Course Code : BCSL-058
Course Title : Computer oriented Numerical techniques Lab
Assignment Number : BCA(5)/L-058/Assignment/2024-25
Maximum Marks : 50
Weightage : 25% www.ignousite.com
Last Dates for Submission : 31st October, 2024 (For July 2024 Session)
: 30th April, 2025 (For January 2025 Session)

Q1. Write a program in C that accepts a decimal number and displays its floating-point equivalent number. You may make assumptions to simplify the program, however, your representation of floating-point number should be closer to IEEE 754 standard 32-bit representation.

Ans. A simplified version of converting a decimal number into its floating-point equivalent following the IEEE 754 standard in C, we can break it down into three main components: the sign bit, the exponent, and the mantissa (or significand).

Assumptions:

1. The program will only deal with positive numbers for simplicity, so the sign bit will always be 0.
2. We'll ignore special cases like NaN, infinity, or denormalized numbers.
3. We will approximate the floating-point conversion, not implement the exact IEEE 754 standard (which involves a lot of details like biasing, exact bit manipulation, etc.).

Simplified Steps:

1. Convert the decimal number to its binary equivalent.
2. Normalize the binary number.
3. Extract the exponent and mantissa.
4. Adjust the exponent using bias (127 for single-precision).
5. Combine sign, exponent, and mantissa to form the final floating-point representation.

Program:

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

// Function to convert decimal to binary and normalize it
void convertToBinary(float num, char *binary) {
  int intPart = (int)num;
  float fracPart = num - intPart;
  int idx = 0;

  // Convert integer part to binary
  while (intPart > 0) {
    binary[idx++] = (intPart % 2) + '0';
    intPart /= 2;
  }
```

```
    // Reverse the integer part binary string
    for (int i = 0; i < idx / 2; ++i) {
        char temp = binary[i];
        binary[i] = binary[idx - 1 - i];
        binary[idx - 1 - i] = temp;
    }

    binary[idx++] = '.';  // Add the decimal point

    // Convert fractional part to binary
    for (int i = 0; i < 23; ++i) { // Limiting to 23 bits for mantissa
        fracPart *= 2;
        if (fracPart >= 1) {
            binary[idx++] = '1';
            fracPart -= 1;
        } else {
            binary[idx++] = '0';
        }
    }
    binary[idx] = '\0';
}

// Function to get normalized binary representation
void normalizeBinary(char *binary, int *exponent, char *mantissa) {
    int dotPos = 0, firstOnePos = -1;
    while (binary[dotPos] != '.') {
        ++dotPos;
    }

    for (int i = 0; binary[i] != '\0'; ++i) {
        if (binary[i] == '1') {
            firstOnePos = i;
            break;
        }
    }

    if (firstOnePos == -1) { // If there's no '1' in the binary
        *exponent = 0;
        mantissa[0] = '\0';
        return;
    }

    *exponent = dotPos - firstOnePos - 1;
    int mantissaIndex = 0;
    for (int i = firstOnePos + 1; i < 24 && binary[i] != '\0'; ++i) {
        if (binary[i] != '.') {
            mantissa[mantissaIndex++] = binary[i];
        }
    }
    for (int i = mantissaIndex; i < 23; ++i) {
        mantissa[i] = '0';  // Fill remaining bits with 0
    }
    mantissa[23] = '\0';
}

int main() {
    float num;
    printf("Enter a decimal number: ");
    scanf("%f", &num);

    char binary[64] = {0};   // Buffer to store binary representation
    char mantissa[24] = {0}; // Buffer to store mantissa
    int exponent = 0;

    convertToBinary(num, binary);
    normalizeBinary(binary, &exponent, mantissa);

    // Adjust exponent using bias (127)
    exponent += 127;

    // Display results
    printf("Sign bit: 0\n");
    printf("Exponent: ");
    for (int i = 7; i >= 0; --i) {
        printf("%d", (exponent >> i) & 1);
    }
    printf("\nMantissa: %s\n", mantissa);

    return 0;
}
```

Output:

```
Enter a decimal number: 10.75
Sign bit: 0
Exponent: 10000010
Mantissa: 01011000000000000000000
```

Q2. Write a program in C to implement Gauss Seidel method for finding the roots of linear equations.

Ans. The Gauss-Seidel method is an iterative technique used to solve a system of linear equations of the form Ax=b. The method iteratively improves an initial guess until the solution converges to the actual solution within a specified tolerance.

Program:

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

#define MAX_ITER 100
#define TOLERANCE 0.0001

void gaussSeidel(int n, double a[n][n], double b[n], double x[n]) {
  double x_old[n];
  int iter = 0;

  while (iter < MAX_ITER) {
    for (int i = 0; i < n; i++) {
      x_old[i] = x[i];
    }

    for (int i = 0; i < n; i++) {
      double sum = 0.0;
      for (int j = 0; j < n; j++) {
        if (i != j) {
          sum += a[i][j] * x[j];
        }
      }
      x[i] = (b[i] - sum) / a[i][i];
    }

    // Check for convergence
    int converged = 1;
    for (int i = 0; i < n; i++) {
      if (fabs(x[i] - x_old[i]) > TOLERANCE) {
        converged = 0;
        break;
      }
    }
    if (converged) {
      break;
    }

    iter++;
  }
  if (iter == MAX_ITER) {
    printf("Solution did not converge within %d iterations.\n", MAX_ITER);
  }
else {
    printf("Solution converged in %d iterations.\n", iter);
  }

}

int main() {
  int n;
  printf("Enter the number of equations: ");
  scanf("%d", &n);

  double a[n][n], b[n], x[n];

  printf("Enter the coefficients of the matrix A:\n");
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      scanf("%lf", &a[i][j]);
    }

  }

  printf("Enter the constants vector B:\n");
  for (int i = 0; i < n; i++) {
    scanf("%lf", &b[i]);
  }

  printf("Enter the initial guess vector X:\n");
  for (int i = 0; i < n; i++) {

   scanf("%lf", &x[i]);
  }

  gaussSeidel(n, a, b, x);

  printf("The solution is:\n");
  for (int i = 0; i < n; i++) {
    printf("x[%d] = %.4lf\n", i, x[i]);
  }

  return 0;
}
```

Output:

```
Enter the number of equations: 3
Enter the coefficients of the matrix A:
3 2 1
2 3 1
1 1 2
Enter the constants vector B:
1
2
0
Enter the initial guess vector X:
0
0
0
Solution converged in 14 iterations.
The solution is:
x[0] = 0.0909
x[1] = 0.6364
x[2] = -0.3636
```

Q3. Write a program in C to implement Bisection method for finding a positive root of the equation $X^2 - 9x + 21 = 0$. You have to make suitable choice for the bounds.

Ans. The Bisection method is a numerical technique used to find roots of a continuous function. It works by repeatedly narrowing the interval within which the root lies until the interval is sufficiently small.

Program:

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

// Function to calculate f(x) = x² - 9x + 21
double f(double x) {
   return x * x - 9 * x + 21;
}

// Bisection method function
void bisection(double a, double b, double tolerance) {
   if (f(a) * f(b) >= 0) {
      printf("Incorrect initial guesses.\n");
      return;
   }

   double c;
   int iter = 0;
```

```
    while ((b - a) >= tolerance) {
        // Calculate midpoint
        c = (a + b) / 2;

        // Check if the midpoint is a root
        if (fabs(f(c)) <= tolerance) {
            break;
        }

        // Decide the side to repeat the steps
        if (f(c) * f(a) < 0) {
            b = c;
        } else {
            a = c;
        }
        iter++;
    }

    printf("The root is: %.6lf\n", c);
    printf("Number of iterations: %d\n", iter);
}

int main() {
    double a, b, tolerance;

    // Choose appropriate bounds
    a = 3;  // Initial guess for the lower bound
    b = 5;  // Initial guess for the upper bound

    // Define tolerance level
    tolerance = 0.0001;

    bisection(a, b, tolerance);

    return 0;
}
```

Output:

```
The root is: 3.898979
Number of iterations: 12
```

Q4. Write a program in C for the demonstration of Newton's Backward Interpolation Formula.

Ans. Newton's Backward Interpolation is a numerical method used to estimate the value of a function for a given value of the independent variable when the values of the function at a set of equidistant points are known. It's particularly useful when the point at which the function is to be estimated is closer to the end of the tabulated values.

Program:

```
#include <stdio.h>

// Function to calculate factorial of a number

int factorial(int n) {

  int fact = 1;

  for (int i = 2; i <= n; i++) {

    fact *= i;

  }

  return fact;

}


// Function to perform backward differences

void calculateBackwardDifferences(int n, float y[n][n]) {

  for (int i = 1; i < n; i++) {

    for (int j = n - 1; j >= i; j--) {

      y[j][i] = y[j][i - 1] - y[j - 1][i - 1];

    }

  }

}


// Function to perform Newton's Backward Interpolation

float newtonBackwardInterpolation(int n, float x[], float y[n][n], float value) {

  float result = y[n - 1][0];

  float u = (value - x[n - 1]) / (x[1] - x[0]);

  float u_term = 1;
```

```
    for (int i = 1; i < n; i++) {
        u_term *= (u + i - 1);
        result += (u_term * y[n - 1][i]) / factorial(i);
    }

    return result;
}

int main() {
    int n;
    printf("Enter the number of data points: ");
    scanf("%d", &n);

    float x[n], y[n][n];

    printf("Enter the data points (x and y):\n");
    for (int i = 0; i < n; i++) {
        printf("x[%d] = ", i);
        scanf("%f", &x[i]);
        printf("y[%d] = ", i);
        scanf("%f", &y[i][0]);
    }

