MAA OMWATI DEGREE COLLEGE, HASSANPUR

EXAM NOTES

MATHEMATICAL PROGRAMMING IN C AND NUMERICAL METHODS

CLASS – B.A / B.sc 1 ST SEM (SEC)

UNIT 1

1. Programmer's Model of a Computer

The programmer's model views the computer conceptually through its functional components rather than its complex physical circuitry. It describes how software interacts with hardware resources.

- **Central Processing Unit (CPU):**
  - **Arithmetic Logic Unit (ALU):** Performs mathematical computations (addition, subtraction) and logical evaluations (AND, OR, comparisons).
  - **Control Unit (CU):** Directs the flow of data and execution of instructions fetching from memory.
  - **Registers:** High-speed, small-capacity storage locations directly inside the CPU used to hold data temporarily during execution (e.g., Program Counter, Accumulator).
- **Memory Unit (Primary / RAM):** Linear array of storage cells, each with a unique physical address. The programmer visualizes RAM as bytes where variables, constants, and compiled instructions reside.
- **Input / Output (I/O) Subsystem:** Mechanisms through which data flows from external environments into the computer system and output results are presented to the user.

2. Algorithms

An **algorithm** is a well-defined, finite sequence of computational steps or instructions designed to solve a specific problem or perform a particular task.

Characteristics of a Good Algorithm:

1. **Input:** Zero or more quantities are externally supplied.
2. **Output:** At least one quantity is produced.
3. **Definiteness:** Each instruction must be clear and unambiguous.
4. **Finiteness:** The algorithm must terminate after a finite number of steps.
5. **Effectiveness:** Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper.

Example: Algorithm to check whether a number is Even or Odd

- **Step 1:** Start
- **Step 2:** Read/Input number n
- **Step 3:** Calculate remainder r = n \pmod 2
- **Step 4:** If r = 0, then print "n is Even", else print "n is Odd"
- **Step 5:** Stop

3. Flow Charts

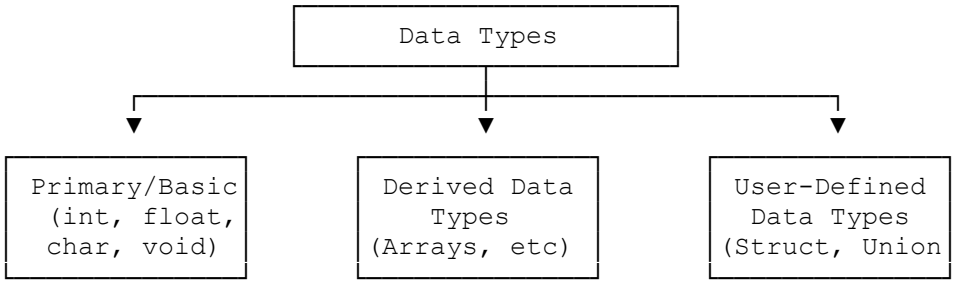
A **flowchart** is a graphical or pictorial representation of an algorithm. It uses standard geometric symbols connected by arrows to illustrate the sequence of operations and data flow.

Standard Flowchart Symbols:

- **Oval (Terminal):** Indicates the **Start** or **End** of the program flow.
- **Parallelogram (Data):** Represents **Input** (`scanf`) and **Output** (`printf`) operations.
- **Rectangle (Process):** Represents internal **Arithmetic/Data Manipulation** operations (e.g., x = a + b).
- **Diamond (Decision):** Represents a **Conditional Branching** or logical test (e.g., Is x > y?).
- **Flowlines (Arrows):** Indicate the precise direction of program execution.

4. Data Types

Data types specify the type of data a variable can hold, the memory space allocated to it, and the range of values it can represent. In languages like C, data types are broadly classified into:



1. **Primary / Basic Data Types:**
    - `int`: Stores whole numbers (typically takes 2 or 4 bytes). Example: `int age = 21;`
    - `float`: Stores single-precision floating-point numbers (decimals, 4 bytes). Example: `float pi = 3.14;`
    - `double`: Stores double-precision floating-point numbers (8 bytes). Example: `double val = 3.1415926535;`

    - `char`: Stores a single character (1 byte) enclosed in single quotes. Example: `char grade = 'A';`
    - `void`: Represents an empty set of values or absence of a data type (commonly used with functions returning nothing).
2. **Modifiers:** Keywords used to alter the base size and range of basic data types (e.g., `signed`, `unsigned`, `short`, `long`).

5. Operators and Expressions

Operators

An **operator** is a symbol that tells the compiler to perform specific mathematical, relational, or logical manipulations on operands.

- **Arithmetic Operators:** Basic math operations (`+`, `-`, `*`, `/`, `%`). Note: `%` is the modulus operator used for remainder and works only with integers.
- **Relational Operators:** Compare two values (`==`, `!=`, `>`, `<`, `>=`, `<=`). They evaluate to a boolean-equivalent (`1` for true, `0` for false).
- **Logical Operators:** Combine conditions (`&&` Logical AND, `||` Logical OR, `!` Logical NOT).
- **Assignment Operators:** Assign values to variables (`=`, `+=`, `-=`, `*=`, `/=`).
- **Increment / Decrement Operators:** Unary operators to add or subtract 1 (`++`, `--`). E.g., `++x` (pre-increment), `x++` (post-increment).
- **Conditional (Ternary) Operator (`? :`):** Shorthand for `if-else` statements.
    - *Syntax:* `Condition ? Expression1 : Expression2;`
- **Bitwise Operators:** Perform operations at the binary bit-level (`&` AND, `|` OR, `^` XOR, `<<` Left shift, `>>` Right shift).

