**%1.a. Matrix Multiplication**

sizeA = input("Enter the number of rows and columns in A:");

rows1 = sizeA(1);

cols1 = sizeA(2);

cols2 = input("Enter the number of columns in B:");

rows2 = cols1;

A=zeros(rows1,cols1);

B=zeros(rows2,cols2);

C=zeros(rows1,cols2);

disp('Enter the elements of matrix A:');

for i = 1:rows1

for j = 1:cols1

A(i, j) = input(" ");

end

end

disp('Enter the elements of matrix B:');

for i = 1:rows2

for j = 1:cols2

B(i, j) = input(" ");

end

end

for i = 1:rows1

for j = 1:cols2

sum = 0;

for k = 1:cols1

sum = sum + A(i, k) \* B(k, j);

end

C(i, j) = sum;

end

end

disp(C);

**% 1.b. Quadratic**

a = input('Enter the coefficient a: ');

b = input('Enter the coefficient b: ');

c = input('Enter the coefficient c: ');

D = b^2 - 4\*a\*c;

if D > 0

root1 = (-b + sqrt(D)) / (2\*a);

root2 = (-b - sqrt(D)) / (2\*a);

disp('The roots are:');

disp(root1);

disp(root2);

elseif D == 0

root = -b / (2\*a);

disp('The only root is:');

disp(root);

else

realPart = -b / (2\*a);

imagPart = sqrt(-D) / (2\*a);

disp('The roots are:');

disp([realPart + imagPart\*1i, realPart - imagPart\*1i]);

end

**% 2.a. Inverse using Gauss Jordan**

n = input('Enter dimension of matrix: ');

A = zeros(n, n);

disp('Enter the elements of matrix A:');

for i = 1:n

for j = 1:n

A(i, j) = input('');

end

end

I = eye(n);

A\_aug = [A I];

for i = 1:n

if A\_aug(i, i) == 0

%Swap with non-zero

for j = i+1:n

if A\_aug(j, i) ~= 0

temp = A\_aug(i, :);

A\_aug(i, :) = A\_aug(j, :);

A\_aug(j, :) = temp;

break;

end

end

end

% if pivot=0

if A\_aug(i, i) == 0

error('Matrix is singular and cannot be inverted');

end

pivot = A\_aug(i, i);

for k = 1:2\*n

A\_aug(i, k) = A\_aug(i, k) / pivot;

end

% Other elements 0

for j = 1:n

if j ~= i

factor = A\_aug(j, i);

for k = 1:2\*n

A\_aug(j, k) = A\_aug(j, k) - factor \* A\_aug(i, k);

end

end

end

end

% Inverse matrix

A\_inv = A\_aug(:, n+1:end);

disp('The inverse matrix is:');

disp(A\_inv);

**% 2.b. Gaussian Elimination using Partial Pivoting**

disp('Enter the coefficients of the three equations in the form:');

disp('a11 a12 a13 b1');

disp('a21 a22 a23 b2');

disp('a31 a32 a33 b3');

A = zeros(3, 4);

for i = 1:3

A(i, :) = input(sprintf('Enter coefficients and constant for equation %d: ', i));

end

coeff = A(:, 1:3);

const = A(:, 4);

A\_aug = [coeff const];

%Largest Index

n = 3;

for i = 1:n-1

[~, max\_row] = max(abs(A\_aug(i:n, i)));

max\_row = max\_row + i - 1;

%Swap

if max\_row ~= i

temp = A\_aug(i, :);

A\_aug(i, :) = A\_aug(max\_row, :);

A\_aug(max\_row, :) = temp;

end

%Pivot

for j = i+1:n

factor = A\_aug(j, i) / A\_aug(i, i);