    // Calculate backward differences
    calculateBackwardDifferences(n, y);

    float value;
    printf("Enter the value of x to find the corresponding y: ");

    scanf("%f", &value);


    // Perform Newton's Backward Interpolation

    float result = newtonBackwardInterpolation(n, x, y, value);


    printf("The interpolated value at x = %.4f is y = %.4f\n", value, result);


    return 0;

}
```

Output:

```
Enter the number of data points: 4
Enter the data points (x and y):
x[0] = 1
y[0] = 1
x[1] = 2
y[1] = 4
x[2] = 3
y[2] = 9
x[3] = 4
y[3] = 16
Enter the value of x to find the corresponding y: 2.5

The interpolated value at x = 2.5000 is y = 6.2500
```

Q5. Write program in C for the demonstration of Bessel's Formula.

Ans. Bessel's Interpolation Formula is used for interpolating values of a function that is given at equidistant points, particularly when the point where interpolation is required lies near the middle of the given data points. It is especially useful for symmetric data points around the interpolation point.

Program:

```
#include <stdio.h>

#include <math.h>

// Function to calculate factorial of a number

int factorial(int n) {
    int fact = 1;

    for (int i = 2; i <= n; i++) {

        fact *= i;

    }

    return fact;

}

// Function to perform forward differences

void calculateForwardDifferences(int n, float y[n][n]) {

    for (int i = 1; i < n; i++) {

        for (int j = 0; j < n - i; j++) {

            y[j][i] = y[j + 1][i - 1] - y[j][i - 1];

        }

    }

}

// Function to perform Bessel's Interpolation

float besselInterpolation(int n, float x[], float y[n][n], float value) {

    int mid = (n - 1) / 2;

    float h = x[1] - x[0];

    float u = (value - x[mid]) / h;

    float result = y[mid][0];

    result += u * (y[mid][1] + y[mid - 1][1]) / 2.0;

    float u_sq_minus_1 = (u * u) - 1;

    result += (u_sq_minus_1 * y[mid - 1][2]) / (2 * factorial(2));

    float u_sq_minus_4 = u_sq_minus_1 * (u * u - 4);

    result += (u_sq_minus_4 * (y[mid - 2][3] + y[mid - 1][3])) / (8 * factorial(3));

    return result;

}
```

```
int main() {
  int n;
  printf("Enter the number of data points (odd number): ");
  scanf("%d", &n);

  if (n % 2 == 0) {
    printf("Bessel's formula requires an odd number of data points.\n");
    return 1;
  }
  float x[n], y[n][n];

  printf("Enter the data points (x and y):\n");
  for (int i = 0; i < n; i++) {
    printf("x[%d] = ", i);
    scanf("%f", &x[i]);
    printf("y[%d] = ", i);
    scanf("%f", &y[i][0]);
  }

  // Calculate forward differences
  calculateForwardDifferences(n, y);

  float value;
  printf("Enter the value of x to find the corresponding y: ");
  scanf("%f", &value);

  // Perform Bessel's Interpolation
  float result = besselInterpolation(n, x, y, value);
```

```
    printf("The interpolated value at x = %.4f is y = %.4f\n", value, result);

    return 0;

}
```

Output:

```
Enter the number of data points (odd number): 5
Enter the data points (x and y):
x[0] = 1
y[0] = 1.1
x[1] = 2
y[1] = 1.4
x[2] = 3
y[2] = 1.8
x[3] = 4
y[3] = 2.3
x[4] = 5
y[4] = 2.7
Enter the value of x to find the corresponding y: 2.5
The interpolated value at x = 2.5000 is y = 1.6214
```

Q6. Write a program in C to demonstrate the Newton's Divided Difference Method.

Ans. Newton's Divided Difference Method is a numerical technique used for polynomial interpolation. It provides a way to compute the coefficients of the interpolating polynomial using divided differences, which are based on the values of the function at given points.

Program:

```
#include <stdio.h>

// Function to calculate the divided differences table

void dividedDifferenceTable(int n, float x[], float y[n][n]) {

    for (int j = 1; j < n; j++) {

        for (int i = 0; i < n - j; i++) {

            y[i][j] = (y[i + 1][j - 1] - y[i][j - 1]) / (x[i + j] - x[i]);

