Introduction to computational Physics

UPES Dehradun

Introduction to computational Physics - 2024

Nitesh Kumar

November 7, 2024

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.

```
$ nano 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 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.

1.2.1 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.

1.3 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.

1.3.1 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
```

1.4 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.

1.4.1 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.

1.4.2 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.

1.4.3 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.

1.4.4 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.

1.4.5 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.

1.4.6 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.

1.5 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.

1.6 Example Programs

1.6.1 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!
```

1.6.2 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
```

1.6.3 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
```

1.6.4 Arrays

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

```
4
5   a = [1, 2, 3]
6   b = RESHAPE([1.0, 2.0, 3.0, 4.0], [2, 2])
7
8   PRINT *, 'Array a:', a
9   PRINT *, 'Array b:'
10  DO i = 1, 2
11    PRINT *, b(i, :)
12  END DO
13 END PROGRAM arrays
```

To compile the program, use the following commands:

```
1 $ gfortran arrays.f90 -o arrays
```

To run the compiled program, use:

```
1 $ ./arrays
```

Output:

```
1 Array a:           1           2           3
2 Array b:
3    1.00000000        2.00000000
4    3.00000000        4.00000000
```

1.6.5 Functions

The syntax of the function is given below.

```
1     type FUNCTION func-name(arg1, arg2, ....)
2         IMPLICIT NONE
3         [specification part]
4         [execution part]
5         [subprogram part]
6     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:

```
1 program adding
2         IMPLICIT NONE
3         INTEGER :: a, b, addition, add
4         a = 4
5         b = 6
6         addition = add(a, b)
7         print*, a, b, addition
8 END PROGRAM adding
9
10 INTEGER FUNCTION add(x, y)
11        IMPLICIT NONE
12        INTEGER, INTENT(IN) :: x, y
13        add = (x+y)
14 END FUNCTION add
```

Another way of writing functions in Fortran:

```
1 program TwoFunctions
2         IMPLICIT NONE
3         REAL :: a, b, A_mean, G_mean
4         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
```

1.6.6 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
```

1.6.7 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.

1.7 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.

1.7.1 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.

1.7.2 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.

1.7.3 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]

  ! Call LAPACK subroutine to solve the system of equations A*x = B
  CALL dgesv(n, 1, A, n, ipiv, B, n, info)

```



```
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.

1.7.4 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
```

1.7.5 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 $Ax = b$.

1.8 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.

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

```
#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

```
#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:


```cpp
#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
```

The flowchart of switch-case statements:

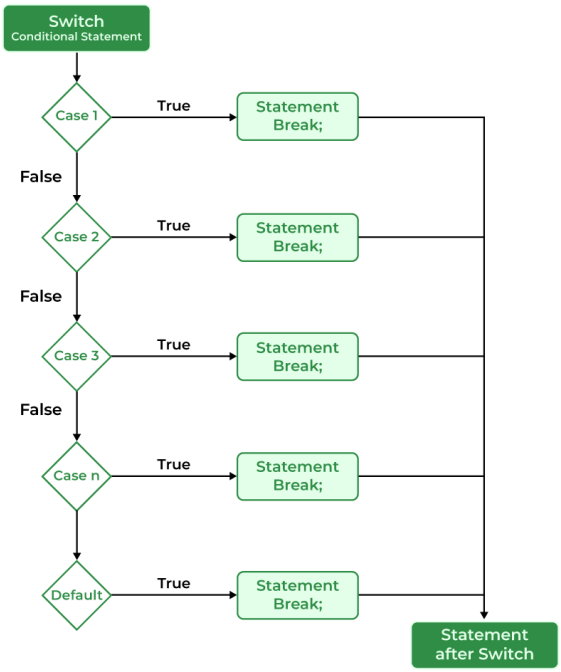

Figure 2.1: Switch case statements

2.9.4 Arrays and vectors

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

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

    cout << "Array elements: ";
    for (int i = 0; i < 3; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    cout << "Vector elements: ";
    for (int i = 0; i < vec.size(); i++) {
        cout << vec[i] << " ";
    }
    cout << endl;

    return 0;
}
```

To compile the program, use the following commands:

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

To run the compiled program, use:

```
$ ./arrays
```

Output:

```
Array elements: 1 2 3
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

```
// 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

```
// 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

```
// 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.

Chapter 3

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.

3.1 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.

3.1.1 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 $b = m$.

- 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.

3.1.2 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 3.1: 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:

$$x \approx 4.49365234375$$

3.1.3 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.

3.1.4 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 $[0, 1]$.

3.2 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.

3.2.1 Method Explanation

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

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

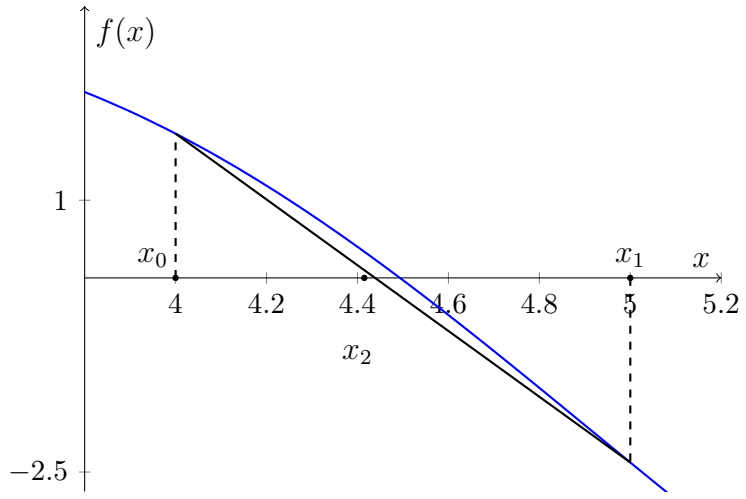

Figure 3.1: Secant method on f(x) = sin(x) - xcos(x).

3.2.2 Example: Solving $f(x) = \sin x - x\cos x = 0$ 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 3.2: 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.

3.2.3 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 $x_1 = 1$.

3.3 Newton-Raphson Method

The Newton-Raphson Method uses the derivative to find a root of a function.

3.3.1 Method Explanation

Starting with an initial guess x_0, update x using:

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

3.3.2 Example

Find the root of $f(x) = x^3 - 3x + 2$ using Newton-Raphson with $x_0 = 0.5$.

3.3.3 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 $x_0 = 4$.

3.4 Summary and Comparison of Methods

In this chapter, we explored three methods of finding roots, each with unique advantages and limitations. Practice and apply these methods to determine which is best suited for a given problem.

