МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

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

Лабораторная работа №4
по курсу «Численные методы»

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

Москва, 2021

Часть 1

Задание

Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки h. С использованием разработанного программного обеспечения решить задачу Коши для ОДУ 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

Вариант: 3

Задача Коши:

$$\begin{cases} y'' - 2y - 4x^2 e^{x^2} = 0 \\ y(0) = 3 \\ y'(0) = 0 \end{cases} \qquad x \in [0, 1], \quad h = 0.1$$

Точное решение:

$$y = e^{x^2} + e^{x\sqrt{2}} + e^{-x\sqrt{2}}$$

Теория

Рассматривается задача Коши для одного дифференциального уравнения первого порядка, разрешенного относительно производной

$$\begin{cases} y' = f(x) \\ y(x_0) = y_0 \end{cases}$$

Требуется найти решение на отрезке $[a, b]$, где $x_0 = a$
Введем разностную сетку на отрезке $[a, b]$:

$$\Omega = x_k = x_0 + kh, \qquad k = 0, 1, \ldots, N, \quad h = |b - a|/N$$

Формула метода Эйлера:

$$y_{k+1} = y_k + hf(x_k, y_k)$$

Все рассмотренные выше явные методы являются вариантами методов Рунге-Кутты. Семейство явных методов Рунге-Кутты p-го порядка записывается в виде совокупности формул:

$$y_{k+1} = y_k + \Delta y$$

$$\Delta y_i = \sum_{j=1}^{p} c_j K_j^i$$

$$K_i^k = hf\left(x_k + a_i h, y_k + h\sum_{j=1}^{i} b_{ij} K_j^k\right), \qquad i = 2, 3, \ldots, p$$

Метод Рунге-Кутты четвертого порядка точности

$$y_{k+1} = y_k + \Delta y$$

$$\Delta y_k = \frac{1}{6}(K_1^k + 2K_2^k + 2K_3^k + K_4^k)$$

$$K_1^k = hf\left(x_k, y_k\right)$$
$$K_2^k = hf\left(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_1^k\right)$$
$$K_3^k = hf\left(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_2^k\right)$$
$$K_4^k = hf\left(x_k + h, y_k + K_3^k\right)$$

Рассматривается задача Коши для системы дифференциальных уравнений первого порядка разрешенных относительно производной

$$\begin{cases} y_1' = f_1(x, y_1, y_2, \ldots, y_n) \\ y_2' = f_2(x, y_1, y_2, \ldots, y_n) \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ y_n' = f_n(x, y_1, y_2, \ldots, y_n) \end{cases}$$

$$y_1(x_0) = y_{01}$$
$$y_2(x_0) = y_{02}$$
$$\ldots\ldots\ldots\ldots$$
$$y_n(x_0) = y_{0n}$$

Формулы метода Рунге-Кутты 4-го порядка точности для решения системы следующие

$$y_{k+1} = y_k + \Delta y$$
$$z_{k+1} = z_k + \Delta z$$

$$\Delta y_k = \frac{1}{6}(K_1^k + 2K_2^k + 2K_3^k + K_4^k)$$
$$\Delta z_k = \frac{1}{6}(L_1^k + 2L_2^k + 2L_3^k + L_4^k)$$

$$
\begin{aligned}
K_1^k &= hf\left(x_k, y_k, z_k\right) \\
L_1^k &= hg\left(x_k, y_k, z_k\right) \\
K_2^k &= hf\left(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_1^k, z_k + \frac{1}{2}L_1^k\right) \\
L_2^k &= hg\left(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_1^k, z_k + \frac{1}{2}L_1^k\right) \\
K_3^k &= hf\left(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_2^k, z_k + \frac{1}{2}L_2^k\right) \\
L_3^k &= hg\left(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_2^k, z_k + \frac{1}{2}L_2^k\right) \\
K_4^k &= hf\left(x_k + h, y_k + K_3^k, z_k + L_3^k\right) \\
L_4^k &= hg\left(x_k + h, y_k + K_3^k, z_k + L_3^k\right)
\end{aligned}
$$

При использовании интерполяционного многочлена 3-ей степени построенного по значениям подынтегральной функции в последних четырех узлах получим метод Адамса четвертого порядка точности:

$$y_{k+1} = y_k + \frac{h}{24}\left(55f_k - 59f_{k-1} + 37f_{k-2} - 9f_{k-3}\right)$$

Метод Адамса как и все многошаговые методы не является самостартующим, то есть для того, что бы использовать метод Адамса необходимо иметь решения в первых четырех узлах. В узле решение известно из начальных условий, а в других трех узлах решения можно получить с помощью подходящего одношагового метода, например: метода Рунге-Кутты четвертого порядка.

