Introduction to Computational Physics

UPES Dehradun

Introduction to Computational Physics - 2025

Dr. Nitesh Kumar

May 23, 2025

Contents

Chapter 1

Introduction to FORTRAN 90 on Linux

1.1 Getting Started with Linux

Before diving into FORTRAN 90, it's essential to understand some basic Linux commands and environment setup to efficiently work with programming on Linux. Linux is a powerful and flexible operating system that is widely used for programming and scientific computing.

1.1.1 Basic Linux Commands

Here are some basic Linux commands you'll use frequently while working with FORTRAN and other programming languages:

- `pwd`: Print the current working directory.

```
$ pwd
/home/user

```

- `ls`: List files and directories.

```
$ ls
Documents  Downloads  hello.f90  Pictures

```

- `cd`: Change directory.

```
$ cd Documents

```

- `mkdir`: Create a new directory.

```
$ mkdir fortran_projects

```

- `rm`: Remove files or directories.

```
$ rm hello.f90

```

- `nano` or `vim`: Command-line text editors. We'll use `nano` for simplicity.

```
$ vim hello.f90

```

- `gfortran`: The GNU Fortran compiler, used for compiling FORTRAN code.

1.1.2 File System Hierarchy

Linux organizes files and directories into a hierarchical structure, starting with the root directory (`/`). Some common directories you'll work with include:

- `/home`: Contains user home directories.
- `/usr`: Contains installed software and libraries.
- `/etc`: Configuration files.

Understanding this structure will help you navigate and manage files while working on your projects.

1.2 Text Processing with `grep`, `sed`, and `awk`

In Unix-like operating systems, efficient text processing is achieved through powerful command-line utilities such as `grep`, `sed`, and `awk`. These tools allow users to search, filter, and manipulate text data directly from the terminal. This section provides an overview of each utility along with detailed examples.

1.2.1 `grep`: Global Regular Expression Print

The `grep` command searches through files for lines that match a specified pattern. It's commonly used for filtering text data based on regular expressions.

Basic Usage

To search for a specific pattern in a file:

```
grep "pattern" filename
```

Example: To find lines containing the word "error" in `system.log`:

```
grep "error" system.log
```


Common Options

- `-i`: Perform case-insensitive matching.
- `-r`: Recursively search directories.
- `-v`: Invert match to select non-matching lines.
- `-n`: Prefix each line with its line number.

Example: Case-insensitive search for "warning" in all `.txt` files:

```
grep -i "warning" *.txt
```

1.2.2 `sed`: Stream Editor

`sed` is a stream editor used for parsing and transforming text. It reads input line by line, applies specified operations, and outputs the modified text.

Basic Substitution

To replace occurrences of a pattern:

```
sed 's/old/new/' filename
```

Example: Replace the first occurrence of "foo" with "bar" in each line of `file.txt`:

```
sed 's/foo/bar/' file.txt
```

Global Substitution

To replace all occurrences in each line:

```
sed 's/old/new/g' filename
```

Example: Replace all occurrences of "foo" with "bar":

```
sed 's/foo/bar/g' file.txt
```

In-Place Editing

To edit files in place:

```
sed -i 's/old/new/g' filename
```

Example: Replace "foo" with "bar" directly in `file.txt`:

```
sed -i 's/foo/bar/g' file.txt
```

1.2.3 `awk`: Pattern Scanning and Processing Language

`awk` is a versatile programming language designed for text processing and data extraction. It operates on files line by line and splits each line into fields.

Printing Specific Fields

Example: Given a file `data.txt`:

```
Name Age Country
Alice 30 USA
Bob 25 UK
Charlie 35 Canada
```

To print the names and ages:

```
awk '{print $1, $2}' data.txt
```

Conditional Processing

Example: Print names of people older than 30:

```
awk '$2 > 30 {print $1}' data.txt
```

Calculations

Example: Calculate the average age:

```
awk '{sum += $2; count++} END {print sum/count}' data.txt
```

Field Separators

`awk` uses whitespace as the default field separator. To specify a different delimiter:

```
awk -F, '{print $1, $2}' data.csv
```

Example: For a comma-separated file `data.csv`:

```
Name,Age,Country
Alice,30,USA
Bob,25,UK
Charlie,35,Canada
```

To print names and countries:

```
awk -F, '{print $1, $3}' data.csv
```

1.2.4 Combining `grep`, `sed`, and `awk`

These tools can be combined to perform complex text processing tasks.

Example: Extract lines containing "error" from `system.log`, replace "error" with "warning", and print the first field:

```
grep "error" system.log | sed 's/error/warning/' | awk '{print $1
    }'
```

This command sequence filters lines with "error", substitutes "error" with "warning", and prints the first field of each resulting line.

Mastering `grep`, `sed`, and `awk` enhances one's ability to efficiently process and analyze text data in Unix-like systems. These tools offer robust functionalities that, when combined, provide powerful solutions for a wide range of text manipulation tasks.

1.3 Historical Development of FORTRAN

FORTRAN (FORmula TRANslation) is one of the oldest high-level programming languages. Originally developed in the 1950s by IBM, it has evolved significantly over the decades, with FORTRAN 90 being a major revision.

1.3.1 Evolution of FORTRAN

- **FORTRAN I (1957)**: The first compiled high-level language, primarily designed for scientific and engineering computations.
- **FORTRAN IV and 66 (1960s)**: Introduced subroutines, functions, and better control structures.
- **FORTRAN 77**: Improved string handling and more complex control structures.
- **FORTRAN 90 (1991)**: Introduced modern programming concepts like recursion, modules, dynamic memory allocation, and array programming.

FORTRAN 90 represents a significant step forward from FORTRAN 77, incorporating many new features designed to improve the flexibility and readability of code.

1.4 Setting Up the FORTRAN Environment on Linux

Before writing any code, you need to install the GNU Fortran compiler. Most Linux distributions provide the `gfortran` package.

1.4.1 Installing GNU Fortran Compiler (gfortran)

To install `gfortran` on a Debian-based system (like Ubuntu), run:

```
$ sudo apt-get update
$ sudo apt-get install gfortran
```

For Red Hat-based systems, use:

```
$ sudo yum install gfortran
```

After installation, you can check if the compiler is installed correctly:

```
$ gfortran --version
```

1.5 Introduction to Fortran

Fortran (short for Formula Translation) is a general-purpose, imperative programming language that is particularly suited for scientific and engineering applications. It was developed in the 1950s and has since evolved into several versions, with Fortran 90 and Fortran 95 being the most widely used.

1.5.1 Basic Syntax

Fortran programs are composed of statements, which are written in a fixed-format style. Each statement begins in column 1 and can extend up to column 72. Statements are typically written in uppercase, although lowercase is also allowed.

1.5.2 Variables and Data Types

Fortran supports several data types, including integers, real numbers, complex numbers, and character strings. Variables are declared using the `INTEGER`, `REAL`, `COMPLEX`, or `CHARACTER` keywords, followed by the variable name.

1.5.3 Control Structures

Fortran provides various control structures for program flow, including `IF-THEN-ELSE` statements, `DO` loops, and `SELECT CASE` statements. These structures allow for conditional execution and repetitive tasks.

1.5.4 Arrays

Arrays are an essential part of Fortran programming. They allow you to store and manipulate multiple values of the same data type. Fortran supports both one-dimensional and multi-dimensional arrays.

1.5.5 Subroutines and Functions

Subroutines and functions are used to modularize code and improve code reusability. Subroutines are blocks of code that perform a specific task, while functions return a value.

1.5.6 File Handling

Fortran provides built-in functions and subroutines for reading from and writing to files. This allows you to interact with external data files and perform input/output operations.

1.6 Advanced Topics

Fortran also offers advanced features such as modules, derived types, and object-oriented programming. These features enhance code organization and allow for more complex programming structures.

1.7 Example Programs

1.7.1 Basic syntax

```
PROGRAM hello
  PRINT *, 'Hello, World!'
END PROGRAM hello
```


To compile the program, use the following commands:

```
$ gfortran hello.f90 -o hello
```

To run the compiled program, use:

```
$ ./hello
```

Output:

```
Hello, World!
```

1.7.2 Variables and data types

```
PROGRAM variables
  INTEGER :: i
  REAL :: x
  COMPLEX :: z
  CHARACTER(LEN=10) :: name
  LOGICAL ::

  i = 10
  x = 3.14
  z = (1.0, 2.0)
  name = 'Fortran'

  PRINT *, 'Integer:', i
  PRINT *, 'Real:', x
  PRINT *, 'Complex:', z
  PRINT *, 'Character:', name
END PROGRAM variables
```

To compile the program, use the following commands:

```
$ gfortran variables.f90 -o variables
```

To run the compiled program, use:

```
$ ./variables
```

Output:

```
Integer:           10
Real:    3.14000000
Complex:   (1.00000000,2.00000000)
Character: Fortran
```

1.7.3 Control structures

```
PROGRAM control
  INTEGER :: i
  i = 5

```

```
  IF (i > 0) THEN
    PRINT *, 'Positive'
  ELSE
    PRINT *, 'Negative'
  END IF

  DO i = 1, 5
    PRINT *, i
  END DO

  SELECT CASE (i)
    CASE (1)
      PRINT *, 'One'
    CASE (2)
      PRINT *, 'Two'
    CASE DEFAULT
      PRINT *, 'Other'
  END SELECT
END PROGRAM control
```

To compile the program, use the following commands:

```
$ gfortran control.f90 -o control
```

To run the compiled program, use:

```
$ ./control
```

Output:

```
Positive
           1
           2
           3
           4
           5
Other
```

1.7.4 Arrays

```
PROGRAM arrays
  INTEGER, DIMENSION(3) :: a
  REAL, DIMENSION(2, 2) :: b

  a = [1, 2, 3]
  b = RESHAPE([1.0, 2.0, 3.0, 4.0], [2, 2])

  PRINT *, 'Array a:', a
  PRINT *, 'Array b:'
  DO i = 1, 2
    PRINT *, b(i, :)
  END DO
END PROGRAM arrays
```


To compile the program, use the following commands:

```
$ gfortran arrays.f90 -o arrays
```

To run the compiled program, use:

```
$ ./arrays
```

Output:

```
Array a:           1           2           3
Array b:
   1.00000000        2.00000000
   3.00000000        4.00000000
```

1.7.5 Functions

The syntax of the function is given below.

```
    type FUNCTION func-name(arg1, arg2, ....)
        IMPLICIT NONE
        [specification part]
        [execution part]
        [subprogram part]
    END FUNCTION func-name
```

where 'type' is the data types like 'INTEGER', 'REAL', ... etc.

A sample code to add two numbers using Fortran function is given below:

```
program adding
        IMPLICIT NONE
        INTEGER :: a, b, addition, add
        a = 4
        b = 6
        addition = add(a, b)
        print*, a, b, addition
END PROGRAM adding

INTEGER FUNCTION add(x, y)
        IMPLICIT NONE
        INTEGER, INTENT(IN) :: x, y
        add = (x+y)
END FUNCTION add
```

Another way of writing functions in Fortran:

```
program TwoFunctions
        IMPLICIT NONE
        REAL :: a, b, A_mean, G_mean
        READ(*,*) a, b
        A_mean = ArithMean(a, b)
        G_mean = GeoMean(a, b)
        WRITE(*,*) a, b, A_mean, G_Mean
CONTAINS
        REAL FUNCTION ArithMean(a, b)
```



```
                IMPLICIT NONE
                REAL, INTENT(IN) ::a, b
                ArithMean = (a+b)/2.0
        END FUNCTION ArithMean

        REAL FUNCTION GeoMean(a, b)
                IMPLICIT NONE
                REAL, INTENT(IN) ::a, b
                GeoMean = SQRT(a*b)
        END FUNCTION GeoMean
END PROGRAM TwoFunctions
```

1.7.6 Subroutines

```
PROGRAM main
  IMPLICIT NONE
  INTEGER :: x, y, z
  x = 5
  y = 2
  CALL ADD(x, y, z)
  print*, 'ADDITION IS', z
END PROGRAM main

SUBROUTINE ADD(a, b, c)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: a, b
  INTEGER, INTENT(OUT) :: c
  c = a + b
END SUBROUTINE ADD
```

To compile the program, use the following commands:

```
$ gfortran main.f90 -o main
```

To run the compiled program, use:

```
$ ./main
```

Output:

```
Sum:           15
```

1.7.7 File handling

This code demonstrates how to write data to a file in Fortran. It opens a file, writes multiple lines to it, and then closes the file.

```
PROGRAM write_to_file
  INTEGER :: unit_number
  INTEGER :: i
  CHARACTER(len=20) :: filename

```



```
  ! Set the file name and the unit number
  filename = 'output.txt'
  unit_number = 10

  ! Open the file for writing
  OPEN(unit=unit_number, file=filename, status='unknown')

  ! Write some data into the file
  DO i = 1, 5
     WRITE(unit_number, *) 'Line number:', i
  END DO

  ! Close the file
  CLOSE(unit_number)

  PRINT *, 'Data has been written to ', filename
END PROGRAM write_to_file
```

Explanation

- `PROGRAM write_to_file`: The program starts with a main program block named `write_to_file`.
- `INTEGER :: unit_number, i`: The variables `unit_number` and `i` are declared as integers. `unit_number` represents the file identifier, and `i` is used in the loop.
- `CHARACTER(len=20) :: filename`: This declares a character variable `filename` with a length of 20 characters to store the name of the file.
- `OPEN(unit=unit_number, file=filename, status='unknown')`: Opens the file `output.txt` with the file unit specified by `unit_number`. The `status='unknown'` allows Fortran to create the file if it doesn't exist or overwrite it if it already exists.
- `DO i = 1, 5`: This loop runs from 1 to 5, writing a line to the file in each iteration.
- `WRITE(unit_number, *) 'Line number:', i`: This statement writes the text `'Line number:'` followed by the value of `i` to the file.
- `CLOSE(unit_number)`: Closes the file associated with `unit_number`.

To compile the program, use the following commands:

```
$ gfortran file_handling.f90 -o file_handling
```

To run the compiled program, use:

```
$ ./file_handling
```

These example programs demonstrate the basic syntax, variables, control structures, arrays, subroutines, functions, and file handling in Fortran programming. By understanding these concepts, you can start writing your own Fortran programs for scientific and engineering applications.

1.8 Linking external libraries

Linking to external libraries in Fortran is a common task when you want to leverage precompiled libraries such as LAPACK, BLAS, or others for numerical and scientific computations. This document will explain the steps to link Fortran programs with external libraries using the GNU Fortran compiler (`gfortran`), and provide an example of linking to the LAPACK library.

1.8.1 Steps to Link to External Libraries

To link an external library to your Fortran program, follow these steps:

1. **Install the necessary libraries:** Ensure that the external library is installed on your system. For example, you can install LAPACK and BLAS on Linux using the following command:

```
    sudo apt-get install liblapack-dev libblas-dev

```

2. **Compile the Fortran code:** Use the `gfortran` compiler to compile your Fortran code and link it to the library using the `-l` option.

3. **Link during compilation:** Use the `-L` option to specify the path to the external library and the `-l` option to link against the library.

1.8.2 Example: Solving a Linear System using LAPACK

Below is an example Fortran program that solves a system of linear equations $Ax = b$ using the LAPACK routine `dgesv`, which performs LU decomposition.

1.8.3 Fortran Code

```
PROGRAM solve_linear_system
  USE, INTRINSIC :: iso_c_binding
  IMPLICIT NONE

  INTEGER, PARAMETER :: n = 3
  INTEGER :: info
  REAL(KIND=c_double), DIMENSION(n,n) :: A
  REAL(KIND=c_double), DIMENSION(n) :: B
  INTEGER, DIMENSION(n) :: ipiv

  ! Matrix A (3x3)
  A = RESHAPE([3.0d0, 1.0d0, 2.0d0, &
               6.0d0, 3.0d0, 4.0d0, &
               9.0d0, 5.0d0, 8.0d0], [n,n])

  ! Right-hand side vector B (3x1)
  B = [1.0d0, 0.0d0, 2.0d0]

```



```
19    ! Call LAPACK subroutine to solve the system of equations A*x =
        B
20    CALL dgesv(n, 1, A, n, ipiv, B, n, info)
21
22    ! Check for errors
23    IF (info /= 0) THEN
24      PRINT *, 'Error: LAPACK dgesv failed with info =', info
25    ELSE
26      PRINT *, 'Solution vector X:'
27      PRINT *, B
28    END IF
29
30  END PROGRAM solve_linear_system
```

In this program:

- The matrix A is a 3x3 matrix, and B is a 3x1 vector. The goal is to solve $Ax = B$ for the unknown vector x.
- `dgesv` is the LAPACK routine that performs the LU decomposition and solves the system of equations.