Expressions

An **expression** is a valid combination of constants, variables, operators, and function calls that evaluates to a single specific value.

- *Example:* `result = (a + b) * c / 2;`
- **Operator Precedence & Associativity:** Determines the hierarchy of execution when an expression has multiple operators. Operators with higher precedence are evaluated first. If precedence is equal, associativity (left-to-right or right-to-left) decides the evaluation direction.

6. Input / Output (I/O) Functions

I/O functions allow a program to interact with users by reading input data and writing computed output results. In standard programming environments (like C), these are categorized as formatted and unformatted library functions header-defined in `<stdio.h>`.

1. Formatted I/O Functions

These functions allow input and output data to be read or displayed in a specific user-defined format using format specifiers (like `%d` for integers, `%f` for floats, `%c` for characters).

- **`printf()`:** Writes formatted output to the standard output screen.
  - *Syntax:* `printf("Format Specifiers", variables);`
  - *Example:* `printf("The age is %d", age);`
- **`scanf()`:** Reads formatted input data from the standard input keyboard.
  - *Syntax:* `scanf("Format Specifiers", &variable_address);`
  - *Example:* `scanf("%d", &age);` *(Note the address-of operator `&`)*

2. Unformatted I/O Functions

These functions handle single characters or strings without formatting options.

- **Character I/O:**
  - `getchar()`: Reads a single character from standard input.
  - `putchar()`: Writes a single character to standard output.
- **String I/O:**
  - `gets()`: Reads a string line from input.
  - `puts()`: Writes a string line to output.

6. Input / Output Functions (Continuation)

Formatted Output Functions (`printf`)

The `printf()` function is used to output values of variables, constants, or messages to the standard output screen (monitor). It uses format specifiers preceded by a percent sign (`%`) to match the data type of the variables being printed.

- **Common Format Specifiers:**
  - `%d` or `%i`: Signed integer
  - `%f`: Decimal floating-point
  - `%c`: Single character
  - `%s`: String of characters
  - `%lf`: Double precision floating-point
- **Example:**

  C

```
int roll_no = 15;
float marks = 89.5;
printf("Roll Number: %d and Marks: %.2f", roll_no, marks);
```

7. Decision Control Structure

In general, programs execute sequentially line by line. However, real-world programming often requires making choices based on certain conditions. **Decision control structures** allow a program to alter its execution path depending on whether a condition evaluates to true or false.

A. Decision Statements

Decision statements evaluate an expression and direct the flow of control to specific blocks of code.

1. **`if` Statement:**
    - Evaluates a condition. If it is true, the enclosed block of statements executes; otherwise, it is skipped.
    - *Syntax:*

      C

      ```
      if (condition) {
          // Statements executed if condition is true
      }
      ```

2. **`if-else` Statement:**
    - Provides an alternative path of execution if the condition evaluates to false.
    - *Syntax:*

      C

      ```
      if (condition) {
          // True block
      } else {
          // False block
      }
      ```

3. **Nested `if-else` Statement:**
    - An `if-else` construct placed inside another `if` or `else` block to check multiple dependent conditions.
4. **`if-else-if` Ladder:**
    - Used when there are multiple mutually exclusive choices or branches.
    - *Syntax:*

      C

      ```
      if (condition1) {
          // Code block 1
      } else if (condition2) {
          // Code block 2
      } else {
      ```

```
    // Default code block if all conditions fail
}
```

5. **`switch-case` Statement:**
    - A multi-way branching statement that tests a variable or expression against a list of constant case values. It is cleaner and more readable than a long `if-else-if` ladder for discrete integer or character values.
    - *Syntax:*

      C

```
switch (expression) {
    case value1:
        // Code block
        break;
    case value2:
        // Code block
        break;
    default:
        // Default code block
}
```

B. Logical and Conditional Statements

- **Logical Statements:**
    - Built using logical operators (`&&`, `||`, `!`) to combine multiple relational expressions into a single composite condition.
    - *Example:* Checking if a year is a leap year requires combining conditions: `if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))`
- **Conditional Statements (Ternary Operator `? :`):**
    - Acts as a compact, inline shorthand alternative to a simple `if-else` statement. It evaluates a condition and returns one of two expressions depending on the result.
    - *Syntax:* `variable = (condition) ? expression_if_true : expression_if_false;`
    - *Example:* `max = (a > b) ? a : b;` (Assigns the maximum of `a` and `b` to `max`).

8. Implementation of Loops (Iteration Control Structures)

Loops are used to execute a block of code repeatedly until a specified condition is met. In C, loops are categorized into entry-controlled and exit-controlled loops.

A. Entry-Controlled Loops

The condition is evaluated *before* entering the loop body. If the condition is false initially, the loop body never executes.

1. **`while` Loop:**
    - *Syntax:*

C

```
initialization;
while (condition) {
    // Body of the loop
    update_expression;
}
```

2. **`for` Loop:**
    - A compact loop structure containing initialization, condition, and increment/decrement in a single line.
    - *Syntax:*

C

```
for (initialization; condition; update) {
    // Body of the loop
}
```

B. Exit-Controlled Loop

The condition is evaluated *after* the loop body executes, ensuring the block runs at least once.

1. **`do-while` Loop:**
    - *Syntax:*

C

```
do {
    // Body of the loop
} while (condition);
```

C. Jump Statements in Loops

- **`break`:** Terminates the loop immediately and transfers control to the statement following the loop.
- **`continue`:** Skips the current iteration of the loop and forces the next iteration to begin immediately.
- **`goto`:** Unconditionally transfers control to a labeled statement anywhere in the program (generally avoided in modern structured programming).