Реализация

Метод Эйлера

```
class EilerMethod2 {
public:
    EilerMethod2(Function3* f, double left, double right, double y0, double z0) :
        left(left),
        right(right),
        f(f),
        y0(y0),
        z0(z0) {
    }

    void operator() (double h) {
        x = GetValInRange(left, right, h);
        y.clear();
        z.clear();
        y.push_back(y0);
        z.push_back(z0);

        int n = x.size() - 1;
        for (int i = 0; i < n; ++i) {
            double z_next = z[i] + h * (*f)(x[i], y[i], z[i]);
            z.push_back(z_next);
            double y_next = y[i] + h * z[i];
            y.push_back(y_next);
        }
    }

    vdouble GetY() {
        return y;
    }

    vdouble GetZ() {
        return z;
    }


private:
    double left;
    double right;
    Function3* f;
    double h;
    double y0;
    double z0;

    vdouble x;
    vdouble y;
```

```
46          vdouble z;
47  };
```

Метод Рунге-Кутты

```
1       class RungeKutta2 {
2       public:
3           RungeKutta2(CauchyProblem t) :
4               left(t.left),
5               right(t.right),
6               f(t.f),
7               y0(t.y0),
8               z0(t.z0) {
9           }
10
11          vdouble operator() (double h) {
12              x = GetValInRange(left, right, h);
13              int n = x.size();
14              y.clear();
15              z.clear();
16              y.push_back(y0);
17              z.push_back(z0);
18
19              for (int i = 0; i < n - 1; ++i) {
20                  vdouble k(ORDER);
21                  vdouble l(ORDER);
22                  for (int j = 0; j < ORDER; ++j) {
23                      if (j == 0) {
24                          l[j] = h * (*f)(x[i], y[i], z[i]);
25                          k[j] = h * z[i];
26                      }
27                      else if (j == 3) {
28                          l[j] = h * (*f)(x[i] + h, y[i] + k[j - 1], z[i] + l[j - 1]);
29                          k[j] = h * (z[i] + l[j - 1]);
30                      } else {
31                          l[j] = h * (*f)(x[i] + 0.5*h, y[i] + 0.5*k[j - 1], z[i] + 0.5*l[j
                                - 1]);
32                          k[j] = h * (z[i] + 0.5*l[j - 1]);
33                      }
34                  }
35
36                  double y_next = y[i] + dif(k);
37                  double z_next = z[i] + dif(l);
38                  y.push_back(y_next);
39                  z.push_back(z_next);
40              }
41
42              return y;
43          }
```

```
44
45          vdouble GetY() {
46              return y;
47          }
48
49          vdouble GetZ() {
50              return z;
51          }
52
53          vdouble Get() {
54              return y;
55          }
56
57          Function3* f;
58          double left;
59          double right;
60          double y0;
61          double z0;
62
63      private:
64          double dif (const vdouble& k) {
65              return (k[0] + 2*(k[1] + k[2]) + k[3]) / 6;
66          }
67
68          double g (double z) {
69              return z;
70          }
71
72          static const int ORDER = 4;
73
74          vdouble x;
75          vdouble y;
76          vdouble z;
77  };
```