Explanation

- `PROGRAM solve_linear_system`: This statement starts the main program block named `solve_linear_system`.
- `USE, INTRINSIC :: iso_c_binding`: This module provides definitions for interoperability with C, particularly for specifying precision with `c_double`.
- `IMPLICIT NONE`: This directive requires explicit declaration of all variables, helping to avoid errors due to undeclared variables.
- `INTEGER, PARAMETER :: n = 3`: This declares an integer parameter `n` with a value of 3, representing the size of the matrix and vector.
- `INTEGER :: info`: This integer variable will store the error information returned by the LAPACK subroutine.
- `REAL(KIND=c_double), DIMENSION(n,n) :: A`: Declares a 3x3 matrix `A` of type `REAL(KIND=c_double)` for high-precision floating-point numbers.
- `REAL(KIND=c_double), DIMENSION(n) :: B`: Declares a 3x1 vector `B` of the same floating-point type.
- `INTEGER, DIMENSION(n) :: ipiv`: Declares an integer array `ipiv` used by the LAPACK subroutine to store pivot indices.
- `A = RESHAPE([...] [,n,n])`: Initializes the matrix `A` with specific values and reshapes it to a 3x3 matrix.
- `B = [1.0d0, 0.0d0, 2.0d0]`: Initializes the vector `B` with given values.

- `CALL dgesv(n, 1, A, n, ipiv, B, n, info)`: Calls the LAPACK subroutine `dgesv` to solve the system of linear equations `A*x = B`. Here, `n` is the size of the matrix, `1` is the number of right-hand sides, `A` is the coefficient matrix, `ipiv` is the pivot index array, `B` is the right-hand side vector, and `info` will hold the exit status.

- `IF (info /= 0)`: Checks if the LAPACK subroutine encountered an error. If `info` is not zero, an error message is printed.

- `PRINT *, 'Solution vector X:'`: If there is no error, the solution vector `B` is printed, which contains the solution to the system.

- `END PROGRAM solve_linear_system`: Ends the program.

1.8.4 Compilation and Linking

To compile and link the program with the LAPACK and BLAS libraries, use the following commands:

```
gfortran solve_linear_system.f90 -o solve_linear_system -llapack
   -lblas
```

Here:

- `-llapack` links the LAPACK library.

- `-lblas` links the BLAS library, which is a prerequisite for LAPACK.

If the libraries are not in the default location, you can specify the path using the `-L` option:

```
gfortran solve_linear_system.f90 -o solve_linear_system -L/usr/
   local/lib -llapack -lblas
```

1.8.5 Running the Program

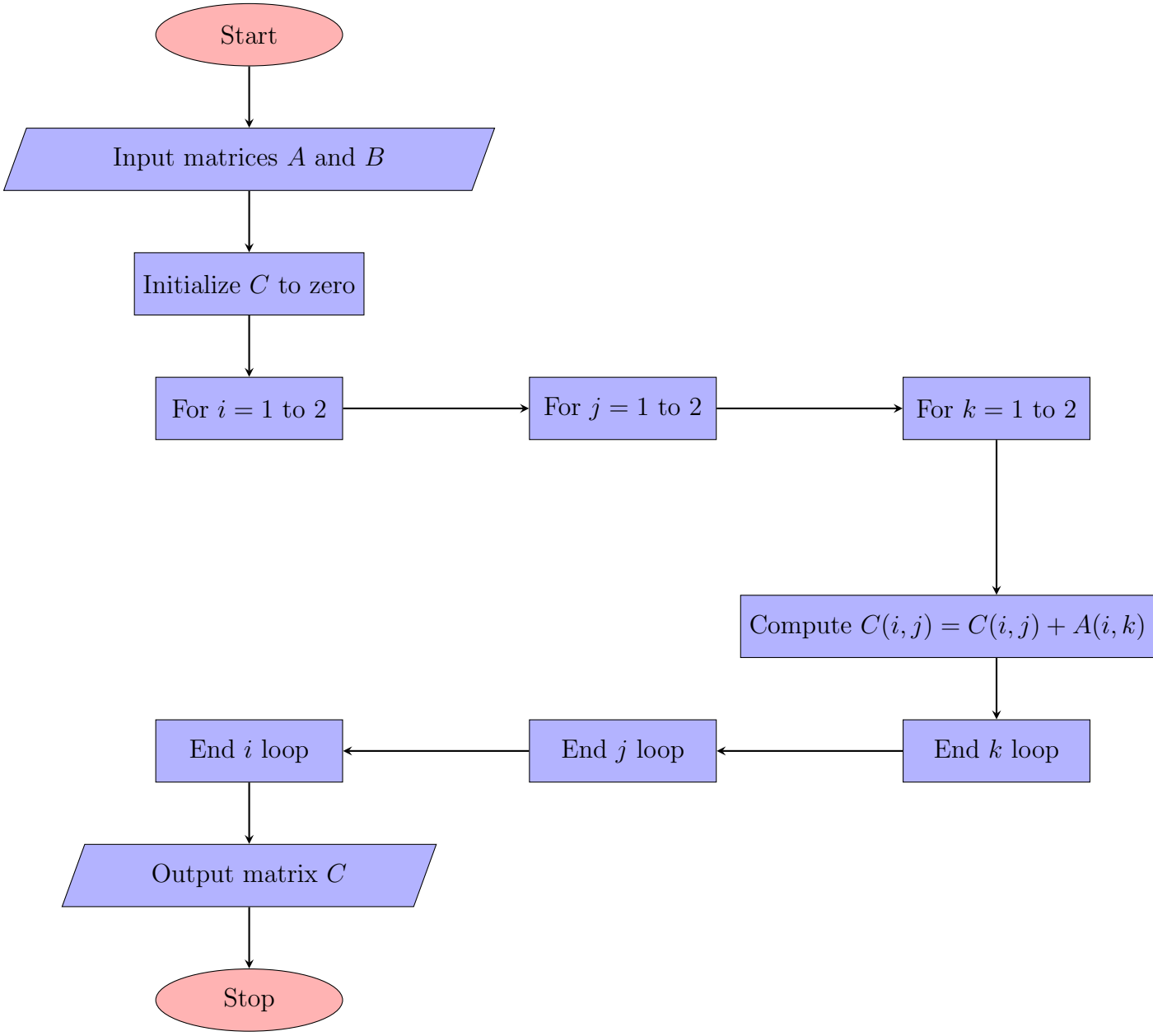
Once the program is compiled, you can run it using:

```
./solve_linear_system
```

The output will display the solution vector x for the system $Ax = b$.

1.9 Matrix Multiplication of size 2x2

1.9.1 Flowchart



1.9.2 Code

```
program matrix_multiplication
    implicit none
    real :: A(2, 2), B(2, 2), C(2, 2)

    ! Input matrices
    A = reshape([1.0, 2.0, 3.0, 4.0], shape(A))  ! Matrix A
    B = reshape([5.0, 6.0, 7.0, 8.0], shape(B))  ! Matrix B

```

```
    ! Call the function to multiply A and B, storing the result
   in C
    C = matrix_multiply(A, B)

    ! Output the result
    print *, "Matrix A:"
    call print_matrix(A)
    print *, "Matrix B:"
    call print_matrix(B)
    print *, "Resultant Matrix C (A * B):"
    call print_matrix(C)

contains

    ! Function to multiply two 2x2 matrices
    function matrix_multiply(A, B) result(C)
        implicit none
        real, intent(in) :: A(2, 2), B(2, 2)
        real :: C(2, 2)
        integer :: i, j, k

        ! Initialize the result matrix C to zero
        C = 0.0

        ! Perform matrix multiplication
        do i = 1, 2
            do j = 1, 2
                do k = 1, 2
                    C(i, j) = C(i, j) + A(i, k) * B(k, j)
                end do
            end do
        end do
    end function matrix_multiply

    ! Subroutine to print a 2x2 matrix
    subroutine print_matrix(M)
        implicit none
        real, intent(in) :: M(2, 2)
        integer :: i

        do i = 1, 2
            write(*, '(F6.2, F6.2)') M(i, 1), M(i, 2)
        end do
    end subroutine print_matrix

end program matrix_multiplication
```

1.10 Conclusion

This chapter introduced you to the basics of working with Linux and FORTRAN 90. You learned how to navigate the Linux file system, write a simple "Hello, World!" program, and compile and execute FORTRAN code using the `gfortran` compiler. In subsequent chapters, we'll dive deeper into advanced FORTRAN features such as arrays, file handling, and scientific computing techniques.

Exercise

Category 1: Easy (Conceptual and Memory-Based)

1. What makes Linux a preferred operating system for scientific computing?
2. Why is FORTRAN still relevant for numerical and scientific applications in the modern era?
3. What is the primary purpose of the `gfortran` compiler in FORTRAN programming?
4. Explain the difference between fixed-format and free-format styles in FORTRAN.
5. Why is the `IMPLICIT NONE` directive critical for error-free programming in FORTRAN?
6. What does the `RESHAPE` function do in FORTRAN? Provide an example scenario.
7. Define the purpose of modules in FORTRAN. How do they improve code organization?
8. Why are control structures such as `DO` loops important for computational tasks?
9. Briefly explain the role of the `SELECT CASE` statement in FORTRAN programs.
10. How does the Linux `nano` editor help in writing and editing FORTRAN code?

Category 2: Mid-Level (Understanding-Based)

1. Compare and contrast FORTRAN's built-in file-handling features with those in other programming languages.
2. Write a FORTRAN code snippet that reads two real numbers from a file and prints their sum.
3. How does FORTRAN's array indexing differ from Python's? What advantages does this provide in scientific computing?
4. Write a program to determine whether a given integer is even or odd using FORTRAN.
5. Describe how you would use FORTRAN to simulate the temperature distribution in a rod (hint: use arrays).
6. Explain how LAPACK can be used to solve a set of linear equations in FORTRAN. Why is linking external libraries beneficial?
7. Design a FORTRAN program to compute the factorial of a number using recursion.
8. How would you modify a FORTRAN program to store the computed results in a text file? Provide a pseudocode outline.

9. Describe how you would debug a FORTRAN program using Linux tools like `gdb` or compiler flags.
10. Explain the role of logical operators in FORTRAN with an example of their use in a physical simulation.

Category 3: Application-Based (Flowchart and Coding for Physics)

1. Write the algorithm and draw a flowchart to compute the determinant of a 3x3 matrix. Implement it in FORTRAN.
2. Create a flowchart and write a FORTRAN program to compute the trajectory of a projectile given its initial velocity and angle.
3. Develop an algorithm and flowchart for simulating the motion of a harmonic oscillator using Euler's method.
4. Create a flowchart and program in FORTRAN to calculate the area under a curve using the trapezoidal rule.
5. Write an algorithm and create a flowchart to compute the orbital velocity of a planet given its distance from the Sun. Implement the solution in FORTRAN.
6. Create a flowchart and write a FORTRAN program to solve the one-dimensional heat equation using finite differences.
7. Design a flowchart and write FORTRAN code to compute the discrete Fourier transform of a signal.
8. Write an algorithm and create a flowchart for calculating the electric field at a point due to multiple charges in 2D space.
9. Create a flowchart and program in FORTRAN to simulate a 2D random walk for a particle.
10. Write an algorithm and flowchart to calculate the energy levels of an electron in a one-dimensional potential well using the Schrödinger equation.

Chapter 2

Introduction to C++

C++ is a general-purpose programming language created as an extension of C by Bjarne Stroustrup in the early 1980s. It supports both procedural and object-oriented programming paradigms, making it versatile for systems programming, game development, and real-time applications.

2.1 Basic Syntax

C++ programs consist of statements that are grouped into functions and classes. The main function, `int main()`, is the starting point of any C++ program. Statements in C++ are terminated by semicolons, and the language is case-sensitive.

2.2 Variables and Data Types

C++ supports several basic data types, such as integers (`int`), floating-point numbers (`float`, `double`), characters (`char`), and booleans (`bool`). Variables are declared by specifying the type followed by the variable name.

2.3 Control Structures

C++ provides control structures like `if-else`, `switch-case`, loops (`for`, `while`, and `do-while`), and `goto` statements for controlling the flow of the program.

2.4 Functions

Functions in C++ allow for code reuse and modularization. A function is defined by specifying a return type, a name, and a list of parameters. The function body is enclosed in curly braces {}.

2.5 Arrays and Vectors

Arrays in C++ are a collection of elements of the same type. They are declared with a fixed size and can be single or multi-dimensional. Vectors, from the Standard Template Library (STL), offer dynamic sizing and more flexibility than arrays.

2.6 Object-Oriented Programming (OOP)

C++ is known for its support of object-oriented programming. It introduces the concepts of classes and objects, inheritance, polymorphism, encapsulation, and abstraction, which allow for modeling real-world entities in a more intuitive way.

2.7 File Handling

C++ provides file handling mechanisms through the `fstream` library. You can read from and write to files using `ifstream` (input file stream) and `ofstream` (output file stream).

2.8 Advanced Topics

Advanced features of C++ include templates, exception handling, operator overloading, and the Standard Template Library (STL) for generic programming. These features provide flexibility and efficiency in coding.

2.9 Example Programs

2.9.1 Basic syntax

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

To compile the program, use the following commands:

```
$ g++ hello.cpp -o hello
```

To run the compiled program, use:

```
$ ./hello
```

Output:

```
Hello, World!
```

2.9.2 Variables and data types

```cpp
#include <iostream>
using namespace std;

int main() {
    int i = 10;
    float f = 3.14;
```



```
    char c = 'A';
    bool b = true;

    cout << "Integer: " << i << endl;
    cout << "Float: " << f << endl;
    cout << "Character: " << c << endl;
    cout << "Boolean: " << b << endl;

    return 0;
}
```

To compile the program, use the following commands:

```
$ g++ variables.cpp -o variables
```

To run the compiled program, use:

```
$ ./variables
```

Output:

```
Integer: 10
Float: 3.14
Character: A
Boolean: 1
```

2.9.3 Control structures

```
#include <iostream>
using namespace std;

int main() {
    int i = 5;

    if (i > 0) {
        cout << "Positive" << endl;
    } else {
        cout << "Negative" << endl;
    }

    for (int j = 1; j <= 5; j++) {
        cout << j << endl;
    }
    return 0;
}
```

To compile the program, use the following commands:

```
$ g++ control.cpp -o control
```

To run the compiled program, use:

```
$ ./control
```

Output:

```
Positive
1
2
3
4
5
```

switch case:

```
#include <iostream>
using namespace std;

int main() {
    int i = 5;

    switch(i) {
        case 1:
            cout << "One" << endl;
            break;
        case 2:
            cout << "Two" << endl;
            break;
        default:
            cout << "Other" << endl;
    }

    return 0;
}
```

To compile the program, use the following commands:

```
$ g++ control_1.cpp -o control_1
```

To run the compiled program, use:

```
$ ./control_1
```

Output:

```
Other
```

2.9.4 Arrays and vectors

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    int arr[3] = {1, 2, 3};
    vector<int> vec = {1, 2, 3, 4};
```

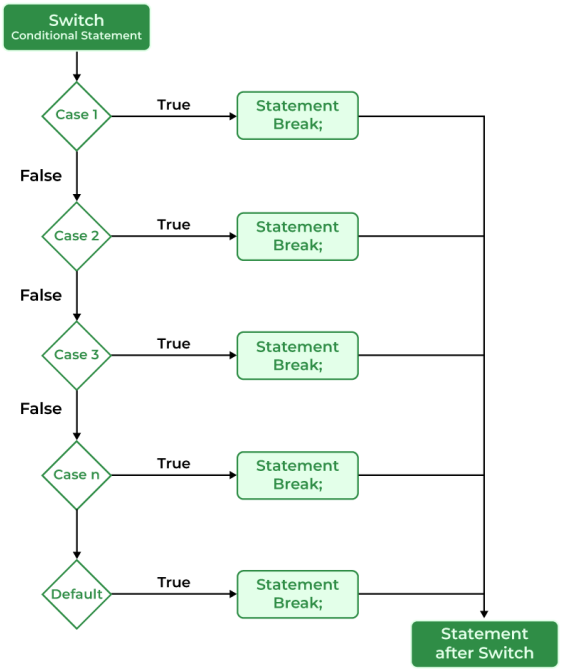


Figure 2.1: Switch case statements

```
8
9       cout << "Array elements: ";
10      for (int i = 0; i < 3; i++) {
11          cout << arr[i] << " ";
12      }
13      cout << endl;
14
15      cout << "Vector elements: ";
16      for (int i = 0; i < vec.size(); i++) {
17          cout << vec[i] << " ";
18      }
19      cout << endl;
20
21      return 0;
22  }
```

To compile the program, use the following commands:

```
1  $ g++ arrays.cpp -o arrays
```

To run the compiled program, use:

```
1  $ ./arrays
```

Output:

```
1  Array elements: 1 2 3
2  Vector elements: 1 2 3 4
```


2.9.5 Functions

```cpp
#include <iostream>
using namespace std;

int add(int a, int b) {
    return a + b;
}

int main() {
    int x = 5, y = 10;
    int sum = add(x, y);
    cout << "Sum: " << sum << endl;
    return 0;
}
```

To compile the program, use the following commands:

```
$ g++ functions.cpp -o functions
```

To run the compiled program, use:

```
$ ./functions
```

Output:

```
Sum: 15
```

2.9.6 File handling

```cpp
// C++ Program to Read a File Line by Line using ifstream
#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    // Open the file "abc.txt" for reading
    ifstream inputFile("abc.txt");

    // Variable to store each line from the file
    string line;

    // Read each line from the file and print it
    while (getline(inputFile, line)) {
        // Process each line as needed
        cout << line << endl;
    }