9. Switch Statement & Case Control Structures (Deep Dive)

The **`switch` statement** is a multi-way branch statement that replaces complex `if-else-if` ladders when a variable is tested against multiple constant values.

Key Rules of `switch-case`:

1. The expression inside `switch` must evaluate to an **integer** or **character** type.
2. `case` labels must be unique, constant, and literal expressions (variables are not allowed as case labels).
3. The **`break` statement** is crucial; without it, execution "falls through" to subsequent cases until a break or the end of the switch block is reached.
4. The **`default` block** is optional and executes if none of the cases match.

- **Syntax Example:**

  C

```
int choice = 2;
switch (choice) {
    case 1:
        printf("Option One");
        break;
    case 2:
        printf("Option Two"); // This executes
        break;
    default:
        printf("Invalid Choice");
}
```

10. Functions

A **function** is a self-contained block of code designed to perform a specific, well-defined task. Functions promote code reusability, modularity, and readability.

Components of a Function:

1. **Function Declaration (Prototype):** Tells the compiler about the function name, return type, and parameters.
   - *Example:* `int add(int a, int b);`
2. **Function Definition:** Contains the actual body and implementation of the function.
3. **Function Call:** Invokes the function from `main()` or another function.
   - *Example:* `result = add(5, 10);`

Parameter Passing Mechanisms:

- **Call by Value:** Copies the actual value of arguments into the function's formal parameters. Changes made inside the function do not affect the original variables.
- **Call by Reference:** Passes the memory address (pointers) of the variables to the function. Changes made inside the function directly reflect on the original variables.

Recursion:

- A process where a function calls itself directly or indirectly to solve smaller instances of the same problem (e.g., finding factorial or Fibonacci series). Requires a strict base case to prevent infinite recursion.

11. Preprocessors

Preprocessors are directives processed by the compiler *before* the actual compilation process begins. They all start with the hash symbol (`#`).

- **`#include` (File Inclusion):** Inserts the contents of a standard or user-defined header file into the program at that point.
  - *Example:* `#include <stdio.h>`
- **`#define` (Macro Definition):** Defines symbolic constants or macro functions.
  - *Example:* `#define PI 3.14159`
- **Conditional Compilation (`#if`, `#ifdef`, `#ifndef`, `#endif`):** Allows compiling specific blocks of code based on conditions, useful for debugging and platform-dependent code.

12. Arrays

An **array** is a derived data type that represents a fixed-size collection of elements of the **same data type** stored in **contiguous (sequential) memory locations**.

A. One-Dimensional (1D) Arrays

- **Declaration:** `data_type array_name[array_size];`
  - *Example:* `int marks[5];` (allocates space for 5 integers indexed from `0` to `4`).
- **Initialization:**
  - `int marks[5] = {90, 85, 88, 92, 79};`

B. Multi-Dimensional Arrays (e.g., 2D Arrays / Matrices)

- Represented in rows and columns, used for mathematical matrices, tables, or grids.
- **Declaration:** `data_type array_name[rows][columns];`
  - *Example:* `int matrix[2][3];` (2 rows and 3 columns).
- **Initialization:**
  - `int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};`

UNIT 2

Strings and Character Data Types

A. Character Data Type (`char`)

- In C and similar programming languages, the `char` data type is used to store a single character.
- It occupies **1 byte** of memory.
- Internally, characters are stored as their corresponding integer ASCII (American Standard Code for Information Interchange) values (e.g., character `'A'` is stored as integer `65`, and `'a'` as `97`).

B. Strings

- A **string** is formally defined as a one-dimensional array of characters terminated by a null character (`'\0'`, whose ASCII value is `0`).
- Unlike some high-level languages, C does not have a built-in string data type; instead, strings are implemented using character arrays.
- **Declaration & Initialization:**
    - `char name[10] = "MDU";` (Automatically appends `'\0'` at the end, taking 4 bytes total).
    - `char str[] = {'C', 'S', 'E', '\0'};`

14. Standard String Handling Functions

C provides a rich library of pre-defined string manipulation functions declared in the **`<string.h>`** header file.

Function	Purpose	Syntax Example	Description

Function	Purpose	Syntax Example	Description
`strlen()`	String Length	`int len = strlen(str);`	Returns the number of characters in the string, excluding the null terminator (`'\0'`).
`strcpy()`	String Copy	`strcpy(dest, src);`	Copies the contents of the source string (`src`) into the destination string (`dest`).
`strcat()`	String Concatenation	`strcat(str1, str2);`	Appends (joins) string `str2` to the end of string `str1`.
`strcmp()`	String Comparison	`int res = strcmp(s1, s2);`	Compares two strings lexicographically. Returns `0` if identical, a negative value if `s1 < s2`, and positive if `s1 > s2`.
`strrev()`	String Reverse	`strrev(str);`	Reverses the characters in the given string (Note: non-standard in strict ANSI C, but common in Turbo C / GCC environments).
`strupr()` / **`strlwr()`**	Case Conversion	`strupr(str);` / `strlwr(str);`	Converts all alphabetic characters in a string to uppercase or lowercase respectively.

15. Arithmetic Operations on Characters

Because characters are represented internally as integer ASCII codes, you can perform standard arithmetic operations directly on `char` variables and values.

Key Concepts:

1. **Implicit Type Promotion / Integer Arithmetic:**
    - When an arithmetic operation is performed on a `char`, the character is automatically promoted to an integer using its ASCII value.

