МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационыые технологиии прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №3
по курсу «Численные методы»

Студент: Д. Д. Наумов
Преподаватель: И. А. Иванов
Группа: М8О-406Б-17
Дата:
Оценка:
Подпись:

Москва, 2021

Часть 1

Задание

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^*.

Вариант: 3

$$y = tg(x)$$

1. $X = \left\{0, \frac{\pi}{8}, \frac{2\pi}{8}, \frac{3\pi}{8}\right\}$
2. $X = \left\{0, \frac{\pi}{8}, \frac{\pi}{3}, \frac{3\pi}{8}\right\}$

$$X^* = \frac{3\pi}{16}$$

Теория

Задача интерполяции – найти функцию $F(x)$, принимающую в точках x_i те же значения y_i. Тогда, условие интерполяции:

$$F(x_i) = y_i$$

При этом предполагается, что среди значений x_i нет одинаковых.

Точки x_i называют узлами интерполяции.

Многочлен Лагранжа

При глобальной интерполяции на всем интервале $[a, b]$ строится единый многочлен. Одной из форм записи интерполяционного многочлена для глобальной интерполяции является многочлен Лагранжа:

$$L_n(x) = \sum_{i=0}^{n} y_i l_i(x)$$

где $l_i(x)$ базисные многочлены степени n:

$$l_i = \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} = \frac{(x - x_0)(x - x_1) \dots (x - x_{i-1})(x - x_{i+1}) \dots (x - x_n)}{(x_i - x_0)(x_i - x_1) \dots (x_i - x_{i-1})(x_i - x_{i+1}) \dots (x_i - x_n)}$$

Многочлен $l_i(x_j)$ удовлетворяет условию

$$l_i(x_j) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$$

Это условие означает, что многочлен равен нулю при каждом x_j кроме x_i, то есть $x_0, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n$ – корни этого многочлена. Таким образом, степень многочлена $L_n(x)$ равна n и при $x \neq x_i$ обращаются в ноль все слагаемые суммы, кроме слагаемого с номером $i = j$, равного y_i.

Многочлен Ньютона

Другая форма записи интерполяционного многочлена – интерполяционный многочлен Ньютона с разделенными разностями. Пусть функция $f(x)$ задана с произвольным шагом, и точки таблицы значений пронумерованы в произвольном порядке.

Разделенные разности нулевого порядка совпадают со значениями функции в узлах. Разделенные разности первого порядка определяются через разделенные разности нулевого порядка:

$$f(x_i, x_{i+1}) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}$$

Разделенные разности второго порядка определяются через разделенные разности первого порядка:

$$f(x_i, x_{i+1}, x_{i+2}) = \frac{f(x_{i+1}, x_{x+2}) - f(x_i, x_{i+1})}{x_{i+2} - x_i}$$

Разделенные разности k-го порядка определяются через разделенные разности порядка $k-1$:

$$f(x_i, x_{i+1}, \ldots, x_{i+k}) = \frac{f(x_{i+1}, \ldots, x_{x+k}) - f(x_i, \ldots, x_{i+k-1})}{x_{i+k} - x_i}$$

Используя понятие разделенной разности интерполяционный многочлен Ньютона можно записать в следующем виде:

$$\begin{aligned} P_n(x) = f(x_0) + f(x_0, x_1) \cdot (x - x_0) + f(x_0, x_1, x_2) \cdot (x - x_0) \cdot (x - x_1) + \\ f(x_0, x_1, \ldots, x_n) \cdot (x - x_0) \cdot (x - x_1) \cdot \ldots \cdot (x - x_{n-1}) \end{aligned}$$