        }

    }

}
```

```
// Function to apply Newton's Divided Difference formula

float applyNewtonFormula(int n, float x[], float y[n][n], float value) {

   float result = y[0][0];

   float term;

   for (int i = 1; i < n; i++) {

      term = y[0][i];

      for (int j = 0; j < i; j++) {

         term *= (value - x[j]);

      }

      result += term;

   }

   return result;

}

int main() {

   int n;

   printf("Enter the number of data points: ");

   scanf("%d", &n);

   float x[n], y[n][n];

   printf("Enter the data points (x and y):\n");

   for (int i = 0; i < n; i++) {

      printf("x[%d] = ", i);

      scanf("%f", &x[i]);

      printf("y[%d] = ", i);

      scanf("%f", &y[i][0]);

   }

   // Calculate the divided difference table

   dividedDifferenceTable(n, x, y);
```

```
    float value;

    printf("Enter the value of x to find the corresponding y: ");

    scanf("%f", &value);

    // Apply Newton's Divided Difference formula

    float result = applyNewtonFormula(n, x, y, value);

    printf("The interpolated value at x = %.4f is y = %.4f\n", value, result);

    return 0;

}
```

Output:

```
Enter the number of data points: 4
Enter the data points (x and y):
x[0] = 1
y[0] = 1
x[1] = 2
y[1] = 4
x[2] = 3
y[2] = 9
x[3] = 4
y[3] = 16
Enter the value of x to find the corresponding y: 2.5
The interpolated value at x = 2.5000 is y = 6.2500
```

Q7. Write a program in C to find the approximate value of the following definite integral using Simpson's 1/3 rule:

$$\int_0^{\frac{\pi}{4}} \tan x \, dx$$

Ans.

Program:

```
#include <stdio.h>

#include <math.h>

// Function to calculate f(x) = tan(x)

double f(double x) {

    return tan(x);
}

// Function to perform Simpson's 1/3 Rule

double simpsonsRule(double a, double b, int n) {

  double h = (b - a) / n;

  double sum = f(a) + f(b);

    for (int i = 1; i < n; i++) {

    double x = a + i * h;

    if (i % 2 == 0) {

      sum += 2 * f(x);

    } else {

      sum += 4 * f(x);

    }

  }

  return (h / 3) * sum;

}

int main() {

  double a = 0.0;       // Lower limit

  double b = M_PI / 4.0;  // Upper limit (π/4)

  int n = 6;          // Number of subintervals (must be even)

  if (n % 2 != 0) {

    printf("Number of subintervals (n) must be even.\n");

    return 1;

  }

  double result = simpsonsRule(a, b, n);

  printf("The approximate value of the integral is: %.6f\n", result);

  return 0;

}
```

Output:

```
The approximate value of the integral is: 0.346574
```

Q8. Write a C program to implement Euler's rule/method, of approximating solution of the i.v.p.:*y'(x)*=(*dy/dx*)= *f(x,y) with initial condition at x=a as y(a)*= y0*over an interval [a,b]*.

Ans. Euler's method is a first-order numerical procedure for solving ordinary differential equations (ODEs) with a given initial value. It is one of the simplest methods for solving initial value problems (IVPs) numerically.

Program:

```
#include <stdio.h>

// Define the function f(x, y) = dy/dx

double f(double x, double y) {

   return (x + y); // Example: dy/dx = x + y

}

// Function to implement Euler's method

void eulerMethod(double a, double b, double y0, int n) {

   double h = (b - a) / n; // Calculate step size

   double x = a;        // Initialize x to the initial condition

   double y = y0;        // Initialize y to the initial value


   printf("x0 = %.4f, y0 = %.4f\n", x, y);


   // Perform the Euler method iteration

   for (int i = 0; i < n; i++) {

      y = y + h * f(x, y);

      x = x + h;

      printf("x%d = %.4f, y%d = %.4f\n", i+1, x, i+1, y);

   }

}
```

```
int main() {
    double a = 0.0; // Initial value of x
    double b = 1.0; // End value of x
    double y0 = 1.0; // Initial value of y
    int n = 10; // Number of steps
    printf("Euler's Method:\n");
    eulerMethod(a, b, y0, n);
    return 0;
}
```

Output:

```
Euler's Method:
x0 = 0.0000, y0 = 1.0000
x1 = 0.1000, y1 = 1.1000
x2 = 0.2000, y2 = 1.2200
x3 = 0.3000, y3 = 1.3620
x4 = 0.4000, y4 = 1.5282
x5 = 0.5000, y5 = 1.7210
x6 = 0.6000, y6 = 1.9431
x7 = 0.7000, y7 = 2.1974
x8 = 0.8000, y8 = 2.4871
x9 = 0.9000, y9 = 2.8158
x10 = 1.0000, y10 = 3.1874
```