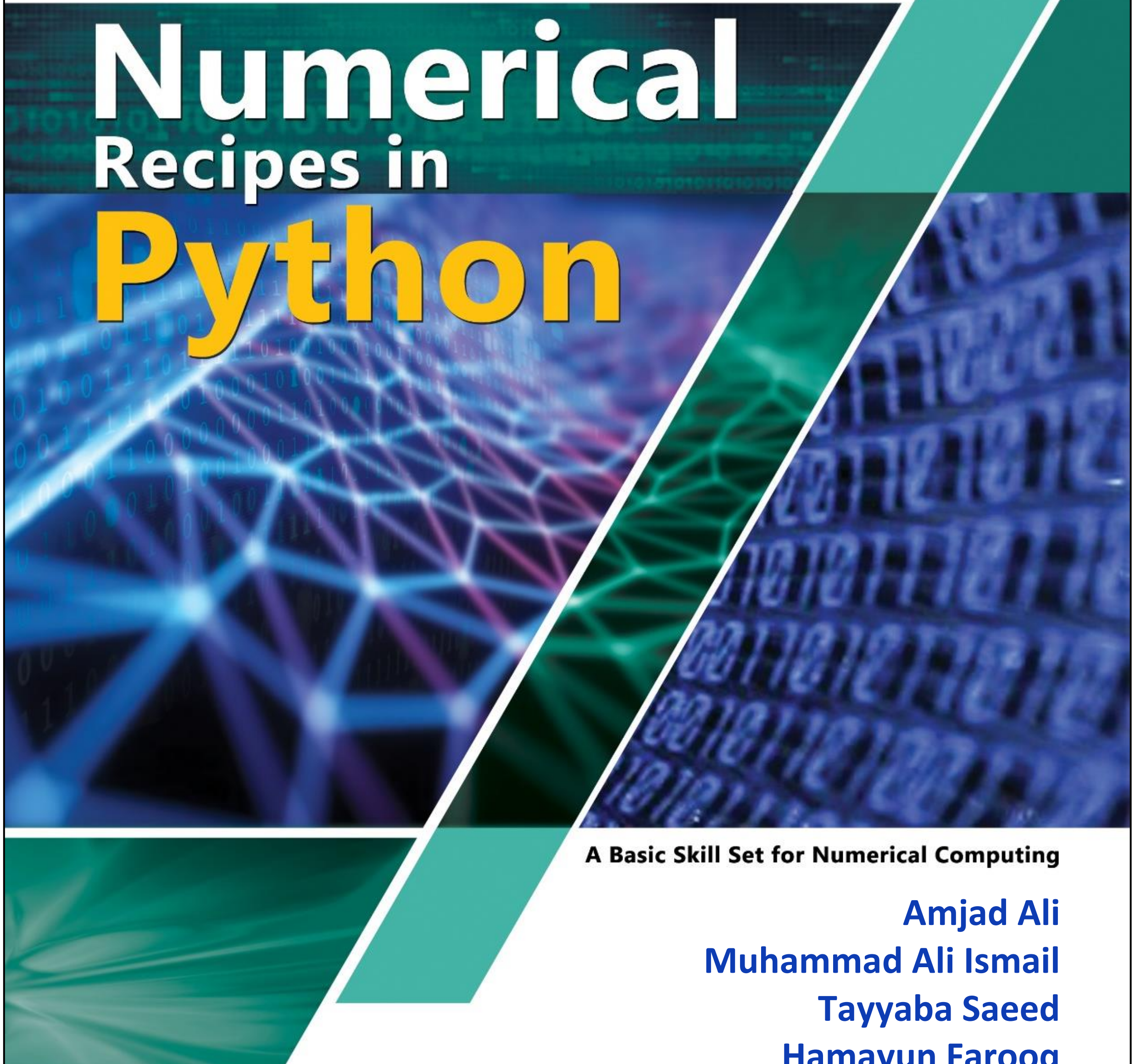
4th Edition

Laboratory Manual of Simplified Numerical Analysis (Python Version)

Numerical Recipes in Python



A Basic Skill Set for Numerical Computing

Amjad Ali
Muhammad Ali Ismail
Tayyaba Saeed
Hamayun Farooq
Uzair Abid

Numerical Recipes in Python ISBN: 978-969-7821-11-2

Laboratory Manual of Simplified Numerical Analysis (Python Version)

A Companion book of the principal book:

Simplified Numerical Analysis (Fourth Edition)
©2023, Amjad Ali, Ph.D. (The Principal Author)
ISBN: 978-969-7821-14-3

Accessible through: www.timerenders.com.pk

For availability of the codes, please visit:

GitHub - DrAmjadAli11/SimplifiedNumericalAnalysis
https://github.com/DrAmjadAli11/SimplifiedNumericalAnalysis

Laboratory Manual of Simplified Numerical Analysis
(Python Version)

Numerical Recipes in Python

Fourth Edition

Amjad Ali, Ph.D.
Bahauddin Zakariya University (BZU), Multan

Muhammad Ali Ismail, Ph.D.
Exascale Open Data Analytics Lab
National Centre for Big Data and Cloud Computing (NCBC)
NED University of Engineering and Technology, Karachi

Tayyaba Saeed, M.Phil.
Bahauddin Zakariya University (BZU), Multan

Hamayun Farooq, Ph.D.
Government Degree College, Muzaffar Garh

Uzair Abid, M.S.
Exascale Open Data Analytics Lab
NCBC, NED University of Engineering and Technology, Karachi

Esteemed Panel of the Supporters

Ms. Syeda Zahra Kazmi, Bahauddin Zakariya University, Multan
Ms. Anam Zahra, Bahauddin Zakariya University, Multan
Ms. Noreen Ilyas, Bahauddin Zakariya University, Multan

Numerical Recipes in Python ISBN: 978-969-7821-11-2

Laboratory Manual of Simplified Numerical Analysis (Python version)

A Companion book of the principal book:

Simplified Numerical Analysis (Fourth Edition)
©2023, Amjad Ali, Ph.D. (The Principal Author)
ISBN: 978-969-7821-14-3

Please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

For codes, please visit: https://github.com/DrAmjadAli11/SimplifiedNumericalAnalysis

Typeset: Mostly personally by **Dr. Amjad Ali** (The Principal Author), also contributed by the supporters.

Title Design: **Mr. Muhammad Rizwan Qadeer** (mrizwanqadeer@gmail.com)

The proofreading is powered by various **AI-driven** proprietary software.

TimeRender Publishers www.timerender.com.pk

Distribution Point:
238-B (PRIDE), Near Girls Comprehensive School, Gulgasht Colony, Multan, Pakistan.
Cell Phone: +923486981925, timerenderpublishers@gmail.com

Table of Contents

Numerical

Analysis

is

the

mathematics

of

Scientific

Computing

Chapter **1**

Preliminary Concepts in Numerical Analysis

To unleash the topics of this chapter, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

■■■

Computing Resources

The numerical methods are devised just to be used on computers. It makes no sense to study a numerical method without considering its practicality using some computing tools. A variety of numerical computing tools, both freeware and proprietary, are available. The students are advised to understand the algorithmic (step-by-step) style of the numerical methods they learn. This book suggests the following resources for beginners.

(1) **C++**: The numerical methods can be programmed in any programming language, especially C++, FORTRAN, and Python. The book discusses a wide variety of C++ programs of the numerical methods in this book. One can modify the as per need. Several C++ IDEs (Integrated Development Environments) are available, such as Dev-C++, and Code::Blocks for Windows and GNU-C++ for Linux operating system. One can even find C++ Apps (apps is an acronym for computer application software) for Android or iOS devices. Some online C++ IDEs are also available, which can be used for executing C++ programs without installing them.

(2) **Python**: There are several free Python IDEs available for the Desktop use (such as Spyder, Jupyter, and PyCharm) or On-line use (such as Google Colab). It is quite a pertinent skill of the day that the students of computational sciences are familiar with programming in Python. The companion website of this book (www.timerender.com.pk) shares a Python Library having a variety of codes for the numerical methods discussed in this book.

(3) **MATLAB®**: It is a proprietary software, by The MathWorks, Inc., available in both Desktop and Online versions. MATLAB® offers a wide variety of built-in functions and programming capabilities for mathematical computations (both symbolic and numeric, although more suitable and expert for numeric computations), for all modern areas of science and engineering. The book discusses a wide variety of MATLAB® programs and MATLAB® built-in functions for the numerical methods in this book.

(4) **GNU-Octave**: It is an open-source (and freeware) version of MATLAB®, available in both Desktop and Online versions. Most of the MATLAB® codes and built-in functions discussed in this book can be executed in GNU-Octave and Octave-online.

(5) **MATHEMATICA®:** It is a proprietary software by Wolfram Research. It is one of the best Computer Algebra Systems (CAS) available. It offers an extensive variety of built-in functions and programming capabilities for mathematical computations (both symbolic and numeric), for all modern areas of science and engineering.

(6) **MAPLE®**: It is a proprietary software by Maplesoft for mathematical computations (both symbolic and numeric), for all modern areas of science and engineering. It is also one of the best Computer Algebra Systems (CAS).

(7) **Spread-Sheet**: A spread-sheet software (such as Excel by Microsoft®) can be used for computations involved in simple numerical methods. The companion website of this book (www.timerender.com.pk) may shares a spread-sheet workbook having a variety of sheets for most of the numerical methods discussed in this book.

(8) **Various Math Solver Tools: Wolfram|Alpha**, **Symbolab**, and **Microsoft® Math Solver** are three of the advanced tools for math education to be used as calculators. These are extensive, feature-rich, online tools, accessible both through the web browser and the relevant android/iOS apps. These tools provide automated step by step solutions to algebra and calculus problems covering from middle school through college. The premier versions of these tools are freely available, whereas professional (pro) versions are not free.

(9) **Various Other Online Tools/Websites:** There are various other online tools and websites that offer basic computing facilities for numerical and symbolic computations. Examples include:

- **AtoZmath.com** [https://atozmath.com/]
- **CalculatorSoup®** [https://www.calculatorsoup.com/].
- **Keisan - CASIO®** [https://keisan.casio.com/]

■■■

Question 06: What are the significant figures (or significant digits) of an approximate number?

Significant figures of a number (that approximates a true value) are the digits that are used to express the number meaningfully. The significant figures are counted for a number that approximates some other number to express the degree of precision in the approximate number.

The significant figures begin with the leftmost nonzero digit and end with the rightmost correct digit. The rightmost zeros, which are exact, are also significant. That is,

- All the nonzero digits ($\text{i.e.}, 1, 2, 3, \cdots, 9$) are significant.
- Zeroes appearing anywhere between two nonzero digits are significant (e.g., in 3005.00102 there are nine significant digits).
- Leading zeros (i.e., left to the first nonzero digit) are not significant (e.g., the number 0.000081 has only two significant digits, namely 8 and 1). The leading zeros are used to fix the decimal place.
- Trailing zeroes are significant if they are exact with regard to some true value. Trailing zeros may or may not be significant. It depends on the context; how the number is approximated or obtained by rounding-off some other number.

■

Remark: The significant figures of a number can easily be identified by using its normalized scientific notation. The digits in the fractional part (or mantissa) are regarded as significant figures. For example, each of the numbers 42.134, 6.0013, and 0.0015784 has five significant figures, which can be identified easily by converting these numbers into their **normalized scientific notation** as:

$$\begin{aligned} 42.134 &= 0.42134 \times 10^2 \\ 6.0013 &= 0.60013 \times 10 \\ 0.0015784 &= 0.15784 \times 10^{-2} \end{aligned}$$

■

Remarks: For the following, set the rounding rule (like MS Excel and Python) that “to round a number to k decimal places, if the $(k + 1)$th digit is 5 or greater than 5, then add 1 to the kth digit.”

- 6500 has 2 significant figures (i.e., the digits 6 and 5) if it has been obtained by rounding-off a number to the nearest 100 (e.g., by rounding-off the numbers 6497 or 6543.88 to the nearest hundred). In fact, any number in the interval $[6450, 6550)$ gives 6500, when rounded to the nearest 100.

- 6500 has 3 significant figures (i.e., the digits 6, 5, and the following 0) if it has been obtained by rounding-off a number to the nearest 10 (e.g., by rounding-off the numbers 6497 or 6504.99 to the nearest ten). In fact, any number in the interval $[6495, 6505)$ gives 6500, when rounded to the nearest 10.

- 6500 has 4 significant figures if it has been obtained by rounding-off a number to the nearest whole number (e.g., by rounding-off the numbers 6499.8 or 6500.47 to the nearest whole number). In fact, any number in the interval [6499.5, 6500.5) gives 6500, when rounded to the nearest whole number.

- 70500 has at least 3 significant figures (i.e., the digits 7, 5, and the 0 in between 7 and 5). Depending upon the context, as just explained, it may have 3 to 5 significant figures.

- 0.00364300 has 4 significant figures (i.e., the digits 3, 6, 4, and 3) if it has been obtained by rounding-off a number to 4 significant figures (e.g., by rounding-off the numbers 0.003642859 or 0.0036432099 to 4 significant figures). Usually, in that case, the approximate number is written as 0.003643, without any non-significant trailing zero. In fact, any number in the interval [0.0036425,0.0036435) gives 0.003643, when rounded to 4 significant figures.

- 0.00364300 has 5 significant figures (i.e., the digits 3, 6, 4, 3, and the following 0) if it has been obtained by rounding-off a number to 5 significant figures (e.g., by rounding-off the numbers 0.003642978001 or 0.003643049 to 5 significant figures). Usually, in that case, the approximate number is written as 0.0036430, without any non-significant trailing zero. In fact, any number in the interval [0.00364295,0.00364305) gives 0.0036430, when rounded to 5 s.f.

- 0.00364300 has 6 significant figures (i.e., the digits 3, 6, 4, 3, and the following two 0s) if it has been obtained by rounding-off a number to 6 significant figures (e.g., by rounding-off the numbers 0.003642998001 or 0.003643001 to 6 significant figures). In fact, any number in the interval [0.003642995,0.003643005) gives 0.00364300, when rounded to 6 significant figures.

■

Remark:

An approximation x^* to a number x is called accurate to t significant figures if there are exactly t digits in the mantissa of x^* that agree with the first t digits of the mantissa of x, where x has the same exponent as x^*. Suppose that the number x is represented in the following form

$$x \quad = \quad \pm 0.\, d_1 d_2 d_3 \cdots d_t d_{t+1} \cdots \times 10^e$$

Then, the number x^* is accurate to t significant figures to the number x if it can be written in the following form

$$x^* \quad = \quad \pm 0.\, d_1 d_2 d_3 \cdots d_t d'_{t+1} \cdots \times 10^e$$

■

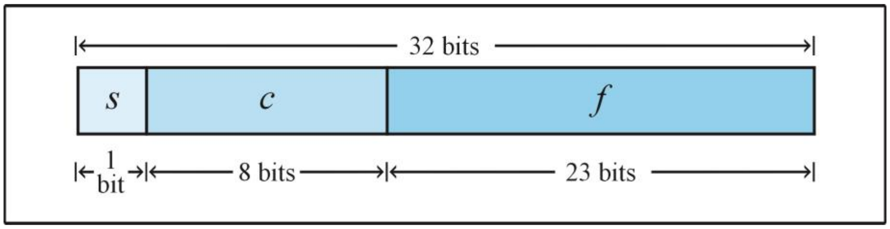


Fig. (1.3): According to the IEEE 754 standard, single-precision floating point representation of a binary real number $x = \pm 1.b_2 b_3 b_4 \cdots \times 2^e$ is $(1 - 2s) \times 2^{c-127} \times (1 + f)$.

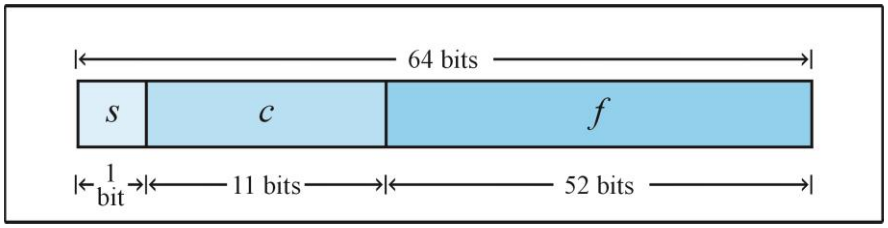


Fig. (1.4): According to the IEEE 754 standard, double-precision floating point representation of a binary real number $x = \pm 1.b_2 b_3 b_4 \cdots \times 2^e$ is $(1 - 2s) \times 2^{c-1023} \times (1 + f)$.

Here, s is used for the ***sign*** of the number (0 means positive, 1 means negative). c in the exponent is called the ***biased exponent***. f is the mantissa minus 1 (the hidden bit).

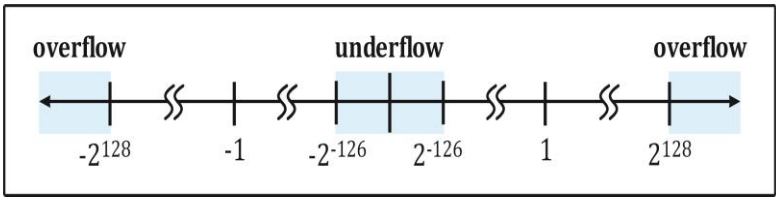


Fig. (1.5): Overflow/Underflow for single-precision floating-point representation

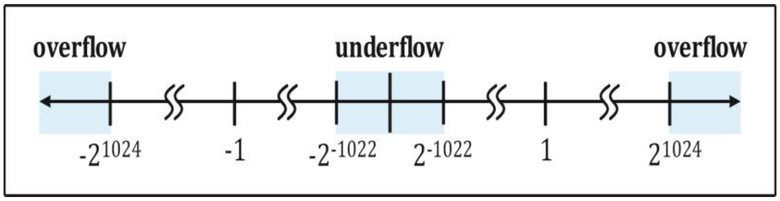


Fig. (1.6): Overflow/Underflow for double-precision floating-point representation

■■■

Chapter Summary

- The **numerical methods** obtain some approximate solution of the problems, usually in the numeric form, in contrast to the **analytic** or **exact methods**, which obtain the exact solution of the problem.
- **Numerical Analysis** is the field of deriving, analyzing, and implementing the numerical methods.
- The most common approach followed by the numerical methods is the **iterative** approach. According to this, choose an initial approximation or guess to the solution and apply a set of simple computational steps to obtain a better approximation. Repeatedly apply the same set of steps to the better approximations, ultimately obtaining a sufficiently accurate solution and then stop the repetition. Each course of repetition of the set of computational steps is called **iteration**. Geometrically, a root of an equation $f(x) = 0$ is the point where the graph of $f(x)$ intersects the x-axis.
- For selecting a numerical method from several choices, the characteristics of ***accuracy***, ***efficiency***, and ***robustness*** are taken into consideration.
- The numerical analysis may be regarded as the “mathematics of scientific computing”.
- Errors can be quantified as:
    - Absolute Error $= |\text{True value} - \text{Approximate value}|$
    - Relative Error $= \frac{\text{absolute error}}{|\text{True value}|} = \frac{|\text{True value}-\text{Approximate value}|}{|\text{True value}|}$
    - Percentage Relative Error $= \frac{\text{absolute error}}{|\text{True value}|} \times 100\ \%$
- The errors can be categorized in three major categories in regard to their sources: **Data Error** or **Inherent Error** (quite unrelated to the numerical methods; occur as blunders, mistakes, model simplification, or data uncertainty), **Round-off Error** (occurs due to number approximation by humans and computers), **Truncation Error** (occurs due to approximation of a mathematical procedure to avoid insignificance), and **Discretization error** (occurs due to approximation of a continuous function by a set of discrete data points).
- **Significant figures** of a real number (which is an approximation of the true value) are the digits that are used to express the number meaningfully. The significant digits begin with the leftmost nonzero digit and end with the rightmost correct digit. The rightmost zeros, which are exact are also significant.
- An approximation x^* to a number x is called accurate to t significant figures if there are exactly t digits in the mantissa of x^* that agreed with the first t digits of the mantissa of x having the same exponent or characteristics.
- **Accuracy** of an approximate value is a measure of how much the approximate value agrees with the true value. **Precision**, on the other hand, has nothing to do with how much the approximate value agrees with the true value. Precision is only concerned about the size of the number.
- The following four are the commonly used number systems, even supported by the computer architectures.

Decimal number system (base 10)	Binary number system (base 2)

Octal number system (base 8) Hexadecimal number system (base 16)

- Any nonzero real decimal number x can be represented in floating-point form: $x = \pm 0.d_1 d_2 d_3 \cdots \times 10^e$. Here $d_i, i = 1, 2, \cdots$ are digits from 0 to 9 with $d_1 \neq 0$, called most significant digit and e is an integer that might be positive, negative or zero, called an ***exponent*** or *characteristic*. The number $0.d_1 d_2 d_3 \cdots$, may be denoted by m, is called the finite normalized ***mantissa***. For numbers in the decimal system with base $10, \frac{1}{10} \leq m < 1$. That is, $m \in \left[\frac{1}{10}, 1\right)$.

- For numbers in the binary system, the floating-point representation of a number x can be given by, $x = \pm 0.b_1 b_2 b_3 \cdots \times 2^e = \pm m \times 2^e$, were each of b_i is a bit, either 0 or 1, with $b_1 \neq 0$, and $\frac{1}{2} \leq m < 1$.

- The numbers that are representable precisely in a computer are called ***machine numbers***. The real numbers with a non-terminating fractional part (such as 1/3) cannot be represented, precisely. So many other numbers (for example, 0.01) also has not a precise representation in computer (i.e., a machine number).

- If the number lies within the allowable range of the possible numbers according to the precision level of the computer, then it is rounded to a nearby machine number (incurring the round-off error) for storing it. The rounding options involve ***correct rounding*** (round to nearest machine number), *rounding up*, *rounding down* or towards zero, etc.

- There are commonly two ways to terminate the mantissa of a number to obtain its nearest machine number, namely, correct ***chopping*** and correct ***rounding***. The chopping or rounding of the number to the nearest machine number (representable in a computer) for representation in computers (for storage or for using in computations) causes the error in a number called the ***round-off error***.

- The floating-point form of a number x representable in a computer can be regarded as consisting of the three parts: $x \quad = \quad \pm m \times \beta^e \quad = \quad \boldsymbol{sign} \times \boldsymbol{mantissa} \times (base)^{\boldsymbol{exponent}}$

  The sign is either positive (+) or negative (−), the finite normalized mantissa is from the interval $\left[\frac{1}{\beta}, 1\right)$, and the integer exponent either positive, negative, or zero as a power of the base.

- An account on the **IEEE Binary Floating-Point Arithmetic Standard 754-1985** for representing the real numbers in computers can be found under Question 13 in this chapter.

- If a number $\boldsymbol{x}^*$ is accurate to $\boldsymbol{t}$ significant figures in approximating a number $\boldsymbol{x}$ then the relative error is bounded above by 5×10^{-t}. That is, $\frac{|x - x^*|}{|x|} \quad \leq \quad 5 \times 10^{-t}$

- If an iterative process is to be stopped when the successive approximations become accurate to t significant figures, the relative error bound might be set as 5×10^{-t}. Thus, the relative error is computed after every iteration using the result of the current iteration and that of the previous iteration. If the relative error is smaller than the bound of 5×10^{-t}, then it ensures that the approximation the accurate to t significant digits.

- Whenever two nearly equal numbers are subtracted, some loss of significance might occur. The risk of loss of significance can be eliminated by avoiding the subtraction through some mathematical manipulation.

Chapter Exercises

Exercise 01: Compute the absolute error E_a and relative error E_r in an approximation of x by x^*

(i) $x = \log_{10} 2, x^* = 0.301$ (ii) $x = 17/6, x^* = 2.8333$

(iii) $x = \sqrt{\pi}, x^* = 1.77245$ (iv) $x = e^{-1}, x^* = 0.36787$

Exercise 02: Write the following numbers in floating-point form and identify their mantissa and exponent:

(i) $x = -23.500128$ (ii) $x = 658.000012$ (iii) $x = 0.010023$

(iv) $x = -0.0000782$ (v) $x = \dfrac{1}{234.24}$ (vi) $x = 541000$

Exercise 03: Simplify the following expression by performing the computations

(a) Exactly
(b) Using four-digit chopping arithmetic
(c) Using four-digit rounding arithmetic
(d) Compute the relative errors

(i) $\dfrac{7}{4} - \dfrac{5}{3}$ (ii) $\dfrac{5}{4}\left(\dfrac{2}{3} + 4\right)$ (iii) $\dfrac{\pi - 1}{\frac{4}{3}}$

(iv) $10\pi - 2e + 1$ (v) $\left(\dfrac{432 - 0.0012}{101}\right)$ (vi) $\left(\dfrac{2}{9}\right) \cdot \left(\dfrac{9}{7}\right)$

Consider π and e expressed with fifteen significant digits as the exact numbers.

Exercise 04: Calculate the roundoff error if chopping and rounding is used to write the following numbers accurate to four decimal digits:

(i) $355/113$ (ii) $\sqrt{3/142}$ (iii) $\sqrt[3]{\ln 2}$

Exercise 05: We want to round-off each the following numbers to three decimal places. For which number, the result of "round-off by chopping" and "round-off by rounding-rule" will be the same:

(A) 5.5555 (B) 3.3575 (C) 5.5565 (D) 4.4555

Exercise 06: Find the absolute and relative errors involved in rounding 4.9997 to 5.000.

Exercise 07: Suppose a real number x is represented approximately by 0.6032 with the relative error is at most 0.1%. What is x?

Exercise 08: Suppose that a number is accurate to n significant figures and a_1 is the first significant figure than show that the relative error is bounded above by $\frac{1}{a_1} \times 10^{1-n}$.

Exercise 09: Show that if a number is rounded off to n digits than the relative error is bounded by $\frac{1}{2} \times 10^{1-n}$.

■■■

Chapter **2**

Solution of a Nonlinear Equation in One Variable

Corridor I: BASICS

Let's plan it

To unleash the topics of this Corridor, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

■■■

Corridor II: ANALYSIS

Let's think deep

To unleash the topics of this Corridor, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

■■■

Corridor III: PROGRAMMING ARCADE

Let's do it

To see more examples for practicing, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

For codes, please visit: https://github.com/DrAmjadAli11/SimplifiedNumericalAnalysis

■■■

2.6 Algorithms and Implementations

Question 36: Write down the algorithm (pseudo code) of the Newton's method to solve $f(x) = 0$. The algorithm should perform a fixed number of iterations.

To find a root of a non-linear equation $f(x) = 0$ **the Newton-Raphson method** requires an initial solution x_0 and considers the x-intercept of the tangent line to the function $f(x)$ at $x = x_0$ as the new approximation. Then, the x-intercept of the tangent line to the function at the new approximation is considered as the next approximation. This way, the process is repeated with the successive approximations until sufficient convergence is achieved. The formula to generate the sequence of successive approximations based on the said approach is given by

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}, \qquad \text{for } k = 1, 2, 3, \cdots$$

Algorithm: To solve $f(x) = 0$ using the following iterative formula (given an initial approximation x_0):

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}, \qquad \text{for } k = 1, 2, 3, \cdots$$

INPUTS: $\begin{cases} \boldsymbol{x0}\text{: a real value as the initial approximation } x_0 \text{ sufficiently close to the root} \\ \boldsymbol{N}\text{: an integer as the maximum number of iterations} \end{cases}$

OUTPUT: $\begin{cases} \boldsymbol{xn}\text{: a real value as the approximate solution} \\ \text{(on completing } \boldsymbol{N} \text{ iterations)} \end{cases}$

```
Step 1    Receive the inputs as stated above
Step 2    Set $xn = x0$            (initialize $xn$ with the initial approximation)
Step 3    for $k = 1, 2, 3, \cdots, N$ perform Steps 4-6
              Step 4    Set $xp = xn$    {$xp$ is to keep a copy of the approximation $xn$,
                                          because $xn$ is going to be updated.
              Step 5    Set $fxp$ as the value of $f(xp)$
                        Set $dfxp$ as the value of $f'(xp)$
              Step 6
                        $xn = xp - \frac{fxp}{dfxp}$    {Computing a new
                                                        approximation to the root
          end for    (Go to Step 4 for the next iteration)
Step 7    Print the output: $xn$
          [Additionally, the initial approximation ($x0$), number of iterations ($k$), and $f(xn)$ can be printed]
STOP.
```

Remark: In the algorithm, it is assumed that neither any pitfall of the method will occur, nor $f(x)$ will be equal to zero (or the machine-epsilon) in any iteration for the given problem and initial approximation.

Question 37: Write a Python program to find a real root of $f(x) = 4x + \sin x - e^x = 0$ using the Newton-Raphson method. Take initial approximation as $x_0 = 0$. Here $f'(x) = 4 + \cos x - e^x$. The program should perform a fixed number of iterations.

script_2.1: newton1.ipynb

```
from numpy import *
N = 12         # setting the maximum number of iterations

x0 = float(input("Enter the initial approximation: "))
print("iter.              xk                  f(xk)")

xk = x0
fxk = 4*xk + sin(xk) – exp(xk)
for k in range(1,N+1):
        xp = xk
        fxp = fxk
        dfxp = 4 + cos(xp) – exp(xp)
        xk = xp – (fxp/dfxp)
        fxk = 4*xk + sin(xk) – exp(xk)

        #print(k, xk, fxk, sep="\t")
        print(f"{k}\t {xk:.16f}\t{fxk:.16f}")

print(N , "iterations completed.")
```

Output Console:

```
Enter the initial approximation: 0
iter.           xk                       f(xk)
1         0.2500000000000000      -0.0366214574332184
2         0.2599382850500705      -0.0000759982664056
3         0.2599589955313102      -0.0000000003332497
4         0.2599589956221257      0.0000000000000000
5         0.2599589956221257      0.0000000000000000
6         0.2599589956221257      0.0000000000000000
7         0.2599589956221257      0.0000000000000000
8         0.2599589956221257      0.0000000000000000
9         0.2599589956221257      0.0000000000000000
10        0.2599589956221257      0.0000000000000000
11        0.2599589956221257      0.0000000000000000
12        0.2599589956221257      0.0000000000000000
12 iterations completed.
```

Remark: In the program, it is assumed that neither any pitfall of the method will occur, nor $f(x)$ will be equal to zero (or the machine-epsilon) in any iteration for the given problem and initial approximation.

Remark: In the program, it is assumed that neither any pitfall of the method will occur, nor $f(x)$ will be equal to zero (or the machine-epsilon) in any iteration.

Remark: The algorithm in Question 36 (likewise 18) has a shortcoming that on completion of the given fixed number of N iterations the solutions might not have been converged (the desired accuracy might not have been achieved). Moreover, the algorithm has a shortcoming if the convergence has been achieved (or divergence has occurred) in few iterations, even then the iterations would not stop immediately; the algorithm will complete the fixed number of iterations. These shortcomings in the algorithm can be addressed by incorporating the two convergence criteria such that if the convergence is achieved (i. e., $\text{error} < \text{tolerence}$), then no more iterations will be performed, however, the number of iterations would not exceed the maximum limit on the number of iterations. Such an indispensable modification regarding the stopping criteria is adopted throughout the subsequent part of the book.

Remark: The Numpy library has functions sin(x), cos(x), and exp(x). In the script newton1.ipynb the Numpy library is imported completely using the wildcard *. Therefore, it is sufficient to write sin(x), cos(x), and exp(x) in the script to use these functions. If the Numpy library is imported as:

```
import numpy as np
```

Then, it is required to write np.sin(x), np.cos(x), and np.exp(x) for using these functions.

Interestingly, these functions are also available in the math module of the Standard Python Library.

Question 38: Write down the algorithm (pseudo code) of the Newton's method to solve $f(x) = 0$.

Algorithm: To solve $f(x) = 0$ using the following iterative formula (given an initial approximation x_0):

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}, \qquad \text{for } k = 1, 2, 3, \cdots$$

INPUTS: $\begin{cases} \boldsymbol{x0}\text{: a real value as the initial approximation } x_0 \text{ sufficiently close to the root} \\ \boldsymbol{TOL}\text{: a real value as the tolerance (permissible error)} \\ \boldsymbol{N}\text{: an integer as the maximum number of iterations} \end{cases}$

OUTPUT: $\begin{cases} \boldsymbol{xn}\text{: a real value as the approximate solution} \\ \text{(either on convergence or on completing } \boldsymbol{N} \text{ iterations — whichever happens first)} \end{cases}$

```
Step 1    Receive the inputs as stated above
Step 2    Set $xn = x0$        (initialize $xn$ with the initial approximation)
Step 3    for $k = 1, 2, 3, \cdots, N$ perform Steps 4-8
    Step 4    Set $xp = xn$    {$xp$ is to keep a copy of the approximation $xn$, because $xn$ is going to be updated.
    Step 5    Set $fxp$ as the value of $f(xp)$
              Set $dfxp$ as the value of $f'(xp)$
    Step 6
              $xn = xp - \frac{fxp}{dfxp}$    {Computing a new approximation to the root
```

```
            Step 7     Set $\boldsymbol{err} = |\boldsymbol{xn} - \boldsymbol{xp}|$          (or $err = |xn - xp|/|xp|$)

            Step 8
                       if ($\boldsymbol{err} < \boldsymbol{TOL}$) then        }  This means that the consecutive
                           Exit/Break the loop          }  approximations are nearly the same.
                                                           Therefore, stop iterations.

         end for   (Go to Step 4 for the next iteration)

Step 9     Print the output: $\boldsymbol{xn}$
           [Additionally, the initial approximation ($\boldsymbol{x0}$), number of iterations ($\boldsymbol{k}$), $f(\boldsymbol{xn})$, and error ($\boldsymbol{err}$) can be printed]
           if ($\boldsymbol{err} < \boldsymbol{TOL}$)  OUTPUT ('The desired accuracy achieved; Solution converged.')
           else              OUTPUT ('The number of iterations exceeded the maximum limit.')  because $\boldsymbol{k} > \boldsymbol{N}$
STOP.
```

Remark: In the algorithm, it is assumed that neither any pitfall of the method will occur, nor $f(x)$ will be equal to zero (or the machine-epsilon) in any iteration for the given problem and initial approximation.

Question 39: Write a Python program to find a real root of $f(x) = 4x + \sin x - e^x = 0$ using the Newton-Raphson method. Take initial approximation as $x_0 = 0$. Here $f'(x) = 4 + \cos x - e^x$. The iterations of the method should stop when either the approximation is accurate within 10^{-5}, or the number of iterations exceed 100, whichever happens first.

script_2.2: newton2.ipynb

```
from numpy import *

TOL = 0.000001        # setting the tolerance
N = 50                # setting the maximum number of iterations

x0 = float(input("Enter the initial approximation: "))
print("iter.        xk              f(xk)                       Error")

xk = x0   ;   fxk = 4*xk + sin(xk) – exp(xk)
for k in range(1,N+1):
        xp = xk
        fxp = fxk
        dfxp = 4 + cos(xp) – exp(xp)
        xk = xp - (fxp/dfxp)

        fxk = 4*xk + sin(xk) – exp(xk)

        err = abs(xk – xp)/abs(xk)

        #print(k, xk, fxk, err, sep="\t")
        print(f"{k}\t {xk:.16f}\t{fxk:.16f}\t{err:.12f}")
        if err < TOL:
            break

if err < TOL:
        print("Required accuracy achieved; Solution is convergent.")
else:
        print("The Number of iterations exceeded the maximum limit.")
```

Output Console:

```
Enter the initial approximation: 0
iter.        xk                       f(xk)                         Error
1        0.2500000000000000      -0.0366214574332184       1.000000000000
2        0.2599382850500705      -0.0000759982664056       0.038233248511
3        0.2599589955313102      -0.0000000003332497       0.000079668261
4        0.2599589956221257      0.0000000000000000        0.000000000349
Required accuracy achieved; Solution is convergent.
```

Remark: This program is based on the assumption that neither any pitfall of the method will occur, nor $f(x)$ will be equal to zero (or machine-epsilon) in any iteration for the given problem and data.

Question 40: Write a Python program to find a real root of the equation $f(x) = 4x + \sin x - e^x = 0$ using the Newton-Raphson method. Take initial approximation as $x_0 = 0$. Here $f'(x) = 4 + \cos x - e^x$. Use #define directive to evaluate $f(x)$ and $f'(x)$ wherever required. The iterations of the method should stop when either the approximation is accurate within 10^{-5}, or the number of iterations exceed 100, whichever happens first.

script_2.3: newton3.ipynb

```
from numpy import *
N = 500                 # setting the maximum number of iterations
TOL = 0.000001          # setting the tolerance

def fval(x):
        y = 4 * x + sin(x) – exp(x)
        return (y)

def dfval(x):
        dy = 4 + cos(x) – exp(x)
        return (dy)

x0 = float(input("Enter the initial approximation: "))
print("iter.         xk              f(xk)           Error")

xk = x0 ;  fxk = fval(xk)
for k in range(1,N+1):
        xp = xk
        fxp = fxk
        dfxp = dfval(xp)
        xk = xp – (fxp/dfxp)
        fxk = fval(xk)

        err = abs(xk – xp)/abs(xk)

        #print(k, xk, fxk, err, sep="\t")
        print(f"{k}\t {xk:.16f}\t{fxk:.16f}\t{err:.12f}")
        if err < TOL:
                break

if err < TOL:
        print("Required accuracy achieved; Solution is convergent.")
else:
        print("The Number of iterations exceeded the maximum limit.")
```

Output Console:

```
Enter the initial approximation: 0
iter.         xk                     f(xk)                   Error
1        0.2500000000000000      -0.0366214574332184      1.000000000000
2        0.2599382850500705      -0.0000759982664056      0.038233248511
3        0.2599589955313102      -0.0000000003332497      0.000079668261
4        0.2599589956221257      0.0000000000000000       0.000000000349
Required accuracy achieved; Solution is convergent.
```

Question 41: Write down the algorithm (pseudo code) of the Fixed-Point Iteration method to solve $f(x) = 0$.

The Fixed-Point Iteration method is an open method that approximates a root of the equation $f(x) = 0$ by rearranging the equation $f(x) = 0$ to get an appropriate form $x = g(x)$ and generating a sequence of successive approximations $\{x_k\}_{k=1}^{\infty}$ by the iterative formula $x_k = g(x_{k-1})$, for $k = 1, 2, 3, \cdots$. The said sequence may

- converge but could be different for different forms of $x = g(x)$,
- converge but could be different for different choices of the initial approximation x_0 for a particular form of $x = g(x)$, or
- diverge for some unsuitable form of $x = g(x)$ or an initial approximation x_0.

Algorithm: To solve $f(x) = 0 \Leftrightarrow x = g(x)$, using the following iterative formula (given an initial approximation x_0)

$$x_k \quad = \quad g(x_{k-1}), \qquad \text{for } k = 1, 2, 3, \cdots$$

INPUTS: $\begin{cases} \boldsymbol{x0}\text{: a real value as the initial approximation } x_0 \text{ sufficiently close to the root} \\ \boldsymbol{TOL}\text{: a real value as the tolerance (permissible error)} \\ \boldsymbol{N}\text{: an integer as the maximum number of iterations} \end{cases}$

OUTPUT: $\begin{cases}\boldsymbol{xn}\text{: a real value as the approximate solution} \\ \text{(either on convergence or on completing } \boldsymbol{N} \text{ iterations − whichever happens first)}\end{cases}$

```
Step 1      Receive the inputs as stated above
Step 2      Set $xn = x0$          (initialize $xn$ with the initial approximation)
Step 3      for $k = 1, 2, 3, \cdots, N$ perform Steps 4-7
    Step 4      Set $xp = xn$   {$xp$ is to keep a copy of the approximation $xn$, because $xn$ is going to be updated.}
    Step 5      Set $xn$ as the value of $g(xp)$   {Computing a new approximation to the root}
    Step 6      Set $err = |xn - xp|$            (or $err = |xn - xp|/|xp|$)
    Step 7      if ($err < TOL$) then
                    Exit/Break the loop   } This means that the consecutive approximations are nearly the same. Therefore, stop iterations.
            end for    (Go to Step 4 for the next iteration)
Step 8      Print the output: $xn$
            [Additionally, the initial approximation ($x0$), number of iterations ($k$), $f(xn)$, and error ($err$) can be printed]
            if ($err < TOL$)  OUTPUT ('The desired accuracy achieved; Solution converged.')
            else              OUTPUT ('The number of iterations exceeded the maximum limit.')
STOP.
```

Question 42: Write a Python program to find a real root of the equation $f(x) = 4x + \sin x - e^x = 0$ using the Fixed-Point Iteration method. Take $x = g(x) = \frac{1}{4}(e^x - \sin x)$ and $x_0 = 0$ as an initial approximation. The iterations of the method should stop when either the approximation is accurate within 10^{-5}, or the number of iterations exceeds 100, whichever happens first.

script_2.4: fixed_point.ipynb

```python
from numpy import *
N = 500                # setting the maximum number of iterations
TOL = 0.000001        # setting the tolerance

def fval(x):
        y = 4 * x + sin(x) – exp(x)
        return (y)

def gval(x):
        g = 0.25 * (exp(x) – sin(x))
        return (g)

x0 = float(input("Enter the initial approximation: "))
print("iter.          xk              f(xk)            Error")

xk = x0
for k in range(1,N+1):
        xp = xk
        xk = gval(xp)

        fxk = fval(xk)

        err = abs(xk – xp)/abs(xk)

        #print(k, xk, fxk, err, sep="\t")
        print(f"{k}\t {xk:.16f}\t{fxk:.16f}\t{err:.12f}")
        if err < TOL:
                break

if err < TOL:
        print("Required accuracy achieved; Solution is convergent.")
else:
        print("The Number of iterations exceeded the maximum limit.")
```

Output Console:

```
Enter the initial approximation: 0
iter.         xk                     f(xk)                  Error
1       0.2500000000000000      -0.0366214574332184      1.000000000000
2       0.2591553643583046      -0.0029494454793164      0.035327705375
3       0.2598927257281337      -0.0002431823527489      0.002837175868
4       0.2599535213163210      -0.0000200881176871      0.000233870993
5       0.2599585433457428      -0.0000016596390247      0.000019318578
6       0.2599589582554989      -0.0000001371177203      0.000001596059
7       0.2599589925349290      -0.0000000113285412      0.000000131865
Required accuracy achieved; Solution is convergent.
```

Question 43: Write down the algorithm (pseudo code) of the Secant method to solve $f(x) = 0$.

