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:	М. Д. Жилин
Преподаватель:	Д. Е. Пивоваров
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: 9

$y = arccos(x), a)X_i = -0.4, -0.1, 0.2, 0.5;)X_i = -0.4, 0, 0.2, 0.5; X^* = 0.1$

2 Результаты работы

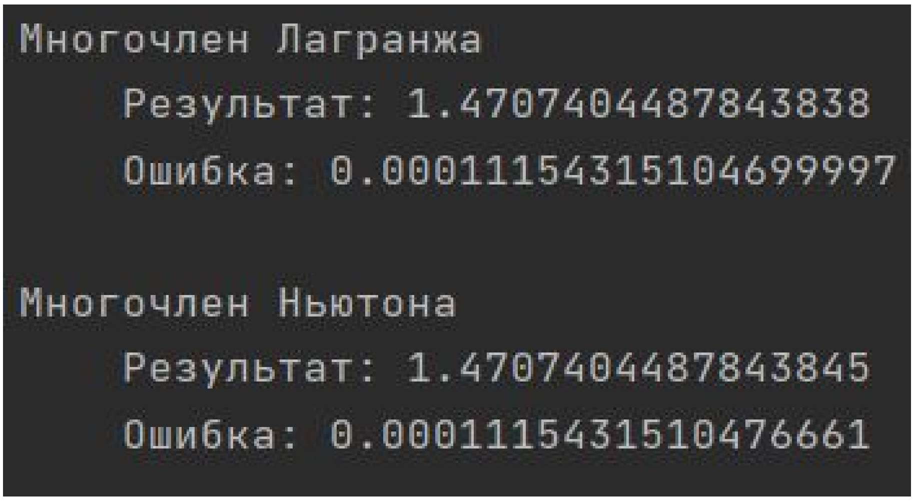

Рис. 1: Вывод программы в консоли

3 Исходный код

```cpp
#include <bits/stdc++.h>

using namespace std;

double f(double x) {
    return acos(x);
}

double lagrange_interpolation(double x, const vector<pair<double, double>>& coords) {
    vector<double> coefficients(coords.size());
    int n = coords.size();

    for (int i = 0; i < n; ++i)
        coefficients[i] = coords[i].second;

    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (i != j)
                coefficients[i] /= coords[i].first - coords[j].first;

    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            if (i != j)
                coefficients[i] *= x - coords[j].first;

    return accumulate(coefficients.begin(), coefficients.end(), 0.0);
}

double newton_interpolation(double x, const vector<pair<double, double>>& coords) {
    vector<double> coefficients(coords.size());
    int n = coords.size();

    for (int i = 0; i < n; ++i)
        coefficients[i] = coords[i].second;

    for (int i = 1; i < n; ++i)
        for (int j = n - 1; j > i-1; --j)
            coefficients[j] = (coefficients[j] - coefficients[j - 1]) / (coords[j].first - coords[j - i].first);

    for (int i = 1; i < n; ++i) {
        for (int j = 0; j < i; ++j) {
            coefficients[i] *= x - coords[j].first;
        }
    }

    return accumulate(coefficients.begin(), coefficients.end(), 0.0);
```

```
}

int main() {
    vector<double> x_vect = {-0.4, -0.1, 0.2, 0.5};
    double x_marked = 0.1;
    vector<pair<double, double>> cord;

    for (double x : x_vect)
        cord.emplace_back(x, f(x));

    cout << "\nThe Lagrange polynomial\n";
    cout << "\tResult: " << lagrange_interpolation(x_marked, cord) << endl;
    cout << "\tLoss: " << abs(lagrange_interpolation(x_marked, cord) - f(x_marked)) << endl;
    cout << "\nThe Newton polynomial\n";
    cout << "\tResult: " << newton_interpolation(x_marked, cord) << endl;
    cout << "\tLoss: " << abs(newton_interpolation(x_marked, cord) - f(x_marked)) << 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: 9

9. $X^* = 0.1$

i	0	1	2	3	4
x_i	-0.4	-0.1	0.2	0.5	0.8
f_i	1.9823	1.6710	1.3694	1.0472	0.64350

Рис. 2: Условие

5 Результаты работы