    // Always close the file when done
```

```
    inputFile.close();

    return 0;
}
```

To compile the program, use the following commands:

```
$ g++ file_handling.cpp -o file_handling
```

To run the compiled program, use:

```
$ ./file_handling
```

The example program to write into a file in C++ using 'ofstream':

```
#include <iostream>
#include <fstream>  // Required for file handling
using namespace std;

int main() {
    // Declare an output file stream (ofstream) object
    ofstream outputFile;

    // Open a file named "example.txt"
    outputFile.open("example.txt");

    // Check if the file opened successfully
    if (!outputFile) {
        cout << "Error opening file!" << endl;
        return 1;  // Exit the program with an error code
    }

    // Write data to the file
    outputFile << "This is a simple example of writing to a file
   in C++.\n";
    outputFile << "File handling is important for many
   applications.\n";
    outputFile << "Learning how to write and read files is
   essential!\n";

    // Close the file
    outputFile.close();

    // Notify the user
    cout << "Data successfully written to the file!" << endl;

    return 0;
}
```

2.10 Pointers in C++

A pointer in C++ is a variable that stores the memory address of another variable. Pointers are widely used in C++ for dynamic memory management, passing parameters

by reference, and for working with arrays and data structures.

The syntax for declaring a pointer is:

```
type* pointer_name = &var_name;
```

Here, type refers to the data type that the pointer will point to.
Key operations with pointers:

- **Address-of operator** (`&`): Used to get the address of a variable.
- **Dereference operator** (`*`): Used to access the value stored at the address the pointer holds.

2.10.1 Examples

```
// Example 1: Basic pointer usage
#include <iostream>
using namespace std;

int main() {
    int var = 10;
    int* ptr = &var;  // Pointer to var

    cout << "Value of var: " << var << endl;
    cout << "Address of var: " << &var << endl;
    cout << "Value stored in ptr (address of var): " << ptr << endl;
    cout << "Dereferencing ptr to get value of var: " << *ptr << endl;

    return 0;
}
```

2.11 Arrays in C++

An array in C++ is a collection of elements of the same data type, stored in contiguous memory locations. Arrays can be accessed using index values starting from 0.

The syntax for declaring an array is:

```
type array_name[size] = {_, _, ...};
```

Important properties of arrays:

- Arrays can store multiple values in a single variable.
- The elements in an array are stored in contiguous memory locations.
- Arrays can be passed to functions by reference, meaning the memory address of the first element is passed.

2.11.1 Examples

```cpp
// Example 2: Working with arrays
#include <iostream>
using namespace std;

int main() {
    int arr[5] = {1, 2, 3, 4, 5};

    // Accessing array elements using indices
    cout << "First element: " << arr[0] << endl;
    cout << "Third element: " << arr[2] << endl;

    // Using a loop to print all elements
    for(int i = 0; i < 5; i++) {
        cout << "Element at index " << i << ": " << arr[i] << endl;
    }

    return 0;
}
```

2.12 Pointers and Arrays

In C++, arrays and pointers are closely related. The name of an array acts as a pointer to the first element of the array. This allows for pointer arithmetic and manipulation of array elements via pointers.

2.12.1 Examples

```cpp
// Example 3: Pointers and arrays
#include <iostream>
using namespace std;

int main() {
    int arr[3] = {10, 20, 30};
    int* ptr = arr;  // ptr points to the first element of the array

    // Accessing array elements via pointer
    for (int i = 0; i < 3; i++) {
        cout << "Element " << i << ": " << arr[i] << endl;
        cout << "Element " << i << ": " << *(ptr + i) << endl;
    }

    return 0;
}
```

2.13 Dynamic list Example

In this example, we need to manage the scores of a class of students. Since the number of students is unknown at compile-time, we will use dynamic memory allocation to create an array to store their scores at runtime. This demonstrates how pointers are used in dynamic memory management.

```
// Example: Managing a dynamic list of student scores
#include <iostream>
using namespace std;

int main() {
    int numStudents;

    // Asking the user for the number of students
    cout << "Enter the number of students: ";
    cin >> numStudents;

    // Dynamically allocating an array to store student scores
    float* scores = new float[numStudents];

    // Taking input for student scores
    for(int i = 0; i < numStudents; ++i) {
        cout << "Enter score for student " << i+1 << ": ";
        cin >> scores[i];
    }

    // Calculating the average score
    float sum = 0;
    for(int i = 0; i < numStudents; ++i) {
        sum += scores[i];
    }
    float average = sum / numStudents;

    // Displaying the average score
    cout << "Average score: " << average << endl;

    // Freeing the dynamically allocated memory
    delete[] scores;

    return 0;
}
```

2.13.1 Explanation

In this program, the number of students is provided by the user at runtime. The program dynamically allocates memory for the student scores using a pointer. The key steps are as follows:

- **Dynamic Memory Allocation**: `new float[numStudents]` allocates memory for an array of floats based on the number of students entered by the user. This is useful

when the size of data is not known during compile-time.

- **Pointer Usage**: The pointer `scores` stores the address of the first element of the dynamically allocated array. The notation `scores[i]` is used to access the array elements. This is equivalent to `*(scores + i)`, where pointer arithmetic is applied to traverse the memory.
- **Memory Deallocation**: The program uses `delete[]` to free the dynamically allocated memory after it is no longer needed, preventing memory leaks.

2.14 Significance of Using Pointers

Pointers in C++ are crucial for dynamic memory management, which provides several benefits in real-world applications:

- **Efficient Memory Usage**: Pointers allow for dynamic allocation of memory, meaning we can allocate memory based on actual needs at runtime. This avoids wastage of memory that occurs when arrays are declared with a fixed size at compile-time.
- **Flexibility**: Since the size of the array is determined at runtime, the program can handle variable amounts of data. This is particularly important when dealing with user input or data that fluctuates during program execution.
- **Performance**: Pointers can directly access and manipulate memory, making them more efficient in scenarios where performance is critical, such as handling large datasets, network buffers, or game engines.
- **Dynamic Data Structures**: Many advanced data structures like linked lists, trees, and graphs rely on pointers to manage memory and relationships between elements. These structures are widely used in algorithm design and systems programming.

Exercise

Category 1: Easy (Conceptual and Memory-Based)

1. What are the main features of C++ that make it suitable for systems programming and real-time applications?
2. Briefly explain the difference between procedural programming and object-oriented programming.
3. What is the purpose of the `int main()` function in a C++ program?
4. List the basic data types in C++ and provide an example of how to declare each.
5. What is the significance of the `#include` directive in C++?
6. Why is the semicolon (`;`) important in C++? Provide an example of its usage.
7. What is the difference between an array and a vector in C++?
8. Describe the role of the Standard Template Library (STL) in C++.
9. What are the key features of object-oriented programming supported by C++?
10. Explain the importance of the `fstream` library in C++ file handling.

Category 2: Mid-Level (Understanding-Based)

1. Write a simple C++ program to print the Fibonacci sequence up to `n` terms, where `n` is input by the user.
2. Describe how a `switch-case` statement works in C++. Provide an example of its usage.
3. Compare and contrast `for`, `while`, and `do-while` loops in C++.
4. Write a C++ function to calculate the factorial of a number using recursion.
5. How do pointers work in C++? Write a program to demonstrate the use of pointers to access and modify an integer variable.
6. Explain how to dynamically allocate and deallocate memory for an array in C++ using pointers.
7. Describe the difference between `ifstream` and `ofstream`. Provide an example of each.
8. Write a program to demonstrate the usage of vectors in C++ for storing and manipulating a list of integers.
9. How does operator overloading work in C++? Write a program to overload the `+` operator for adding two complex numbers.
10. Explain the use of templates in C++ with an example of a function template for swapping two variables.

Category 3: Application-Based (Flowchart and Coding)

1. Write the algorithm and draw a flowchart to compute the roots of a quadratic equation using the quadratic formula. Implement it in C++.
2. Create a flowchart and write a program to simulate the motion of a pendulum using simple harmonic motion equations.
3. Develop an algorithm and flowchart to compute the dot product of two vectors. Implement the solution in C++.
4. Create a flowchart and program to simulate the motion of a projectile given initial velocity and angle of projection.
5. Write a program to solve a system of linear equations using matrices and Gaussian elimination. Provide the algorithm and flowchart.
6. Develop an algorithm and flowchart to compute the numerical integration of a function using Simpson's rule. Write the corresponding C++ code.
7. Write the algorithm and create a flowchart to calculate the electric field at a point due to multiple charges in 2D space. Implement it in C++.
8. Design a flowchart and write a C++ program to simulate a simple 2D random walk of a particle.
9. Create an algorithm and flowchart for managing a dynamic list of student scores, including input, average calculation, and memory deallocation. Implement it in C++.
10. Write an algorithm and flowchart for generating the first `n` terms of a geometric progression, then implement the program in C++.

Chapter 3

Introduction to Gnuplot

3.1 Overview

Gnuplot is a portable command-line driven graphing utility for visualizing mathematical functions and data. It supports various types of plots in both 2D and 3D and can output to multiple formats, including PNG, PDF, SVG, and LaTeX. Gnuplot is widely used for its flexibility and ability to produce publication-quality graphics. Its versatility makes it a favorite among scientists, engineers, and data analysts for creating detailed and customizable visualizations.

3.2 Getting Started with Gnuplot

To begin using Gnuplot, ensure it is installed on your system. You can download it from the official website: http://www.gnuplot.info/. After installation, launch the Gnuplot command-line interface by typing `gnuplot` in your terminal.

3.3 Plotting Mathematical Functions

Gnuplot allows for straightforward plotting of mathematical functions. For example, to plot the sine function:

```
gnuplot> plot sin(x)
```

This command will display a 2D plot of $\sin(x)$ over a default range. To specify a range for the x-axis:

```
gnuplot> plot [-10:10] sin(x)
```

You can also plot more complex functions, such as:

```
gnuplot> plot sin(x)/x
```

This will plot the sinc function, which is commonly used in signal processing.

3.4 Plotting Data from Files

Gnuplot can plot data from files where data is organized in columns. Consider a data file named `data.dat` with the following content:

```
# X    Y
1    2
2    4
3    6
4    8
5   10
```

To plot this data:

```
gnuplot> plot 'data.dat' using 1:2 with linespoints
```

This command tells Gnuplot to plot the first column as the x-axis and the second column as the y-axis, using lines and points to represent the data.

3.5 Customizing Plots

Gnuplot offers various customization options:

- **Titles and Labels**: Add titles and axis labels to your plot.

```
    gnuplot> set title "Sample Data Plot"
    gnuplot> set xlabel "X-axis"
    gnuplot> set ylabel "Y-axis"

```

- **Grid and Key (Legend)**: Enable grid lines and position the legend.

```
    gnuplot> set grid
    gnuplot> set key right top

```

- **Output to Files**: Save plots to files in various formats.

```
    gnuplot> set terminal png
    gnuplot> set output 'plot.png'
    gnuplot> replot
    gnuplot> set output

```

- **Line Styles and Colors**: Customize the appearance of your plots by changing line styles and colors.

```
    gnuplot> plot sin(x) with lines linecolor rgb "blue"
   linewidth 2

```

- **Multiple Plots**: Create multiplot layouts to display several plots in one figure.

```
    gnuplot> set multiplot layout 2,1
    gnuplot> set title 'sin(x)'
    gnuplot> plot sin(x)
```



```
    gnuplot> set title 'cos(x)'
    gnuplot> plot cos(x)
    gnuplot> unset multiplot

```

3.6 Advanced Plotting Techniques

Example 1: Plotting Multiple Functions

To plot multiple functions on the same graph:

```
gnuplot> plot sin(x) title `sin(x)`, cos(x) title `cos(x)'
```

This command plots both $\sin(x)$ and $\cos(x)$ with respective titles.

Example 2: 3D Plotting

Gnuplot can create 3D plots using the `splot` command:

```
gnuplot> set hidden3d
gnuplot> splot sin(x)*cos(y) title `sin(x)cos(y)'
```

This will render a 3D surface plot of the function $\sin(x)\cos(y)$. You can also add contour lines to your 3D plot:

```
gnuplot> set contour
gnuplot> splot sin(x)*cos(y)
```

Example 3: Plotting Data with Error Bars

If your data file `data_with_errors.dat` includes errors:

```
# X   Y   Y_Error
1   2   0.1
2   4   0.2
3   6   0.1
4   8   0.3
5  10   0.2
```

Plot with error bars using:

```
gnuplot> plot "data_with_errors.dat" using 1:2:3 with yerrorbars
```

This command plots the data points with vertical error bars.

Example 4: Histograms

Gnuplot can also create histograms. Suppose you have a data file `histogram.dat`:

```
# Bin   Frequency
1   5
2   10
3   7
```



```
4    12
5    8
```

To create a histogram:

```
gnuplot> set style data histograms
gnuplot> plot 'histogram.dat' using 2:xtic(1)
```

This will plot a histogram with the frequency on the y-axis and the bin numbers on the x-axis.

Example 5: Polar Plots

Gnuplot supports polar plots, which are useful for visualizing data in polar coordinates. To create a polar plot:

```
gnuplot> set polar
gnuplot> set grid polar
gnuplot> plot sin(2*t)
```

This will plot the function $\sin(2t)$ in polar coordinates.

Example 6: Animations

Gnuplot can create animations by generating a series of plots and combining them into an animated GIF. For example, to create an animation of a sine wave:

```
gnuplot> set terminal gif animate
gnuplot> set output 'sine_wave.gif'
gnuplot> do for [i=0:100] {
  plot sin(x + i*0.1)
}
gnuplot> set output
```

This script generates 100 frames of a sine wave with a shifting phase and saves them as an animated GIF.

Gnuplot is a powerful tool for creating a wide variety of plots and visualizations. Its flexibility and customization options make it suitable for both simple and complex plotting tasks. Whether you are plotting mathematical functions, data from files, or creating advanced 3D plots and animations, Gnuplot provides the tools you need to produce high-quality graphics. With practice, you can unlock the full potential of Gnuplot and create stunning visualizations for your data and research.

3.7 Data Analysis using Gnuplot

Gnuplot is a versatile command-line tool for data visualization and analysis. It is widely used in scientific research for generating high-quality plots and performing advanced data analysis.

3.7.1 Installation

To install Gnuplot, use:

```
sudo apt-get install gnuplot  # On Debian-based systems
brew install gnuplot          # On macOS with Homebrew
```

3.7.2 Basic Usage

To quickly visualize a mathematical function, use:

```
gnuplot> plot sin(x)
```

This plots the sine function over the default range.

3.7.3 Example 1: Plotting Data from a File

Suppose we have a data file `data.txt` with the following content:

```
# X Y
1 2
2 3
3 5
4 4
5 6
```

Plot the data with lines and points:

```
gnuplot> plot 'data.txt' using 1:2 with linespoints title 'Data
   Plot'
```

3.7.4 Example 2: Fitting a Curve to Data

Gnuplot supports fitting functions to data using non-linear least squares fitting:

```
gnuplot> f(x) = a * x + b
gnuplot> fit f(x) 'data.txt' using 1:2 via a, b
gnuplot> plot 'data.txt' using 1:2 title 'Data', f(x) title '
   Fitted Line'
```

3.7.5 Example 3: 3D Data Visualization

For 3D data, use the `splot` command:

```
gnuplot> splot '3d_data.txt' using 1:2:3 with points palette
```

3.7.6 Example 4: Histogram Plotting

To create histograms, Gnuplot can bin data using `smooth freq`:


```
gnuplot> set style data histograms
gnuplot> set style fill solid border -1
gnuplot> plot 'data.txt' using 2:xtic(1) smooth freq with boxes
   title 'Histogram'
```

3.7.7 Example 5: Statistical Analysis

Gnuplot can calculate basic statistics like mean and standard deviation:

```
gnuplot> stats 'data.txt' using 2 name 'A'
gnuplot> print "Mean: ", A_mean
gnuplot> print "Standard Deviation: ", A_stddev
```

3.7.8 Example 6: Data Transformation and Scripting

Gnuplot can perform in-line data transformations:

```
gnuplot> plot 'data.txt' using 1:($2**2) with lines title '
   Squared Y values'
```

For scripting, save commands in a file (e.g., `script.gp`) and run:

```
gnuplot script.gp
```

3.7.9 Example 7: Multiplot Layouts

To create a grid of plots in a single output:

```
gnuplot> set multiplot layout 2,2 title "Multiplot Example"
gnuplot> plot sin(x) title 'Sine'
gnuplot> plot cos(x) title 'Cosine'
gnuplot> plot tan(x) title 'Tangent'
gnuplot> plot x**2 title 'Parabola'
gnuplot> unset multiplot
```

3.7.10 Example 8: Heatmaps

To visualize data density, Gnuplot offers heatmaps:

```
gnuplot> set pm3d map
gnuplot> splot 'heatmap_data.txt' matrix with pm3d
```

3.7.11 Example 9: Error Bars

For datasets with uncertainties, use error bars:

```
gnuplot> plot 'error_data.txt' using 1:2:3 with yerrorbars title
   'Data with Errors'
```


3.7.12 Example 10: Exporting Plots

To export a plot as a PNG image:

```
gnuplot> set terminal pngcairo
gnuplot> set output 'plot.png'
gnuplot> plot 'data.txt' using 1:2 with lines title 'Data Plot'
gnuplot> set output
```

You can also export to other formats like PDF, SVG, or EPS by changing the terminal type.

3.8 Statistical Data Analysis with GNUPLOT

Gnuplot is a versatile plotting utility that, beyond generating graphs, can perform basic statistical analysis. In these notes, we explain how to:

- Compute basic statistics such as maximum, minimum, and mean.
- Perform simple (univariate) regression.
- Fit multivariate regression models.
- Smooth data to reduce noise and reveal trends.

Each section includes code snippets that can be executed in Gnuplot.

3.9 Basic Statistical Analysis

Gnuplot's `stats` command processes data files to compute various statistics.

3.9.1 Example Data File

Suppose we have a data file `data.txt` containing a single column of numbers:

```
1.2
3.4
2.5
4.0
0.9
```

3.9.2 Computing Minimum, Maximum, and Mean

The `stats` command computes several statistics and assigns them to variables. For example:

```
stats "data.txt" using 1 name "A"
print A_min    # Displays the minimum value
print A_max    # Displays the maximum value
print A_mean   # Displays the mean (average)
```

Here, the prefix `A` is used to store the computed values, such as:

- `A_min`: Minimum value.
- `A_max`: Maximum value.
- `A_mean`: Mean of the data.

3.10 Regression Analysis

Gnuplot can fit functions to data. We discuss both simple linear regression and an example of multivariate regression.

3.10.1 Simple Regression

Simple regression fits a straight line to data. Consider a data file `regression.txt` with two columns:

```
# x    y
1    2.1
2    3.9
3    6.1
4    8.0
5    10.2
```

Fitting a Line

Define a linear function and use the `fit` command to determine its parameters:

```
f(x) = a*x + b
fit f(x) "regression.txt" using 1:2 via a,b
```

After fitting, you can plot both the raw data and the fitted function:

```
plot "regression.txt" using 1:2 title "Data" with points, \
     f(x) title sprintf("Fit: y = %.2fx + %.2f", a, b) with lines
```

3.10.2 Multivariate Regression

While Gnuplot is primarily a plotting tool, it can also handle models with more than one independent variable. Consider a data file `multivar.txt` with three columns:

```
# x    y    z
1    2    3.1
2    3    5.8
3    4    8.2
4    5   10.5
5    6   13.0
```

Assume a linear model of the form:

$$z = a\,x + b\,y + c$$

Define the model in Gnuplot and fit the data:


```
model(x,y) = a*x + b*y + c
fit model(x,y) "multivar.txt" using 1:2:3 via a,b,c
```

After the fit, the variables `a`, `b`, and `c` will contain the optimized parameters.

Notes on Multivariate Regression

Keep in mind that while Gnuplot can handle such fits, it is not designed as a full-fledged statistical analysis package. For more complex multivariate analyses, consider using software like R or Python with libraries such as `statsmodels`.

3.11 Data Smoothing

Data smoothing helps reduce noise and highlight underlying trends. Gnuplot provides several smoothing options when plotting.

3.11.1 Smoothing Methods

Some common smoothing techniques in Gnuplot include:

- **csplines**: Cubic spline interpolation.
- **acsplines**: Adjusted cubic spline interpolation.
- **bezier**: Bezier curve smoothing.
- **unique**: Smoothing by eliminating duplicate points.

3.11.2 Example: Cubic Spline Smoothing

For a noisy data file `noisy.txt`, you can plot with cubic spline smoothing:

```
plot "noisy.txt" using 1:2 with lines smooth csplines title "
   Cubic Spline Smoothing"
```

3.11.3 Example: Bezier Smoothing

Alternatively, use Bezier smoothing:

```
plot "noisy.txt" using 1:2 with lines smooth bezier title "Bezier
    Smoothing"
```

Extra: C++ Code to generate noisy data

This C++ program generates a file named `noisy.txt` containing noisy data points based on the equation $y = 2x + 3$ with added random noise. The output file can be used in Gnuplot for plotting and analysis.

Below is the C++ program:


```cpp
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <ctime>

using namespace std;

int main() {
    // Seed the random number generator
    srand(time(0));

    // Open file for writing
    ofstream outFile("noisy.txt");
    if (!outFile) {
        cerr << "Error: Could not create noisy.txt" << endl;
        return 1;
    }

    // Generate noisy data (x, y)
    int numPoints = 50; // Number of data points
    double noiseRange = 2.0; // Maximum noise deviation

    for (int i = 1; i <= numPoints; ++i) {
        double x = i;
        double noise = ((rand() % 1000) / 500.0 - 1.0) *
   noiseRange; // Random noise between -2 and 2
        double y = 2 * x + 3 + noise; // y = 2x + 3 with noise
        outFile << x << " " << y << endl;
    }

    outFile.close();
    cout << "noisy.txt has been created successfully!" << endl;
    return 0;
}
```

Usage

Compile and run the program using:

```
g++ noisy_generator.cpp -o noisy_generator
./noisy_generator
```

This will generate `noisy.txt`, which contains noisy data for further analysis.

3.12 Conclusion

These notes provided an overview of statistical analysis with Gnuplot:

- **Basic statistics:** Using `stats` to compute minimum, maximum, and mean.

- **Regression:** Fitting simple linear models and a basic example of multivariate regression.
- **Data Smoothing:** Applying smoothing techniques such as cubic splines and Bezier curves.

This guide serves as a starting point for exploring Gnuplot's capabilities for both visualization and basic data analysis.

Exercise

Category 1: Easy (Conceptual and Memory-Based)

1. What is Gnuplot, and what are its primary uses?
2. List three types of plots that Gnuplot can generate.
3. How can you set the title of a plot in Gnuplot?
4. Describe the purpose of the `set xlabel` and `set ylabel` commands.
5. What command is used to plot a mathematical function in Gnuplot?
6. What is the default output format for Gnuplot plots?
7. How do you enable grid lines in a Gnuplot plot?
8. What is the purpose of the `set key` command in Gnuplot?
9. How do you exit the Gnuplot command-line interface?
10. What is the purpose of the `replot` command in Gnuplot?

Category 2: Mid-Level (Understanding-Based)

1. Explain how to plot data from a file in Gnuplot. What does the `using` keyword specify?
2. How can you customize the range of the x-axis and y-axis in a plot?
3. Describe the steps to save a plot as a PNG file.
4. What is the difference between the `plot` and `splot` commands?
5. How can you add a legend to your plot, and where can it be positioned?
6. What is the purpose of the `set terminal` command in Gnuplot?
7. How do you plot multiple functions on the same graph in Gnuplot?
8. Explain how to create a histogram in Gnuplot using a data file.
9. What is the purpose of the `set hidden3d` command in 3D plotting?
10. How do you plot data with error bars in Gnuplot?

Category 3: Application-Based

1. Create a Gnuplot script to plot the function $f(x) = e^{-x^2}$ over the range $[-2 : 2]$.
2. Given a data file `experiment.dat` with three columns (time, measurement, error), write a Gnuplot command to plot the measurements with error bars.
3. Write a Gnuplot script to generate a 3D surface plot of $z = \sin(x) \times \cos(y)$.
4. How would you modify the appearance of the plot to use lines instead of points for data visualization?
5. Develop a Gnuplot script to plot multiple datasets from different files on the same graph, each with a distinct style and title.
6. Create a Gnuplot script to plot a polar graph of $r = \sin(2\theta)$.
7. Write a Gnuplot script to generate an animated GIF of a sine wave with a shifting phase.
8. Given a data file `histogram.dat` with two columns (bin, frequency), create a histogram using Gnuplot.
9. Write a Gnuplot script to create a multiplot layout with two plots: one for $\sin(x)$ and another for $\cos(x)$.
10. Develop a Gnuplot script to plot a function with custom line color, line width, and point style.
11. The Bessel functions of the first kind, $J_n(x)$, satisfy the recurrence relations:

$$J_{n-1}(x) + J_{n+1}(x) = (2n+1)\frac{J_n(x)}{x},$$

Given the first two functions:

$$J_0(x) = \frac{\sin(x)}{x},$$

$$J_1(x) = \frac{\sin(x)}{x^2} - \frac{\cos x}{x},$$

use the recurrence relations to compute and plot $J_2(x)$, $J_3(x)$, and $J_4(x)$ for $0 \leq x \leq 10$.

Chapter 4

Introduction to LaTeX

4.1 Introduction to LaTeX

LaTeX is a powerful typesetting system extensively used in academia, especially for scientific documents that involve complex mathematical equations, figures, and references. It allows users to focus on the content while managing the formatting and layout efficiently. Unlike WYSIWYG (what you see is what you get) editors like Microsoft Word, LaTeX operates using plain text markup, which means you define structure and style using commands.

Key features of LaTeX include:

- Precise control over document formatting.
- Easy management of bibliographies, references, and citations.
- Automatic numbering and cross-referencing.
- Superior handling of mathematical formulas.

This document will guide you through the basics of LaTeX and demonstrate how to create well-structured documents with high-quality formatting.

4.2 Getting Started with LaTeX

4.2.1 Installing LaTeX

LaTeX is available on most platforms:

1. **Windows**: Use **MikTeX** or **TeX Live**.
2. **Mac**: Install **MacTeX**.
3. **Linux**: Install via package managers, e.g., `sudo apt-get install texlive-full`.

Popular editors:

- **TeXworks** (included with MikTeX).
- **Overleaf** (online collaborative LaTeX editor).
- **Texmaker** or **VS Code** with LaTeX plugins.

4.2.2 First LaTeX Document

A typical LaTeX document contains a preamble and a body. Below is an example of a basic document:

LaTeX Code:

```
\documentclass{article}
\usepackage[utf8]{inputenc}

\title{My First Document}
\author{John Doe}
\date{\today}

\begin{document}
\maketitle

Hello, this is my first
   document created with \
   LaTeX.
\end{document}

```

Output:

My First Document

John Doe

May 23, 2025

Hello, this is my first document created with LaTeX.

To compile this, run `pdflatex` and a PDF will be generated.

4.3 The Preamble and Body of a LaTeX Document

A LaTeX document consists of two main parts: the **preamble** and the **body**.

4.3.1 The Preamble

The preamble is the part of the document before the `\begin{document}` command. It is used to set up the overall structure and formatting of the document. Key components of the preamble include:

- `\documentclass{...}`: This command defines the type of document you are writing (e.g., `article`, `report`, `book`, etc.). You can also pass options to modify the appearance of the document, such as font size or paper size:

```
    \documentclass[12pt, a4paper]{article}

```

- `\usepackage{...}`: This command imports additional packages to enhance the functionality of your document. For example, to support UTF-8 character encoding or to add mathematical capabilities:

```
    \usepackage[utf8]{inputenc}
    \usepackage{amsmath}

```

- Title information commands:

- \title{...}: Sets the document title.
- \author{...}: Sets the author's name.
- \date{...}: Sets the date. You can use \today to automatically insert the current date.

These settings are later used when the \maketitle command is called in the body of the document.

4.3.2 The Body

The body of the document begins after the \begin{document} command. This is where the actual content of your document is written. You can include sections, text, lists, tables, figures, equations, and other elements. Here is an example of a simple document body:

```
\begin{document}
\maketitle

This is the body of the document. You can add sections like this:
\section{Introduction}
This is an introduction to my document.

You can also include mathematical equations, figures, and tables
   here.
\end{document}
```

The body ends with the \end{document} command, which signals the end of the document.

4.4 Document Structure

4.4.1 Basic Structure

A LaTeX document is organized using sections, subsections, and paragraphs. Here's a quick example:

LaTeX Code:

```
\section{Introduction}
This is the introduction.

\subsection{Background}
This is the background.

\subsubsection{Details}
Further details go here.

\paragraph{Note} This is a
   note.

```

Output:

1. Introduction

This is the introduction.

1.1 Background

This is the background.

1.1.1 Details

Further details go here.

Note This is a note.

4.4.2 Lists

Unordered List:

```
\begin{itemize}
    \item First item
    \item Second item
\end{itemize}
```

Output:

- First item
- Second item

Ordered List:

```
\begin{enumerate}
    \item First item
    \item Second item
\end{enumerate}
```

Output:

1. First item
2. Second item

4.5 Mathematical Typesetting

4.5.1 Inline Math

Inline math is simple to include. For example, the equation of a line can be written as follows:

```
The equation of a line is $y = mx + c$.
```

Output:

The equation of a line is $y = mx + c$.

4.5.2 Displayed Equations

For more complex math that needs its own line, use displayed math:

```

$$ E = mc^2 $$
```

Output:

$$E = mc^2$$

4.5.3 Complex Equations

Integrals can be written as:

```
$$ \int_a^b f(x) dx $$
```

Output:

$$\int_a^b f(x)dx$$

For more advanced math, use the `amsmath` package:

```
\documentclass{article}
\usepackage{amsmath}
% other packages in preamble

\begin{document}

% Your code


\end{document}
```

The `amsmath` package allows for advanced mathematical formatting. After including this package, you can use environments like `align`, `gather`, and more.

Complex Equations Using the `align` Environment

`align`: The `align` environment is used for aligning equations at the equal sign or other relation symbols:

LaTeX Code:

```
\begin{align}
    \int_0^{\infty} e^{-x} \, dx &= 1 \\
    \frac{d}{dx}(x^2) &= 2x \\
    \lim_{x \to 0} \frac{\sin x}{x} &= 1 \\
    e^{i\pi} + 1 &= 0
\end{align}
```

Output:

$$\int_0^{\infty} e^{-x}\,dx = 1 \tag{4.1}$$

$$\frac{d}{dx}(x^2) = 2x \tag{4.2}$$

$$\lim_{x\to 0} \frac{\sin x}{x} = 1 \tag{4.3}$$

$$e^{i\pi} + 1 = 0 \tag{4.4}$$

Complex Equations Using the `gather` Environment

The following equations include integrals, differentiation, and other mathematical symbols:

```
\begin{gather}
    \int_{a}^{b} f(x) \, dx = F(b) - F(a) \\
    \frac{d^2y}{dx^2} + p\frac{dy}{dx} + qy = 0 \\
    \sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6} \\
    \sqrt{a^2 + b^2} = c
\end{gather}
```

Output:

$$\int_a^b f(x)\,dx = F(b) - F(a) \tag{4.5}$$

$$\frac{d^2y}{dx^2} + p\frac{dy}{dx} + qy = 0 \tag{4.6}$$

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6} \tag{4.7}$$

$$\sqrt{a^2 + b^2} = c \tag{4.8}$$

4.6 Figures and Tables

4.6.1 Inserting Figures

To include images in your LaTeX document, you need to use the `graphicx` package, which provides commands for handling graphics and images.

Include the `graphicx` Package

First, ensure you have the following line in the preamble of your document:

```
\usepackage{graphicx}
```

Inserting a Figure

To insert a figure, you use the `figure` environment. Below is the basic syntax:

```
\begin{figure}[h!]
    \centering
    \includegraphics[width=0.75\textwidth]{image.png}
    \caption{Sample Image}
    \label{fig:image1}
\end{figure}
```

- **Figure Environment**: The `figure` environment is a floating container for figures, which allows LaTeX to place the figure at an optimal location in the document. The

optional argument `[h!]` suggests that LaTeX should place the figure "here," but it can be overridden to maintain document flow.

- `\centering`: This command centers the figure within the figure environment.
- `\includegraphics`: This command is used to include the actual image file. The `width` parameter can be specified as a relative value (e.g., `0.75\textwidth` to make the image three quarter the width of the text area) or as an absolute dimension.
- `\caption`: This command provides a caption for the figure that appears below the image, helping to explain or describe it.
- `\label`: This command creates a reference label for the figure, allowing you to refer to it elsewhere in your document using `\ref{fig:image1}`.

Output:

Figure 4.1: UPES

We can refer Figure 4.1 anywhere in the document using it's label.

Important Notes

- Make sure that the image file (e.g., `image.png`) is in the same directory as your `.tex` file for it to be displayed correctly.
- Adjust the `width` parameter as needed to fit your document layout.
- The `figure` environment allows LaTeX to manage the placement of the image, which might not always be exactly where you placed the code, depending on the surrounding content. Use placement options like `[h!]`, `[t]`, `[b]`, or combinations to suggest preferred placement.

4.6.2 Tables

Tables can be created using the `tabular` environment, which provides a flexible way to arrange data in rows and columns. The structure of a table is defined using a combination of alignment specifiers, formatting commands, and optional features.

Basic Structure

The basic structure of a table consists of the following components: - The `table` environment, which allows for the placement of the table in a floating manner. - The `tabular` environment, which defines the actual content of the table.

Here is an example of a simple table:

```
\begin{table}[h!]
    \centering
    \begin{tabular}{|c|c|c|}
        \hline
        A & B & C \\
        \hline
        1 & 2 & 3 \\
        4 & 5 & 6 \\
        \hline
    \end{tabular}
    \caption{Sample Table}
    \label{tab:table1}
\end{table}
```

Output:

A	B	C
1	2	3
4	5	6

Table 4.1: Sample Table

Components of the Table Example

- `table`[h!]: This environment wraps the table and allows it to float in the document. The optional argument `[h!]` suggests placing the table "here" if possible.
- `\centering`: Centers the table on the page.
- `tabular`{|c|c|c|}: This command defines the table structure. The | character adds vertical lines between the columns, and `c` denotes center alignment for each column. You can also use `l` for left alignment and `r` for right alignment.
- `\hline`: Inserts a horizontal line in the table, creating a clear separation between rows.
- `A & B & C` \\: This line specifies the first row of the table. The ampersand `&` separates the columns, and the double backslash \\ indicates the end of the row.
- `\caption{Sample Table}`: Provides a caption for the table that appears above or below the table, depending on the document class and settings.
- `\label{tab:table1}`: This command creates a reference label for the table, allowing you to refer to it elsewhere in the document using `\ref{tab:table1}`.

Customizing Tables

You can customize tables in several ways:

1. **Changing Column Widths**: You can adjust the width of columns using the `pwidth` specifier instead of `c`, `l`, or `r`. For example, `p3cm` sets a fixed width for a column:

```
    \begin{tabular}{|p{3cm}|p{3cm}|p{3cm}|}
    \hline
    Long text in a column & Another column & More text \\
    \hline
    \end{tabular}

```

Output:

Long text in a column	Another column	More text

Table 4.2: My Caption

2. **Merging Cells**: To merge cells horizontally, you can use the `\multirow` or `\multicolumn` commands from the `multirow` package. Example of merging two columns:

```
\usepackage{multirow} % IN PREAMBLE (BEFORE \begin{document})

    \begin{table}[!h]
        \centering
           \begin{tabular}{|c|c|}
            \hline
            \multicolumn{2}{|c|}{Merged Cell} \\
            \hline
            1 & 2 \\
            3 & 4 \\
            \hline
            \end{tabular}
        \caption{Merged cells table}
        \label{tab:my_label_2}
    \end{table}

```

Output:

Merged Cell	
1	2
3	4

Table 4.3: Merged cells table

3. **Adding Borders and Color**: You can enhance the appearance of tables using the `booktabs` package, which provides commands like `\toprule`, `\midrule`, and `\bottomrule` for cleaner horizontal lines:

```latex
\usepackage{booktabs} % IN PREAMBLE (BEFORE \begin{document})

   \begin{table}[!h]
        \centering
        \caption{Tables using booktabs.}
        \begin{tabular}{ccc}
             \toprule
             A & B & C \\
             \midrule
             1 & 2 & 3 \\
             4 & 5 & 6 \\
             \bottomrule
             \end{tabular}
        \label{tab:my_label_3}
   \end{table}


```

Output:

Table 4.4: Tables using booktabs.

A	B	C
1	2	3
4	5	6

Example of a More Complex Table

Here's a more complex example that includes merged cells and custom widths:

```latex
\begin{table}[h!]
\centering
    \begin{tabular}{|p{4cm}|p{4cm}|c|}
    \hline
    \multicolumn{2}{|c|}{Combined Columns} & Single Column \\
    \hline
    Item 1 & Item 2 & Item 3 \\
    \hline
    \end{tabular}
\caption{Complex Table Example}
\label{tab:complex_table}
\end{table}
```

Output:

Combined Columns		Single Column
Item 1	Item 2	Item 3

Table 4.5: Complex Table Example

4.7 Cross-referencing and Bibliography

LaTeX provides powerful tools for cross-referencing and managing bibliographies. These features are particularly useful in larger documents like academic papers, theses, or reports, where you often need to refer to figures, tables, sections, or external references.

4.7.1 Cross-referencing

Cross-referencing in LaTeX allows you to refer to sections, figures, tables, equations, and more, without hardcoding specific numbers. This way, if your document structure changes, all references update automatically.

To set up a cross-reference, you use the `\label` command to mark a specific element, and then use `\ref` or `\pageref` to refer back to that element.

Cross-referencing Sections

For referencing sections, you can place the `\label` command immediately after the section heading. Here's an example:

```
\section{Introduction}\label{sec:intro}

This is the introduction to the paper.

\section{Methodology}\label{sec:method}

As discussed in Section \ref{sec:intro}, the problem is defined
   ...
```

Expected Output
As discussed in Section 1, the problem is defined...

Here, `\ref{sec:intro}` automatically inserts the section number ("1" in this case) into the text. If the order of sections changes, the reference will update to reflect the new numbering.

Cross-referencing Figures and Tables

Cross-referencing is also helpful for figures and tables. Here's an example for referencing a figure:

```
\begin{figure}[h!]
    \centering
    \includegraphics[width=0.5\textwidth]{example-image}
    \caption{An example image.}
    \label{fig:image1}
\end{figure}

As shown in Figure \ref{fig:image1}, the result is clear.
```

Expected Output
As shown in Figure 1, the result is clear.

In this case, the \label command inside the figure environment allows you to reference it using \ref{fig:image1}. The output will insert the figure number ("1" in this case) automatically.

You can similarly cross-reference tables by labeling them within the table environment:

```
\begin{table}[h!]
    \centering
    \begin{tabular}{|c|c|}
        \hline
        Item & Description \\
        \hline
        A & Example A \\
        B & Example B \\
        \hline
    \end{tabular}
    \caption{Example Table.}
    \label{tab:example}
\end{table}

Table \ref{tab:example} shows the details of items.
```

Expected Output
Table 1 shows the details of items.

Cross-referencing Pages

To refer to the page where an element appears, use the \pageref command. This is useful for long documents where you want to direct readers to the exact page of a figure, table, or section:

```
Figure \ref{fig:image1} is found on page \pageref{fig:image1}.
```

Expected Output
Figure 1 is found on page 2.

This command inserts the page number of the referenced element.

4.7.2 Bibliography

LaTeX is widely used in academic writing due to its excellent citation and bibliography management. LaTeX works with tools like BibTeX and BibLaTeX to handle references efficiently. BibTeX allows you to manage and format bibliographic data separately, while BibLaTeX provides more flexibility and modern features.

Basic Bibliography Using BibTeX

To use BibTeX, create a '.bib' file containing your references. In the main LaTeX file, include the following commands to generate the bibliography:

```
\bibliographystyle{plain}
\bibliography{references}
```


Here, `plain` is the style of the bibliography, and `references` is the name of your bibliography file (e.g., `references.bib`).

Your `.bib` file might look like this:

```
@book{lamport1994latex,
  title={LaTeX: A Document Preparation System},
  author={Lamport, Leslie},
  year={1994},
  publisher={Addison-Wesley}
}

@article{knuth1984texbook,
  title={The TeXbook},
  author={Knuth, Donald},
  journal={Computers \& Typesetting},
  volume={A},
  year={1984},
  publisher={Addison-Wesley}
}
```

When compiling your document, BibTeX automatically formats and adds the references at the end of your document. Citations can be added using the `\cite` command:

```
According to \cite{lamport1994latex}, \LaTeX{} is a powerful tool
    for document preparation.
```

Expected Output

According to [1], LaTeX is a powerful tool for document preparation.

At the end of your document, BibTeX generates the bibliography:

References

[1] Lamport, Leslie. LaTeX: A Document Preparation System. Addison-Wesley, 1994.

[2] Knuth, Donald. The TeXbook. Addison-Wesley, 1984.

Bibliography Using BibLaTeX

BibLaTeX is an advanced package for managing citations and bibliographies. To use it, load the package and specify the backend (e.g., biber):

```
\usepackage[backend=biber,style=numeric]{biblatex}
\addbibresource{references.bib}
```

This setup allows for more flexible citation styles, including numeric, alphabetic, author-year, and more. You can then cite sources using the `\cite` command just like in BibTeX:

```
\cite{knuth1984texbook}
```

At the end of the document, print the bibliography using:

```
\printbibliography
```


Citation Styles

Both BibTeX and BibLaTeX offer various citation styles. Common ones include:

- `plain`: Simple numbered style.
- `alpha`: Citations are based on authors' initials and publication year.
- `ieeetr`: IEEE citation style, commonly used in technical and engineering fields.
- `apalike`: APA-style citations, widely used in social sciences.

For example, to use the APA-like citation style:

```
\bibliographystyle{apalike}
```

Or, with BibLaTeX:

```
\usepackage[style=apa]{biblatex}
```

These commands will automatically format your citations and bibliography according to the selected style.

4.8 Customizing LaTeX Documents

Customization in LaTeX allows you to modify the appearance of your document to meet various formatting requirements. In this section, we'll cover some essential aspects of page layout, fonts, and text styles, all of which can be easily adjusted to suit your needs.

4.8.1 Page Layout

The page layout in a LaTeX document, such as paper size, margins, and orientation, can be customized using the `geometry` package. This package provides flexibility in adjusting the dimensions of the page to fit specific formatting needs.

To change the paper size and margins, you can specify options directly when loading the `geometry` package. Here's an example for setting A4 paper size and 1-inch margins:

```
\usepackage[a4paper, margin=1in]{geometry}
```

Customizing Margins

You can also specify custom margins for different sides of the page. For example, to set a 2-inch top margin, 1-inch bottom margin, 1.5-inch left margin, and 1-inch right margin, use:

```
\usepackage[top=2in, bottom=1in, left=1.5in, right=1in]{geometry}
```


Changing Paper Size and Orientation

To change the paper size to legal (8.5 x 14 inches) and make it landscape oriented, you can modify the options as follows:

```
\usepackage[legalpaper, landscape, margin=1in]{geometry}
```

Output: The page will be oriented horizontally (landscape) on legal-sized paper with 1-inch margins.

These changes will apply globally across the entire document unless you specify otherwise.

4.8.2 Font and Style

Font customization in LaTeX is managed by various packages, such as `fontenc` for encoding and `inputenc` for character sets. Changing fonts and text styles can improve readability and give your document a personalized look.

Font Encoding

Using the `fontenc` package ensures that fonts are properly encoded. For example, to enable T1 encoding, which allows for proper hyphenation and accented characters, use:

```
\usepackage[T1]{fontenc}
```

T1 encoding is essential when working with European languages or documents requiring accented characters.

Changing Fonts

You can change the font family to one of LaTeX's default font families, such as `serif`, `sans-serif`, or `monospace`, using the following commands:

```
\renewcommand{\familydefault}{\sfdefault}  % Sans-serif as
    default
```

To use a specific font, such as the popular `Times New Roman`, you can load the corresponding package:

```
\usepackage{times}  % Times New Roman
```

Output:
The entire document's font will switch to Times New Roman.
Other common font packages include:

1. `helvet` for Helvetica (sans-serif)
2. `courier` for Courier (monospace)

Text Styles

In LaTeX, text styles such as bold, italic, and underlined text are easily applied using the following commands:

1. **Bold Text:** Use `\textbf{...}` to make text bold.


```
This is \textbf{bold text}.
```

Output:

This is **bold text**.

2. **Italic Text:** Use `\textit{...}` to italicize text.

```
  This is \textit{italic text}.

```

Output:

This is *italic text*.

3. **Underlined Text:** While LaTeX doesn't have a direct underline command, you can use the `ulem` package to underline text:

```
  \usepackage{ulem}
  This is \uline{underlined text}.

```

Output:

This is underlined text.

Alternatively, for a simpler underlining solution, you can use the `underline` command from standard LaTeX:

```
  This is \underline{underlined text}.

```

Customizing Font Sizes

Font size can be adjusted globally or locally within the document. To set the font size for the entire document, modify the document class as follows:

```
\documentclass[12pt]{article}
```

This will set the default font size to 12 points.
For local font size adjustments, use the following commands within the document:

- `\tiny`: Very small text
- `\scriptsize`: Smaller than small
- `\footnotesize`: Slightly larger than `scriptsize`
- `\small`: Small text
- `\large`: Slightly larger text
- `\Large`, `\LARGE`: Progressively larger text
- `\huge`, `\Huge`: Very large text

Example:

```
This is \tiny{tiny text}, and this is \Huge{huge text}.
```

Output:

This is tiny text, and this is huge text.

By combining these commands, you can customize the look and feel of your document, ensuring it matches specific formatting guidelines or personal preferences.

4.8.3 Color and Highlighting

In LaTeX, the `xcolor` package is commonly used to apply colors to text and other elements. You can highlight important sections, change font colors, and even define your own custom colors.

To load the `xcolor` package:

```
\usepackage{xcolor}
```

Changing Text Color

To change the color of specific text, use the `\textcolor` command. Here's an example that sets the text color to red:

```
This is \textcolor{red}{red text}.
```

Output:

This is red text.

Highlighting Text

You can also highlight text with a background color using the `\colorbox` command:

```
\colorbox{yellow}{This text is highlighted in yellow.}
```

Output:

This text is highlighted in yellow.

Customizing LaTeX documents provides a great deal of flexibility in terms of page layout, fonts, text styles, and colors. Using the `geometry` package, you can control the page dimensions and margins. Text appearance can be easily managed with font encodings, font family selection, and local style adjustments like bold, italics, and color. Together, these tools allow you to craft a professional and visually appealing document.

4.9 Error Handling and Debugging

When working with LaTeX, errors can arise during the compilation process. Understanding common error messages and how to resolve them is essential for smooth document preparation. LaTeX editors, such as Overleaf, TeXShop, or TeXworks, provide detailed logs that can help you trace and fix errors.

In this section, we'll look at common errors, their causes, and strategies for debugging.

4.9.1 Common LaTeX Errors

Here are some of the most common errors you might encounter when compiling a LaTeX document:

Missing or Mismatched Braces

One of the most frequent errors is missing or unmatched braces (i.e., {...}). Every opening brace { must have a corresponding closing brace }.

Example Error:

```
This is an \textbf{example of missing brace.
```

The error message may look like this:

```
! LaTeX Error: \textbf on input line 1 ended by \end{document}.
```

Solution: Ensure that every { has a matching }. The correct syntax is:

```
This is an \textbf{example of correct brace}.
```

If you're dealing with nested braces, carefully check that each pair is properly closed.

Undefined References

Undefined references occur when you try to reference a section, figure, table, or citation that has not been labeled correctly or is missing entirely. You will see a warning like this during compilation:

```
LaTeX Warning: There were undefined references.
```

Example Error:

```
As shown in Figure \ref{fig:missing}, the results are clear.
```

If no figure with the label `fig:missing` exists, you'll get an error.

Solution: Ensure that you have labeled the element you're referencing. For example:

```
\begin{figure}[h!]
    \includegraphics[width=0.5\textwidth]{example-image}
    \caption{An example image.}
    \label{fig:image1}
\end{figure}

As shown in Figure \ref{fig:image1}, the results are clear.
```

Also, make sure to run multiple compilation steps (e.g., in Overleaf, press "Recompile" twice) to resolve cross-references.

Package Errors

Using incorrect or incompatible packages can lead to compilation errors. This happens if a required package is missing from your LaTeX installation or if two packages conflict with each other.

Example Error: If you try to load a non-existent package:

```
\usepackage{nonexistent}
```

You will see an error message like this:

```
! LaTeX Error: File 'nonexistent.sty' not found.
```

Solution: Ensure that the package you're trying to use is installed or available in your LaTeX distribution. For example, replace it with a valid package:

```
\usepackage{graphicx}  % A valid package
```

For package conflicts, try commenting out one of the conflicting packages or look for a compatible alternative.

4.9.2 Debugging Tips

Here are some strategies to debug your LaTeX documents effectively:

Read the Log File

Most LaTeX editors provide a detailed log file that lists all warnings and errors encountered during compilation. This log can help pinpoint the exact line where the error occurred. The log will often include the following types of messages:

- **Error messages**: These are critical and stop the compilation.
- **Warnings**: These indicate potential issues but do not stop the compilation.
- **Overfull/Underfull boxes**: These warn about text that overflows or does not properly fit in the margins.

To view the log, look for the "Log" or "Compiler" section in your editor. In Overleaf, for instance, the log is displayed in a separate window after compilation.

Isolate the Problem

If you're facing a complex issue and cannot locate the source of the error, try commenting out large sections of your document. You can use the % symbol to comment out lines of text or code temporarily:

```
%\section{Introduction}
%This section is commented out to isolate the problem.
```

Once you've isolated the error, you can start uncommenting sections one by one to find the problematic code.

Check for Typos in Labels

Typographical errors in labels are a common cause of undefined references. Double-check that your labels are spelled correctly and match the references exactly. LaTeX is case-sensitive, so {`fig:image1`} and {`fig:Image1`} will be treated as different labels.

Run Multiple Compilation Passes

When using cross-references or bibliographies (especially with BibTeX or BibLaTeX), LaTeX often requires multiple compilation passes to resolve all references. You may need to run:

1. `pdflatex`
2. `bibtex`
3. `pdflatex` (twice more)

This ensures that all citations and references are updated correctly.

4.9.3 Warnings

While warnings do not stop compilation, they can indicate formatting problems or overlooked issues. Some common warnings include:

Overfull or Underfull Boxes

These occur when text exceeds the margins (overfull) or does not fill the available space properly (underfull). The message may look like this:

```
Overfull \hbox (5.0pt too wide) in paragraph at lines 22--23
```

Solution: Adjust the text, font size, or use the `\sloppy` command to relax the formatting rules.

Undefined Citations

If a citation is not defined in the bibliography, you'll see a warning like:

