the solution of the initial-boundary-value problem

$$u_t - u_{xx} = -2e^{x-t}, 0 < x < 1,\ t > 0 \tag{3.1}$$

$$u(x, 0) = e^x, 0 < x < 1 \tag{3.2}$$

$$u(0, t) = e^{-t}, u(1, t) = e^{1-t}, t > 0 \tag{3.3}$$

(a) Choose $k = 0.0025$ and $nmax = 9$ (so $h = 0.1$) and compare the numerical and exact solutions, $u(x, t) = e^{x-t}$, at time $t = 0.5$.
(b) Choose $k = 0.01$ and $nmax = 9$ and explain the numerical results.

```
import numpy as np

# Forward Difference Method - Dirichlet Initial-Boundary-Value Problem
def algorithm_8_1(diffusivity,
                  endpoint,
                  time_step,
                  number_of_time_steps,
                  number_of_nodes,
                  right_side,
                  initial_condition,
                  boundary_condition_left,
                  boundary_condition_right):
    # Define a grid
    increment = endpoint / (number_of_nodes + 1)
    coefficient_r = diffusivity * time_step / increment ** 2
    if coefficient_r > 0.5:
        print("WARNING: algorithm_8_1 is unstable")

    # Initialize numerical solution
    t = np.zeros((number_of_time_steps + 1,))
    #x = np.zeros((1, number_of_nodes + 2))
    x = np.zeros((number_of_nodes + 2,))
    x[0] = 0
    #V = np.zeros((1, number_of_nodes + 2))
    V = np.zeros((number_of_nodes + 2,))
```

```
    V[0] = (boundary_condition_left(0) + initial_condition(0)) / 2
    for n in np.arange(number_of_nodes):
        x[n + 1] = x[n] + increment
        V[n + 1] = initial_condition(x[n + 1])
    x[number_of_nodes + 1] = endpoint
    V[number_of_nodes + 1] = (boundary_condition_right(0) + \
     initial_condition(endpoint)) / 2

    # Begin time stepping
    #U = np.zeros((1, number_of_nodes + 2))
    U = np.zeros((number_of_nodes + 2,))
    for j in np.arange(number_of_time_steps):
        # Advance solution one time step
        for n in np.arange(number_of_nodes):
            U[n + 1] = coefficient_r * V[n]
            U[n + 1] += (1 -  2 * coefficient_r) * V[n + 1]
            U[n + 1] += coefficient_r * V[n + 2]
            U[n + 1] += time_step * right_side(x[n + 1], t[j])
        t[j + 1] = t[j]  + time_step
        U[0] = boundary_condition_left(t[j + 1])
        U[number_of_nodes + 1] = boundary_condition_right(t[j + 1])
        # Output numerical solution
        # Prepare for next time step
        for n in np.arange(number_of_nodes + 2):
            V[n] = U[n]

    #x = x[1:-1]
    t = t[:-1]

    return U, x, t

import math

# right_side
def S(x, t):
    return -2.0 * math.e ** (x - t)

# initial_condition
def f(x):
    return math.e ** x

# boundary_condition_left
def p(t):
    return  math.e ** -t

# boundary_condition_right
def q(t):
    return math.e ** (1 - t)

# exact_answer
def u(x, t):
 return math.e ** (x - t)
```

```
a2 = 1 # diffusivity
L = 1 # endpoint
k = 0.0025 # time_step
nmax = 9 # number_of_nodes
end_time = 0.5
jmax = int(end_time / k) # number_of_time_steps

numerical_answer, x, t = algorithm_8_1(a2,
                        L,
                        k,
                        jmax,
                        nmax,
                        S,
                        f,
                        p,
                        q)
exact_answer =  u(x, t[-1])
answer_error = (numerical_answer - exact_answer) / (exact_answer)
answer_error = answer_error * 100

from matplotlib import pyplot as plt

fig, ax1 = plt.subplots()

ax1.set_xlabel('Position')
ax1.set_ylabel('Temperature')
ax1.plot(x, numerical_answer, 'r', label='Numerical Solution')
ax1.plot(x, exact_answer, 'g', label='Exact Solution')

ax2 = ax1.twinx()
ax2.set_ylabel('Percent Error')
ax2.plot(x, answer_error, 'b', label='Percent Error')

ax1.legend()
ax2.legend()

plt.show()
```

