LAB 2
Experiment No: 1

TITLE: Lagrange's Interpolating Polynomials.

OBJECTIVES

To implement Lagrange's Interpolating Polynomials to approximate the value of a function at a given point in C programming.

THEORY

Lagrange's Interpolating Polynomials are used to find a polynomial that passes through a given set of data points. The polynomial is constructed as a linear combination of basis polynomials, each of which is zero at all given points except one.

Algorithm Steps:

1. Input:
   - Number of data points n.
   - Arrays x and y of size n containing the data points.
   - The point xi at which to interpolate.
2. Initialize:
   - Set result=0.
3. Compute Lagrange Polynomial:
   - For each i*i* from 0 to n−1:
     - Set term=y[i]
     - For each j from 0 to n−1:
       - If j ≠ i, multiply term by (xi − x[j]) / (x[i]−x[j])
     - Add term to result.
4. Output:
   - The interpolated value result at xi.

Advantages:

- Simple to implement.
- Does not require equally spaced data points.

Limitations:

- Computationally expensive for large datasets.
- Susceptible to Runge's phenomenon for high-degree polynomials.

Demostration

```
#include <stdio.h>
// Function to calculate Lagrange's interpolating polynomial
double lagrangeInterpolation(double x[], double y[], int n, double xi)
{
  double result = 0.0;
for (int i = 0; i < n; i++)
{
  double term = y[i]; // Initialize term with y[i]
  for (int j = 0; j < n; j++)
  {
    if (j != i)
    {
```

```
        term *= (xi - x[j]) / (x[i] - x[j]); // Multiply by (x - x_j) / (x_i - x_j)
      }
    }
    result += term; // Add the term to the result
  }

  return result;
}

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

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

  double xi;
  printf("Enter the point at which to interpolate (xi): ");
  scanf("%lf", &xi);

  // Perform interpolation
  double yi = lagrangeInterpolation(x, y, n, xi);

  // Print the result
  printf("Interpolated value at x = %.2f is y = %.6f\n", xi, yi);

  return 0;
}
```

Output 1:

```
Enter the number of data points: 4
Enter the data points (x):
x[0]: 1
x[1]: 3
x[2]: 5
x[3]: 7
Enter the data points (y):
y[0]: 3
y[1]: 6
y[2]: 8
y[3]: 9
Enter the point at which to interpolate (xi): 4
Interpolated value at x = 4.00 is y = 7.125000
```

Output 2:

```
Enter the number of data points: 3
Enter the data points (x):
x[0]: 5
x[1]: 8
x[2]: 12
Enter the data points (y):
y[0]: 20
y[1]: 40
y[2]: 70
Enter the point at which to interpolate (xi): 15.34
Interpolated value at x = 15.34 is y = 97.968524
```

RESULT AND DISCUSSION

The program implements Lagrange's Interpolating Polynomials to approximate the value of a function at a user-specified point xi. The user provides the number of data points n, the arrays x and y, and the point xi at which to interpolate.

The program calculates

- the Interpolated value at x = 4.00 is y = 7.125000
- Interpolated value at x = 15.34 is y = 97.968524

CONCLUSION

Lagrange's Interpolating Polynomials provide a simple and effective way to approximate the value of a function at a given point using a set of data points. While the method is easy to implement and does not require equally spaced data, it is not suitable for large datasets due to its computational complexity. This experiment demonstrates the practical implementation of Lagrange's Interpolation in C programming and validates its effectiveness for interpolation tasks.

Experiment No: 2

TITLE: Newton's divided difference.

OBJECTIVES

To implement **Newton's Divided Difference Interpolation** to approximate the value of a function at a given point in C programming.

THEORY

Newton's Divided Difference Interpolation is a method used to construct an interpolating polynomial for a given set of data points. It is based on the concept of divided differences, which are used to compute the coefficients of the polynomial.

Given n+1 data points (x0,y0),(x1,y1),…,(xn,yn) the Newton's interpolating polynomial P(x) is given by:

$$P(x)=f[x0]+(x-x0)f[x0,x1]+(x-x0)(x-x1)f[x0,x1,x2]+\cdots+(x-x0)(x-x1)\ldots(x-xn-1)\,f[x0,x1,\ldots,xn]$$

where f[x0,x1,…,xk] are the divided differences, computed recursively as:

f[xi]=yi

f[xi,xi+1,…,xi+k] = (f[xi+1,…,xi+k] − f[xi,…,xi+k−1]) / (xi+k−xi)

Algorithm Steps:

1. **Input**:
   - Number of data points n*n*.
   - Arrays x*x* and y of size n containing the data points.
   - The point xi at which to interpolate.
2. **Compute Divided Differences**:
   - Initialize a 2D array dd to store divided differences.
   - Set dd[i][0]=y[i] for all i.
   - For each k from 1 to n−1:
     - For each i*i* from 0 to n−k−1:
       - Compute dd[i][k] = (dd[i+1][k−1]−dd[i][k−1]) / (x[i+k]−x[i])
3. **Compute Interpolated Value**:
   - Set result=dd[0][0].
   - For each i from 1 to n−1:
     - Multiply result by (xi−x[i−1]) and add dd[0][i].
4. **Output**:
   - The interpolated value result at xi.