The iterative formula of **the Secant method** for solving $f(x) = 0$ (with $x = x_0$ and $x = x_1$ as the initial approximations) is given by

$$x_k \quad = \quad x_{k-1} - \frac{f(x_{k-1})(x_{k-1} - x_{k-2})}{f(x_{k-1}) - f(x_{k-2})}, \qquad \text{for } k = 2, 3, 4, \cdots$$

Algorithm: To solve $f(x) = 0$ using the iterative formula (given the root containing interval):

$$x_k = x_{k-1} - \frac{f(x_{k-1})(x_{k-1} - x_{k-2})}{f(x_{k-1}) - f(x_{k-2})}, \qquad \text{for } k = 2, 3, 4, \cdots$$

```
INPUTS:   {  a and b: two real values as the initial approximations sufficiently close to the root
          {  TOL: a real value as the tolerance (permissible error)
          {  N: an integer as the maximum number of iterations

OUTPUT:   {  xn: a real value as the approximate solution
          {  (either on convergence or on completing N iterations — whichever happens first)

Step 1    Receive the inputs as stated above

Step 2    Set $xn = b$                    (initialize xn with any of the two endpoints)

Step 3    Set $x0 = a$
          Set $x1 = b$
          Set $fx0$ as the value of $f(x0)$
          Set $fx1$ as the value of $f(x1)$

Step 4    for $k = 2, 3, \cdots, N + 1$ perform Steps 5-10

    Step 5    Set $xp = xn$    { xp is to keep a copy of the approximation xn,
                               { because xn is going to be updated.
    Step 6
              $xn = x1 - \frac{fx1(x1 - x0)}{fx1 - fx0}$    { Computing a new
                                                           { approximation to the root

    Step 7    Set $fxn$ as the value of $f(xn)$

    Step 8    Set $err = |xn - xp|/|xp|$                 (or $err = |xn - xp|$)

    Step 9
              if ($err < TOL$) then          }  This means that the consecutive
                      Exit/Break the loop    }  approximations are nearly the same.
                                                Therefore, stop iterations.

              else
                      Set $x0 = x1$       }
                      Set $fx0 = fx1$     }  preparing two approximations
                      Set $x1 = xn$       }  for the next iteration
                      Set $fx1 = fxn$     }

    end for    (Go to Step 5 for the next iteration)

Step 10   Print the output: $xn$
          [Additionally, the initial approx. ($x0$ and $x1$), number of iterations ($k - 1$), $f(xn)$, and error ($err$) can be printed]

          if ($err < TOL$)  OUTPUT ('The desired accuracy achieved; Solution converged.')
          else              OUTPUT ('The number of iterations exceeded the maximum limit.')
STOP.
```

Question 44: Write a Python program to find a real root of the equation $f(x) = 4x + \sin x - e^x = 0$ using the Secant method. Take initial approximation as $x_0 = 0$ and $x_1 = 1$. The iterations of the method should stop when either the approximation is accurate within 10^{-5}, or the number of iterations exceeds 100, whichever happens first.

script_2.5: secant.ipynb

```
from numpy import *
N = 500                   # setting the maximum number of iterations
TOL = 0.000001        # setting the tolerance

def fval(x):
        y = 4 * x + sin(x) – exp(x)
        return (y)

a = float(input("Enter the first initial approximation: "))
b = float(input("Enter the second initial approximation: "))
xk = b
x0 = a
x1 = b
fx0 = fval(x0)
fx1 = fval(x1)
print("k     xk–2       xk–1       xk          f(xk)          Error")

for k in range(2,N+2):
        xp = xk
        xk = x1 – (fx1 * (x1 – x0)) / (fx1 – fx0)

        fxk = fval(xk)

        err = abs(xk – xp)/abs(xk)

        #print(k, xk, fxk, err, sep="            ")
        print(f"{k}\t {xk:.16f}\t{fxk:.16f}\t{err:.12f}")

        if err < TOL:
                break
        else:
                x0 = x1
                fx0 = fx1
                x1 = xk
                fx1 = fxk

if err < TOL:
        print("Required accuracy achieved; Solution is convergent.")
else:
        print("The Number of iterations exceeded the maximum limit.")
```

Output Console:

```
Enter the first initial approximation: 0
Enter the second initial approximation: 1
k        xk                      f(xk)                    Error
2        0.3201855379035207      0.2181015285252024       2.123189156349
3        0.2423578458166424      -0.0648264842999824      0.321127182100
4        0.2601902817383949      0.0008486682977629       0.068536133643
5        0.2599598472066112      0.0000031249088128       0.000886423554
6        0.2599589955804161      -0.0000000001530542      0.000003276002
7        0.2599589956221257      0.0000000000000000       0.000000000160
Required accuracy achieved; Solution is convergent.
```

Question 45: Write down the algorithm (pseudo code) of the Bisection method to solve $f(x) = 0$.

The Bisection method selects $c \in (a + b)$, as the midpoint of the interval $[a, b]$, using the formula

$$c = a + \frac{(b - a)}{2}$$

Algorithm: To solve $f(x) = 0$ using the iterative formula (given the root containing interval):

$$x_k = x_{k-2} + \frac{x_{k-1} - x_{k-2}}{2}, \qquad \text{for } k = 2, 3, 4, \cdots$$

INPUTS: $\begin{cases} \boldsymbol{a} \text{ and } \boldsymbol{b}\text{: two real values as the initial approximations bracketing the root} \\ \boldsymbol{TOL}\text{: a real value as the tolerance (permissible error)} \\ \boldsymbol{N}\text{: an integer as the maximum number of iterations} \end{cases}$

OUTPUT: $\begin{cases} \boldsymbol{xn}\text{: a real value as the approximate solution} \\ \text{(either on convergence or on completing } \boldsymbol{N} \text{ iterations — whichever happens first)} \end{cases}$

```
Step 1    Receive the inputs as stated above
Step 2    Set $xn = b$                  (initialize $xn$ with any of the two endpoints)
Step 3    Set $x0 = a$
          Set $x1 = b$
          Set $fx0$ as the value of $f(x0)$
          Set $fx1$ as the value of $f(x1)$
Step 4    for $k = 2, 3, \cdots, N + 1$ perform Steps 5-10
    Step 5    Set $xp = xn$   {$xp$ is to keep a copy of the approximation $xn$, because $xn$ is going to be updated.}
    Step 6
              $xn = x0 + \frac{x1 - x0}{2}$   {Computing a new approximation to the root}
    Step 7    Set $fxn$ as the value of $f(xn)$
```

```
        Step 8      Set $\boldsymbol{err1} = |\boldsymbol{xn} - \boldsymbol{xp}|/|\boldsymbol{xn}|$          (or $err = |xn - xp|$)
                    Set $\boldsymbol{err2} = |\boldsymbol{fxn}|$
                    Set $\boldsymbol{err} = \boldsymbol{min}(\boldsymbol{err1}, \boldsymbol{err2})$

        Step 9
                    if ($\boldsymbol{err} < \boldsymbol{TOL}$ )then        }  This means that either $\boldsymbol{f}(\boldsymbol{xn})$ is the close to
                            Exit/Break the loop   }  zero, or the consecutive approximations are
                                                     nearly the same. Therefore, stop iterations.

                    else if ($\boldsymbol{f}(\boldsymbol{x0})\boldsymbol{f}(\boldsymbol{xn}) < 0$) then  }
                            Set $\boldsymbol{x1} = \boldsymbol{xn}$           }  Adjusting one endpoint
                            Set $\boldsymbol{fx1} = \boldsymbol{fxn}$         }  of the interval such that
                    else                              }  half of the interval will be
                            Set $\boldsymbol{x0} = \boldsymbol{xn}$           }  used in the next iteration
                            Set $\boldsymbol{fx0} = \boldsymbol{fxn}$         }

    end for         (Go to Step 5 for the next iteration)

Step 10    Print the output: $\boldsymbol{xn}$
           [Additionally, the starting interval $[a, b]$, number of iterations $(\boldsymbol{k} - \boldsymbol{1})$, $f(\boldsymbol{xn})$, and error $(\boldsymbol{err})$ can be printed]
           if ($\boldsymbol{err} < \boldsymbol{TOL}$)  OUTPUT ('The desired accuracy achieved; Solution converged.')
           else                OUTPUT ('The number of iterations exceeded the maximum limit.')
STOP.
```

■

Remark: While using a bracketing method, there might arise a situation in which the two consecutive approximations to the roots are not sufficiently close to each other (i.e., the sequence of successive approximations has not converged), but the function values at the approximations are sufficiently close to zero (i.e., $|f(x_k)| <$ tolerence). Therefore, there is no point to proceed the iterations further. The iterations should be stopped. Therefore, the algorithm of a bracketing method (the Bisection, or Regula-Falsi method) should include both of the convergence criteria of testing the convergence of the roots, and closeness of the function values to zero. The iterations should be terminated on whichever criterion is met first, ensuring the convergence. To accommodate this in the algorithm, the two kinds of errors are computed and the minimum of the two errors is found to compare with the tolerance:

Set $err1 = |xn - xp|/|xn|$ (or $err = |xn - xp|$)

Set $err2 = |fxn|$

Set $err = min(err1, err2)$

■

Question 46: Write a Python program to find a real root of the equation $f(x) = 4x + \sin x - e^x = 0$ in $[0, 1]$ using the Bisection method. The two function values at the endpoints of the interval have opposite signs. The iterations of the method should stop when either the approximation is accurate within 10^{-5}, or the number of iterations exceeds 100, whichever happens first.

script_2.6: bisection.ipynb

```
from numpy import *
N = 500                # setting the maximum number of iterations
TOL = 0.000001         # setting the tolerance

def fval(x):
        y = 4*x + sin(x) – exp(x)
        return (y)

a = float(input("Enter the first initial approximation: "))
b = float(input("Enter the second initial approximation: "))
xk = b
x0 = a
x1 = b
fx0 = fval(x0)
fx1 = fval(x1)
print("k     a       b       ck          f(c)          Error")

for k in range(2,N+2):
        xp = xk
        xk = x0 + (x1 – x0)/2

        fxk = fval(xk)

        err1 = abs(xk – xp)/abs(xk)
        err2 = abs(fxk)
        err = min(err1,err2)

        #print(k, x0, xp, xk, fxk, err, sep="\t")
        print(f"{k}\t{x0:.7f}\t{xp:.7f}\t {xk:.10f}\t{fxk:.10f}\t{err:.8f}")
        if err < TOL:
                break
        elif fx0 * fxk < 0:
                x1 = xk
                fx1 = fxk
        else:
                x0 = xk
                fx0 = fxk

if err < TOL:
        print("Required accuracy achieved; Solution is convergent.")
else:
        print("The Number of iterations exceeded the maximum limit.")
```

Output Console:

```
Enter the first initial approximation: 0
Enter the second initial approximation: 1
k       a              b           ck                     f(c)          Error
2        0.0000000   1.0000000   0.5000000000      0.8307042679    0.83070427
3        0.0000000   0.5000000   0.2500000000     -0.0366214574    0.03662146
4        0.2500000   0.2500000   0.3750000000      0.4112811145    0.33333333
5        0.2500000   0.3750000   0.3125000000      0.1906005734    0.19060057
6        0.2500000   0.3125000   0.2812500000      0.0777719929    0.07777199
7        0.2500000   0.2812500   0.2656250000      0.0207665250    0.02076653
8        0.2500000  0.26562500    0.2578125000     -0.0078801924    0.00788019
9        0.2578125   0.25781250  0.2617187500      0.0064550521    0.00645505
10       0.2578125   0.2617188   0.2597656250     -0.0007096071    0.00070961
11       0.2597656   0.2597656   0.2607421875      0.0028734643    0.00287346
12       0.2597656   0.2607422   0.2602539062      0.0010821139    0.00108211
13       0.2597656   0.2602539   0.2600097656      0.0001862997    0.00018630
14       0.2597656   0.2600098   0.2598876953     -0.0002616421    0.00026164
15       0.2598877   0.2598877   0.2599487305     -0.0000376683    0.00003767
16       0.2599487   0.2599487   0.2599792480      0.0000743164    0.00007432
17       0.2599487   0.2599792   0.2599639893      0.0000183242    0.00001832
18       0.2599487   0.2599640   0.2599563599     -0.0000096720    0.00000967
19       0.2599564   0.2599564   0.2599601746      0.0000043261    0.00000433
20       0.2599564   0.2599602   0.2599582672     -0.0000026729    0.00000267
21       0.2599583   0.2599583   0.2599592209      0.0000008266    0.00000083
Required accuracy achieved; Solution is convergent.
```

■

Question 47: Write down the algorithm (pseudo code) of the Regula-Falsi method to solve $f(x) = 0$.

The Regula-Falsi method selects $c \in (a + b)$, as the point where the line segment joining $f(a)$ and $f(b)$ intersects the x-axis, using the formula

$$c = b - \frac{f(b)(b-a)}{f(b) - f(a)}$$

Algorithm: To solve $f(x) = 0$ using the iterative formula (given the root containing interval):

$$x_k = x_{k-1} - \frac{f(x_{k-1})(x_{k-1} - x_{k-2})}{f(x_{k-1}) - f(x_{k-2})}, \qquad \text{for } k = 2, 3, 4, \cdots$$

```
INPUTS:  { a and b: two real values as the initial approximations bracketing the root
         { TOL: a real value as the absolute error tolerance
         { N: an integer as the maximum number of iterations

OUTPUT:  { xn: a real value as the approximate solution
         { (either on convergence or on completing N iterations — whichever happens first)

Step 1      Receive the inputs as stated above

Step 2      Set $xn = b$                    (initialize $xn$ with any of the two endpoints)

Step 3      Set $x0 = a$
            Set $x1 = b$
            Set $fx0$ as the value of $f(x0)$
            Set $fx1$ as the value of $f(x1)$

Step 4      for $k = 2, 3, \cdots, N+1$ perform Steps 5-10

    Step 5      Set $xp = xn$     { $xp$ is to keep a copy of the approximation $xn$,
                                  { because $xn$ is going to be updated.

    Step 6
                $xn = x1 - \frac{fx1(x1 - x0)}{fx1 - fx0}$     { Computing a new
                                                               { approximation to the root

    Step 7      Set $fxn$ as the value of $f(xn)$

    Step 8      Set $err1 = |xn - xp|/|xn|$                   (or $err = |xn - xp|$)
                Set $err2 = |fxn|$
                Set $err = min(err1, err2)$

    Step 9
               if ($err < TOL$)then            } This means that either $f(xn)$ is the close to
                           Exit/Break the loop } zero, or the consecutive approximations are
                                                 nearly the same. Therefore, stop iterations.
                else if ($f(x0)f(xn) < 0$) then }
                           Set $x1 = xn$        }  Adjusting one endpoint
                           Set $fx1 = fxn$      }  of the interval such that
                else                            }  a shorter interval will be
                           Set $x0 = xn$        }  used in the next iteration
                           Set $fx0 = fxn$      }

            end for        (Go to Step 5 for the next iteration)

Step 10     Print the output: $xn$
            [Additionally, the starting interval $[a, b]$, number of iterations $(k-1)$, $f(xn)$, and error $(err)$ can be printed]
            if ($err < TOL$)  OUTPUT ('The desired accuracy achieved; Solution converged.')
            else              OUTPUT ('The number of iterations exceeded the maximum limit.')

STOP.
```

Question 48: Write a Python program to find a real root of the equation $f(x) = 4x + \sin x - e^x = 0$ in $[0, 1]$ using the Regula-Falsi method. The two function values at the endpoints of the interval have opposite signs. Use #define directive to evaluate $f(x)$ wherever required. The iterations of the method should stop when either the approximation is accurate within 10^{-5}, or the number of iterations exceeds 100, whichever happens first.

script_2.7: regula_falsi.ipynb

```
import numpy as np
N = 500                 # setting the maximum number of iterations
TOL = 0.000001          # setting the tolerance

def fval(x):
        y = 4*x + sin(x) – exp(x)
        return (y)

a = float(input("Enter the left endpoint of the interval: "))
b = float(input("Enter the right endpoint of the interval: "))
xk = b
x0 = a
x1 = b
fx0 = fval(x0)
fx1 = fval(x1)
print("k      a       b       ck        f(c)        Error")

for k in range(2,N+2):
        xp = xk
        xk = x1 - (fx1 * (x1 – x0))/(fx1 – fx0)

        fxk = fval(xk)

        err1 = abs(xk – xp) / abs(xk)
        err2 = abs(fxk)
        err = min(err1,err2)

        #print(k, x0, xp, xk, fxk, err, sep="\t")
        print(f"{k}\t{x0:.7f}\t{xp:.7f}\t {xk:.10f}\t{fxk:.10f}\t{err:.8f}")
        if ( fxk < TOL ):
                break
        elif err < TOL:
                break
        elif fx0 * fxk < 0:
                x1 = xk
                fx1 = fxk
        else:
                x0 = xk
                fx0 = fxk

```

```
if err < TOL:
        print("Required accuracy achieved; Solution is convergent")
else:
        print("The Number of iterations exceeded the maximum limit.")
```

Output Console:

```
Enter the left endpoint of the interval: 0
Enter the right endpoint of the interval: 1
k            a           b        ck               f(c)               Error
2         0.0000000  1.0000000   0.3201855379    0.2181015285       0.21810153
3         0.0000000  0.3201855   0.2628561991    0.0106248258       0.01062483
4         0.0000000  0.2628562   0.2600927589    0.0004908334       0.00049083
5         0.0000000  0.2600928   0.2599651593    0.0000226176       0.00002262
6         0.0000000  0.2599652   0.2599592796    0.0000010421       0.00000104
7         0.0000000  0.2599593   0.2599590087    0.0000000480       0.00000005
Required accuracy achieved; Solution is convergent
```

■

Remark: An interesting online calculator by CASIO® at **https://keisan.casio.com** has the following webpage to approximate the root of a non-linear equation using different methods.

https://keisan.casio.com/menu/system/000000001000

■■■

Chapter Summary

- The root-finding problem refers to find some appropriate value $x = \alpha$ in the domain of a function f such that $f(\alpha) = 0$. Every such possible value α is called a ***root* of the equation** $f(x) = 0$.

- Geometrically, a root of an equation $f(x) = 0$ is the point where the graph of f intersects the x-axis.

- An iterative numerical method to approximate the root starts with some appropriate or reasonable estimation (also called **initial approximation** or guess) of the exact root and attempts to refine the approximation, iteratively. The iterations are repeated until a desired level of accuracy is achieved.

- Let x_0 denotes the initial approximation and $x_1, x_2, x_3, \cdots$ denote the successive iterative solutions to an exact root α of the equation $f(x) = 0$. The sequence $\{x_k\}_{k=0}^{\infty}$ of the successive approximations is said to ***converge*** to the exact root α, if the successive approximations approach α. In such a case, the iterative method is also said to converge. In other words, the iterative method is said to be ***convergent*** for a given initial approximation if the corresponding sequence of successive approximations is convergent to the exact solution. Under certain conditions, it is possible for an iterative method that the sequence of successive approximations might ***diverge*** from a desired exact root α.

- **Stopping Criteria:** The most common convergence criterion to stop the iterative process is based on the comparison of the estimated error with the error tolerance. For this purpose, the current approximation is considered as the true solution and the previous approximation is considered as the approximate solution for estimating the error and any appropriate one of the following criteria is used,

$$(1) \quad |x_k - x_{k-1}| \leq \tau$$

$$(2) \quad \left|\frac{x_k - x_{k-1}}{x_k}\right| \leq \tau$$

$$(3) \quad \left|\frac{x_k - x_{k-1}}{x_k}\right| \times 100 \leq \tau$$

  Here x_k and x_{k-1} denote the current and previous approximations, respectively, and τ denotes the tolerance.

- **Another Stopping Criterion:** Note that the values of the function f tend to zero with the progress of the iterative process. Thus, falling of the difference between the function values and zero beyond a certain level might also indicate convergence.

- The numerical methods of finding a root of $f(x) = 0$ can be categorized as bracketing methods and open methods.

- **Bracketing methods** start with an interval containing a root and squeeze down the interval, iteratively. Two well-known root bracketing methods are the Bisection method and the Regula-Falsi (False-Position) method.

- **Open Methods** are those who obtain successive single approximations irrespective of their location at any side of the root. Some of the well-known open methods are the Fixed-Point Iteration method, the Newton-Raphson method (Newton's method), and the Secant method.

- A bracketing method for finding a root/zero of a continuous function f starts with an interval $[a, b]$ containing a root. The opposite signs of $f(a)$ and $f(b)$ ensure (due to the Intermediate value theorem) that there exists a root α of $f(x) = 0$ in (a, b). To get closer to the root α, first a point $c \in (a + b)$ is chosen. If $f(c) = 0$, then c is the exact root. Otherwise, either of the intervals $[a, c]$ or $[c, b]$ is chosen as the squeezed interval containing the root. The root lies in $[a, c]$ if $f(a)f(c) < 0$, or in $[c, b]$ if $f(c)f(b) < 0$. The selected interval is relabeled as $[a, b]$ and the process is repeated. This way, a sequence of points $c_1, c_2, c_3, \cdots$, is formed. The iterations are performed until the approximations of the root of $f(x)$ in two consecutive iterations are sufficiently close to each other.

- **The Bisection method** selects $c \in (a + b)$, as the midpoint of the interval $[a, b]$, using the formula

$$c = \frac{(a + b)}{2}$$

- **The Regula-Falsi method** selects $c \in (a + b)$, as the point where the line segment joining $f(a)$ and $f(b)$ intersects the x-axis, using the formula

$$c = b - \frac{f(b)(b - a)}{f(b) - f(a)}$$

- For the Bisection method, the error-bound is given by,

$$|\alpha - c_k| \leq \frac{b - a}{2^k}, \qquad \text{for } k = 1, 2, 3, \cdots,$$

  Here α is the exact root of the equation $f(x) = 0$ in (a, b) and $c_k = \frac{a_{k-1} + b_{k-1}}{2}$ is the midpoint of the interval in kth iteration.

- The formula to determine the maximum number of iterations N of the Bisection method after which the error associated with any point in the squeezed interval is not greater than a given permissible absolute error τ_a is as below:

$$N \geq \frac{\log(b - a) - \log(\tau_a)}{\log(2)}$$

  This formula tells that, for an interval of unit length, it is sure that after 10, 14, 17, and 20 iterations the length of the squeezed interval (or the absolute error) is not greater than 10^{-3}, 10^{-4}, 10^{-5}, and 10^{-6}, respectively.

- **The Fixed-Point Iteration method** is an open method that approximates a root of the equation $f(x) = 0$ by rearranging the equation $f(x) = 0$ to get an appropriate form $x = g(x)$ and generating a sequence of successive approximations $\{x_k\}_{k=1}^{\infty}$ by the iterative formula $x_k = g(x_{k-1})$, for $k = 1, 2, 3, \cdots$. The said sequence may
  - converge but could be different for different forms of $x = g(x)$,

  - converge but could be different for different choices of the initial approximation x_0 for a particular form of $x = g(x)$, or
  - diverge for some unsuitable form of $x = g(x)$ or an initial approximation x_0.

- Suppose that f is a continuous function and the equation $f(x) = 0$ has a real root α. Suppose that the equation $f(x) = 0$ can be rearranged in the form $x = g(x)$ such that α is a fixed-point of the function g, and g and g' are continuous in some neighbourhood I around α. If

$$|g'(x)| \quad \leq \quad K \quad < \quad 1, \qquad \text{for all } x \in I,$$

then for any initial approximation $x_0 \in I$, the sequence $\{x_k\}_{k=1}^{\infty}$ of successive approximations, generated by the iterative formula $x_k = g(x_{k-1})$, for $k = 1, 2, 3, \cdots$, converges to the solution α.

- To find a root of a non-linear equation $f(x) = 0$ **the Newton-Raphson method** requires an initial solution x_0 and considers the x-intercept of the tangent line to the function $f(x)$ at $x = x_0$ as the new approximation. Then, the x-intercept of the tangent line to the function at the new approximation is considered as the next approximation. This way, the process is repeated with the successive approximations until sufficient convergence is achieved. The formula to generate the sequence of successive approximations based on the said approach is given by

$$x_k \quad = \quad x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}, \qquad \text{for } k = 1, 2, 3, \cdots$$

- A sufficient condition of convergence for the Newton-Raphson method: Suppose that α is a root of the equation $f(x) = 0$. Suppose that I is a neighbourhood of α such that $f(x)$, $f'(x)$ and $f''(x)$ are continuous on I. If $|f(x)f''(x)| \leq |f'(x)|^2$, for all $x \in I$, then for an initial approximation $x_0 \in I$, the sequence $\{x_k\}_{k=1}^{\infty}$ of successive approximations, generated by the Newton's formula, converges to the solution α.

- The iterative formula of **the Secant method** for solving $f(x) = 0$ (with $x = x_0$ and $x = x_1$ as the initial approximations) is given by

$$x_k \quad = \quad x_{k-1} - \frac{f(x_{k-1})(x_{k-1} - x_{k-2})}{f(x_{k-1}) - f(x_{k-2})}, \qquad \text{for } k = 2, 3, 4, \cdots$$

- Comparison of the False-Position method and the Secant method:
  - The False-Position method is a bracketing method, whereas the Secant method is an open method.
  - The False-Position method keeps the root bracketed by selects out the root bracketing subintervals out the two subintervals obtains in each of the iterations. On the other hand, the Secant method selects the two most recent approximations out of the three available approximations in any iteration to proceed to the next iteration.
  - The False-Position method always converges, whereas the Secant method may not converge for certain situations.
  - If the Secant method is convergent, it converges faster than the False-Position method. That is, it has a higher convergence rate than that of the False-Position method.

- The order/rate of convergence of the Bisection method is 1 (i.e., linear) and the asymptotic error constant is (1/2)

- The order/rate of convergence of the False-Position or Regula-Falsi method is 1 (i.e., linear) and the asymptotic error constant is $-\frac{1}{2}\frac{f''(\alpha)}{f'(\alpha)}\varepsilon_0$

- The order/rate of convergence of the Fixed-Point Iteration method is 1 (i.e., linear) and the asymptotic error constant is the maximum value of the function $g'(x)$ in some neighbourhood around the solution α.

- The order/rate of convergence of the Newton-Raphson method is 2 (i.e., quadratic) and the asymptotic error constant is $-\frac{1}{2}\frac{f''(\alpha)}{f'(\alpha)}$

- The order/rate of convergence of the Secant method is 1.62 (i.e., superlinear).

- The Newton-Raphson method may fail to converge to a root in different situations including where $f'(x)$ or $f''(x)$ becomes zero at any approximation.

- The Newton-Raphson method converges to a multiple root very slowly (instead of exhibiting quadratic convergence).

- **The Aitken's Δ^2 method** offers a technique for accelerating the convergence of any sequence that is linearly convergent. From the given sequence $\{x_k\}_{k=1}^{\infty}$ that linearly converges to α, another sequence $\{\bar{\bar{x}}_k\}_{k=1}^{\infty}$ that also converges to α with possibly improved convergence rate is constructed by using the Aitken's acceleration formula given as

$$\bar{\bar{x}}_k \quad \cong \quad x_k - \frac{(\Delta x_k)^2}{\Delta^2 x_k}$$

■■■

Chapter Exercises

Exercise 01: Find a real root of the following equations using the Bisection method accurate to four decimal places.

(i) $\log(x) - \cos x = 0$
(ii) $e^{-x} - 10x = 0$
(iii) $x^3 + x^2 - 1 = 0$

Exercise 02: Find a real root of the following equations using the Bisection method accurate to three decimal places.

(i) $x^6 - x^4 - x^3 - 1 = 0$
(ii) $x^3 - \sin x + 1 = 0$
(iii) $x \log_{10} x = 4.77$

Exercise 03: Approximate the solution of the following equations using the Regula-Falsi method accurate to three decimal places.

(i) $3x + \sin x - e^x = 0$
(ii) $4x^3 - 1 - e^{(x^2/2)}$
(iii) $x^2 = (e^{-2x} - 2)/x$

Exercise 04: Find the approximation to a real root of the equation $2\sin x - \frac{e^x}{4} - 1 = 0$ starting with $[-5, -3]$ using the Regula-Falsi method.

Exercise: Find a real root of each of the following equations using (a) the Bisection method, (b) the Regula-Falsi method, (c) the Newton's method, (d) the Secant method. Choose the initial approximation/s in the given interval. Assume that the tolerance for the approximate root is 0.001. The numeric values should not be rounded to less than 5 decimal places. (x is in radians, wherever applicable).

(i) $\cos x - xe^x = 0$, in $[0, 1]$
(ii) $\cos x - x + 2 = 0$, in $[1, 2]$
(iii) $e^x - x - 3 = 0$, in $[1, 2]$
(iv) $\ln(x) + x - 4 = 0$, in $[2, 3]$
(v) $4x + \sin x - e^x = 0$, in $[0, 1]$.

Exercise 06: Find a real root of the *Chebyshev* polynomial of degree four, $T_4(x) = 8x^4 - 8x^2 + 1$ using the Newton's method accurate to four decimal places.

Exercise 07: Find a root of the *Laguerre* polynomial of degree four, $L_4(x) = x^4 - 16x^3 + 72x^2 - 96x + 24$ using the Newton's method accurate to four decimal places.

Exercise 08: Find a root of the following equations using the Newton's method accurate to 4 decimal places.

(i) $2x + 3\cos x - e^x = 0,$
(ii) $x^2 - 4x + 4 - \ln x = 0$
(iii) $\tan x - 6 = 0$

Exercise 09: Find the roots accurate to within 10^{-3} of the Legendre polynomial $P_4(x) = x^4 - \frac{6}{7}x^2 + \frac{3}{35}$ on each interval, using the Secant method.

(i) $[-1, -0.5]$
(ii) $[-0.5, 0]$
(iii) $[0, 0.5]$
(iv) $[0.5, 1]$

Exercise 10: Approximate the value of $\sqrt[3]{4}$ using the Secant method accurate to 10^{-4}.

Exercise 11: Find a real root of the following equations using the Secant method accurate to 10^{-3} .

(i) $x^3 - 2x + 2 = 0$
(ii) $10 - 2x + \sin x = 0$
(iii) $2e^{-3}x + 1 - 3e^{-3x} = 0$

Exercise 12: Use the Fixed-Point method to find a root of the following, accurate to 3 decimal places.

(i) $e^x - 2x^2$ for $0 \leq x \leq 2$
(ii) $xe^x = 0$ for $1 \leq x \leq 2$
(iii) $x^2 - \sin x - x = 0$

Exercise 13: Find the solutions of the following equations using the fixed-point method accurate to 10^{-3} .

(i) $x = \tan x$
(ii) $x = \cos x$
(iii) $x = \sin(x + 2)$

Exercise 14: Find the solution of the equation (relevant to the vibrating beam),

$$\cos x \cosh x = 1$$

near $x = -\frac{3}{2}\pi$ using the Newton-Raphson method.

Exercise 15: The velocity V, in meters per second (m/s), of a free falling sky diver is expressed as:

$$V = \frac{gm}{D_c}\left(1 - exp\left(\frac{-D_c t}{m}\right)\right)$$

Here m is the mass of the falling body in kilograms (kg), D_c is the drag coefficient in kilogram per second (kg/s), t is the time in seconds (s), and $g = 9.8m/s^2$ is the gravitation acceleration. If the velocity of a body of mass $85kg$ is $40m/s$ after 5 seconds of free fall, then calculate the drag coefficient.

Hint for the Solution:

Given $m = 80kg, V = 40m/s, g = 9.8m/s^2$, and $t = 5s$, the equation takes the form:

$$40 = \frac{(9.8)(85)}{D_c}\left(1 - exp\left(\frac{-5D_c}{85}\right)\right)$$

or

$$f(D_c) = D_c + 17\ln(1 - 0.04802D_c) = 0$$

Solve this equation for D_c, using any appropriate iterative method. To obtain an initial guess of D_c, a trick is to calculate V for different assumed values of D_c. The values of the D_c, which produce values of V close to 40, can offer reasonable initial guess of D_c. While using an iterative method, approximate error should be calculated at each iteration. $[exp(x) = e^x]$

Exercise 16: The velocity V, in meters per second (m/s), of a free falling sky diver is expressed as:

$$V = \frac{gm}{D_c}\left(1 - exp\left(\frac{-D_c t}{m}\right)\right)$$

Here m is the mass of the falling body in kilograms (kg), D_c is the drag coefficient in kilogram per second (kg/s), t is the time in seconds (s), and $g = 9.8m/s^2$ is the gravitation acceleration. If the velocity of a falling body with drag coefficient of 18 kg/s is $50m/s$ after 7 seconds of free fall, then calculate the mass m of the body, accurate to 0.0001. $[exp(x) = e^x]$

Hint for the Solution:

Given $D_c = 18kg/s, V = 50m/s, g = 9.8m/s^2$, and $t = 7s$, the equation takes the form:

$$50 = \frac{(9.8)m}{18}\left(1 - exp\left(\frac{-126}{m}\right)\right)$$

or

$$f(m) = m\ln\left(1 - \frac{91.83673}{m}\right) + 126 = 0$$

Solve this equation for m, using any appropriate iterative method. To obtain an initial guess of m, a trick is to calculate V for different assumed values of m. The values of the m, which produce values of V close to 50, can offer reasonable initial guess of m. While using an iterative method, approximate the error at each iteration.

Exercise 17: The volume V of spherical water-tank in cubic meters can be calculated as:

$$V = \frac{\pi H^2(3R - H)}{3}$$

where H denotes the height of water level in meters from the base of the tank, and R denotes the radius of the spherical tank in meters. If the radius R of a tank is 2.5 meters, then how much water level must be raised in the tank to hold 27 cubic meters of water.

Hint for the Solution:

Given $R = 2.5$ and $V = 27$, and taking $\pi = 3.14159$ the equation takes the form

$$27 = \frac{\pi H^2(7.5 - H)}{3}$$

or

$$f(H) = 3.14159H^3 - 23.56193H^2 + 81 = 0$$

Solve this equation for H, using any appropriate iterative method. Intuitively, appropriate initial guesses for H can be taken from $[0,2R]$. While using an iterative method, approximate error should be calculated at each iteration.

Exercise 18: Numerically, compare the convergence of the method:

$$x_k = x_{k-1} - 2\frac{f(x_{k-1})}{f'(x_{k-1})}, \qquad \text{for } k = 1, 2, 3, \cdots$$

with the Newton-Raphson method on a function with a known double root.

Exercise 19: The ideal gas equation relates the volume (V in L), temperature (T in K), pressure (P in atm), and the amount of gas (number of moles n) by:

$$P = \frac{nRT}{V}$$

where R = 0.08206 $(L\ atm)/(\text{mol}\ K)$ is the gas constant.

The van der Waals equation gives the relationship between these quantities for a real gas by

$$\left(P + \frac{n^2a}{V^2}\right)(V - nb) = nRT$$

where a and b are constants that are specific for each gas.

Calculate the volume of 2 mol CO_2 at temperature of 50°C, and pressure of 6 atm. For CO_2, $a = 3.59\ (L^2\ atm)/mol^2$, and b = 0.0427 L/ mol. Because CO_2 is a real gas, so we need to use the second equation for the solution. But for solving the second equation for the volume, obtain an appropriate guess of the volume from the first equation: ideal gas equation.

Exercise 20: Golden-ratio corresponds to the order of which method:

(A) Secant (B) Regula-Falsi (C) Fixed-Point Iteration (D) Newton-Raphson

Exercise 21: Which of the following methods, has an explicit formula that can be used to determine the required number of iterations in advance for achieving a given accuracy:

(A) Bisection (B) Regula-Falsi (C) Fixed-Point Iteration (D) Newton-Raphson (E) Secant

Exercise 22: The convergence rate of which of the following methods is highest:

(A) Bisection (B) Regula-Falsi (C) Fixed-Point Iteration (D) Newton-Raphson (E) Secant

■■■

Chapter **3**

Polynomial Interpolation

Corridor I: BASICS

Let's plan it

To unleash the topics of this Corridor, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

■■■

Corridor II: ANALYSIS

Let's think deep

3.8 Error of Interpolation

To unleash the topics of this Corridor, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

■■■

Corridor III: PROGRAMMING ARCADE

Let's do it

3.9 Algorithms and Implementations

The Newton's Divided Difference Interpolation Formula

To see more examples for practicing, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

For codes, please visit: https://github.com/DrAmjadAli11/SimplifiedNumericalAnalysis

■■■

3.9 Algorithms and Implementations

Question 21: Write down an algorithm (pseudo code) to interpolate or extrapolate the function at a point using the nth-degree Newton's Divided difference interpolating polynomial.

Algorithm: Given $\boldsymbol{n}+1$ data points, approximate $f(x)$ at $x = \boldsymbol{xp}$ with $P_{\boldsymbol{n}}(\boldsymbol{xp})$.

INPUTS:
$\left\{\begin{array}{l} \boldsymbol{n}\text{: an integer as the degree of interpolating polynomial} \\ \boldsymbol{x_i}, 0 \le i \le \boldsymbol{n}\text{: real values as the aribrary nodes} \\ \boldsymbol{f_i}, 0 \le i \le \boldsymbol{n}\text{: real values as the function values corresponding to } \boldsymbol{x_i} \text{ nodes} \\ \boldsymbol{xp}\text{: real values as the entries} \end{array}\right.$

OUTPUT: $\boldsymbol{fxp}$: a real number as an interpolated value at $\boldsymbol{x} = \boldsymbol{xp}$

Step 1 Receive the inputs as stated above

Step 2
```
for $i = 0, 1, \cdots, \boldsymbol{n}$
    $ddf_{i,0} = \boldsymbol{f_i}$        (Computing zeroth divided differences, $f[\boldsymbol{x_i}] = \boldsymbol{f_i}$)
```

Step 3 (Computing the divided differences of order 1 to $\boldsymbol{n}$)

```
for $j = 1, 2, \cdots, \boldsymbol{n}$
    for $i = 0, 1, \cdots, \boldsymbol{n} - j$
        $ddf_{i,j} = \frac{[ddf_{i+1,j-1} - ddf_{i,j-1}]}{[x_{i+j} - x_i]}$
```

$$\left(f[x_i, \cdots, x_{i+j}] = \frac{f[x_{i+1}, \cdots, x_{i+j}] - f[x_i, \cdots, x_{i+j-1}]}{x_{i+j} - x_i} \right)$$

Step 4 (Evaluating the interpolation polynomial at $\boldsymbol{xp}$)

```
Set $pro = 1$
Set $\boldsymbol{fxp} = ddf_{0,0}$
for $k = 1, 2, \cdots, \boldsymbol{n}$
    $pro = pro \times (xp - \boldsymbol{x}_{k-1})$
    $\boldsymbol{fxp} = \boldsymbol{fxp} + pro \times ddf_{0,k}$
```

$$\left(P_n = f[x_0] + \sum_{k=1}^{n} \left[f[x_0, \cdots, x_k] \prod_{t=0}^{k-1} (xp - x_t) \right] \right)$$

Step 5 Print the output: $\boldsymbol{fxp}$

STOP.

Question 22: Write a Python program for the second order Newton's Divided Difference Interpolation.

script_3.1: divided_difference2.ipynb

```
from numpy import *

n = 2                                   # degree of interpolating polynomial
f = [3,13, -23]
x = [1, -4,0]
ddf= zeros([3,3])

xp = float(input("Enter a value which the interpolate is to be obtained: "))

#computing zeroth divided difference
for i in range(n+1):
        ddf[i][0] = f[i]

#computing the divided difference of Order 1 to n
for j in range(1,n+1):
        for i in range(n-j+1):
                ddf[i][j] = (ddf[i+1][j-1] - ddf[i][j-1]) / (x[i+j] - x[i] )

pro = 1                         #evaluting the interpolating polynomial at xp
fxp = ddf[0][0]
for k in range(1,n+1):
        pro = pro *(xp - x[k-1])
        fxp = fxp + pro * ddf[0][k]

print("The interpolate or extrapolate value of function at x = xp:", fxp)
```

Output Console:

```
Enter a value which the interpolate is to be obtained: 0.5
The interpolate or extrapolate value of function at x = xp: -11.75
```

Question 23: Write a Python program for the Newton's Divided Difference Interpolation.

script_3.2: divided_differenceN.ipynb

```
from numpy import *

n = 3                   # degree of interpolating polynomial
f = zeros([n+1])
x = zeros([n+1])
ddf= zeros([n+1,n+1])
```

```

print("The Divided Difference Interpolation.")
print("Enter real values as the arbitrary nodes")
for i in range(n+1):
        x[i] = float(input(" "))                                # x[i] = float(x[i])

print("Enter real values as the function values corresponding to x_i nodes ")
for i in range(n+1):
        f[i] = float(input(" "))

xp = float(input("Enter a value which the interpolate is to be obtained: "))

#computing zeroth divided difference
for i in range(n+1):
        ddf[i][0] = f[i]

#computing the divided difference of Order 1 to n
for j in range(1,n+1):
        for i in range(n-j+1):
                ddf[i][j] = (ddf[i+1][j–1] – ddf[i][j–1]) / (x[i+j] – x[i] )

pro = 1                         #Evaluting the interpolation polynomial at xp
fxp = ddf[0][0]
for k in range(1,n+1):
        pro = pro *(xp – x[k–1])
        fxp = fxp + pro * ddf[0][k]

print("The interpolate or extrapolate value of function at x = xp is",fxp)
```

Output Console:

```
The Divided Difference Interpolation.
Enter real values as the arbitrary nodes
 -1
 2
 3
 6
Enter real values as the function values corresponding to x_i nodes
 -3
 5
 17
 21
Enter a value which the interpolate is to be obtained: 2
The interpolate or extrapolate value of function at x = xp is 5.0
```

■■■

Chapter Summary

- **Curve fitting** refers to the process of constructing a curve (a mathematical function) that reasonably fits the given discrete data points along a continuum. The obtained curve offers a simpler alternative to the original function (whose values at discrete points were given) that might be used to estimate the data values at points between the given points (and sometimes beyond the given data points, as well).

- ***Regression*** and ***Interpolation*** are the two basic approaches for curve fitting. Regression is the process of deriving a single curve that provides for the general trend of the data (and that curve is not required to pass through any of the data points). Interpolation is the process of fitting a curve (a single function or a piecewise function) that interpolates (passes through) each of the given data points.

- Suppose that the values of a function f at different points $x_0, x_1, x_2, \cdots, x_n$ are given. The points x_i are referred to as **nodes** or **arguments** and the $n+1$ ordered pairs $\left(x_i, f(x_i)\right), i = 0, 1, 2, \cdots, n$, are referred to as **data points** of f. ***Interpolation*** (or, more precisely, *polynomial interpolation*) refers to the process of approximating the value of f at any intermediate point to the given data points.