5. Use Algorithm 8.3 or 8.4 to approximate the solution of the initial-boundary-value problem

$$u_t - u_{xx} = -2e^{x-t}, 0 < x < 1,\ t > 0 \tag{3.4}$$

$$u(x, 0) = e^x, 0 < x < 1 \tag{3.5}$$

$$u(0, t) = e^{-t}, u(1, t) = e^{1-t}, t > 0 \tag{3.6}$$

(a) Choose $k = 0.0025$ and $nmax = 9$ (so $h = 0.1$) and compare the numerical and exact solutions, $u(x, t) = e^{x-t}$, at time $t = 0.5$.
(b) Choose $k = 0.01$ and $nmax = 9$ and compare the numerical and exact solutions at time $t = 0.5$.
(c) Choose $k = 0.01$ and $nmax = 99$ (so $h = 0.01$) and compare the numerical and exact solutions at time $t = 0.5$ at the positions $x = 0.1, 0.2, \ldots, 0.9$.

```
import numpy as np

# Solution of a Tridiagonal Linear System
def algorithm_8_2(a, # subdiagonal
                  b, # diagonal
                  c, # superdiagonal
                  d, # right-hand side
                  number_of_nodes=None):
    if number_of_nodes is None:
        number_of_nodes = d.size
    # Forward substitute to eliminate subdiagonal
    for n in np.arange(number_of_nodes - 1):
        ratio = a[n + 2] / b[n + 1]
        b[n + 2] = b[n + 2] - ratio * c[n + 1]
        d[n + 2] = d[n + 2] - ratio * d[n + 1]
    # Back substitude and store in solution array in d
    d[number_of_nodes] = d[number_of_nodes] / b[number_of_nodes]
    for l in np.arange(number_of_nodes - 1):
        n = number_of_nodes - (l + 1)
        d[n] = (d[n] - c[n] * d[n + 1]) / b[n]
    return d

# Backward Difference Method - Dirichlet Initial-Boundary-Value Problem
def algorithm_8_3(diffusivity,
                  endpoint,
                  time_step,
                  number_of_time_steps,
                  number_of_nodes,
                  right_side,
                  initial_condition,
                  boundary_condition_left,
                  boundary_condition_right):
    # Define a grid
    increment = endpoint / (number_of_nodes + 1)
    coefficient_r = diffusivity * time_step / increment ** 2

    # Initialize numerical solution
    t = np.zeros((number_of_time_steps + 1,))
    x = np.zeros((number_of_nodes + 2,))
    x[0] = 0
    U = np.zeros((number_of_nodes + 2,))
    U[0] = (boundary_condition_left(0) + initial_condition(0)) / 2
    for n in np.arange(number_of_nodes):
        x[n + 1] = x[n] + increment
        U[n + 1] = initial_condition(x[n + 1])
    U[number_of_nodes + 1] = (boundary_condition_right(0) + \
     initial_condition(endpoint)) / 2
    x[number_of_nodes + 1] = endpoint