Метод Адамса

```
1       class AdamsMethod {
2           public:
3               AdamsMethod(CauchyProblem t) :
4                   f(t.f),
5                   left(t.left),
6                   right(t.right),
7                   y0(t.y0),
8                   z0(t.z0),
9                   task(t) {}
10
11              void operator() (double h) {
12                  RungeKutta2 rk(task);
```

```
            rk(h);
            x = GetValInRange(left, right, h);
            y = rk.GetY();
            z = rk.GetZ();

            y.resize(4);
            z.resize(4);

            int n = x.size();
            for (int i = 3; i < n - 1; ++i) {
                double z_next = z[i] + h / 24 * df(i);
                z.push_back(z_next);
                double y_next = y[i] + h / 24 * dg(i);
                y.push_back(y_next);
            }
        }

        vdouble GetY() {
            return y;
        }

        vdouble GetZ() {
            return z;
        }
    private:

        double df(int i) {
            double f0 = (*f)(x[i], y[i], z[i]);
            double f1 = (*f)(x[i - 1], y[i - 1], z[i - 1]);
            double f2 = (*f)(x[i - 2], y[i - 2], z[i - 2]);
            double f3 = (*f)(x[i - 3], y[i - 3], z[i - 3]);

            return (55*f0 - 59*f1 + 37*f2 - 9*f3);
        }

        double dg(int i) {
            return (55*z[i] - 59*z[i - 1] + 37*z[i - 2] - 9*z[i - 3]);
        }

        CauchyProblem task;
        Function3* f;
        double left;
        double right;
        double y0;
        double z0;

        vdouble x;
        vdouble y;
        vdouble z;
```

```
62 ||      };
```

Результаты

Пример работы программы

```
(base) MacBook-Air-Dima:Program dandachok$ cat tests/input/part1.t
4
9 -5 -6 3
1 -7 1 0
3 -4 -9 0
6 -1 9 8

-8 38 47 -8
(base) MacBook-Air-Dima:Program dandachok$ ./part1 <tests/input/part1.t
L: {[1, 0, 0, 0]
[0.1111, 1, 0, 0]
[0.3333, 0.3621, 1, 0]
[0.6667, -0.3621, -1.789, 1]
}
U:{[9, -5, -6, 3]
[0, -6.444, 1.667, -0.3333]
[0, 0, -7.603, -0.8793]
[0, 0, 0, 4.306]
}
LU solve:[-15.98, -8.668, -6.697, 17.44]
Gausse solve:[-15.98, -8.668, -6.697, 17.44]
```

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0.1111 & 1 & 0 & 0 \\ 0.3333 & 0.3621 & 1 & 0 \\ 0.6667 & -0.3621 & -1.789 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 9 & -5 & -6 & 3 \\ 0 & -6.444 & 1.667 & -0.3333 \\ 0 & 0 & -7.603 & -0.8793 \\ 0 & 0 & 0 & 4.306 \end{pmatrix}$$

$$x = \begin{pmatrix} -8 \\ 38 \\ 47 \\ -8 \end{pmatrix}$$

Часть 2

Задание

Реализовать метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ в виде программ. С использованием разработанного программного обеспечения решить краевую задачу для обыкновенного дифференциального уравнения 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением

Вариант: 3

Задача Коши:

$$x^2(x+1)y'' - 2y = 0$$
$$y'(1) = -1$$
$$2y(2) - 4y'(2) = 4$$

Точное решение:

$$y(x) = \frac{1}{x} + 1$$

Теория

Метод стрельбы

Суть метода заключена в многократном решении задачи Коши для приближенного нахождения решения краевой задачи.

Пусть надо решить краевую задачу на отрезке. Вместо исходной задачи формулируется задача Коши с начальными условиями

$$y(a) = \eta$$
$$y'(b) = y_0$$

Задачу можно сформулировать таким образом: требуется найти такое значение переменной η, чтобы решение $y(a, y_0, \eta)$ в правом конце отрезка совпало со значением из начальных условий. Другими словами, решение исходной задачи эквивалентно нахождению корня уравнения

$$\Phi(\eta) = 0$$

где $\Phi(\eta) = \alpha_1 y + \alpha_2 y' - \beta$, α_1, α_2, β - коэффициенты уравнения правой границы.