```
LaTeX Warning: Citation 'key' on page 3 undefined.
```

Solution: Ensure that the citation key matches the reference in your `.bib` file.

4.9.4 Tools for Error-Free LaTeX

Here are some tools and techniques that can help you avoid and fix errors in LaTeX documents:

Online Editors

Using online editors like Overleaf can make error handling easier since they offer real-time error messages and logs. Overleaf, for example, highlights errors and warnings as you type, making it easier to identify issues immediately.

`lacheck` and `chktex`

These are command-line tools designed to check LaTeX documents for common errors and potential formatting issues. `lacheck` checks the syntax of your document, while `chktex` focuses on typographical issues.

Error handling and debugging are crucial aspects of working with LaTeX. By understanding common errors, reading logs carefully, and using debugging strategies, you can efficiently resolve issues and ensure smooth compilation. With the right tools and techniques, error-free LaTeX documents are easy to achieve.

4.10 Title Page and Its Customization in LaTeX

The title page is the first page of a LaTeX document, serving as the cover for your work. It typically includes the title of the document, the author's name, the institution, the date, and sometimes additional information like the course name or the supervisor's name. Customizing the title page can help create a professional and polished look for your document.

4.10.1 Basic Title Page

To create a basic title page in LaTeX, you can use the `\title`, `\author`, and `\date` commands, followed by the `\maketitle` command. Here's a simple example:

```
\documentclass{article}

\title{The Title of Your Document}
\author{Your Name}
\date{\today}  % Automatically inserts today's date

\begin{document}

\maketitle  % Generates the title page

\end{document}
```

4.10.2 Customizing the Title Page

- **Changing Fonts and Sizes:** You can customize the font size and style of the title, author, and date by using font commands. For example:

```
\title{\huge \textbf{The Title of Your Document}}
\author{\Large Your Name}

```


- **Adding a Logo:** If you want to include a logo (e.g., your institution's logo), you can use the `graphicx` package:

```
\usepackage{graphicx}

\title{\includegraphics[width=0.5\textwidth]{logo.png}\\[1em]
    \Huge \textbf{The Title of Your Document}}

```

- **Customizing Layout:** To further customize the layout of the title page, you can create your own title page using the `titlepage` environment. This allows more flexibility in positioning elements:

```
\begin{titlepage}
    \centering
    \vspace*{2cm}  % Adds vertical space
    {\Huge \textbf{The Title of Your Document}}\\[1.5cm]
    {\Large Your Name}\\
    {\large Institution Name}\\
    {\large \today}\\[2cm]
    \includegraphics[width=0.3\textwidth]{logo.png}\\[1cm]
    {\large Course Name}\\
    {\large Supervisor Name}
    \vfill
\end{titlepage}

```

- **Using Packages:** You can also explore packages like `titling` or `fancyhdr` for more advanced customization of the title page and headers/footers.

4.10.3 Example of a Customized Title Page

Here's a complete example with a customized title page:

```
\documentclass{article}
\usepackage{graphicx}

\title{\huge \textbf{The Title of Your Document}}
\author{\Large Your Name}
\date{\today}

\begin{document}

\begin{titlepage}
    \centering
    \vspace*{2cm}
    \includegraphics[width=0.3\textwidth]{logo.png}\\[1.5cm]
    {\Huge \textbf{The Title of Your Document}}\\[1.5cm]
    {\Large Your Name}\\[0.5cm]
    {\large Institution Name}\\[1.5cm]
    {\large \today}\\[2cm]
    {\large Course Name}\\
```

```
    {\large Supervisor Name}
    \vfill
\end{titlepage}

\end{document}
```

Customizing the title page in LaTeX is straightforward, allowing you to create a visually appealing introduction to your document. With simple commands and environments, you can adjust the layout, include graphics, and modify text styles to match your preferences or institutional requirements.

Further Resources

- Overleaf LaTeX Documentation: https://www.overleaf.com/learn/latex/Main_Page
- CTAN (Comprehensive TeX Archive Network): https://ctan.org/

Exercise

Category 1: Easy (Conceptual and Memory-Based)

1. What is LaTeX, and how does it differ from WYSIWYG editors like Microsoft Word?
2. List three key features of LaTeX that make it popular for academic document preparation.
3. What is the purpose of the preamble in a LaTeX document?
4. Describe the function of the following commands in LaTeX: `\documentclass`, `\usepackage`, and `\maketitle`.
5. How can you include mathematical equations in a LaTeX document? Provide an example of an inline equation.
6. What are the differences between ordered and unordered lists in LaTeX? Write a simple example for each.
7. Why is the `graphicx` package used in LaTeX documents?
8. What are the basic components of a title page in LaTeX?
9. Explain the difference between `\section`, `\subsection`, and `\paragraph` in structuring a LaTeX document.
10. What is the advantage of using `\label` and `\ref` commands for cross-referencing in LaTeX?

Category 2: Mid-Level (Understanding-Based)

1. Write a simple LaTeX document that includes a title page with a title, author, date, and a centered image.
2. Describe how the `align` environment is used for complex equations in LaTeX. Provide an example.
3. How can you customize the margins of a LaTeX document? Write the command to set all margins to 1 inch.
4. Create a simple table using the `tabular` environment with three columns: Name, Age, and Country.
5. Explain the difference between `\textbf`, `\textit`, and `\underline`. Write an example showing their usage.
6. How can you handle errors like "undefined references" in a LaTeX document? Suggest debugging strategies.
7. Write a LaTeX code snippet to display the equation $E = mc^2$ as a standalone equation.

8. How can you include a bibliography in a LaTeX document? Provide the basic steps.
9. Explain how the `xcolor` package is used to change the color of text in LaTeX. Write an example to display text in red.
10. How can you dynamically create numbered references for figures and tables in LaTeX? Write a small example showing a labeled figure.

Category 3: Application-Based

1. Write a LaTeX document to include a centered image, a table, and a displayed equation.
2. Create a LaTeX document for a two-column article layout with separate sections for Introduction and Conclusion.
3. Write a LaTeX document with cross-references to figures, tables, and sections.
4. Insert and reference a figure in a LaTeX document. Provide the corresponding LaTeX code.
5. Write a LaTeX code snippet for creating a custom title page that includes a title, author name, date, course, and institution name.
6. Write a LaTeX document to demonstrate the use of inline, displayed, and complex equations (using the `amsmath` package).
7. Create a LaTeX document to manage multiple sections, each containing a figure, table, and equation.
8. Write a LaTeX document to generate a bibliography using BibTeX with at least two references.
9. Demonstrate how to use the `booktabs` package for creating professional-looking tables in LaTeX.
10. Write a LaTeX document with a custom page layout, including specific margin settings and page orientation.

Chapter 5

Finding Roots of an Equation

In this chapter, we will explore three fundamental numerical methods for finding roots of equations: the Bisection Method, the Secant Method, and Newton-Raphson Method. Each method will be introduced with theoretical concepts, illustrated with examples, and followed by practice questions to strengthen your understanding.

Introduction to Root-Finding Methods

Root-finding algorithms are essential in numerical analysis for solving equations of the form $f(x) = 0$. We will discuss three methods here:

- **Bisection Method** - a simple and reliable method.
- **Secant Method** - a faster approach that avoids calculating derivatives.
- **Newton-Raphson Method** - a powerful method using derivatives for rapid convergence.

5.1 Bisection Method

The Bisection Method is a numerical approach to find a root of a continuous function $f(x)$ within a specified interval. It is particularly useful when the function changes sign over an interval, indicating the presence of a root.

5.1.1 Method Explanation

The Bisection Method works as follows:

1. Choose an interval $[a, b]$ such that $f(a) \cdot f(b) < 0$. This guarantees that there is at least one root in $[a, b]$.
2. Calculate the midpoint $m = \frac{a+b}{2}$.
3. Evaluate $f(m)$. If $f(m) = 0$, then m is the root. Otherwise, update the interval as follows:
    - If $f(a) \cdot f(m) < 0$, set $b = m$.

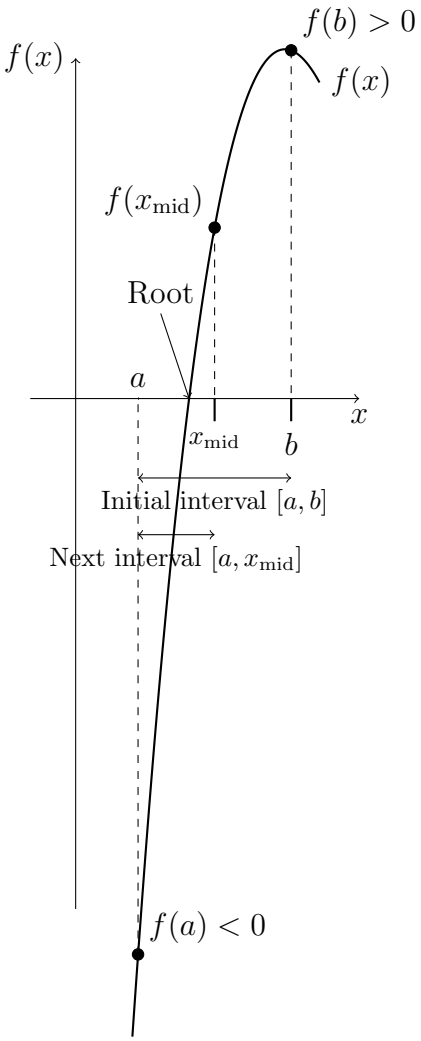


Figure 5.1: Illustration of the Bisection method using the function $f(x) = x^3 - 9x^2 + 23x - 15$. The method starts with the interval $[a, b]$, then iteratively bisects the interval to find the root by checking sign changes in $f(x)$.

- If $f(b) \cdot f(m) < 0$, set $a = m$.

4. Repeat the steps until the interval $[a, b]$ is sufficiently small, or until the midpoint m is accurate to the desired precision.

5.1.2 Example: Finding the Root of f(x) = sinx - x cosx

Let's find the root of the function $f(x) = \sin x - x \cos x$ in the interval $[4, 5]$ using the Bisection Method. We'll proceed step-by-step, calculating each midpoint and evaluating the function to see if we've narrowed down the root.

Initial Setup

$$f(x) = \sin x - x \cos x$$

Evaluating $f(x)$ at the endpoints:

$$f(4) = \sin(4) - 4\cos(4) \approx 1.8577719881465196$$

$$f(5) = \sin(5) - 5\cos(5) \approx -2.3772352019792695$$

Since $f(4) \cdot f(5) < 0$, there is a root between $x = 4$ and $x = 5$.

Iterative Steps

The following table shows the iterative steps for the Bisection Method applied to $f(x) = \sin x - x\cos x$ in the interval $[4, 5]$:

Iteration	a	b	$m = \frac{a+b}{2}$	$f(a)$	$f(b)$	$f(a) \cdot f(b)$	Interval Update
1	4	5	4.5	1.85777	-2.37724	-4.42 (< 0)	[4,5]
2	4	4.5	4.25	1.85777	-0.02895	-0.05 (< 0)	[4,4.5]
3	4.25	4.5	4.375	1.00088	-0.02895	-0.03 (< 0)	[4.25,4.5]
4	4.375	4.5	4.4375	0.50461	-0.02895	-0.01 (< 0)	[4.375,4.5]
5	4.4375	4.5	4.46875	0.24206	-0.02895	-0.01 (< 0)	[4.4375,4.5]
6	4.46875	4.5	4.484375	0.10756	-0.02895	-0.00 (< 0)	[4.46875,4.5]
7	4.484375	4.5	4.4921875	0.03955	-0.02895	-0.00 (< 0)	[4.484375,4.5]
8	4.4921875	4.5	4.49609375	0.00536	-0.02895	-0.00 (< 0)	[4.4921875,4.5]
9	4.4921875	4.49609375	4.494140625	0.00536	-0.01178	-0.00 (< 0)	[4.4921875,4.49609375]
10	4.4921875	4.494140625	4.4931640625	0.00536	-0.00321	-0.00 (< 0)	[4.4921875,4.494140625]
11	4.4931640625	4.494140625	4.49365234375	0.00108	-0.00321	-0.00 (< 0)	[4.4931640625,4.494140625]

Table 5.1: Bisection Method Iterations for $f(x) = \sin x - x\cos x$ in the interval $[4, 5]$

Final Answer

After continuing the iterations, we find that the root of $f(x) = \sin x - x\cos x$ to the desired precision in the interval $[4, 5]$ is approximately:

$$x \approx 4.49365234375$$

5.1.3 Error Estimation in the Bisection Method

In the Bisection Method, we iteratively narrow down the interval $[a, b]$ that contains the root. With each iteration, the interval's length is halved, allowing us to estimate the error and the convergence rate.

Absolute Error Bound

If we define the root as r, then after n iterations, the interval $[a_n, b_n]$ contains r. The error in the approximation $m_n = \frac{a_n+b_n}{2}$, which is the midpoint of the interval, is bounded by half the interval length:

$$|m_n - r| \leq \frac{b_n - a_n}{2} = \frac{b-a}{2^n}$$

where $[a, b]$ is the initial interval.

As n increases, the interval $[a_n, b_n]$ becomes smaller, leading to a smaller error bound. This error bound tells us how close our approximation m_n is to the actual root r.

Number of Iterations for Desired Accuracy

To achieve a specific accuracy ϵ, we can calculate the required number of iterations N as follows:

$$N \geq \log_2 \left(\frac{b-a}{\epsilon} \right)$$

This formula allows us to determine the minimum number of iterations needed to ensure that our approximation is within a specified tolerance ϵ from the true root.

Convergence Rate

The Bisection Method has a convergence rate of $\mathcal{O}(2^{-n})$, which means the error decreases by approximately half with each iteration. This linear convergence is slower compared to other methods like the Newton-Raphson Method, which has quadratic convergence, but the Bisection Method is more robust and guarantees convergence as long as the initial interval contains a root.

Example of Error Estimation

Suppose we start with an interval $[4, 5]$ and want to find the root of $f(x) = \sin x - x \cos x$ to within $\epsilon = 0.001$. Using the formula above, we can estimate the number of iterations needed:

$$N \geq \log_2 \left(\frac{5-4}{0.001} \right) = \log_2(1000) \approx 10$$

Therefore, at least 10 iterations are required to ensure that the error in our approximation is less than 0.001.

This error estimation helps us plan the number of iterations in advance and gives confidence that our final approximation is close to the true root within the desired accuracy.

5.1.4 Disadvantages of the Bisection Method

While the bisection method is reliable and simple, it has several limitations:

- **Slow Convergence:** The bisection method converges linearly, which makes it slower compared to other methods such as Newton-Raphson that converge quadratically. This can be a disadvantage when higher accuracy is needed in fewer iterations.
- **Requires an Interval with Opposite Signs:** The method requires the initial interval $[a, b]$ to satisfy $f(a)f(b) < 0$, meaning that a sign change between $f(a)$ and $f(b)$ is essential. If no such interval is known, the method cannot be applied.
- **Not Suitable for Multiple or Complex Roots:** The Bisection method can only find one real root within an interval, and it does not work for complex roots or multiple roots within the same interval unless the function is redefined or additional methods are employed.
- **Cannot Handle Discontinuous Functions:** The method assumes the function is continuous over the interval $[a, b]$. If the function has discontinuities, the Bisection method might fail or produce incorrect results.

C++ Code:

```cpp
// C++ program to find the root of a polynomial using Bisection method
#include <iostream>
#include <cmath>
#include <fstream>
using namespace std;

// Define the polynomial function f(x) = x^3 - x - 2
double f(double x) {
    return x*x*x - x - 2;
}

// Bisection method to find root
double BisectionMethod(double a, double b, double tol, ofstream &outfile) {
    double mid;
    int iterations = 0;
    outfile << "# Iteration\tBisection_Root" << endl;
    while ((b - a) >= tol) {
        mid = (a + b) / 2.0;
        outfile << iterations << "\t" << mid << endl;
        if (f(mid) == 0.0)  // Exact root found
            break;
        else if (f(mid) * f(a) < 0)
            b = mid;
        else
            a = mid;
        iterations++;
    }
    return mid;
}

int main() {
    double a, b, x0, tol;

    // Open file to store the output
    ofstream outfile("roots_output.txt");

    // Input interval for Bisection
    cout << "Enter the interval [a, b] for Bisection method: ";
    cin >> a >> b;

    // Input initial guess for Newton-Raphson
    cout << "Enter the initial guess for Newton-Raphson method: ";
    cin >> x0;

    // Input tolerance level
    cout << "Enter the tolerance level: ";
```

```
47      cin >> tol;
48
49      // Finding root using Bisection Method
50      double bisection_root = BisectionMethod(a, b, tol, outfile);
51
52      return 0;
53      }
```

Fortran Code:

```
1  PROGRAM BisectionMethod
2    IMPLICIT NONE
3    REAL :: a, b, tol, c, fa, fb, fc
4    INTEGER :: max_iter, iter
5
6    ! Input values
7    PRINT *, 'Enter the lower bound (a):'
8    READ *, a
9    PRINT *, 'Enter the upper bound (b):'
10   READ *, b
11   PRINT *, 'Enter the tolerance (tol):'
12   READ *, tol
13   PRINT *, 'Enter the maximum number of iterations:'
14   READ *, max_iter
15
16   ! Function values at a and b
17   fa = f(a)
18   fb = f(b)
19
20   ! Check if initial bounds are valid
21   IF (fa * fb > 0) THEN
22     PRINT *, 'Error: The function must have different signs at a
     and b.'
23     STOP
24   END IF
25
26   ! Bisection loop
27   iter = 0
28   DO WHILE (ABS(b - a) > tol .AND. iter < max_iter)
29     c = (a + b) / 2.0
30     fc = f(c)
31
32     IF (ABS(fc) < tol) THEN
33       EXIT
34     ELSE IF (fa * fc < 0) THEN
35       b = c
36       fb = fc
37     ELSE
38       a = c
39       fa = fc
```



```
    END IF

    iter = iter + 1
  END DO

  ! Output results
  IF (iter >= max_iter) THEN
    PRINT *, 'Maximum iterations reached without convergence.'
  ELSE
    PRINT *, 'Root found at c = ', c, ' after ', iter, '
   iterations.'
  END IF

CONTAINS

  ! Define the function f(x) = x^3 - x^2 - 1 (example function)
  REAL FUNCTION f(x)
    REAL, INTENT(IN) :: x
    f = x**3 - x**2 - 1.0
  END FUNCTION

END PROGRAM BisectionMethod
```

5.1.5 Practice Questions

1. Use the Bisection Method to find the root of $f(x) = x^2 - 4$ on the interval $[0, 3]$ to three decimal places.

2. Apply the Bisection Method to find the root of $f(x) = \cos x - x$ on $[0, 1]$.

5.2 Secant Method

The Secant method is a numerical technique used to find the root of a function $f(x)$ by using a secant line to approximate the function near the root. Unlike the Bisection method, the two initial points for the Secant method do not need to lie on opposite sides of the root, but they must be sufficiently close to it. However, choosing points on opposite sides of the root often improves the stability of the method.

The Secant method uses two initial points, x_1 and x_2, and approximates the function by a straight line passing through these two points. The root is then estimated as the x-intercept of this secant line. The equation of the secant line passing through the points $(x_1, f(x_1))$ and $(x_2, f(x_2))$ is given by:

$$y - f(x_2) = \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x - x_2)$$

Setting $y = 0$ to find the x-intercept (the approximation of the root), we get:

$$0 - f(x_2) = \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x_3 - x_2)$$

Solving for x_3, the next approximation of the root is:

$$x_3 = x_2 - f(x_2)\frac{x_2 - x_1}{f(x_2) - f(x_1)}$$

This formula is iterated with the newly found point x_3 replacing x_1, and x_2 replacing x_3 in subsequent steps. The process is repeated until the values of x_n converge to a root with the desired level of accuracy.

5.2.1 Method Explanation

Given two points x_0 and x_1 close to the root, the secant method approximates the root using:

$$x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$

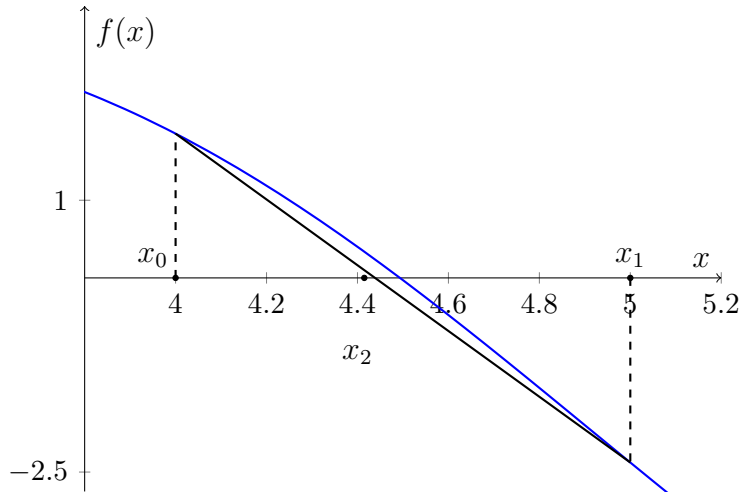


Figure 5.2: Secant method on f(x) = sin(x) - xcos(x).

5.2.2 Example: Solving $f(x) = \sin x - xcosx = 0$ for $x \in [4, 5]$

Let's apply the Secant Method to find the root of $f(x) = \sin x - x\cos x$ with x in radians and initial guesses $x_0 = 4.0$ and $x_1 = 5.0$. We will continue the iterations until the function value is close to zero, recording the process in a table.

$$f(x) = \sin x - x\cos x$$

Detailed Iterations in Table

Table 5.2: Solving $f(x) = \sin x - x\cos x$ using Secant method.

Iteration	x_n	x_{n-1}	$f(x_n)$	$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$
0	4.0	-	$f(4.0) = \sin(4.0) - 4.0\cos(4.0) \approx -2.613$	-
1	5.0	4.0	$f(5.0) = \sin(5.0) - 5.0\cos(5.0) \approx 3.418$	$x_2 = 4.0 - (-2.613)\frac{4.0-5.0}{-2.613-3.418} \approx 4.433$
2	4.433	5.0	$f(4.433) \approx -0.432$	$x_3 = 4.433 - (-0.432)\frac{4.433-5.0}{-0.432-3.418} \approx 4.490$
3	4.490	4.433	$f(4.490) \approx -0.030$	$x_4 = 4.490 - (-0.030)\frac{4.490-4.433}{-0.030+0.432} \approx 4.494$
4	4.494	4.490	$f(4.494) \approx 0.0005$	$x_5 = 4.494 - 0.0005\frac{4.494-4.490}{0.0005+0.030} \approx 4.4934$
5	4.4934	4.494	$f(4.4934) \approx 0$	Converged to root

Explanation of Iterations

In this table:
- **Iteration 0:** We start with initial guesses $x_0 = 4.0$ and $x_1 = 5.0$, calculating $f(x_0) \approx -2.613$ and $f(x_1) \approx 3.418$.
- **Iteration 1:** Using the Secant formula, we find $x_2 \approx 4.433$.
- **Iteration 2 to 4:** We continue the iterations, refining our approximations.
- **Iteration 5:** We reach $x \approx 4.4934$, where $f(x) \approx 0$, indicating the approximate root is $x \approx 4.4934$.

The Secant Method has successfully approximated the root of $f(x) = \sin x - x\cos x$ in the interval $[4, 5]$ to be around $x \approx 4.4934$. This iterative approach converges quickly and avoids the need for derivatives, making it a practical alternative to other root-finding methods.

5.2.3 Practice Questions

1. Find the root of $f(x) = x^2 - 2x + 1$ using the Secant Method with initial guesses $x_0 = 1.5$ and $x_1 = 2$.

2. Use the Secant Method to approximate the root of $f(x) = \sin x - 0.5$ with initial guesses $x_0 = 0.5$ and $x_1 = 1$.

5.3 Newton-Raphson Method

The Newton-Raphson method is an iterative numerical technique for approximating the roots of a real-valued function $f(x)$. The derivation is based on the Taylor series expansion of $f(x)$ around a given point.

5.3.1 Taylor Series Expansion using h

Let a be an initial approximation of the root of $f(x)$, such that $f(a)$ is close to zero. Define a small correction term h such that the exact root is given by:

$$r = a + h.$$

Expanding $f(a + h)$ in a Taylor series around a, we get:

$$f(a + h) = f(a) + f'(a)h + \frac{f''(a)}{2!}h^2 + \frac{f'''(a)}{3!}h^3 + \dots$$

Since r is a root of $f(x)$, we have $f(r) = f(a+h) = 0$. Therefore, setting $f(a+h) = 0$, we obtain:

$$0 = f(a) + f'(a)h + \frac{f''(a)}{2!}h^2 + \frac{f'''(a)}{3!}h^3 + \dots$$

5.3.2 First-Order Approximation

If h is small, the higher-order terms $h^2, h^3, \dots$ become negligible. Keeping only the first-order terms, we approximate the equation as:

$$0 \approx f(a) + f'(a)h.$$

Solving for h:

$$h \approx -\frac{f(a)}{f'(a)}.$$

Since h is the correction term to refine our approximation, the next approximation of the root is:

$$x_1 = a + h = a - \frac{f(a)}{f'(a)}.$$

Generalizing this to an iterative formula, we define:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

This is the Newton-Raphson iterative formula.

5.3.3 Convergence of the Method

The Newton-Raphson method exhibits quadratic convergence under suitable conditions. If x_0 is sufficiently close to the root r and $f'(r) \neq 0$, the error in each iteration approximately satisfies:

$$|x_{n+1} - r| \approx C|x_n - r|^2,$$

where C is a constant. This rapid error reduction makes the Newton-Raphson method highly efficient.

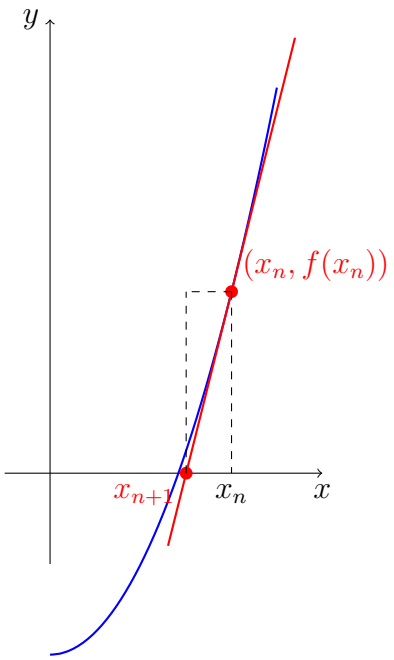


Figure 5.3: In the diagram, the blue curve represents $f(x) = x^2 - 2$, the red dot at $(2, 2)$ is the current approximation $(x_n, f(x_n))$, and the red dashed tangent line (with equation $y = 4x - 6$) crosses the x-axis at $x = 1.5$, which is the next approximation x_{n+1}.

5.3.4 Geometric Interpretation

The method can be understood geometrically. The idea is to approximate $f(x)$ by its tangent at x_n and use the x-intercept of this tangent as the next approximation x_{n+1}.

For example, consider the function

$$f(x) = x^2 - 2,$$

with $x_n = 2$. Then:

$$f(2) = 2 \quad \text{and} \quad f'(2) = 4.$$

The tangent line at $(2, 2)$ is given by:

$$y - 2 = 4(x - 2) \quad \Longrightarrow \quad y = 4x - 6.$$

Its x-intercept is found by setting $y = 0$:

$$4x - 6 = 0 \quad \Longrightarrow \quad x = 1.5,$$

so $x_{n+1} = 1.5$.

5.3.5 A new function

Consider the function

$$f(x) = 9x - 6x^2 + x^3.$$

Let the initial approximation be $x_0 = 1.3$. Then, we compute

$$f(1.3) = 9(1.3) - 6(1.3)^2 + (1.3)^3 \approx 11.7 - 10.14 + 2.197 \approx 3.76,$$

and the derivative is

$$f'(x) = 9 - 12x + 3x^2, \quad f'(1.3) \approx 9 - 15.6 + 5.07 \approx -1.53.$$

The tangent line at the point (1.3, 3.76) is given by

$$y - 3.76 = -1.53(x - 1.3).$$

Setting $y = 0$ to find its x-intercept:

$$-3.76 = -1.53(x - 1.3) \quad \Longrightarrow \quad x - 1.3 \approx \frac{3.76}{1.53} \approx 2.46,$$

so that the next approximation is

$$x_1 \approx 1.3 + 2.46 \approx 3.76.$$

Now, starting with $x_1 \approx 3.76$, we compute

$$f(3.76) = 9(3.76) - 6(3.76)^2 + (3.76)^3 \approx 33.84 - 84.83 + 53.19 \approx 2.20,$$

and

$$f'(3.76) = 9 - 12(3.76) + 3(3.76)^2 \approx 9 - 45.12 + 42.41 \approx 6.29.$$

The tangent line at (3.76, 2.20) is

$$y - 2.20 = 6.29(x - 3.76).$$

Setting $y = 0$ gives

$$-2.20 = 6.29(x - 3.76) \quad \Longrightarrow \quad x - 3.76 \approx \frac{-2.20}{6.29} \approx -0.35,$$

so that

$$x_2 \approx 3.76 - 0.35 \approx 3.41.$$

5.3.6 Detailed Iterations in Table

We solve

$$f(x) = 9x - 6x^2 + x^3,$$

using the Newton–Raphson method with the iteration formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

For this function, note that it can be factored as

$$f(x) = x(x - 3)^2,$$

and its derivative is

$$f'(x) = 9 - 12x + 3x^2.$$

Starting with an initial approximation $x_0 = 1.3$, the following table summarizes the iterations:

The table shows the convergence of the method as the approximations approach the repeated root at $x = 3$.

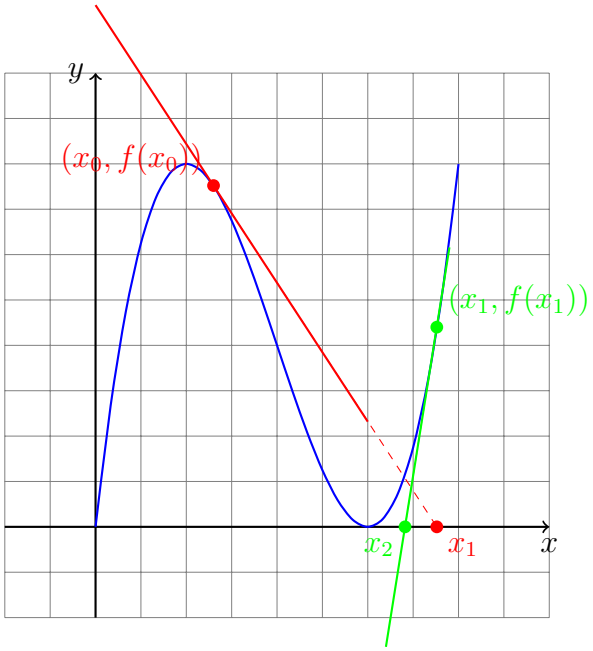


Figure 5.4: The figure illustrates the first two iterations of the Newton-Raphson method on $f(x) = 9x - 6x^2 + x^3$. A grid is added to the x and y axes. The red elements correspond to the first iteration starting at $x_0 = 1.3$, and the green elements correspond to the second iteration starting at $x_1 \approx 3.76$.

Table 5.3: Solving $f(x) = 9x - 6x^2 + x^3$ using the Newton–Raphson Method.

Iteration	x_n	$f(x_n)$	$f'(x_n)$	$x_{n+1} = x_n - \dfrac{f(x_n)}{f'(x_n)}$
0	1.3000	3.757	-1.530	3.7560
1	3.7560	2.146	6.300	3.4130
2	3.4130	0.581	2.994	3.2190
3	3.2190	0.154	1.473	3.1130
4	3.1130	0.040	0.717	3.0580
5	3.0580	0.0101	0.3576	3.0290
6	3.0290	0.00257	0.1800	3.0150
7	3.0150	0.000642	0.0890	3.0090

5.3.7 Practice Questions

1. Use the Newton-Raphson method to find the root of $f(x) = x^2 - 4x + 3$ starting with $x_0 = 2.5$.

2. Find the root of $f(x) = \tan(x) - x$ using an initial guess of $x_0 = 4$.

5.3.8 Detailed Iterations in Table

We solve

$$f(x) = \tan(x) - x,$$

using the Newton–Raphson method with the iteration formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

The derivative of the function is

$$f'(x) = \sec^2(x) - 1.$$

Starting with an initial approximation $x_0 = 4$, the following table summarizes the iterations:

Chapter 6

Function Approximation

6.1 Introduction

Function approximation is a fundamental concept in numerical analysis, where the goal is to represent given paired data with simpler functions (often polynomials) that are easier to work with. One important application of function approximation is interpolation, in which a function is approximated by a polynomial that exactly matches the function's values at a set of specified nodes. In this chapter, we explore the Lagrange interpolation formula, a classical method to construct such an interpolating polynomial.

6.2 Lagrange Interpolation Formula

6.2.1 Definition and Notation

Suppose we are given a set of $n+1$ distinct points

$$(x_0, y_0),\ (x_1, y_1),\ \ldots,\ (x_n, y_n)$$

where the x_i are mutually distinct. The **Lagrange interpolation formula** provides the unique polynomial $P(x)$ of degree at most n that satisfies

$$P(x_i) = y_i, \quad i = 0, 1, \ldots, n.$$

It is expressed as:

$$P(x) = \sum_{i=0}^{n} y_i\, L_i(x),$$

where each $L_i(x)$ is the **Lagrange basis polynomial** defined by

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}.$$

Notice that by construction,

$$L_i(x_k) = \delta_{ik} = \begin{cases} 1, & \text{if } i = k, \\ 0, & \text{if } i \neq k, \end{cases}$$

which ensures that $P(x_i) = y_i$ for each node.

6.2.2 Properties of the Lagrange Basis Polynomials

The Lagrange basis polynomials have several noteworthy properties:

- **Partition of Unity:** For any x, the sum of the basis polynomials equals one:

$$\sum_{i=0}^{n} L_i(x) = 1.$$

  This follows because the constant function $f(x) = 1$ is exactly interpolated by these polynomials.

- **Degree:** Each $L_i(x)$ is a polynomial of degree n, and therefore $P(x)$ is a polynomial of degree at most n.

- **Independence from y_i:** The basis polynomials depend only on the x-values (nodes). Once these are computed, they can be reused to interpolate different sets of y-values.

6.2.3 Worked Example

Problem Statement

Find the interpolating polynomial that passes through the points $(1, 1)$, $(2, 4)$, and $(3, 9)$, and evaluate the polynomial at $x = 2.5$.

Step 1: Write the Interpolating Polynomial

For three points, the Lagrange interpolation formula gives:

$$P(x) = y_0 L_0(x) + y_1 L_1(x) + y_2 L_2(x).$$

Here, $y_0 = 1$, $y_1 = 4$, and $y_2 = 9$.

Step 2: Compute the Basis Polynomials

The basis polynomials are computed as follows:

$$L_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} = \frac{(x-2)(x-3)}{(1-2)(1-3)} = \frac{(x-2)(x-3)}{(-1)(-2)} = \frac{(x-2)(x-3)}{2},$$

$$L_1(x) = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} = \frac{(x-1)(x-3)}{(2-1)(2-3)} = \frac{(x-1)(x-3)}{(1)(-1)} = -(x-1)(x-3),$$

$$L_2(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} = \frac{(x-1)(x-2)}{(3-1)(3-2)} = \frac{(x-1)(x-2)}{2 \cdot 1} = \frac{(x-1)(x-2)}{2}.$$

Step 3: Form the Interpolating Polynomial

Substitute the y-values into the polynomial:

$$P(x) = 1 \cdot \frac{(x-2)(x-3)}{2} \; - \; 4\,(x-1)(x-3) \; + \; 9 \cdot \frac{(x-1)(x-2)}{2}.$$

Step 4: Evaluate at $x = 2.5$

Now, compute each basis polynomial at $x = 2.5$:

$$L_0(2.5) = \frac{(2.5-2)(2.5-3)}{2} = \frac{(0.5)(-0.5)}{2} = -0.125,$$

$$L_1(2.5) = -(2.5-1)(2.5-3) = -(1.5)(-0.5) = 0.75,$$

$$L_2(2.5) = \frac{(2.5-1)(2.5-2)}{2} = \frac{(1.5)(0.5)}{2} = 0.375.$$

Thus,

$$P(2.5) = 1 \cdot (-0.125) + 4 \cdot 0.75 + 9 \cdot 0.375 = -0.125 + 3 + 3.375 = 6.25.$$

Interpretation

In this example, the given data points happen to lie on the function $f(x) = x^2$, which is why the interpolation polynomial simplifies exactly to $P(x) = x^2$. Evaluating at $x = 2.5$ yields $2.5^2 = 6.25$, consistent with our calculation.

6.3 Error Analysis and Convergence

While the Lagrange interpolation polynomial exactly matches the function at the nodes, the quality of the approximation between nodes depends on several factors. The error in polynomial interpolation can be expressed using the **Lagrange remainder**:

$$f(x) - P(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i),$$

where ξ is some number in the interval containing the nodes. This error bound reveals:

- The magnitude of the error depends on the $(n+1)$-th derivative of the function, which reflects the function's smoothness.
- The product $\prod_{i=0}^{n}(x - x_i)$ shows that the error is zero at the nodes, but can grow between them.

A notable issue when using equispaced nodes is **Runge's phenomenon**, where the interpolation error can increase dramatically at the interval's ends. To mitigate this, one may use **Chebyshev nodes**, which minimize the maximum error by clustering more nodes near the endpoints.

6.4 Newton's Divided Difference Formula

Newton's divided difference formula provides an efficientincremental method to construct the unique interpolation polynomial for a given set of data points. Unlike the Lagrange form, which builds the polynomial as a weighted sum of basis polynomials, the Newton form expresses the polynomial as an expanding series that can be easily updated when additional data becomes available.

The general form of the Newton interpolation polynomial is:

$$P(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + \cdots$$

$$+ f[x_0, x_1, \ldots, x_n](x - x_0)(x - x_1) \cdots (x - x_{n-1}),$$

where the notation $f[x_0, x_1, \ldots, x_k]$ represents the divided difference of order k. These divided differences are computed recursively by the formula:

$$f[x_i, x_{i+1}, \ldots, x_{i+k}] = \frac{f[x_{i+1}, \ldots, x_{i+k}] - f[x_i, \ldots, x_{i+k-1}]}{x_{i+k} - x_i}.$$

The zeroth divided differences are simply the function values:

$$f[x_i] = f(x_i).$$

Advantages of the Newton Form

- **Incremental Construction:** New data points can be added without recomputing the entire polynomial.
- **Efficiency:** The recursive structure of divided differences leads to a straightforward computation that often requires fewer arithmetic operations than the Lagrange form.
- **Numerical Stability:** When implemented carefully (especially using the nested multiplication form), the Newton form can offer better numerical stability.

Example: Constructing the Newton Interpolating Polynomial

Consider the following set of data points:

x	$f(x)$
1	3
2	6
4	5
5	4

We will construct the Newton interpolation polynomial $P(x)$ using these points.

Step 1: Compute Divided Differences **Zeroth Divided Differences:**

$$f[x_0] = f(1) = 3, \quad f[x_1] = f(2) = 6, \quad f[x_2] = f(4) = 5, \quad f[x_3] = f(5) = 4.$$

First Divided Differences:

$$\begin{aligned} f[x_0, x_1] &= \frac{f(2) - f(1)}{2 - 1} = \frac{6 - 3}{1} = 3, \\ f[x_1, x_2] &= \frac{f(4) - f(2)}{4 - 2} = \frac{5 - 6}{2} = -0.5, \\ f[x_2, x_3] &= \frac{f(5) - f(4)}{5 - 4} = \frac{4 - 5}{1} = -1. \end{aligned}$$

Second Divided Differences:

$$\begin{aligned} f[x_0, x_1, x_2] &= \frac{f[x_1, x_2] - f[x_0, x_1]}{4 - 1} = \frac{-0.5 - 3}{3} = \frac{-3.5}{3} \approx -1.1667, \\ f[x_1, x_2, x_3] &= \frac{f[x_2, x_3] - f[x_1, x_2]}{5 - 2} = \frac{-1 - (-0.5)}{3} = \frac{-0.5}{3} \approx -0.1667. \end{aligned}$$

Third Divided Difference:

$$f[x_0, x_1, x_2, x_3] = \frac{f[x_1, x_2, x_3] - f[x_0, x_1, x_2]}{5 - 1} = \frac{-0.1667 - (-1.1667)}{4} = \frac{1}{4} = 0.25.$$

The divided difference table can be summarized as follows:

x	$f[x]$	$f[x_0, x_1]$	$f[x_0, x_1, x_2]$	$f[x_0, x_1, x_2, x_3]$
1	3	3	-1.1667	0.25
2	6	-0.5	-0.1667	
4	5	-1		
5	4			

Step 2: Write the Newton Interpolating Polynomial Using the divided differences, the Newton form of the interpolation polynomial is:

$$P(x) = f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + f[x_0, x_1, x_2, x_3](x - x_0)(x - x_1)(x - x_2).$$

Substitute the computed values:

$$P(x) = 3 + 3(x - 1) - 1.1667\,(x - 1)(x - 2) + 0.25\,(x - 1)(x - 2)(x - 4).$$

This polynomial $P(x)$ is the unique polynomial of degree at most 3 that passes through the four given points.

Step 3: Using the Polynomial The Newton form is especially useful if we wish to evaluate $P(x)$ at a particular x or if more points are to be added later. The nested multiplication (Horner's method) can be applied to the Newton form for efficient evaluation.

Summary Table of Divided Differences:

x	$f(x)$	$f[x_i, x_{i+1}]$	$f[x_i, x_{i+1}, x_{i+2}]$	$f[x_i, x_{i+1}, x_{i+2}, x_{i+3}]$
1	3	3	-1.1667	0.25
2	6	-0.5	-0.1667	
4	5	-1		
5	4			

Conclusion: The Newton divided difference method not only constructs the interpolating polynomial in an incremental manner but also organizes the computation into a table that makes the process transparent. Each level of divided differences provides additional terms for the polynomial, and the final polynomial is expressed in a nested form that is both computationally efficient and easy to update.

This approach is particularly advantageous when interpolating data that may be extended or modified, allowing us to add new points with minimal extra work.

Chapter 7

Numerical Integration

In many practical scenarios, the analytical evaluation of definite integrals is either very difficult or impossible. Numerical integration offers a suite of techniques to approximate definite integrals. These methods are particularly valuable in physics and engineering, where integrals often arise in applications such as area estimation, work, energy, or probability.

We aim to approximate

$$\int_a^b f(x)\,dx$$

by replacing the integrand $f(x)$ with a simpler function (linear, quadratic, cubic, etc.) that closely matches $f(x)$ on small subintervals.

7.1 Trapezoidal Rule

7.1.1 Derivation

Divide the interval $[a, b]$ into n equal subintervals of width $h = \frac{b-a}{n}$. Over each subinterval $[x_i, x_{i+1}]$, we approximate $f(x)$ using a straight line:

$$f(x) \approx f(x_i) + \frac{f(x_{i+1}) - f(x_i)}{h}(x - x_i)$$

The area under this line is the area of a trapezoid:

$$\int_{x_i}^{x_{i+1}} f(x)\,dx \approx \frac{h}{2}\left[f(x_i) + f(x_{i+1})\right]$$

Adding all trapezoids:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)\right]$$

7.1.2 Error Term

The error in the trapezoidal rule is proportional to the second derivative:

$$E_T = -\frac{(b-a)^3}{12n^2} f''(\xi), \quad \xi \in (a, b)$$

7.2 Simpson's 1/3 Rule

7.2.1 3.1 Derivation

Simpson's 1/3 Rule fits a second-degree polynomial (a parabola) through every three consecutive points. This requires an even number of subintervals (n must be even). For subintervals of width h:

$$\int_{x_i}^{x_{i+2}} f(x)\,dx \approx \frac{h}{3}\left[f(x_i) + 4f(x_{i+1}) + f(x_{i+2})\right]$$

Summing over all such pairs:

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(x_0) + 4\sum_{\text{odd } i}^{n-1} f(x_i) + 2\sum_{\text{even } i}^{n-2} f(x_i) + f(x_n)\right]$$

7.2.2 Error Term

The error in Simpson's 1/3 Rule depends on the fourth derivative:

$$E_S = -\frac{(b-a)^5}{180n^4} f^{(4)}(\xi), \quad \xi \in (a,b)$$

7.3 Simpson's 3/8 Rule

7.3.1 Derivation

Simpson's 3/8 Rule uses a cubic polynomial to approximate $f(x)$ over three subintervals (four points). So n must be a multiple of 3.

For $h = \frac{b-a}{n}$:

$$\int_{x_i}^{x_{i+3}} f(x)\,dx \approx \frac{3h}{8}\left[f(x_i) + 3f(x_{i+1}) + 3f(x_{i+2}) + f(x_{i+3})\right]$$

Applying this over all such groups:

$$\int_a^b f(x)\,dx \approx \frac{3h}{8}\left[f(x_0) + 3\sum_{\substack{i=1 \\ i \mod 3 \neq 0}}^{n-1} f(x_i) + 2\sum_{\substack{i=3 \\ i \mod 3 = 0}}^{n-3} f(x_i) + f(x_n)\right]$$

7.3.2 Error Term

$$E_{3/8} = -\frac{3h^5}{80} f^{(4)}(\xi), \quad \xi \in (a,b)$$

7.4 Example: Approximate $\int_0^1 x^2\,dx$

Let us compute the integral using $n = 4$ subintervals for Trapezoidal and Simpson's 1/3, and $n = 3$ for Simpson's 3/8 Rule.

7.4.1 Exact Value

$$\int_0^1 x^2\,dx = \left[\frac{x^3}{3}\right]_0^1 = \frac{1}{3} \approx 0.3333$$

7.4.2 Trapezoidal Rule ($n = 4$)

$$h = \frac{1-0}{4} = 0.25$$

$$f(x_0) = 0^2 = 0, \quad f(x_1) = 0.0625, \quad f(x_2) = 0.25, \quad f(x_3) = 0.5625, \quad f(x_4) = 1$$

$$\int_0^1 x^2 dx \approx \frac{0.25}{2}\left[0 + 2(0.0625 + 0.25 + 0.5625) + 1\right] = 0.125 \cdot 2.75 = \boxed{0.34375}$$

7.4.3 Simpson's 1/3 Rule ($n = 4$)

$$\int_0^1 x^2 dx \approx \frac{0.25}{3}\left[0 + 4(0.0625 + 0.5625) + 2(0.25) + 1\right] = \frac{1}{12} \cdot 4 = \boxed{0.3333}$$

7.4.4 Simpson's 3/8 Rule ($n = 3$)

$$h = \frac{1-0}{3} = 0.3333, \quad x_0 = 0,\ x_1 = \frac{1}{3},\ x_2 = \frac{2}{3},\ x_3 = 1$$

$$f(x_0) = 0, \quad f(x_1) = \left(\frac{1}{3}\right)^2 = \frac{1}{9}, \quad f(x_2) = \frac{4}{9}, \quad f(x_3) = 1$$

$$\int_0^1 x^2 dx \approx \frac{3 \cdot 0.3333}{8}\left[0 + 3\left(\frac{1}{9} + \frac{4}{9}\right) + 1\right] = 0.125 \cdot \left(3 \cdot \frac{5}{9} + 1\right) = 0.125 \cdot \left(\frac{15}{9} + 1\right)$$

$$= 0.125 \cdot \left(\frac{24}{9}\right) = 0.125 \cdot 2.\overline{6} = \boxed{0.3333}$$

7.5 Conclusion

- The Trapezoidal Rule is simple but less accurate for nonlinear functions.
- Simpson's 1/3 Rule improves accuracy by using parabolas and is ideal when n is even.
- Simpson's 3/8 Rule uses cubic interpolation and is particularly useful when n is a multiple of 3.
- All methods give close results for smooth functions, with Simpson's rules typically offering higher accuracy.

7.6 C++ code