A\_aug(j, i:end) = A\_aug(j, i:end) - factor \* A\_aug(i, i:end);

end

end

% Back substitution

x = zeros(n, 1);

for i = n:-1:1

x(i) = (A\_aug(i, end) - A\_aug(i, i+1:n) \* x(i+1:n)) / A\_aug(i, i);

end

disp('The solutions are:');

disp(['x = ', num2str(x(1))]);

disp(['y = ', num2str(x(2))]);

disp(['z = ', num2str(x(3))]);

**% 3.a. Gauss Siedel Iteration Method**

% Given system of equations

% 3x + y − 2z + w = 6

% x + 4y + z − w = 2

% 2x − y + 5z + 2w = 12

% x − y + z + 3w = 4

% Coefficients matrix A

A = [7 9 -3 8;

2 -6 5 -3;

6 -5 5 2;

11 8 -2 3];

% Right-hand side vector b

b = [5; 2; 1; 7];

% Initial guess

x = [0; 0; 0; 0];

% Tolerance and maximum number of iterations

tol = 0.001;

max\_iterations = 4;

% Gauss-Seidel Iteration

for k = 1:max\_iterations

x\_old = x;

% Update each variable x(i)

x(1) = (b(1) - (A(1,2) \* x(2) + A(1,3) \* x(3) + A(1,4) \* x(4))) / A(1,1);

x(2) = (b(2) - (A(2,1) \* x(1) + A(2,3) \* x(3) + A(2,4) \* x(4))) / A(2,2);

x(3) = (b(3) - (A(3,1) \* x(1) + A(3,2) \* x(2) + A(3,4) \* x(4))) / A(3,3);

x(4) = (b(4) - (A(4,1) \* x(1) + A(4,2) \* x(2) + A(4,3) \* x(3))) / A(4,4);

% Display current iteration and results

fprintf('Iteration %d: x = %.4f, y = %.4f, z = %.4f, w = %.4f\n', k, x(1), x(2), x(3), x(4));

% Check for convergence

if norm(x - x\_old, inf) < tol

fprintf('Converged in %d iterations.\n', k);

break;

end

end

% If maximum iterations are reached without convergence

if k == max\_iterations

fprintf('Reached maximum iterations without convergence.\n');

end

**% 3.b. Cholesky decomposition method**

% Given matrix A and vector b

A = [9 -7 4;

-2 5 2;

4 2 -1];

b = [7; 13; 17];

% Number of rows (or columns, since A is square)

n = size(A, 1);

% Initialize the lower triangular matrix L with zeros

L = zeros(n, n);

% Cholesky decomposition (without built-in function)

for i = 1:n

for j = 1:i

if i == j

L(i, j) = sqrt(A(i, i) - sum(L(i, 1:j-1).^2));

else

L(i, j) = (A(i, j) - sum(L(i, 1:j-1) .\* L(j, 1:j-1))) / L(j, j);

end

end

end

% Display the lower triangular matrix L

disp('Lower triangular matrix L:');

disp(L);

% Solve Ly = b using forward substitution

y = zeros(n, 1);

for i = 1:n

y(i) = (b(i) - L(i, 1:i-1) \* y(1:i-1)) / L(i, i);

end

% Solve L'x = y using backward substitution

x = zeros(n, 1);

for i = n:-1:1

x(i) = (y(i) - L(i+1:n, i)' \* x(i+1:n)) / L(i, i);

end

% Display the solution vector x

disp('Solution vector x:');

disp(x);

**% 4.a. Lagrange Interpolation**

% MATLAB Script to Read Data and Perform Lagrange Interpolation

% Function to read data from the file

function [roll\_numbers, heights] = readHeightData(filename)

% Open the file

fileID = fopen(filename, 'r');

% Read the data

data = fscanf(fileID, '%d %f', [2, Inf]);

% Close the file

fclose(fileID);

