**DESCRIPTION**

Below are the detailed explanation of each of the numerical methods:

**Non-linear Equations**

**Algorithm-01: Bi-Section Method**

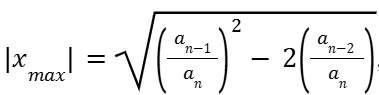
(bi\_section.hpp)

Bi-section method is a method that is used for finding solution for **Non-linear** equations. My implementation of the bi-section finds the roots of a polynomial equation with a specific degree, coefficients and error/tolerance level.

“BiSection” is the class inherited from the class “NonLinear”.

* **Initial Range Calculation:**

An initial maximum value, xmax, is determined by estimating the range where roots might exist. This is calculated by the formula-



in interval (-|xmax|, |xmax|), where, the polynomial is-



* **Calculation Method:**
  + After finding the initial range, for all i(i is incremented by 0.5 for more precision), when -|xmax| <= i <= |xmax|, f(i) & f(i + 1) was checked whether f(i) \* f(i + 1) <= 0.
    - If the condition is satisfied, a = i & b = i + 1
    - Otherwise the loop continues
  + After finding a & b, there is another internal do-while loop to find the approximate root,

*c = (a + b) / 2;*

*If (f(c) == 0): root = c;*

*else if (f(c) \* f(a) < 0) b = c;*

*else a = c;*

* + This process continues until (b – a <= error) or 100 iterations complete. Within this constraint, the answer is found for different equations with little to no error.
  + Finally, all the roots are recorded and showed in runtime.

**Algorithm-02: False position Method**

(false\_position.hpp)

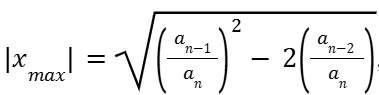
False position method is another method that is used for finding solution for **Non-linear** equations. My implementation of the false position finds the roots of a polynomial equation with a specific degree, coefficients and error/tolerance level.

“FalsePosition” is the class inherited from the class “NonLinear”.

**Implementation:**

* **Initial Range Calculation:**

An initial maximum value, xmax, is determined by estimating the range where roots might exist. This is calculated by the formula-



in interval (-|xmax|, |xmax|), where, the polynomial is-



* **Calculation Method:**
  + After finding the initial range, for all i(i is incremented by 0.5 for more precision), when -|xmax| <= i <= |xmax|, f(i) & f(i + 1) was checked whether f(i) \* f(i + 1) <= 0.
    - If the condition is satisfied, a = i & b = i + 1
    - Otherwise the loop continues
  + After finding a & b, there is another internal do-while loop to find the approximate root,

*c =(a \* f(b) – b \* f(a))/(f(b) – f(a));*

*If (f(c) == 0): root = c;*

*else if (f(c) \* f(a) < 0) b = c;*

*else a = c;*

* + This process continues until (b – a <= error) or 100 iterations complete. Within this constraint, the answer is found for different equations with little to no error.
  + Finally, all the roots are recorded and showed in runtime.

**Algorithm-03: Secant Method**

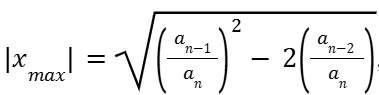
(secant.hpp)

Secant method is another method that is used for finding solution for **Non-linear** equations. My implementation of the secant finds the roots of a polynomial equation with a specific degree, coefficients and error/tolerance level.

“Secant” is the class inherited from the class “NonLinear”.

* **Initial Range Calculation:**

An initial maximum value, xmax, is determined by estimating the range where roots might exist. This is calculated by the formula-



in interval (-|xmax|, |xmax|), where, the polynomial is-



* **Calculation Method:**
  + The traversal starts from the –xmax. Each time two values, x0 and x1 are considered, where x0 is initially –xmax, and x1 = -xmax + 0.5(for better precision).
  + Secant Iteration:
    - First f(x0) and f(x1) is calculated.
    - Then, x2 is calculated using the formula-

**x2 = x1 – ((f(x1) \* (x1 – x0)) / (f(x1) – f(x0));**

which is the standard Secant formula.