Следующее значение искомого корня определяется по соотношению

$$\eta_{j+2} = \eta_{j+1} - \frac{\eta_{j+1} - \eta_j}{\Phi(\eta_{j+1}) - \Phi(\eta_j)}\Phi(\eta_{j+1})$$

Конечно-разностный метод

Рассмотрим двухточечную краевую задачу для линейного дифференциального уравнения второго порядка на отрезке $[a, b]$:

$$y'' + p(x)y' + q(x)y = f(x)$$
$$y'(a) = z_0$$
$$\alpha_1 y(b) + \alpha_2 y'(b) = \beta$$

Введем разностную аппроксимацию производных следующим образом

$$y'_k = \frac{y_{k+1} - y_{k-1}}{2h} + O(h^2)$$
$$y''_k = \frac{y_{k+1} - 2y_k + y_{k-1}}{h^2} + O(h^2)$$

Подставляя аппроксимации, приводя подобные и учитывая граничные условия, получим систему линейных алгебраических уравнений с трехдиагональной матрицей коэффициентов

$$\begin{cases} -y_1 + y_2 = hz_0 \\ \left(1 - \dfrac{hp(x_k)}{2}\right) y_{k-1} + (h^2 q(x_k) - 2) y_k + \left(1 + \dfrac{hp(x_k)}{2}\right) y_{k+1} = h^2 f(x_k), \qquad k = 2, \ldots, N-2 \\ -\alpha_1 y_{N-1} + (h\alpha_2 + \alpha_1) y_N = h\beta \end{cases}$$

Реализация

Метод стрельбы

```
class ShootingMethodL2R3 {
    public:
        ShootingMethodL2R3(BVProblemL2R3 task) : task(task), f(task.f) {}

        void Calc (double h, double eps) {
            x = GetValInRange(task.left, task.right, h);
            CauchyProblem ctask1 = task;
            CauchyProblem ctask2 = task;
            ctask1.y0 = 1;
            ctask2.y0 = 0;
            RungeKutta2 rg1(ctask1);
            RungeKutta2 rg2(ctask2);
            double& n1 = rg1.y0;
            double& n2 = rg2.y0;
            vdouble y1 = rg1(h);
            vdouble y2 = rg2(h);
            if (isFinish(y1, eps)) {
                y = y1;
                return;
            } else if (isFinish(y2, eps)) {
                y = y2;
                return;
            }
            do {
                double next_n = GetNextN(n1, n2, y1, y2);
                n1 = n2;
                n2 = next_n;
                y1 = y2;
                y2 = rg2(h);
            } while (!isFinish(y2, eps));

            y = y2;
        }

        vdouble GetY() const {
            return y;
        }

    private:

        double GetYDer(const vdouble& y, double xp) {
            int i = 0;
            for (; i < x.size() - 2; ++i) {
                if (x[i] <= xp && xp <= x[i + 1]) {
                    break;
```

```
46                }
47            }
48            return (y[i + 1] - y[i]) / (x[i + 1] - x[i]);
49        }
50
51        double GetPhi (const vdouble& y) {
52            double y_der = GetYDer(y, task.right);
53            double phi = task.re.b * y_der + task.re.a * y[y.size() - 1] - task.re.c;
54            return phi;
55        }
56
57        double GetNextN(double n1, double n2, const vdouble& y1, const vdouble& y2) {
58            double phi1 = GetPhi(y1);
59            double phi2 = GetPhi(y2);
60
61            return n2 - (n2 - n1) / (phi2 - phi1) * phi2;
62        }
63
64        bool isFinish(const vdouble& y, double eps) {
65            double phi = GetPhi(y);
66            return std::abs(phi) < eps;
67        }
68
69        Function3* f;
70        BVProblemL2R3 task;
71
72        vdouble x;
73        vdouble y;
74    };
```

Конечно-разностный метод

```
1   class FDMethod {
2       public:
3       FDMethod(Function1* p, Function1* q, BVProblemL2R3 t) :
4           p(p), q(q), task(t) {}
5
6       void Calc(double h) {
7           x = GetValInRange(task.left, task.right, h);
8           int n = (task.right - task.left) / h;
9           vvdouble mat(n + 1, vdouble(4));
10          mat[0][0] = 0;
11          mat[0][1] = -1;
12          mat[0][2] = 1;
13          mat[0][3] = task.z0 * h;
14
15          for (int i = 1; i < n; ++i) {
16              mat[i][0] = 1 - (*p)(x[i]) * h / 2;
17              mat[i][1] = (*q)(x[i]) * h * h - 2;
```

```
18            mat[i][2] = 1 + (*p)(x[i]) * h / 2;
19            mat[i][3] = 0;
20        }
21
22        mat[n][0] = -task.re.b;
23        mat[n][1] = h * task.re.a + task.re.b;
24        mat[n][2] = 0;
25        mat[n][3] = task.re.c * h;
26
27        y = SweepMethod(mat);
28    }
29
30    vdouble GetY() {
31        return y;
32    }
33
34    private:
35        Function1* p;
36        Function1* q;
37        BVProblemL2R3 task;
38
39        vdouble x;
40        vdouble y;
41 };
```