2. **Examples of Character Arithmetic:**
    - `char ch = 'A';` (`65`)
    - `ch = ch + 1;`
        - Result: `65 + 1 = 66`, which corresponds to the character `'B'`.
    - **Finding Case Conversion manually:**
        - To convert lowercase to uppercase: `char upper = 'a' - 32;` (Since the ASCII gap between lowercase and uppercase letters is 32).
    - **Digit Character to Numeric Value Conversion:**
        - `int val = '5' - '0';` (ASCII of `'5'` is `53` and `'0'` is `48`; `53 - 48 = 5`, yielding the actual integer value `5`).

Structures in C

A. Definition of a Structure

A **structure** is a user-defined data type in C that allows grouping variables of **different data types** under a single name. Unlike arrays (which hold elements of the same data type), structures are ideal for representing complex, real-world entities (e.g., a Student record containing an ID, Name, and Marks).

- **Syntax for Declaration:**

  C

```
struct structure_name {
    data_type member1;
    data_type member2;
    // ...
};
```

- **Example:**

  C

```
struct Student {
    int roll_no;
    char name[50];
    float marks;
};
```

B. Using Structures

Once a structure is declared, structure variables can be defined and their members can be accessed.

1. **Defining Structure Variables:**
    - *Method 1 (During declaration):*

C

```
struct Student {
    int roll_no;
    char name[50];
    float marks;
} s1, s2; // s1 and s2 are structure variables
```

- *Method 2 (Later in code):*

C

```
struct Student s1;
```

2. **Accessing Structure Members:**
   - Members of a structure are accessed using the **dot operator (.)** (also known as the direct selection operator).
   - *Example:*

C

```
s1.roll_no = 101;
strcpy(s1.name, "Aman");
s1.marks = 88.5;

printf("Roll No: %d, Name: %s, Marks: %.2f", s1.roll_no, s1.name,
s1.marks);
```

C. Structures in Arrays (Array of Structures)

When you need to store records for multiple entities (e.g., details of 50 students), you use an **array of structures**, where each element of the array is a separate structure variable.

- **Declaration & Initialization Example:**

C

```
struct Student class_mdu[3]; // Array of 3 student structures

// Storing data for the first student
class_mdu[0].roll_no = 1;
strcpy(class_mdu[0].name, "Rahul");
class_mdu[0].marks = 92.0;

// Accessing via loop
for(int i = 0; i < 3; i++) {
    printf("Roll: %d\tName: %s\n", class_mdu[i].roll_no,
class_mdu[i].name);
}
```

D. Arrays in Structures (Arrays as Structure Members)

A structure can contain one or more arrays as its members. This is useful when an entity requires storing a collection of uniform items inside it (e.g., a student structure holding marks for multiple subjects).

- **Declaration & Usage Example:**

  C

```
struct ReportCard {
    int roll_no;
    int marks[5]; // Array as a member inside a structure
};

struct ReportCard student1;
student1.roll_no = 10;

// Assigning values to the array inside the structure
student1.marks[0] = 80;
student1.marks[1] = 85;
// ...
```

. Pointers in C

A. Definition of a Pointer

A **pointer** is a derived data type in C that stores the **memory address** of another variable (instead of storing its actual data value). Pointers provide direct access to hardware memory and are crucial for efficient memory management, dynamic allocation, and passing parameters by reference.

- **Key Operators:**
  - **Address-of Operator (`&`):** Returns the memory address of a variable.
  - **Dereference / Indirection Operator (`*`):** Accesses the value stored at the memory address held by a pointer.
- **Declaration and Initialization Syntax:**

  C

```
data_type *pointer_name;
```

  - *Example:*

    C

```
int var = 20;     // Normal integer variable
int *ptr;         // Pointer to an integer
```

```
ptr = &var;       // ptr now holds the memory address of var
```

B. Pointer Data Types

While a pointer always stores an address (which is fundamentally an integer value representing a memory location), it must be explicitly typed so the compiler knows how many bytes of memory to read or write when dereferencing.

- **Type Compatibility:**
  - An `int *` pointer points to an integer variable.
  - A `char *` pointer points to a character variable.
- **Pointer to Void (`void *`):** A generic pointer that can point to any data type, but it cannot be directly dereferenced without explicit typecasting.

18. Pointers and Arrays

There is a deep and fundamental relationship between pointers and arrays in C. The array name itself acts as a **constant pointer** pointing to the base address (the first element, index `0`) of the array.

A. Array Element Access via Pointers

- If `int arr[5] = {10, 20, 30, 40, 50};` and `int *ptr = arr;` (or `ptr = &arr[0];`), then:
  - `arr[i]` is equivalent to `*(arr + i)`
  - `ptr[i]` is equivalent to `*(ptr + i)`
- **Pointer Arithmetic:** When you add an integer `i` to a pointer, the memory address shifts by `i * sizeof(data_type)` bytes, not just `i` raw bytes.

19. Pointers and Functions

Pointers allow variables to be manipulated across different function scopes. This is primarily used to implement **Call by Reference**.

A. Call by Reference (Passing Pointers to Functions)

By default, C uses *Call by Value*, meaning functions receive copies of variables and cannot modify the original values. By passing the memory address of variables using pointers, a function can directly modify the variables in the calling function.