* + - Then, f(x2) is calculated
    - Finally, x0 is replaced by x1 & x1 is replaced by x2 simultaneously.
    - If (|x1 – x0| <= error), the loop breaks
  + Root Comparison:
    - Before adding x1 as a root, it checks if it’s close to any previously found roots(within 0.5 units for better precision) to avoid duplicates.
    - If x1 is sufficiently distinct from previous roots, it is added to “roots” vector and the loop moves to the next root.
  + This process continues until it reaches xmax. Within this constraint, the answer is found for different equations with little to no error.
  + To display the roots, the “roots” vector is traversed.

Finally, all the roots are showed in runtime.

**Algorithm-04: Newton-Raphson Method**

(newton\_raphson.hpp)

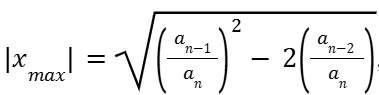
Newton-Raphson method is another method that is used for finding solution for **Non-linear** equations. My implementation of the Newton-Raphson finds the roots of a polynomial equation with a specific degree, coefficients and error/tolerance level.

“NewtonRaphson” is the class inherited from the class “NonLinear”.

**Implementation:**

* **Derivative function(fp(x)):**
  + The derivative function calculates f’(x).
  + Initializes a variable named total.
  + Iterates through all the terms with their derivatives, along with the value of x, and adds them to total.
  + Returns total.
* **Initial Range Calculation:**

An initial maximum value, xmax, is determined by estimating the range where roots might exist. This is calculated by the formula-



in interval (-|xmax|, |xmax|), where, the polynomial is-



* **Calculation Method:**
  + The traversal starts from the –xmax. Each time a value, x0 and is considered, where x0 is initially –xmax, and in each iteration the value increases by 0.5 with the previous value(for better precision).
  + Newton-Raphson Iteration:
    - First f(x0) is calculated.
    - Then, f’(x0) is calculated.
    - If, f’(x0) becomes 0, the loop breaks;
    - Then, a variable x1 is calculated using the formula-

**x1 = x0 – (f(x0) / f’(x0));**

which is the standard Newton-Raphson formula

* + - Finally, x0 is replaced by x1.
    - The loop continues for 100 iterations, for almost all of the non-linear equations, the roots are found under 100 iterations.
  + Root Comparison:
    - Before adding x1 as a root, it checks if it’s close to any previously found roots(within 0.5 units for better precision) to avoid duplicates.
    - If x1 is sufficiently distinct from previous roots, it is added to “roots” vector and the loop moves to the next root.
  + This process continues until it reaches xmax. Within this constraint, the answer is found for different equations with little to no error.
  + To display the roots, the “roots” vector is traversed.

Finally, all the roots are showed in runtime.

**Linear Equations**

**Algorithm-05: Jacobi iterative Method**

(Jacobi\_iterative.hpp)

Jacobi iterative method is a method that is used for finding solution for **Linear** equations.

“JacobiIterative” is the class inherited from the class “Linear”.

* **Row Swapping for Dominant Diagonal:**
  + To ensure convergence, first the matrix needs to be diagonally dominant.
  + This part aims to reorder the rows so that each row’s diagonal element is the largest in its column.
  + This loop compares the diagonal elements of different rows(i and j), and if a larger element is found below the current diagonal element, it swaps the rows.
  + This reordering helps make the matrix diagonally dominant, which is critical for Jacobi iterative method.
* **Initialization of Vectors:**
  + vector xp holds the values from the previous iteration, initialized to 0.
  + vector xn stores the newly calculated values of the unknowns.
  + vector ex stores the differences between xn and xp in each case for convergence checking.
* **Calculation Method:**
  + The iteration runs up to 100 times(or fewer if the convergence is achieved)
  + Jacobi Iteration:
    - For unknown xi,
      * xn[i] is initialized with the isolated term(ci/aii) for the current equation.
      * Then, the weighted contributions of other unknowns from the previous iteration, xp[j], multiplied by the corresponding coefficient aij, along with the division of aii are subtracted from it.
    - After updating xn for each unknown:
      * **Error Calculation:**
        + The error vector ex[i] holds the difference between the new and the old values of each unknown. Convergence would typically be checked here using ex[i] <= error.
      * **Update xp:**
        + The values in xp are updated with xn for the next iteration, which the main logic of Jacobi iterative method.
* **Solution Assignment and Display:**
  + Once iterations are complete, xp is assigned to this->x, which is the solution vector.
  + The showX() method displays the final solution after iterations.