- The interpolation process consists of determining the *unique* polynomial $P_n(x)$ of degree at most n that interpolates (passes through) the given data points, i.e.,

$$P_n(x_i) \quad = \quad f(x_i)$$

  And then, the polynomial $P_n(x)$ serves as the formula to approximate the function values at intermediate points to the given data points and, thus, is referred to as ***interpolating polynomial***. If the polynomial $P_n(x)$ is used approximate the function values at beyond the given data points, then the process is called ***extrapolation***.

- **Newton's Divided Difference Interpolation:** For $n+1$ arbitrarily spaced data points, (x_0, f_0), (x_1, f_1), $\cdots$, (x_n, f_n), of a function f, the Newton's Divided Difference interpolation formula for the interpolating polynomial $P_n(x)$ of degree at most n is given by

$$\begin{aligned} P_n(x) \quad &= \quad f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) \\ &\quad + \cdots + f[x_0, x_1, x_2, \cdots, x_n](x - x_0)(x - x_1)\cdots(x - x_{n-1}) \end{aligned}$$

  or

$$P_n(x) \quad = \quad f[x_0] + \sum_{k=1}^{n} f[x_0, x_1, x_2, \cdots, x_k](x - x_0)(x - x_1)\cdots(x - x_{k-1})$$

  Here the ***k*th divided difference** of the function f with respect to the nodes $x_i, x_{i+1}, \cdots, x_{i+k}$ is denoted by $f[x_i, x_{i+1}, \cdots, x_{i+k}]$ and is recursively defined by

$$f[x_i, x_{i+1}, \cdots, x_{i+k}] \quad = \quad \frac{f[\,x_{i+1}, x_{i+2}, \cdots, x_{i+k}] - f\left[x_i, x_{i+1}, \cdots, x_{i+(k-1)}\right]}{x_{i+k} - x_i}$$

  with $f[x_i] = f(x_i) = f_i$ as the zeroth divided difference.

- **Lagrange Interpolation:** For $n+1$ arbitrarily spaced data points, $(x_0, f_0), (x_1, f_1), \cdots, (x_n, f_n)$, of a function f, the Lagrange interpolation formula for the interpolating polynomial $P_n(x)$ of degree at most n is given by

$$\begin{aligned} P_n(x) &= L_0(x)f(x_0) + L_1(x)f(x_1) + \cdots + L_n(x)f(x_n) \\ &= \sum_{k=0}^{n} L_k(x)\, f(x_k) \end{aligned}$$

Here $L_k(x)$ denotes the ***k*th Lagrange coefficient** (also called **cardinal polynomial**) and is defined by

$$L_k(x) = \frac{(x-x_0)(x-x_1)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_n)}{(x_k-x_0)(x_k-x_1)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)} = \prod_{\substack{j=0\\ j\neq k}}^{n} \frac{x-x_j}{x_k-x_j}$$

and satisfies the **Kronecker delta** equation:

$$L_k(x) = \begin{cases} 1 & \text{for } x = x_k \\ 0 & \text{for all } x, \text{except } x = x_k \end{cases}$$

- **First Theorem on Interpolation Error**: If $P_n(x)$ is the polynomial of degree at most n that interpolates a function f at $n+1$ arbitrary nodes $x_0, x_1, \cdots, x_n$ in an interval $[a,b]$ and if $f \in C^{(n+1)}[a,b]$, then for each x in $[a,b]$, there exists an ξ in (a,b) for which

$$E(x) = f(x) - P_n(x) = (x-x_0)(x-x_1)\cdots(x-x_n)\frac{f^{(n+1)}(\xi)}{(n+1)!}$$

Here $E(x)$ is the truncation error of the polynomial interpolation.

- A Lagrange interpolation formula can be obtained from the relevant Newton's Divided Difference interpolation formula, after some rearrangements.

- Suppose that $n+1$ data points, (x_0, f_0), $(x_1, f_1), \cdots, (x_n, f_n)$, of a function f are given on the interval $[a,b]$ for consecutively arranged and equispaced nodes $x_0, x_1, x_2, \cdots, x_n$, such that

$a = x_0 < x_1 < x_2 < \cdots < x_{n-1} < x_n = b$

with astep size of length $h = x_i - x_{i-1}$, for $i = 1, 2, 3, \cdots, n$

and $f(x_i) = f_i$

The **Newton Forward-Difference Interpolation formula** for the interpolating polynomial $P_n(x)$ of degree at most n is given by

$$P_n(x) = f_0 + \alpha\Delta f_0 + \frac{\alpha(\alpha-1)}{2!}\Delta^2 f_0 + \cdots + \frac{\alpha(\alpha-1)(\alpha-2)\cdots\big(\alpha-(n-1)\big)}{n!}\Delta^n f_0$$

where

$$\alpha = \frac{x - x_0}{h}$$

Here the ***k*th forward-difference** of f at x_i is denoted by $\Delta^k f_i$ and is recursively defined by

$$\Delta^k f_i = \Delta(\Delta^{k-1} f_i) = \Delta^{k-1} f_{i+1} - \Delta^{k-1} f_i \qquad \text{for } k = 2, 3, \cdots, n$$

with $\Delta f_i = f_{i+1} - f_i$

The **Newton Backward-Difference Interpolation formula** for the interpolating polynomial $P_n(x)$ of degree at most n is given by

$$P_n(x) = f_n + \beta \nabla f_n + \frac{\beta(\beta+1)}{2!} \nabla^2 f_n + \cdots + \frac{\beta(\beta+1)(\beta+2)\cdots(\beta+(n-1))}{n!} \nabla^n f_n$$

where

$$\beta = \frac{x - x_n}{h}$$

Here the ***k*th backward-difference** of f at x_i is denoted by $\nabla^k f_i$ and is recursively defined by

$$\nabla^k f_i = \nabla(\nabla^{k-1} f_i) = \nabla^{k-1} f_i - \nabla^{k-1} f_{i-1} \qquad \text{for } k = 2, 3, \cdots, n$$

with $\nabla f_i = f_i - f_{i-1}$

- There are central difference interpolation formulas also available in the literature, which are more suited for approximation of a function value around mid of the interval of interpolation. Following are the examples of some well-known central difference interpolation formulas:
    - Gauss Forward Difference Interpolation Formula
    - Gauss Backward Difference Interpolation Formula
    - Stirling's Central Difference Interpolation Formula
    - Bessel's Central Difference Interpolation Formula
    - Everrett's Central Difference Interpolation Formula

■■■

Chapter Exercises

Exercise 01: Find the linear interpolating polynomial passing through the following set of pairs of the points.

(i) $\{(0.1, \sin(0.1)), \quad (0.2, \sin(0.2))\}$

(ii) $\left\{\left(1.2, \frac{1}{(1.2)^2}\right), \quad \left(1.4, \frac{1}{(1.4)^2}\right)\right\}$

(iii) $\{(1,7), \quad (2,4)\}$

(iv) $\left\{(1, e^{-1}), \quad \left(1.5, e^{-\frac{1}{1.5}}\right)\right\}$

Exercise 02: Construct the interpolating polynomial to approximate the following functions at $x = 0.25$. Use the arguments $x_0 = -0.3, x_1 = 0, x_2 = 0.4$.

(i) $f(x) = \ln(1 + x)$

(ii) $f(x) = e^{-x^2}$

(iii) $f(x) = \tan x^2$

(iv) $f(x) = \frac{1}{\sqrt{x^2-1}}$

Exercise 03: Use the Lagrange Interpolating Polynomial and the Newton's Divided Difference Interpolating polynomial of the appropriate degree to interpolate the following:

(i) Compute $f(1.5)$, given that, $f(0.5) = 0.479, f(1.0) = 0.841, f(2.0) = 0.909$

(ii) Compute $f(3.6)$, given that $f(3.0) = 0.1506, f(4.0) = 0.3001, f(4.5) = 0.2663, f(4.7) = 0.2346$

(iii) Compute $f(2/3)$, given that,

$f(1.1) = -0.071812, \quad f(1.3) = -0.024750, \quad f(1.7) = 0.334937, \quad f(2.0) = 1.101000$

Exercise 04: Find the missing value in the following table using the Newton's Divided Difference Interpolating polynomial.

x	-1	1	2	3
$f(x)$	-21	15	?	3

Exercise 05: Find the missing value in the following table using Lagrange Interpolating Polynomial

x	-2	0	2	4	6
$f(x)$	33	5	9	?	113

Exercise 06: Find, for what values of x, y attained extreme values using the data given below

x	3	4	5	6	7	8
y	0.205	0.240	0.259	0.262	0.250	0.224

Exercise 07: Use Lagrange Interpolating Polynomial of the appropriate degree to complete the record of the export of a certain commodity during six years

Year: x	1981	1982	1983	1984	1985	1986
Export: y	43	84	93	?	105	157

Exercise 08: Use the Newton's Divided Difference Interpolating Polynomial to obtain an interpolation that passing through the following points

x	0	0.1	0.3	0.4	0.7	0.8
y	-1.5	-1.27	-0.98	-0.63	-0.22	0.25

Exercise 09: Find a bound for the error associated with linear polynomial interpolation for the following function. Use the arguments $x_0 = 0, x_1 = 0.4$.

(i) $f(x) = \ln(1 + x)$

(ii) $f(x) = e^{-x^2}$

(iii) $f(x) = \tan x^2$

(iv) $f(x) = \frac{1}{\sqrt{x^2-1}}$

Exercise 10: Find a bound for the error associated with quadratic polynomial interpolation for the following function. Use the arguments $x_0 = 0, x_1 = 0.1, x_2 = 0.4$. $f(x) = \sin x + \cos x$

(i) $f(x) = x \ln x$

(ii) $f(x) = x \sin x - x^3 + 2x - 1$

(iii) $f(x) = \sqrt{x - x^2}$

Exercise 11: Find a bound for the error associated with cubic polynomial interpolation for the following function. Use the arguments $x_0 = 1, x_1 = 1.3, x_2 = 1.6, x_3 = 2.0$

(i) $f(x) = \sin(e^{-x} - 1)$

(ii) $f(x) = \ln x - x^4 + x^2 - 1$

(iii) $f(x) = x^2 e^{-x^2}$

(iv) $f(x) = \frac{1}{\sqrt{1+x}}$

Exercise 12: Construct the Newton's Forward and Backward Difference Interpolating polynomials passes through the points $(0.2, 0.9980)$, $(0.4, 0.9686)$, $(0.6, 0.8443)$, and $(0.8, 0.5358)$.

Exercise 13: Construct the Newton's Forward and Backward Difference Interpolating polynomials to approximate the following functions at $x = 1.2$ and 2.0. Use the arguments $x_0 = 1.1, x_1 = 1.3, x_2 = 1.5, x_3 = 1.7, x_4 = 1.9$

(i) $f(x) = \ln(1 + x)$

(ii) $f(x) = e^{-x^2}$

(iii) $f(x) = \tan x^2$

(iv) $f(x) = \frac{1}{\sqrt{x^2-1}}$

Exercise 14: Some data of the speed (V) versus drag coefficient (C_d) of a cricket ball is given in the following table: Estimate C_d at $V = 150\ km/h$.

V in km/h	C_d
0	0.5
80	0.48
120	0.39
160	0.32

Exercise 15: The mileages covered by a car per liter of fuel at different speeds are shown is the table below:

Speed in km/h	Mileage covered in km/l
60	14.2
75	16.1
90	14.8
105	13.7
120	11.5

Using interpolation, approximate the fuel efficiency of the car at the speed of 100 km/h.

Hint for the Solution: Use any interpolation formula, preferable the Newton's Backward Difference Interpolation formula.

Exercise 16: Some recorded data of number of deaths due to Novel Coronavirus (2019-nCoV) is given in the table below. Use interpolation to determine number of deaths on January 29 and 31, 2020.

Date	Number of Deaths
Jan. 24	16
Jan. 26	24
Jan. 28	26
Jan. 30	43
Feb. 1	45

Hint for the Solution: The given data spans over 9 days. The function values are given for $x = 1, 3, 5, 7, 9$. Find an interpolating polynomial and use it to calculate value at $x = 6$ and $x = 8$ for the desired solutions.

Exercise 17: The census data of Pakistan is given in the following table (source: Pakistan Bureau of Statistics):

Census Year	Population in thousands
1951	33740
1961	42880
1972	65309
1981	84254
1998	132352
2017	207774

Use interpolation to determine the population for the year 2010.

Hint for the Solution: The given data spans over 67 years. The function values are given for $x = 1, 11, 22, 31, 48, 67$. Find an interpolating polynomial and use it to calculate value at $x = 60$ for the desired solution.

Exercise 18: Suppose that a table lists the values of the tangent function for the angles ranging from 0^o to 45^o in increments of 5^o. What is the largest error that we would introduce by performing linear interpolation between successive values in this table?

■■■

Chapter **4**

Numerical Integration

Corridor I: BASICS

Let's plan it

To unleash the topics of this Corridor, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

■■■

Question 12: Tabulate Closed Newton-Cotes Integration formulas with relevant features, for both the basic and the composite forms, separately.

Suppose that n data points, (x_j, f_j), where $f(x_j) = f_j$, of the integrand $f(x)$ are given on the interval $[a, b] = [x_0, x_1]$ for consecutively arranged and equispaced nodes x_j such that $h = (b - a)/n$. The Closed Newton Cotes quadrature formulas for the definite integral $= \int_{x_0}^{x_n} f(x)\, dx$ are tabulated below.

Numerical Integration Method	**Formula**	Required number of function values at equidistant points	**Interpolating polynomial used for integral evaluation** (to derive the formula)
Rectangular Rule	$\mathrm{I} = h(f_0)$ (starting-point rule) or $\mathrm{I} = h(f_1)$ (end-point rule) or $\mathrm{I} = h(f^*)$ (mid-point rule) where $f^* = f\left(\frac{x_0+x_1}{2}\right)$	one	Polynomial of degree 0 (constant function)
Trapezoidal Rule	$\mathrm{I} = \frac{h}{2}[f_0 + f_1]$	two	Polynomial of degree 1 (linear polynomial)
Simpson's 1/3 Rule	$\mathrm{I} = \frac{h}{3}[f_0 + 4f_1 + f_2]$	three	Polynomial of degree 2 (quadratic polynomial)
Simpson's 3/8 Rule	$\mathrm{I} = \frac{3h}{8}[f_0 + 3(f_1 + f_2) + f_3]$	four	Polynomial of degree 3 (cubic polynomial)
Boole's Rule (Milne's Rule)	$\mathrm{I} = \frac{2h}{45}[7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4]$	five	Polynomial of degree 4
Six-Point Rule	$\mathrm{I} = \frac{5h}{288}[19f_0 + 75f_1 + 50f_2 + 50f_3 + 75f_4 + 19f_5]$	six	Polynomial of degree 5
Weddle's Rule	$\mathrm{I} = \frac{h}{140}[41f_0 + 216f_1 + 27f_2 + 272f_3 + 27f_4 + 216f_5 + 41f_6]$	seven	Polynomial of degree 6

Numerical Integration Method	**Formula** (for $n+1$ data points, $(x_j, f_j), j = 0,1,2,\cdots,n$, and n subintervals of equal length $h = (x_n - x_0)/n$)	**Possible values of n** (K represents the number of multiple applications of the formula)	**Interpolating polynomial used for integral evaluation** (to derive the formula)
Composite Rectangular Rule	$\mathrm{I} = h[f_0 + f_1 + f_2 + \cdots + f_{n-1}]$ (starting-point rule) or $\mathrm{I} = h[f_1 + f_2 + \cdots + f_{n-1} + f_n]$ (end-point rule) or $\mathrm{I} = h[f_1^* + f_2^* + \cdots + f_{n-1}^* + f_n^*]$ (mid-point rule) where $f_j^* = f\left(\frac{x_{j-1}+x_j}{2}\right)$, for $j = 1,2,3,\ldots,n$	n = 1, 2, 3, ... (i.e., $n = K$ could be any positive integer)	Piecewise polynomial of degree 0 (piecewise-constant function)
Composite Trapezoidal Rule	$\mathrm{I} = \frac{h}{2}[f_0 + 2(f_1 + f_2 + \cdots + f_{n-1}) + f_n]$	$n = 1,2,3,\cdots$ (i.e., $n = K$ could be any positive integer)	Piecewise polynomial of degree 1 (piecewise-linear)
Composite Simpson's 1/3 Rule	$\mathrm{I} = \frac{h}{3}[f_0 + 4(f_1 + f_3 + \cdots + f_{n-1}) + 2(f_2 + f_4 + \cdots + f_{n-2}) + f_n]$	n = 2, 4, 6, ... (i.e., $n = 2K$, where K = 1, 2, 3, ...)	Piecewise polynomial of degree 2 (piecewise-quadratic)
Composite Simpson's 3/8 Rule	$\mathrm{I} = \frac{3h}{8}[f_0 + 3(f_1 + f_2) + 2(f_3) + 3(f_4 + f_5) + 2(f_6) + \cdots + 3(f_{n-2} + f_{n-1}) + f_n]$	n = 3, 6, 9, ... (i.e., $n = 3K$, where K = 1, 2, 3, ...)	Piecewise polynomial of degree 3 (piecewise-cubic)
Composite Boole's Rule (Composite Milne's Rule)	$\mathrm{I} = \frac{2h}{45}[7f_0 + 32(f_1 + f_5 + f_9 + \cdots + f_{n-3}) + 12(f_2 + f_6 + f_{10} + \cdots + f_{n-2}) + 32(f_3 + f_7 + f_{11} + \cdots + f_{n-1}) + 14(f_4 + f_8 + f_{12} + \cdots + f_{n-4}) + 7f_n]$	n = 4, 8, 12, ... (i.e., $n = 4K$, where K = 1, 2, 3, ...)	Piecewise polynomial of degree 4
Composite Six-Point Rule	$\mathrm{I} = \frac{5h}{288}[19f_0 + 75(f_1 + f_6 + f_{11} + \cdots + f_{n-4}) + 50(f_2 + f_7 + f_{12} + \cdots + f_{n-3}) + 50(f_3 + f_8 + f_{13} + \cdots + f_{n-2}) + 75(f_4 + f_9 + f_{14} + \cdots + f_{n-1}) + 38(f_5 + f_{10} + f_{15} + \cdots + f_{n-5}) + 19f_n]$	n = 5, 10, 15, ... (i.e., $n = 5K$, where K = 1, 2, 3, ...)	Piecewise polynomial of degree 5
Composite Weddle's Rule	$\mathrm{I} = \frac{h}{140}[41f_0 + 216(f_1 + f_7 + f_{13} + \cdots + f_{n-5}) + 27(f_2 + f_8 + f_{14} + \cdots + f_{n-4}) + 272(f_3 + f_9 + f_{15} + \cdots + f_{n-3}) + 27(f_4 + f_{10} + f_{16} + \cdots + f_{n-2}) + 216(f_5 + f_{11} + f_{17} + \cdots + f_{n-1}) + 82(f_6 + f_{12} + f_{18} + \cdots + f_{n-6}) + 41f_n]$	n = 6, 12, 18, ... (i.e., $n = 6K$, where K = 1, 2, 3, ...)	Piecewise polynomial of degree 6

■■■

Corridor II: ANALYSIS

Let's think deep

4.5 Truncation Error of the Trapezoidal Rule
4.6 Truncation Error of the Simpson's 1/3 Rule
4.7 Further Discussions
4.8 The Gaussian Quadrature

To unleash the topics of this Corridor, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

■■■

Corridor III: PROGRAMMING ARCADE

Let's do it

4.9 Algorithms and Implementations
The Composite Trapezoidal Rule
The Composite Simpson's 1/3 Rul
The Composite Simpson's 3/8 Rule

To see more examples for practicing, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

For codes, please visit: https://github.com/DrAmjadAli11/SimplifiedNumericalAnalysis

■■■

4.9 Algorithms and Implementations

Question 13: Write down the algorithm (pseudo-code) of the Composite Trapezoidal rule for numerical integration of definite integrals.

The Composite Trapezoidal rule to integrate a function f over the interval $[a, b]$ is given by,

$$I = \int_a^b f(x)dx \cong \frac{h}{2}[f_0 + 2(f_1 + f_2 + \cdots + f_{n-1}) + f_n]$$

$$\text{where} \quad h = \frac{b-a}{n} = \frac{x_n - x_0}{n}, \qquad f_j = f(x_j) \quad \text{and} \quad x_j = x_0 + jh \quad \text{for } j = 0, 1, 2, \cdots, n$$

Algorithm: To approximate the definite integral $I = \int_a^b f(x)\,dx$ using the formula:

$$I = \frac{h}{2}[f_0 + 2(f_1 + f_2 + f_3 + \cdots + f_{n-1}) + f_n]$$

INPUTS: $\begin{cases} \boldsymbol{a} \text{ and } \boldsymbol{b}\text{: two real values as the endpoints of the interval of integration} \\ \boldsymbol{n}\text{: a positive integer as the number of subintervals} \end{cases}$

OUTPUT: $\boldsymbol{I}$: a real number as an approximation to the integral

```
Step 1    Receive the inputs as stated above

Step 2    Set real number $x0 = a$
          Set real number $xn = b$
          Set real number $h = (xn - x0)/n$
          Set real number $fx0$ as the value $f(a)$
          Set real number $fxn$ as the value $f(b)$

Step 3    Set $I = fx0 + fxn$

Step 4    Set real number $xc = x0$
          Set real number $sum = 0$

          for $j = 1, 2, \cdots, n - 1$
              Set $xc = xc + h$
              Set $fxc$ as the value $f(xc)$
              Set $sum = sum + fxc$          (Forming $f_1 + f_2 + \cdots + f_{n-1}$)
          end for

Step 5    Set $I = (h/2) \times (I + 2 \times sum)$

Step 6    Print the output: $I$

STOP.
```

Question 14: Write a Python program to evaluate the integral of $f(x) = \sqrt{x^2+1}$ over $[0, 2]$ with 12 subintervals using the Composite Trapezoidal rule.

script_4.1: trapezoidal.ipynb

```
from numpy import *

print("The Composite Trapezoidal Rule")
x0 = float(input("Enter the lower limit of integral: "))
xn = float(input("Enter the upper limit of integral: "))
n = int(input("Enter the number of subintervals n: "))

h = (xn – x0) / n
fx0 = sqrt(x0**2 + 1)
fxn = sqrt(xn**2 + 1)
I = fx0 + fxn
xc = x0
sum = 0

for i in range(1,n):
        xc = xc + h
        fxc = sqrt(xc**2 + 1)
        sum = sum + fxc
I = (h / 2) * (I + 2 * sum)

print("The Approximate Integral =", I)
```

Output Console:

```
The Composite Trapezoidal Rule
Enter the lower limit of integral: 0
Enter the upper limit of integral: 2
Enter the number of subintervals n: 12
The Approximate Integral = 2.9599562632284453
```

Question 15: Write a Python program to evaluate the integral of $f(x) = \sqrt{x^2+1}$ over $[0, 2]$ with 12 subintervals using the Composite Trapezoidal rule. Use #define directive for evaluating $f(x)$ at the different nodes (i.e., for finding the values of f at the different nodes).

script_4.2: trapezoidal2.ipynb

```
from numpy import *

def fval(x):
        y = sqrt(x**2 + 1)
        return(y)
```

```python

print("The Composite Trapezoidal Rule")
x0 = float(input("Enter the lower limit of integral: "))
xn = float(input("Enter the upper limit of integral: "))
n = int(input("Enter the number of subintervals n: "))

h = (xn – x0) / n
fx0 = fval(x0)
fxn = fval(xn)
I = fx0 + fxn
xc = x0
sum = 0

for i in range(1,n):
        xc = xc + h
        fxc = fval(xc)
        sum = sum + fxc
I = (h / 2) * (I + 2 * sum)

print("The Approximate Integral =",I)
```

Output Console:

```
The Composite Trapezoidal Rule
Enter the lower limit of integral: 0
Enter the upper limit of integral: 2
Enter the number of subintervals n: 12
The Approximate Integral = 2.9599562632284453
```

Question 16: Write down the algorithm (pseudo-code) of the Composite Simpson's 1/3 rule for numerical integration of definite integrals.

The Composite Simpson's 1/3 rule to integrate a function f over the interval $[a, b]$ is given by,

$$I = \int_a^b f(x)dx \cong \frac{h}{3}[f_0 + 4(f_1 + f_3 + \cdots + f_{n-1}) + 2(f_2 + f_4 + \cdots + f_{n-2}) + f_n]$$

$$\text{where} \quad h = \frac{b-a}{n} = \frac{x_n - x_0}{n}, \qquad f_j = f(x_j) \quad \text{and} \quad x_j = x_0 + jh \quad \text{for } j = 0, 1, 2, \cdots, n$$

Algorithm: To approximate the definite integral $I = \int_a^b f(x)\,dx$ using the formula:

$$I = \frac{h}{3}[f_0 + 4(f_1 + f_3 + \cdots + f_{n-1}) + 2(f_2 + f_4 + \cdots + f_{n-2}) + f_n]$$

INPUTS: $\begin{cases} \boldsymbol{a} \text{ and } \boldsymbol{b}\text{: two real values as the endpoints of the interval of integration} \\ \boldsymbol{n}\text{: a positive even integer as the number of subintervals} \end{cases}$

OUTPUT: ***I***: a real number as an approximation to the integral

```
Step 1    Receive the inputs as stated above

Step 2    Set real number $\boldsymbol{x0} = \boldsymbol{a}$
          Set real number $\boldsymbol{xn} = \boldsymbol{b}$
          Set real number $\boldsymbol{h} = (\boldsymbol{xn} - \boldsymbol{x0})/\boldsymbol{n}$
          Set real number $\boldsymbol{fx0}$ as the value $f(\boldsymbol{a})$
          Set real number $\boldsymbol{fxn}$ as the value $f(\boldsymbol{b})$

Step 3    Set $\boldsymbol{I} = \boldsymbol{fx0} + \boldsymbol{fxn}$

Step 4    Set real number $\boldsymbol{xc} = \boldsymbol{x0}$
          Set real number $\boldsymbol{sum1} = \boldsymbol{0}$
          Set real number $\boldsymbol{sum2} = \boldsymbol{0}$

          for $j = 1, 2, \cdots, \boldsymbol{n} - 1$
              Set $\boldsymbol{xc} = \boldsymbol{xc} + \boldsymbol{h}$
              Set $\boldsymbol{fxc}$ as the value $f(\boldsymbol{xc})$
              if $j$ is odd
                  Set $\boldsymbol{sum1} = \boldsymbol{sum1} + \boldsymbol{fxc}$        (Forming $f_1 + f_3 + \cdots + f_{n-1}$)
              else
                  Set $\boldsymbol{sum2} = \boldsymbol{sum2} + \boldsymbol{fxc}$        (Forming $f_2 + f_4 + \cdots + f_{n-2}$)
          end for

Step 5    Set $\boldsymbol{I} = (\boldsymbol{h}/\boldsymbol{3}) \times (\boldsymbol{I} + \boldsymbol{4} \times \boldsymbol{sum1} + \boldsymbol{2} \times \boldsymbol{sum2})$

Step 6    Print the output: $\boldsymbol{I}$ ; STOP.
```

Question 17: Write a Python program to evaluate the integral of $f(x) = \sqrt{x^2 + 1}$ over $[0, 2]$ with 12 subintervals using the Composite Simpson's 1/3 rule.

script_4.3: simpsons13.ipynb

```python
from numpy import *

def fval(x):
        y = sqrt(x**2 + 1)
        return(y)

print("The Composite Simpson’s Rule")
x0 = float(input("Enter the lower limit of integral: "))
xn = float(input("Enter the upper limit of integral: "))
n = int(input("Enter the number of subintervals n: "))

h = (xn – x0) / n
fx0 = fval(x0)
fxn = fval(xn)
```

```
I = fx0 + fxn
xc = x0
sum1 = 0
sum2 = 0

for i in range(1,n):
        xc = xc + h
        fxc = fval(xc)
        if i%2 != 0:
                sum1 = sum1 + fxc
        else:
                sum2 = sum2 + fxc
I = (h / 3) * (I + 4 * sum1 + 2 * sum2)

print("The Approximate Integral =", I)
```

Output Console:

```
The Composite Simpson’s Rule
Enter the lower limit of integral: 0
Enter the upper limit of integral: 2
Enter the number of subintervals n: 12
The Approximate Integral = 2.957885258976941
```

Question 18: Write down the algorithm (pseudo-code) of the Composite Simpson’s 3/8 rule for numerical integration of definite integrals.

Algorithm: To approximate the definite integral $I = \int_a^b f(x)\,dx$ using the formula:

$$I = \frac{3h}{8}[f_0 + 3(f_1 + f_2 + f_4 + f_5 + \cdots + f_{n-2} + f_{n-1}) + 2(f_3 + f_6 + \cdots + f_{n-3}) + f_n]$$

INPUTS: $\begin{cases} \boldsymbol{a} \text{ and } \boldsymbol{b}\text{: two real values as the endpoints of the interval of integration} \\ \boldsymbol{n}\text{: a positive integer (multiple of 3) as the number of subintervals} \end{cases}$

OUTPUT: $\boldsymbol{I}$: a real number as an approximation to the integral

```
Step 1    Receive the inputs as stated above

Step 2    Set real number $x0 = a$
          Set real number $xn = b$
          Set real number $h = (xn - x0)/n$
          Set real number $fx0$ as the value $f(a)$
          Set real number $fxn$ as the value $f(b)$

Step 3    Set $I = fx0 + fxn$

Step 4    Set real number $xc = x0$
          Set real number $sum1 = 0$
          Set real number $sum2 = 0$
```

```
for $j = 1, 2, \cdots, \boldsymbol{n} - 1$
    Set $\boldsymbol{xc} = \boldsymbol{xc} + \boldsymbol{h}$
    Set $\boldsymbol{fxc}$ as the value $f(\boldsymbol{xc})$
    if $j$ is divisible by 3
        Set $\boldsymbol{sum2} = \boldsymbol{sum2} + \boldsymbol{fxc}$        (Forming $f_3 + f_6 + \cdots + f_{n-3}$)
    else
        Set $\boldsymbol{sum1} = \boldsymbol{sum1} + \boldsymbol{fxc}$        (Forming $f_1 + f_2 + f_4 + f_5 + \cdots + f_{n-2} + f_{n-1}$)
end for
```

Step 5 Set $\boldsymbol{I} = (\boldsymbol{3} \times \boldsymbol{h}/\boldsymbol{8}) \times (\boldsymbol{I} + \boldsymbol{3} \times \boldsymbol{sum1} + \boldsymbol{2} \times \boldsymbol{sum2})$

Step 6 Print the outpu.

STOP.

Question 19: Write a Python program to evaluate the integral of $f(x) = \sqrt{x^2 + 1}$ over $[0, 2]$ with 12 subintervals using the Composite Simpson's $3/8$ rule.

script_4.3: simpson38.ipynb

```python
from numpy import *

def fval(x):
	y = sqrt(x**2 + 1)
	return(y)

print("The Composite Simpson’s 3/8 Rule")
x0 = float(input("Enter the lower limit of integral: "))
xn = float(input("Enter the upper limit of integral: "))
n = int(input("Enter the number of subintervals n: "))

h = (xn – x0) / n
fx0 = fval(x0)
fxn = fval(xn)
I = fx0 + fxn
xc = x0
sum1 = 0
sum2 = 0

for i in range(1,n):
	xc = xc + h
	fxc = fval(xc)
	if i%3 != 0:
		sum2 = sum2 + fxc
	else:
		sum1 = sum1 + fxc
I = (3 * h/8) * (I + 3*sum1 + 2*sum2)

print("The Approximate Integral =",I)
```

Output Console:

```
The Composite Simpson’s 3/8 Rule
Enter the lower limit of integral: 0
Enter the upper limit of integral: 2
Enter the number of subintervals n: 12
The Approximate Integral = 2.490906146724771
```

Remark: Likewise the programs in the solutions of Problem 19, the programmer can modify the programs in the solutions of Problems 20 and 22 to evaluate the function values at the desired nodes through the use of user-defined function and inline function (using #define).

■■■

Chapter Summary

- **Numerical integration** or **quadrature** refers to the process of numerically approximating the value of the integral $I = \int_a^b f(x)\,dx$, by using the values of f at a finite number of sample points. The limits of integration could be finite, semi-finite, or infinite.

- The integral is approximated by a numerical **integration rule** or **quadrature formula**, Q_f, which is a linear combination of certain function values:

$$I = \int_a^b f(x)\,dx \cong Q_f = \sum_{j=0}^{n} \omega_j \cdot f(x_j)$$

  Here x_i are the ordered points, called the ***quadrature nodes*** (or simply ***nodes***), taken usually within the limits of integration at which the function values $f(x_j)$ are known and ω_j are called the ***weights*** of the quadrature formula.

- The quadrature formula satisfies the property that

$$I = \int_a^b f(x)\,dx = Q_f + E_f,$$

  where E_f is the **truncation error** (also called the **error term**) associated with the quadrature formula.

- The Newton-Cotes integration formulas are based on the approach that $n+1$ number of equispaced and ordered nodes are chosen within the limits of integration and the integrand function is replaced by an interpolating polynomial of degree at most n by using the nodes, and then the analytic integration of the polynomial is performed to obtain the formula. A Composite Newton-Cotes integration formula is obtained by applying the relevant Newton-Cotes formula in each of the different consecutive segments of the interval of integration and then summing the integrals over all the segments.

- The examples of Newton-Cotes integration formulas include Trapezoidal rule, Simpson's 1/3 rule, Simpson's 3/8 rule, Boole's rule, Six-Point rule, and Weddle's rule.
- The Trapezoidal rule to numerically integrate the function f over the interval $[a, b]$ is

$$I = \int_a^b f(x)dx \cong \frac{(b-a)}{2}[f(a) + f(b)]$$

- The Composite Trapezoidal rule to integrate a function f over the interval $[a, b]$ is given by,

$$I = \int_a^b f(x)dx \cong \frac{h}{2}[f_0 + 2(f_1 + f_2 + \cdots + f_{n-1}) + f_n]$$

$$\text{where} \quad h = \frac{b-a}{n} = \frac{x_n - x_0}{n}, \qquad f_j = f(x_j) \quad \text{and} \quad x_j = x_0 + jh \quad \text{for } j = 0, 1, 2, \cdots, n$$

- The Simpson's 1/3 rule to integrate a function f over the interval $[a, b]$ is given by,

$$I = \int_a^b f(x)dx \cong \frac{h}{3}[f_0 + 4f_1 + f_2]$$

$$\text{where} \quad h = \frac{b-a}{2} = \frac{x_2 - x_0}{2}, \qquad f_j = f(x_j) \quad \text{and} \quad x_j = x_0 + jh \quad \text{for } j = 0, 1, 2$$

- The Composite Simpson's 1/3 rule to integrate a function f over the interval $[a, b]$ is given by,

$$I = \int_a^b f(x)dx \cong \frac{h}{3}[f_0 + 4(f_1 + f_3 + \cdots + f_{n-1}) + 2(f_2 + f_4 + \cdots + f_{n-2}) + f_n]$$

$$\text{where} \quad h = \frac{b-a}{n} = \frac{x_n - x_0}{n}, \qquad f_j = f(x_j) \quad \text{and} \quad x_j = x_0 + jh \quad \text{for } j = 0, 1, 2, \cdots, n$$

- A comprehensive summary of the Newton-Cotes formulas and the Composite Newton-Cotes formulas can be found under Question 12 (page 252).

- The error term E_T of order $\mathcal{O}(h^3)$ associated with the Trapezoidal rule in approximating $I = \int_a^b f(x)\,dx$ is given by,

$$E_T = -\frac{1}{12}h^3 f''(\xi),$$

for some appropriate point ξ in (a, b) and $h = b - a$.

- The error term E_{CT} of order $\mathcal{O}(h^2)$ associated with the Composite Trapezoidal rule in approximating $I = \int_a^b f(x)\,dx$ is given by,

$$E_{CT} = -\frac{b-a}{12}h^2 f''(\eta),$$

for some appropriate point η in (a, b) and $h = (b - a)/n$, where n is the number of subintervals of $[a, b]$.

- The error term E_S of order $O(h^5)$ associated with the Simpson's 1/3 rule in approximating $I = \int_a^b f(x)\,dx$ is given by,

$$E_S = -\frac{1}{90} h^5 f^{(4)}(\xi),$$

for some appropriate point ξ in (a, b) and $h = (b - a)/2$.

- The error term E_{CS} of order $O(h^4)$ associated with the Composite Simpson's 1/3 rule in approximating $I = \int_a^b f(x)\,dx$ is given by,

$$E_{CS} = -\frac{b-a}{180} h^4 f^{(4)}(\eta)$$

for some appropriate point η in (a, b) and $h = (b - a)/n$, where n is the number of subintervals of $[a, b]$.

- Suppose I_h denotes the approximate integral using a quadrature formula with step size h, and E_h denotes the associated error. Then, the exact integral $= I_h + E_h$

  Similarly, suppose $I_{h/2}$ denotes the approximate integral using the same quadrature formula with a step size $h/2$, and $E_{h/2}$ denotes the associated error. Then, the exact integral $= I_{h/2} + E_{h/2}$

  According to the interval halving method, for a Newton-Cotes integration formula with an error of order $O(h^N)$ an estimate of the error $E_{h/2}$ is given by,

$$E_{h/2} \cong \frac{1}{2^N - 1}\left(I_{h/2} - I_h\right)$$

  This leads to a better approximation of the integral as below:

$$I \cong I_{h/2} + \frac{1}{2^N - 1}\left(I_{h/2} - I_h\right)$$

  This corresponds to a special process called ***Richardson Extrapolation***, in which two estimates of the solution are used to obtain a third approximation, which is a more accurate one. This approach for numerical integration forms an initial stage of a relatively broader way of numerical integration, called ***Romberg Integration***. Recall that for the Composite Trapezoidal rule $N = 2$, and for the Composite Simpson's 1/3 rule $N = 4$.

- There could be several approaches for improving the estimates of the integrals:
  - Using smaller step size (or larger number of subintervals)
  - Using higher-order formula (e.g., using the Simpson's rule instead of the Trapezoidal rule)
  - Using Richardson's extrapolation (i.e., using two less accurate estimates to obtain a more accurate estimate).

- The ***degree of precision***, also referred to as the *order of accuracy*, of a quadrature formula is p if and only if the associated truncation error is zero for all polynomials of degree less than or equal to p, and the error is not zero for some polynomial of degree greater than p. Note that the Trapezoidal rule is based on the interpolating polynomial of degree 1 (linear polynomial). Therefore, it produces the exact result while integrating a polynomial of degree 1. Hence it has the degree of precision as 1. The Simpson's 1/3 rule might be expected to have a degree of precision as 2 because it is based on interpolating polynomial of degree 2 (quadratic polynomial). However, it produces the exact result while integrating a polynomial of degree 2, as well as degree 3. Hence, it has the degree of precision as 3. This fact is also evident while deriving the error term for the Simpson's 1/3 rule. This property, together with certain other reasons, makes the Composite Simpson's 1/3 rule often the best choice among the Newton-Cotes integration formulas.

- A concise description of the error terms associated with the Newton-Cotes formulas and relevant degrees of precision can be found under Question 23 (page 276).

- The Gaussian Quadrature is an advanced numerical integration technique in which the quadrature nodes are selected in the interval of integration using the roots of some special polynomial to obtain an optimal approximation of the integral.

■■■

Chapter Exercises

Exercise 01: Approximate the integral $\int_a^b f(x)dx$ for the following functions over the interval $[0, 1]$ using the Trapezoidal, Simpson's 1/3 and Simpson's 3/8 rules.

$(i)\quad f(x) = x^2 + x - 1 \qquad (ii)\quad f(x) = \ln(1 + x) \qquad (iii)\quad f(x) = \dfrac{1}{1 + x^2}$

$(iv)\quad f(x) = \cos\left(\dfrac{x}{\pi}\right) \qquad (v)\quad f(x) = \dfrac{1}{\sqrt{x^2 + 4}}$

Exercise 02: Approximate the integral $\int_a^b f(x)dx$ for the following functions over the interval $[0, 1]$ using the Composite Trapezoidal, Simpson's 1/3, and Simpson's 3/8 rules with $h = 0.1$.

$(i)\quad f(x) = x^2 + x - 1 \qquad (ii)\quad f(x) = \ln(1 + x) \qquad (iii)\quad f(x) = \dfrac{1}{1 + x^2}$

$(iv)\quad f(x) = \cos\left(\dfrac{x}{\pi}\right) \qquad (v)\quad f(x) = \dfrac{1}{\sqrt{x^2 + 4}}$

Exercise 03: Approximate the integral

$$\int_4^{16} \sin\left(\frac{\pi\sqrt{x}}{4}\right) dx$$

using the Composite Trapezoidal rule with $h = 1$ and five-digit rounding arithmetic.

Exercise 04: Find an approximate value of the integral $\int_0^2 \left(2 + \sin\left(2\sqrt{x}\right)\right) dx$ using the Composite Trapezoidal rule for $n = 10$ and five-digit rounding arithmetic.

Exercise 05: Approximate the arc length of the following functions over the interval $[0, \pi]$

$$(i) \quad f(x) = \sin^2 x \qquad\qquad (ii) \quad f(x) = \ln\left(\frac{4+x}{\pi}\right)$$

using the Composite Simpson's 1/3 rule for $h = \frac{\pi}{6}$ and four-digit rounding arithmetic.

Exercise 06: Find the approximate value of the integral $\int_3^8 \left(f(x)\right)^2 dx$ using the Composite Simpson's 1/3 rule, given that

x_j	3	4	5	6	7	8	9
$f(x_j)$	0.205	0.240	0.259	0.262	0.250	0.224	0.220

Exercise 07: Approximate the area of a surface of revolution of the following curves:

$$(i) \quad x = 4y, \qquad\qquad (ii) \quad x = \tan y$$

about the $y - axis$ from $0 \le y \le 1$ using the Composite Simpson's 3/8 rule for $n = 10$ and four-digit rounding arithmetic.

Exercise 08: Find the approximate value of the integral

$$f(x) = \int_0^3 \frac{x}{x^2+3} dx$$

using the Composite Boole's rule with step size $h = 0.25$ and five-digit rounding arithmetic.

Exercise 09: Find the approximate value of the integral

$$f(x) = \int_2^5 \ln(x-1)\, dx$$

using the Composite Six-Point rule with step size $h = 0.3$ and five-digit rounding arithmetic.

Exercise 10: Find the approximate value of the integral

$$f(x) = \int_1^4 \sinh(x^2)\, dx$$

using the Composite Weddle's rule with step size $h = 0.25$ and five-digit rounding arithmetic.

Exercise 11: Suppose that $f(0) = 1$, $f(0.5) = 2.5$, $f(1) = 2$ and $f(0.25) = f(0.75) = \alpha$. Find α if the Composite Trapezoidal rule with $n = 4$ gives the value 1.75 for $\int_0^1 f(x)dx$.

