Московский авиационный институт
(национальный исследовательский университет)

Факультет компьютерных наук и прикладной математики

Кафедра математической кибернетики

Лабораторные работы по курсу «Численные методы»

Лабораторная работа №3

Студент: Ершов С.Г.
Преподаватель: Пивоваров Д.Е.
Дата:
Оценка:
Подпись:

Москва, 2024

3 Методы приближения функций

1 Полиномиальная интерполяция

1.1 Постановка задачи

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

1.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/1.in
4
-3 -1 1 3
-0.5
$ ./solution <tests/1.in
Интерполяционный многочлен Лагранжа: 0.831529 * x -0.0461312 * x ^ 3
Погрешность в точке X*: 0.0536493
Интерполяционный многочлен Ньютона: 0.831529 * x -0.0461312 * x ^ 3
Погрешность в точке X*: 0.0536493
$ cat tests/2.in
4
-3 0 1 3
-0.5
$ ./solution <tests/2.in
Интерполяционный многочлен Лагранжа: 0.831529 * x -0.0461312 * x ^ 3
Погрешность в точке X*: 0.0536493
Интерполяционный многочлен Ньютона: 0.831529 * x -0.0461312 * x ^ 3
Погрешность в точке X*: 0.0536493
```

1.3 Исходный код

```
#ifndef INTERPOLATOR_HPP
#define INTERPOLATOR_HPP

#include "../polynom.hpp"

using vec = std::vector<double>;

class inter_lagrange {
    vec x;
    vec y;
    size_t n;

  public:
    inter_lagrange(const vec& _x, const vec& _y) : x(_x), y(_y), n(x.size()){};

    polynom operator()() {
        polynom res(vec({0}));
        for (size_t i = 0; i < n; ++i) {
            polynom li(vec({1}));
            for (size_t j = 0; j < n; ++j) {
                if (i == j) {
                    continue;
                }
                polynom xij(vec({-x[j], 1}));
                li = li * xij;
                li = li / (x[i] - x[j]);
            }
            res = res + y[i] * li;
        }
        return res;
    }
};

class inter_newton {
  private:
    using vvd = std::vector<std::vector<double> >;
    using vvb = std::vector<std::vector<bool> >;

    vec x;
    vec y;
    size_t n;

    vvd memo;
    vvb calc;

    double f(int l, int r) {
        if (calc[l][r]) {
```

```
            return memo[l][r];
        }
        calc[l][r] = true;
        double res;
        if (l + 1 == r) {
            res = (y[l] - y[r]) / (x[l] - x[r]);
        } else {
            res = (f(l, r - 1) - f(l + 1, r)) / (x[l] - x[r]);
        }
        return memo[l][r] = res;
    }

  public:
    inter_newton(const vec& _x, const vec& _y) : x(_x), y(_y), n(x.size()) {
        memo.resize(n, std::vector<double>(n));
        calc.resize(n, std::vector<bool>(n));
    };

    polynom operator()() {
        polynom res(vec({y[0]}));
        polynom li(vec({-x[0], 1}));
        int r = 0;
        for (size_t i = 1; i < n; ++i) {
            res = res + f(0, ++r) * li;
            li = li * polynom(vec({-x[i], 1}));
        }
        return res;
    }
};

#endif /* INTERPOLATOR_HPP */
```

2 Сплай-интерполяция

2.1 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 $x = X^*$.

2.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/2.in
5
-3.0 -1.0 1.0 3.0 5.0
-1.2490 -0.78540 0.78540 1.2490 1.3734
-0.5
$ ./solution <tests/2.in
Полученные сплайны:
i = 1,a = -1.2490,b = 0.0470,c = 0.0000,d = 0.0462
i = 2,a = -0.7854,b = 0.6014,c = 0.2772,d = -0.0926
i = 3,a = 0.7854,b = 0.5990,c = -0.2784,d = 0.0474
i = 4,a = 1.2490,b = 0.0542,c = 0.0060,d = -0.0010

Значение функции в точке x0 = -0.5000,f(x0) = -0.4270
```