```
Result = 1.4694391534391535
```

Рис. 3: Вывод программы в консоли

6 Исходный код

```cpp
#include <bits/stdc++.h>

using namespace std;
using matrix = vector<vector<double> >;

matrix multiple_matrix(matrix& matrix1, matrix& matrix2) {
    int n1 = matrix1.size(), m1 = matrix1[0].size(), m2 = matrix2[0].size();
    matrix res(n1);
    for (int i=0; i<n1; i++)
        for (int j=0; j<m2; j++)
            res[i].push_back(0);

    for (int i=0; i<n1; i++) {
        for (int j=0; j<m2; j++) {
            double cntr = 0;
            for (int k=0; k<m1; k++)
               cntr += matrix1[i][k] * matrix2[k][j];
            res[i][j] = cntr;
        }
    }
    return res;
}

matrix tridiagonal_algorithm(matrix& coefficients, matrix& results) {
    double a, b, c, d;
    a = 0;
    b = coefficients[0][0];
    c = coefficients[0][1];
    d = results[0][0];
    vector<double> P(coefficients[0].size(), 0), Q(coefficients[0].size(), 0);

    P[0] = -c/b;
    Q[0] = d/b;
    for (int i=1; i < coefficients.size() - 1; i++){
        a = coefficients[i][i-1];
        b = coefficients[i][i];
        c = coefficients[i][i+1];
        d = results[i][0];

        P[i] = -c/(b + a*P[i-1]);
        Q[i] = (d - a*Q[i-1])/(b + a*P[i-1]);
    }

    a = coefficients[coefficients.size()-1][coefficients[0].size()-2];
    b = coefficients[coefficients.size()-1][coefficients[0].size()-1];
    c = 0;
    d = results[results.size()-1][0];
```

```
    Q[Q.size()-1] = (d - a * Q[Q.size()-2]) / (b + a * P[P.size()-2]);

    matrix decision(results.size());
    for(int i=0; i<decision.size(); i++)
        decision[i].push_back(0);

    decision[decision.size()-1][0] = Q[Q.size()-1];
    for (int i = decision.size()-2; i > -1; i--)
        decision[i][0] = P[i]*decision[i+1][0] + Q[i];

    return decision;
}


void print_matrix(matrix& matrix1) {
    for(const auto& vect: matrix1) {
        for (auto x: vect)
            cout << x << " ";
        cout << endl;
    }
}

int main() {
    double x_marked = 0.1;
    vector<double> x = {-0.4, -0.1, 0.2, 0.5, 0.8};
    vector<double> y = {1.9823, 1.6710, 1.3694, 1.0472, 0.64350};

    vector<double> h = {0.0};
    for (int i = 0; i < 4; ++i) {
        h.push_back(x[i + 1] - x[i]);
    }
    int n = x.size() - 1;

    vector<vector<double>> matr_data = {{2 * (h[1] + h[2]), h[2], 0}}, root_data = {};

    for (int i=3; i<n; i++)
        matr_data.push_back({h[i - 1], 2 * (h[i - 1] + h[i]), h[i]});

    for (int i=0; i<n-1; i++)
        root_data.push_back({3 * ((y[i + 2] - y[i + 1]) / h[i + 2] - (y[i + 1] - y[i])
            / h[i + 1])});

    matr_data.push_back({0, h[n - 1], 2 * (h[n - 1] + h[n])});

    matrix matr(matr_data);
    matrix root(root_data);

    vector<double> coeff_a(y.begin(), y.end() - 1);
```

```
    vector<double> coeff_c = {0};
    auto result = tridiagonal_algorithm(matr, root);
    for (auto val : result) {
        coeff_c.push_back(val[0]);
    }
    vector<double> coeff_b;
    for (int i = 1; i < n; ++i) {
        coeff_b.push_back((y[i] - y[i - 1]) / h[i] - h[i] * (coeff_c[i] + 2 * coeff_c[i
             - 1]) / 3);
    }
    coeff_b.push_back((y[n] - y[n - 1]) / h[n] - 2 * h[n] * coeff_c[n - 1] / 3);