    term_a = np.zeros((number_of_nodes + 2,))
    term_b = np.zeros((number_of_nodes + 2,))
    term_c = np.zeros((number_of_nodes + 2,))
    term_d = np.zeros((number_of_nodes + 2,))
```

```
    # Begin time stepping
    for j in np.arange(number_of_time_steps):
        # Define tridiagonal system
        t[j + 1] = t[j]  + time_step
        for n in np.arange(number_of_nodes):
            term_a[n + 1] = - coefficient_r
            term_b[n + 1] = 1 + 2 * coefficient_r
            term_c[n + 1] = - coefficient_r
            term_d[n + 1] = U[n + 1] + time_step * right_side(x[n + 1], t[j + 1])
        term_d[1] = term_d[1] + coefficient_r * boundary_condition_left(t[j + 1])
        term_d[number_of_nodes] = term_d[number_of_nodes] + \
         coefficient_r * boundary_condition_right(t[j + 1])
        # Advance solution one time step
        term_d = algorithm_8_2(term_a, term_b, term_c, term_d, number_of_nodes)
        for n in np.arange(number_of_nodes):
            U[n + 1] = term_d[n + 1]
        U[0] = boundary_condition_left(t[j + 1])
    # Output numerical solution
    U[0] = boundary_condition_left(t[j + 1])
    U[number_of_nodes + 1] = boundary_condition_right(t[j + 1])

    return U, x, t

import math

# right_side
def S(x, t):
    return -2.0 * math.e ** (x - t)

# initial_condition
def f(x):
    return math.e ** x

# boundary_condition_left
def p(t):
    return  math.e ** -t

# boundary_condition_right
def q(t):
    return math.e ** (1 - t)

# exact_answer
def u(x, t):
 return math.e ** (x - t)

a2 = 1 # diffusivity
L = 1 # endpoint
k = 0.0025 # time_step
nmax = 9 # number_of_nodes
end_time = 0.5
jmax = int(end_time / k) # number_of_time_steps

numerical_answer, x, t = algorithm_8_3(a2,
```

```
                        L,
                        k,
                        jmax,
                        nmax,
                        S,
                        f,
                        p,
                        q)
exact_answer = u(x, t[-1])
answer_error = (numerical_answer - exact_answer) / (exact_answer)
answer_error = answer_error * 100

from matplotlib import pyplot as plt

fig, ax1 = plt.subplots()

ax1.set_xlabel('Position')
ax1.set_ylabel('Temperature')
ax1.plot(x, numerical_answer, 'r', label='Numerical Solution')
ax1.plot(x, exact_answer, 'g', label='Exact Solution')

ax2 = ax1.twinx()
ax2.set_ylabel('Percent Error')
ax2.plot(x, answer_error, 'b', label='Percent Error')

ax1.legend()
ax2.legend()

plt.show()
```