```cpp
#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;

// Function to integrate
double f(double x) {
    return x * x; // Change this function as needed
}

// Trapezoidal Rule
double trapezoidal(double a, double b, int n) {
    double h = (b - a) / n;
    double sum = f(a) + f(b);
    for (int i = 1; i < n; ++i) {
        sum += 2 * f(a + i * h);
    }
    return (h / 2) * sum;
}

// Simpson's 1/3 Rule (n must be even)
double simpsonsOneThird(double a, double b, int n) {
    if (n % 2 != 0) {
        cerr << "Simpson's 1/3 Rule requires even number of intervals.\n";
        return NAN;
    }
    double h = (b - a) / n;
    double sum = f(a) + f(b);
    for (int i = 1; i < n; ++i) {
        sum += (i % 2 == 0 ? 2 : 4) * f(a + i * h);
    }
    return (h / 3) * sum;
}

// Simpson's 3/8 Rule (n must be a multiple of 3)
double simpsonsThreeEighth(double a, double b, int n) {
    if (n % 3 != 0) {
        cerr << "Simpson's 3/8 Rule requires number of intervals to be a multiple of 3.\n";
        return NAN;
    }
    double h = (b - a) / n;
    double sum = f(a) + f(b);
    for (int i = 1; i < n; ++i) {
        sum += ((i % 3 == 0) ? 2 : 3) * f(a + i * h);
    }
    return (3 * h / 8) * sum;
```



```
}

int main() {
    double a = 0.0, b = 1.0;
    int n_trap = 4;
    int n_simp13 = 4;
    int n_simp38 = 3;

    cout << fixed << setprecision(6);

    cout << "Integral of f(x) = x^2 from 0 to 1:\n";
    cout << "---------------------------------------\n";
    cout << "Exact Value          : " << 1.0 / 3.0 << "\n";
    cout << "Trapezoidal Rule     : " << trapezoidal(a, b, n_trap
   ) << "\n";
    cout << "Simpson's 1/3 Rule   : " << simpsonsOneThird(a, b,
   n_simp13) << "\n";
    cout << "Simpson's 3/8 Rule   : " << simpsonsThreeEighth(a, b
   , n_simp38) << "\n";

    return 0;
}
```


Chapter 8

ODE

```cpp

#include <iostream>

using namespace std;

// Define dy/dx = x + y
double f(double x, double y) {
    return x + y;
}

// Define dz/dx = z - x
double g(double x, double z) {
    return z - x;
}

int main() {
    // Initial conditions
    double x = 0.0;
    double y = 1.0;
    double z = 0.0;
    double h = 0.1;
    double x_end = 0.2;

    // Number of steps
    int N = (x_end - x) / h;

    // Print header
    cout << fixed << setprecision(4);
    cout << "x\t\ty\t\tz\n";
    cout << x << "\t\t" << y << "\t\t" << z << endl;

    // Euler's method loop
    for (int i = 0; i < N; ++i) {
        double f_val = f(x, y);
        double g_val = g(x, z);

        y = y + h * f_val;
```

```
38          z = z + h * g_val;
39          x = x + h;
40
41          cout << x << "\t\t" << y << "\t\t" << z << endl;
42      }
43
44      return 0;
45  }
```