    vector<double> coeff_d;
    for (int i = 0; i < n - 1; ++i) {
        coeff_d.push_back((coeff_c[i + 1] - coeff_c[i]) / (3 * h[i + 1]));
    }
    coeff_d.push_back(-coeff_c[n - 1] / (3 * h[n]));

    bool flag = false;
    for (int i = 0; i < n; ++i) {
        if (x[i] <= x_marked && x_marked <= x[i + 1]) {
            double res = coeff_a[i] + coeff_b[i]*(x_marked-x[i]) + coeff_c[i]*(x_marked
                -x[i])*(x_marked-x[i]) + coeff_d[i]*(x_marked-x[i])*(x_marked-x[i])*(
                x_marked-x[i]);
            cout << "Result = " << res << endl;
            flag = true;
            break;
        }
    }

    if (flag)
        cout << "Incorrect value" << 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: 9

9.

i	0	1	2	3	4	5
x_i	-0.7	-0.4	-0.1	0.2	0.5	0.8
y_i	2.3462	1.9823	1.671	1.3694	1.0472	0.6435

Рис. 4: Условия

8 Результаты работы

```
F1(x) = 1.5652685714285715 - 1.1067047619047614*x
Loss = 0.003940351047619047

F2(x) = 1.5777836507936505 - 1.1018912698412695*x - 0.04813492063491959*x^2
Loss = 0.0032396991428571102
```