6. Use Algorithm 8.5 to approximate the solution of the initial-boundary-value problem

$$u_t - u_{xx} = -2e^{x-t}, 0 < x < 1,\ t > 0 \tag{3.7}$$

$$u(x, 0) = e^x, 0 < x < 1 \tag{3.8}$$

$$u(0, t) = e^{-t}, u(1, t) = e^{1-t}, t > 0 \tag{3.9}$$

(a) Choose $k = 0.0025$ and $nmax = 9$ (so $h = 0.1$) and compare the numerical and exact solutions, $u(x, t) = e^{x-t}$, at time $t = 0.5$.
(b) Choose $k = 0.01$ and $nmax = 9$ and compare the numerical and exact solutions at time $t = 0.5$.
(c) Choose $k = 0.01$ and $nmax = 99$ (so $h = 0.01$) and compare the numerical and exact solutions at time $t = 0.5$.

```
import numpy as np

# Backward Difference Method - Neumann Initial-Boundary-Value Problem
def algorithm_8_5(diffusivity,
                  endpoint,
                  time_step,
                  number_of_time_steps,
                  number_of_nodes,
```

```
                right_side,
                initial_condition,
                boundary_condition_left,
                boundary_condition_right):
    # Define a grid
    increment = endpoint / (number_of_nodes - 1)
    coefficient_r = diffusivity * time_step / increment ** 2

    # Initialize numerical solution
    t = np.zeros((number_of_time_steps + 1,))
    x = np.zeros((number_of_nodes + 1,))
    x[0] = -increment
    U = np.zeros((number_of_nodes + 1,))
    for n in np.arange(number_of_nodes):
        x[n + 1] = x[n] + increment
        U[n + 1] = initial_condition(x[n + 1])

    term_a = np.zeros((number_of_nodes + 1,))
    term_b = np.zeros((number_of_nodes + 1,))
    term_c = np.zeros((number_of_nodes + 1,))
    term_d = np.zeros((number_of_nodes + 1,))
    # Begin time stepping
    for j in np.arange(number_of_time_steps):
        # Define tridiagonal system
        t[j + 1] = t[j]  + time_step
        for n in np.arange(number_of_nodes):
            term_a[n + 1] = - coefficient_r
            term_b[n + 1] = 1 + 2 * coefficient_r
            term_c[n + 1] = - coefficient_r
            term_d[n + 1] = U[n + 1] + time_step * right_side(x[n + 1], t[j + 1])
        term_d[1] = term_d[1] - \
         2 * increment * coefficient_r * boundary_condition_left(t[j + 1])
        term_d[number_of_nodes] = term_d[number_of_nodes] + \
         2 * increment * coefficient_r * boundary_condition_right(t[j + 1])
        term_c[1] = -2 * coefficient_r
        term_a[number_of_nodes] = -2 * coefficient_r
        # Advance solution one time step
        term_d = algorithm_8_2(term_a, term_b, term_c, term_d, number_of_nodes)
        for n in np.arange(number_of_nodes):
            U[n + 1] = term_d[n + 1]
    # Output numerical solution

    #t = t[:-1]
    x = x[1:]
    U = U[1:]

    return U, x, t

import math

# right_side
def S(x, t):
    return -2.0 * math.e ** (x - t)
```

```
# initial_condition
def f(x):
    return math.e ** x

# boundary_condition_left
def p(t):
    return  math.e ** -t

# boundary_condition_right
def q(t):
    return math.e ** (1 - t)

# exact_answer
def u(x, t):
 return math.e ** (x - t)

a2 = 1 # diffusivity
L = 1 # endpoint
k = 0.0025 # time_step
nmax = 9 # number_of_nodes
end_time = 0.5
jmax = int(end_time / k) # number_of_time_steps

numerical_answer, x, t = algorithm_8_5(a2,
                        L,
                        k,
                        jmax,
                        nmax,
                        S,
                        f,
                        p,
                        q)
exact_answer = u(x, t[-1])
answer_error = (numerical_answer - exact_answer) / (exact_answer)
answer_error = answer_error * 100

from matplotlib import pyplot as plt

fig, ax1 = plt.subplots()

ax1.set_xlabel('Position')
ax1.set_ylabel('Temperature')
ax1.plot(x, numerical_answer, 'r', label='Numerical Solution')
ax1.plot(x, exact_answer, 'g', label='Exact Solution')

ax2 = ax1.twinx()
ax2.set_ylabel('Percent Error')
ax2.plot(x, answer_error, 'b', label='Percent Error')

ax1.legend()
ax2.legend()
```

```
plt.show()
```