За точностью расчета можно следить по убыванию членов суммы. Если функция достаточно гладкая, то справедливо приближенное равенство

$$f(x) - P_n(x) \approx P_{n+1}(x) - P_n(x)$$

Это приближенное равенство можно использовать для практической оценки погрешности интерполяции:

$$\varepsilon_n = |P_{n+1}(x) - P_n|$$

Реализация

Полином Лагранжа

```
    vdouble CoefPolynomLagrange (const vdouble& x, const vdouble& y) {
        int n = x.size();
        vdouble w(n, 1);
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (j != i) {
                    w[i] *= x[i] - x[j];
                }
            }
            w[i] = y[i] / w[i];
        }
        return w;
    }

    double PolynomLagrange(const vdouble& x, const vdouble& y, double x0) {
        int n = x.size();
        vdouble coef = CoefPolynomLagrange(x, y);
        double y0 = 0;
        for (int i = 0; i < n; ++i) {
            double p = 1;
            for (int j = 0; j < n; ++j) {
                if (i != j) {
                    p *= x0 - x[j];
                }
            }
            y0 += p * coef[i];
        }

        return y0;
    }
```

Полином Ньютона

```
class PolynomNewton {
public:

    PolynomNewton (const vdouble& x, const vdouble& y) {
        ft = vvdouble(x.size(), vdouble());
        X = x;
        int n = x.size();
        for (int i = 0; i < n; ++i) {
            ft[i].push_back(y[i]);
        }
        for (int j = 0; j < n - 1; ++j) {
```

```
12              for (int i = 0; i < n - j - 1; ++i) {
13                  double f = ft[i][j] - ft[i + 1][j];
14                  f /= x[i] - x[i + j + 1];
15                  ft[i].push_back(f);
16              }
17          }
18      }
19
20      void AddPoint(double x, double y) {
21          int n = X.size();
22          X.push_back(x);
23          ft.push_back(vdouble());
24          ft[n].push_back(y);
25          for (int i = n; i > 0; --i) {
26              double f = ft[i - 1][n - i] - ft[i][n - i];
27              f /= X[i - 1] - X[n];
28              ft[i - 1].push_back(f);
29          }
30      }
31
32      const vvdouble& GetFT() const {
33          return ft;
34      }
35
36      double operator() (double x0) const {
37          int n = X.size();
38          double y0 = 0;
39          double p = 1;
40          for (int i = 0; i < n; ++i) {
41              y0 += ft[0][i] * p;
42              p *= (x0 - X[i]);
43
44          }
45          return y0;
46      }
47  private:
48      vvdouble ft;
49      vdouble X;
50      vdouble Y;
51  };
```

Результаты

Пример работы программы

```
(base) MacBook-Air-Dima:Program dandachok$ ./a.out
X*: 0.589049
Real: 0.668179

Test case 1: [0, 0.3927, 0.7854, 1.178]
Lagrange method: 0.6446, Error: 0.02357
Newton method: 0.6446, Error: 0.02357

Test case 2: [0, 0.3927, 1.047, 1.178]
Lagrange method: 0.5853, Error: 0.08293
Newton method: 0.5853, Error: 0.08293
```

Часть 2

Задание

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: 3

$X^* = 1.5$

i	0	1	2	3	4
x_i	0.0	0.9	1.8	2.7	3.6
f_i	0.0	0.36892	0.85408	1.7856	6.3138

Теория

Использование одной интерполяционной формулы на большом числе узлов нецелесообразно. Интерполяционный многочлен может проявить свои колебательные свойства, его значения между узлами могут сильно отличаться от значений интерполируемой функции. Одна из возможностей преодоления этого недостатка заключается в применении сплайн-интерполяции. Суть сплайн-интерполяции заключается в определении интерполирующей функции по формулам одного типа для различных непересекающихся промежутков и в стыковке значений функции и её производных на их границах.

Наиболее широко применяемым является случай, когда между любыми двумя точками разбиения исходного отрезка строится многочлен n-й степени:

$$S(x) = \sum_{k=0}^{n} a_{ik} x^k, \qquad x_{i-1} \leqslant x \leqslant x_i, \quad i = 1, \ldots, n$$

который в узлах интерполяции принимает значения аппроксимируемой функции и непрерывен вместе со своими $(n-1)$ производными. Такой кусочно-непрерывный интерполяционный многочлен называется сплайном. Его коэффициенты находятся из условий равенства в узлах сетки значений сплайна и приближаемой функции, а также равенства $n-1$ производных соответствующих многочленов. На практике наиболеечасто используется интерполяционный многочлен третьей степени, который удобно представить, как

$$S(x) = a_i + b_i(x - x_{i-1}) + c_i(x - x_{i-1})^2 + d_i(x - x_{i-1})^3$$

где $x_{i-1} \leqslant x \leqslant x_i, \quad i = 1, \ldots, n$

Для построения кубического сплайна необходимо построить n многочленов третьей степени, т.е. определить $4n$ неизвестных a,b,c,d. Эти коэффициенты ищутся из условий в узлах сетки.