- **Example Code Structure:**

  C

  ```
  #include <stdio.h>
  ```

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 10, y = 20;
    printf("Before swap: x = %d, y = %d\n", x, y);

    swap(&x, &y); // Passing addresses

    printf("After swap: x = %d, y = %d\n", x, y);
    return 0;
}
```

B. Functions Returning Pointers

A function can also return a pointer to the calling function. This is frequently used with dynamic memory allocation or when returning strings/arrays.

- *Syntax:* `data_type* function_name(parameters) { ... }`

UNIT -3

1. Basic Definitions

- **Algebraic Equation:** An equation of the form f(x) = 0 consisting of polynomial terms (e.g., x^3 - 4x - 9 = 0).
- **Transcendental Equation:** An equation containing trigonometric, logarithmic, exponential, or other non-algebraic functions (e.g., \sin(x) - 1 = 0 or e^x - 3x = 0).
- **Root of an Equation:** A value of x(say alpha) for which f(alpha) = 0.
- **Bracketing Methods vs. Open Methods:**
    - *Bracketing methods* (like Bisection and Regula-Falsi) start with two points enclosing the root and always converge.
    - *Open methods* (like Secant and Newton-Raphson) require one or more initial guesses and may not always converge.

2. Bisection Method (Bolzano's Method)

The **Bisection method** is the simplest bracketing method based on the Intermediate Value Theorem. If a function f(x) is continuous in the interval [a, b] and f(a) ,f(b) < 0 (meaning opposite signs), then at least one real root exists between a and b.

- **Formula for Midpoint:**

  x_n = {a + b}/2

- **Working Procedure:**
    1. Find two initial guesses a and b such that f(a) \f(b) < 0.
    2. Calculate midpoint x_1 = (a + b) / 2.
    3. Check the value of f(x_1):
        - If f(x_1) = 0, then x_1 is the exact root.
        - If f(a) \ f(x_1) < 0, the root lies between a and x_1, so set b = x_1.
        - If f(b) \ f(x_1) < 0, the root lies between x_1 and b, so set a = x_1.
    4. Repeat the steps until the desired accuracy is achieved.
- **Characteristics:** Guaranteed convergence, but slow linear rate of convergence.

3. Regula-Falsi Method (False Position Method)

The **Regula-Falsi method** is also a bracketing method, but instead of bisecting the interval, it uses a straight line (chord) connecting the points (a, f(a)) and (b, f(b)) to intersect the x-axis. It converges faster than the Bisection method.

- **Formula:**
- x_1 = {a f(b) - b \ f(a)}/{f(b) - f(a).

  (Alternatively expressed as: x_1 = a - {(b - a) /f(a)}{f(b) - f(a)})

- **Working Procedure:**
    1. Choose initial interval [a, b] such that (a) f(b) < 0.
    2. Calculate the next approximation x_1 using the formula above.

3. Check f(x_1):
    - If f(x_1) = 0, x_1 is the root.
    - If f(a) \ f(x_1) < 0, set b = x_1.
    - Otherwise, set a = x_1.
4. Repeat until convergence.

4. Secant Method

The **Secant method** is an open-bracket variant of the Regula-Falsi method. It does *not* require the function to change signs across the initial points, making it an open method. It uses a secant line passing through two previous guesses.

- **Formula:**

  x_{n+1} = x_n - {x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n)

- **Working Procedure:**
    1. Start with two initial approximations, x_0 and x_1 (they do not need to bracket the root).
    2. Apply the formula to find the next approximation x_2.
    3. Update values (x_0 = x_1, x_1 = x_2) and iterate until x_{n+1} - x_n< tolerence.
- **Characteristics:** Faster convergence than Bisection and Regula-Falsi, but can diverge if initial guesses are poor.

5. Newton-Raphson Method (Tangent Method)

(Commonly included alongside the above three in Section III) The **Newton-Raphson method** is a powerful open method that uses the tangent line to the curve at the current guess to find a closer approximation to the root.

- **Formula:**x_{n+1} = x_- {f(x_n)}/{f'(x_n)}

  where f'(x_n) is the first derivative of the function evaluated at x_n.

- **Working Procedure:**
    1. Choose an initial guess x_0 close to the root.
    2. Compute the next approximation using the formula.
    3. Repeat until x_{n+1} - x_n is less than the desired error tolerance.
- **Characteristics:** Quadratic convergence (very fast), but requires the derivative f'(x) to be easily calculable

Newton’s Iterative Method for the p^{{th}} Root

To find the p^{{th}} root of a positive real number A, we want to solve for x in the equation:

x^p - A = 0

Let our function be f(x) = x^p - A. Its derivative with respect to x is:

f'(x) = p x^{p-1}

Using the standard Newton-Raphson iteration formula (x_{n+1} = x_n - {f(x_n)}/{f'(x_n)}):

x_{n+1} = x_n - {x_n^p - A}{p x_n^{p-1}}

Simplifying this expression:

x_{n+1} = x_n - {x_n}/{p} + {A}/{p x_n^{p-1}}

x_{n+1} = {1}/{p} *[(p - 1)x_n + {A}{x_n^{p-1}}]

Thus, the general iterative formula to find the p^{\text{th}} root of A is:

x_{n+1} = {1}/*{p} ((p-1)x_n + {A}/{x_n^{p-1}}

Order of Convergence

The order of convergence of Newton's iterative method for finding the p^{text{th}} root (provided p is a finite integer and the initial guess x_0) is **quadratic**, which means its **order of convergence is 2**.