% Extract roll numbers and heights

roll\_numbers = data(1, :);

heights = data(2, :);

end

% Lagrange Interpolation function

function interpolated\_value = lagrangeInterpolation(roll\_numbers, heights, query\_roll)

n = length(roll\_numbers);

interpolated\_value = 0;

for i = 1:n

% Compute the Lagrange basis polynomial

L = 1;

for j = 1:n

if j ~= i

L = L \* (query\_roll - roll\_numbers(j)) / (roll\_numbers(i) - roll\_numbers(j));

end

end

% Accumulate the result

interpolated\_value = interpolated\_value + L \* heights(i);

end

end

% Main Script

% Specify the filename

filename = 'height\_data.txt';

% Read the data

[roll\_numbers, heights] = readHeightData(filename);

% Specify the roll number for which you want to interpolate the height

query\_roll = 104; % Example integral roll number

% Ensure that the query\_roll is an integer

if ~isinteger(query\_roll) && query\_roll ~= floor(query\_roll)

error('Roll number must be an integer.');

end

% Perform Lagrange interpolation

interpolated\_height = lagrangeInterpolation(roll\_numbers, heights, query\_roll);

% Display the interpolated height

fprintf('The interpolated height for roll number %d is %.2f cm.\n', query\_roll, interpolated\_height);

**% 4.b. Chebyshev Polynomial Interpolation**

% Degree of the polynomial

N = 4;

% Compute the Chebyshev nodes

i = 0:N;

x\_i = cos((2 .\* i + 1) \* pi / (2 \* (N + 1)));

% Evaluate the function at the Chebyshev nodes

y\_i = exp(x\_i);

% Initialize the coefficients

a\_k = zeros(1, N + 1);

% Calculate the coefficients of the interpolating Chebyshev polynomial

for k = 0:N

a\_k(k + 1) = (2 / (N + 1)) \* sum(y\_i .\* cos(k \* (2 .\* i + 1) \* pi / (2 \* (N + 1))));

end

a\_k(1) = a\_k(1) / 2; % Adjust the first coefficient

% Display the coefficients

disp('Coefficients of the Chebyshev polynomial:');

disp(a\_k);

% Define the Chebyshev polynomial using the calculated coefficients

P\_N = @(x) a\_k(1) \* ones(size(x));

for k = 1:N

P\_N = @(x) P\_N(x) + a\_k(k + 1) \* cos(k \* acos(x));

end

% Generate points for the original function and the Chebyshev polynomial

x\_vals = linspace(-1, 1, 100);

f\_vals = exp(x\_vals);

P\_vals = P\_N(x\_vals);

% Plot the original function and the interpolating polynomial

figure;

plot(x\_vals, f\_vals, 'r', 'LineWidth', 2);

hold on;

plot(x\_vals, P\_vals, 'b--', 'LineWidth', 2);

legend('e^x', 'Chebyshev Polynomial Interpolation');

title('Chebyshev Polynomial Interpolation of e^x');

xlabel('x');

ylabel('y');

grid on;

hold off;

**% 5.1.a: Bisection method**

function root = bisectionMethod(f, a, b, tol)

if f(a) \* f(b) > 0

error('f(a) and f(b) must have opposite signs');

end

while (b - a)/2 > tol

c = (a + b)/2;

if f(c) == 0

break;

elseif f(a) \* f(c) < 0

b = c;

else

a = c;

end

end

root = (a + b)/2;

end

% Usage

f = @(x) x^3 - 5\*x + 3;

a = -5;

b = 5;

tol = 1e-6;

root = bisectionMethod(f, a, b, tol);

fprintf('Root found using Bisection Method: %.6f\n', root);

**% 5.1.b Regula Falsi**

function root = regulaFalsi(f, a, b, tol)

if f(a) \* f(b) > 0

error('f(a) and f(b) must have opposite signs');

end

c = a;

while abs(f(c)) > tol

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

if f(c) == 0

break;

elseif f(a) \* f(c) < 0

b = c;

else

a = c;

end

end

root = c;

end

% Usage

f = @(x) x^2 - 4\*x + 2;

a = -2;

b = 2;

tol = 1e-6;

root = regulaFalsi(f, a, b, tol);

fprintf('Root found using Regula Falsi Method: %.6f\n', root);