Exercise 12: Suppose that $f(4) = 0.240$, $f(6) = 0.262$, $f(8) = 0.224, f(3) = f(5) = f(7) = \alpha$, and $f(9) = 0.220$ Find α if the Composite Simpson's 1/3 Rule gives the value 1.473 for

$$I = \int_{3}^{9} f(x)dx$$

Exercise 13: Suppose that $f(0.2) = 1.56$, $f(0.4) = 2.00$, $f(0.6) = 3.01, f(0.1) = f(0.3) = f(0.5) = \alpha$, and $f(0.7) = 3.32$ Find α if the Composite Simpson's 3/8 rule gives the value 1.30312 for

$$I = \int_{0.1}^{0.7} f(x)dx$$

Exercise 14: To approximate the integral of $f(x)$ over the interval $[0, 1]$ with an absolute error less than $\frac{1}{2} \times 10^{-4}$, how many subintervals are needed, in case of (a) the Composite Trapezoidal rule, (b) the Composite Simpson's 1/3 rule, and (c) the Composite Simpson's 3/8 rule? Given that,

(i) $f(x) = x^2 + x - 1$ (ii) $f(x) = \ln(1 + x)$ (iii) $f(x) = \dfrac{1}{1 + x^2}$

(iv) $f(x) = \cos\left(\dfrac{x}{\pi}\right)$ (v) $f(x) = \dfrac{1}{\sqrt{x^2 + 4}}$

Exercise 15: Suppose we wish to evaluate the integral

$$f(x) = \int_0^\pi \sin(\sqrt{x})dx$$

numerically, with an error of magnitude less than 10^{-5}. How many subintervals are needed if we wish to use the Composite Trapezoidal and Composite Simpson 1/3 rules?

Exercise 16: Find the number of subintervals n or step length h so that the error E_{TC} for the Composite Trapezoidal rule and error E_{SC} for the Composite Simpson's 1/3 rule is less than 5×10^{-4} for numerically integrating the Legendre polynomial,

$$P_4(x) = x^4 - \frac{6}{7}x^2 + \frac{3}{35}$$

over the interval $[-1, 1]$.

Exercise 17: Obtain an upper bound on the absolute error when the *Chebyshev* polynomial of degree four,

$$T_4(x) = 8x^4 - 8x^2 + 1$$

is integrated over the interval $[-1, 1]$ by means of the Composite Simpson's 3/8 rule.

Exercise 18: Obtain an upper bound on the absolute error when the *Laguerre* polynomial of degree four

$$L_4(x) = x^4 - 16x^3 + 72x^2 - 96x + 24$$

is integrated over the interval $[-1, 1]$, by means of the Composite Simpson's 3/8 rule.

Exercise 19: A car travels the loop of the racing track in 65 seconds. The speed of the car in meter/second is recorded after every 5 seconds as shown in the following table:

Time	0	5	10	15	20	25	30	35	40	45	50	55	60	65
Speed	0	40	62	70	72	65	71	79	75	72	68	63	75	82

Estimate the length of the loop of the racing track?

Hint for the Solution:

Clearly, the speed say S is shown to be a function of time, say t, and its values $S(t)$ for different time instants t are given. Obtain the estimate of the integral distance $= \int_0^{65} S(t)\,dt$ using any appropriate numerical integration rule with the data given in the Table.

Exercise 20: The prime number theorem states that the number of primes in an interval $a \leq x \leq b$ is approximately

$$\int_a^b \frac{1}{\ln x}\,dx$$

Estimate the number of primes existing in [50,150].

Hint for the Solution: Numerically evaluate the given integral for $a = 50$ and $b = 150$ using different values of $f(x) = \frac{1}{\ln x}$ at equispaced nodes in [50,150], separated by step length $h = 10$ or 20.

Exercise 21: The depths D (in meters) of a 80 meters wide river at different horizontal distances s from the bank is given in the following table.

s	0	10	20	30	40	50	60	70	80
D	0	3.5	6	12	10	15	9	5	0

Estimate the area of the cross-section of the river.

Hint for the Solution: Clearly, D is shown to be a function of s and its values $D(s)$ for different points s are given. Obtain the estimate of the integral, $area = \int_0^{80} D(s)\,ds$ using any appropriate numerical integration rule with the data given in the Table.

Exercise 22: A rectangular swimming pool is 35 feet wide and 60 feet long. At different positions P in feet along the length of the pool, the depths D in feet are shown in the following Table. Estimate the volume of the pool using numerical integration.

P	0	6	12	18	24	30	36	42	48	54	60
D	3	3.5	4	4.5	5	5.5	6	6.5	7	7.5	8

Hint for the Solution:

Clearly, D is shown to be a function of P and its values $D(P)$ for different points P are given. Obtain the estimate of the integral $w = \int_0^{60} D(P)\, dP$ using any appropriate numerical integration rule with the data given in the Table. Note that w is the estimated area of one side-wall of the pool along the length. Multiplying it with the width of 35 feet will give the volume of the pool.

Exercise 23: We know that

$$\int_0^1 \frac{1}{1+x^2}\, dx = \tan^{-1} x|_0^1 = \tan^{-1} 1 = \frac{\pi}{4}$$

This means that the value of π can be obtained evaluating the above integral and then multiplying the answer by 4. Suppose that we want to approximate π to four decimal places. This means absolute error must be less than 5.0×10^{-5}. This means that the error in approximating the integral must be less than $\frac{1}{4} \times (5.0 \times 10^{-5}) = 1.25 \times 10^{-5}$. Use the Composite Simpson's 1/3 rule to approximate the value of π. For this, first determine that what should be the minimum number of subintervals that would keep the error less than the tolerance.

Exercise 24: The number of subintervals required to apply the Composite Simpson's 1/3 rule should be

(A) Multiple of 1 (B) Multiple of 2

(C) Multiple of 3 (D) unconditionally many (E) None of above

Exercise 25: The Simpson's 1/3 rule is based on the integration of interpolating polynomial of degree 2. The Simpson's 1/3 rule can accurately integrate the polynomials of degree

(A) up to 1 (B) up to 2

(C) up to 3 (D) up to any (E) None of above

Exercise 26: The Gaussian quadrature is different from the Newton's Cotes Integration in regards to

(A) selection of polynomial degree (B) selection of quadrature nodes

(C) problem dependence (D) None of above

■■■

Chapter **5**

Numerical Differentiation

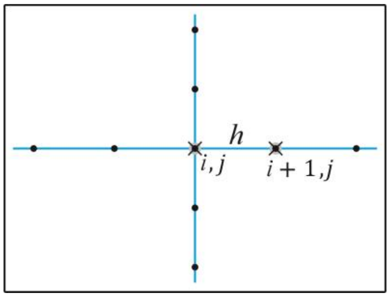
Corridor I: BASICS

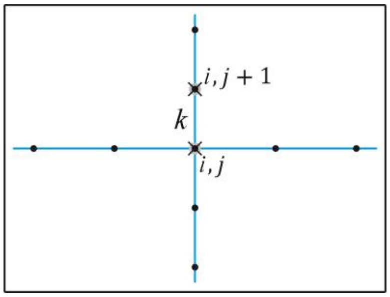
Let's plan it

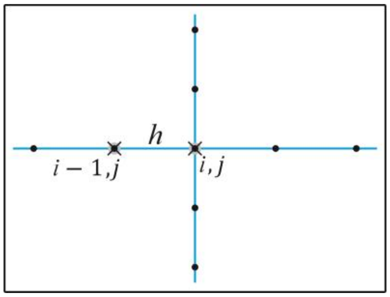
To unleash the topics of this Corridor, please delve into the principal book:

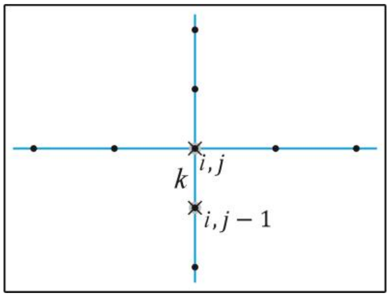
Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

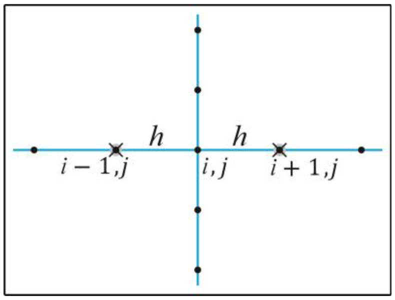
■■■

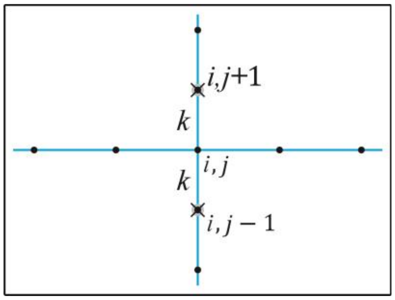


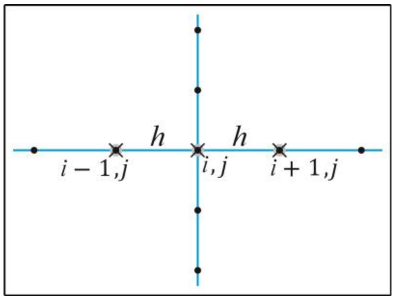


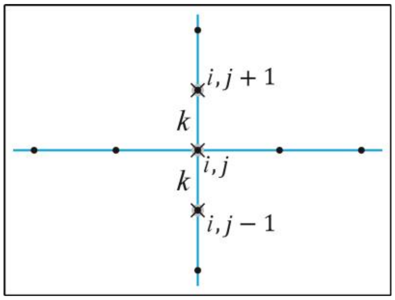

h
i − 1,j
i,j

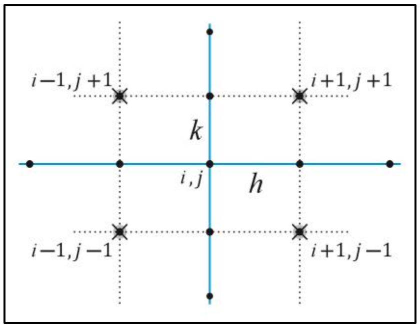

i,j
k
i,j − 1


h
h
i − 1,j
i,j
i + 1,j


i,j+1
k
i,j
k
i,j − 1


h
h
i − 1,j
i,j
i + 1,j


i,j + 1
k
i,j
k
i,j − 1


i−1,j +1
i+1,j +1
k
i,j
h
i−1,j −1
i+1,j −1

Chapter 6

Direct Linear Solvers

Corridor I: BASICS

Let's plan it

To unleash the topics of this Corridor, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

■■■

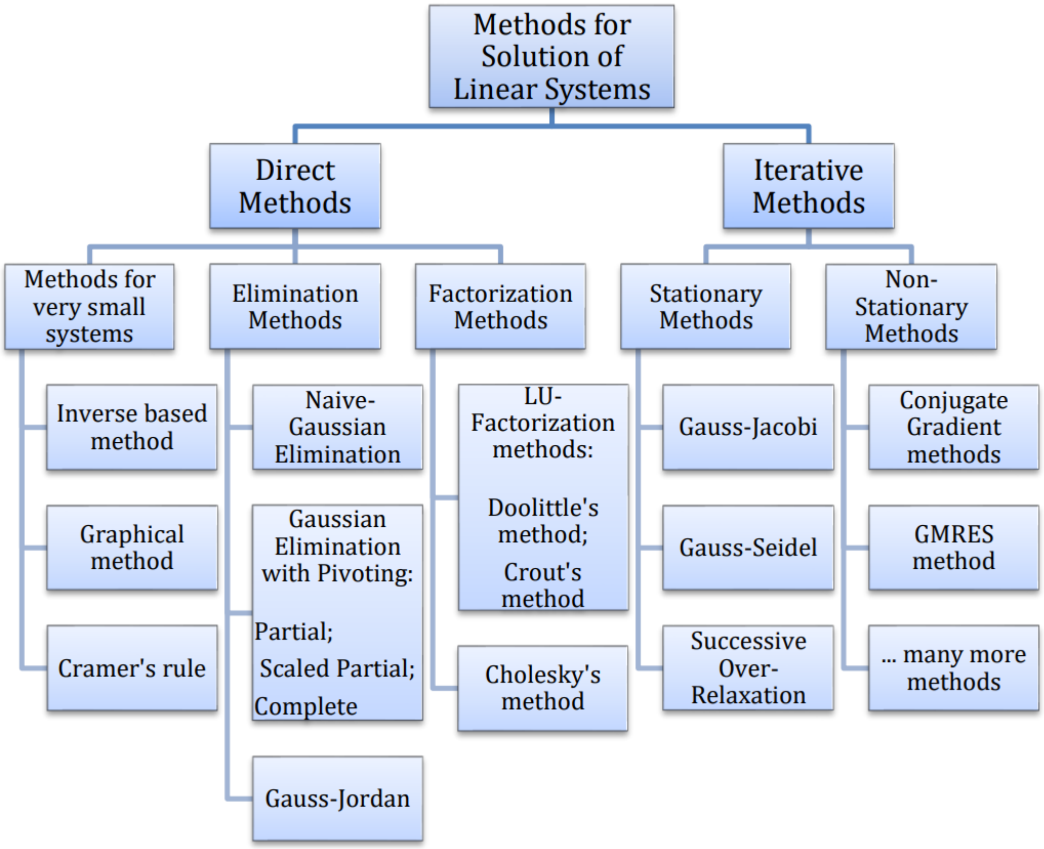

Methods for Solution of Linear Systems
Direct Methods
Iterative Methods
Methods for very small systems
Elimination Methods
Factorization Methods
Stationary Methods
Non-Stationary Methods
Inverse based method
Graphical method
Cramer's rule
Naive-Gaussian Elimination
Gaussian Elimination with Pivoting: Partial; Scaled Partial; Complete
Gauss-Jordan
LU-Factorization methods: Doolittle's method; Crout's method
Cholesky's method
Gauss-Jacobi
Gauss-Seidel
Successive Over-Relaxation
Conjugate Gradient methods
GMRES method
... many more methods

Fig. (6.3): *A classification chart of linear solvers.*

Corridor II: ANALYSIS

Let's think deep

6.6 Operation Count Analysis

6.7 Matrix Inversion

To unleash the topics of this Corridor, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

■■■

Corridor III: PROGRAMMING ARCADE

Let's do it

People have been communicating and interconnecting since the beginning, but in this era the communications and interconnections without modern technologies (like phones, networks, internet, radio, and television) stand nowhere in regards to possibility or survival. Likewise, people have been doing mathematics since early ages, but in this modern era the mathematical applicability without making use of the computers stands nowhere. Let's modernize "OUR" culture of doing mathematics so that it can be useful for all the disciplines of science and engineering. It's time to lead the frontiers of the knowledge and its applicability, rather than

Remark: Suggestion: Before this Section, study, Corridor III of Chapter 07 to cope the difficulty level.

To see more examples for practicing, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

For codes, please visit: https://github.com/DrAmjadAli11/SimplifiedNumericalAnalysis

■■■

6.8 Algorithms and Implementations

Question 20: Write down an algorithm (pseudo code) to solve a linear system $\boldsymbol{AX} = \boldsymbol{B}$ using the Gaussian Elimination method with partial pivoting.

Algorithm: To solve $\boldsymbol{AX} = \boldsymbol{B}$.

INPUTS: $\begin{cases} \boldsymbol{n}\text{: an integer as the number of equations and unknowns} \\ \boldsymbol{A} = (\boldsymbol{a_{ij}}), 1 \le i,j \le \boldsymbol{n}\text{: a real valued square matrix as the coefficient matrix} \\ \boldsymbol{B} = [\boldsymbol{b_1}, \boldsymbol{b_2}, \cdots, \boldsymbol{b_n}]^T\text{: a real valued vector as the vector of right hand side constants} \end{cases}$

OUTPUTS: $\begin{cases} \boldsymbol{X} = [\boldsymbol{x_1}, \boldsymbol{x_2}, \cdots, \boldsymbol{x_n}]^T\text{: a real valued vector as the solution vector} \\ \text{or a message that the given system has no unique solution} \end{cases}$

```
Step 1    Receive the inputs as stated above

Step 2    (Forward Elimination Phase)

          for $i = 1,2,\cdots,\boldsymbol{n-1}$
                Set $r = i$
                for $j = i+1,\cdots,\boldsymbol{n}$
                      if $(|\boldsymbol{a_{ri}}| < |\boldsymbol{a_{ji}}|)$   $r = j$
                      (Searching largest absolute coefficient in $i$th column for partial pivoting)

                if $(\boldsymbol{a_{ri}} = 0)$      OUTPUT ('The given system has no unique solution') and STOP

                else
                      if $(r \ne i)$, then interchange the $ith$ row with $rth$ row, and $b_i$ with $b_r$

             for $k = i+1, i+2, \cdots, \boldsymbol{n}$
                $multiplier = \dfrac{\boldsymbol{a_{ki}}}{\boldsymbol{a_{ii}}}$

                for $j = i+1, i+2, \cdots, \boldsymbol{n}$
                      $\boldsymbol{a_{kj}} = \boldsymbol{a_{kj}} - multiplier \times \boldsymbol{a_{ij}}$

                $\boldsymbol{b_k} = \boldsymbol{b_k} - multiplier \times \boldsymbol{b_i}$
                (row replacement in the augmented matrix for eliminating the coefficients below the pivot)

Step 3    if $(\boldsymbol{a_{nn}} = 0)$      OUTPUT ('The given system has no unique solution') and STOP

          else      go to step 4

Step 4    (Back Substitution Phase)

          $\boldsymbol{x_n} = \dfrac{\boldsymbol{b_n}}{\boldsymbol{a_{nn}}}$
          for $i = \boldsymbol{n} - 1, \cdots, 2, 1$
                $sum = 0$
                for $j = i+1, i+2, \cdots, \boldsymbol{n}$
                      $sum = sum + \boldsymbol{a_{ij}} \times \boldsymbol{x_j}$
                $\boldsymbol{x_i} = \dfrac{[\boldsymbol{b_i} - sum]}{\boldsymbol{a_{ii}}}$
                $\left(x_i = \dfrac{1}{a_{ii}}\left[b_i - \sum_{j=i+1}^{n} a_{ij}x_j\right]\right)$

Step 5    Print the output: $\boldsymbol{X} = [\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}]^T$ and STOP.
```

Question 21: Write a Python program to solve the following linear system using the Gaussian Elimination method with partial pivoting. For simplification, specify the linear system within the program.

$$\begin{array}{rcrcrcr} 1.7x_1 & + & 2.3x_2 & - & 1.5x_3 & = & 2.35 \\ 1.1x_1 & + & 1.6x_2 & - & 1.9x_3 & = & -0.94 \\ 2.7x_1 & - & 2.2x_2 & + & 1.5x_3 & = & 2.70 \end{array}$$

script_6.1: gauss_elimination.ipynb

```
from numpy import *

a = [[1.7, 2.3, −1.5],[1.1, 1.6, −1.9],[2.7, −2.2, 1.5]]
b = [2.35, −0.94, 2.70]
n = 3
t = zeros(n)

# Forward Elimination phase
for i in range(n):
    r = i
    for j in range(i+1,n):
        if abs(a[r][i]) < abs(a[i][j]):
            r = j
    if a[r][i] == 0:
        print("System has no unique solution")
        break
    else:
        if r != i:
            for j in range(n):
                temp = a[i][j]
                a[i][j] = a[r][j]
                a[r][j] = temp
    temp1 = b[i]
    b[i] = b[r]
    b[r] = temp1

    for k in range(i+1,n):
        multiplier = a[k][i]/a[i][i]
        for j in range(i+1,n):
            a[k][j] = a[k][j] - multiplier * a[i][j]
        b[k] = b[k] - multiplier * b[i]

if a[n−1][n−1] == 0:
    print("The system has no unique solution")
else:
    t[n−1] = b[n−1] / a[n−1][n−1]
for i in reversed(range(n−1)):
    sum=0
    for j in range(i+1,n) :
```

```
                sum = sum + a[i][j] * t[j]
t[i] = (b[i] – sum) / a[i][i]

print("The solution of given system is", t)
```

Output Console:

```
The solution of given system is [-0.61111111  0.          2.9        ]
```

Remark: The Python program in Question 21 can be modified to receive the linear system at the execution time (instead of fixing in the code). For this, lines 3 and 4 in the solution of Question 21should be replaced by the following code segment:

```
# Input Section
print("\nEnter the coefficient matrix row-wise for", n, "unknowns:")
a = [[0.0] * n for _ in range(n)]
for i in range(n):
    for j in range(n):
        a[i][j] = float(input())

print("Enter the elements of constant vector B:")
b = [0.0] * n
for i in range(n):
    b[i] = float(input())
```

Question 22: Write down an algorithm (pseudo code) to solve a linear system using the Doolittle's method.

Algorithm: To solve a linear system $\boldsymbol{AX} = \boldsymbol{B}$, for which the factorization $\boldsymbol{A} = \boldsymbol{LU}$ is possible.

INPUTS: $\begin{cases} \boldsymbol{n}\text{: an integer as the number of equations and unknowns} \\ \boldsymbol{A} = (\boldsymbol{a_{ij}}), 1 \le i,j \le \boldsymbol{n}\text{: a real valued square matrix as the coefficient matrix} \\ \boldsymbol{B} = [\boldsymbol{b_1}, \boldsymbol{b_2}, \cdots, \boldsymbol{b_n}]^{\boldsymbol{T}}\text{: a real valued vector as the vector of right hand side constants} \end{cases}$

OUTPUTS: $\begin{cases} \boldsymbol{L} = (\boldsymbol{l_{ij}}), 1 \le i,j \le \boldsymbol{n}\text{: a real valued square matrix as the lower triangular matrix} \\ \boldsymbol{U} = (\boldsymbol{u_{ij}}), 1 \le i,j \le \boldsymbol{n}\text{: a real valued square matrix as the upper triangular matrix} \\ \boldsymbol{X} = [\boldsymbol{x_1}, \boldsymbol{x_2}, \cdots, \boldsymbol{x_n}]^{\boldsymbol{T}}\text{: a real valued vector as the solution vector} \end{cases}$

```
Step 1    (Formation of $\boldsymbol{L}$ and $\boldsymbol{U}$ as factors of $\boldsymbol{A}$, i.e., $\boldsymbol{A} = \boldsymbol{LU}$)

          for $i = 1, 2, \cdots, \boldsymbol{n}$

              Set $\boldsymbol{l_{ii}} = 1$
              For $j = i, i + 1, \cdots, \boldsymbol{n}$
```

```
            $sum = 0$
            for $s = 1, 2, \cdots, i-1$
                $sum = sum + \boldsymbol{l_{is}} \times \boldsymbol{u_{sj}}$
            $\boldsymbol{u_{ij}} = \boldsymbol{a_{ij}} - sum$
```

$$\left(u_{ij} = a_{ij} - \sum_{s=1}^{i-1} l_{is}u_{sj}\right)$$

```
        for $j = i+1, i+2, \cdots, \boldsymbol{n}$
            $sum = 0$
            for $s = 1, 2, \cdots, i-1$
                $sum = sum + \boldsymbol{l_{js}} \times \boldsymbol{u_{si}}$
            $\boldsymbol{l_{ji}} = \frac{[\boldsymbol{a_{ji}} - sum]}{\boldsymbol{u_{ii}}}$
```

$$\left(l_{ji} = \frac{1}{u_{ii}}\left[a_{ji} - \sum_{s=1}^{i-1} l_{js}u_{si}\right]\right)$$

Step 2 (Forward substitution phase for solving $\boldsymbol{LY} = \boldsymbol{B}$)

```
    $\boldsymbol{y_1} = \boldsymbol{b_1}$
    for $i = 2, 3, \cdots, \boldsymbol{n}$
            $sum = 0$
            for $j = 1, 2, \cdots, i-1$
                $sum = sum + \boldsymbol{l_{ij}} \times \boldsymbol{y_j}$
            $\boldsymbol{y_i} = \boldsymbol{b_i} - sum$
```

$$\left(y_i = b_i - \sum_{j=1}^{i-1} l_{ij}y_j\right)$$

Step 3 (Back Substitution Phase for solving $\boldsymbol{UX} = \boldsymbol{Y}$)

```
    $\boldsymbol{x_n} = \frac{\boldsymbol{y_n}}{\boldsymbol{u_{nn}}}$
    for $i = \boldsymbol{n} - 1, \cdots, 2, 1$
            $sum = 0$
            for $j = i+1, i+2, \cdots, \boldsymbol{n}$
                $sum = sum + \boldsymbol{u_{ij}} \times \boldsymbol{x_j}$
            $\boldsymbol{x_i} = \frac{[\boldsymbol{y_i} - sum]}{\boldsymbol{u_{ii}}}$
```

$$\left(x_i = \frac{1}{u_{ii}}\left[y_i - \sum_{j=i+1}^{n} u_{ij}x_j\right]\right)$$

STOP.

Question 23: Write a Python program to solve the following linear system using the Doolittle's method. For simplification, specify the linear system within the program.

$$\begin{aligned} 1.7x_1 + 2.3x_2 - 1.5x_3 &= 2.35 \\ 1.1x_1 + 1.6x_2 - 1.9x_3 &= -0.94 \\ 2.7x_1 - 2.2x_2 + 1.5x_3 &= 2.70 \end{aligned}$$

script_6.2: dolittles.ipynb

```python
from numpy import *

n = 3
a = [[1.7, 2.3, −1.5], [1.1, 1.6, −1.9], [2.7, −2.2, 1.5]]
b = [2.35, −0.94, 2.70]
x = zeros(n)
```

```
y = zeros(n)
l = diag(ones(n))
u = zeros([n,n])

for j in range(n):
    u[0][j] = a[0][j]
    l[j][0] = a[j][0] / u[0][0]

for i in range(1,n):
    l[i][i]=1
    for j in range(i,n):
        sum = 0
        for k in range(i):
            sum = sum + l[i][k] * u[k][j]
        u[i][j] = a[i][j] – sum
    for j in range(i+1,n):
        sum = 0
        for k in range(i):
            sum = sum + l[j][k] * u[k][i]
        l[j][i] = ( a[j][i] – sum ) / u[i][i]

#Forward substitution phase for solving LY=B
y[0] = b[0]
for i in range(n):
    sum = 0
    for j in  range(i):
        sum = sum + l[i][j] * y[j]
    y[i] = b[i] – sum

#Back substitution phase for solving UX=Y
x[n–1] = y[n–1] / u[n–1][n–1]
for i in reversed(range(n–1)):
    sum = 0
    for j in range(i+1,n):
        sum = sum + (u[i][j] * x[j])
    x[i] = ( y[i] – sum ) / u[i][i]

print("The L matrix is:")
for i in range(n):
    for j in range(n):
        print(l[i][j], "            ",end=" ")
    print(" ")

print("The U matrix is:")
for i in range(n):
    for j in range(n):
        print(u[i][j], "        ",end=" ")
    print(" ")

print("The required solution is:")
for i in range(n):
    print(x[i], "        ",end=" ")
```

Output Console:

```
The L matrix is:
1.0              0.0               0.0
0.6470588235294118             1.0               0.0
1.5882352941176472             -52.36842105263156              1.0

The U matrix is:
1.7          2.3           -1.5
0.0          0.11176470588235299           -0.9294117647058822

0.0          0.0           -44.7894736842105
The required solution is:
1.1000000000000056           2.0999999999999965           2.90000000000000
```

■

Remark: Replace the lines 4 and 5 in the solution of Question 23 with the following code segment to receive the linear system at the execution time (instead of fixing in the code):

```
# Input Section
print("\nEnter the coefficient matrix row-wise for", n, "unknowns:")
a = []
for i in range(n):
   row = []
   for j in range(n):
      row.append(float(input()))
   a.append(row)

print("Enter the elements of the constant vector B:")
b = []
for i in range(n):
   b.append(float(input()))
```

■

Question 24: Write down an algorithm (pseudo code) to solve a linear system using the Crout's method.

Algorithm: To solve a linear system $\boldsymbol{AX} = \boldsymbol{B}$, for which the factorization $\boldsymbol{A} = \boldsymbol{LU}$ is possible.

INPUTS: $\begin{cases} \boldsymbol{n}\text{: an integer as the number of equations and unknowns} \\ \boldsymbol{A} = (\boldsymbol{a_{ij}}), 1 \le i, j \le \boldsymbol{n}\text{: a real valued square matrix as the coefficient matrix} \\ \boldsymbol{B} = [\boldsymbol{b_1}, \boldsymbol{b_2}, \cdots, \boldsymbol{b_n}]^{\boldsymbol{T}}\text{: a real valued vector as the vector of right hand side constants} \end{cases}$

OUTPUTS: $\begin{cases} \boldsymbol{L} = (\boldsymbol{l_{ij}}), 1 \le i, j \le \boldsymbol{n}\text{: a real valued square matrix as the lower triangular matrix} \\ \boldsymbol{U} = (\boldsymbol{u_{ij}}), 1 \le i, j \le \boldsymbol{n}\text{: a real valued square matrix as the upper triangular matrix} \\ \boldsymbol{X} = [\boldsymbol{x_1}, \boldsymbol{x_2}, \cdots, \boldsymbol{x_n}]^{\boldsymbol{T}}\text{: a real valued vector as the solution vector} \end{cases}$

<u>**Step 1**</u> (Formation of $\boldsymbol{L}$ and $\boldsymbol{U}$ as factors of $\boldsymbol{A}$, i.e., $\boldsymbol{A} = \boldsymbol{LU}$)

```
for $i = 1, 2, \cdots, \boldsymbol{n}$
    Set $\boldsymbol{u_{ii}} = 1$
    for $j = i, i + 1, \cdots, \boldsymbol{n}$
        $sum = 0$
        for $s = 1, 2, \cdots, i - 1$
            $sum = sum + \boldsymbol{l_{js}} \times \boldsymbol{u_{si}}$
        $\boldsymbol{l_{ji}} = \boldsymbol{a_{ji}} - sum$
    for $j = i + 1, i + 2, \cdots, \boldsymbol{n}$
        $sum = 0$
        for $s = 1, 2, \cdots, i - 1$
            $sum = sum + \boldsymbol{l_{is}} \times \boldsymbol{u_{sj}}$
        $\boldsymbol{u_{ij}} = \frac{[\boldsymbol{a_{ij}} - sum]}{\boldsymbol{l_{ii}}}$
```

$$\left(l_{ji} = a_{ji} - \sum_{s=1}^{i-1} l_{js} u_{si}\right)$$

$$\left(u_{ij} = \frac{1}{l_{ii}}\left[a_{ij} - \sum_{s=1}^{i-1} l_{is} u_{sj}\right]\right)$$

<u>**Step 2**</u> (Forward Substitution Phase for solving $\boldsymbol{LY} = \boldsymbol{B}$)

```
$\boldsymbol{y_1} = \frac{\boldsymbol{b_1}}{\boldsymbol{l_{11}}}$
for $i = 2, 3, \cdots, \boldsymbol{n}$
        $sum = 0$
        for $j = 1, 2, \cdots, i - 1$
            $sum = sum + \boldsymbol{l_{ij}} \times \boldsymbol{y_j}$
        $\boldsymbol{y_i} = \frac{[\boldsymbol{b_i} - sum]}{\boldsymbol{l_{ii}}}$
```

$$\left(y_i = \frac{1}{l_{ii}}\left[b_i - \sum_{j=1}^{i-1} l_{ij} y_j\right]\right)$$

<u>**Step 3**</u> (Back Substitution Phase for solving $\boldsymbol{UX} = \boldsymbol{Y}$)

```
$\boldsymbol{x_n} = \boldsymbol{y_n}$
for $i = \boldsymbol{n} - 1, \cdots, 2, 1$
        $sum = 0$
        for $j = i + 1, i + 2, \cdots, \boldsymbol{n}$
            $sum = sum + \boldsymbol{u_{ij}} \times \boldsymbol{x_j}$
        $\boldsymbol{x_i} = \boldsymbol{y_i} - sum$
```

$$\left(x_i = y_i - \sum_{j=i+1}^{n} u_{ij} x_j\right)$$

STOP.

Question 25: Write a Python program to solve the following linear system using the Crout's method. For simplification, specify the linear system within the program.

$$\begin{array}{rcrcrcr} 1.7x_1 & + & 2.3x_2 & - & 1.5x_3 & = & 2.35 \\ 1.1x_1 & + & 1.6x_2 & - & 1.9x_3 & = & -0.94 \\ 2.7x_1 & - & 2.2x_2 & + & 1.5x_3 & = & 2.70 \end{array}$$

script_6.3: crouts.ipynb

```
from numpy import *

n = 3
a = [[1.7, 2.3, −1.5], [1.1, 1.6, −1.9], [2.7, −2.2, 1.5]]
b = [2.35, −0.94, 2.70]
x = zeros(n)
y = zeros(n)
l = zeros([n,n])
u = diag(ones(n))

for j in range(n):                              #Crouts Method
        l[j][0] = a[j][0]
        u[0][j] = a[0][j] / l[0][0]

for i in range(1,n):
        u[i][i] = 1
        for j in range(i,n):
                sum = 0
                for k in range(i):
                        sum = sum + l[i][k] * u[k][j]
                l[j][i] = a[j][i] – sum
        for j in range(i+1,n):
                sum = 0
                for k in range(i):
                        sum = sum + l[i][k] * u[k][j]
                u[i][j] = ( a[i][j] – sum ) / l[i][i]

#Forward substitution phase for solving LY=B
y[0] = b[0] / l[0][0]
for i in range(n):
        sum = 0
        for j in  range(i):
                sum = sum + l[i][j] * y[j]
        y[i] = (b[i] – sum) / l[i][i]

#Back substitution phase for solving UX=Y
x[n−1] = y[n−1]
```

```
for i in reversed(range(n-1)):
        sum = 0
        for j in range(i+1,n):
                sum = sum + (u[i][j] * x[j])
        x[i] = ( y[i] - sum )

print("The L matrix is:")
for i in range(n):
        for j in range(n):
                print(l[i][j], "            ",end=" ")
        print(" ")

print("The U matrix is:")
for i in range(n):
        for j in range(n):
                print(u[i][j], "         ",end=" ")
        print(" ")

print("The required solution is:")
for i in range(n):
        print(x[i], "         ",end=" ")
```

Output Console:

```
The L matrix is:
1.7             0.0             0.0
1.1             0.11176470588235299             0.0
2.7             -1.2294117647058824             -6.341176470588229

The U matrix is:
1.0          1.352941176470588           -0.8823529411764706
0.0          1.0            -8.315789473684205
0.0          0.0            1.0
The required solution is:
-14.775803144224206          14.832877648667132          4.4311688311688
35
```

■

Remark: Replace the lines 4 and 5 in the solution of Question 25with the following code segment to receive the linear system at the execution time (instead of fixing in the code):

```
# Input Section
print("\nEnter the coefficient matrix row-wise for", n, "unknowns:")
a = []
for i in range(n):
    row = []
    for j in range(n):
        row.append(float(input()))
    a.append(row)

print("Enter the elements of the constant vector B:")
b = []
for i in range(n):
    b.append(float(input()))
```

■

Question 26: Write down an algorithm (pseudo code) to solve a linear system using the Cholesky's method.

Algorithm: To solve a linear system $\boldsymbol{AX} = \boldsymbol{B}$, for which the factorization $\boldsymbol{A} = \boldsymbol{LL^T}$ is possible.

INPUTS: $\begin{cases} \boldsymbol{n}\text{: an integer as the number of equations and unknowns} \\ \boldsymbol{A} = (\boldsymbol{a_{ij}}), 1 \le i,j \le \boldsymbol{n}\text{: a real valued square matrix as the coefficient matrix} \\ \boldsymbol{B} = [\boldsymbol{b_1}, \boldsymbol{b_2}, \cdots, \boldsymbol{b_n}]^{\boldsymbol{T}}\text{: a real valued vector as the vector of right hand side constants} \end{cases}$

OUTPUTS: $\begin{cases} \boldsymbol{L} = (\boldsymbol{l_{ij}}), 1 \le i,j \le \boldsymbol{n}\text{: a real valued square matrix as the lower triangular matrix} \\ \boldsymbol{X} = [\boldsymbol{x_1}, \boldsymbol{x_2}, \cdots, \boldsymbol{x_n}]^{\boldsymbol{T}}\text{: a real valued vector as the solution vector} \end{cases}$

Step 1 (Formation of $\boldsymbol{L}$ as factors of $\boldsymbol{A}$, i.e., $\boldsymbol{A} = \boldsymbol{LL^T}$)

```
for $i = 1, 2, \cdots, \boldsymbol{n}$
        $sum = 0$
        for $k = 1, 2, \cdots, i-1$
                $sum = sum + \boldsymbol{l_{ik}} \times \boldsymbol{l_{ik}}$
        $\boldsymbol{l_{ii}} = \boldsymbol{sqrt}(\boldsymbol{a_{ii}} - sum)$
```

$$\left(l_{ii} = \left[a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2\right]^{\frac{1}{2}}\right)$$

```
        for $j = i+1, i+2, \cdots, \boldsymbol{n}$
            $sum = 0$
            for $k = 1, 2, \cdots, i-1$
                    $sum = sum + \boldsymbol{l_{ik}} \times \boldsymbol{l_{jk}}$
            $\boldsymbol{l_{ji}} = \frac{[\boldsymbol{a_{ji}} - sum]}{\boldsymbol{l_{ii}}}$
```

$$\left(l_{ji} = \frac{1}{l_{ii}}\left[a_{ji} - \sum_{k=1}^{i-1} l_{ik} l_{jk}\right]\right)$$

Step 2 (Forward Substitution Phase for solving $\boldsymbol{LY} = \boldsymbol{B}$)

```
$\boldsymbol{y_1} = \frac{\boldsymbol{b_1}}{\boldsymbol{l_{11}}}$
for $i = 2, 3, \cdots, \boldsymbol{n}$
        $sum = 0$
        for $j = 1, 2, \cdots, i-1$
                $sum = sum + \boldsymbol{l_{ij}} \times \boldsymbol{y_j}$
        $\boldsymbol{y_i} = \frac{[\boldsymbol{b_i} - sum]}{\boldsymbol{l_{ii}}}$
```

$$\left(y_i = \frac{1}{l_{ii}}\left[b_i - \sum_{j=1}^{i-1} l_{ij} y_j\right]\right)$$

Step 3 (Back Substitution Phase for solving $\boldsymbol{L^T X = Y}$)

```
$x_n = \frac{y_n}{l_{nn}}$
for $i = n-1, \cdots, 2, 1$
        $sum = 0$
        for $j = i+1, i+2, \cdots, n$
                $sum = sum + l_{ji} \times x_j$
        $x_i = \frac{[y_i - sum]}{l_{ii}}$
```

$$\left(x_i = \frac{1}{l_{ii}}\left[y_i - \sum_{j=i+1}^{n} l_{ji}x_j\right]\right)$$

STOP.

Question 27: Write a Python program to solve the following positive definite linear system using the Cholesky's method. For simplification, specify the linear system within the program.

$$\begin{aligned} 0.4x_1 \qquad\qquad\quad + 0.12x_3 &= 1.4 \\ 0.64x_2 + 0.32x_3 &= 1.6 \\ -0.12x_1 + 0.32x_2 + 0.56x_3 &= 5.4 \end{aligned}$$

script_6.4: cholesky.ipynb

```
from numpy import *

n = 3
a = [[0.4, 0.0, 0.12], [0.0, 0.64, 0.32],[-0.12, 0.32, 0.56]]
b = [1.4, 1.6, 5.4]
x = zeros(n)
y = zeros(n)
l = zeros([n,n])
l[0][0] = sqrt(a[0][0])

for j in range(n):                          #Cholesky Method
        l[j][0] = a[j][0] / l[0][0]

for i in range(1,n):
        sum=0
        for k in range(i):
                sum = sum + l[i][k] * l[i][k]
        l[i][i] = a[i][i] – sum
        l[i][i] = sqrt(l[i][i])
        for j in range(i+1,n):
                sum = 0
                for k in range(i):
                        sum = sum + l[i][k] * l[j][k]
                l[j][i] = ( a[j][i] – sum ) / l[i][i]

```

```
#Forward substitution phase for solving LY=B
y[0] = b[0] / l[0][0]
for i in range(n):
        sum = 0
        for j in  range(i):
                sum = sum + l[i][j] * y[j]
        y[i] = (b[i] – sum)/l[i][i]

#Back substitution phase for solving L^t X = Y
x[n−1] = y[n−1] / l[n−1][n−1]
for i in reversed(range(n−1)):
        sum = 0
        for j in range(i+1,n):
                sum = sum + (l[j][i] * x[j])
        x[i] = ( y[i] – sum )/l[i][i]

print("The L matrix is:")
for i in range(n):
        for j in range(n):
                print(l[i][j], "           ",end=" ")
        print(" ")

print("The required solution is:")
for i in range(n):
        print(x[i], "        ",end=" ")
```

Output Console:

```
The L matrix is:
0.6324555320336759            0.0              0.0
0.0              0.8              0.0
-0.18973665961010275           0.39999999999999997            0.60332
41251599343
The required solution is:
7.637362637362637          -4.395604395604395          13.79120879120879
2
```

■

Remark: Following are some notations and formulas that might be useful in carrying out operation count analysis of the algorithms.

$$\begin{aligned}
\sum_{p=1}^{n} cf(p) &= c\sum_{p=1}^{n} f(p) \\
\sum_{p=1}^{n}[f(p)+g(p)] &= \sum_{p=1}^{n} f(p)+\sum_{p=1}^{n} g(p) \\
\sum_{p=1}^{n} 1 &= 1+1+\cdots+1 \quad = \quad n \\
\sum_{p=k}^{n} 1 &= n-k+1 \\
\sum_{p=1}^{n} p &= 1+2+3+\cdots+n \quad = \quad \frac{n(n+1)}{2} \quad = \quad \frac{n^2}{2}+\mathcal{O}(n) \\
\sum_{p=1}^{n} p^2 &= 1^2+2^2+\cdots+n^2 \quad = \quad \frac{n(n+1)(2n+1)}{6} \quad = \quad \frac{n^3}{3}+\mathcal{O}(n^2)
\end{aligned}$$

Question 28: Perform the operation count analysis of the algorithm that involves the following phases to solve an $n \times n$ linear system:

(1) Forward elimination to obtain the upper triangular form using the Gauss Elimination method.
(2) Back substitution to solve the upper triangular system.

(1) The forward elimination phase occurs just after setting the inputs in the algorithm. This phase contains three nested loops. The first loop, say i-loop (which ranges from $i = 1$ to $n-1$), corresponds to the $n-1$ elimination stages of the method. For each row i, the ith element is considered a pivot element. The second loop, say k-loop (which ranges from $k = i+1$ to n), corresponds to the elements below the pivot element to make them zero. The third loop, say j-loop (which ranges from $j = i+1$ to n) corresponds to the columns after the pivot element.