$S(x_{i-1}) = a_i = a_{i-1} + b_{i-1}(x_{i-1} - x_{i-2}) + c_{i-1}(x_{i-1} - x_{i-2})^2 + d_{i-1}(x_{i-1} - x_{i-2})^3 = f_{i-1}$
$S'(x_{i-1}) = b_i = b_{i-1} + 2c_{i-1}(x_{i-1} - x_{i-2}) + 3d_{i-1}(x_{i-1} - x_{i-2})^2$
$S''(x) = 2c_i = 2c_{i-1} + 6d_{i-1}(x_{i-1} - x_{i-2})$
$S(x_0) = a_1 = f_0$
$S''(x_0) = c_1 = 0$
$S(x_n) = a_n + b_n(x_n - x_{i-1}) + c_n(x_n - x_{n-1})^2 + d_n(x_n - x_{n-1})^3 = f_n$
$S''(x_n) = c_n + 3d_n(x_n - x_{n-1}) = 0$

Предполагается, что сплайны имеют нулевую кривизну на концах отрезка. В общем случае могут быть использованы и другие условия.

Если ввести обозначение $h_i = x_i - x_{i-1}$, и исключить из системы a_i,b_i,d_i, то можно получить систему из $n-1$ линейных алгебраических уравнений относительно c_i , $i = 2, \ldots, n$ с трехдиагональной матрицей:

$$
\begin{aligned}
&2(h_1 + h_2)c_2 + h_2 c_3 = 3\left(\frac{f_2 - f_1}{h_2} - \frac{f_1 - f_0}{h_1}\right) \\
&h_{i-1}c_{i-1} + 2(h_{i-1} + h_i)c_i + h_i c_{i+1} = 3\left(\frac{f_i - f_{i-1}}{h_i} - \frac{f_{i-1} - f_{i-2}}{h_{i-1}}\right) \qquad i = 3, \ldots, n-1 \\
&h_{n-1}c_{n-1} + 2(h_{n-1} + h_i)c_n + = 3\left(\frac{f_n - f_{n-1}}{h_n} - \frac{f_{n-1} - f_{n-2}}{h_{n-1}}\right)
\end{aligned}
$$

Остальные коэффициенты сплайнов могут быть восстановлены по формулам:

$$
\begin{aligned}
&a_i = f_{i-1} && i = 1, \ldots, n \\
&b_i = \frac{f_i - f_{i-1}}{h_i} - \frac{1}{3}h_i(c_{i+1} + 2c_i) && i = 1, \ldots, n-1 \\
&d_i = \frac{c_{i+1} - c_i}{3h_i} && i = 1, \ldots, n-1 \\
&c_1 = 0 \\
&b_n = \frac{f_n - f_{n-1}}{h_n} - \frac{2}{3}h_n c_n \\
&d_n = -\frac{c_n}{3h_n}
\end{aligned}
$$