Finally, all the roots are showed in runtime.

**Algorithm-06: Gauss-Seidel iterative Method**

(gauss\_seidel\_iterative.hpp)

Gauss-Seidel iterative method is another method that is used for finding solution for **Linear** equations.

“GaussSeidelIterative” is the class inherited from the class “Linear”.

* **Row Swapping for Dominant Diagonal:**
  + To ensure convergence, first the matrix needs to be diagonally dominant.
  + This part aims to reorder the rows so that each row’s diagonal element is the largest in its column.
  + This loop compares the diagonal elements of different rows(i and j), and if a larger element is found below the current diagonal element, it swaps the rows.
  + This reordering helps make the matrix diagonally dominant, which is critical for Jacobi iterative method.
* **Initialization of Vectors:**
  + vector xp holds the values from the previous iteration, initialized to 0.
  + vector xn stores the newly calculated values of the unknowns.
  + vector ex stores the differences between xn and xp in each case for convergence checking.
* **Calculation Method:**
  + The iteration runs up to 100 times(or fewer if the convergence is achieved)
  + Gauss-Seidel Iteration:
    - For unknown xi,
      * xn[i] is initialized with the isolated term(ci/aii) for the current equation.
      * Then, the weighted contributions of other unknowns from the previous iteration, xp[j], multiplied by the corresponding coefficient aij, along with the division of aii are subtracted from it.
      * **Error Calculation:**
        + The error vector ex[i] holds the difference between the new and the old values of each unknown. Convergence would typically be checked here using ex[i] <= error. But in this case it produces wrong answer sometimes. Thereby, going through 100 iterations seems to be the most optimal.
      * **Update xp:**
        + The values in xp are updated with xn for the next iteration immediately, which the main logic of Gauss-Seidel iterative method.
* **Solution Assignment and Display:**
  + Once iterations are complete, xp is assigned to this->x, which is the solution vector.
  + The showX() method displays the final solution after iterations.

Finally, all the roots are showed in runtime.

**Algorithm-07: Gauss elimination Method**

(gauss\_jordan\_elimination.hpp)

Gauss elimination method is another method that is used for finding solution for **Linear** equations.

* **swapp Function:**
  + Ensures that each diagonal entry of the matrix mat is non-zero, making the matrix easier to work with in elimination steps. If a diagonal entry is zero, it swaps rows with another row in the column with a non-zero entry.
* **gauss\_elimination Function:**
  + Performs Gaussian elimination to convert the matrix to an upper triangular form.
  + After the elimination phase, it performs back-substitution to find the solution vector.
  + **Output:** Prints the solution vector.

Finally, all the roots are showed in runtime.

**Algorithm-08: Gauss-Jordan elimination Method**

(gauss\_jordan\_elimination.hpp)

Gauss-Jordan elimination method is the extension method of Gauss elimination method that is used for finding solution for **Linear** equations.

* **swapp Function:**
  + Ensures that each diagonal entry of the matrix mat is non-zero, making the matrix easier to work with in elimination steps. If a diagonal entry is zero, it swaps rows with another row in the column with a non-zero entry.
* **gauss\_eli Function:**
  + Performs Gaussian elimination to convert the matrix to an upper triangular form without back-substitution.
* **gauss\_jordan Function:**
  + This function performs Gauss-Jordan elimination, converting the matrix into row-reduced echelon form (RREF) with zeros both above and below each pivot.
  + Returns the matrix after Gauss-Jordan elimination.
* **rref Function:**
  + Converts each pivot row in the matrix mat into leading 1s, creating a reduced row-echelon form.
  + This function normalizes each row in the matrix by dividing each element in the row by the pivot value.
  + Returns the matrix in reduced row-echelon form.
* **check\_solution Funcion:**
  + Examines the nature of the solutions by checking each row of the matrix.
  + If a row has all zeros in the coefficient part but a non-zero constant term, it indicates no solutions.
  + If a row has all zeros including the constant term, it indicates infinite solutions.
  + **Output:** Prints whether the system has a unique solution, infinite solutions, or no solution.