4. Fourier's Method for Difference Equations

5. Given the initial-boundary-value problem

$$u_t - u_{xx} = 0, 0 < x < 1,\ t > 0 \tag{4.1}$$

$$u(x, 0) = x^2, 0 < x < 1 \tag{4.2}$$

$$u_x(0, t) = 0 = u_x(1, t), t > 0 \tag{4.3}$$

and the grid $x_1 = 0$, $x_2 = 1/2$, and $x_3 = 1$, so $N = 3$,
(a) Use the matrix $\mathbf{F}_2$ to formulate a forward-in-time difference system.
(b) Use the matrix $\mathbf{F}_2$ to formulate a backward-in-time difference system.
(c) Use the matrix $\mathbf{F}_2$ to formulate a Crank-Nicolson difference system.
(d) Use Fourier's method to solve the system (a).
(e) Use Fourier's method to solve the system (b).
(f) Use Fourier's method to solve the system (c).

```
import numpy as np
import math

def node_points(endpoint, number_of_nodes):
    increment = endpoint / (number_of_nodes - 1)
    x = [i * increment for i in range(number_of_nodes)]
    #x = np.array(x)
    return x

def eigenvalue(n, N):
    return 2 * (1 - math.cos((n - 1) * math.pi / (N - 1)))

# eigenvector Vn
def eigenvector_v(n, N, L):
    #x = [((i + 1) - 1) / (N - 1) for i in range(N)]
    x = node_points(L, N)
    output = [math.cos((n - 1) * math.pi * y) for y in x]
    output = np.array(output)
    return output

# eigenvector Wn
def eigenvector_w(n, N, L):
    #x = [((i + 1) - 1) / (N - 1) for i in range(N)]
    x = node_points(L, N)
    output = [2 * math.cos((n - 1) * math.pi * y) for y in x]
    output[0] = output[0] / 2
    output[-1] = output[-1] / 2
    output = np.array(output)
    return output

# initial_condition
def f(x):
    return x ** 2
```

```
def coefficients(N, initial_condition, L):
    output = [coefficient(i + 1, N, initial_condition, L) for i in range(N)]
    output = np.array(output)
    return output

def coefficient(n, N, initial_condition, L):
    x = node_points(L, N)
    initial = [initial_condition(y) for y in x]
    initial = np.array(initial)
    output = np.dot(initial, eigenvector_w(n, N, L))
    output = output / np.dot(eigenvector_v(n, N, L), eigenvector_w(n, N, L))
    return output

# Forward-in-time (FIT)
# Neumann-Neumann (NN) Initial-Boundary-Value Problem
def solve_nn_fit(endpoint,
                 time_step,
                 number_of_time_steps,
                 number_of_nodes,
                 right_side,
                 initial_condition,
                 boundary_condition_left,
                 boundary_condition_right):
    j = number_of_time_steps
    increment = endpoint / (number_of_nodes - 1)
    coefficient_r = time_step / increment ** 2
    matrix_b = lambda number: 1 - coefficient_r * number
    matrix_a = lambda number: 1
    U = np.zeros((number_of_nodes,))
    for n in np.arange(number_of_nodes):
        l = eigenvalue(n + 1, number_of_nodes)
        V = eigenvector_v(n + 1, number_of_nodes, endpoint)
        c_n = coefficient(n + 1, number_of_nodes, initial_condition, endpoint)
        U = U + (matrix_b(l) / matrix_a(l)) ** j * c_n * V
    return U

# Backward-in-time (BIT)
# Neumann-Neumann (NN) Initial-Boundary-Value Problem
def solve_nn_bit(endpoint,
                 time_step,
                 number_of_time_steps,
                 number_of_nodes,
                 right_side,
                 initial_condition,
                 boundary_condition_left,
                 boundary_condition_right):
    U = np.zeros((number_of_nodes,))
    return U

# Crank-Nicolson (CN)
# Neumann-Neumann (NN) Initial-Boundary-Value Problem
def solve_nn_cn(endpoint,
```

```
                    time_step,
                    number_of_time_steps,
                    number_of_nodes,
                    right_side,
                    initial_condition,
                    boundary_condition_left,
                    boundary_condition_right):
    U = np.zeros((number_of_nodes,))
    return U

L = 1 # endpoint
k = 0.0025 # time_step
nmax = 3 # number_of_nodes
end_time = 0.5
jmax = int(end_time / k) # number_of_time_steps


answer_a = solve_nn_fit(L,
                    k,
                    jmax,
                    nmax,
                    0,
                    f,
                    0,
                    0)
```