Реализация

```
class CubicSplines {
public:
    CubicSplines (vdouble x, vdouble y) :
    x(x) {
        int n = x.size();
        c = SweepMethod(GetSweepMatrix(x, y));
        c.insert(c.begin(), 0);
        a = y;
        a.pop_back();
        b.resize(n - 1);
        d.resize(n - 1);
        for (int i = 0; i < n - 2; ++i) {
            double h = (x[i + 1] - x[i]);
            b[i] = (y[i + 1] - y[i]) / h;
            b[i] -= h * (c[i + 1] + 2*c[i]) / 3;
            d[i] = (c[i + 1] - c[i]) / h / 3;
        }
        double h = x[n - 1] - x[n - 2];
        b[n - 2] = (y[n - 1] - y[n - 2]) / h - 2*h*c[n - 2] / 3;
        d[n - 2] = - c[n - 2] / h / 3;
    }

    double operator() (double x0) {
        double y0;
        int i;
        for (i = 0; i < x.size() - 1; ++i) {
            if (x[i] <= x0 && x0 <= x[i + 1]) {
                break;
            }
        }
        double dx = x0 - x[i];
        y0 = a[i] + b[i]*dx + c[i]*pow(dx, 2) + d[i]*pow(dx, 3);
        return y0;
    }
private:

    vvdouble GetSweepMatrix (const vdouble& x, const vdouble& y) {
        int n = x.size();
        vvdouble m(n - 2, vdouble(4));
        vdouble h = GetH(x);
        m[0][0] = 0;
        m[0][1] = 2 * (h[1] + h[2]);
        m[0][2] = h[2];
        m[0][3] = GetD(y, h, 0);

        for (int i = 1; i < n - 2; ++i) {
            m[i][0] = h[i + 1];
```

```
48              m[i][1] = 2 * (h[i + 1] + h[i + 2]);
49              m[i][2] = h[i + 2];
50              m[i][3] = GetD(y, h, i);
51          }
52          m[n - 3][2] = 0;
53          return m;
54      }
55
56      vdouble GetH (const vdouble& x) {
57          int n = x.size();
58          vdouble h(x.size());
59          h[0] = 1;
60          for (int i = 1; i < n; ++i) {
61              h[i] = x[i] - x[i - 1];
62          }
63          return h;
64      }
65
66      double GetD (const vdouble& y, const vdouble& h, int i) {
67          double res = (y[i + 2] - y[i + 1]) / h[i + 2];
68          res -= (y[i + 1] - y[i]) / h[i + 1];
69          return 3*res;
70      }
71
72      vdouble x;
73      vdouble a;
74      vdouble b;
75      vdouble c;
76      vdouble d;
77  };
```

Результаты

```
1
2 (base) MacBook-Air-Dima:Program dandachok$ ./part2.out
3 Cubic spline in X* = 1.5: 0.724543
```

Часть 3

Задание

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: 3

i	0	1	2	3	4	5
x_i	-0.9	0.0	0.9	1.8	2.7	3.6
y_i	-0.36892	0.0	0.36892	0.85408	1.7856	6.3138

Теория

Метод наименших квадратов

Пусть задана таблично в узлах x_j функция $y_j = f(x_j), \quad j = 0, 1, \ldots, N$. При этом значения функции y_j определены с некоторой погрешностью, также из физических соображений известен вид функции, которой должны приближенно удовлетворять табличные точки, например: многочлен степени n, у которого неизвестны коэффициенты a_i, $F_n(x) = \sum_{i=0}^{n} a_i x^i$. Неизвестные коэффициенты будем находить из условия минимума $i = 0$квадратичного отклонения многочлена от таблично заданной функции.

$$\Phi = \sum_{j=0}^{N} [F_n(x_j) - y_j]^2$$

Минимума Ф можно добиться только за счет изменения коэффициентов многочлена $F_n(x)$. Необходимые условия экстремума имеют вид

$$\frac{\partial \Phi}{\partial a_k} = 2 \sum_{j=0}^{N} \left[\sum_{i=0}^{n} a_i x_j^i - y_j \right] x_j^k = 0, \qquad k = 0, 1, \ldots, n$$

Эту систему для удобства преобразуют к следующему виду:

$$\sum_{i=0}^{n} a_i \sum_{j=0}^{N} x_j^{k+i} = \sum_{j=0}^{N} y_j x_j^k, \qquad k = 0, 1, \ldots, n$$

Такая система называется нормальной системой метода наименьших квадратов (МНК) представляет собой систему линейных алгебраических уравнений относительно коэффициентов a_i. Решив систему, построим многочлен $F_n(x)$, приближающий функцию $f(x)$ и минимизирующий квадратичное отклонение.