Proof / Verification via Fixed-Point Theory:

An iterative method can be written in the general form x_{n+1} = g(x_n), where the iteration function here is:

g(x) = {1}/{p} 8((p-1)x + {A}{x^{p-1}}

Let alpha = sqrt[p]{A} be the exact root, meaning alpha^p = A.

1. **Evaluate g(\alpha):**

   g(\alpha) = {1}/{p} ((p-1)alpha + {alpha^p}/{alpha^{p-1}})

UNIT 4

Introduction to Gauss-Elimination Method

The **Gauss-elimination method** is a direct algebraic technique used to solve a system of simultaneous linear equations. The primary goal is to transform the system's augmented matrix into an **upper triangular matrix** using elementary row operations. Once in this form, the system is easily solved using **back-substitution**.

Steps to Solve Using Gauss-Elimination

1. **Write the Augmented Matrix:** Represent the system of linear equations in the matrix form [A {} B], where A is the coefficient matrix and B is the column vector of constants.
2. **Forward Elimination:** Use elementary row operations to convert the matrix into an upper triangular matrix. The operations allowed are:
    - Interchanging any two rows.
    - Multiplying a row by a non-zero scalar.
    - Adding or subtracting a multiple of one row to another row.
3. **Back-Substitution:** Once the upper triangular form is achieved, solve for the variables starting from the last row (bottom-up).

Example Problem

Solve the following system of linear equations:

$$x + y + z \&= 7$$

$$x + 2y + 3z = 16$$

$$x + 3y + 4z \&= 22$$

Step 1: Write the Augmented

[1 1]

[1 7],

[1 2]

[3 16]

[1 & 3]
[4,16]

Step 2: Forward Elimination

We want to make the elements below the main diagonal (1, 1) in the first column equal to zero.

- Perform R_2 \ R_2 - R_1
- Perform R_3 \R_3 - R_1

[1 & 1 & 1 & 7]

[0 & 1 & 2 & 9]

[0 & 2 & 3 & 15]

Next, we want to make the element below the diagonal in the second column (the 2 in row 3) equal to zero.

- Perform R_3 \ R_3 - 2R_2

[1 & 1 & 1 & 7]

[0 & 1 & 2 & 9]

[0 & 0 & -1 & -3]

The matrix is now in **upper triangular form**.

Step 3: Back-Substitution

Translate the rows back into equations:

1. From row 3:

$$-1z = -3 \implies z = 3$$

2. From row 2:

$$y + 2z = 9 \implies y + 2(3) = 9 \implies y + 6 = 9 \implies y = 3$$

3. From row 1:

$$x + y + z = 7 \implies x + 3 + 3 = 7 \implies x + 6 = 7 \implies x = 1$$

Final Solution

- **x = 1**
- **y = 3**
- **z = 3**

Triangularization Method

The **Triangularization Method** is a direct numerical method used to solve a system of simultaneous linear equations. In this method, the coefficient matrix is transformed into an **upper triangular matrix** (or sometimes a lower triangular matrix) using elementary row operations. The unknowns are then obtained by **back substitution** (or forward substitution).

It is essentially the basis of **Gaussian Elimination**.

Definition

The triangularization method is the process of converting a system of linear equations into an equivalent **triangular system** by applying elementary row operations and then solving the system by substitution.

General System

Consider the system

$$a11x+a12y+a13z$$

$$a21x+a22y+a23z$$

a31x+a32y+a33z= [b1 b2 b3]

In matrix form,

AX=B

where

A=a11a21a31

a12a22a32

a13a23a33

The objective is to transform A into an upper triangular matrix:

∗00∗∗0∗∗∗

Steps of the Triangularization Method

Step 1

Write the augmented matrix of the system.

Step 2

Use elementary row operations to eliminate the elements below the main diagonal.

Step 3

Continue until the coefficient matrix becomes upper triangular.

Step 4

Solve the triangular system using **back substitution**.

Numerical Example

Solve

$$2x+y-z-3x-y+2z-2x+y+2z=8=-11=-3$$

Step 1: Augmented Matrix

$$2-3-21-11-1228-11-3$$

Step 2: Eliminate the First Column

Perform

$$R2\to 2R2+3R1 \; R3\to R3+R1$$

This gives

$$200112-111825$$

Step 3: Eliminate the Second Column

Perform

$$R3\to R3-2R2$$

Result:

$$200110-11-1821$$

The system is now in **upper triangular form**.

Step 4: Back Substitution

From the third equation,

$$-z=1 \; z=-1$$

From the second equation,

y+z=2 y=3

From the first equation,

2x+y−z=8 2x+3+1=8 2x=4 x=2

Hence,

x=2,y=3,z=−1

Advantages

- Simple and systematic procedure.
- Gives the exact solution (subject to rounding errors).
- Suitable for solving small and medium-sized systems.
- Forms the basis of Gaussian elimination and LU decomposition.

Limitations

- Requires more arithmetic operations than iterative methods for very large systems.
- Pivoting may be necessary to avoid division by very small numbers.
- Less efficient than LU decomposition when solving multiple systems with the same coefficient matrix.

Applications

- Solving simultaneous linear equations.
- Engineering and scientific computations.
- Structural analysis.
- Electrical circuit analysis.
- Finite element and numerical modeling.

Comparison with Other Methods

Method	Type	Main Idea	Best Used For
Triangularization (Gaussian Elimination)	Direct	Convert matrix to triangular form and use back substitution	Small to medium systems
