Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторные работы по курсу «Численные методы»

Студент:	Белов И.А.
Преподаватель:	Пивоваров Д.Е.
Группа:	М8О-303Б-21
Дата:	
Оценка:	
Подпись:	

Москва, 2024

3.1

1 Постановка задачи

Используя таблицу значений Y_i функции $y = f(x)$, вычисленных в точках $X_i, i = 0, ..3$ построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки $\{X_i, Y_i\}$. Вычислить значение погрешности интерполяции в точке X^*.

Вариант: 2

2. $y = \cos(x)$, а) $X_i = 0, \frac{\pi}{6}, \frac{2\pi}{6}, \frac{3\pi}{6}$; б) $X_i = 0, \frac{\pi}{6}, \frac{5\pi}{12}, \frac{\pi}{2}$; $X^* = \frac{\pi}{4}$.

2 Результаты работы

```
// Для пункта а
Lagrange Interpolation Value at X* = 0.785398: 0.705889
Newton Interpolation Value at X* = 0.785398: 0.705889
Exact Value at X* = 0.707107
Lagrange Interpolation Error: 0.00121749
Newton Interpolation Error: 0.00121749

// Для пункта б
Lagrange Interpolation Value at X* = 0.785398: 0.70481
Newton Interpolation Value at X* = 0.785398: 0.70481
Exact Value at X* = 0.707107
Lagrange Interpolation Error: 0.0022963
Newton Interpolation Error: 0.0022963
```

Рис. 1: Вывод программы

3 Исходный код

```cpp
#include <iostream>
#include <cmath>
#include <vector>

#define _USE_MATH_DEFINES
#include<math.h>

using namespace std;

//
double factorial(int n) {
    double result = 1.0;
    for (int i = 2; i <= n; ++i) {
        result *= i;
    }
    return result;
}

//         x
double lagrangeInterpolation(const vector<double>& X, const vector<double>& Y, double x) {
    int n = X.size();
    double result = 0.0;

    for (int i = 0; i < n; ++i) {
        double term = Y[i];
        for (int j = 0; j < n; ++j) {
            if (i != j) {
                term *= (x - X[j]) / (X[i] - X[j]);
            }
        }
        result += term;
    }

    return result;
}

//         x
double newtonInterpolation(const vector<double>& X, const vector<double>& Y, double x) {
    int n = X.size();
    vector<vector<double>> dividedDifference(n, vector<double>(n));

    //
    for (int i = 0; i < n; ++i) {
        dividedDifference[i][0] = Y[i];
    }

    //
```

```
    for (int j = 1; j < n; ++j) {
        for (int i = 0; i < n - j; ++i) {
            dividedDifference[i][j] = (dividedDifference[i + 1][j - 1] -
                dividedDifference[i][j - 1]) / (X[i + j] - X[i]);
        }
    }

    //       x
    double result = dividedDifference[0][0];
    double term = 1.0;
    for (int i = 1; i < n; ++i) {
        term *= (x - X[i - 1]);
        result += term * dividedDifference[0][i];
    }

    return result;
}

//       cos(x)
double calculateError(double exactValue, double interpolatedValue) {
    return fabs(exactValue - interpolatedValue);
}

int main() {
    //   Xi       Yi
    vector<double> X = { 0, M_PI / 6, 2 * M_PI / 6, 3 * M_PI / 6 };

    //
    // vector<double> X = { 0, M_PI / 6, 5 * M_PI / 12, M_PI / 2 };

    vector<double> Y;

    for (double x : X) {
        Y.push_back(cos(x));
    }

    // ,
    double X_star = M_PI / 4;
    double exactValue = cos(X_star);

    //        X_star
    double lagrangeValue = lagrangeInterpolation(X, Y, X_star);
    double newtonValue = newtonInterpolation(X, Y, X_star);

    //
    double lagrangeError = calculateError(exactValue, lagrangeValue);
    double newtonError = calculateError(exactValue, newtonValue);

    //
```

```
    cout << "Lagrange Interpolation Value at X* = " << X_star << ": " << lagrangeValue << endl;
    cout << "Newton Interpolation Value at X* = " << X_star << ": " << newtonValue << endl;
    cout << "Exact Value at X* = " << exactValue << endl;
    cout << "Lagrange Interpolation Error: " << lagrangeError << endl;
    cout << "Newton Interpolation Error: " << newtonError << endl;

    return 0;
}
```

3.2

4 Постановка задачи

Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при $x = x_0$ и $x = x_4$. Вычислить значение функции в точке $x = X^*$.