Note that, for any loop with index ranging from $i+1$ to n, the number of passes/iterations will be $n-(i+1)+1$ (or simply $(n-i)$ passes). Therefore, each of the k-loop and j-loop has $(n-i)$ passes.

Each pass of k-loop will perform one division to obtain the multiplier, and one multiplication and subtraction to update the right-hand side constant, b_k. Moreover, in each pass of k-loop, $(n-i)$ multiplications and $(n-i)$ subtractions will be performed in j-loop to update the relevant entries of the coefficient matrix, a_{kj}. Thus, in each pass of k-loop, the total number of multiplications/divisions will be $\big(1+1+(n-i)\big)$ or $(n-i+2)$ and the total number of additions/subtractions will be $(1+n-i)$.

As there are $(n-i)$ passes of k-loop in each pass of i-loop, therefore there will be $(n-i)\times(n-i+2)$ multiplications/divisions and $(n-i)\times(n-i+1)$ additions/subtractions in each pass of i-loop.

Hence, the total number of multiplications/divisions in $n-1$ passes of i-loop of the forward elimination phase will be

$$
\begin{aligned}
\sum_{i=1}^{n-1}(n-i)(n-i+2) &= \sum_{i=1}^{n-1}(n-i)\big((n+2)-i\big) \\
&= \sum_{i=1}^{n-1}[n(n+2)-ni-i(n+2)+i^2] = \sum_{i=1}^{n-1}[n(n+2)-2i(n+1)+i^2] \\
&= n(n+2)\sum_{i=1}^{n-1}1-2(n+1)\sum_{i=1}^{n-1}i+\sum_{i=1}^{n-1}i^2 \\
&= n(n+2)(n-1)-2(n+1)\left[\frac{(n-1)n}{2}\right]+\left[\frac{(n-1)n(2n-1)}{6}\right] \\
&= n(n-1)\left[n+2-n-1+\frac{n}{3}-\frac{1}{6}\right] \\
&= (n^2-n)\left[\frac{n}{3}+\frac{5}{6}\right] = \frac{n^3}{3}+\frac{n^2}{2}-\frac{5n}{6} = \frac{n^3}{3}+\mathcal{O}(n^2)
\end{aligned}
$$

Similarly, the total number of additions/subtractions in $n-1$ passes of i-loop of the forward elimination phase will be

$$
\begin{aligned}
\sum_{i=1}^{n-1}(n-i)(n-i+1) &= \sum_{i=1}^{n-1}(n-i)\big((n+1)-i\big) \\
&= \sum_{i=1}^{n-1}[n(n+1)-ni-i(n+1)+i^2] \\
&= \sum_{i=1}^{n-1}[n(n+1)-i(2n+1)+i^2] \\
&= n(n+1)\sum_{i=1}^{n-1}1-(2n+1)\sum_{i=1}^{n-1}i+\sum_{i=1}^{n-1}i^2 \\
&= n(n+1)(n-1)-(2n+1)\left[\frac{(n-1)n}{2}\right]+\left[\frac{(n-1)n(2n-1)}{6}\right] \\
&= n(n-1)\left[n+1-n-\frac{1}{2}+\frac{n}{3}-\frac{1}{6}\right] \\
&= (n^2-n)\left[\frac{n}{3}+\frac{1}{3}\right] = \frac{n^3}{3}-\frac{n}{3} = \frac{n^3}{3}+\mathcal{O}(n)
\end{aligned}
$$

The summary of the operation count of the Gaussian Elimination phase is given as:

	Operations		Total flops
	Multiplications/divisions	Additions/subtractions	
Forward Elimination	$\frac{n^3}{3}+\frac{n^2}{2}-\frac{5n}{6}$	$\frac{n^3}{3}-\frac{n}{3}$	$\frac{2n^3}{3}+\mathcal{O}(n^2)$

(2) The back substitution phase occurs after the forward elimination phase. This phase contains two nested loops. The first loop, say i-loop (which ranges from $i = n-1$ to 1), corresponds to $n-1$ of the components of the solution vector. The second loop, say j-loop (which ranges from $j = i+1$ to n), corresponds to the columns after the diagonal elements.

Each pass of i-loop will perform one subtraction and one division to obtain the value of x_i. Moreover, in each pass of i-loop, the number of both of the multiplications and additions will be $n-(i+1)+1$ (or simply $(n-i)$) in j-loop. Thus, in each pass of i-loop, the total number of both of the multiplications/divisions and additions/subtractions will be $(n-i+1)$.

Hence, the total number of the multiplications/divisions in the back substitution phase will be

$$\begin{aligned}1+\sum_{i=1}^{n-1}(n+1-i) &= 1+(n+1)\sum_{i=1}^{n-1}1-\sum_{i=1}^{n-1}i = 1+(n+1)(n-1)-\left[\frac{(n-1)n}{2}\right]\\ &= 1+n^2-1-\frac{n^2}{2}+\frac{n}{2} = \frac{n^2}{2}+\frac{n}{2} = \frac{n^2}{2}+\mathcal{O}(n)\end{aligned}$$

Similarly, the total number of the additions/subtractions in the back substation phase will be

$$\sum_{i=1}^{n-1}(n+1-i) = \frac{n^2}{2}+\frac{n}{2}-1 = \frac{n^2}{2}+\mathcal{O}(n)$$

Finally, the summary of the operation count of the complete algorithm (including the two phases) is given as:

	Operations		Total flops
	Multiplications/divisions	Additions/subtractions	
Forward elimination	$\frac{n^3}{3}+\frac{n^2}{2}-\frac{5n}{6}$	$\frac{n^3}{3}-\frac{n}{3}$	$\frac{2n^3}{3}+\frac{n^2}{2}-\frac{7n}{6}$
Back Substitution	$\frac{n^2}{2}+\frac{n}{2}$	$\frac{n^2}{2}-\frac{n}{2}$	n^2
Totals	$\frac{n^3}{3}+n^2-\frac{n}{3}$	$\frac{n^3}{3}+\frac{n^2}{2}-\frac{5n}{6}$	$\frac{2n^3}{3}+\frac{3n^2}{2}-\frac{7n}{6}$

Question 29: Perform the operation count analysis of the algorithm that involves the following phases to solve an $n \times n$ linear system:

(1) Factorization of the coefficient matrix using the Doolittle's method.
(2) Forward substitution to solve the lower triangular system.
(3) Back substitution to solve the upper triangular system.

(1) The factorization of the coefficient matrix A into the product of the unit lower triangular L and the upper triangular U matrices occurs just after setting the inputs in the algorithm. The formulation of L and U as the factors of A contains three nested loops. The first loop, say i-loop (which ranges from $i = 1$ to n), corresponds to the ith row and column of U and L respectively. The second loop, say j-loop (ranges from $j = i$ to n), corresponds to the column j of U and (ranges from $j = i + 1$ to n), corresponds to the row j of L. The third loop, say s-loop (which ranges from $s = 1$ to $i - 1$), corresponds to the multiplication of the rows of L and columns of U.

Note that, the j-loop corresponding to column j of U ranging from i to n, the number of passes/iterations will be $n - i + 1$. Similarly, the number of passes/iterations in j-loop, corresponds to row j of L ranging from $i + 1$ to n, will be $n - (i + 1) + 1$ (or simply $(n - i)$).

Each pass of j-loop will perform one subtraction to obtain the entry u_{ij} of U. Moreover, in each pass of j-loop, the number of both of the multiplications and additions will be $(i - 1) - 1 + 1$ (or simply $(i - 1)$) in s-loop. Thus, in each pass of j-loop, the total number of multiplications/divisions will be $(i - 1)$ and the total number of additions/subtractions will be $(1 + i - 1)$ or (i). Similarly, in each pass of j-loop, to obtain the entry l_{ji} of L, the total number of both of the multiplications and additions will be $(i - 1) + 1$ (or simply (i)).

As there are $(n - i + 1)$ passes of j-loop in each pass of i-loop, therefore there will be $(n - i + 1) \times (i - 1)$ multiplications/divisions and $(n - i + 1) \times (i)$ additions/subtractions in each pass of i-loop for the formulation of row i of U. Similarly, in each pass of i-loop, there will be $(n - i) \times (i)$ multiplications/divisions and $(n - i) \times (i)$ additions/subtractions in each pass of i-loop for the formulation of column i of L.

Hence, the total number of multiplications/divisions in n passes of i-loop for the formulation of upper triangular matric U will be

$$\sum_{i=1}^{n}(n - i + 1)(i - 1) = \sum_{i=1}^{n}(n + 1 - i)(i - 1)$$

$$= \sum_{i=1}^{n}[(n + 1)i - (n + 1) - i^2 + i] = \sum_{i=1}^{n}[(n + 2)i - i^2 - (n + 1)]$$

$$
\begin{aligned}
&= (n+2)\sum_{i=1}^{n} i - \sum_{i=1}^{n} i^2 - (n+1)\sum_{i=1}^{n} 1 \\
&= (n+2)\left[\frac{n(n+1)}{2}\right] - \left[\frac{n(n+1)(2n+1)}{6}\right] - (n+1)n \\
&= n(n+1)\left[\frac{n}{2} + 1 - \frac{n}{3} - \frac{1}{6} - 1\right] = (n^2+n)\left[\frac{n}{6} - \frac{1}{6}\right] \\
&= \frac{n^3}{6} - \frac{n}{6} = \frac{n^3}{6} + \mathcal{O}(n)
\end{aligned}
$$

Similarly, the total number of additions/subtractions in n passes of i-loop for the formulation of upper triangular matric U will be

$$
\sum_{i=1}^{n}(n-i+1)(i) = \frac{n^3}{6} + \frac{n}{6} = \frac{n^3}{6} + \mathcal{O}(n)
$$

Moreover, the total number of multiplications/divisions and additions/subtractions in n passes of i-loop for the formulation of unit lower triangular matric L will be

$$
\sum_{i=1}^{n}(n-i)(i) = \sum_{i=1}^{n}(n-i)(i) = \sum_{i=1}^{n}(ni - i^2)
$$

$$
\begin{aligned}
&= n\sum_{i=1}^{n} i - \sum_{i=1}^{n} i^2 = n\left[\frac{n(n+1)}{2}\right] - \left[\frac{n(n+1)(2n+1)}{6}\right] \\
&= n(n+1)\left[\frac{n}{2} - \frac{n}{3} - \frac{1}{6}\right] = (n^2+n)\left[\frac{n}{6} - \frac{1}{6}\right] \\
&= \frac{n^3}{6} - \frac{n}{6} = \frac{n^3}{6} + \mathcal{O}(n)
\end{aligned}
$$

The summary of the operation count of the LU-factorization is given as:

	Operations		Total flops
	Multiplication/Division	Addition/Subtraction	
Upper Triangular Matrix U	$\frac{n^3}{6} - \frac{n}{6}$	$\frac{n^3}{6} + \frac{n}{6}$	$\frac{n^3}{3}$
Lower Triangular Matrix L	$\frac{n^3}{6} - \frac{n}{6}$	$\frac{n^3}{6} - \frac{n}{6}$	$\frac{n^3}{3} - \frac{n}{3}$

LU-factorization	$\frac{n^3}{3}-\frac{n}{3}$	$\frac{n^3}{3}$	$\frac{2n^3}{3}-\frac{n}{3}$

(2) The forward substitution phase occurs after the formulation of L and U as factors of the coefficient matrix for solving the lower triangular system. This phase contains two nested loops. The first loop, say i-loop (which ranges from $i = 2$ to n), corresponds to $n-1$ of the components of the intermediate vector Y. The second loop, say j-loop (which ranges from $j = 1$ to $i-1$), corresponds to the columns before the diagonal elements.

Each pass of i-loop will perform one subtraction to obtain the value of y_i. Moreover, in each pass of i-loop, the number of both of the multiplications and additions will be $(i-1)-1+1$ (or simply $(i-1)$) in j-loop. Thus, in each pass of i-loop, the total number of multiplications/divisions will be $(i-1)$ and the total number of additions/subtractions will be $(1+i-1)$ or (i).

Hence, the total number of the multiplications/divisions in the forward substitution phase will be

$$\sum_{i=2}^{n}(i-1) = \sum_{i=2}^{n} i - \sum_{i=2}^{n} 1 = \left[\frac{n(n+1)}{2}-1\right]-(n-2+1)$$

$$= \frac{n^2}{2}+\frac{n}{2}-1-n+1 = \frac{n^2}{2}-\frac{n}{2} = \frac{n^2}{2}+\mathcal{O}(n)$$

Similarly, the total number of the additions/subtractions in the forward substation phase will be

$$\sum_{i=2}^{n}(i) = \frac{n^2}{2}+\frac{n}{2}-1 = \frac{n^2}{2}+\mathcal{O}(n)$$

The summary of the operation count of the Unit Lower triangular system $LY = B$ is given as:

	Operations		Total flops
	Multiplication/Division	Addition/Subtraction	
Unit Lower triangular system $LY = B$	$\frac{n^2}{2}-\frac{n}{2}$	$\frac{n^2}{2}+\frac{n}{2}-1$	n^2-1

(3) The back substitution phase occurs after the solution of the lower triangular system. This phase contains two nested loops. The first loop, say i-loop (which ranges from $i = n-1$ to 1), corresponds to $n-1$ of the components of solution vector X. The second loop, say j-loop (which ranges from $j = i+1$ to n), corresponds to the columns after the diagonal elements.

Each pass of i-loop will perform one subtraction and one division to obtain the value of x_i. Moreover, in each pass of i-loop, the number of both of the multiplications and additions will be $n-(i+1)+1$ (or simply $(n-i)$) in j-loop. Thus, in each pass of i-loop, the total number of both of the multiplications/divisions and additions/subtractions will be $(n-i+1)$.

Hence, the total number of the multiplications/divisions in the back substitution phase will be

$$\begin{aligned} 1+\sum_{i=1}^{n-1}(n+1-i) &= 1+(n+1)\sum_{i=1}^{n-1}1-\sum_{i=1}^{n-1}i \\ &= 1+(n+1)(n-1)-\left[\frac{(n-1)n}{2}\right] \\ &= 1+n^2-1-\frac{n^2}{2}+\frac{n}{2} = \frac{n^2}{2}+\frac{n}{2} = \frac{n^2}{2}+\mathcal{O}(n) \end{aligned}$$

Similarly, the total number of the additions/subtractions in the back substation phase will be

$$\sum_{i=1}^{n-1}(n+1-i) = \frac{n^2}{2}+\frac{n}{2}-1 = \frac{n^2}{2}+\mathcal{O}(n)$$

The summary of the operation count of the Upper triangular system $UX=Y$ is given as:

	Operations		Total flops
	Multiplications/Divisions	Additions/Subtractions	
Upper triangular system $UX=Y$	$\frac{n^2}{2}+\frac{n}{2}$	$\frac{n^2}{2}+\frac{n}{2}-1$	n^2+n-1

Question 30: Perform the operation count analysis of the algorithm that involves the following phases to solve an $n\times n$ linear system:

(1) Factorization of the coefficient matrix using the Doolittle's method
(2) Forward substitution to solve the lower triangular system.
(3) Back substitution to solve the upper triangular system.

(1) The factorization of the coefficient matrix A into the product of the lower triangular L and the unit upper triangular U matrices occur just after setting the inputs in the algorithm. The formulation of L and U as the factors of A contains three nested loops. The first loop, say i-loop (which ranges from $i=1$ to n), corresponds to the ith column of L and ith row of U respectively. The second loop, say j-loop (ranges from $j=i$ to n), corresponds to the jth row of L and (ranges from $j=i+1$ to n), corresponds to the jth column of U. The third

loop, say s-loop (which ranges from $s = 1$ to $i-1$), corresponds to the multiplication of the rows of L and columns of U.

Note that, the j-loop corresponding to row j of L ranging from i to n, the number of passes/iterations will be $n - i + 1$. Similarly, the number of passes/iterations in j-loop, corresponds to column j of U ranging from $i + 1$ to n, will be $n - (i+1) + 1$ (or simply $(n-i)$).

Each pass of j-loop will perform one subtraction to obtain the entry l_{ji} of L. Moreover, in each pass of j-loop, the number of both of the multiplications and additions will be $(i-1) - 1 + 1$ (or simply $(i-1)$) in s-loop. Thus, in each pass of j-loop, the total number of multiplications/divisions will be $(i-1)$ and the total number of additions/subtractions will be $(1 + i - 1)$ or (i). Similarly, in each pass of j-loop, to obtain the entry u_{ij} of U, the total number of both of the multiplications and additions will be $(i-1)+1$ (or simply (i)).

As there are $(n-i+1)$ passes of j-loop in each pass of i-loop, therefore there will be $(n-i+1)\times(i-1)$ multiplications/divisions and $(n-i+1)\times(i)$ additions/subtractions in each pass of i-loop for the formulation of column i of L. Similarly, in each pass of i-loop, there will be $(n-i)\times(i)$ multiplications/divisions and $(n-i)\times(i)$ additions/subtractions in each pass of i-loop for the formulation of row i of U.

Hence, the total number of multiplications/divisions in n passes of i-loop for the formulation of the lower triangular matric L will be

$$\begin{aligned}
\sum_{i=1}^{n}(n-i+1)(i-1) &= \sum_{i=1}^{n}(n+1-i)(i-1) \\
&= \sum_{i=1}^{n}[(n+1)i-(n+1)-i^2+i] = \sum_{i=1}^{n}[(n+2)i-i^2-(n+1)] \\
&= (n+2)\sum_{i=1}^{n} i - \sum_{i=1}^{n} i^2 - (n+1)\sum_{i=1}^{n} 1 \\
&= (n+2)\left[\frac{n(n+1)}{2}\right] - \left[\frac{n(n+1)(2n+1)}{6}\right] - (n+1)n \\
&= n(n+1)\left[\frac{n}{2}+1-\frac{n}{3}-\frac{1}{6}-1\right] = (n^2+n)\left[\frac{n}{6}-\frac{1}{6}\right] \\
&= \frac{n^3}{6}-\frac{n}{6} = \frac{n^3}{6}+\mathcal{O}(n)
\end{aligned}$$

Similarly, the total number of additions/subtractions in n passes of i-loop for the formulation of the lower triangular matric L will be

$$\sum_{i=1}^{n}(n-i+1)(i) \quad = \quad \frac{n^3}{6}+\frac{n}{6} \quad = \quad \frac{n^3}{6}+\mathcal{O}(n)$$

Moreover, the total number of multiplications/divisions and additions/subtractions in n passes of i-loop for the formulation of the unit upper triangular matric U will be

$$\sum_{i=1}^{n}(n-i)(i) \quad = \quad \sum_{i=1}^{n}(n-i)(i) \quad = \quad \sum_{i=1}^{n}(ni-i^2)$$

$$= \quad n\sum_{i=1}^{n} i-\sum_{i=1}^{n} i^2 \quad = \quad n\left[\frac{n(n+1)}{2}\right]-\left[\frac{n(n+1)(2n+1)}{6}\right]$$

$$= \quad n(n+1)\left[\frac{n}{2}-\frac{n}{3}-\frac{1}{6}\right] \quad = \quad (n^2+n)\left[\frac{n}{6}-\frac{1}{6}\right]$$

$$= \quad \frac{n^3}{6}-\frac{n}{6} \quad = \quad \frac{n^3}{6}+\mathcal{O}(n)$$

The summary of the operation count of the LU-factorization is given as:

	Operations		Total flops
	Multiplication/Division	Addition/Subtraction	
Lower Triangular Matrix L	$\frac{n^3}{6}-\frac{n}{6}$	$\frac{n^3}{6}+\frac{n}{6}$	$\frac{n^3}{3}$
Upper Triangular Matrix U	$\frac{n^3}{6}-\frac{n}{6}$	$\frac{n^3}{6}-\frac{n}{6}$	$\frac{n^3}{3}-\frac{n}{3}$
LU-factorization	$\frac{n^3}{3}-\frac{n}{3}$	$\frac{n^3}{3}$	$\frac{2n^3}{3}-\frac{n}{3}$

(2) The forward substitution phase occurs after the formulation of L and U as factors of the coefficient matrix for solving the lower triangular system. This phase contains two nested loops. The first loop, say i-loop (which ranges from $i=2$ to n), corresponds to $n-1$ of the components of the intermediate vector Y. The second loop, say j-loop (which ranges from $j=1$ to $i-1$), corresponds to the columns before the diagonal elements.

Each pass of i-loop will perform one subtraction and one division to obtain the value of y_i. Moreover, in each pass of i-loop, the number of both of the multiplications and additions will be $(i-1)-1+1$ (or simply $(i-1)$) in j-loop. Thus, in each pass of i-loop, the total number of both of the multiplications/divisions and additions/subtractions will be $(1+i-1)$ or simply (i).

Hence, the total number of the multiplications/divisions in the forward substitution phase will be

$$\begin{aligned} 1+\sum_{i=2}^{n}(i) &= 1+\sum_{i=2}^{n} i = 1+\left[\frac{n(n+1)}{2}-1\right] \\ &= \frac{n^2}{2}+\frac{n}{2} = \frac{n^2}{2}+\frac{n}{2} = \frac{n^2}{2}+\mathcal{O}(n) \end{aligned}$$

Similarly, the total number of the additions/subtractions in the forward substation phase will be

$$\sum_{i=2}^{n}(i) = \frac{n^2}{2}+\frac{n}{2}-1 = \frac{n^2}{2}+\mathcal{O}(n)$$

The summary of the operation count of the Unit Lower triangular system $LY = B$ is given as:

	Operations		Total flops
	Multiplication/Division	Addition/Subtraction	
Lower triangular system $LY = B$	$\frac{n^2}{2}+\frac{n}{2}$	$\frac{n^2}{2}+\frac{n}{2}-1$	n^2+n-1

(3) The back substitution phase occurs after the solution of the lower triangular system. This phase contains two nested loops. The first loop, say i-loop (which ranges from $i = n-1$ to 1), corresponds to $n-1$ of the components of solution vector X. The second loop, say j-loop (which ranges from $j = i+1$ to n), corresponds to the columns after the diagonal elements.

Each pass of i-loop will perform one subtraction to obtain the value of x_i. Moreover, in each pass of i-loop, the number of both of the multiplications and additions will be $n-(i+1)+1$ (or simply $(n-i)$) in j-loop. Thus, in each pass of i-loop, the total number of multiplications/divisions will be $(n-i)$ and the total number of additions/subtractions will be $(n-i+1)$.

Hence, the total number of the multiplications/divisions in the back substitution phase will be

$$\begin{aligned} \sum_{i=1}^{n-1}(n-i) &= n\sum_{i=1}^{n-1}1-\sum_{i=1}^{n-1} i = n(n-1)-\left[\frac{(n-1)n}{2}\right] \\ &= n^2-n-\frac{n^2}{2}+\frac{n}{2} = \frac{n^2}{2}-\frac{n}{2} = \frac{n^2}{2}+\mathcal{O}(n) \end{aligned}$$

Similarly, the total number of the additions/subtractions in the back substation phase will be

$$\sum_{i=1}^{n-1}(n+1-i) \quad = \quad \frac{n^2}{2}+\frac{n}{2}-1 \quad = \quad \frac{n^2}{2}+\mathcal{O}(n)$$

The summary of the operation count of the Upper triangular system $UX = Y$ is given as:

	Operations		Total flops
	Multiplications/Divisions	Additions/Subtractions	
Upper triangular system $UX = Y$	$\frac{n^2}{2}-\frac{n}{2}$	$\frac{n^2}{2}+\frac{n}{2}-1$	n^2-1

■■■

Chapter Summary

- A ***system of linear equations*** (simply called as ***linear system***) is a set or collection of two or more linear equations with the same set of variables whose simultaneous solution satisfies all the equations. Precisely, a linear system can be referred to as a set of ***simultaneous linear algebraic equations***.

- If $m > n$, where m is the number of equations and n is the number of unknowns, then the linear system is called ***over-determined***. If $m < n$, then the linear system is called ***under-determined***.

- A linear system $AX = B$ is called ***homogenous*** if B is a zero vector (i.e., $B = \bar{\mathbf{0}}$), and ***non-homogeneous*** or ***inhomogeneous*** if $B \neq \bar{\mathbf{0}}$.

- A non-homogeneous linear system $AX = B$ is called **consistent** if it has a unique solution or infinitely many solutions, and it is called ***inconsistent*** if it has no solution.

- If A^{-1} does not exist, then matrix A is called ***singular*** or ***non-invertible***. If A^{-1}exists then A is called ***non-singular*** matrix and is ***invertible***.

- If $\det(A) = 0$, then A^{-1} does not exist and the system $AX = B$ does not have a unique solution; the system either has no solution or infinitely many solutions.

- Although the steps of the algorithms for the solution of a linear system are elementary in nature, there might be certain pitfalls. This raises the need of skillful selection and use of an appropriate algorithm for obtaining the solution.

- In general, methods for the solution of linear systems (also called **linear solvers**) are evaluated based on their *accuracy*, *speed of convergence*, and computer *resource requirements* (CPU-requirements, memory requirements).

- A linear equation in two variables, say x and y, represents a **line** in xy-plane. If there exists a unique solution of the system then it is the point where the two lines intersect.

- A linear equation in three variables, say x, y, and z, represents a **plane** in xyz-space. If there exists a unique solution of such a system then it is the point where the three planes intersect.

- There are two broad categories of methods to solve linear systems: the **direct** (also called **exact**) methods and **iterative** methods. The prominent features of these two categories can be found in Question 5 (Section 6.1).

- An $n \times n$ square matrix $U = (u_{ij})$ is called the ***upper triangular matrix*** if $u_{ij} = 0$ whenever $i > j$. A linear system $UX = Y$ is said to be ***upper triangular system*** if it's coefficient matrix is an upper triangular one. It has a unique solution if no diagonal element is zero (i.e., $|u_{ii}| \neq 0$, for $i = 1,2,\cdots,n$), otherwise it has either no solution or infinitely many solutions. If there is a unique solution of an upper triangular system then the solution can easily be obtained by a so-called ***back substitution*** process. In analogy, the said propositions also hold for a lower triangular matrix $L = (l_{ij})$ for which $l_{ij} = 0$ whenever $i < j$. The solution of a lower-triangular system can be obtained by a similar so-called ***forward substitution*** process.

- To solve a linear system $AX = B$, the ***Gaussian Elimination*** method aims at obtaining an upper triangular system $UX = Y$, equivalent to $AX = B$. This process may be termed as ***forward elimination***. The upper triangular system can then be solved by back substitution.

- To guard against the pitfalls of the ***Gaussian Elimination*** method, the process of ***pivoting*** is performed while using the method. The pivoting could be any of ***partial***, ***scaled*** or ***complete***.

- ***Pivoting*** refers to the interchanging of two rows of the augmented matrix so that the diagonal coefficient (to be used as the pivot element) is of greatest magnitude among the possible ones for the row under consideration.

- Pivoting **must be** performed if the main diagonal coefficient is zero (to make the triangular system non-singular). Pivoting **should be** performed if the magnitude of the main diagonal element is a smaller one (to prevent the propagation of the round-off error).

- The ***Gauss-Jordan method*** is a variant of the Gaussian Elimination method. It is based on the same elementary row operations; however, it eliminates all the elements below as well as above the pivot element (in the same column) . Thus it does not produce an upper-triangular system for back-substitution; rather it obtains a diagonal matrix in which the solution vector is almost readily available.

- The LU Factorization or LU Decomposition method is another direct solver. A concise description of this method (and its variants) can be found in Question 12 (Section 6.5).

- The operation count analysis of an algorithm usually refers to the counting of the arithmetic operations involved. This is useful in determining the execution time required by the algorithm. For numerical computations, the operation count analysis is mostly considered as the counting of the ***floating-point operations*** (simply called as ***flops***) involved in the algorithm.

- The additions/subtractions are considered to be requiring less CPU-time (being lighter operations) as compared to the multiplications/divisions. Therefore, it might be appropriate to count the two types of operations separately for the operation count analysis.

■■■

Chapter Exercises

Exercise 01: Solve the following system using the Gaussian Elimination method with back substitution.

$$\begin{array}{rcrcrcr} 2x_1 & - & 3x_2 & + & x_3 & = & -1 \\ 4x_1 & + & 4x_2 & - & 3x_3 & = & 3 \\ -2x_1 & + & 3x_2 & + & x_3 & = & 7 \end{array}$$

Exercise 02: Solve the following system using the Gaussian Elimination method with partial pivoting.

$$\begin{array}{rcrcrcr} x_1 & + & x_2 & + & x_3 & = & 6 \\ 3x_1 & + & 3x_2 & + & x_3 & = & 12 \\ 2x_1 & + & x_2 & + & 5x_3 & = & 20 \end{array}$$

Exercise 03: Solve the following system using the Gaussian Elimination method with partial pivoting and three-digit rounding arithmetic.

$$\begin{array}{rcrcrcr} 2.5x_1 & - & 3x_2 & + & 4.6x_3 & = & -1.05 \\ -3.5x_1 & + & 2.6x_2 & + & 1.5x_3 & = & -14.46 \\ -6.5x_1 & + & -3.5x_2 & + & 7.3x_3 & = & -17.735 \end{array}$$

Exercise 04: Solve the following system using the Gaussian Elimination method with scaled partial pivoting.

$$\begin{array}{rcrcrcr} x_1 & + & x_2 & - & 2x_3 & = & 3 \\ 4x_1 & - & 2x_2 & + & x_3 & = & 5 \\ 3x_1 & - & x_2 & + & 3x_3 & = & 8 \end{array}$$

Exercise 05: Solve the following system using the Gaussian Elimination method with scaled partial pivoting and four-digit rounding arithmetic.

$$\begin{array}{rcrcrcr} 3.03x_1 & - & 12.1x_2 & + & 14x_3 & = & -119 \\ -3.03x_1 & + & 12.1x_2 & - & 7x_3 & = & 120 \\ 6.11x_1 & - & 14.2x_2 & + & 21x_3 & = & -139 \end{array}$$

Exercise 06: Solve the following system using the Gaussian Elimination method with complete pivoting.

$$\begin{array}{rcrcrcr} x_1 & + & 2x_2 & + & 2x_3 & = & 1 \\ 2x_1 & + & 6x_2 & + & 10x_3 & = & -2 \\ 3x_1 & + & 14x_2 & + & 28x_3 & = & -11 \end{array}$$

Exercise 07: Solve the following system using the Gaussian Elimination method with complete pivoting and three-digit rounding arithmetic.

$$\begin{array}{rcrcrcr} 1.012x_1 & - & 2.132x_2 & + & 3.104x_3 & = & 1.984 \\ -2.132x_1 & + & 4.096x_2 & - & 7.013x_3 & = & -5.049 \\ 3.104x_1 & - & 7.013x_2 & + & 0.014x_3 & = & -3.895 \end{array}$$

Exercise 08: Solve the following system using the Gauss-Jordan method

$$\begin{array}{rcrcrcr} x_1 & + & 2x_2 & + & x_3 & = & 6 \\ 2x_1 & + & 3x_2 & + & 4x_3 & = & 12 \\ 4x_1 & + & 3x_2 & + & 2x_3 & = & 12 \end{array}$$

Exercise 09: Solve the following system using the Gauss-Jordan method and three-digit rounding arithmetic.

$$\begin{array}{rcrcrcr} 0.125x_1 & + & 0.201x_2 & + & 0.401x_3 & = & 2.306 \\ 0.375x_1 & + & 0.501x_2 & + & 0.601x_3 & = & 4.806 \\ 0.501x_1 & + & 0.301x_2 & + & 0.001x_3 & = & 2.91 \end{array}$$

Exercise 10: Solve the following linear system $AX = B$ using the Doolittle's method.

$$\begin{array}{rcrcrcr} x_1 & + & x_2 & + & x_3 & = & 3 \\ 2x_1 & - & x_2 & + & 2x_3 & = & 16 \\ 3x_1 & + & x_2 & + & x_3 & = & -3 \end{array}$$

Exercise 11: Solve the following linear system $AX = B$ using the Doolittle's method.

$$\begin{array}{rcrcrcrcr} x_1 & + & x_2 & + & 2x_3 & + & 2x_4 & = & 9 \\ 2x_1 & + & 4x_2 & + & 7x_3 & + & 3x_4 & = & 25 \\ -x_1 & - & 5x_2 & - & 6x_3 & + & 2x_4 & = & -17 \\ x_1 & - & x_2 & + & 3x_3 & + & 8x_4 & = & 15 \end{array}$$

Exercise 12: Solve the following linear system $AX = B$ using the Crout's method.

$$\begin{array}{rcrcrcr} 8x_1 & + & x_2 & - & x_3 & = & 8 \\ 2x_1 & + & x_2 & + & 9x_3 & = & 12 \\ x_1 & - & 7x_2 & + & 2x_3 & = & -4 \end{array}$$

Exercise 13: Solve the following linear system $AX = B$ using the Crout's method.

$$\begin{array}{rcrcrcrcr} x_1 & + & x_2 & + & 0x_3 & + & 3x_4 & = & 9 \\ 2x_1 & + & x_2 & - & x_3 & + & x_4 & = & 5 \\ 3x_1 & - & x_2 & + & x_3 & + & 2x_4 & = & 6 \\ -x_1 & + & 2x_2 & + & 3x_3 & - & x_4 & = & 4 \end{array}$$

Exercise 14: Solve the following linear system $AX = B$ using the Cholesky's method.

$$\begin{array}{rcrcrcr} 2x_1 & + & 3x_2 & + & 4x_3 & = & 1 \\ 3x_1 & + & 8x_2 & + & 5x_3 & = & 6 \\ 4x_1 & + & 5x_2 & + & 10x_3 & = & -1 \end{array}$$

Exercise 15: Solve the given linear system $AX = B$ using the Cholesky's method

$$\begin{array}{rcrcrcrcr} 4x_1 & + & x_2 & + & x_3 & + & x_4 & = & 9 \\ x_1 & + & 3x_2 & - & x_3 & + & x_4 & = & 4 \\ x_1 & - & x_2 & + & 2x_3 & + & 0x_3 & = & 4 \\ x_1 & + & x_2 & + & 0x_3 & + & 2x_4 & = & 6 \end{array}$$

Exercise 16: The upward velocity of a rocket at three different times after its launching are given as follows:

Time, t in (s)	Velocity, v in (m/s)
6	115.7
9	182.5
12	295.6

The velocity data is approximated by a polynomial as

$$v(t) = a_1 t^2 + a_2 t + a_3, \quad 5 \le t \le 12$$

Thus, the coefficients a_1, a_2 and a_3 for the above expression are given by

$$\begin{bmatrix} 36 & 6 & 1 \\ 81 & 9 & 1 \\ 144 & 12 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 115.7 \\ 182.5 \\ 295.6 \end{bmatrix}$$

Find the values of a_1, a_2 and a_3 using a linear solver. Then, calculate the velocity at $t = 7, 8, 10,$ and 11 seconds.

Exercise 17: A factory produces three products, say Prod1, Prod2, and Prod3, by using three kinds of raw materials, say Raw1, Raw2, and Raw3. The units of each of the raw materials needed to produce one unit of each of the products are shown the table below.

Sectors	Raw1	Raw2	Raw3
Prod1	5	3	1
Prod2	4	4	3
Prod3	2	1	3

If 335 units of Raw1, 532 units of Raw2, and 440 units of Raw3 are available, then how much each of the three products can be produced.

Hint for the Solution:

Assume that x_1, x_2 and x_3 represent the quantities of the products: Prod1, Prod2, and Prod3, respectively. The problem can be represented by a linear system whose solution would provide the required values.

$$\begin{array}{ccccccc} 5x_1 & + & 4x_2 & + & 2x_3 & = & 335 \\ 3x_1 & + & 4x_2 & + & x_3 & = & 532 \\ x_1 & + & 3x_2 & + & 3x_3 & = & 440 \end{array}$$

Exercise 18: Assume that the economy of a country depends on the three sectors: Food, Cloth, and House. The production of one unit of each of these needs certain units of each of these sectors, as shown in the following table:

Sectors	Food Units Needed	Cloth Units Needed	House Units Needed
Food	0.45	0.18	0.15
Cloth	0.25	0.27	0.07
House	0.30	0.40	0.45

The consumer demand is as in the table below:

Sector	worth in billion rupees
Food	220
Cloth	185
House	550

For satisfying the above demands, what total output is required from each of the sectors.

Hint for the Solution in MATLAB: Assume that x_1, x_2 and x_3 represent the total outputs in units from Food, Cloth and House sectors, respectively. The problem can be represented by a linear system whose solution would provide the required values.

Exercise 19: A bakery produces three products: Cake, Pastry, and Muffin. It uses three kinds of materials: Flour, Milk, and Sugar. The units of each of the raw materials needed to produce one unit of each of the bakery products are shown the table below.

Product ->	Cake	Pastry	Muffin
Flour	6	5	3
Milk	4	5	2
Sugar	2	3	3

If 347 units of Flour, 604 units of Milk, and 502 units of Sugar are available, then how much each of the three products can be produced.

Exercise 20: Pivoting is necessary with the Gaussian elimination if

(A) the coefficient matrix is singular (B) the linear system is homogenous

(C) the linear system is ill conditioned (D) None of above

Exercise 21: Cholesky decomposition for a linear system is not possible, if

(A) the linear system is ill conditioned (B) the linear system is homogenous

(C) the coefficient matrix is asymmetric (D) None of above

■■■

Chapter 7

Iterative Linear Solvers

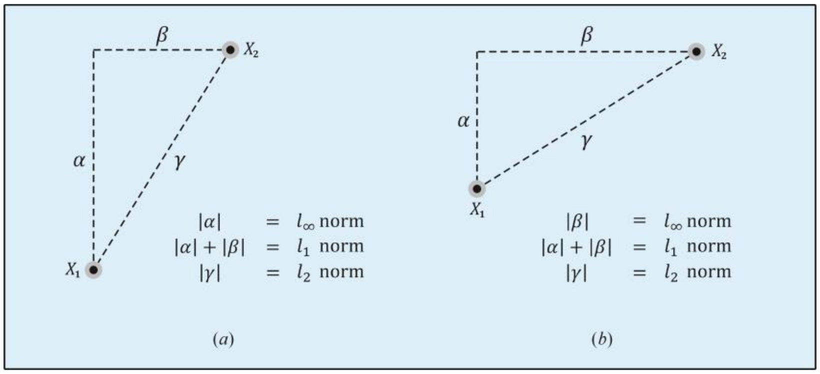
Corridor I: BASICS

Let's plan it

To unleash the topics of this Corridor, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

■■■



β
X_2
α
γ
X_1
$|\alpha| = l_\infty$ norm
$|\alpha| + |\beta| = l_1$ norm
$|\gamma| = l_2$ norm
(a)
β
X_2
α
γ
X_1
$|\beta| = l_\infty$ norm
$|\alpha| + |\beta| = l_1$ norm
$|\gamma| = l_2$ norm
(b)

Fig. (7.4): *Explanation of the different types of distances between the two vectors.*

Corridor II: ANALYSIS

Let's think deep

To unleash the topics of this Corridor, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

■■■

Corridor III: PROGRAMMING ARCADE

Let's do it

To see more examples for practicing, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

For codes, please visit: https://github.com/DrAmjadAli11/SimplifiedNumericalAnalysis

■■■

7.6 Algorithms and Implementations

Question 22: Write down an algorithm (pseudo code) to solve a linear system using the Jacobi method.

The ***Jacobi method*** can be written in a compact form as

$$x_i^{(k)} = \frac{1}{a_{ii}}\left[b_i - \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_j^{(k-1)}\right], \qquad \text{for } i = 1, 2, \cdots, n$$

Algorithm: To solve $AX = B$, given an initial approximation $X^{(0)}$.

INPUTS:
$$\begin{cases} \boldsymbol{n}\text{: an integer as the number of equations and unknowns} \\ \boldsymbol{A} = (\boldsymbol{a_{ij}}), 1 \le i, j \le \boldsymbol{n}\text{: a real valued square matrix as the coefficient matrix} \\ \boldsymbol{B} = [\boldsymbol{b_1}, \boldsymbol{b_2}, \cdots, \boldsymbol{b_n}]^{\boldsymbol{T}}\text{: a real valued vector as the vector of right hand side constants} \\ \boldsymbol{X} = [\boldsymbol{x_1}, \boldsymbol{x_2}, \cdots, \boldsymbol{x_n}]^{\boldsymbol{T}}\text{: a real valued vector (having initial approximation, } X^{(0)}) \\ \boldsymbol{TOL}\text{: a real value as the error tolerance} \\ \boldsymbol{N}\text{: an integer as the maximum number of iterations} \end{cases}$$

OUTPUT:
$$\begin{cases} \boldsymbol{X} = [\boldsymbol{x_1}, \boldsymbol{x_2}, \cdots, \boldsymbol{x_n}]^{\boldsymbol{T}}\text{: a real valued vector as the approximate solution} \\ \text{(either on convergence, or on completing } \boldsymbol{N} \text{ iterations — which ever happens first)} \end{cases}$$

```
Step 1    Receive the inputs as stated above

Step 2    for $\boldsymbol{k} = \boldsymbol{1}, \boldsymbol{2}, \boldsymbol{3}, \cdots, \boldsymbol{N}$ perform steps 3-6

    Step 3    for $i = 1, 2, \cdots, \boldsymbol{n}$   Set $\boldsymbol{xp_i} = \boldsymbol{x_i}$
              ($\boldsymbol{XP} = [\boldsymbol{xp_1}, \boldsymbol{xp_2}, \cdots, \boldsymbol{xp_n}]^{\boldsymbol{T}}$ is to keep a copy of present
               approximation $\boldsymbol{X}$, because $\boldsymbol{X}$ is going to be updated)

    Step 4    for $i = 1, 2, \cdots, \boldsymbol{n}$        (compute the components of solution vector $\boldsymbol{X}$)

                  $sum = 0$
                  for $j = 1, 2, \cdots, \boldsymbol{n}$
                      $if\ (\boldsymbol{j} \neq \boldsymbol{i})\quad sum = sum + \boldsymbol{a_{ij}} \times \boldsymbol{XP_j}$
                  $\boldsymbol{x_i} = \frac{[\boldsymbol{b_i} - sum]}{\boldsymbol{a_{ii}}}$
              $\left( x_i^{(k)} = \frac{1}{a_{ii}}\left[b_i - \sum_{j=1, j\neq i}^{n} a_{ij}x_j^{(k-1)}\right]\right)$

    Step 5    Compute $\boldsymbol{err} = \|\boldsymbol{X} - \boldsymbol{XP}\|$     (or $\boldsymbol{err} = \|\boldsymbol{X} - \boldsymbol{XP}\|/\|\boldsymbol{X}\|$) Here $\|\cdot\|$ is any suitable norm.

    Step 6

          if ($\boldsymbol{err} < \boldsymbol{TOL}$ )then        This means that the consecutive
                 Exit/Break the loop         approximations are nearly the same.
                                             Therefore, stop iterations.

    end for loop of Step 2      (Go to Step 3)

Step 7    Print the output: $\boldsymbol{X} = [\boldsymbol{x_1}, \boldsymbol{x_2}, \dots, \boldsymbol{x_n}]^{\boldsymbol{T}}$
```

```
if ($\boldsymbol{err} < \boldsymbol{TOL}$)    OUTPUT ('The desired accuracy achieved; Solution converged.')
else                 OUTPUT ('The number of iterations exceeded the maximum limit.')
```

STOP.