2.3 Исходный код

```
#ifndef CUBIC_SPLINE_HPP
#define CUBIC_SPLINE_HPP

#include "../lab1_2/tridiag.hpp"

class cubic_spline_t {
    using vec = std::vector<double>;
    using tridiag = tridiag_t<double>;
    size_t n;
    vec x;
    vec y;
    vec a, b, c, d;

    void build_spline() {
        vec h(n + 1);
        h[0] = NAN;
        for (size_t i = 1; i <= n; ++i) {
            h[i] = x[i] - x[i - 1];
        }
        vec eq_a(n - 1);
        vec eq_b(n - 1);
        vec eq_c(n - 1);
        vec eq_d(n - 1);
        for (size_t i = 2; i <= n; ++i) {
            eq_a[i - 2] = h[i - 1];
            eq_b[i - 2] = 2.0 * (h[i - 1] + h[i]);
            eq_c[i - 2] = h[i];
            eq_d[i - 2] = 3.0 * ((y[i] - y[i - 1]) / h[i] -
                                 (y[i - 1] - y[i - 2]) / h[i - 1]);
        }
        eq_a[0] = 0.0;
        eq_c.back() = 0.0;
        // for (size_t i = 0; i < n - 1; ++i) {
        // printf("%lf %lf %lf %lf\n", eq_a[i], eq_b[i], eq_c[i], eq_d[i]);
        // }
        tridiag system_of_eq(eq_a, eq_b, eq_c);
        vec c_solved = system_of_eq.solve(eq_d);
        for (size_t i = 2; i <= n; ++i) {
            c[i] = c_solved[i - 2];
        }
        for (size_t i = 1; i <= n; ++i) {
            a[i] = y[i - 1];
        }
        for (size_t i = 1; i < n; ++i) {
            b[i] =
                (y[i] - y[i - 1]) / h[i] - h[i] * (c[i + 1] + 2.0 * c[i]) / 3.0;
            d[i] = (c[i + 1] - c[i]) / (3.0 * h[i]);
```

```
        }
        c[1] = 0.0;
        b[n] = (y[n] - y[n - 1]) / h[n] - (2.0 / 3.0) * h[n] * c[n];
        d[n] = -c[n] / (3.0 * h[n]);
    }

  public:
    cubic_spline_t(const vec& _x, const vec& _y) {
        if (_x.size() != _y.size()) {
            throw std::invalid_argument("Sizes does not match");
        }
        x = _x;
        y = _y;
        n = x.size() - 1;
        a.resize(n + 1);
        b.resize(n + 1);
        c.resize(n + 1);
        d.resize(n + 1);
        build_spline();
    }

    friend std::ostream& operator<<(std::ostream& out,
                                    const cubic_spline_t& spline) {
        for (size_t i = 1; i <= spline.n; ++i) {
            out << "i = " << i << ", a = " << spline.a[i]
                << ", b = " << spline.b[i] << ", c = " << spline.c[i]
                << ", d = " << spline.d[i] << '\n';
        }
        return out;
    }

    double operator()(double x0) {
        for (size_t i = 1; i <= n; ++i) {
            if (x[i - 1] <= x0 and x0 <= x[i]) {
                double x1 = x0 - x[i - 1];
                double x2 = x1 * x1;
                double x3 = x2 * x1;
                return a[i] + b[i] * x1 + c[i] * x2 + d[i] * x3;
            }
        }
        return NAN;
    }
};

#endif /* CUBIC_SPLINE_HPP */
```

3 Метод наименьших квадратов

3.1 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.

3.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/2.in
6
-5.0 -3.0 -1.0 1.0 3.0 5.0
-1.3734 -1.249 -0.7854 0.7854 1.249 1.3734
$ ./solution <tests/2.in
Полученная функция первого порядка: 0.0000 0.3257
Значение суммы квадратов ошибков: 0.7007
Полученная функция второго порядка: 0.0000 0.3257 -0.0000
Значение суммы квадратов ошибков: 0.7007
```

3.3 Исходный код

```
#ifndef MINIMAL_SQUARE_HPP
#define MINIMAL_SQUARE_HPP

#include <functional>

#include "../lab1_1/lu.hpp"
#include "../polynom.hpp"

class minimal_square_t {
    using vec = std::vector<double>;
    using matrix = matrix_t<double>;
    using lu = lu_t<double>;

    using func = std::function<double(double)>;
    using vf = std::vector<func>;

    size_t n;
    vec x;
    vec y;
    size_t m;
    vec a;
    vf phi;

    void build() {
        matrix lhs(n, m);
        for (size_t i = 0; i < n; ++i) {
            for (size_t j = 0; j < m; ++j) {
                lhs[i][j] = phi[j](x[i]);
            }
        }
        matrix lhs_t = lhs.t();
        lu lhs_lu(lhs_t * lhs);
        vec rhs = lhs_t * y;
        a = lhs_lu.solve(rhs);
    }

    double get(double x0) {
        double res = 0.0;
        for (size_t i = 0; i < m; ++i) {
            res += a[i] * phi[i](x0);
        }
        return res;
    }

   public:
    minimal_square_t(const vec& _x, const vec& _y, const vf& _phi) {
        if (_x.size() != _y.size()) {
```

```
            throw std::invalid_argument("Sizes does not match");
        }

        n = _x.size();
        x = _x;
        y = _y;
        m = _phi.size();
        a.resize(m);
        phi = _phi;
        build();
    }


    friend std::ostream& operator<<(std::ostream& out,
                                    const minimal_square_t& item) {
        for (size_t i = 0; i < item.m; ++i) {
            if (i) {
                out << ' ';
            }
            out << item.a[i];
        }
        return out;
    }

    double mse() {
        double res = 0;
        for (size_t i = 0; i < n; ++i) {
            res += std::pow(get(x[i]) - y[i], 2.0);
        }
        return res;
    }

    double operator()(double x0) { return get(x0); }
};