Вариант: 2

2. $X^* = 1.5$

i	0	1	2	3	4
x_i	0.0	1.0	2.0	3.0	4.0
f_i	1.0	0.86603	0.5	0.0	-0.5

Рис. 2: Условие

5 Результаты работы

```
Value of the cubic spline at x* = 1.5 is: 0.71087
```

Рис. 3: Вывод программы

6 Исходный код

```cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <iomanip>

using namespace std;

//
struct Spline {
    double a, b, c, d, x;
};
```

```
//
void cubicSpline(const vector<double>& x, const vector<double>& y, vector<Spline>& splines) {
    int n = x.size();
    vector<double> h(n - 1), alpha(n - 1), l(n), mu(n), z(n);

    for (int i = 0; i < n - 1; ++i) {
        h[i] = x[i + 1] - x[i];
    }

    for (int i = 1; i < n - 1; ++i) {
        alpha[i] = (3 / h[i] * (y[i + 1] - y[i])) - (3 / h[i - 1] * (y[i] - y[i - 1]));
    }

    l[0] = 1;
    mu[0] = 0;
    z[0] = 0;

    for (int i = 1; i < n - 1; ++i) {
        l[i] = 2 * (x[i + 1] - x[i - 1]) - h[i - 1] * mu[i - 1];
        mu[i] = h[i] / l[i];
        z[i] = (alpha[i] - h[i - 1] * z[i - 1]) / l[i];
    }

    l[n - 1] = 1;
    z[n - 1] = 0;
    splines[n - 1].c = 0;

    for (int j = n - 2; j >= 0; --j) {
        splines[j].c = z[j] - mu[j] * splines[j + 1].c;
        splines[j].b = (y[j + 1] - y[j]) / h[j] - h[j] * (splines[j + 1].c + 2 * splines[j].c) / 3;
        splines[j].d = (splines[j + 1].c - splines[j].c) / (3 * h[j]);
        splines[j].a = y[j];
    }
}

//       x
double splineValue(const vector<Spline>& splines, double x) {
    int n = splines.size();
    Spline s;

    //      x
    for (int i = 0; i < n - 1; ++i) {
        if (x >= splines[i].x && x <= splines[i + 1].x) {
            s = splines[i];
            break;
        }
```

```
    }

    double dx = x - s.x;
    return s.a + s.b * dx + s.c * dx * dx + s.d * dx * dx * dx;
}

int main() {
    vector<double> x = {0.0, 1.0, 2.0, 3.0, 4.0};
    vector<double> y = {1.0, 0.86603, 0.5, 0.0, -0.5};
    int n = x.size();

    vector<Spline> splines(n);

    for (int i = 0; i < n; ++i) {
        splines[i].x = x[i];
    }

    cubicSpline(x, y, splines);

    double x_star = 1.5;
    double result = splineValue(splines, x_star);

    cout << "Value of the cubic spline at x* = " << x_star << " is: " << fixed <<
        setprecision(5) << result << endl;

    return 0;
}
```

3.3

7 Постановка задачи

Для таблично заданной функции путем решения нормальной системы МНК найти приближающие многочлены а) 1-ой и б) 2-ой степени. Для каждого из приближающих многочленов вычислить сумму квадратов ошибок. Построить графики приближаемой функции и приближающих многочленов.