Рис. 5: Вывод программы в консоли

9 Исходный код

```
#include <bits/stdc++.h>

using namespace std;
using matrix = vector<vector<double> >;

matrix multiple_matrix(matrix& matrix1, matrix& matrix2) {
    int n1 = matrix1.size(), m1 = matrix1[0].size(), m2 = matrix2[0].size();
    matrix res(n1);
    for (int i=0; i<n1; i++)
        for (int j=0; j<m2; j++)
            res[i].push_back(0);

    for (int i=0; i<n1; i++) {
        for (int j=0; j<m2; j++) {
            double cntr = 0;
            for (int k=0; k<m1; k++)
               cntr += matrix1[i][k] * matrix2[k][j];
            res[i][j] = cntr;
        }
    }
    return res;
}

pair<matrix, matrix> lu_decomposition(matrix& coefficients, matrix& results) {
    int n1=coefficients.size(), m1=coefficients[0].size(), m2 = results[0].size();
    matrix L(n1), U = coefficients;
    for (int i=0; i<n1; i++)
        for (int j=0; j<m1; j++)
            L[i].push_back(0);

    for (int k=0; k<n1; k++) {
        if (U[k][k] == 0) {
            for (int i=k+1; i<n1; i++) {
                if (U[i][k] != 0) {
                    swap(U[k], U[i]);
                    swap(L[k], L[i]);
                    swap(coefficients[k], coefficients[i]);
                    swap(results[k], results[i]);
                    break;
                }
            }
        }
        L[k][k] = 1;
        for (int i=k+1; i<n1; i++) {
            L[i][k] = U[i][k]/U[k][k];
            if (U[i][k] == 0)
                continue;
```

```cpp
            for(int j=k; j<m1; j++)
                U[i][j] -= L[i][k]*U[k][j];

        }
    }

    return make_pair(L, U);
}

matrix calculate_decisions(matrix& coefficients, matrix& results) {
    auto [L, U] = lu_decomposition(coefficients, results);
    matrix res = results;

    for (int k=0; k<res[0].size(); k++)
        for (int i=0; i<res.size(); i++)
            for (int j=0; j<i; j++)
                res[i][k] -= res[j][k]*L[i][j];
    for (int k=0; k<res[0].size(); k++) {
        for (int i=coefficients.size()-1; i>-1; i--) {
            for (int j=i+1; j<results.size(); j++) {
                res[i][k] -= res[j][k]*U[i][j];
            }
            res[i][k] /= U[i][i];
        }
    }

    return res;
}

void print_matrix(const matrix& matrix1) {
    for(const auto& vect: matrix1) {
        for (auto x: vect)
            cout << x << " ";
        cout << endl;
    }
}

int main() {
    vector<double> x = {-0.7, -0.4, -0.1, 0.2, 0.5, 0.8}, y = {2.3462, 1.9823, 1.671,
        1.3694, 1.0472, 0.6435};

    double n = x.size();
    double sum_x = 0, sum_x2 = 0, sum_x3 = 0, sum_x4 = 0, sum_y = 0, sum_yx = 0,
        sum_yx2 = 0;

    for (int i=0; i<x.size(); i++){
        sum_x += x[i];
        sum_x2 += pow(x[i], 2);
        sum_x3 += pow(x[i], 3);
```

```
        sum_x4 += pow(x[i], 4);
        sum_y += y[i];
        sum_yx += y[i]*x[i];
        sum_yx2 += y[i]*pow(x[i], 2);
    }

    matrix matr = {
        {n, sum_x},
        {sum_x, sum_x2}
    };

    matrix root = {
        {sum_y},
        {sum_yx}
    };

    matrix decisions = calculate_decisions(matr, root);

    cout << "F1(x) = (" << decisions[0][0] << ") + (" << decisions[1][0] << ")*x" <<
        endl;
    double loss_f1 = 0;
    for (int i = 0; i < n; i++)
        loss_f1 += pow(decisions[0][0] + decisions[1][0] * x[i] - y[i], 2);
    cout << "Loss = " << loss_f1 << endl << endl;


    matr = {
        {n, sum_x, sum_x2},
        {sum_x, sum_x2, sum_x3},
        {sum_x2, sum_x3, sum_x4}
    };

    root = {
        {sum_y},
        {sum_yx},
        {sum_yx2}
    };

    decisions = calculate_decisions(matr, root);

    cout << "F2(x) = (" << decisions[0][0] << ") + (" << decisions[1][0] << ")*x + ("
        << decisions[2][0] << ")*x^2" << endl;
    double loss_f2 = 0;
    for (int i = 0; i < n; i++)
        loss_f2 += pow(decisions[0][0] + decisions[1][0] * x[i] + decisions[2][0] * pow
            (x[i], 2) - y[i], 2);
    cout << "Loss = " << loss_f2 << endl;

    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: 9

9. $X^* = 1.0$

i	0	1	2	3	4
x_i	-1.0	0.0	1.0	2.0	3.0
y_i	2.3562	1.5708	0.7854	0.46365	0.32175

Рис. 6: Условия

11 Результаты работы

```
The left-hand first derivative 1.1620000000000008
The left-hand first derivative 1.2850000000000008

First derivative 1.23
```

Рис. 7: Вывод программы в консоли

12 Исходный код

```cpp
#include <bits/stdc++.h>


using namespace std;


int main() {
    double x_marked = 0.2;
    vector<double> x = {0.0, 0.1, 0.2, 0.3, 0.4}, y = {1.0, 1.1052, 1.2214, 1.3499, 1.4918};

    int n = x.size();
    vector<double> derivative_first;
    for (int i = 0; i < n - 1; ++i) {
        derivative_first.push_back((y[i + 1] - y[i]) / (x[i + 1] - x[i]));
    }

    vector<double> derivative_second;
    for (int i = 0; i < n - 2; ++i) {
        derivative_second.push_back(2 * ((y[i + 2] - y[i + 1]) / (x[i + 2] - x[i + 1]) - (y[i + 1] - y[i]) / (x[i + 1] - x[i])) / (x[i + 2] - x[i]));
    }

    for (int i = 0; i < n - 1; ++i) {
        if (x[i] == x_marked) {
            cout << "The left-hand first derivative: " << derivative_first[i - 1] << endl;
            cout << "The right-hand first derivative: " << derivative_first[i] << endl;
            break;
        } else if (x[i] < x_marked && x_marked < x[i + 1]) {
            cout << "First derivative: " << derivative_first[i] << endl;
        }
    }

    cout << endl;

    for (int i = 0; i < n - 2; ++i) {
        if (x[i] <= x_marked && x_marked <= x[i + 1]) {
            cout << "Second derivative: " << derivative_second[i] << endl;
            break;
        }
    }

