Solutions Manual to Applied Partial Differential Equations

Stewart Nash

(S. Nash) 362 Lowell Street
Andover, MA 01810
Email address, S. Nash: Stewart.M.Nash@gmail.com
URL: http://www.pervigilent.com

Dedicated to those who came before me and those who will come after.

The Author thanks DuChateau and Zachmann for such a good book. The internet for endless resources.

Abstract. This work consists of solutions to the exercises from the volume "Applied Partial Differential Equations" by Paul DuChateau and David Zachmann.

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. 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

$$u_n^{j+1} = 0.1u_{n+1}^j + 0.8u_n^j + 0.1u_{n-1}^j \tag{1.1}$$

Suppose

$$u_n^0 \begin{cases} 1 & \text{for } n = 4, 5, 6 \\ 0 & \text{for all other } n \end{cases} \tag{1.2}$$

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

$$u_0^j = 1 \,\text{and} \quad u_{10}^j = -1 \quad \text{for all}\, j > 0 \tag{1.3}$$

and

$$u_n^0 = 0 \quad \text{for all}\, n \tag{1.4}$$

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)))
```

The front matter has a number of sample entries that you should replace with your own.

Replace this text with the body of your book. Do not delete the `mainmatter` TeX field found above in a paragraph by itself or the numbering of different objects will be wrong.

The typesetting specification selected by this document uses the default class options. There are, however, a number of class options. The available options include setting the paper size and the point size of the font used in the body of the document etc. Details are given as comments right after the `documentclass` command.

CHAPTER 2

Finite Difference Methods for Parabolic Equations

1. Computational Methods

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

$$u_t - u_x x = -2e^{x-t}, 0 < x < 1,\ t > 0 \tag{1.1}$$

$$u(x,0) = e^x, 0 < x < 1 \tag{1.2}$$

$$u(0,t) = e^{-t}, u(1,t) = e^{1-t}, t > 0 \tag{1.3}$$

(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

$$u_t - u_x x = -2e^{x-t}, 0 < x < 1,\ t > 0 \tag{1.4}$$

$$u(x, 0) = e^x, 0 < x < 1 \tag{1.5}$$

$$u(0, t) = e^{-t}, u(1, t) = e^{1-t}, t > 0 \tag{1.6}$$

(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 $x = 0.1, 0.2, \ldots, 0.9$.

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

$$u_t - u_x x = -2e^{x-t}, 0 < x < 1,\ t > 0 \tag{1.7}$$

$$u(x,0) = e^x, 0 < x < 1 \tag{1.8}$$

$$u(0,t) = e^{-t}, u(1,t) = e^{1-t}, t > 0 \tag{1.9}$$

(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

# 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
        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])
    x[number_of_nodes] = endpoint
    U[number_of_nodes + 1] = (boundary_condition_right(0) + initial_condition(endpoint)) / 2

    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[
        # 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])

    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_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()
```

CHAPTER 3

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 $0 \leq x \leq 1$ for $0 \leq t_j \leq 1.5$ with $h = 0.1$ and $k = 0.075$ using
(a) FTBS method
(b) Lax-Friederichs method
(c) leapfrog method

Bibliography

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