Вариант: 2

2.

i	0	1	2	3	4	5
x_i	-1.0	0.0	1.0	2.0	3.0	4.0
y_i	0.86603	1.0	0.86603	0.50	0.0	-0.50

Рис. 4: Условия

8 Результаты работы

```
Coefficients for linear polynomial (1st degree): 1.038095, 0.752381, 0.466667, 0.180953, -0.104761, -0.39
Sum of squared errors for linear polynomial: 0.715612
Coefficients for quadratic polynomial (2nd degree): 1.169047, 0.904762, 0.497619, -0.052382,-0.745241, -1
Sum of squared errors for quadratic polynomial: 0.147016
```

Рис. 5: Вывод программы

9 Исходный код

```
#include <iostream>
#include <vector>
#include <cmath>
#include <Eigen/Dense> //
#include <matplotlibcpp.h> //

namespace plt = matplotlibcpp;
```

```

using namespace std;
using namespace Eigen;

//
VectorXd solveNormalEquations(const MatrixXd& A, const VectorXd& b) {
    return (A.transpose() * A).ldlt().solve(A.transpose() * b);
}

//
double calculateError(const vector<double>& x, const vector<double>& y, const VectorXd& coeffs) {
    double error = 0.0;
    int n = x.size();
    for (int i = 0; i < n; ++i) {
        double yi_approx = coeffs[0];
        for (int j = 1; j < coeffs.size(); ++j) {
            yi_approx += coeffs[j] * pow(x[i], j);
        }
        error += pow(y[i] - yi_approx, 2);
    }
    return error;
}

int main() {
    //
    vector<double> x = {-1.0, 0.0, 1.0, 2.0, 3.0, 4.0};
    vector<double> y = {0.86603, 1.0, 0.86603, 0.50, 0.0, -0.50};
    int n = x.size();

    //  1-  ( )
    MatrixXd A1(n, 2);
    VectorXd b1(n);
    for (int i = 0; i < n; ++i) {
        A1(i, 0) = 1;
        A1(i, 1) = x[i];
        b1[i] = y[i];
    }
    VectorXd coeffs1 = solveNormalEquations(A1, b1);
    double error1 = calculateError(x, y, coeffs1);

    //  2-  ( )
    MatrixXd A2(n, 3);
    VectorXd b2(n);
    for (int i = 0; i < n; ++i) {
        A2(i, 0) = 1;
        A2(i, 1) = x[i];
        A2(i, 2) = x[i] * x[i];
        b2[i] = y[i];
```

```
    }
    VectorXd coeffs2 = solveNormalEquations(A2, b2);
    double error2 = calculateError(x, y, coeffs2);

    //
    cout << "Coefficients for linear polynomial (1st degree): " << coeffs1.transpose() << endl;
    cout << "Sum of squared errors for linear polynomial: " << error1 << endl;
    cout << "Coefficients for quadratic polynomial (2nd degree): " << coeffs2.transpose() << endl;
    cout << "Sum of squared errors for quadratic polynomial: " << error2 << endl;

    //
    vector<double> y1_approx, y2_approx;
    for (double xi : x) {
        double y1i = coeffs1[0] + coeffs1[1] * xi;
        double y2i = coeffs2[0] + coeffs2[1] * xi + coeffs2[2] * xi * xi;
        y1_approx.push_back(y1i);
        y2_approx.push_back(y2i);
    }

    plt::figure();
    plt::plot(x, y, "bo", {{"label", "Data points"}});
    plt::plot(x, y1_approx, "r-", {{"label", "Linear polynomial"}});
    plt::plot(x, y2_approx, "g-", {{"label", "Quadratic polynomial"}});
    plt::legend();
    plt::title("Approximating polynomials using least squares");
    plt::xlabel("x");
    plt::ylabel("y");
    plt::show();

    return 0;
}
```

3.4

10 Постановка задачи

Вычислить первую и вторую производную от таблично заданной функции $y_i = f(x_i), i = 0, 1, 2, 3, 4$ в точке $x = X_i$.