(a)

$$A(\mathbf{F}_2)\mathbf{U}^{j+1} = B(\mathbf{F}_2)\mathbf{U}^j \tag{4.4}$$

$$\mathbf{U}^0 = \mathbf{f} \tag{4.5}$$

$$c_n^0 = \frac{\mathbf{f} \cdot \mathbf{W}_n}{\mathbf{V}_n \cdot \mathbf{W}_n} \tag{4.6}$$

$$A(\lambda) = 1 \tag{4.7}$$

$$B(\lambda) = 1 - r\lambda \tag{4.8}$$

(b)

$$A(\mathbf{F}_2)\mathbf{U}^{j+1} = B(\mathbf{F}_2)\mathbf{U}^j \tag{4.9}$$

$$\mathbf{U}^0 = \mathbf{f} \tag{4.10}$$

$$c_n^0 = \frac{\mathbf{f} \cdot \mathbf{W}_n}{\mathbf{V}_n \cdot \mathbf{W}_n} \tag{4.11}$$

$$A(\lambda) = 1 + r\lambda \tag{4.12}$$

$$B(\lambda) = 1 \tag{4.13}$$

(c)

$$A(\mathbf{F}_2)\mathbf{U}^{j+1} = B(\mathbf{F}_2)\mathbf{U}^j \tag{4.14}$$

$$\mathbf{U}^0 = \mathbf{f} \tag{4.15}$$

$$c_n^0 = \frac{\mathbf{f}\cdot\mathbf{W}_n}{\mathbf{V}_n\cdot\mathbf{W}_n} \tag{4.16}$$

$$A(\lambda) = 1 + r\lambda/2 \tag{4.17}$$

$$B(\lambda) = 1 - r\lambda/2 \tag{4.18}$$

6. Given the initial-boundary-value problem

$$u_t - u_{xx} = 0, 0 < x < 1,\ t > 0 \tag{4.19}$$

$$u(x,0) = x^2, 0 < x < 1 \tag{4.20}$$

$$u_x(0,t) = 0 = u_x(1,t), t > 0 \tag{4.21}$$

and the grid $x_1 = 0$, $x_2 = 1/3$, and $x_3 = 2/3$, so $N = 3$,
(a) Use the matrix $\mathbf{F}_2$ to formulate a forward-in-time difference system.
(b) Use the matrix $\mathbf{F}_2$ to formulate a backward-in-time difference system.
(c) Use the matrix $\mathbf{F}_2$ to formulate a Crank-Nicolson difference system.
(d) Use Fourier's method to solve the system (a).
(e) Use Fourier's method to solve the system (b).
(f) Use Fourier's method to solve the system (c).

(a)

$$A(\mathbf{F}_3)\mathbf{U}^{j+1} = B(\mathbf{F}_3)\mathbf{U}^j \tag{4.22}$$

$$\mathbf{U}^0 = \mathbf{f} \tag{4.23}$$

$$A(\lambda) = 1 \tag{4.24}$$

$$B(\lambda) = 1 - r\lambda \tag{4.25}$$

$$c_n^j = \frac{\mathbf{U}^j\cdot\mathbf{W}_n}{\mathbf{V}_n\cdot\mathbf{W}_n} \tag{4.26}$$

$$= \left[\frac{B(\lambda_n)}{A(\lambda_n)}\right]^j c_n^0 \tag{4.27}$$

$$= \left[\frac{B(\lambda_n)}{A(\lambda_n)}\right]^j \frac{\mathbf{f}\cdot\mathbf{W}_n}{\mathbf{V}_n\cdot\mathbf{W}_n} \tag{4.28}$$

$$\mathbf{U}^j = \sum_{n=0}^{N} c_n^j \mathbf{V}_n \tag{4.29}$$

$$= \sum_{n=0}^{N} \left[\frac{B(\lambda_n)}{A(\lambda_n)}\right]^j c_n^0 \mathbf{V}_n \tag{4.30}$$

$$= \sum_{n=0}^{N} (1 - r\lambda_n)^{-j} c_n^0 \mathbf{V}_n \tag{4.31}$$

(4.32)