Finally, all the roots are showed in runtime.

**Algorithm-09: LU factorization Method**

(lu\_factorization.hpp)

LU factorization method is another method that is used for finding solution for **Linear** equations.

“LUFactorization” is the class inherited from the class “Linear”.

* **LU Decomposition (luDecomposition):**
  + This method decomposes the input matrix into a lower triangular matrix L and an upper triangular matrix U.
  + It iterates through rows and columns to update L and U based on the values in coefficients.
* **Forward Substitution (forwardSubstitution):**
  + This method is used to solve Ly = b for y.
  + It computes the intermediate solution vector y by iterating from the first row to the last, using previously computed values of y in each step.
* **Backward Substitution (backwardSubstitution):**
  + This method solves Ux = y for x after obtaining y.
  + This It iterates from the last row to the first, using previously computed values of x in each step.
* **Calculate (calculate):**
  + The calculate() method calls forwardSubstitution and backwardSubstitution in sequence and then displays the final solution vector x.

Finally, all the roots are showed in runtime.

**Differential Equations**

**Algorithm-10: Runge-Kutta Method**

(runge\_kutta.hpp)

This code implements the 4th-order Runge-Kutta method to solve differential equations numerically. It provides four different differential equations as options, each corresponding to a specific function (rk, rk2, rk3, and rk4), and allows the user to choose one. The main DE function prompts the user to input initial conditions, interval boundaries, and step size before applying the Runge-Kutta algorithm.

**Key parts:**

* **Function Definitions (rk, rk2, rk3, rk4):**
  + Each function returns the derivative dy/dx based on the chosen equation type.
  + **rk: dy/dx = x−y**
  + **rk2: dy/dx = sin(x)**
  + **rk3: dy/dx = cos(x)**
  + **rk4: dy/dx = x2 + 2xy**
* **Main** **Runge-Kutta Implementation (DE function):**
  + The user selects an equation type (1 through 4) and inputs the step size h, initial values of x and y, and interval [a, b].
  + For each step:
    - Four slopes k1, k2, k3, and k4 are calculated for the current x and y.
    - These slopes are combined to find the next value of y, according to the Runge-Kutta formula:

ynext = y + (1 / 6) \* (k1 + 2k2 + 2k3 + k4)

* + - Results are displayed for each step with updated x and y values.
* **Program Output:**
  + The program prints the values of x and y for each step, allowing users to observe the solution progression across the interval.

Finally, all the roots are showed in runtime.

**Algorithm-11: Matrix Inversion**

(matrix\_inversion.hpp)

The “MatrixInversion” is the class inherited from the class “Linear”,

implements the Gauss-Jordan method for inverting a square matrix.

**Key Components:**.

* **Augmenting with the Identity Matrix**
  + For each row i, the constructor appends an identity matrix column to the input matrix. This makes the augmented matrix [A | I], where A is the original matrix and I is the identity matrix.
* **Swapping Rows to Handle Zeros (swapp function)**
  + If a pivot element (mat[i][i]) is zero, the method searches the rows below it for a non-zero element in the same column and swaps rows to ensure a non-zero pivot.
* **Gaussian Elimination (gauss\_eli function)**
  + This function performs forward elimination, transforming the matrix into an upper triangular form by making all elements below each pivot zero.
* **Gauss-Jordan Elimination (gauss\_jordan function)**
  + This function performs backward elimination on the upper triangular matrix, ensuring that all elements above each pivot are zero, creating a diagonal matrix.
* **Row Reduction to Reduced Row Echelon Form (rref function)**
  + Each pivot row is normalized to make the diagonal elements 1. This final matrix is in the form [I | A⁻¹], with I on the left and the inverse A⁻¹ on the right.
* **Displaying the Inverse (showInverseMatrix function)**
  + This function prints the inverse matrix portion from the augmented matrix, which is in columns order to 2 \* order - 1.
* **Calculation (calculate function)**
  + This function orchestrates the steps: Gaussian elimination, Gauss-Jordan elimination, and RREF normalization, then displays the result