Вариант: 2

2. $X^* = 1.0$

i	0	1	2	3
x_i	-1.0	0.0	1.0	2.0
y_i	-0.5	0.0	0.5	0.86603

Рис. 6: Условия

11 Результаты работы

```
Central first derivative at x* = 1: 0.43301
Left first derivative at x* = 1.00000: 0.50000
Right first derivative at x* = 1.00000: 0.36603
Second derivative at x* = 1.00000: -0.13397
```

Рис. 7: Вывод программы

12 Исходный код

```
#include <iostream>
#include <vector>
#include <iomanip>

using namespace std;

//
double centralFirstDerivative(const vector<double>& x, const vector<double>& y, int i)
    {
    return (y[i + 1] - y[i - 1]) / (x[i + 1] - x[i - 1]);
```

```
}

//
double leftFirstDerivative(const vector<double>& x, const vector<double>& y, int i) {
    if (i == 0) {
        cerr << "Error: Left derivative not defined for the first point." << endl;
        return 0.0;
    }
    return (y[i] - y[i - 1]) / (x[i] - x[i - 1]);
}

//
double rightFirstDerivative(const vector<double>& x, const vector<double>& y, int i) {
    if (i == x.size() - 1) {
        cerr << "Error: Right derivative not defined for the last point." << endl;
        return 0.0;
    }
    return (y[i + 1] - y[i]) / (x[i + 1] - x[i]);
}

//
double secondDerivative(const vector<double>& x, const vector<double>& y, int i) {
    if (i == 0 || i == x.size() - 1) {
        cerr << "Error: Second derivative not defined for the first or last point." << endl;
        return 0.0;
    }
    return (y[i + 1] - 2 * y[i] + y[i - 1]) / ((x[i] - x[i - 1]) * (x[i] - x[i - 1]));
}

int main() {
    vector<double> x = { -1.0, 0.0, 1.0, 2.0, 3.0 };
    vector<double> y = { -0.5, 0.0, 0.5, 0.86603, 1.0 };
    double x_star = 1.0;
    int i = 2; //   x* = 1.0   x

    //
    double central_first_derivative = centralFirstDerivative(x, y, i);
    double left_first_derivative = leftFirstDerivative(x, y, i);
    double right_first_derivative = rightFirstDerivative(x, y, i);
    double second_derivative = secondDerivative(x, y, i);

    //
    cout << "Central first derivative at x* = " << x_star << ": " << fixed << setprecision(5) << central_first_derivative << endl;
    cout << "Left first derivative at x* = " << x_star << ": " << fixed << setprecision(5) << left_first_derivative << endl;
    cout << "Right first derivative at x* = " << x_star << ": " << fixed << setprecision(5) << right_first_derivative << endl;
```

```cpp
    cout << "Second derivative at x* = " << x_star << ": " << fixed << setprecision(5)
        << second_derivative << endl;

    return 0;
}
```

3.5

13 Постановка задачи

Вычислить определенный интеграл $\int\limits_{X_0}^{X_1} ydx$, методами прямоугольников, трапеций, Симпсона с шагами h_1, h_2. Оценить погрешность вычислений, используя Метод Рунге-Ромберга:
Вариант: 2

2. $y = \dfrac{x}{(3x+4)^2}$, $X_0 = 0, \quad X_k = 4, \quad h_1 = 1.0, \quad h_2 = 0.5$;