    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: 9
$y = \frac{x}{x^2+9}$ $X_0 = 0, X_k = 2, h_1 = 0.5, h_2 = 0.25$

14 Результаты работы

```
Rectangular method
    Integral = 0.1847192474595487       (Step = 0.5)
    Integral = 0.18407516757447664      (Step = 0.25)
    Integral = 0.18386254390732204      (Step = 1e-06)
    Integral = 0.18386047427945262      (Runge-Romberg-Richardson estimation, step 1 = 0.5, step 2 = 0.25)


Trapeze method
    Integral = 0.18215523215523216      (Step = 0.5)
    Integral = 0.18343723980739043      (Step = 0.25)
    Integral = 0.1838625439073104       (Step = 1e-06)
    Integral = 0.18386457569144318      (Runge-Romberg-Richardson estimation, step 1 = 0.5, step 2 = 0.25)


Simpson method
    Integral = 0.1838992838992839       (Step = 0.5)
    Integral = 0.18386457569144318      (Step = 0.25)
    Integral = 0.18386249262526086      (Step = 1e-06)
    Integral = 0.1838530062888296       (Runge-Romberg-Richardson estimation, step 1 = 0.5, step 2 = 0.25)
```

Рис. 8: Вывод программы в консоли

15 Исходный код

```cpp
#include <bits/stdc++.h>

using namespace std;


double rectangular_method(function<double(double)> f, double x_start, double x_end, double step){
    double x = x_start, res = 0;
    while (x < x_end){
        res += f((2*x + step)/2);
        x += step;
    }
    return step*res;
}


double trapeze_method(function<double(double)> f, double x_start, double x_end, double step){
    double x = x_start+step, res = f(x_start)/2 + f(x_end)/2;
    while (x < x_end){
        res += f(x);
        x += step;
    }
    return step * res;
}


double simpson_method(function<double(double)> f, double x_start, double x_end, double step){
    double x = x_start + step, res = f(x_start) + f(x_end);
    bool flag = true;
    while (x < x_end){
        res += f(x) * ((flag) ? 4 : 2);
        x += step;
        flag = !flag;
    }
    return step * res / 3;
}


double RRR_estimation(double F_1, double F_2, double step_1, double step_2, double p){
    return F_1 + (F_1 - F_2)/(pow((step_2/step_1), p) - 1);
}


int main() {
    auto y = [](double x) { return x / (x*x + 9); };
```

```
    double x_0 = 0, x_k = 2;
    vector<double> precision = {0.5, 0.25, 0.000001};

    // auto y = [](double x) { return x / ((3*x + 4)*(3*x + 4)); };
    // double x_0 = -1, x_k = 1;
    // vector<double> precision = {0.5, 0.25, 0.000001};

    vector<pair<string, function<double(function<double(double)>, double, double,
        double)>>> methods = {
        {"Rectangular", rectangular_method},
        {"Trapeze", trapeze_method},
        {"Simpson", simpson_method}
    };

    for (auto& method : methods) {
        cout << method.first << " method" << endl;
        vector<double> F;
        for (auto h : precision) {
            F.push_back(method.second(y, x_0, x_k, h));
            cout << "\tIntegral = " << F.back() << "\t\t(Step = " << h << ")" << endl;
        }
        cout << "\tIntegral = " << RRR_estimation(F[0], F[1], precision[0], precision
            [1], 2) << "\t\t(Runge-Romberg-Richardson estimation, step 1 = " <<
            precision[0] << ", step 2 = " << precision[1] << ")" << endl;
        cout << "\n";
    }
    return 0;
}
```