Question 23: What modification a programmer needs to make in the algorithm (pseudo code) of the Jacobi method (as given in the answer of Question 22) to convert it into the Gauss-Seidel method for solving a linear system.

The algorithm (pseudo code) of the Jacobi method (as given in the answer of Question 22) can be converted into the algorithm of the Gauss-Seidel method simply by replacing its Step 4 with the following:

```
Step 4    for $i = 1, 2, \cdots, \boldsymbol{n}$        (compute the components of solution vector $\boldsymbol{X}$)
              $sum = 0$
              for $j = 1, 2, \cdots, \boldsymbol{n}$
                  $if\ (\boldsymbol{j} \neq \boldsymbol{i})\quad sum = sum + \boldsymbol{a_{ij}} \times \boldsymbol{x_j}$
              $\boldsymbol{x_i} = \frac{[\boldsymbol{b_i} - sum]}{\boldsymbol{a_{ii}}}$
```

$$\left(\begin{array}{l} sum = \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} + \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)} \\ x_i^{(k)} = \frac{1}{a_{ii}} [b_i - sum] \end{array}\right)$$

■

Question 24: What modification a programmer needs to make in the algorithm (pseudo code) of the Jacobi method (as given in the answer of Question 22) to convert it into the Gauss-Seidel method with over-relaxation (i.e., the SOR method) for solving a linear system.

The algorithm (pseudo code) of the Jacobi method (as given in the answer of Question 22) can be converted into the algorithm of the SOR method simply by taking one more input:

$\boldsymbol{WF} = 1.3$:   a real value as the over − relaxation / weighting factor

And then replacing Step 4 with the following:

```
Step 4
      for $i = 1, 2, \cdots, \boldsymbol{n}$        (compute the components of solution vector $\boldsymbol{X}$)
              $sum = 0$
              for $j = 1, 2, \cdots, \boldsymbol{n}$
                  $if\ (\boldsymbol{j} \neq \boldsymbol{i})\quad sum = sum + \boldsymbol{a_{ij}} \times \boldsymbol{x_j}$
              $\boldsymbol{x_i} = \frac{[\boldsymbol{b_i} - sum]}{\boldsymbol{a_{ii}}}$

              $\boldsymbol{x_i} = \boldsymbol{WF} \times \boldsymbol{x_i} + (1 - \boldsymbol{WF})\boldsymbol{XP_i}$        (apply over − relaxation)
```

$$\left(\begin{array}{l} sum = \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} + \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)} \\ x_i^{(k)} = \frac{1}{a_{ii}} [b_i - sum] \end{array}\right)$$

■

Question 24: Write a Python program to solve the following linear system using the Jacobi method. Take initial approximate solution as: $X^{(0)} = [0, \quad 0, \quad 0]^T$. The iterations of the method should stop when either the approximation is accurate within 10^{-6}, or the number of iterations exceeds 200, whichever happens first.

$$5x_1 + 3x_2 + 2x_3 = 17, \qquad 3x_1 + 4x_2 - x_3 = 8, \qquad -x_1 + x_2 - 3x_3 = -8$$

script_7.1: jacobi.ipynb

```
from numpy import *

N = 200
TOL = 0.000001
n = 3
a = [[5, 3, 2], [3, 4, -1], [-1, 1, -3]]
b = [17, 8, -8]

x = zeros(n)
xp = zeros(n)
print("iter.       x1          x2          x3            Error")

for k in range(1,N+1):
        print(k, end=" ")
        for i in range(n):
                xp[i] = x[i]
        for i in range(n):
                sum = 0
                for j in range(n):
                        if j!=i:
                                sum = sum + a[i][j] * xp[j]
                x[i] = (b[i] – sum) / a[i][i]
                print("      ", round(x[i],8), end=" ")

        sum = 0
        for i in range(n):
                sum = sum + ((x[i] – xp[i]) * (x[i] – xp[i]))
        err = sqrt(sum)
        print("    ", round(err,8))
        if err < TOL:
                break

print("The latest approximate solution vector is given:")
for i in range(n):
        print(x[i], "\t", end=" ")

if err < TOL:
        print("\nThe desire accuracy is achieved; Solution is convergent.")
else:
        print("\nThe number of iterations exceeded the maximum limit.")
```

Output Console:

```
iter.       x1            x2            x3              Error
1       3.4             2.0           2.66666667      4.76141902
2       1.13333333      0.11666667    2.2             2.98370575
3       2.45            1.7           2.32777778      2.06322144
4       1.44888889      0.74444444    2.41666667      1.38679887
5       1.98666667      1.5175        2.43185185      0.94183355
6       1.51675926      1.11796296    2.51027778      0.62176639
7       1.72511111      1.49          2.53373457      0.42705067
8       1.49250617      1.33960031    2.5882963       0.28231562
.
.
.
128     1.00000857      1.99999121    2.99999365      1.34e-06
129     1.00000781      1.99999198    2.99999421      1.22e-06
130     1.00000713      1.9999927     2.99999472      1.11e-06
131     1.00000649      1.99999333    2.99999519      1.01e-06
132     1.00000593      1.99999393    2.99999561      9.3e-07
The latest approximate solution vector is given:
1.00000593       1.99999393       2.99999561
The desire accuracy is achieved; Solution convergent
```

Remark: Replacing xp[j] by x[j] in line 34 of the Python program in Problem 17 would convert the program for the Gauss-Seidel method, because it would then correspond to computing

$$sum = \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} + \sum_{j=i+1}^{n} a_{ij} x_j^{(k-1)}$$

Remark: In the program of Problem 17, the code segment of lines 45-47 can be placed just before line 45 to print the latest result on completion of each of the iterations.

■

Question 25: Write a Python program to solve the following linear system using the Gauss-Seidel method with over-relaxation (the SOR method). Take initial approximate solution as: $X^{(0)} = [0, \quad 0, \quad 0]^T$ and over-relaxation factor as 1.2. The iterations of the method should stop when either the approximation is accurate within 10^{-6}, or the number of iterations exceeds 200, whichever happens first.

$$5x_1 + 3x_2 + 2x_3 = 17, \qquad 3x_1 + 4x_2 - x_3 = 8, \qquad -x_1 + x_2 - 3x_3 = -8$$

script_7.2: gauss_seidel.ipynb

```python
from numpy import *

N = 200
TOL = 0.000001
```

```
n = 3
wf = 1.2
a = [[5, 3, 2], [3, 4, −1],[−1, 1, −3]]
b = [17, 8, −8]

x = zeros(n)
xp = zeros(n)
print("iter.       x1           x2           x3            Error")

for k in range(1,N+1):
        print(k, end=" ")
        for i in range(n):
                xp[i] = x[i]
        for i in range(n):
                sum = 0
                for j in range(n):
                        if j!=i:
                                sum = sum + a[i][j] * x[j]
                x[i] = (b[i] − sum) / a[i][i]
                x[i] = (wf * x[i]) + ((1 − wf) * xp[i])
                print("      ", round(x[i],8), end=" ")

        sum = 0
        for i in range(n):
                sum = sum + ((x[i] − xp[i]) * (x[i] − xp[i]))
        err = sqrt(sum)
        print("     ", round(err,8))
        if err < TOL:
                break

print("The latest approximate solution vector is given:")
for i in range(n):
        print(x[i], "\t", end=" ")

if err < TOL:
        print("\nThe desire accuracy is achieved; Solution is convergent.")
else:
        print("\nThe number of iterations exceeded the maximum limit.")
```

Output Console:

```
iter.         x1           x2            x3              Error
1           4.081        -1.2721        1.0592          4.40298633
2          3.6714242     -0.33212162    1.38674176      1.07591327
3          2.919206713   0.255160813    1.85703329      1.06391052
4          2.42106694    0.727117624    2.15101363      0.74653261
5          2.039775395   1.064082715    2.3795202       0.55780278
6          1.763735676   1.313677426    2.54407266      0.40690657
7          1.560250257   1.496261097    2.6655898       0.29918143
8          1.411158868   1.630381758    2.7545712       0.21939527
.
```

```
.
.
45     1.0000043845     1.9999960645      2.99999738      2.32e-06
46     1.0000032246     1.999997146       2.99999808      1.71e-06
47     1.0000023747     1.9999978747      2.99999858      1.25e-06
48     1.0000017448     1.9999984348      2.99999896      9.2e-07
The latest approximate solution vector is given:
1.00000174       1.99999843       2.99999896
The desire accuracy is achieved; Solution convergent
```

■

Remark: The Python programs in Question 23 and 25 can be modified to receive the linear system at the execution time (instead of fixing in the code). For this, the lines 6-7 in the program of Question 23 and the lines 7-8 in the program of Question 25 should be replaced by the following code segment:

```
n = int(input("Enter the number of unknowns: "))

# Initialize coefficient matrix 'a' as a list of lists
a = []
print("Enter the coefficient matrix row-wise:")
for i in range(n):
    row = []
    for j in range(n):
        row.append(float(input()))
    a.append(row)

# Initialize the constant vector 'b' as a list
b = []
print("Enter the elements of constant vector 'B':")
for i in range(n):
    b.append(float(input()))

# Display the coefficient matrix 'a' and vector 'b'
print("Coefficient Matrix 'a':")
for row in a:
    print(row)

print("Constant Vector 'b':")
print(b)
```

Chapter Summary

- The ***norm of a vector*** is a real-valued function that provides a measure of "size", "length", or "magnitude" of the vector. Let $\mathbb{R}$ denotes the set of real numbers, and $\mathbb{R}^n$ denotes the space of n-dimensional real-valued column vectors. A norm of a vector on $\mathbb{R}^n$ is a function, $\|\cdot\| : \mathbb{R}^n \to \mathbb{R}$, with the following properties,

    1. $\|X\| \geq 0$, for all $X \in \mathbb{R}^n$
    2. $\|X\| = 0$, if and only if $X = \mathbf{0}$ in $\mathbb{R}^n$
    3. $\|\alpha X\| = |\alpha|\|X\|$, for all $\alpha \in \mathbb{R}$ and $X \in \mathbb{R}^n$
    4. $\|X + Y\| \leq \|X\| + \|Y\|$, for all $X, Y \in \mathbb{R}^n$

- The vector norm definitions, as well as the concerning illustrations, can be found in Question 01 (Section 7.1).
- The norm of a vector gives a measure for the distance between an arbitrary vector and the zero vector, just as the absolute value of a real number is its distance from 0.
- The ***distance between two vectors*** is defined as the norm of the "difference vector" of the two vectors, just as the distance between two real numbers is the absolute value of their difference. The definitions of different vector distances, as well as the concerning illustrations, can be found in Question 02 (Section 7.1).
- To determine the convergence of an iterative solution, the norm of the difference vector of every two consecutive approximations is ensured to be smaller than a pre-specified error tolerance τ, i.e.,

$$\left\|X^{(k)} - X^{(k-1)}\right\| \quad < \quad \tau$$

- A square matrix, say $A = \left(a_{ij}\right)_{n\times n}$, is said to be ***diagonally dominant*** if, for $i = 1, 2, \cdots, n$

$$|a_{ii}| \quad \geq \quad \sum_{\substack{j=1 \\ j\neq i}}^{n} \left|a_{ij}\right|,$$

- A linear system is said to be diagonally dominant if its coefficient matrix is ***diagonally dominant*** (i.e., the magnitude of the diagonal entry in a row is greater than or equal to the sum of the magnitudes of all other entries in that row).
- If "$\geq$" is replaced by "$>$" in the above equation, then A is said to be ***strictly diagonally dominant.*** A strictly diagonally dominant matrix is always non-singular.

- If a linear system is not diagonally dominant, then a rearrangement of its rows might make it diagonally dominant.
- The Gauss-Jacobi, Gauss-Seidel, and SOR methods must converge if the linear system to be solved is diagonally dominant.
- Suppose that $AX = B$ is a $n \times n$ linear system to be solved such that $A = \left(a_{ij}\right)_{n\times n}$ is the coefficient matrix, $B = (b_i)_{n\times 1}$ is the vector of right-hand side constants, and $X = (x_i)_{n\times 1}$ is the vector of unknowns.
    - The ***Jacobi method*** can be written in a compact form as

$$x_i^{(k)} = \frac{1}{a_{ii}}\left[b_i - \sum_{\substack{j=1\\ j\neq i}}^{n} a_{ij}x_j^{(k-1)}\right], \qquad \text{for } i = 1, 2, \cdots, n$$

    - The ***Gauss-Seidel method*** can be written in a compact form as

$$x_i^{(k)} = \frac{1}{a_{ii}}\left[b_i - \left(\sum_{j=1}^{i-1} a_{ij}x_j^{(k)} + \sum_{j=i+1}^{n} a_{ij}x_j^{(k-1)}\right)\right], \qquad \text{for } i = 1, 2, \cdots, n$$

    - The ***successive over-relaxation*** (**SOR**) ***method*** can be written in a compact form as

$$\bar{\bar{x}}_i^{(k)} = \frac{1}{a_{ii}}\left[b_i - \left(\sum_{j=1}^{i-1} a_{ij}x_j^{(k)} + \sum_{j=i+1}^{n} a_{ij}x_j^{(k-1)}\right)\right], \qquad \text{for } i = 1, 2, \cdots, n$$

$$x_i^{(k)} = \omega\bar{\bar{x}}_i^{(k)} + (1-\omega)x_i^{(k-1)} \qquad (\text{for } 1 \le \omega \le 2, \text{ usually the best is around } 1.2\ )$$

Here $k = 1, 2, 3, \cdots$, represents the iterations and $x_i^{(k)}$ represents the kth approximation of the ith unknown. The iterative procedure is started with an initial approximation vector $X^{(0)} = \left[x_1^{(0)},\ x_2^{(0)},\ x_3^{(0)},\ \cdots,\ x_n^{(0)}\right]^{\boldsymbol{T}}$ and produces a sequence of successive approximations $\left\{X^{(k)}\right\}_{k=1}^{\infty}$, such that $X^{(k)} = \left[x_1^{(k)},\ x_2^{(k)},\ x_3^{(k)},\ \cdots,\ x_n^{(k)}\right]^{\boldsymbol{T}}$. The sequence is anticipated to refine/improve the approximate solution gradually and ultimately converge to the exact solution vector (theoretically). In practice, the iterations of the method are stopped when a sufficient level of accuracy is achieved.

- The ***norm of a matrix*** is a real-valued function that provides a measure of "size", "length", or "magnitude" of the matrix. Let $\mathbb{R}$ denotes the set of real numbers, and $\mathbb{M}^n$ denotes the set of $n \times n$ real-valued matrices. The norm of a matrix on $\mathbb{M}^n$ is a function, $||\cdot|| : \mathbb{M}^n \to \mathbb{R}$, with the following properties,

    1. $||A|| \ge 0$, for all $A \in \mathbb{M}^n$

2. $\|A\| = 0$, if and only if $A = \mathbf{0}$ in $\mathbb{M}^n$
3. $\|\alpha A\| = |\alpha|\|A\|$, for all $\alpha \in \mathbb{R}$ and $A \in \mathbb{M}^n$
4. $\|A + B\| \leq \|A\| + \|B\|$, for all $A, B \in \mathbb{M}^n$
5. $\|AB\| \leq \|A\|\|B\|$, for all $A, B \in \mathbb{M}^n$

- The matrix norm definitions can be found in Question 11 (Section 7.4).
- The ***distance between two matrices*** A and B with respect to a certain norm $\|\cdot\|$ is defined as the norm of the "difference matrix" of the two matrices, i.e., $\|A - B\|$.
- The ***condition number*** of a non-singular matrix A with respect to a matrix norm $\|\cdot\|$ is defined as

$$\mathcal{K}(A) = \|A\|\|A^{-1}\|, \quad (\text{and } \mathcal{K}(A) \geq 1)$$

- The ***condition number of a linear system*** is the condition number of its coefficient matrix.
- A computational problem is called ***ill-conditioned*** (or ill-posed) if small changes in the data (the input) cause large changes in the solution (the output). On the other hand, a problem is called ***well-conditioned*** (or well-posed) if small changes in the data cause only small changes in the solution.
- The main issue while solving an ill-conditioned problem is that the round-off errors can cause production of wide range worthless solutions (which appear to be original ones because they approximately satisfy the given problem). Therefore, minimizing the round-off errors becomes more relevant for the ill-conditioned problems.
- If $AX = B$ is an ***ill-conditioned linear system*** then the solution of its perturbed system (the one which is obtained by making small changes in the original system, either through small changes in A, or in B) is much different from that of the original linear system. In that case, the matrix A is said to be an ill-conditioned matrix. The determinant of an ill-conditioned matrix A is usually close to zero (NOT the zero). Remind that if the determinant is exactly zero then a relevant linear system $AX = B$ has either no solution, or an infinite number of solutions.
- There is no strict line between the well-conditioning and ill-conditioning of a system, as these concepts are qualitative. A linear system whose condition number (i.e., the condition number of its coefficient matrix) is close to 1 is well-conditioned, whereas a condition number significantly larger than 1 indicates that the linear system is ill-conditioned. If the condition number is below 100, it is usually not a reason for concern. However, a condition number of more than 100 calls for caution. It may be noted that a coefficient matrix, having magnitudes of diagonal elements larger than that of other elements in each of the rows, indicates well-conditioning of the linear system.
- In general, an iterative linear solver involves a process that converts an $n \times n$ system $AX = B$ into an equivalent system of the form $X = TX + C$ for some fixed matrix T and vector C. After

the initial vector $X^{(0)}$ is selected, the sequence of approximate solution vectors, $X^{(1)}$, $X^{(2)}$, $X^{(3)}, \cdots$, is generated by computing

$$X^{(k)} = TX^{(k-1)} + C, \qquad \text{for } k = 1, 2, 3, \cdots$$

The matrix T is called the ***iteration matrix*** of the iterative method, and the relation is called the ***matrix form*** of the iterative method.

- The iterative linear solvers for which the iteration matrix remains unchanged (or fixed) during the iterative process are said to be ***stationary solvers***, whereas the iterative linear solvers for which the iteration matrix changes from iteration to iteration are referred to as ***non-stationary solvers***.
- Examples of stationary solvers include simple methods like the Jacobi, Gauss-Seidel, and SOR methods. Examples of the non-stationary solvers include more sophisticated methods like the Krylov subspace methods: especially, Conjugate Gradient (CG) methods, Minimal Residual methods (especially GMRES), and many more.

■■■

Chapter Exercises

Exercise 01: Workout first three iterations of (i) the Jacobi method, (ii) the Gauss-Seidel method, and (iii) the Gauss-Seidel method with successive over-relaxation factor $\omega = 1.2$ and $\omega = 1.5$ for solving the following systems for any initial approximation. Perform computations with a precision of 4 decimal digits, at least. Assume the error tolerance as 0.0001.

(a)
$$\begin{array}{rcrcrcr} x_1 & - & 0.25x_2 & - & 0.25x_3 & = & 9 \\ -0.25x_1 & + & x_2 & - & 0.25x_3 & = & 4 \\ -0.25x_1 & - & 0.25x_2 & + & x_3 & = & -1 \end{array}$$

(b)
$$\begin{array}{rcrcrcrcr} 4x_1 & + & x_2 & - & x_3 & + & x_4 & = & 2.5 \\ x_1 & + & 4x_2 & - & x_3 & - & x_4 & = & 0.5 \\ -x_1 & - & x_2 & + & 5x_3 & + & x_4 & = & 5 \\ x_1 & - & x_2 & + & x_3 & + & 3x_4 & = & 4 \end{array}$$

(c)
$$\begin{array}{rcrcrcr} 2x_1 & - & x_2 & + & x_3 & = & -3 \\ 2x_1 & + & 4x_2 & + & 2x_3 & = & 8 \\ -x_1 & - & x_2 & + & 2x_3 & = & 1 \end{array}$$

(d)
$$\begin{array}{rcrcrcrcr} x_1 & - & 0.25x_2 & - & 0.25x_3 & + & 0x_4 & = & 11 \\ -0.25x_1 & + & x_2 & + & 0x_3 & - & 0.25x_4 & = & 7 \\ -0.25x_1 & + & 0x_2 & + & x_3 & - & 0.25x_4 & = & 3 \\ 0x_1 & - & 0.25x_2 & - & 0.25x_3 & + & x_4 & = & -1 \end{array}$$

(e)
$$\begin{array}{rcrcrcr} 0.2x_1 & + & 0.3x_2 & + & 0x_3 & = & 0.1 \\ 0.3x_1 & + & 0x_2 & + & 0.2x_3 & = & 0.1 \\ 0x_1 & + & 0.2x_2 & + & 0.3x_3 & = & 0.8 \end{array}$$

(f)
$$\begin{array}{rcrcrcrcr} 8x_1 & + & 4x_2 & + & 0x_3 & + & 0x_4 & = & 10 \\ 4x_1 & + & 12x_2 & + & 2x_3 & + & 0x_4 & = & 12 \\ 0x_1 & + & 2x_2 & + & 7x_3 & + & 2.5x_4 & = & 9.25 \\ 0x_1 & + & 0x_2 & + & 2.5x_3 & + & 4.5x_4 & = & 4.75 \end{array}$$

■■■

Chapter **8**

Eigenvalues and Eigenvectors

Corridor I: BASICS

Let's plan it

To unleash the topics of this Corridor, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

Corridor II: ANALYSIS

Let's think deep

To unleash the topics of this Corridor, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

Corridor III: PROGRAMMING ARCADE

Let's do it

8.5 Algorithms and Implementations
The Power Method

To see more examples for practicing, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

For codes, please visit: https://github.com/DrAmjadAli11/SimplifiedNumericalAnalysis

■■■

8.5 Algorithms and Implementations

Question 12: Write down an algorithm (pseudo code) to find dominant eigenvalue and a corresponding eigenvector of a matrix using the Power method.

Algorithm: To approximate the dominant eigenvalue and associated eigenvector of an $n \times n$ matrix A, given a nonzero normalized vector X (i.e., having 1 as the largest component) as the initial approximation.

INPUTS:
- $\boldsymbol{n}$: an integer as the length of the vector X
- $\boldsymbol{X} = [\boldsymbol{x_1}, \boldsymbol{x_2}, \cdots, \boldsymbol{x_n}]^{\boldsymbol{T}}$: a real valued vector (as a normalised initial approximation)
- $\boldsymbol{A} = (\boldsymbol{a_{ij}}), 1 \le i, j \le \boldsymbol{n}$: a real valued square matrix whose eigenvalue is to be obtained
- $\boldsymbol{TOL}$: a real value as the tolerance
- $\boldsymbol{N}$: an integer as the maximum number of iterations

OUTPUT:
- $\boldsymbol{B}$: a real value as the approximate eigenvalue
- $\boldsymbol{X} = [\boldsymbol{x_1}, \boldsymbol{x_2}, \cdots, \boldsymbol{x_n}]^{\boldsymbol{T}}$: a normalized vector as the eigenvector corresponding to $\boldsymbol{B}$

```
Step 1    Receive the inputs as stated above
Step 2    for $k = 1, 2, 3, \cdots, N$ perform steps 3-6
    Step 3    for $i = 1, 2, \cdots, n$  Set $xp_i = x_i$    {$XP = [xp_1, xp_2, \cdots, xp_n]^T$ is to keep a copy of present approximation $X$, because $X$ is going to be updated
Step 4      (Compute the vector such that $X^{(k)} = AX^{(k-1)}$)

        for $i = 1, 2, \cdots, \boldsymbol{n}$
            sum = 0
            for $j = 1, 2, \cdots, \boldsymbol{n}$
                sum = sum + $\boldsymbol{a_{ij}} \times \boldsymbol{xp_j}$
            $x_i$ = sum                                   $\left( x_i^{(k)} = \sum_{j=1}^{n} a_{ij}\, x_j^{(k-1)} \right)$

Step 5      (Approximate the eigenvalue $\boldsymbol{B}$ and normalize the vector $\boldsymbol{X}$)

set $r = 1$
for $i = 1, 2, \cdots, \boldsymbol{n}$
    if $(|x_i| > |x_r|)$  $r = i$                (Finding the element of $X$ with the largest absolute value and then setting it as $\boldsymbol{B}$)
set $\boldsymbol{B} = x_r$
        for $i = 1, 2, \cdots, \boldsymbol{n}$
            $x_i = x_i/\boldsymbol{B}$                              (Normalizing the vector $X$)
Step 6

      if ($\boldsymbol{err} < \boldsymbol{TOL}$ )then        This means that the consecutive
            Exit/Break the loop          approximations are nearly the same.
                                         Therefore, stop iterations.

end for loop of Step 2          (Go to Step 3)

Step 9      Print the output: eigenvalue $\boldsymbol{B}$, and eigenvector $\boldsymbol{X} = [\boldsymbol{x_1}, \boldsymbol{x_2}, \cdots, \boldsymbol{x_n}]^{\boldsymbol{T}}$

      if ($\boldsymbol{err} < \boldsymbol{TOL}$)     OUTPUT ('The desired accuracy achieved; Solution converged.')
      else                  OUTPUT ('The number of iterations exceeded the maximum limit.')
STOP.
```

Question 13: Write a Python program to find the dominant eigenvalue of the following matrix using the Power method. For simplification, specify the matrix within the program. Take $X^{(0)} = [1, \quad 1, \quad 1]^T$ as the initial approximation. The iterations of the method should stop when either the approximation is accurate within 10^{-5}, or the number of iterations exceeds 100, whichever happens first.

$$A = \begin{bmatrix} 4 & 1 & 0 \\ 2 & 5 & 0 \\ 7 & 2 & 1 \end{bmatrix}$$

script_8.1: power.ipynb

```python
from numpy import *

N = 100
TOL = 0.00001
n = 3
a = [[4, 1, 0], [2, 5, 0], [7, 2, 1]]
x = [1, 1, 1]

xp = zeros(n)
print("iter.      x1         x2         x3            Error")

```

```
for k in range(1,N+1):
        print(k, end=" ")
        for i in range(n):
                xp[i] = x[i]
        for i in range(n):
                sum = 0
                for j in range(n):
                        sum = sum + a[i][j] * xp[j]
                x[i] = sum

        r = 0
        for i in range(n):
                if abs(x[i]) > abs(x[r]):
                        r = i

        B = x[r]
        for i in range(n):
                x[i] = x[i] / B
                print("      ", round(x[i], 8), end=" ")

#Computing the error as L2-Norm
        sum1 = 0
        for i in range(n):
                sum1 = sum1 + (x[i] – xp[i]) * (x[i] – xp[i])
        err = sqrt(sum1)
        print("     ",round(err,8))
        if err < TOL:
                break

print("The approximate dominant eigenvalue is", B)
print("The corresponding eigenvector is:")

for i in range(n):
        print(x[i], "    ",end=" ")

if err < TOL:
        print("\nThe desire accuracy is achieved; Solution convergent")
else:
        print("\nThe number of iteration exceeded the maximum limit")
```

Output Console:

```
iter.      x1              x2                  x3           Error
1        0.51            0.71               1.0          0.58309519
2        0.457627122     0.762711862        1.0          0.07568513
3        0.452662723     0.825443783        1.0          0.06292805
4        0.452974074     0.864768684        1.0          0.03932613
5        0.453644675     0.886351895        1.0          0.02159363
6        0.454074026     0.897588186        1.0          0.01124449
```

```
7        0.454305837     0.903308497       1.0          0.00572501
8        0.45442498      0.906192358       1.0          0.00288632
9        0.454485039     0.907639869       1.0          0.00144876
10       0.4545152110    0.9083649610      1.0          0.00072573
11       0.4545303311    0.9087278311      1.0          0.00036318
12       0.4545378912    0.9089093412      1.0          0.00018167
13       0.4545416713    0.9090001213      1.0          9.086e-05
14       0.4545435614    0.9090455114      1.0          4.543e-05
15       0.4545445115    0.9090682115      1.0          2.272e-05
16       0.4545449816    0.9090795616      1.0          1.136e-05
17       0.4545452217    0.9090852417      1.0          5.69e-06
The approximate dominant eigenvalue is 5.99997398
The corresponding eigenvector is:
0.45454522     0.90908524     1.0
The desired accuracy achieved; Solution converged
```

Remark: The Python program in Question 13 can be modified to receive the square matrix and the initial approximation of the Eigenvector at the execution time (instead of fixing in the code). For this, the code segment at lines 6 and 7 in the program of Question 13 should be replaced by the following code segment:

```
# Initialize the matrix 'a' and initial approximation 'b' as empty lists
a = []

print("Enter the matrix row-wise:")
for i in range(n):
    row = list(map(float, input().split()))
    a.append(row)

# Convert the 'a' list into a NumPy array
a = np.array(a)

b = np.empty(n, dtype=float)

print("Enter the initial approximation:")
b = np.array(list(map(float, input().split())))

# Now, you can use the 'a' matrix and 'b' vector in your Python code as needed.
```

■■■

Chapter Summary

- An ***eigenvalue*** of a square matrix $A = (a_{ij})_{n\times n}$ is a number λ such that the vector equation
$$AX = \lambda X$$
- has a non-zero solution vector X. The solution vector X is then called an ***eigenvector*** of the matrix A corresponding to the eigenvalue λ. The set of all eigenvalues of a matrix is called the ***spectrum*** of the matrix. An eigenvalue is also called a *characteristic value* or *latent root*. Likewise, an eigenvector is also called a *characteristic vector* or *latent vector*.

- A concise account of the results and techniques relevant to the eigenvalues and eigenvectors is given in Section 8.1.

- The theorem of the Power method: Suppose that an $n \times n$ matrix A has n eigenvalues $\lambda_1, \lambda_2, \cdots, \lambda_n$ and associated n linearly independent eigenvectors, $V_1, V_2, \cdots, V_n$. Further, suppose that $X^{(0)}$ is a normalized vector (i.e., a vector having maximum absolute value as 1) in the space of the said eigenvectors. The sequence of normalized vectors $\{X^{(k)}\}_{k=1}^{\infty}$ and the sequence of scalars $\{\beta_k\}_{k=1}^{\infty}$ generated recursively by
$$X^{(k)} = \frac{1}{\beta_k}Y^{(k)},$$
$$\text{where} \quad Y^{(k)} = AX^{(k-1)}, \quad \text{and} \quad \beta_k = y_r^{(k)} \quad \text{such that} \quad \left|y_r^{(k)}\right| = \left\|Y^{(k)}\right\|_{\infty},$$
will converge to the dominant eigenvector and eigenvalue, respectively.

- In the Power method, both the sequences of the scalars $\{\beta_k\}_{k=1}^{\infty}$ and the normalized vectors $\{X^{(k)}\}_{k=1}^{\infty}$ converge linearly to the dominant eigenvalue λ_1 and a corresponding eigenvector V_1, respectively. Thus, the order of convergence of the Power method is linear.

- Aitken's Δ^2 method offers a technique for accelerating the convergence of any sequence that is linearly convergent. Using a given sequence, say $\{\beta_k\}_{k=1}^{\infty}$, which converges linearly to λ_1, another sequence $\{\hat{\beta}_k\}_{k=1}^{\infty}$ (that also converges to λ_1 with possibly improved convergence rate) is constructed by using the Aitken's Δ^2 process as:
$$\hat{\beta}_k = \beta_k - \frac{(\beta_{k+1}-\beta_k)^2}{\beta_{k+2}-2\beta_{k+1}+\beta_k} = \beta_k - \frac{(\Delta\beta_k)^2}{\Delta^2\beta_k}, \qquad \text{for } k = 0, 1, 2, \cdots$$

- Suppose that λ is a non-zero eigenvalue of a square matrix A and X is an eigenvector corresponding to λ. Then, $1/\lambda$ is an eigenvalue of A^{-1} and the same X is an eigenvector corresponding to $1/\lambda$. Thus, the reciprocal of all the non-zero eigenvalues of a square matrix A are the eigenvalues of A^{-1} (having the same set of eigenvectors). Hence, the largest of the absolute eigenvalues of A is the smallest of the eigenvalues of A^{-1} (and vice-versa). Thus, the Power method can be used to obtain the largest eigenvalue of A^{-1} and then taking its reciprocal gives the smallest eigenvalue of A.

■■■

Chapter Exercises

Exercise 01: Find all the eigenvalues and eigenvectors of the following matrices using the characteristic equations. Also find the spectrum, spectral radius, trace, and determinant of the given matrix.

(i) $\begin{bmatrix} 3 & 2 & -1 \\ 2 & 6 & 4 \\ -1 & 4 & 5 \end{bmatrix}$ (ii) $\begin{bmatrix} 3 & -2 & 0.5 \\ -1 & -2 & 1.5 \\ -4 & 0 & 4 \end{bmatrix}$ (iii) $\begin{bmatrix} 2 & 0 & 0 \\ -6 & 8 & -14 \\ 0 & 0 & -6 \end{bmatrix}$

(iv) $\begin{bmatrix} -15.5 & -10 & 10 \\ 3 & 4.5 & -3 \\ -17 & -10 & 11.5 \end{bmatrix}$ (v) $\begin{bmatrix} 4.5 & 0 & 1.5 \\ -6 & 9 & 6 \\ 1.5 & 0 & 4.5 \end{bmatrix}$

Exercise 02: Apply the Power method to find the dominant eigenvalue and corresponding eigenvector of the given matrices.

(i) $\begin{bmatrix} 3 & 2 & -1 \\ 2 & 6 & 4 \\ -1 & 4 & 5 \end{bmatrix}$ (ii) $\begin{bmatrix} 3 & -2 & 0.5 \\ -1 & -2 & 1.5 \\ -4 & 0 & 4 \end{bmatrix}$ (iii) $\begin{bmatrix} 2 & 0 & 0 \\ -6 & 8 & -14 \\ 0 & 0 & -6 \end{bmatrix}$

(iv) $\begin{bmatrix} -15.5 & -10 & 10 \\ 3 & 4.5 & -3 \\ -17 & -10 & 11.5 \end{bmatrix}$ (v) $\begin{bmatrix} 4.5 & 0 & 1.5 \\ -6 & 9 & 6 \\ 1.5 & 0 & 4.5 \end{bmatrix}$

Exercise 03: Apply the Power method to find the dominant eigenvalue and corresponding eigenvector of the given matrices.

(i) $\begin{bmatrix} 8 & 1 & 0 & 0 \\ 0 & 7 & 0 & 0 \\ -2 & 1 & 10 & 0 \\ -4 & -1 & 4 & 6 \end{bmatrix}$ (ii) $\begin{bmatrix} 1 & 10 & 6 & -6 \\ 0 & -9 & 0 & 0 \\ -0.5 & 16.5 & 7.5 & 0.5 \\ -6.5 & 10.5 & 6.5 & 1.5 \end{bmatrix}$

Exercise 04: Use Householder's method to place the following matrices in tridiagonal form.

(i) $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$ (ii) $\begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}$

(iii) $\begin{bmatrix} 5 & -2 & -0.5 & 1.5 \\ -2 & 5 & 1.5 & -0.5 \\ -0.5 & 1.5 & 5 & -2 \\ 1.5 & -0.5 & -2 & 5 \end{bmatrix}$ (iv) $\begin{bmatrix} 2 & -1 & -1 & 0 \\ -1 & 3 & 0 & 0 \\ -1 & 0 & 4 & 1 \\ 0 & -2 & 2 & 3 \end{bmatrix}$

Exercise 05: Apply two iterations of the QR Factorization method without shifting the following matrices.

(i) $\begin{bmatrix} 4 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 2 \end{bmatrix}$ (ii) $\begin{bmatrix} 3 & 1 & 0 \\ 1 & 4 & 2 \\ 0 & 2 & 1 \end{bmatrix}$

(iii) $\begin{bmatrix} 4 & 2 & 0 & 0 \\ 2 & 4 & 2 & 0 \\ 0 & 2 & 4 & 2 \\ 0 & 0 & 2 & 4 \end{bmatrix}$ (iv) $\begin{bmatrix} 0.5 & 0.25 & 0 & 0 \\ 0.25 & 0.8 & 0.4 & 0 \\ 0 & 0.4 & 0.6 & 0.1 \\ 0 & 0 & 0.1 & 1 \end{bmatrix}$

Exercise 06: Determine a singular value decomposition for the following matrices.

(i) $\begin{bmatrix} 1 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix}$ (ii) $\begin{bmatrix} 2 & 1 \\ -1 & 1 \\ 1 & 1 \end{bmatrix}$

(iii) $\begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$ (iv) $\begin{bmatrix} 2 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$

■■■

Chapter **9**

Numerical Solution of Ordinary Differential Equations (ODEs)

Corridor I: BASICS

Let's plan it

To unleash the topics of this Corridor, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

■■■

Corridor II: ANALYSIS

Let's think deep

9.6 Some Theoretical Concepts and Error Analysis

To unleash the topics of this Corridor, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

■■■

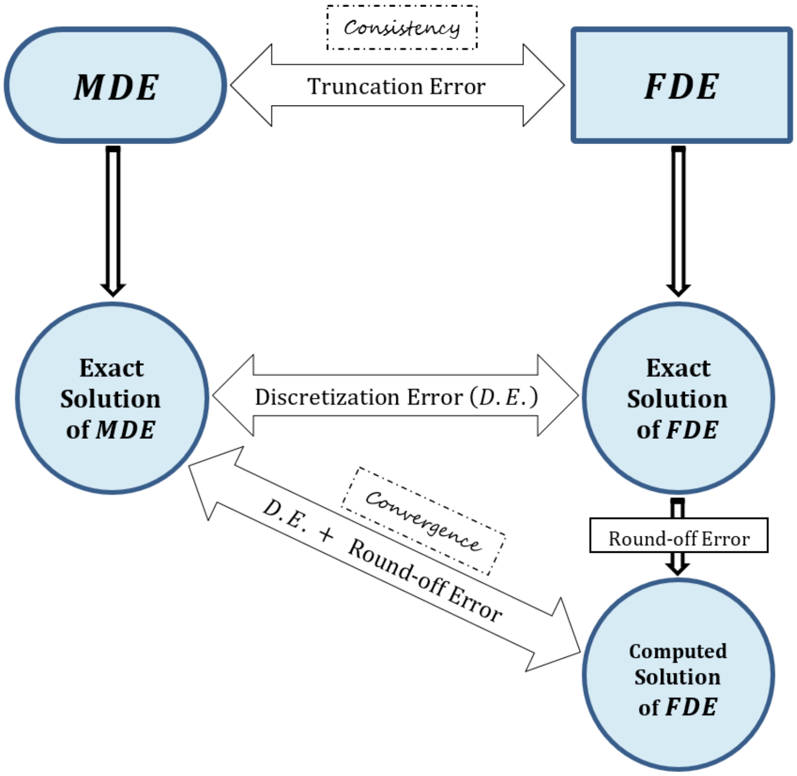


Figure: The connection between various terms related to MDE (Model Differential Equation/s - ODE/PDE) and the related FDE (Finite Difference Equation/s).

Corridor III: PROGRAMMING ARCADE

Let's think deep

To see more examples for practicing, please delve into the principal book:

Simplified Numerical Analysis (Fourth Edition; by Dr. Amjad Ali; www.TimeRenders.com.pk)

For codes, please visit: https://github.com/DrAmjadAli11/SimplifiedNumericalAnalysis

■■■

9.7 Algorithms and Implementations

Question 16: Write down an algorithm (pseudo code) to solve a first-order ODE using the Explicit Euler's method (the Taylor method of order 1).