Advantages:

- Efficient for adding new data points.
- Computationally less expensive than Lagrange's method.

Limitations:

- Requires the data points to be distinct.

Demostration

```
#include <stdio.h>

// Function to compute Newton's divided difference interpolation
double newtonDividedDifference(double x[], double y[], int n, double xi)
{
    double dd[n][n]; // Divided difference table

    // Initialize the divided difference table
    for (int i = 0; i < n; i++)
    {
        dd[i][0] = y[i];
    }

    // Compute divided differences
    for (int k = 1; k < n; k++)
    {
        for (int i = 0; i < n - k; i++)
        {
            dd[i][k] = (dd[i + 1][k - 1] - dd[i][k - 1]) / (x[i + k] - x[i]);
        }
    }

    // Compute the interpolated value
    double result = dd[0][0];
    double term = 1.0;
    for (int i = 1; i < n; i++)
    {
        term *= (xi - x[i - 1]);
        result += dd[0][i] * term;
    }

    return result;
}

// Function to input data points
void inputData(double x[], double y[], int n)
{
    printf("Enter the x data points:\n");
    for (int i = 0; i < n; i++)
    {
        printf("x[%d]: ", i);
        scanf("%lf", &x[i]);
    }
    printf("Enter the y data points:\n");
    for (int i = 0; i < n; i++)
    {
        printf("y[%d]: ", i);
        scanf("%lf", &y[i]);
    }
}

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

    if (n <= 0)
    {
        printf("Error: Number of data points must be greater than 0.\n");
        return 1;
    }

    double x[n], y[n];
    inputData(x, y, n);

    double xi;
    printf("Enter the point at which to interpolate (xi): ");
    scanf("%lf", &xi);

    // Perform interpolation
    double yi = newtonDividedDifference(x, y, n, xi);

    // Print the result
    printf("Interpolated value at x = %.2f is y = %.6f\n", xi, yi);

    return 0;
}
```

Output 1:

```
Enter the number of data points: 3
Enter the x data points:
x[0]: 1
x[1]: 5
x[2]: 9
Enter the y data points:
y[0]: 5
y[1]: 10
y[2]: 20
Enter the point at which to interpolate (xi): 3
Interpolated value at x = 3.00 is y = 6.875000
```

Output 2:

```
Enter the number of data points: 3
Enter the x data points:
x[0]: 1
x[1]: 5
x[2]: 9
Enter the y data points:
y[0]: 5
y[1]: 10
y[2]: 20
Enter the point at which to interpolate (xi): 7
Interpolated value at x = 7.00 is y = 14.375000
```

RESULT AND DISCUSSION

The program implements Newton's Divided Difference Interpolation to approximate the value of a function at a user-specified point xi. The user provides the number of data points nn, the arrays x and y, and the point xi at which to interpolate.

The program calculates the interpolated value

- at x = 3.00 is y = 6.875000
- at x = 7.00 is y = 14.375000

CONCLUSION

Newton's Divided Difference Interpolation provides an efficient way to approximate the value of a function at a given point using a set of data points. It is computationally less expensive than Lagrange's method and is particularly useful when adding new data points. This experiment demonstrates the practical implementation of Newton's Divided Difference Interpolation in C programming and validates its effectiveness for interpolation tasks.

Experiment No: 3

TITLE: Newton's forward difference

OBJECTIVES

To implement Newton's Forward Difference Interpolation to approximate the value of a function at a given point in C programming.

THEORY

Newton's Forward Difference Interpolation is a method used to construct an interpolating polynomial for a given set of equally spaced data points. It is based on the concept of forward differences, which are used to compute the coefficients of the polynomial.