14 Результаты работы

15 Исходный код

```
#include <iostream>
#include <vector>
#include <fstream>
#include <cmath>

using namespace std;

//     y = x / (3x + 4)^2
double func(double x) {
    return x / pow((3 * x + 4), 2);
}

//
double rectangle_method(double x0, double xk, double h) {
    double F = 0;
    double n = (int)((xk - x0) / h);
    n += 1;
    vector<double> x_values(n, 0);
    for (int i = 0; i < n; i++) {
        x_values[i] = x0 + h * i;
    }
    for (int i = 1; i < n; i++) {
        F += h * func((x_values[i] + x_values[i - 1]) / 2);
    }
    return F;
}

//
double trapez_method(double x0, double xk, double h) {
    double F = 0;
    double n = (int)((xk - x0) / h);
    n += 1;
    vector<double> x_values(n, 0);
    for (int i = 0; i < n; i++) {
        x_values[i] = x0 + h * i;
    }
    for (int i = 1; i < n; i++) {
        F += (func(x_values[i]) + func(x_values[i - 1])) / 2 * h;
    }
    return F;
}

//
double simps_method(double x0, double xk, double h) {
    int n = (int)((xk - x0) / h);
    double F = 0;
    for (int i = 0; i < n; i++) {
```

```
        double x1 = x0 + i * h;
        double x2 = x0 + (i + 1) * h;
        double x3 = x0 + (i + 0.5) * h;
        F += (h / 6) * (func(x1) + 4 * func(x3) + func(x2));
    }
    return F;
}

//    --
vector<double> runge_romb_rich(double x0, double xk, double h, int p) {
    vector<double> results(3, 0);
    results[0] = rectangle_method(x0, xk, h / 2) + (rectangle_method(x0, xk, h / 2) -
        rectangle_method(x0, xk, h)) / (pow(2, p) - 1);
    results[1] = trapez_method(x0, xk, h / 2) + (trapez_method(x0, xk, h / 2) -
        trapez_method(x0, xk, h)) / (pow(2, p) - 1);
    results[2] = simps_method(x0, xk, h / 2) + (simps_method(x0, xk, h / 2) -
        simps_method(x0, xk, h)) / (pow(2, p) - 1);
    return results;
}

int main() {
    ofstream fout("answer5.txt");

    double x0 = 0, xk = 4, h1 = 1.0, h2 = 0.5;

    fout << "Rectangle with h = 1\n";
    double rect1 = rectangle_method(x0, xk, h1);
    fout << rect1;
    fout << "\nRectangle with h = 0.5\n";
    double rect2 = rectangle_method(x0, xk, h2);
    fout << rect2;

    fout << "\n\nTrapez with h = 1\n";
    double trap1 = trapez_method(x0, xk, h1);
    fout << trap1;
    fout << "\nTrapez with h = 0.5\n";
    double trap2 = trapez_method(x0, xk, h2);
    fout << trap2;

    fout << "\n\nSimpson with h = 1\n";
    double simp1 = simps_method(x0, xk, h1);
    fout << simp1;
    fout << "\nSimpson with h = 0.5\n";
    double simp2 = simps_method(x0, xk, h2);
    fout << simp2 << endl << endl;

    vector<double> RRR1 = runge_romb_rich(x0, xk, h1, 1);
    vector<double> RRR2 = runge_romb_rich(x0, xk, h2, 1);

```

```
    fout << "with Runge-Romberg-Richardson method" << endl;

    fout << "Rectangle with h = 1\n";
    fout << RRR1[0];
    fout << "\nEstimate:" << fabs(rect1 - RRR1[0]);

    fout << "\nRectangle with h = 0.5\n";
    fout << RRR2[0];
    fout << "\nEstimate:" << fabs(rect2 - RRR2[0]);

    fout << "\n\nTrapez with h = 1\n";
    fout << RRR1[1];
    fout << "\nEstimate:" << fabs(trap1 - RRR1[1]);

    fout << "\nTrapez with h = 0.5\n";
    fout << RRR2[1];
    fout << "\nEstimate:" << fabs(trap2 - RRR2[1]);

    fout << "\n\nSimpson with h = 1\n";
    fout << RRR1[2];
    fout << "\nEstimate:" << fabs(simp1 - RRR1[2]);

    fout << "\nSimpson with h = 0.5\n";
    fout << RRR2[2];
    fout << "\nEstimate:" << fabs(simp2 - RRR2[2]);

    fout.close();

    return 0;
}
```

```
Rectangle with h = 1
0.0728406
Rectangle with h = 0.5
0.0713277

Trapez with h = 1
0.0659721
Trapez with h = 0.5
0.0694064

Simpson with h = 1
0.0705511
Simpson with h = 0.5
0.0706872

with Runge-Romberg-Richardson method
Rectangle with h = 1
0.0698147
Estimate:0.00302589
Rectangle with h = 0.5
0.0704009
Estimate:0.000926725

Trapez with h = 1
0.0728406
Estimate:0.00686847
Trapez with h = 0.5
0.0713277
Estimate:0.00192129

Simpson with h = 1
0.0708234
Estimate:0.00027223
Simpson with h = 0.5
0.0707099
Estimate:2.2613e-05
```

Рис. 8: Вывод программы