Algorithm: To solve $y' = f(x, y)$, for $a \le x \le b$ and $y(a) = \alpha$ by approximating $y = y(x)$ at $(m + 1)$ equispaced nodes $x_0, x_1, x_2, \cdots, x_m$, such that $a = x_0 < x_1 < x_2 < \cdots < x_m = b$, $h = (b - a)/m$ and $y(x_i) = y_i$ using the Explicit Euler's method (the Taylor method of order 1): For $i = 1,2,3, \cdots, m$

$$w_i = w_{i-1} + h \times f(x_{i-1}, y_{i-1})$$

INPUTS: $\begin{cases} a, b\text{: real values as the endpoints of the interval: } x \in [a, b] \\ m\text{: an integer as the number of nodes (other than } a\text{) in the } x \text{ direction} \\ alpha\text{: a real value as the initial condition } y(a) \\ \text{A definition of the function } f(x, y) \text{ in an appropriate way} \end{cases}$

OUTPUT: $\begin{cases} W = [w_0, w_1, \cdots, w_m]^T\text{: a real valued vector as the approximate solution } w(x_i) \\ \text{at the nodes } x_0, x_1, x_2, \cdots, x_m \end{cases}$

Auxiliary Variables: $\begin{cases} h\text{: a real value as the step length in } x \text{ direction such that } h = (b - a)/m \\ x = (x_i) \text{ for } i = 0, 1, \cdots, m\text{: a real valued vector to represent } x_i s \end{cases}$

```
Step 1      Receive the inputs as stated above

Step 2      Set $h = (b - a)/m$
            Set $x(0) = a$
            Set $x(m) = b$

Step 3      for $i = 1, 2, \cdots, m - 1$
                Set $x(i) = x(0) + i \times h$         (Constructing interior mesh points, $x_i$ )
            end for

Step 4      Set $w(0) = alpha$                        (Setting the initial condition)

Step 5      for $i = 1, 2, \cdots, m$
                $fval = f(X(i - 1), w(i - 1))$         (Computing the value $f(x_{i-1}, y_{i-1})$ )
                $w(i) = w(i - 1) + h \times fval$
            end for

Step 6      Print the output: $W = [w_0, w_1, \cdots, w_m]^T$ ;

STOP.
```

Question 17: Write down a Python program to solve the IVP, $y' = 4y + 4x^2 + 3x$, for $0 \le x \le 1$, with initial condition $y(0) = \alpha = 1/2$, using the Explicit Euler's method (the Taylor method of order 1). Computer the solution for 10 steps. At each step, compare the approximate solution with the exact solution, to be obtained by $y(x) = -x^2 - \frac{5}{4}x - \frac{5}{16} + \frac{13}{16}e^{4x}$, by finding the relative error between the two solutions.

script_9.1: explicit_euler1.ipynb

```python
from numpy import *
```

```

a = 0.0                                  # starting point of domain
b = 1.0                                  # ending point of domain
alpha = 0.5                              # initial condition
m = 10                                   # number of steps

def fval(x, y):
        return 4 * y + 4 * x ** 2 + 3 * x

def fexact(x):
        return –x ** 2 – 1.2 * x – (5.0 / 16.0) + (13.0 / 16.0) * exp(4 * x)

h = (b – a) / m
x = zeros(m+1)
w = zeros(m+1)

x[0] = a
for i in range(1,m+1):
        x[i] = x[i–1] + h

w[0]= alpha                      # setting initial condition

#------ Computing solutions with the Euler method ------

for i in range(1,m+1):
        fv = fval(x[i–1], w[i–1])
        w[i] = w[i–1] + h*fv

# ----------------- Printing Solutions -----------------
print("Node       x[i]          w[i]         Exact Sol         Relative Error")
for i in range(0,m+1):
        sol = fexact(x[i])
        err = abs(sol–w[i]) / abs(sol)

        print(i,"\t","%.2f" %x[i],"\t","%.7f" %w[i],"\t","%.7f" %sol,"\t","%.8f" %err)
```

Output Console:

```
Node     x[i]        w[i]        Exact Sol         Relative Error
0         0.00    0.5000000        0.5000000         0.00000000
1         0.10    0.7000000        0.7696076         0.09044553
2         0.20    1.0140000        1.2157520         0.16594832
3         0.30    1.4956000        1.9350950         0.22711805
4         0.40    2.2198400        3.0718388         0.27735792
5         0.50    3.2917760        4.8411081         0.32003666
6         0.60    4.8584864        7.5638308         0.35766855
7         0.70    7.1258810        11.7187755        0.39192615
8         0.80    10.3822333       18.0201808        0.42385521
9         0.90    15.0311267       27.5335655        0.45407991
10        1.00    21.6375774       41.8484969        0.48295449
```

The above program can be written in a better way that a Python function for the Euler method is formed to compute the solution. This makes the program better manageable and modular. The new program is given as follows.

script_9.2: explicit_euler2.ipynb

```
from numpy import *

a = 0.0                                     # starting point of domain
b = 1.0                                     # ending point of domain
alpha = 0.5                                 # initial condition
m = 10                                      # number of steps

def fval(x, y):
        return 4 * y + 4 * x ** 2 + 3 * x

def fexact(x):
        return –x ** 2 –1.2 * x – (5.0 / 16.0) + (13.0 / 16.0) * exp(4 * x)

#----- User-defined function for the Euler's one-step -----

def euler(x, w, h):
        for i in range(1,m+1):
                fv = fval(x[i–1],w[i–1])
                w[i] = w[i–1] + h*fv

h = (b – a) / m
x = zeros(m+1)
w = zeros(m+1)

x[0] = a
x[m] = b
for i in range(1,m+1):
        x[i] = x[i–1] + h

w[0]= alpha                         # setting initial condition

#Call the Euler method function

euler(x, w, h)

# ----------------- Printing Solutions -----------------
print("Node       x[i]          w[i]        Exact Sol        Relative Error")
for i in range(0,m+1):
        sol = fexact(x[i])
        err = abs(sol–w[i]) / abs(sol)

        print(i,"\t","%.2f" %x[i],"\t","%.7f" %w[i],"\t","%.7f" %sol,"\t","%.8f" %err)
```

Question 18: Write down an algorithm (pseudo code) to solve a first-order ODE using the Midpoint method (which is an RK method of order 2).

Algorithm: To solve $y' = f(x,y)$, for $a \le x \le b$ and $y(a) = \alpha$ by approximating $y = y(x)$ at $(m+1)$ equispaced nodes $x_0, x_1, x_2, \cdots, x_m$, such that $a = x_0 < x_1 < x_2 < \cdots < x_m = b$, $h = (b-a)/m$ and $y(x_i) = y_i$ using the Midpoint method: For $i = 1,2,3,\cdots,m$

$$\begin{aligned}\bar{\bar{y}}_i &= y_{i-1} + \frac{h}{2} \times f(x_{i-1}, y_{i-1}) \\ y_i &= y_{i-1} + h \times f\left(x_{i-1} + \frac{h}{2}, \bar{\bar{y}}_i\right)\end{aligned}$$

INPUTS: $\begin{cases} a, b\text{: real values as the endpoints of the interval: } x \in [a,b] \\ m\text{: an integer as the number of nodes (other than } a\text{) in the } x \text{ direction} \\ alpha\text{: a real value as the initial condition } y(a) \\ \text{A definition of the function } f(x,y) \text{ in an appropriate way}\end{cases}$

OUTPUT: $\begin{cases} W = [w_0, w_1, \cdots, w_m]^T\text{: a real valued vector as the approximate solution } w(x_i) \\ \text{at the nodes } x_0, x_1, x_2, \cdots, x_m\end{cases}$

Auxiliary Variables: $\begin{cases} h\text{: a real value as the step length in } x \text{ direction such that } h = (b-a)/m \\ X = (x_i) \text{ for } i = 0, 1, \cdots, m\text{: a real valued vector to represent } x_i s\end{cases}$

```
Step 1    Receive the inputs as stated above

Step 2    Set $h = (b-a)/m$
          Set $x(0) = a$
          Set $x(m) = b$

Step 3    for $i = 1, 2, \cdots, m-1$
              Set $x(i) = x(0) + i \times h$          (Constructing interior mesh points, $x_i$ )
          end for

Step 4    Set $w(0) = alpha$                          (Setting the initial condition)

Step 5    for $i = 1, 2, \cdots, m$
              $fval1 = f(x(i-1), w(i-1))$             (Computing the value $f(x_{i-1}, y_{i-1})$)
              $aux = w(i-1) + (h/2) \times fval1$
              $fval2 = f(x(i-1) + (h/2), aux)$        (Computing $f(x_{i-1} + h/2, aux)$ )
              $w(i) = w(i-1) + h \times fval2$
          end for

Step 6    Print the output: $W = [w_0, w_1, \cdots, w_m]^T$ ; STOP.
```

Question 19: Write down a Python program to solve the IVP, $y' = 4y + 4x^2 + 3x$, for $0 \le x \le 1$, with initial condition $y(0) = \alpha = 1/2$, using the Midpoint method (which is an RK method of order 2). Computer the solution for 10 steps. At each step, compare the approximate solution with the exact solution, to be obtained by $y(x) = -x^2 - \frac{5}{4}x - \frac{5}{16} + \frac{13}{16}e^{4x}$, by finding the relative error between the two solutions.

script_9.3: euler_mid.ipynb

```
from numpy import *

a = 0.0                                  # starting point of domain
b = 1.0                                  # ending point of domain
alpha = 0.5                              # initial condition
m = 10                                   # number of steps

def fval(x, y):
        return 4 * y + 4 * x ** 2 + 3 * x

def fexact(x):
        return –x ** 2 – 1.2 * x – (5.0 / 16.0) + (13.0 / 16.0) * exp(4 * x)

#----- User-defined function for the Euler's midpoint -----

def eulermid(xa, wa, h):
        for i in range(1,m+1):
                fv = fval(xa[i – 1], wa[i – 1])
                whalf = wa[i –1] + (h / 2.0) * fv
                xhalf = xa[i – 1] + (h / 2.0)
                fv = fval(xhalf, whalf)
                wa[i] = wa[i – 1] + h * fv

h = (b – a) / m
x = zeros(m+1)
w = zeros(m+1)

x[0] = a
x[m] = b
for i in range(1,m+1):
        x[i] = x[i–1] + h

w[0]= alpha                         # setting initial condition

#Call the Euler midpoint method function

eulermid(x, w, h)

# ----------------- Printing Solutions -----------------
print("Node       x[i]           w[i]         Exact Sol         Relative Error")
for i in range(0,m+1):
        sol = fexact(x[i])
        err = abs(sol–w[i]) / abs(sol)

        print(i,"\t","%.2f" %x[i],"\t","%.7f" %w[i],"\t","%.7f" %sol,"\t","%.8f" %err)
```

Output Console:

```
Node       x[i]       w[i]          Exact Sol         Relative Error
0          0.00     0.5000000         0.5000000          0.00000000
1          0.10     0.7560000         0.7696076          0.01768118
2          0.20     1.1796800         1.2157520          0.02967053
3          0.30     1.8611264         1.9350950          0.03822479
4          0.40     2.9336671         3.0718388          0.04498015
5          0.50     4.5946273         4.8411081          0.05091413
6          0.60     7.1360484         7.5638308          0.05655632
7          0.70     10.9901516        11.7187755         0.06217577
8          0.80     16.7966243        18.0201808         0.06789923
9          0.90     25.5022040        27.5335655         0.07377764
10         1.00     38.5080619        41.8484969         0.07982210
```

Question 20: Write down an algorithm (pseudo code) to solve a first-order ODE using the RK method of order 2 (also known as the Modified or Improved Euler's method).

Algorithm: To solve $y' = f(x,y)$, for $a \le x \le b$ and $y(a) = \alpha$ by approximating $y = y(x)$ at $(m+1)$ equispaced nodes $x_0, x_1, x_2, \cdots, x_m$, such that $a = x_0 < x_1 < x_2 < \cdots < x_m = b$, $h = (b-a)/m$ and $y(x_i) = y_i$ using the Modified Euler's method of order 2: For $i = 1,2,3,\cdots,m$

$$\begin{aligned} K_1 &= h \times f(x_{i-1}, y_{i-1}) \\ K_2 &= h \times f(x_i, y_{i-1} + K_1) \\ y_i &= y_{i-1} + \frac{1}{2} \times [K_1 + K_2] \end{aligned}$$

INPUTS: $\begin{cases} a, b\text{: real values as the endpoints of the interval: } x \in [a,b] \\ m\text{: an integer as the number of nodes (other than } a\text{) in the } x \text{ direction} \\ alpha\text{: a real value as the initial condition } y(a) \\ \text{A definition of the function } f(x,y) \text{ in an appropriate way} \end{cases}$

OUTPUT: $\begin{cases} W = [w_0, w_1, \cdots, w_m]^T\text{: a real valued vector as the approximate solution } w(x_i) \\ \text{at the nodes } x_0, x_1, x_2, \cdots, x_m \end{cases}$

Auxiliary Variables: $\begin{cases} h\text{: a real value as the step length in } x \text{ direction such that } h = (b-a)/m \\ X = (x_i) \text{ for } i = 0, 1, \cdots, m\text{: a real valued vector to represent } x_i s \end{cases}$

```
Step 1        Receive the inputs as stated above

Step 2        Set $h = (b-a)/m$
              Set $x(0) = a$
              Set $x(m) = b$

Step 3        for $i = 1, 2, \cdots, m-1$
                  Set $x(i) = x(0) + i \times h$          (Constructing interior mesh points, $x_i$ )
              end for
```

```
Step 4    Set $w(0) = alpha$                    (Setting the initial condition)

Step 5    for $i = 1, 2, \cdots, m$
              $k1 = h \times f\big(x(i-1), w(i-1)\big)$
              $k2 = h \times f(x(i), w(i-1) + k1)$
              $w(i) = w(i-1) + 0.5 \times (k1 + k2)$
          end for

Step 6    Print the output: $W = [w_0, w_1, \cdots, w_m]^T$

STOP.
```

Question 21: Write down a Python program to solve the IVP, $y' = 4y + 4x^2 + 3x$, for $0 \leq x \leq 1$, with initial condition $y(0) = \alpha = 1/2$, using the RK method of order 2 (also known as the Modified or Improved Euler's method). Computer the solution for 10 steps. At each step, compare the approximate solution with the exact solution, to be obtained by $y(x) = -x^2 - \frac{5}{4}x - \frac{5}{16} + \frac{13}{16}e^{4x}$, by finding the relative error between the two solutions.

script_9.4: modified_euler_rk2.ipynb

```python
from numpy import *

a = 0.0                                   # starting point of domain
b = 1.0                                   # ending point of domain
alpha = 0.5                               # initial condition
m = 10                                    # number of steps

def fval(x, y):
        return 4 * y + 4 * x ** 2 + 3 * x

def fexact(x):
        return –x ** 2 – 1.2 * x – (5.0 / 16.0) + (13.0 / 16.0) * exp(4 * x)

#----- User-defined function for the modified Euler method-----

def eulerimp(xa, wa, h):
        for i in range(1,m+1):
                fv = fval(xa[i – 1], wa[i – 1])
                wnext = wa[i – 1] + h * fv
                fvnext = fval(xa[i], wnext)
                wa[i] = wa[i – 1] + h * (fv + fvnext) / 2.0

h = (b – a) / m
x = zeros(m+1)
w = zeros(m+1)

x[0] = a
```

```
x[m] = b
for i in range(1,m+1):
        x[i] = x[i–1] + h

w[0]= alpha                       # setting initial condition

#Call the improved Euler (RK2) method function

eulerimp(x, w, h)

# ----------------- Printing Solutions -----------------
print("Node      x[i]       w[i]        Exact Sol        Relative Error")
for i in range(0,m+1):
        sol = fexact(x[i])
        err = abs(sol–w[i]) / abs(sol)

        print(i,"\t","%.2f" %x[i],"\t","%.7f" %w[i],"\t","%.7f" %sol,"\t","%.8f" %err)
```

Output Console:

```
Node      x[i]          w[i]        Exact Sol          Relative Error
0          0.00     0.5000000       0.5000000          0.00000000
1          0.10     0.7570000       0.7696076          0.01638181
2          0.20     1.1821600       1.2157520          0.02763064
3          0.30     1.8657968       1.9350950          0.03581126
4          0.40     2.9415793       3.0718388          0.04240443
5          0.50     4.6073373       4.8411081          0.04828869
6          0.60     7.1558592       7.5638308          0.05393716
7          0.70     11.0204716      11.7187755         0.05958847
8          0.80     16.8424980      18.0201808         0.06535355
9          0.90     25.5710971      27.5335655         0.07127549
10         1.00     38.6110237      41.8484969         0.07736176
```

■

Question 22: Write down an algorithm (pseudo code) to solve a first-order ODE using the RK method of order 4.

Algorithm: To solve $y' = f(x,y)$, for $a \le x \le b$ and $y(a) = \alpha$ by approximating $y = y(x)$ at $(m+1)$ equispaced nodes $x_0, x_1, x_2, \cdots, x_m$, such that $a = x_0 < x_1 < x_2 < \cdots < x_m = b$, $h = (b-a)/m$ and $y(x_i) = y_i$ using the RK method of order 4: For $i = 1,2,3,\cdots,m$

$$\begin{aligned}
K_1 &= h \times f(x_{i-1}, y_{i-1}) \\
K_2 &= h \times f(x_{i-1} + 0.5h, y_{i-1} + 0.5K_1) \\
K_3 &= h \times f(x_{i-1} + 0.5h, y_{i-1} + 0.5K_2) \\
K_4 &= h \times f(x_i, y_{i-1} + K_3) \\
y_i &= y_{i-1} + \frac{1}{6} \times [K_1 + 2K_2 + 2K_3 + K_4]
\end{aligned}$$

INPUTS: $\begin{cases} a, b\text{: real values as the endpoints of the interval: } x \in [a, b] \\ m\text{: an integer as the number of nodes (other than } a\text{) in the } x \text{ direction} \\ alpha\text{: a real value as the initial condition } y(a) \\ \text{A definition of the function } f(x, y) \text{ in an appropriate way} \end{cases}$

OUTPUT: $\begin{cases} W = [w_0, w_1, \cdots, w_m]^T\text{: a real valued vector as the approximate solution } y(x_i) \\ \text{at the nodes } x_0, x_1, x_2, \cdots, x_m \end{cases}$

Auxiliary Variables: $\begin{cases} h\text{: a real value as the step length in } x \text{ direction such that } h = (b-a)/m \\ X = (x_i) \text{ for } i = 0, 1, \cdots, m\text{: a real valued vector to represent } x_i s \end{cases}$

```
Step 1    Receive the inputs as stated above

Step 2    Set $h = (b - a)/m$
          Set $x(0) = a$
          Set $x(m) = b$

Step 3    for $i = 1, 2, \cdots, m - 1$
              Set $x(i) = x(0) + i \times h$          (Constructing interior mesh points, $x_i$ )
          end for

Step 4    Set $w(0) = alpha$                        (Setting the initial condition)

Step 5    for $i = 1, 2, \cdots, m$
              $k1 = h \times f(x(i-1), w(i-1))$
              $k2 = h \times f(x(i-1) + 0.5 \times h, w(i-1) + 0.5 \times k1)$
              $k3 = h \times f(x(i-1) + 0.5 \times h, w(i-1) + 0.5 \times k2)$
              $k4 = h \times f(x(i), w(i-1) + k3)$
              $w(i) = w(i-1) + (k1 + 2 \times k2 + 2 \times k3 + k4)/6$
          end for

Step 6    Print the output: $W = [w_0, w_1, \cdots, w_m]^T$

STOP.
```

Question 23: Write down a Python program to solve the IVP, $y' = 4y + 4x^2 + 3x$, for $0 \le x \le 1$, with initial condition $y(0) = \alpha = 1/2$, using the RK method of order 4. Computer the solution for 10 steps. At each step, compare the approximate solution with the exact solution, to be obtained by $y(x) = -x^2 - \frac{5}{4}x - \frac{5}{16} + \frac{13}{16}e^{4x}$, by finding the relative error between the two solutions.

script_9.5: RK4.ipynb

```
from numpy import *

a = 0.0                          # starting point of domain
b = 1.0                          # ending point of domain
alpha = 0.5                      # initial condition
m = 10                           # number of steps

```

```
def fval(x, y):
        return 4 * y + 4 * x ** 2 + 3 * x

def fexact(x):
        return –x ** 2 – 1.2 * x – (5.0 / 16.0) + (13.0 / 16.0) * exp(4 * x)

#----- User-defined function for the RK4 method-----

def rk4(x, w, h):
        for i in range(1,m+1):
                k1 = h * (fval(x[i – 1], w[i – 1]))
                k2 = h * (fval(x[i –1] + (h / 2.0), w[i – 1] + (k1 / 2.0)))
                k3 = h * (fval(x[i – 1] + (h / 2.0), w[i – 1] + (k2 / 2.0)))
                k4 = h * (fval(x[i], w[i – 1] + k3))

                w[i] = w[i – 1] + (k1 + 2 * k2 + 2 * k3 + k4) / 6.0

h = (b – a) / m
x = zeros(m+1)
w = zeros(m+1)

x[0] = a
x[m] = b
for i in range(1,m+1):
        x[i] = x[i–1] + h

w[0]= alpha                        # setting initial condition

#Call the RK4 method function

rk4(x, w, h)

# ----------------- Printing Solutions -----------------
print("Node       x[i]        w[i]         Exact Sol         Relative Error")
for i in range(0,m+1):
        sol = fexact(x[i])
        err = abs(sol–w[i]) / abs(sol)

        print(i,"\t","%.2f" %x[i],"\t","%.7f" %w[i],"\t","%.7f" %sol,"\t","%.8f" %err)
```

Output Console:

```
Node      x[i]       w[i]          Exact Sol          Relative Error
0        0.00     0.5000000       0.5000000          0.00000000
1        0.10     0.7645467       0.7696076          0.00657595
2        0.20     1.2055637       1.2157520          0.00838021
3        0.30     1.9196623       1.9350950          0.00797517
4        0.40     3.0509602       3.0718388          0.00679678
5        0.50     4.8144431       4.8411081          0.00550804
6        0.60     7.5308119       7.5638308          0.00436537
7        0.70     11.6784671      11.7187755         0.00343964
```

```
8          0.80     17.9710547         18.0201808         0.00272617
9          0.90     27.4731440         27.5335655         0.00219447
10         1.00     41.7727886         41.8484969         0.00180910
```

Question 24: Write down an algorithm (pseudo code) to solve a first-order ODE using the Adams-Bashforth method of order 4.

Algorithm: To solve $y' = f(x,y)$, for $a \le x \le b$ and $y(a) = \alpha$ by approximating $y = y(x)$ at $(m+1)$ equispaced nodes $x_0, x_1, x_2, \cdots, x_m$,such that $a = x_0 < x_1 < x_2 < \cdots < x_m = b, h = (b-a)/m$ and $y(x_i) = y_i$. Having $y(x_0) = \alpha_0$, $y(x_1) = \alpha_1$, $y(x_2) = \alpha_2$, and $y(x_3) = \alpha_3$, compute y_i using the 4-step explicit Adams-Bashforth method of order 4: For $i = 4,5,6,\cdots,m$

$$y_i \quad = \quad y_{i-1} + \frac{h}{24} \times [55f(x_{i-1}, y_{i-1}) - 59f(x_{i-2}, y_{i-2}) + 37f(x_{i-3}, y_{i-3}) - 9f(x_{i-4}, y_{i-4})]$$

INPUTS: $\begin{cases} a, b\text{: real values as the endpoints of the interval: } x \in [a,b] \\ m\text{: an integer as the number of nodes (other than } a\text{) in the } x \text{ direction} \\ alpha\text{: a real value as the initial condition } y(a) \\ \text{A definition of the function } f(x,y) \text{ in an appropriate way} \end{cases}$

OUTPUT: $\begin{cases} W = [w_0, w_1, \cdots, w_m]^T\text{: a real valued vector as the approximate solution } w(x_i) \\ \text{at the nodes } x_0, x_1, x_2, \cdots, x_m \end{cases}$

Auxiliary Variables: $\begin{cases} h\text{: a real value as the step length in } x \text{ direction such that } h = (b-a)/m \\ X = (x_i) \text{ for } i = 0, 1, \cdots, m\text{: a real valued vector to represent } x_i s \end{cases}$

```
Step 1    Receive the inputs as stated above

Step 2    Set $h = (b-a)/m$
          Set $x(0) = a$
          Set $x(m) = b$

Step 3    for $i = 1, 2, \cdots, m-1$
              Set $x(i) = x(0) + i \times h$          (Constructing interior mesh points, $x_i$ )
          end for

Step 4    Set $w(0) = alpha$                          (Setting the initial condition)

Step 5    Obtain or compute (using some other basic method for ODEs) the following:

          $w(1) = alpha1$
          $w(2) = alpha2$
          $w(3) = alpha3$

Step 5    for $i = 4, 5, 6, \cdots, m$
            $fv1 = f(x(i-1), w(i-1))$          (Computing the value $f(x_{i-1}, y_{i-1})$)
            $fv2 = f(x(i-2), w(i-2))$          (Computing the value $f(x_{i-2}, y_{i-2})$)
            $fv3 = f(x(i-3), w(i-3))$          (Computing the value $f(x_{i-3}, y_{i-3})$)
```

```
        $fv4 = f(x(i-4), w(i-4))$        (Computing the value $f(x_{i-4}, y_{i-4})$)
        $w(i) = w(i-1) + \left(\frac{h}{24}\right) \times (55fv1 - 59fv2 + 37fv3 - 9fv4)$
    end for
Step 6    Print the output: $W = [w_0, w_1, \cdots, w_m]^T$
STOP.
```

Question 25: Write down a Python program to solve the IVP, $y' = 4y + 4x^2 + 3x$, for $0 \leq x \leq 1$, with initial condition $y(0) = \alpha = 1/2$, using the Adams-Bashforth method of order 4. Compute the solution for 10 steps. For computing the approximate solution at the first three steps, use the RK4 method. At each step, compare the approximate solution with the exact solution, to be obtained by $y(x) = -x^2 - \frac{5}{4}x - \frac{5}{16} + \frac{13}{16}e^{4x}$, by finding the relative error between the two solutions.

script_9.6: adam_bashforth.ipynb

```python
from numpy import *

a = 0.0                                   # starting point of domain
b = 1.0                                   # ending point of domain
alpha = 0.5                               # initial condition
m = 10                                    # number of steps

def fval(x, y):
        return 4 * y + 4 * x ** 2 + 3 * x

def fexact(x):
        return -x ** 2 - 1.2 * x - (5.0 / 16.0) + (13.0 / 16.0) * exp(4 * x)

#----- User-defined function for the adam bashforth method-----

def adamsb4(x, w, h):
        for i in range(4, m + 1):
                k1 = fval(x[i - 1], w[i - 1])
                k2 = fval(x[i - 2], w[i - 2])
                k3 = fval(x[i - 3], w[i - 3])
                k4 = fval(x[i - 4], w[i - 4])

                w[i] = w[i - 1] + (h / 24.0) * (55 * k1 - 59 * k2 + 37 * k3 - 9 * k4)

#----- User-defined function for the RK4 method-----

def rk4(x, w, h):
        for i in range(1,4):
                k1 = h * (fval(x[i - 1], w[i - 1]))
        k2 = h * (fval(x[i – 1] + (h / 2.0), w[i – 1] + (k1 / 2.0)))
        k3 = h * (fval(x[i – 1] + (h / 2.0), w[i – 1] + (k2 / 2.0)))
        k4 = h * (fval(x[i], w[i – 1] + k3))

        w[i] = w[i – 1] + (k1 + 2 * k2 + 2 * k3 + k4) / 6.0

h = (b – a) / m
x = zeros(m+1)
w = zeros(m+1)

x[0] = a
x[m] = b
for i in range(1,m+1):
    x[i] = x[i–1] + h

w[0]= alpha                 # setting initial condition

# Using RK4 as initial steps

rk4(x, w, h)

#Call the adam bashforth function

adamsb4(x, w, h)

# ----------------- Printing Solutions -----------------
print("Node     x[i]     w[i]      Exact Sol      Relative Error")
for i in range(0,m+1):
    sol = fexact(x[i])
    err = abs(sol – w[i]) / abs(sol)

    print(i,"\t","%.2f" %x[i],"\t","%.7f" %w[i],"\t","%.7f" %sol,"\t","%.8f" %err)
```

Output Console:

```
Node     x[i]       w[i]         Exact Sol          Relative Error
0         0.00     0.5000000      0.5000000          0.00000000
1         0.10     0.7645467      0.7696076          0.00657595
2         0.20     1.2055637      1.2157520          0.00838021
3         0.30     1.9196623      1.9350950          0.00797517
4         0.40     3.0446855      3.0718388          0.00883945
5         0.50     4.7930616      4.8411081          0.00992469
6         0.60     7.4820511      7.5638308          0.01081194
7         0.70     11.5813609     11.7187755         0.01172602
8         0.80     17.7896098     18.0201808         0.01279515
9         0.90     27.1477196     27.5335655         0.01401365
10        1.00     41.2058778     41.8484969         0.01535585
```

Question 26: Write down an algorithm (pseudo code) to solve a first-order ODE using the Adams-Bashforth-Moulton method of order 4.

Algorithm: To solve $y' = f(x, y)$, for $a \le x \le b$ and $y(a) = \alpha$ by approximating $y = y(x)$ at $(m+1)$ equispaced nodes $x_0, x_1, x_2, \cdots, x_m$,such that $a = x_0 < x_1 < x_2 < \cdots < x_m = b, h = (b-a)/m$ and $y(x_i) = y_i$. Having $y(x_0) = \alpha_0, y(x_1) = \alpha_1, y(x_2) = \alpha_2$, and $y(x_3) = \alpha_3$, compute y_i using

(1) the 4-step explicit Adams-Bashforth method of order 4 as the predictor:

$$y_i = y_{i-1} + \frac{h}{24} \times [55f(x_{i-1}, y_{i-1}) - 59f(x_{i-2}, y_{i-2}) + 37f(x_{i-3}, y_{i-3}) - 9f(x_{i-4}, y_{i-4})]$$

(2) the 3-step implicit Adams-Moulton method of order 4 as the corrector:

$$y_i = y_{i-1} + \frac{h}{24} \times [9f(x_i, y_i) + 19f(x_{i-1}, y_{i-1}) - 5f(x_{i-2}, y_{i-2}) + f(x_{i-3}, y_{i-3})]$$

for $i = 4, 5, 6, \cdots, m$.

INPUTS: $\begin{cases} a, b\text{: real values as the endpoints of the interval: } x \in [a, b] \\ m\text{: an integer as the number of nodes (other than } a\text{) in the } x \text{ direction} \\ alpha\text{: a real value as the initial condition } y(a) \\ \text{A definition of the function } f(x, y) \text{ in an appropriate way} \end{cases}$

OUTPUT: $\begin{cases} W = [w_0, w_1, \cdots, w_m]^T\text{: a real valued vector as the approximate solution } y(x_i) \\ \text{at the nodes } x_0, x_1, x_2, \cdots, x_m \end{cases}$

Auxiliary Variables: $\begin{cases} h\text{: a real value as the step length in } x \text{ direction such that } h = (b-a)/m \\ X = (x_i) \text{ for } i = 0, 1, \cdots, m\text{: a real valued vector to represent } x_i s \end{cases}$

```
Step 1    Receive the inputs as stated above

Step 2    Set $h = (b-a)/m$
          Set $x(0) = a$
          Set $x(m) = b$

Step 3    for $i = 1, 2, \cdots, m-1$
              Set $x(i) = x(0) + i \times h$          (Constructing interior mesh points, $x_i$ )
          end for

Step 4    Set $w(0) = alpha$                          (Setting the initial condition)

Step 5    Obtain or compute (using some other basic method for ODEs) the following:

          $w(1) = alpha1$
          $w(2) = alpha2$
          $w(3) = alpha3$

Step 5    for $i = 4, 5, 6, \cdots, m$
            $fv1 = f(x(i-1), w(i-1))$           (Computing the value $f(x_{i-1}, y_{i-1})$)
            $fv2 = f(x(i-2), w(i-2))$           (Computing the value $f(x_{i-2}, y_{i-2})$)
```

```
        $fv3 = f(x(i-3), w(i-3))$        (Computing the value $f(x_{i-3}, y_{i-3})$)
        $fv4 = f(x(i-4), w(i-4))$        (Computing the value $f(x_{i-4}, y_{i-4})$)
        $w(i) = w(i-1) + \left(\frac{h}{24}\right) \times (55fv1 - 59fv2 + 37fv3 - 9fv4)$

        $fv = f(x(i), w(i))$             (Computing the value $f(x_{i-4}, y_{i-4})$)
        $w(i) = w(i-1) + \left(\frac{h}{24}\right) \times (9fv + 19fv1 - 5fv2 + fv3)$

    end for
```

Step 6 Print the output: $W = [w_0, w_1, \cdots, w_m]^T$

STOP.

Question 27: Write down a Python program to solve the IVP, $y' = 4y + 4x^2 + 3x$, for $0 \leq x \leq 1$, with initial condition $y(0) = \alpha = 1/2$, using the Adams-Bashforth-Moulton method of order 4. Compute the solution for 10 steps. For computing the approximate solution at the first three steps, use the RK4 method. At each step, compare the approximate solution with the exact solution, to be obtained by $y(x) = -x^2 - \frac{5}{4}x - \frac{5}{16} + \frac{13}{16}e^{4x}$, by finding the relative error between the two solutions.

script_9.8: adam_bashforth_molten.ipynb

```
from numpy import *

a = 0.0                                    # starting point of domain
b = 1.0                                    # ending point of domain
alpha = 0.5                                # initial condition
m = 10                                     # number of steps

def fval(x, y):
        return 4 * y + 4 * x ** 2 + 3 * x

def fexact(x):
        return –x ** 2 – 1.2 * x – (5.0 / 16.0) + (13.0 / 16.0) * exp(4 * x)

#----- User-defined function for the adam bashforth molten method-----

def adamsb4m3(x, w, h):
        for i in range(4, m + 1):
                fv1 = fval(x[i – 1], w[i – 1])
                fv2 = fval(x[i – 2], w[i – 2])
                fv3 = fval(x[i – 3], w[i – 3])
                fv4 = fval(x[i – 4], w[i – 4])

                w[i] = w[i – 1] + (h / 24.0) * (55 * fv1 – 59 * fv2 + 37 * fv3 – 9 * fv4)
                fv = fval(x[i], w[i])
                w[i] = w[i – 1] + (h / 24.0) * (9 * fv + 19 * fv1 – 5 * fv2 + fv3)

#----- User-defined function for the RK4 method-----
```

```

def rk4(x, w, h):
        for i in range(1,4):
                k1 = h * (fval(x[i – 1], w[i – 1]))
                k2 = h * (fval(x[i – 1] + (h / 2.0), w[i – 1] + (k1 / 2.0)))
                k3 = h * (fval(x[i – 1] + (h / 2.0), w[i – 1] + (k2 / 2.0)))
                k4 = h * (fval(x[i], w[i – 1] + k3))

                w[i] = w[i – 1] + (k1 + 2 * k2 + 2 * k3 + k4) / 6.0

h = (b – a) / m
x = zeros(m+1)
w = zeros(m+1)

x[0] = a
x[m] = b
for i in range(1,m+1):
        x[i] = x[i–1] + h

w[0]= alpha                        # setting initial condition

# Using RK4 as initial steps

rk4(x, w, h)

#Call the adam bashforth moulten function

adamsb4m3(x, w, h)

# ----------------- Printing Solutions -----------------
print("Node      x[i]       w[i]        Exact Sol        Relative Error")
for i in range(0,m+1):
        sol = fexact(x[i])
        err = abs(sol – w[i]) / abs(sol)

        print(i,"\t","%.2f" %x[i],"\t","%.7f" %w[i],"\t","%.7f" %sol,"\t","%.8f" %err)
```

Output Console:

```
Node      x[i]        w[i]           Exact Sol          Relative Error
0         0.00     0.5000000         0.5000000         0.00000000
1         0.10     0.7645467         0.7696076         0.00657595
2         0.20     1.2055637         1.2157520         0.00838021
3         0.30     1.9196623         1.9350950         0.00797517
4         0.40     3.0508703         3.0718388         0.00682605
5         0.50     4.8141708         4.8411081         0.00556428
6         0.60     7.5302111         7.5638308         0.00444480
7         0.70     11.6772868        11.7187755        0.00354036
8         0.80     17.9688762        18.0201808        0.00284706
9         0.90     27.4692778        27.5335655        0.00233489
10        1.00     41.7661082        41.8484969        0.00196874
```

Question 28: Write a Python program to solve the following system of two ODEs for the functions $y_1 = y_1(x)$ and $y_2 = y_2(x)$, where $x \in [0,1]$:

$$y_1' = y_1 y_2 - 2$$
$$y_2' = 2y_1 - y_2^3$$

With initial conditions:

$$y_1(0) = 2.0$$
$$y_2(0) = 0.3$$

Use the RK4 method of order 4 for 5 steps.

For 5 steps the domain is discretized as

$$h = \frac{b-a}{m} = \frac{1.0-0.0}{5} = 0.2$$

$$x_0 = 0,\ \ x_1 = 0.2, x_2 = 0.4, x_3 = 0.6, x_4 = 0.8, x_5 = 1.0.$$

According to the initial conditions:

$$w_{10} = y_{10} = y_1(x_0) = y_1(0) = 2.0$$
$$w_{20} = y_{20} = y_2(x_0) = y_2(0) = 0.3$$

The problem is to find approximations w_{1i} to $y_{1i} = y_1(x_i)$ and w_{2i} to $y_{2i} = y_2(x_i)$, for $i = 1,2,3,4,5$.

The Python program for the solution is as follows.

script_9.9: ode_system2.ipynb

```
from numpy import *

a = 0.0                                  # starting point of domain
b = 1.0                                  # ending point of domain
alpha1 = 2.0                             # initial condition
alpha2 = 0.3
m = 5                                    # number of steps

def f1(x, y1, y2):
        return y1 * y2 – 2

def f2(x, y1, y2):
        return 2 * y1 – y2 ** 3

# Define the RK4 solver for the ODE system of two equations

def rk4system2(x, w1, w2, h):
        for i in range(1, m + 1):
                k11 = h * f1(x[i –1], w1[i – 1], w2[i – 1])
                k21 = h * f2(x[i – 1], w1[i – 1], w2[i – 1])

        k12 = h * f1(x[i – 1] + 0.5 * h, w1[i – 1] + 0.5 * k11, w2[i – 1] + 0.5 * k21)

        k22 = h * f2(x[i – 1] + 0.5 * h, w1[i – 1] + 0.5 * k11, w2[i –1] + 0.5 * k21)

        k13 = h * f1(x[i – 1] + 0.5 * h, w1[i – 1] + 0.5 * k12, w2[i – 1] + 0.5 * k22)
        k23 = h * f2(x[i – 1] + 0.5 * h, w1[i – 1] + 0.5 * k12, w2[i – 1] + 0.5 * k22)

        k14 = h * f1(x[i – 1] + h, w1[i – 1] + k13, w2[i – 1] + k23)
        k24 = h * f2(x[i – 1] + h, w1[i – 1] + k13, w2[i – 1] + k23)

        w1[i] = w1[i – 1] + (k11 + 2 * k12 + 2 * k13 + k14) / 6.0
        w2[i] = w2[i – 1] + (k21 + 2 * k22 + 2 * k23 + k24) / 6.0

h = (b – a) / m
x = zeros(m+1)
w1 = zeros(m+1)
w2 = zeros(m+1)
w1[0] = alpha1                  # setting initial condition
w2[0] = alpha2

x[0] = a
x[m] = b

for i in range(1, m):
    x[i] = x[i – 1] + h

# Call the RK4 solver

rk4system2(x, w1, w2, h)

# ----------------- Printing Solutions -----------------
print("Node      x[i]       w1[i]       w2[i]")
for i in range(0,m+1):
    print(i,"\t","%.2f" %x[i],"\t","%.7f" %w1[i],"\t","%.7f" % w2[i])
```

Output Console:

```
Node       x[i]        w1[i]          w2[i]
0        0.00     2.0000000        0.3000000
1        0.20     1.8513219        0.9855220
2        0.40     1.9007946        1.3648472
3        0.60     2.0806503        1.5257072
4        0.80     2.3825142        1.6230648
5        1.00     2.8538285        1.7239291
```

■

Question 29: Write a Python program to solve the following system of three ODEs for the functions $y_1 = y_1(x)$, $y_2 = y_2(x)$, and $y_3 = y_3(x)$, where $x \in [0,1]$:

$$y_1' = y_1 + 3y_2 - 3y_3 + e^{-x}$$

$$y_2' = 2y_2 + y_3 - 3e^{-x}$$

$$y_3' = y_1 + 2y_2 + e^{-x}$$

With initial conditions:

$$y_1(0) = 2.5$$

$$y_2(0) = -1.5$$

$$y_3(0) = -1.0$$

Use the RK4 method of order 4 for 10 steps.

For 10 steps, the domain is discretized as

$$h = \frac{b-a}{m} = \frac{1.0 - 0.0}{10} = 0.1$$

$x_0 = 0, x_1 = 0.1, x_2 = 0.2, x_3 = 0.3, x_4 = 0.4, x_5 = 0.5, x_6 = 0.6, x_7 = 0.7, x_8 = 0.8, x_9 = 0.9, x_{10} = 1.0.$

According to the initial conditions:

$$w_{10} = y_{10} = y_1(x_0) = y_1(0) = 2.5$$

$$w_{20} = y_{20} = y_2(x_0) = y_2(0) = -1.5$$

$$w_{30} = y_{30} = y_3(x_0) = y_3(0) = -1.0$$

The problem is to find approximations w_{1i} to $y_{1i} = y_1(x_i)$, w_{2i} to $y_{2i} = y_2(x_i)$, and w_{3i} to $y_{3i} = y_3(x_i)$, for $i = 1,2,\cdots,10$.

The Python program for the solution is as follows.

script_9.10: ode_system3.ipynb

```
from numpy import *

a = 0.0                               # starting point of domain
b = 1.0                               # ending point of domain
alpha1 = 2.5                          # initial condition
alpha2 = −1.5
alpha3 = −1.0
m = 5                                 # number of steps

def f1(x, y1, y2, y3):
        return y1 + 3 * y2 – 3 * y3 + exp(−x)
```

```

def f2(x, y1, y2, y3):
        return 2 * y2 + y3 – 3 * exp(−x)

def f3(x, y1, y2, y3):
        return y1 + 2 * y2 + exp(−x)

# Define the RK4 solver for the ODE system

def rk4system3(x, w1, w2, w3, h):
        for i in range(1, m + 1):
                k11 = h * f1(x[i−1], w1[i−1], w2[i−1], w3[i−1])
                k21 = h * f2(x[i−1], w1[i−1], w2[i−1], w3[i−1])
                k31 = h * f3(x[i−1], w1[i−1], w2[i−1], w3[i−1])

                k12 = h * f1(x[i−1] + 0.5 * h, w1[i−1] + 0.5 * k11, w2[i−1] + 0.5 * k21,
                w3[i-1] + 0.5 * k31)
                k22 = h * f2(x[i−1] + 0.5 * h, w1[i−1] + 0.5 * k11, w2[i−1] + 0.5 * k21,
                w3[i-1] + 0.5 * k31)
                k32 = h * f3(x[i−1] + 0.5 * h, w1[i−1] + 0.5 * k11, w2[i−1] + 0.5 * k21,
                 w3[i-1] + 0.5 * k31)

                k13 = h * f1(x[i−1] + 0.5 * h, w1[i−1] + 0.5 * k12, w2[i−1] + 0.5 *
                k22, w3[i−1] + 0.5 * k32)
                k23 = h * f2(x[i−1] + 0.5 * h, w1[i−1] + 0.5 * k12, w2[iv1] + 0.5 * k22,
                w3[i−1] + 0.5 * k32)
                k33 = h * f3(x[i−1] + 0.5 * h, w1[i−1] + 0.5 * k12, w2[i−1] + 0.5 *
                k22, w3[i−1] + 0.5 * k32)

                k14 = h * f1(x[i−1] + h, w1[i−1] + k13, w2[i−1] + k23, w3[i−1] + k33)
                k24 = h * f2(x[i−1] + h, w1[i−1] + k13, w2[i−1] + k23, w3[i−1] + k33)
                k34 = h * f3(x[i−1] + h, w1[i−1] + k13, w2[i−1] + k23, w3[i−1] + k33)

                w1[i] = w1[i−1] + (k11 + 2 * k12 + 2 * k13 + k14) / 6.0
                w2[i] = w2[i−1] + (k21 + 2 * k22 + 2 * k23 + k24) / 6.0
                w3[i] = w3[i−1] + (k31 + 2 * k32 + 2 * k33 + k34) / 6.0

h = (b – a) / m
x = linspace(a, b, m+1)
w1 = zeros(m+1)
w2 = zeros(m+1)
w3 = zeros(m+1)
w1[0] = alpha1                    # setting initial condition
w2[0] = alpha2
w3[0] = alpha3

# Call the RK4 solver

rk4system3(x, w1, w2, w3, h)

```

```
# ----------------- Printing Solutions -----------------
print("Node       x[i]        w1[i]         w2[i]        w3[i]")
for i in range(0,m+1):
        print(i,"\t","%.2f" %x[i],"\t","%.7f" %w1[i],"\t","%.7f" % w2[i],"\t","%.8f" % w3[i])
```

Output Console:

```
Node        x[i]         w1[i]           w2[i]            w3[i]
0         0.00     2.5000000         -1.5000000        -1.00000000
1         0.20     2.4526244         -3.1668789        -1.22125420
2         0.40     1.4332465         -5.6843198        -2.39814695
3         0.60     -0.7132915        -9.7983813        -5.20827423
4         0.80     -4.2612434        -16.8302490       -10.77417722
5         1.00     -9.7413880        -29.1043858       -21.00695957
```

Question 30: Write a Python program to find the numerical solution of the ODE, $xy'' - y' + 8x^3y^3 = 0$ with initial condition $y(1) = 0.5$ and $y'(1) = -0.5$ for $y(1.1)$. Consider the step size of $h = 0.1$, thus only step is required. i.e., $m = 1$. Use the exact solution, $y = 1/(1 + x^2)$, to find the error in the numerical solution.