Given n+1 equally spaced data points (x0,y0),(x1,y1),…,(xn,yn), the Newton's forward difference polynomial P(x) is given by:

$$P(x) = y_0 + \frac{u}{1!}\Delta y_0 + \frac{u(u-1)}{2!}\Delta^2 y_0 + \dots + \frac{u(u-1)\dots(u-n+1)}{n!}\Delta^n y_0$$

where:

- $u = \frac{x - x_0}{h}$ (normalized value),
- hh is the spacing between consecutive xx values,
- $\Delta y0, \Delta^2 y0, \dots, \Delta^n y0$ are the forward differences.

Forward Differences:

- First forward difference: $\Delta y_i = y_{i+1} - y_i$
- Second forward difference: $\Delta^2 y_i = \Delta y_{i+1} - \Delta y_i$
- And so on...

Algorithm Steps:

1. **Input**:
   - Number of data points nn.
   - Arrays x and y of size n containing the data points.
   - The point xi at which to interpolate.
2. **Compute Forward Differences**:
   - Initialize a 2D array fd to store forward differences.
   - Compute the first forward differences: $fd_{[i][0]} = y_{[i+1]} - y_{[i]}$.
   - Compute higher-order forward differences recursively.
3. **Compute Interpolated Value**:
   - Calculate u= (xi−x[0])/h .
   - Use the forward difference formula to compute the interpolated value.

4. **Output**:
    - The interpolated value at xi.

Advantages:

- Efficient for equally spaced data points.
- Simple to implement.

Limitations:

- Requires equally spaced data points.
- Not suitable for non-uniformly spaced data.

DEMOSTRATION

```
#include <stdio.h>

// Function to compute Newton's forward difference interpolation
double newtonForwardDifference(double x[], double y[], int n, double xi) {
    double h = x[1] - x[0]; // Spacing between x values
    double u = (xi - x[0]) / h; // Normalized value

    // Create a forward difference table
    double fd[n][n];
    for (int i = 0; i < n; i++) {
        fd[i][0] = y[i];
    }

    // Compute forward differences
    for (int k = 1; k < n; k++) {
        for (int i = 0; i < n - k; i++) {
            fd[i][k] = fd[i + 1][k - 1] - fd[i][k - 1];
        }
    }

    // Compute the interpolated value
    double result = fd[0][0];
    double term = 1.0;
    for (int i = 1; i < n; i++) {
        term *= (u - (i - 1)) / i;
        result += term * fd[0][i];
    }

    return result;
}

// Function to input data points
void inputData(double x[], double y[], int n) {
    printf("Enter the x values:\n");
    for (int i = 0; i < n; i++) {
        printf("x[%d]: ", i);
  return 0;
        scanf("%lf", &x[i]);
    }

    printf("Enter the y values:\n");
    for (int i = 0; i < n; i++) {
        printf("y[%d]: ", i);
        scanf("%lf", &y[i]);
    }
}

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

    if (n <= 0) {
            printf("Error: Number of data points must be greater than 0.\n");
        return 1;
    }

    double x[n], y[n];
    inputData(x, y, n);

    double xi;
    printf("Enter the point at which to interpolate (xi): ");
    scanf("%lf", &xi);

    // Perform interpolation
    double yi = newtonForwardDifference(x, y, n, xi);

    // Print the result
    printf("Interpolated value at x = %.2f is y = %.6f\n", xi, yi);
    return 0;
}
```

Output 1:

```
Enter the number of data points: 2
Enter the x values:
x[0]: 1
x[1]: 3
Enter the y values:
y[0]: 4
y[1]: 5
Enter the point at which to interpolate (xi): 2
Interpolated value at x = 2.00 is y = 4.500000
```

Output 2:

```
Enter the number of data points: 2
Enter the x values:
x[0]: 1
x[1]: 3
Enter the y values:
y[0]: 4
y[1]: 5
Enter the point at which to interpolate (xi): 1.45
Interpolated value at x = 1.45 is y = 4.225000
```

RESULT AND DISCUSSION

The program implements Newton's Forward Difference Interpolation to approximate the value of a function at a user-specified point xi*xi*. The user provides the number of data points n*n*, the arrays x*x* and y*y*, and the point xi*xi* at which to interpolate.

The program calculates the interpolated value

- at x = 2.00 is y = 4.500000
- at x = 1.45 is y = 4.225000

CONCLUSION

Newton's Forward Difference Interpolation provides an efficient way to approximate the value of a function at a given point using a set of equally spaced data points. It is simple to implement and computationally efficient. This experiment demonstrates the practical implementation of Newton's Forward Difference Interpolation in C programming and validates its effectiveness for interpolation tasks.

Experiment No: 4

TITLE: Newton's backward difference.

OBJECTIVES

To implement **Newton's Backward Difference Interpolation** to approximate the value of a function at a given point in C programming.