#endif /* MINIMAL_SQUARE_HPP */
```

4 Численное дифференцирование

4.1 Постановка задачи

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

4.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/2.in
5
0.0 0.5 1.0 1.5 2.0
0.0 0.97943 1.8415 2.4975 2.9093
1.0
$ ./solution <tests/2.in
Первая производная функции в точке x0 = 1.0000,f'(x0) = 1.5181
Вторая производная функции в точке x0 = 1.0000,f''(x0) = -0.8243
```

4.3 Исходный код

```
#ifndef  TABLE_FUNCTION_HPP
#define  TABLE_FUNCTION_HPP

#include <exception>
#include <vector>

const double EPS = 1e-9;

bool leq(double a, double b) { return (a < b) or (std::abs(b - a) < EPS); }

class table_function_t {
    using vec = std::vector<double>;
    size_t n;
    vec x;
    vec y;

  public:
    table_function_t(const vec& _x, const vec& _y) {
        if (_x.size() != _y.size()) {
            throw std::invalid_argument("Sizes does not match");
        }
        x = _x;
        y = _y;
        n = x.size();
    }

    double derivative1(double x0) {
        for (size_t i = 0; i < n - 2; ++i) {
            /* x in (x_i, x_i+1] */
            if (x[i] < x0 and leq(x0, x[i + 1])) {
                double dydx1 = (y[i + 1] - y[i + 0]) / (x[i + 1] - x[i + 0]);
                double dydx2 = (y[i + 2] - y[i + 1]) / (x[i + 2] - x[i + 1]);
                double res = dydx1 + (dydx2 - dydx1) *
                                     (2.0 * x0 - x[i] - x[i + 1]) /
                                     (x[i + 2] - x[i]);
                return res;
            }
        }
        return NAN;
    }

    double derivative2(double x0) {
        for (size_t i = 0; i < n - 2; ++i) {
            /* x in (x_i, x_i+1] */
            if (x[i] < x0 and leq(x0, x[i + 1])) {
                double dydx1 = (y[i + 1] - y[i + 0]) / (x[i + 1] - x[i + 0]);
                double dydx2 = (y[i + 2] - y[i + 1]) / (x[i + 2] - x[i + 1]);
```

```
                double res = 2.0 * (dydx2 - dydx1) / (x[i + 2] - x[i]);
                return res;
            }
        }

        return NAN;
    }
};

#endif /* TABLE_FUNCTION_HPP */
```

5 Численное интегрирование

5.1 Постановка задачи

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

5.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/1.in
0 2
0.5 0.25
$ ./solution <tests/1.in
Метод прямоугольников с шагом 0.5: 0.143739
Метод трапеций с шагом 0.5: 0.148748
Метод Симпсона с шагом 0.5: 0.145408

Метод прямоугольников с шагом 0.25: 0.144993
Метод трапеций с шагом 0.25: 0.146243
Метод Симпсона с шагом 0.25: 0.145409

Погрешность вычислений методом прямоугольников: 0.00167215
Погрешность вычислений методом трапеций: -0.0033396
Погрешность вычислений методом Симпсона: 1.56688e-06
```

5.3 Исходный код

```cpp
#ifndef INTEGRATE_HPP
#define INTEGRATE_HPP

#include <cmath>

#include "../lab3_1/interpolator.hpp"

using func = double(double);

double integrate_rect(double l, double r, double h, func f) {
    double x1 = l;
    double x2 = l + h;
    double res = 0;
    while (x1 < r) {
        res += h * f((x1 + x2) * 0.5);
        x1 = x2;
        x2 += h;
    }
    return res;
}

double integrate_trap(double l, double r, double h, func f) {
    double x1 = l;
    double x2 = l + h;
    double res = 0;
    while (x1 < r) {
        res += h * (f(x1) + f(x2));
        x1 = x2;
        x2 += h;
    }
    return res * 0.5;
}

using vec = std::vector<double>;

double integrate_simp(double l, double r, double h, func f) {
    double x1 = l;
    double x2 = l + h;
    double res = 0;
    while (x1 < r) {
        vec x = {x1, (x1 + x2) * 0.5, x2};
        vec y = {f(x[0]), f(x[1]), f(x[2])};
        inter_lagrange lagr(x, y);
        res += lagr().integrate(x1, x2);
        x1 = x2;
        x2 += h;
    }

}
return res;
inline double runge_romberg(double Fh, double Fkh, double k, double p) {
    return (Fh - Fkh) / (std::pow(k, p) - 1.0);
}
```

```

#endif /* INTEGRATE_HPP */
```