For the solution, consider

$$y' = z$$

Then, the given ODE becomes

$$z' = \frac{(z - 8x^3y^3)}{x}$$

Thus, the second-order IVP is essentially converted to the problem of a first-order system of ODEs of comprising the two equations subject to the initial conditions:

$$w_{10} = y_0 = y(x_0) = y(1) = 0.5$$

$$w_{20} = z_0 = z(x_0) = z(1) = -0.5$$

The problem is to find approximations w_{11} to $y_1 = y(x_1)$ and w_{21} to $z_1 = z(x_1)$.

The Python program for the solution is as follows.

script_9.11: ode_order2.ipynb

```
from numpy import *

a = 1.0                              # starting point of domain
b = 1.1                              # ending point of domain
alpha1 = 0.5                         # initial condition
alpha2 = -0.5
m = 5                                # number of steps

```

```
def f1(x, y1, y2):
        return y2

def f2(x, y1, y2):
        return (y2 - 8 * x * x * x * y1 * y1 * y1) / x

# Define the RK4 solver for the ODE system of two equations

def rk4system2(x, w1, w2, h):
        for i in range(1, m + 1):
                k11 = h * f1(x[i – 1], w1[i – 1], w2[i – 1])
                k21 = h * f2(x[i –1], w1[i – 1], w2[i – 1])

                k12 = h * f1(x[i – 1] + 0.5 * h, w1[i – 1] + 0.5 * k11, w2[i – 1] + 0.5 * k21)

                k22 = h * f2(x[i – 1] + 0.5 * h, w1[i – 1] + 0.5 * k11, w2[i – 1] + 0.5 * k21)

                k13 = h * f1(x[i – 1] + 0.5 * h, w1[i – 1] + 0.5 * k12, w2[i – 1] + 0.5 *
                k22)
                k23 = h * f2(x[i – 1] + 0.5 * h, w1[i – 1] + 0.5 * k12, w2[i – 1] + 0.5 *
                k22)

                k14 = h * f1(x[i – 1] + h, w1[i – 1] + k13, w2[i – 1] + k23)
                k24 = h * f2(x[i – 1] + h, w1[i – 1] + k13, w2[i – 1] + k23)

                w1[i] = w1[i – 1] + (k11 + 2 * k12 + 2 * k13 + k14) / 6.0
                w2[i] = w2[i – 1] + (k21 + 2 * k22 + 2 * k23 + k24) / 6.0

h = (b – a) / m
x = zeros(m+1)
w1 = zeros(m+1)
w2 = zeros(m+1)
w1[0] = alpha1                  # setting initial condition
w2[0] = alpha2

x[0] = a
x[m] = b

for i in range(1, m):
        x[i] = x[i – 1] + h

# Call the RK4 solver

rk4system2(x, w1, w2, h)

# ----------------- Printing Solutions -----------------
print("Node      x[i]       w1[i]        w2[i]")
for i in range(0,m+1):
        print(i,"\t","%.2f" %x[i],"\t","%.7f" %w1[i],"\t","%.7f" % w2[i])
```

Output Console:

```
Node        x[i]         w1[i]          w2[i]
0          1.00     0.5000000        -0.5000000
1          1.02     0.4897001        -0.5299800
2          1.04     0.4788016        -0.5598408
3          1.06     0.4673080        -0.5894651
4          1.08     0.4552253        -0.6187399
5          1.10     0.4425615        -0.6475576
```

■

Question 31: Solve the ODE $y''' = -y'' + 3y' + 3y$ for $y = y(x)$ in $x \in [0,2]$ with the initial conditions: $y(0) = 2.0$, $y'(0) = -1.0$, and $y''(0) = 8.0$. Solve it for 10 steps.

Given the equation,

$$y''' = -y'' + 3y' + 3y \qquad ---(1)$$

For $y = y(x)$ in $x \in [0,2]$ with the initial conditions:

$$y(0) = 2.0$$

$$y'(0) = -1.0$$

$$y''(0) = 8.0$$

consider

$$y' = z_1 \qquad ---(2)$$

$$y'' = z_1' = z_2 \qquad ---(3)$$

Then, the given third-order Eq. (1) becomes

$$z_2' = -z_2 + 3z_1 + 3y \qquad ---(4)$$

Thus, the third-order IVP is essentially converted to the problem of a first-order system of ODEs of comprising the three equations (2) - (4) subject to the initial conditions:

$$y(0) = 2.0$$

$$z_1(0) = -1.0$$

$$z_2(0) = 8.0$$

For 10 steps, the domain is discretized as

$$h = \frac{b-a}{m} = \frac{2.0-0.0}{10} = 0.2$$

$x_0 = 0, x_1 = 0.2, x_2 = 0.4, x_3 = 0.6, x_4 = 0.8, x_5 = 1.0, x_6 = 1.2, x_7 = 1.4, x_8 = 1.6, x_9 = 1.8, x_{10} = 2.0.$

According to the initial conditions:

$$w_{10} = y_0 = y(x_0) = y(0) = 2.0$$

$$w_{20} = z_{10} = z_1(x_0) = z_1(0) = -1.0$$

$$w_{30} = z_{20} = z_2(x_0) = z_2(0) = 8.0$$

The problem is to find approximations w_{1i} to $y_i = y(x_i)$, w_{2i} to $z_{1i} = z_1(x_i)$, and w_{3i} to $z_{2i} = z_2(x_i)$, for $i = 1,2,\cdots,10$.

The Python program for the solution is as follows.

script_9.12: ode_order3.ipynb

```
from numpy import *

a = 0.0                                 # starting point of domain
b = 2.0                                 # ending point of domain
alpha1 = 2.0                            # initial condition
alpha2 = −1.0
alpha3 = 8.0
m = 5                                   # number of steps

def f1(x, y, z1, z2):
        return z1

def f2(x, y, z1, z2):
        return z2

def f3(x, y, z1, z2):
        return −z2 + 3 * z1 + 3 * y

# Define the RK4 solver for the ODE system

def rk4system3(x, w1, w2, w3, h):
        for i in range(1, m + 1):
                k11 = h * f1(x[i−1], w1[i−1], w2[i−1], w3[i−1])
                k21 = h * f2(x[i−1], w1[i−1], w2[i−1], w3[i−1])
                k31 = h * f3(x[i−1], w1[i−1], w2[i−1], w3[i−1])

                k12 = h * f1(x[i−1] + 0.5 * h, w1[i−1] + 0.5 * k11, w2[i−1] + 0.5 * k21,
                w3[i-1] + 0.5 * k31)
                k22 = h * f2(x[i−1] + 0.5 * h, w1[i−1] + 0.5 * k11, w2[i−1] + 0.5 * k21,
                w3[i-1] + 0.5 * k31)
                k32 = h * f3(x[i−1] + 0.5 * h, w1[i−1] + 0.5 * k11, w2[i−1] + 0.5 * k21,
                 w3[i−1] + 0.5 * k31)

                k13 = h * f1(x[i−1] + 0.5 * h, w1[i−1] + 0.5 * k12, w2[i−1] + 0.5 *
                k22, w3[i−1] + 0.5 * k32)
                k23 = h * f2(x[i−1] + 0.5 * h, w1[i−1] + 0.5 * k12, w2[i−1] + 0.5 *
                k22, w3[i−1] + 0.5 * k32)
```

```
            k33 = h * f3(x[i−1] + 0.5 * h, w1[i−1] + 0.5 * k12, w2[i−1] + 0.5 *
            k22, w3[i−1] + 0.5 * k32)

            k14 = h * f1(x[i−1] + h, w1[i−1] + k13, w2[i−1] + k23, w3[i−1] + k33)
            k24 = h * f2(x[i−1] + h, w1[i−1] + k13, w2[i−1] + k23, w3[i−1] + k33)
            k34 = h * f3(x[i−1] + h, w1[i−1] + k13, w2[i−1] + k23, w3[i−1] + k33)

            w1[i] = w1[i−1] + (k11 + 2 * k12 + 2 * k13 + k14) / 6.0
            w2[i] = w2[i−1] + (k21 + 2 * k22 + 2 * k23 + k24) / 6.0
            w3[i] = w3[i−1] + (k31 + 2 * k32 + 2 * k33 + k34) / 6.0

h = (b − a) / m
x = linspace(a, b, m+1)
w1 = zeros(m+1)
w2 = zeros(m+1)
w3 = zeros(m+1)
w1[0] = alpha1                    # setting initial condition
w2[0] = alpha2
w3[0] = alpha3

# Call the RK4 solver

rk4system3(x, w1, w2, w3, h)

# ----------------- Printing Solutions -----------------
print("Node     x[i]     w1[i]      w2[i]      w3[i]")
for i in range(0,m+1):
      print(i,"\t","%.2f" %x[i],"\t","%.7f" %w1[i],"\t","%.7f" % w2[i],"\t","%.8f" % w3[i])
```

Output Console:

```
Node      x[i]        w1[i]           w2[i]            w3[i]
0         0.00     2.0000000       -1.0000000       8.00000000
1         0.40     2.2144000       2.0592000        7.98400000
2         0.80     3.7553715       5.9235814        12.16498688
3         1.20     7.3178564       12.5912247       22.55617331
4         1.60     14.6281541      25.4339096       44.28844810
5         2.00     29.2965237      50.8850897       88.16040310
```

■

Question 32: Write down an algorithm (pseudo code) to solve a second-order linear ODE (BVP) with Dirichlet boundary condition using the finite difference method of second-order accuracy. The algorithm should follow the Gauss-Seidel approach to solve the linear system resulted after discretization of the model equation.

Algorithm: To solve $y'' = f(x, y, y') = p(x)y' + q(x)y + r(x)$, for $a \le x \le b$ subject to the Dirichlet boundary conditions: $y(a) = \alpha$ and $y(b) = \beta$ by approximating $y = y(x)$ at $(m + 2)$ equispaced nodes

$x_0, x_1, x_2, \cdots, x_m, x_{m+1}$, such that $a = x_0 < x_1 < x_2 < \cdots < x_m < x_{m+1} = b$, $h = (b-a)/m$ and $y(x_i) = y_i$ using the finite difference method based on the central difference of second-order accuracy.

INPUTS: $\begin{cases} a, b\text{: real values as the endpoints of the interval: } x \in [a,b] \\ m\text{: an integer as the number of interior nodes in the } x \text{ direction} \\ alpha\text{: a real value as the boundary condition } y(a) \\ beeta\text{: a real value as the boundary condition } y(b) \\ N\text{: an integer as the maximum number of iterations} \\ TOL\text{: a real value as the error tolerance} \\ \text{Definitions of the functions } p(x), q(x), \text{and } r(x) \text{ in an appropriate way} \end{cases}$

OUTPUT: $\begin{cases} Z = [z_0, z_1, \cdots, z_m, z_{m+1}]^T\text{: a real valued vector as the approximate values of } y(x_i) \\ \text{at the nodes } x_0, x_1, x_2, \cdots, x_m, x_{m+1} \end{cases}$

Auxiliary Variables: $\begin{cases} h\text{: a real value as the step length in } x \text{ direction: } h = (b-a)/(m+1) \\ X = (x_i) \text{ for } i = 0, 1, \cdots, m, m+1\text{: a real valued vector to represent } x_i s \\ ZP = [zp_0, zp_1, \cdots, zp_m, zp_{m+1}]^T\text{: a real valued vector to keep a copy of } Z \\ err\text{: a real number to hold the value of error norm in each iteration} \\ B = [b_0, b_1, \cdots, b_m]^T\text{: a real valued vector to hold right hand side constants} \\ D = [d_0, d_1, \cdots, d_m]^T\text{: a real valued vector to hold diagonal entries} \\ U = [u_0, u_1, \cdots, u_m]^T\text{: a real valued vector to hold upper diagonal entries} \\ L = [l_0, l_1, \cdots, l_m]^T\text{: a real valued vector to hold lower diagonal entries} \end{cases}$

```
Step 1    Receive the inputs as stated above

Step 2    Set $h = (b-a)/(m+1)$
          Set $x(0) = a$
          Set $x(m+1) = b$

Step 3    for $i = 1, 2, \cdots, m$
              Set $x(i) = x(0) + i \times h$        (Constructing interior mesh points, $x_i$ )
          end for

Step 4    (Applying the boundary conditions)
          Set $w(0) = alpha$
          Set $w(m+1) = beeta$

Step 5    (Setting the initial conditions on interior nodes)
          for $i = 1, 2, \cdots, m$
                  Set $w(i) = 0$        (Constructing interior mesh points, $x_i$ )
          end for

Step 6    for $i = 1, 2, \cdots, m$
             Set $B(i) = -h \times h \times r(x(i))$;  end for

          for $i = 1, 2, \cdots, m$
             Set $D(i) = 2 + h \times h \times q(x(i))$ ;  end for
```

```
for $i = 1, 2, \cdots, m$
    Set $U(i) = -1 + h \times 0.5 \times p\big(x(i)\big)$ ;  end for

for $i = 1, 2, \cdots, m$
    Set $L(i) = -1 - h \times 0.5 \times p\big(x(i)\big)$ ;  end for
```

```
**Step 7**    for $k = 1, 2, 3, \cdots, N$ perform steps 8-11

    Step 8
    for $i = 1, 2, \cdots, m$   Set $ZP(i) = W$   (keeping a copy of $Z$ in $ZP$ for taking the norm)

    Step 9
        for $i = 1, 2, \cdots, m$        (compute the components of solution vector $Z$)
            $w(i) = \frac{B(i) - L(i) \times w(i-1) - U(i) \times w(i+1)}{D(i)}$
        end for

    Step 10  Compute  $err = \|W - ZP\|$
                    (or $err = \|X - XP\|/\|X\|$)  Here $\|\cdot\|$ is any suitable norm.

    Step 11
        if ($err < TOL$ )then                  } This means that the consecutive
            Exit/Break the loop                } approximations are nearly the same.
                                                 Therefore, stop iterations.

    end for loop of Step 7  (Go to Step 8)

**Step 12**    Print the output: $W = [w_0, w_1, \cdots, w_m]^T$ ;  **STOP**.
```

Question 33: Write a Python program that uses a second-order accurate Finite Difference method to solve the following boundary value problem:

$$y'' = y' + 2y + \cos(x), \quad \text{for } y = y(x), \quad \text{where} \quad 0 \le x \le \frac{\pi}{2}$$

subject to the following Dirichlet boundary conditions: $y(0) = -0.3$ and $y\left(\frac{\pi}{2}\right) = -0.1$.

For domain discretization, take step sizes as $h = \Delta x = \frac{\pi}{8}$

To form the computational domain, the physical domain $\left[0, \frac{\pi}{2}\right]$ is discretized by considering that it consists of a number of equispaced discrete points or nodes, x_i, for $i = 0, 1, 2, \cdots, m+1$. For the given problem,

$$\text{Number of interior nodes} \;=\; m \;=\; 3$$

$p(x) = 1$

$q(x) = 2$

$r(x) = \cos(x)$

The target is to obtain the approximations w_i to the function values $y_i = y(x_i)$ at the interior nodes x_i, for $i = 1, 2, 3$. The values of the solution function are known at x_0 and x_4 due to Dirichlet boundary conditions:

$$w_0 = y(x_0) = -0.3$$
$$w_4 = y(x_4) = -0.1$$

A Python program that uses the Gauss-Seidel approach for the stated solution is as follows.

script_9.13:finite_difference_2nd.ipynb

```
from numpy import *

a = 0.0                                 # starting point of domain
b = pi / 2                              # ending point of domain
alpha = −0.3
beta = −0.1
N = 200
m = 3                                   # number of steps
TOL = 1e-7

def p(x):
        return 1.0

def q(x):
        return 2.0

def r(x):
        return cos(x)

# Define the efficient Gauss-Seidel method

def egs(z, B, D, U, L, h):
        for k in range(1, N + 1):
                zp = copy(z)

                for i in range(1, m + 1):
                        z[i] = (B[i] − L[i] * z[i − 1] − U[i] * z[i + 1]) / D[i]
                print(f"{k:4}: z= {z[0]:.2f} ", end=" ")

                for i in range(1, m + 1):
                        print(f"{z[i]:.8f} ", end=" ")
                print(f"{z[m+1]:.2f} ")

                err = sqrt(sum((z[1:m+1] − zp[1:m+1])**2 / z[1:m+1]**2))

                if err < TOL:
                        break

h = (b − a) / (m+1)
```

```
x = linspace(a, b, m+2)
z = zeros(m + 2)
zp = zeros(m + 2)
B = zeros(m + 1)
D = zeros(m + 1)
U = zeros(m + 1)
L = zeros(m + 1)
x[0] = a
x[m+1] = b

for i in range(1, m + 1):
        x[i] = x[i – 1] + h
        print(f"\tnodes {x[i]:.8f}")

z[0] = alpha
z[m+1] = beta

for i in range(1, m + 1):
        B[i] = –h**2 * r(x[i])

for i in range(1, m + 1):
        D[i] = 2 + h**2 * q(x[i])

for i in range(1, m + 1):
        U[i] = –1.0 + 0.5 * h * p(x[i])

for i in range(1, m + 1):
        L[i] = –1.0 - 0.5 * h * p(x[i])

# ----------------- Printing Solutions -----------------
print(f"{0:4}: z= {z[0]:.2f} ", end=" ")
for i in range(1, m + 1):
        print(f"{z[i]:.2f} ", end=" ")
print(f"{z[m+1]:.2f} ")

# Call theGauss seidel function

egs(z, B, D, U, L, h)
```

Output Console:

```
nodes 0.39269908
nodes 0.78539816
nodes 1.17809725
   0: z= -0.30   0.00              0.00      0.00        -0.10
   1: z= -0.30  -0.21719513  -0.15979987  -0.14319552  -0.10
   2: z= -0.30  -0.27282754  -0.23848337  -0.18397352  -0.10
   3: z= -0.30  -0.30022025  -0.26687611  -0.19868816  -0.10
   4: z= -0.30  -0.31010484  -0.27712156  -0.20399790  -0.10
```

```
  5: z= -0.30  -0.31367167  -0.28081860  -0.20591391  -0.10
  6: z= -0.30  -0.31495875  -0.28215267  -0.20660530  -0.10
  7: z= -0.30  -0.31542319  -0.28263407  -0.20685478  -0.10
  8: z= -0.30  -0.31559078  -0.28280778  -0.20694481  -0.10
  9: z= -0.30  -0.31565126  -0.28287046  -0.20697729  -0.10
 10: z= -0.30  -0.31567308  -0.28289308  -0.20698902  -0.10
 11: z= -0.30  -0.31568095  -0.28290124  -0.20699325  -0.10
 12: z= -0.30  -0.31568380  -0.28290419  -0.20699477  -0.10
 13: z= -0.30  -0.31568482  -0.28290525  -0.20699532  -0.10
 14: z= -0.30  -0.31568519  -0.28290563  -0.20699552  -0.10
 15: z= -0.30  -0.31568532  -0.28290577  -0.20699559  -0.10
 16: z= -0.30  -0.31568537  -0.28290582  -0.20699562  -0.10
 17: z= -0.30  -0.31568539  -0.28290584  -0.20699563  -0.10
```

Chapter Summary

- The numerical solution of an ODE is not a definition of $y = y(x)$. The numerical solution of the ODE is a set of numbers w_i that are approximations to the function values $y(x_i)$ at some pre-specified discrete values $x_i \in [a, b]$. That is, $w_i \cong y_i = y(x_i)$.

- To solve an initial-value problem consisting of a single first-order ODE in $y = y(x)$ for $a \leq x \leq b$ and an initial-value $y(a) = \alpha$, first the domain $[a, b]$ is discretized by selecting $(m + 1)$ equispaced nodes $x_0, x_1, x_2, \cdots, x_m$ in $[a, b]$ such that $a = x_0 < x_1 < x_2 < \cdots < x_m = b$, and $h = (b - a)/m$. Then, approximations w_i to the values $y_i = y(x_i)$ for $i = 1,2,\cdots, m$ are obtained with w_0=$y(a)$. For simplicity, $y(x_i)$ is denoted by y_i.

- There is a wide variety of methods for finding numerical solutions of the ODEs involved in initial value problems (IVPs) and boundary value problems (BVPs).

- Methods for IVPs include single step methods and multi-step methods, each category having explicit and implicit methods. A hybrid method, i.e., predictor-corrector method, involves a combination of explicit and implicit formulas.

- Methods for BVPs are so versatile and involve much richer mathematical constructs.

- The accuracy of the approximate solution can be improved either by using a larger number of steps (a smaller step size), or by using a better numerical method.

- The prime characteristics (or considerations) associated with a finite difference scheme to determine its quality include

- Stability
- Local Truncation Error
- Consistency (Compatibility)
- Discretization Error
- Convergence

■■■

Chapter Exercises

Exercise 01: Find the numerical solution of the ODE, $y' = 3 - 3y - e^{-6x}$, for $0 \leq x \leq 2$, with initial condition $y(0) = 1.0$. Consider the step size of 0.5. Use the exact solution, $y(x) = \frac{1}{3}(e^{-6x} - e^{-3x} + 3)$, to find the error in the numerical solution.

Exercise 02: Find the numerical solution of the ODE, $y' = 1 + (x - y)^2$, for $2 \leq x \leq 3$, with initial condition $y(2) = 1.0$. Consider the step size of 0.5. Use the exact solution, $y(x) = x + 1/(1 - x)$, to find the error in the numerical solution.

Exercise 03: Find the numerical solution of the ODE, $y' = 2 + (x - y)^2$, for $2 \leq x \leq 3$, with initial condition $y(2) = 1.5$. Consider the step size of 0.5. Use the exact solution, $y(x) = x - \tan(-x + 2.463)$, to find the

Exercise 04: Find the numerical solution of the ODE, $y' = (1 + x)/(1 + y)$, for $0 \leq x \leq 1$, with initial condition $y(0) = 2.0$. Consider the step size of 0.5. Use the exact solution, $y(x) = \sqrt{x^2 + 2x + 9} - 1$, to find the error in the numerical solution.

Exercise 05: For the functions $y_1 = y_1(x)$ and $y_2 = y_2(x)$, where $x \in [0,1]$, solve the following system of two ODEs:

$$\begin{aligned} y_1' &= y_1 y_2 - 2 \\ y_2' &= 2y_1 - y_2^3 \end{aligned}$$

With initial conditions:

$$\begin{aligned} y_1(0) &= 2.0 \\ y_2(0) &= 0.3 \end{aligned}$$

Use the RK4 method of order 4 for 5 steps.

HINT: For 5 steps the domain is discretized as

$$h = \frac{b-a}{m} = \frac{1.0-0.0}{5} = 0.2$$

$x_0 = 0, x_1 = 0.2, x_2 = 0.4, x_3 = 0.6, x_4 = 0.8, x_5 = 1.0.$

According to the initial conditions:

$$w_{10} = y_{10} = y_1(x_0) = y_1(0) = 2.0$$

$$w_{20} = y_{20} = y_2(x_0) = y_2(0) = 0.3$$

The problem is to find approximations w_{1i} to $y_{1i} = y_1(x_i)$ and w_{2i} to $y_{2i} = y_2(x_i)$, for $i = 1,2,3,4,5$.

Exercise 06: For the functions $y_1 = y_1(x), y_2 = y_2(x)$, and $y_3 = y_3(x)$, where $x \in [0,1]$, solve the following system of three ODEs:

$$y_1' = y_1 + 3y_2 - 3y_3 + e^{-x}$$

$$y_2' = 2y_2 + y_3 - 3e^{-x}$$

$$y_3' = y_1 + 2y_2 + e^{-x}$$

with initial conditions:

$$y_1(0) = 2.5$$

$$y_2(0) = -1.5$$

$$y_3(0) = -1.0$$

Use the RK4 method of order 4 for 10 steps.

HINT: For 10 steps, the domain is discretized as

$$h = \frac{b-a}{m} = \frac{1.0-0.0}{10} = 0.1$$

$x_0 = 0, x_1 = 0.1, x_2 = 0.2, x_3 = 0.3, x_4 = 0.4, x_5 = 0.5, x_6 = 0.6, x_7 = 0.7, x_8 = 0.8, x_9 = 0.9, x_{10} = 1.0.$

According to the initial conditions:

$$w_{10} = y_{10} = y_1(x_0) = y_1(0) = 2.5$$

$$w_{20} = y_{20} = y_2(x_0) = y_2(0) = -1.5$$

$$w_{30} = y_{30} = y_3(x_0) = y_3(0) = -1.0$$

The problem is to find approximations w_{1i} to $y_{1i} = y_1(x_i)$, w_{2i} to $y_{2i} = y_2(x_i)$, and w_{3i} to $y_{3i} = y_3(x_i)$, for $i = 1,2,\cdots,10$.

Exercise 07: Find the numerical solution of the IVP, $y'' - 8y' + 7y = 16e^{-x}$ for $0 \le x \le 1$, with initial condition $y(0) = 4.0$ and $y'(0) = 4.0$. Also find $y(1.1)$. Consider the step size of 0.1. Use the exact solution, $y = (1/3)(e^{7x} + 8e^{x} + 3e^{-x})$, to find the error in the numerical solution.

HINT: Given the equation,

$$y'' - 8y' + 7y = 16e^{-x} \qquad ---(1)$$

For the solution, consider

$$y' = z \qquad ---(2)$$

Then, the given second-order Eq. (1) becomes

$$z' = 4z - 3y + 7e^{-x} \qquad ---(3)$$

Thus, the second-order IVP is essentially converted to the problem of a first-order system of ODEs of comprising the two equations (2) and (3) subject to the initial conditions:

$$y(0) = 3.0$$
$$z(0) = 3.0$$

For 10 steps, the domain is discretized as

$$h = \frac{b-a}{m} = \frac{1.0-0.0}{10} = 0.1$$

$x_0 = 0, x_1 = 0.1, x_2 = 0.2, x_3 = 0.3, x_4 = 0.4, x_5 = 0.5, x_6 = 0.6, x_7 = 0.7, x_8 = 0.8, x_9 = 0.9, x_{10} = 1.0.$

According to the initial conditions:

$$w_{10} = y_0 = y(x_0) = y(0) = 3.0$$
$$w_{20} = z_0 = z(x_0) = z(0) = 3.0$$

The problem is to find approximations w_{1i} to $y_i = y_i(x_i)$ and w_{2i} to $z_i = z(x_i)$, for $i = 1,2,\cdots,10$.

Exercise 08: Solve the ODE $y'' = 4y' - 3y + 7e^{-x}$ for $y = y(x)$ in $x \in [0,1]$ with the initial conditions: $y(0) = 3.0$ and $y'(0) = 3.0$. Solve it for 10 steps.

Exercise 09: Find the numerical solution of the BVP, $y'' - 9y' + y = x$ for $0 \le x \le 1$, with initial condition $y(0) = 0.0$ and $y'(1) = 6.0$. Consider the step size of 0.1.

Exercise 10: Find the numerical solution of the ODE, $x^2y'' + 3xy' + 3y = 0$, with initial condition $y(1) = 1$ and $y'^{(1)} = -5$ for $y(1.1)$. The exact solution is, $y = \frac{1}{x}\left(\cos(\sqrt{2}\ln x) + \left(\frac{1}{x^2} - 5\right)\sin(\sqrt{2}\ln x)\right)$.

Exercise 11: Find the numerical solution of the ODE, $y'' - 6y' + 9y = x^2e^{3x}$, with initial condition $y(0) = 2$ and $y'(0) = 6$ for $y(1.1)$. The exact solution is, $y = 2e^{3x} + \frac{1}{12}x^4e^{3x}$.

Exercise 12: Solve the ODE $y''' = -y'' + 3y' + 3y$ for $y = y(x)$ in $x \in [0,2]$ with the initial conditions: $y(0) = 2.0$, $y'(0) = -1.0$, and $y''(0) = 8.0$. Solve it for 10 steps.

Exercise 13: Using a second-order accurate Finite Difference method, solve the following BVP:

$$y'' = 9y' - y + x, \quad \text{for } y = y(x), \quad \text{where} \quad 0 \le x \le 1$$

subject to the following Dirichlet boundary conditions: $y(0) = 0$ and $y(1) = 6$.
For domain discretization, take step sizes as $h = \Delta x = 0.25$.

Exercise 14: Using a second-order accurate Finite Difference method, solve the following BVP:

$$y'' = -5y' - 8y + x^2, \quad \text{for } y = y(x), \quad \text{where} \quad 1 \le x \le 2$$

subject to the following Dirichlet boundary conditions: $y(1) = 0$ and $y(2) = 24$. For domain discretization, take step sizes as $h = \Delta x = 0.25$.

■■■

Chapter **10**

Introduction to SciPy

SciPy (Scientific Python) is an open-source library in Python that is used for solving mathematical, scientific, engineering, and technical problems. It allows users to manipulate the data and visualize the data using a wide range of high-level Python commands. SciPy stands for scientific Python and it is built on the Python NumPy extension. It contains varieties of sub-packages that help to solve the most common issue related to scientific computing. Though NumPy provides a number of functions that can help to resolve linear algebra, Fourier transforms, integration, etc., the SciPy module in Python is a fully-featured version of these functions and many more. Most data science features are available in SciPy rather than NumPy. The SciPy library supports integration, gradient optimization, special functions, ordinary differential equation solvers, parallel programming tools, and many more. We can say that SciPy implementation exists in every complex numerical computation.

Following are some useful sub-packages of SciPy.

scipy.io for File I/O

This SciPy sub-package contains modules, classes and functions to read data from and write data to various file formats sch as MATLAB files, unformatted Fortran files wave sound files, etc.

```
import numpy as np
from scipy import io as sio
array = np.ones((2,2))
#store data in example.mat file
sio.savemat("example.mat", { "ar" : array})
#get data from example.mat file
data = sio.loadmat("example.mat")
```

```
data[ "ar"]
```

Output:

```
array([1., 1.] , [1. , 1.]])
```

scipy.special for Special Functions

SciPy special functions include Cubic Root, Exponential, Log sum Exponential, Permutation and Combination, Lambert, Bessel, Hypergeometric functions, etc.

Cube Root Function

```
from scipy.special import cbrt
cb = cbrt(27)
print(cb)
```

Output:

```
3.0
```

Exponential Function

```
from scipy.special import exp10
exp = exp10([1,10])
print(exp)
```

Output:

```
[1.e+01 1.e+10]
```

Permutations and Combinations

```
from scipy.special import comb
from scipy.special import perm
com = comb(5,3)
```

```
per = perm(5,3)
print("Combination = " , com)
print("permutation =" , per)
```

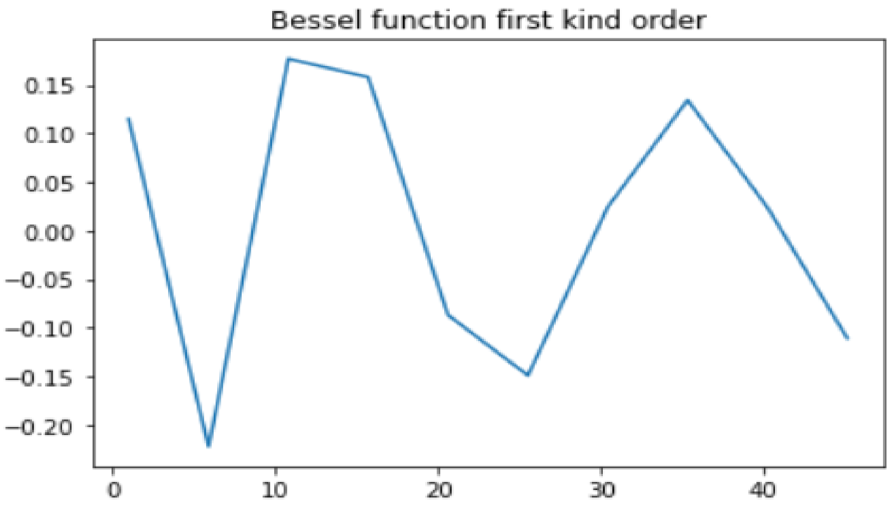
Output:

```
Combination = 10.0
Permutation = 60.0
```

Bessel Function

```
import scipy.special as special
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(1,50,10,0)
jn1 = special.jn(2,x)
plt.title("Bessel function first kind order")
plt.plot(x,jn1)
```

Output:



scipy.linalg for Linear Algebra

It includes the basic functions to find the inverse, determinant of a matrix and to solve Eigenvalue problems, Decompositions, Matrix functions, etc.

Finding the inverse and determinant

```
import numpy as np
from scipy import linalg
A = np.array([[5,2] , [3,6]])
B = linalg.inv(A)
C = linalg.det("determinant = " , A)
Print(B)
Print(C)
```

Output:

```
[[ 0.25    -0.08333333]
 [-0.125    0.20833333]]
Determinant = 24.0
```

Eigenvalues and Eigenvector

```
import numpy as np
from scipy import linalg
A = np.array([[5,2] , [3,6]])
Eg_val , Eg_vect = linalg.eig(A)
print("Eigen value = " , Eg_val)
print("Eigen vector = " , Eg_vect)
```

Output:

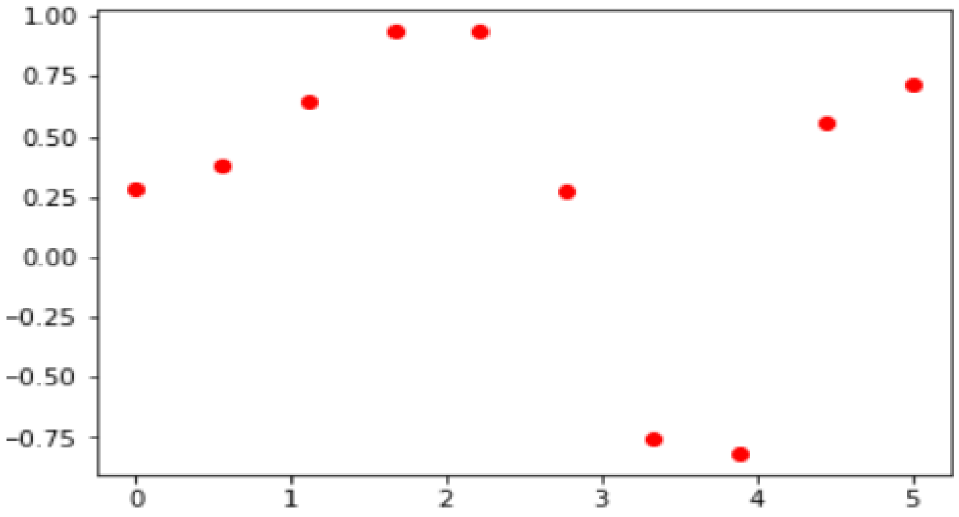
```
Eigen value = [3.+0.j 8.+0.j]
Eigen vector = [[-0.7071067 -0.554700]
 [ 0.70710678 -0.83205029]]
```

scipy.interpolate for Interpolation

It includes spline functions and classes 1-D and multidimensional interpolation classes, Lagrange and Taylor polynomial interpolators.

```
import numpy as np
from scipy import interpolate
import matplotlib.pyplot as plt
x = np.linspace(0,5,10)
y = np.cos(x**2/3+5)
plt.scatter(x,y, c = 'r')
plt .show()
```

Output:



scipy.integrate for Numerical Integration

It includes functions to solve single integration, double, triple, multiple Gaussian quadrate, Trapezoidal, and Simpson's rules.

Single Integration

```
from scipy import integrate
```

```
f = lambda x: x**3
a = 0
b = 1
integration = integrate.quad(f,0,1)
print(integration)
```

Output:

```
(0.25, 2.7755575615628914e-15)
```

Double Integration

```
from scipy import integrate
from math import sqrt
f = lambda x , y : 64*x*y
#lower limit of second integral
p = lambda x: 0
#upper limit of first integral
q = lambda y : sqrt(1 – 2*y**2)
#double integration
integration = integrate.dblquad(f,0,2/4,p,q)
print(integration)
```

Output:

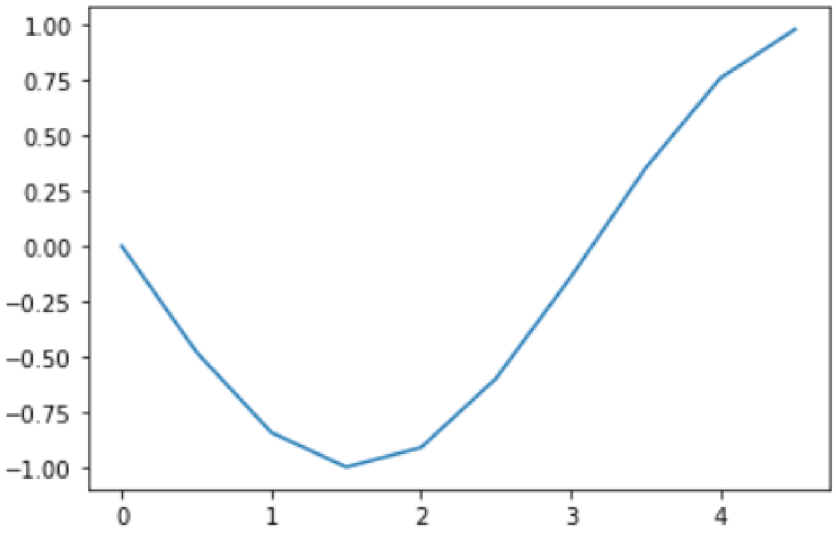
```
(3.0, 9.657432734515774e-14)
```

scipy.optimize for Optimization

It provides a function for minimizing and maximizing objective functions. It also solvers for non-linear problems, linear programming, root findings, and curve fitting.

```
from scipy import optimize
import matplotlib.pyplot as plt
import numpy as np
def f(x):
    return -np.sin(x)
x = np.linspace(0,5,10,0)
start = 3
optimize.fmin(f,start)
plt.plot(x , f(x))
plt.scatter(optimized, f(optimized))
plt.legend(['Function -sin(x) ' , 'Starting point' , 'Optimized
minimum'])
```

Output:



scipy.stats for Statistical functions

This sub-module of SciPy is having a large number of probability distributions and a growing library of statistical functions.

Uniform Distribution

```
from scipy.stats import uniform
```

```
a = np.array([8,7,5,3,2])

print(uniform.cdf(a, loc = 5 , scale = 3))
```

Output:

```
[1.         0.66666667 0.         0.         0.        ]
```

■■■

Bibliography

1. Richard L. Burden & J. Douglas Faires, (2011), Numerical Analysis, (9th Edition), USA, Brooks/Cole Pub. Co.

2. Steven C. Chapra & Raymond P. Canale, (2006), Numerical Methods for Engineers, (5th Edition), NY, USA, McGraw-Hill Co.

3. David R. Kincaid & E. Ward Cheney, (2002), Numerical Analysis: Mathematics of Scientific Computing, (3rd Edition), USA, Brooks/Cole Pub. Co.

4. E. Ward Cheney & David R. Kincaid, (2013), Numerical Mathematics and Computing, (7th Edition), New-Delhi India, Cengage Learning India Pvt. Ltd.

5. Brian Bradie, (2005), A Friendly Introduction to Numerical Analysis, Pearson.

6. John H. Mathews & Kurtis D. Fink, (2015), Numerical Methods using MATLAB, (4th Edition), India, Pearson India Education Services Pvt. Ltd.

7. M. K. Jain, S. R. K. Iyengar & R. K. Jain, (2012), Numerical Methods for Scientific and Engineering Computation, (6th Edition), New-Delhi India, New Age International Pvt. Ltd.

8. George R. Lindfield & John E. T. Penny, (2013), Numerical Methods using MATLAB, (3rd Edition), USA, Academic Press, An imprint of Elsevier.

9. Amos Gillat, (2011), MATLAB: An Introduction with Applications, (4th Edition), USA, John Wiley & Sons, Inc.

10. Laurene V. Fausett, (2009), Applied Numerical Analysis using MATLAB, (2nd Edition), India, PEARSON Education Inc.

11. Babu ram, (2010), Numerical Methods, India, PEARSON Education Inc.

12. Francis Schied, (1990), 2000 Solved Problems in Numerical Analysis, (International Edition), NY, USA, McGraw-Hill Co.

13. P. Siva Ramakrishna Das & C. Vijayakumari, (2004), Numerical Analysis, (1st Edition), India, Dorling Kindersley Pvt. Ltd.

14. Saeed Akhtar Bhatti & Naveed Akhtar Bhatti, (2008), A First Course in Numerical Analysis with C++, (5th Edition), Lahore, Pakistan, A-ONE Publishers.

15. Mohammad Iqbal, (1990), An Introduction to Numerical Analysis, Urdu Bazar Lahore, Pakistan, Ilmi Kitab Khana.

16. Fiaz Ahmad & Muhammad Afzal Rana, (1995), Elements of Numerical Analysis, Islamabad, Pakistan, National Book Foundation

17. Amjad Pervez, (1996), An Introduction to Numerical Analysis, Urdu Bazar Lahore, Pakistan, A.H. Publishers.

18. Germund Dahlquist & Ake Björck, (2003), Numerical Methods, New Jersey, USA, Prentice-Hall Inc.

19. Erwin Kreyszig, (2011), Advanced Engineering Mathematics, (10th Edition), USA, John Wiley & Sons, Inc.

20. S. S. Sastry, (2019), Introductory Methods of Numerical Analysis, (Fifth Edition), PHI Learning Private Limited.

21. Curtis F. Gerald & Patrick O. Wheatley, (2003), Applied Numerical Analysis, (7th Edition), India, PEARSON Education Inc.

22. K. Sankara Rao, (2009), Numerical Methods for Scientists and Engineers, (Third Edition), PHI Learning Private Limited.

23. Lal Din Baig, (2014), Numerical Analysis, Ilmi Kitab Khana, Lahore.