(b)

$$A(\mathbf{F}_3)\mathbf{U}^{j+1} = B(\mathbf{F}_3)\mathbf{U}^j \tag{4.33}$$

$$\mathbf{U}^0 = \mathbf{f} \tag{4.34}$$

$$c_n^0 = \frac{\mathbf{f}\cdot\mathbf{W}_n}{\mathbf{V}_n\cdot\mathbf{W}_n} \tag{4.35}$$

$$A(\lambda) = 1 + r\lambda \tag{4.36}$$

$$B(\lambda) = 1 \tag{4.37}$$

(c)

$$A(\mathbf{F}_3)\mathbf{U}^{j+1} = B(\mathbf{F}_3)\mathbf{U}^j \tag{4.38}$$

$$\mathbf{U}^0 = \mathbf{f} \tag{4.39}$$

$$c_n^0 = \frac{\mathbf{f}\cdot\mathbf{W}_n}{\mathbf{V}_n\cdot\mathbf{W}_n} \tag{4.40}$$

$$A(\lambda) = 1 + r\lambda/2 \tag{4.41}$$

$$B(\lambda) = 1 - r\lambda/2 \tag{4.42}$$

7. Given the initial-boundary-value problem

$$u_t - u_{xx} = 0, 0 < x < 1,\, t > 0 \tag{4.43}$$

$$u(x,0) = x, 0 < x < 1 \tag{4.44}$$

$$u_x(0,t) = 0 = u_x(1,t), t > 0 \tag{4.45}$$

and the grid $x_1 = 0$, $x_2 = 1/4, x_3 = 1/2, x_4 = 3/4$, and $x_5 = 1$, so $N = 5$,
(a) Use the matrix $\mathbf{F}_2$ to formulate a forward-in-time difference system.
(b) Use the matrix $\mathbf{F}_2$ to formulate a backward-in-time difference system.
(c) Use the matrix $\mathbf{F}_2$ to formulate a Crank-Nicolson difference system.
(d) Use Fourier's method to solve the system (a).
(e) Use Fourier's method to solve the system (b).
(f) Use Fourier's method to solve the system (c).

5. Stability of Finite-Difference Methods

6. Difference Methods in Two Space Variables

7. Conservation Law Difference Equations

8. Material Balance Difference Equation in Two Space Variables

CHAPTER 9

Numerical Solutions of Hyperbolic Equations

1. Difference Methods for a Scalar Initial-Value Problem

1. Modify Algorithm 9.1 to implement
(a) FTBS method
(b) FTFS method
(c) Lax-Friedrichs method
(d) leapfrog method

2. Approximate the solution of the initial-value problem of Example 9.1.3 on the interval $0 \le x \le 1$ for $0 \le t_j \le 1.5$ with $h = 0.1$ and $k = 0.075$ using
(a) FTBS method
(b) Lax-Friederichs method
(c) leapfrog method

3. Repeat exercise 2 with $h = 0.1$ and $k = 0.1$.

4. Given the initial-value problem

$$u_x + u_t = 2xt + x^2, -\infty < x < \infty, t > 0 \tag{1.1}$$

$$u(x, 0) = 0, -\infty < x < \infty \tag{1.2}$$

(1.3)

whose exact solution is $u(x, t) = x^2t$. Find the local truncation error for this problem when using **(a)** FTBS method
(b) Lax-Wendroff method

8. Consider the initial-value problem

$$u_x + u_t = 0, \qquad -\infty < x < \infty, t > 0 \tag{1.4}$$

$$u(x, 0) = \begin{cases} 1, & |x| < 0.5, \\ 0, & |x| \ge 0.5, \end{cases} \qquad -\infty < x < \infty \tag{1.5}$$

2. Difference Methods for a Scalar Initial-Boundary-Value Problem

3. Scalar Conservation Laws

4. Dispersion and Dissipation

5. Systems of Equations

6. Second-Order Equations

7. Method of Characteristics

CHAPTER 10

Finite-Difference Methods for Elliptic Equations

1. Difference Equations for Elliptic Equations

2. Direct Solution of Linear Equations

3. Fourier's Method

4. Iterative Methods

5. Convergence of Iterative Methods

Bibliography

[1] DuChateau, P., Zachmann, D. *Applied Partial Differential Equations*, Dover Publications Inc, Mineola, New York, 1989