THEORY

Newton's Backward Difference Interpolation is a method used to construct an interpolating polynomial for a given set of equally spaced data points. It is based on the concept of backward differences, which are used to compute the coefficients of the polynomial.

Given n+1 equally spaced data points (x0,y0),(x1,y1),…,(xn,yn) the Newton's backward difference polynomial P(x) is given by:

$$P(x) = y_n + \frac{u}{1!} \nabla y_n + \frac{u(u+1)}{2!} \nabla^2 y_n + \ldots + \frac{u(u+1)\ldots(u+n-1)}{n!} \nabla^n y_n \| 1$$

where:

- $u = \frac{x-x_n}{h}$ (normalized v alue),
- hh is the spacing between consecutive xx values,
- ∇yn,∇2yn,…,∇nyn are the backward differences.

Backward Differences:

- First backward difference: $\nabla yi = yi - yi - 1\nabla yi \quad = yi \quad - yi - 1$
- Second backward difference: $\nabla 2yi = \nabla yi - \nabla yi - 1$
- And so on...

Algorithm Steps:

1. **Input**:
   - Number of data points n.
   - Arrays xx and y of size n containing the data points.
   - The point xixi at which to interpolate.
2. **Compute Backward Differences**:
   - Initialize a 2D array bd to store backward differences.
   - Compute the first backward differences: $bd[i][0] = y[i] - y[i-1]$
   - Compute higher-order backward differences recursively.
3. **Compute Interpolated Value**:
   - Calculate $u = xi - xn$
   - Use the backward difference formula to compute the interpolated value.
4. **Output**:
   - The interpolated value at xixi.

Advantages:

- Efficient for equally spaced data points.
- Simple to implement.

Limitations:

- Requires equally spaced data points.
- Not suitable for non-uniformly spaced data.

DEMOSTRATION

```
#include <stdio.h>

// Function to compute Newton's forward difference interpolation
double newtonForwardDifference(double x[], double y[], int n, double xi) {
    double h = x[1] - x[0]; // Spacing between x values
    double u = (xi - x[0]) / h; // Normalized value

    // Create a forward difference table
    double fd[n][n];
    for (int i = 0; i < n; i++) {
        fd[i][0] = y[i];
    }

    // Compute forward differences
    for (int k = 1; k < n; k++) {
        for (int i = 0; i < n - k; i++) {
            fd[i][k] = fd[i + 1][k - 1] - fd[i][k - 1];
        }
    }

    // Compute the interpolated value
    double result = fd[0][0];
    double term = 1.0;
    for (int i = 1; i < n; i++) {
        term *= (u - (i - 1)) / i;
        result += term * fd[0][i];
    }

    return result;
}

// Function to input data points
void inputData(double x[], double y[], int n) {
    printf("Enter the x values:\n");
    for (int i = 0; i < n; i++) {
        printf("x[%d]: ", i);
        scanf("%lf", &x[i]);
    }

    printf("Enter the y values:\n");
    for (int i = 0; i < n; i++) {
        printf("y[%d]: ", i);
        scanf("%lf", &y[i]);
    }
}

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

    if (n <= 0) {
            printf("Error: Number of data points must be greater than 0.\n");
        return 1;
    }

    double x[n], y[n];
    inputData(x, y, n);

    double xi;
    printf("Enter the point at which to interpolate (xi): ");
    scanf("%lf", &xi);

    // Perform interpolation
    double yi = newtonForwardDifference(x, y, n, xi);

    // Print the result
    printf("Interpolated value at x = %.2f is y = %.6f\n", xi, yi);

    return 0;
}
```

Output 1:

```
{ .\newtonbackwardDiff }
Enter the number of data points: 4
Enter the x values:
x[0]: 1
x[1]: 2
x[2]: 4
x[3]: 5
Enter the y values:
y[0]: 56
y[1]: 78
y[2]: 99
y[3]: 143
Enter the point at which to interpolate (xi): 3
Interpolated value at x = 3.00 is y = 78.000000
PS F:\s3 csit\NumericalMethod-main\LabCodes>
```

Output 2:

```
{ .\newtonbackwardDiff }
Enter the number of data points: 4
Enter the x values:
x[0]: 1
x[1]: 2
x[2]: 4
x[3]: 5
Enter the y values:
y[0]: 56
y[1]: 78
y[2]: 99
y[3]: 143
Enter the point at which to interpolate (xi): 4.67
Interpolated value at x = 4.67 is y = 124.460402
PS F:\s3 csit\NumericalMethod-main\LabCodes>
```

RESULT AND DISCUSSION

The program implements Newton's Backward Difference Interpolation to approximate the value of a function at a user-specified point xi*xi*. The user provides the number of data points n*n*, the arrays x*x* and y*y*, and the point xi*xi* at which to interpolate.