Реализация

```
vdouble SimpleIter (const vvdouble& m, vdouble b, double eps) {
    int n = m.size();
    vvdouble a = m;
    for (int i = 0; i < n; ++i) {
        b[i] /= m[i][i];
        for (int j = 0; j < n; ++j) {
            a[i][j] = i == j ? 0: -m[i][j] / m[i][i];
        }
    }
    vdouble x = b;
    vdouble prev_x;
    int count_iter = 0;
    do {
        count_iter++;
        prev_x = x;
        x = b + a * prev_x;
    } while (Norma(x - prev_x) > eps);
    std::cout << "Simple method iters count: " << count_iter << '\n';
    return x;
}

vdouble ZeidelMethod (const vvdouble& m, vdouble b, double eps) {
    int n = m.size();
    vvdouble a = m;
    for (int i = 0; i < n; ++i) {
        b[i] /= m[i][i];
        for (int j = 0; j < n; ++j) {
            a[i][j] = i == j ? 0: -m[i][j] / m[i][i];
        }
    }
    vdouble x = b;
    vdouble prev_x;
    double count_iter = 0;
    do {
        count_iter++;
        prev_x = x;
        for (int i = 0; i < n; ++i) {
            x[i] = b[i];
            for (int j = 0; j < n; ++j) {
                x[i] += x[j] * a[i][j];
            }
        }
    } while (Norma(x - prev_x) > eps);
    std::cout << "Zeidel method iter count: " << count_iter << '\n';
    return x;
}
```

Результаты

```
(base) MacBook-Air-Dima:Program dandachok$ cat tests/input/part3.t
4 4
-23 -7 5 2
-7 -21 4 9
9 5 -31 -8
0 1 -2 10

-26 -55 -58 -24

0.01
(base) MacBook-Air-Dima:Program dandachok$ ./part3 <tests/input/part3.t
Simple method iters count: 7
Zeidel method iter count: 5
Simple ans:
[0.9998, 2, 3, -2]
Zeidel ans:
[0.9999, 2, 3, -2]
(base) MacBook-Air-Dima:Program dandachok$
```

С помощью метода простых итераций ответ был получен за 7 шагов($\varepsilon = 0.01$):

$$x = \begin{pmatrix} 0.9998 \\ 2 \\ 3 \\ -2 \end{pmatrix}$$

С помощью улучшения Зейделя ответ был получен за 5 шагов($\varepsilon = 0.01$):

$$x = \begin{pmatrix} 0.9999 \\ 2 \\ 3 \\ -2 \end{pmatrix}$$

Как и ожидалось метод Зейдаля за меньшее число итераций доходит до заданной точности.

Часть 4

Задание

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

Вариант: 3

$X^* = 2.0$

i	0	1	2	3	4
x_i	1.0	1.5	2.0	2.5	3.0
y_i	0.0	0.40547	0.69315	0.91629	1.0986

Теория

Формулы численного дифференцирования в основном используются при нахождении производных от функции, заданной таблично. Исходная функция, заменяется некоторой приближающей, легко вычисляемой функцией. Наиболее часто в качестве приближающей функции берется интерполяционный многочлен, а производные соответствующих порядков определяются дифференцированием многочлена. При решении практических задач, как правило, используются аппроксимации первых и вторых производных.

В первом приближении, таблично заданная функция может быть аппроксимирована отрезками прямой

$$y(x) \approx \varphi(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i), \qquad x \in [x_i, x_{i+1}]$$

Тогда

$$y'(x) \approx \varphi'(x) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} = const, \qquad x \in [x_i, x_{i+1}]$$

производная является кусочно-постоянной функцией и рассчитывается с первым порядком точности в крайних точках интервала, и со вторым порядком точности в средней точке интервала.