Crout's Method	Direct	Decompose A=LU (diagonal of U=1)	Repeated solutions with the same coefficient matrix
Cholesky Decomposition	Direct	Decompose A=LLT	Symmetric positive definite matrices
Jacobi Method	Iterative	Use values from the previous iteration	Large diagonally dominant systems
Gauss-Seidel Method	Iterative	Use the latest available values immediately	Faster convergence than Jacobi
Relaxation (SOR)	Iterative	Gauss-Seidel with a relaxation factor ω	Faster convergence for many large sparse systems

Crout's Method (LU Decomposition) is a numerical method used to solve a system of linear equations:

$AX=B$

It decomposes the coefficient matrix A into:

$A=LU$

where:

- **L** = Lower triangular matrix
- **U** = Upper triangular matrix with all diagonal elements equal to **1**.

Steps of Crout's Method

1. Write the system in matrix form:

   $AX=B$

2. Decompose the matrix:

   $A=LU$

3. Find the elements of **L** and **U** by comparing A with the product LU.
4. Solve:
   - LY=B using **forward substitution**.
   - UX=Y using **backward substitution**.

Advantages

- Efficient for solving several systems with the same coefficient matrix but different right-hand sides.
- Requires fewer computations than repeated Gaussian elimination.
- Widely used in numerical analysis and scientific computing.

Cholesky Decomposition Method

The **Cholesky Decomposition Method** is a numerical technique used to solve systems of linear equations of the form

$AX=B$

where the coefficient matrix A is **symmetric** and **positive definite**.

Unlike Crout's or Doolittle's methods, Cholesky decomposition factorizes the matrix as

$A=LL^T$

where:

- L is a **lower triangular matrix**.
- L^T is the **transpose** of L.

Conditions for Cholesky Decomposition

The matrix A must satisfy:

1. **Symmetric**

   $A=A^T$

2. **Positive Definite**

   $x^TAx>0$

   for every non-zero vector x.

General Form

For a 3×3 matrix,

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{13} & a_{23} & a_{33} \end{bmatrix}$$

Assume

$$L=\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix}$$

Then

$A=LL^T$

where

$$L^T=\begin{bmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{bmatrix}$$

Formulae for Computing the Elements

Diagonal elements

$$l_{ii} = a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2$$

Off-diagonal elements

$$l_{ij} = \frac{1}{l_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk}\right), i > j$$

Steps of Cholesky Method

1. Check whether the matrix is **symmetric**.
2. Compute the lower triangular matrix L.
3. Solve

$$LY = B$$

using **forward substitution**.

4. Solve

$$LTX = Y$$

using **backward substitution**.

Example

Solve

$$4x + yx + 3y = 9 = 10$$

Step 1: Matrix form

$$A = \begin{bmatrix} 4 & 1 \\ 1 & 3 \end{bmatrix}, B = \begin{bmatrix} 9 \\ 10 \end{bmatrix}$$

Since A is symmetric,

Assume

$$L = \begin{bmatrix} l_{11} & l_{21} \\ 0 & l_{22} \end{bmatrix}$$

Then

$$LLT=\begin{bmatrix} l11^2 & l11l21 \\ l11l21 & l21^2+l22 \end{bmatrix}=\begin{bmatrix} 4 & 11 \\ 3 \end{bmatrix}$$

Comparing the elements,

$l11=2$ $l21=21$ $l22=3-(21)2=411=211$

Hence,

$$L=\begin{bmatrix} 2 & 0 \\ 21 & 211 \end{bmatrix}$$

Step 2: Solve LY=B

$2y1=9$ $y1=4.5$

Next,

$21(4.5)+211y2=10$ $y2=1115.5$

Step 3: Solve LTX=Y

From

$211y=y2$ $y=1131=2.818$

Then

$2x+21(2.818)=4.5$ $x=1.545$

Thus,

$x=1.545, y=2.818$

which satisfies the given equations.

Advantages

- Faster than Gaussian elimination for symmetric positive definite matrices.
- Requires approximately **half the computations** of LU decomposition.

- Numerically stable.
- Requires less memory because only one triangular matrix is stored.

Limitations

- Applicable **only to symmetric positive definite matrices**.
- Cannot be directly used for general (non-symmetric) matrices.

Comparison of LU Decomposition Methods

Method	Matrix Decomposition	Applicable To
Crout's Method	A=LU, with diagonal of U=1	General non-singular matrices
Doolittle's Method	A=LU, with diagonal of L=1	General non-singular matrices
Cholesky Method	A=LLT	Symmetric positive definite matrices

Iterative Methods?

Direct methods (Gaussian elimination, LU decomposition, Cholesky decomposition) obtain the exact solution in a finite number of steps (ignoring round-off errors). However, for **large systems of equations**, direct methods may require excessive computation and memory.

Iterative methods:

- Start with an initial guess.
- Generate improved approximations.
- Continue until the error becomes sufficiently small.

Types of Iterative Methods

The two most common iterative methods are:

1. **Jacobi Method**
2. **Gauss-Seidel Method**

General System of Equations

Consider

$$a11x1+a12x2+\cdots+a1nxna21x1+a22x2+\cdots+a2nxn:an1x1+an2x2+\cdots+annxn=b1=b2=bn$$

Rewrite each equation by solving for its diagonal variable:

$$xi=aii1bi-j=i\sum aijxj$$

These equations are then used iteratively.

Convergence Condition

For the iterative method to converge, the coefficient matrix should preferably be **strictly diagonally dominant**, i.e.,

$$|aii|>j=i\sum|aij|$$

for every row.

---

1. Jacobi Iterative Method