The program calculates the interpolated value

- at x = 3.00 is y = 78.000000
- at x = 4.67 is y = 124.460402

CONCLUSION

Newton's Backward Difference Interpolation provides an efficient way to approximate the value of a function at a given point using a set of equally spaced data points. It is simple to implement and computationally efficient. This experiment demonstrates the practical implementation of Newton's Backward Difference Interpolation in C programming and validates its effectiveness for interpolation tasks.

Experiment No: 5

TITLE: OLS method to fit a straight line.

OBJECTIVES

To implement the **Ordinary Least Squares (OLS) method** for fitting a straight line to a given dataset in C programming.

THEORY

The Ordinary Least Squares (OLS) method is a statistical approach used to determine the best-fitting line for a given set of data points. The best-fitting line minimizes the sum of squared residuals, where a residual is the difference between an observed value and the predicted value.

The equation of a straight line is given by:

$$[\,y = mx + c\,]$$

where:

- Y is the dependent variable,
- X is the independent variable,
- m is the slope of the line, and
- c is the y-intercept.

Using the **OLS method**, the best-fitting values of and are calculated using the formulas:

$$m = \frac{n\sum xy - \sum x \sum y}{n\sum x^2 - (\sum x)^2} \qquad c = \frac{\sum y - m\sum x}{n}$$

where:

- n is the number of data points,
- $\sum x$ and $\sum y$ are the summations of the given and values,
- $\sum xy$ is the summation of the product of corresponding and values,
- $\sum x2$ is the summation of squared values.

Algorithm Steps:

1. Input:
   - Number of data points .
   - Arrays and containing the data points.
2. Compute Required Summations:
   - Compute each summations and .
   - Calculate Slope and Intercept using the OLS formulas.
   - Output the Equation of the Best-Fitting Line.

Advantages:

1. Provides the best linear fit by minimizing error.
2. Computationally simple and efficient.

Limitations:

1. Assumes a linear relationship between variables.
2. Sensitive to outliers.

DEMOSTRATION

```c
#include <stdio.h>

void leastSquaresFit(double x[], double y[], int n, double *m, double *c) {
   double sumX = 0, sumY = 0, sumXY = 0, sumX2 = 0;

   for (int i = 0; i < n; i++) {
      sumX += x[i];
      sumY += y[i];
      sumXY += x[i] * y[i];
      sumX2 += x[i] * x[i];
   }

     *m = (n * sumXY - sumX * sumY) / (n * sumX2 - sumX * sumX);
    *c = (sumY - (*m) * sumX) / n;
}

int main() {
   int n;
   printf("Enter the number of data points: ");
   scanf("%d", &n);
   double x[n], y[n];
   printf("Enter the x values:\n");
   for (int i = 0; i < n; i++) {
      printf("x[%d]: ", i);
      scanf("%lf", &x[i]);
   }

   printf("Enter the y values:\n");
   for (int i = 0; i < n; i++) {
      printf("y[%d]: ", i);
      scanf("%lf", &y[i]);
   }

   double m, c;
   leastSquaresFit(x, y, n, &m, &c);

   printf("Best-fitting line: y = %.4fx + %.4f\n", m, c);

   return 0;
}
```

Output 1:

```
Enter the number of data points: 5
Enter the x values:
x[0]: 1
x[1]: 2
x[2]: 4
x[3]: 5
x[4]: 7
Enter the y values:
y[0]: 5
y[1]: 12
y[2]: 45
y[3]: 89
y[4]: 104
Best-fitting line: y = 18.1140x + -17.8333
```

Output 2:

```
Enter the number of data points: 4
Enter the x values:
x[0]: 1
x[1]: 3
x[2]: 5
x[3]: 7
Enter the y values:
y[0]: 12
y[1]: 34
y[2]: 56
y[3]: 67
Best-fitting line: y = 9.3500x + 4.8500
```

RESULT AND DISCUSSION

The program successfully implements the **Ordinary Least Squares (OLS) method** to determine the best-fitting straight line. The computed equation of the line approximates the given dataset efficiently.

- Accuracy: The OLS method minimizes errors in linear regression.
- Efficiency: The algorithm efficiently computes the required parameters using basic summations.

CONCLUSION

The **Ordinary Least Squares (OLS) method** provides an effective way to fit a straight line to a dataset by minimizing the sum of squared residuals. The implementation in **C programming** demonstrates the practical application of regression in numerical methods.