Для вычисления второй производной, необходимо использовать интерполяционный многочлен, как минимум второй степени. После дифференцирования многочлена получаем

$$y''(x) \approx \varphi''(x) = 2\frac{\dfrac{y_{i+2} - y_{i+1}}{x_{i+2} - x_{i+1}} - \dfrac{y_{i+1} - y_i}{x_{i+1} - x_i}}{x_{i+2} - x_i}, \qquad x \in [x_i, x_{i+1}]$$

Реализация

```
vdouble JakobiMethod (const vvdouble& mat, double eps) {
    int n = mat.size();
    int im;
    int jm;
    vvdouble a = mat;
    vvdouble r = CreateIdentity(n);
    for (int i = 0; SqrtSumNDElem(a) > eps; ++i) {

        FindMaxNDElem(a, im, jm); //find max not diagonal elem
        double q;
        if (a[im][im] == a[jm][jm]) {
            q = pi / 4;
        } else {
            q = 0.5 * atan (2*a[im][jm] / (a[im][im] - a[jm][jm]));
        }

        vvdouble u = CreateIdentity(n);
        u[im][im] = cos(q);
        u[im][jm] = -sin(q);
        u[jm][im] = sin(q);
        u[jm][jm] = cos(q);
        vvdouble ut = Trans(u);

        r = r*u;
        a = (ut * a) * u;

    }

    for (int i = 0; i < n; ++i) {
        std::cout << "x" << i << ": " << r[i];
    }
    std::cout << '\n';
    vdouble res;
    for (int i = 0; i < n; ++i) {
        res.push_back(a[i][i]);
    }
    return res;
}
```

Результаты

Пример работы программы

```
(base) MacBook-Air-Dima:Program dandachok$ cat tests/input/part4.t
3

5 5 3
5 -4 1
3 1 2

0.3
(base) MacBook-Air-Dima:Program dandachok$ ./part4 <tests/input/part4.t
x0: [0.8306, -0.4152, -0.3712]
x1: [0.3604, 0.9088, -0.2103]
x2: [0.4246, 0.04089, 0.9045]

[8.705, -6.239, 0.5339]
(base) MacBook-Air-Dima:Program dandachok$
```

Для матрицы $A = \begin{pmatrix} 5 & 5 & 3 \\ 5 & -4 & 1 \\ 3 & 1 & 2 \end{pmatrix}$ найдены собственные значения

$$\lambda_1 = 8.705, \quad \lambda_2 = -6.239, \quad \lambda_3 = 0.5339$$

и следующие собственные векторы

$$x_1 = \begin{pmatrix} 0.8306 \\ -0.4152 \\ -0.3712 \end{pmatrix}, \quad x_2 = \begin{pmatrix} 0.3604 \\ 0.9088 \\ -0.2103 \end{pmatrix}, \quad x_3 = \begin{pmatrix} 0.4246 \\ 0.04089 \\ 0.9045 \end{pmatrix},$$

Часть 5

Задание

Вычислить определенный интеграл $F = \int\limits_{x_0}^{x_1} y\,dx$, методами прямоугольников, трапеций, Симпсона с шагами h_1, h_2. Оценить погрешность вычислений, используя Метод Рунге-Ромберга:

Вариант: 3

$$y = \frac{x}{(3x+4)^3}, \qquad X_0 = -1, \quad X_k = 1, \quad h_1 = 0.5, \quad h_2 = 0.25$$

Теория

Формулы численного интегрирования используются в тех случаях, когда вычислить аналитически определенный интеграл не удается. Отрезок разбивают точками с достаточно мелким шагом и на одном или нескольких отрезках подынтегральную функцию заменяют такой приближающей так что она, во-первых, близка, а, во-вторых, интеграл от неё легко вычисляется. В нашем случае будем использовать интерполяционный многочлен, при чем коэффициенты различны на каждом отрезке.

$$f(x) = P_n(x, \overline{a}_i) + R_n(x, \overline{a}_i), \qquad x \in [x_i, x_{i+k}]$$

где R_n - остаточный член интерполяции.
Тогда

$$F = \sum_{i=1}^{N} \int_{x_i-1}^{x_i} P_n(x, \overline{a}_i)\, dx + R$$

где $R = \sum_{i=1}^{N} \int_{x_i-1}^{x_i} R_n(x, \overline{a}_i)\, dx$ - остаточный член формулы численного интегрирования или её погрешность.