In the **Jacobi method**, all new values are computed **only from the previous iteration**.

Suppose the kth approximation is

$$x1(k),x2(k),\ldots,xn(k)$$

Then

$$x1(k+1)=a111(b1-a12x2(k)-\cdots-a1nxn(k))$$

Similarly,

$$xi(k+1)=aii1bi-j=i\sum aijxj(k)$$

Every variable uses only values from the previous iteration.

Algorithm

1. Choose initial values (usually zeros).
2. Compute all new values.
3. Replace old values with new ones.
4. Repeat until the desired accuracy is obtained.

Example (Jacobi Method)

Solve

$10x+y=11$

$2x+10y=12$

Rearrange:

$x10=11-y$,

$y10=12-2x$

Take initial guess

$x(0)=0, y(0)=0$

Iteration 1

$x(1)=1011-0=1.1$ $y(1)=1012-0=1.2$

Iteration 2

$x(2)=1011-1.2=0.98$ $y(2)=1012-2(1.1)=0.98$

Iteration 3

$x(3)=1011-0.98=1.002$ $y(3)=1012-2(0.98)=1.004$

Continuing,

$x\approx 1, y\approx 1$

---

2. Gauss-Seidel Method

The **Gauss-Seidel method** is similar to the Jacobi method, but each newly computed value is **used immediately** in the same iteration.

For example,

$$x1(k+1)=a111(b1−a12x2(k)−\cdots)$$

Then the newly computed x1(k+1) is used to compute x2(k+1), and so on.

Thus,

$$xi(k+1)=aii1(bi−j<i∑aijxj(k+1)−j>i∑aijxj(k))$$

Example (Gauss-Seidel Method)

Using the same equations,

$$10x+y=11\ ,$$

$$2x+10y=12$$

Initial guess

$$x(0)=0,y(0)=0$$

Iteration 1

$$x=1011−0=1.1$$

Use this value immediately,

$$y=1012−2(1.1)=0.98$$

Iteration 2

$$x=1011−0.98=1.002\ y=1012−2(1.002)=0.9996$$

Iteration 3

x=1.00004,y=0.99999

Hence,

x=1,y=1

The solution converges **faster** than the Jacobi method.

Comparison: Jacobi vs. Gauss-Seidel

Feature	Jacobi Method	Gauss-Seidel Method
Values used	Previous iteration only	Updated values used immediately
Speed	Slower	Faster
Memory requirement	More	Less
Convergence	Slower	Faster in most cases
Parallel computation	Easier	More difficult

Advantages of Iterative Methods

- Suitable for **large systems** of equations.
- Require less memory than direct methods.
- Simple to implement.
- Efficient for sparse matrices.

Limitations

- May **not converge** if the matrix is not diagonally dominant or otherwise unsuitable.
- Requires an initial guess.
- Produces approximate rather than exact solutions.

Relaxation Method (Successive Over-Relaxation – SOR)

The **Relaxation Method** is an iterative numerical technique used to solve a system of linear equations. It is an improvement over the **Gauss-Seidel method**, where a **relaxation factor** (ω) is introduced to accelerate convergence.

It is particularly useful for solving **large sparse systems of linear equations** arising in engineering and scientific computations.

System of Linear Equations

Consider the system

$$AX=B$$

or

$$a11x1+a12x2+\cdots+a1nxn$$

$$a21x1+a22x2+\cdots+a2nxn$$

$$\vdots an1x1+an2x2+\cdots+annxn =[\ b1,b2,\ ..bn\ ]$$

Principle of Relaxation

After computing a new value by the Gauss-Seidel method, it is modified using a **relaxation factor** ω:

$$xi(k+1)=(1-\omega)xi(k)+\omega\bar{x}i(k+1)$$

where

- xi(k) = old approximation,
- $\bar{x}i(k+1)$ = value obtained from the Gauss-Seidel iteration,
- ω = relaxation factor.

Types of Relaxation

1. Under-Relaxation

When

$0<\omega<1$

- Reduces the correction in each iteration.
- Provides greater stability for slowly converging or oscillatory problems.
- Convergence is usually slower.

---

2. Gauss-Seidel Method

When

$\omega=1$

the relaxation method reduces exactly to the **Gauss-Seidel method**.

---

3. Over-Relaxation (SOR)

When

$1<\omega<2$

- Accelerates convergence.
- Most commonly used in practice.
- If ω is chosen too large, the method may diverge.

Algorithm

1. Rewrite each equation by solving for the diagonal variable.
2. Choose an initial approximation.
3. Compute the Gauss-Seidel value $x^{-}i$.
4. Apply the relaxation formula

$xi(k+1)=(1-\omega)xi(k)+\omega x^{-}i(k+1)$

5. Continue iterations until the required accuracy is achieved.

Numerical Example

Solve

$4x+yx+3y=9=10$

Take

$\omega=1.2$

Initial approximation

$x(0)=0, y(0)=0$

Step 1

Gauss-Seidel value

$\bar{x}=49-0=2.25$

Relaxed value

$x=(1-1.2)(0)+1.2(2.25)=2.70$

Next,

Gauss-Seidel value

$\bar{y}=310-2.70=2.433$

Relaxed value

$y=(1-1.2)(0)+1.2(2.433)=2.92$

Further iterations continue in the same manner until the solution converges.

The exact solution is

$x=1117\approx1.545, y=1131\approx2.818$

Advantages

- Faster convergence than the Gauss-Seidel method when a suitable relaxation factor is chosen.
- Efficient for large systems of linear equations.
- Widely used in engineering, computational fluid dynamics, heat transfer, and finite element analysis.
- Reduces computation time for many practical problems.

Limitations

- The choice of the relaxation factor ω is critical.
- If ω is too large, the method may fail to converge.
- Generally applicable only to systems that satisfy convergence conditions (such as diagonal dominance or suitable matrix properties).

Comparison of Iterative Methods

Method	Relaxation Factor (ω)	Convergence Speed
Jacobi	Not used	Slow
Gauss-Seidel	ω=1	Moderate
Successive Over-Relaxation (SOR)	1<ω<2	Fast (if ω is chosen appropriately)
Under-Relaxation	0<ω<1	Slower but more stable