**% 5.1.c: Newton Raphson for root evaluation**

function root = newtonRaphson(f, df, x0, tol)

x = x0;

while abs(f(x)) > tol

x = x - f(x) / df(x);

end

root = x;

end

% Usage

f = @(x) x^2 - 4\*x + 1;

df = @(x) 2\*x - 4;

x0 = 3;

tol = 1e-5;

root = newtonRaphson(f, df, x0, tol);

fprintf('Root found using Newton-Raphson Method: %.6f\n', root);

**% 5.2 Population Interpolation**

% Given data

years = [1951, 1961, 1971]; % Census years

population = [2.8, 3.2, 4.5]; % Population in million

% Year to interpolate

x\_interp = 1970;

% Lagrange Interpolation

n = length(years);

y\_interp = 0; % Initialize interpolated population

for i = 1:n

% Calculate Lagrange basis polynomial L\_i(x)

L = 1;

for j = 1:n

if i ~= j

L = L \* (x\_interp - years(j)) / (years(i) - years(j));

end

end

% Accumulate the result

y\_interp = y\_interp + L \* population(i);

end

% Display the interpolated population

fprintf('The interpolated population in 1966 is approximately %.2f million.\n', y\_interp);

**% 6.a. Newton Backward Difference**

% Given data points

x = [-0.75, -0.5, -0.25, 0];

f = [-0.0718125, -0.02475, 0.3349375, 1.10100];

% Calculate backward differences

n = length(x);

diff\_table = zeros(n, n);

diff\_table(:,1) = f';

for j = 2:n

for i = n:-1:j

diff\_table(i,j) = diff\_table(i,j-1) - diff\_table(i-1,j-1);

end

end

% Display the backward difference table

disp('Backward difference table:');

disp(diff\_table);

% Step size (assuming uniform step)

h = x(2) - x(1);

% Interpolation point

x\_interp = -1/3;

% Calculate u

u = (x\_interp - x(end)) / h;

% Newton's backward difference formula

f\_interp = diff\_table(n,1);

u\_term = 1;

for k = 1:n-1

u\_term = u\_term \* (u + (k-1));

f\_interp = f\_interp + (u\_term \* diff\_table(n,k+1)) / factorial(k);

end

% Display the interpolated value at x = -1/3

fprintf('The interpolated value f(-1/3) is: %.6f\n', f\_interp);

**% 6.b. Hermite Interpolation**

% Given data points and derivatives

x = [0.5, 1];

f = [4, 1];

f\_prime = [-16, -2];

% Initialize Hermite divided difference table

n = 2; % Two data points

Q = zeros(2\*n, 2\*n);

z = zeros(1, 2\*n);

% Fill in z values (duplicating x values)

for i = 1:n

z(2\*i-1) = x(i);

z(2\*i) = x(i);

Q(2\*i-1,1) = f(i);

Q(2\*i,1) = f(i);

Q(2\*i,2) = f\_prime(i);

if i ~= 1

Q(2\*i-1,2) = (Q(2\*i-1,1) - Q(2\*i-2,1)) / (z(2\*i-1) - z(2\*i-2));

end

end

% Fill the rest of the table

for i = 3:2\*n

for j = 3:i

Q(i,j) = (Q(i,j-1) - Q(i-1,j-1)) / (z(i) - z(i-j+1));

end

end

% Display the divided difference table

disp('Hermite Divided Difference Table:');

disp(Q);

% Evaluate the Hermite polynomial at a given value of X (example X = 0.75)

X = 0.75; % Change this value for different evaluations

% Construct the Hermite polynomial value at X using divided differences

H\_val = Q(1,1);

term = 1;

for i = 1:2\*n-1

term = term \* (X - z(i));

H\_val = H\_val + Q(i+1,i+1) \* term;

end

% Display the result

disp(['Hermite Interpolating Polynomial evaluated at X = ', num2str(X), ' is: ', num2str(H\_val)]);