Заменим подынтегральную функцию, интерполяционным многочленом Лагранжа нулевой степени, проходящим через середину отрезка, получим формулу прямоугольников.

$$\int_a^b f(x)\, dx \approx \sum_{i=1}^{N} h_i f\left(\frac{x_{i-1-x_i}}{2}\right)$$

В случае таблично заданных функций удобно в качестве узлов интерполяции выбрать начало и конец отрезка интегрирования, т.е. заменить функцию многочленом Лагранжа первой степени.

$$F = \int_a^b f(x)\, dx \approx \frac{1}{2} \sum_{i=1}^{N} (f_{i-1} + f_i) h_i$$

Эта формула носит название формулы трапеций.

Для повышения порядка точности формулы численного интегрирования заменим подынтегральную кривую параболой – интерполяционным многочленом второй степени, выбрав в качестве узлов интерполяции концы и середину отрезка интегрирования.

Для случая $h_i = \dfrac{x_i - x_{i-1}}{2}$, получим формулу Симпсона(парабол):

$$F = \int_a^b f(x)\,dx \approx \frac{1}{3}\sum_{i=1}^{N}(f_{i-1} + 4f_{i-\frac{1}{2}} + f_i)h_i$$

Метод Рунге-Ромберга-Ричардсона позволяет получать более высокий порядок точности вычисления. Если имеются результаты вычисления определённого интеграла на сетке с шагом $h - F = F_h + O(h^p)$ и на сетке с шагом $kh - F = F_{kh} + O((kh)^p)$, то

$$F = \int_a^b f(x)\,dx = F_h + \frac{F_h - F_{hk}}{k^p - 1} + O(h^{p+1})$$

Реализация

```
vvdouble HausseholderMatrix (const vvdouble& v) {
    return CreateIdentity(v.size()) - 2 / (Trans(v) * v)[0][0] * (v * Trans(v));
}

void QRDecomposition (vvdouble a, vvdouble& q, vvdouble& r) {
    int n = a.size();
    q = CreateIdentity(n);
    for (int i = 0; i < n - 1; ++i) {
        vvdouble b = GetColumn(a, i);
        vvdouble h = HausseholderMatrix(b);
        a = h * a;
        q = q * h;
    }
    r = a;
}

vvdouble QRMethod (vvdouble a, double eps) {
    int n = a.size();
    vvdouble q, r, an = a;
    vcomplex l(n);
    int i = 0;
    do {
        std::swap(a, an);
        QRDecomposition(a, q, r);
        an = r * q;
    } while (!FinishIterProc(an, a, eps));
    return an;
}
```

Результаты

Пример работы программы

```
(base) MacBook-Air-Dima:Program dandachok$ cat tests/input/part5.t
3

5 -5 -6
-1 -8 -5
2 7 -3

0.1
(base) MacBook-Air-Dima:Program dandachok$ ./part5 <tests/input/part5.t
QR Method iter count: 41
A:
{[-7.358, 5.286, -8.236]
[-7.147, -3.473, 1.266]
[-1.089e-07, 1.565e-08, 4.832]}
(base) MacBook-Air-Dima:Program dandachok$
```

В результате получилась матрица:

$$A^{(41)} = \begin{pmatrix} -7.358 & 5.286 & -8.236 \\ -7.147 & -3.473 & 1.266 \\ -1.08e-7 & 1.6e-8 & 4.832 \end{pmatrix}$$

Видно, что поддиагональные элементы достаточно малые, в то же время отчетливо прослеживается комплексно-сопряженная пара собственных значений, соответствующая блоку, образуемому элементами первого и второго столбцов. Несмотря на значительное изменение в ходе итераций самих этих элементов, собственные значения, соответствующие данному блоку и определяемые из решения квадратного уравнения $(a_{11} - \lambda)(a_{22} - \lambda) = a_{12}a_{21}$

После решения уравнения получаются следующие собстенные значения:

$$\lambda_1 \approx -5.4155 + 5.83i, \quad \lambda_2 \approx -5.4155 - 5.83i, \quad \lambda_3 \approx 4.832$$