Результаты

```
(base) MacBook-Air-Dima:Program dandachok$ cat tests/input/part2.t
5
0 13 -5 -66
-4 9 -5 -47
-1 -12 -6 -43
6 20 -5 -74
4 5 0 14
(base) MacBook-Air-Dima:Program dandachok$ ./part2 <tests/input/part2.t
[-7, -5, 6, -4, 6]
(base) MacBook-Air-Dima:Program dandachok$
```

$$x = \begin{pmatrix} -7 \\ -5 \\ 6 \\ -4 \\ 6 \end{pmatrix}$$

Часть 3

Задание

Реализовать метод простых итераций и метод Зейделя в виде программ, задавая в качестве входных данных матрицу системы, вектор правых частей и точность вычислений. Используя разработанное программное обеспечение, решить СЛАУ. Проанализировать количество итераций, необходимое для достижения заданной точности.

Вариант: 3

$$\begin{cases} -23x_1 - 7x_2 + 5x_3 + 2x_4 = -26 \\ -7x_1 - 21x_2 + 4x_3 + 9x_4 = -55 \\ 9x_1 + 5x_2 - 31x_3 - 8x_4 = -58 \\ x_2 - 2x_3 + 10x_4 = -24 \end{cases}$$

Теория

Метод простых итераций

При большом числе уравнений прямые методы решения СЛАУ (за исключением метода прогонки) становятся труднореализуемыми на ЭВМ прежде всего из-за сложности хранения и обработки матриц большой размерности. В то же время характерной особенностью ряда часто встречающихся в прикладных задачах СЛАУ является разреженность матриц. Число ненулевых элементов таких матриц мало по сравнению с их размерностью. Для решения СЛАУ с разреженными матрицами предпочтительнее использовать итерационные методы. Методы последовательных приближений, в которых при вычислении последующего приближения решения используются предыдущие, уже известные приближенные решения, называются итерационными Рассмотрим СЛАУ

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \quad\vdots \quad\quad \vdots \quad\quad \ddots \quad\quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{cases}$$

Приведем СЛАУ к эквивалентному виду

$$\begin{cases} x_1 = \beta_1 + \alpha_{11}x_1 + \alpha_{12}x_2 + \cdots + \alpha_{1n}x_n \\ x_2 = \beta_2 + \alpha_{21}x_1 + \alpha_{22}x_2 + \cdots + \alpha_{2n}x_n \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ x_n = \beta_n + \alpha_{n1}x_1 + \alpha_{n2}x_2 + \cdots + \alpha_{nn}x_n \end{cases}$$

или в векторно-матричной форме

$$x = \beta + \alpha x$$

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \quad \beta = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_n \end{pmatrix}, \quad \alpha \neq \begin{pmatrix} \alpha_{11} & \dots & \alpha_{1n} \\ \vdots & \ddots & \dots \\ \alpha_{n1} & \dots & \alpha_{nn} \end{pmatrix}$$

Разрешим систему относительно неизвестных при ненулевых диагональных элементах $a_{ii} \neq 0$, $i = \overline{1,n}$ (если какой-либо коэффициент на главной диагонали равен нулю, достаточно соответствующее уравнение поменять местами с любым другим уравнением).

Получим следующие выражения для компонентов вектора β и матрицы α эквивалентной системы

$$\begin{aligned} \beta_i &= \frac{b_i}{a_{ii}}, \qquad i = \overline{1,n}; \\ \alpha_{ij} &= -\frac{a_{ij}}{a_{ii}}, \qquad i = \overline{1,n}, i \neq j; \\ \alpha_{ij} &= 0, \qquad i = \overline{1,n}, i = j; \end{aligned}$$

При таком способе приведения исходной СЛАУ к эквивалентному виду метод простых итераций носит название метода Якоби. Тогда метод простых итераций примет вид

$$\begin{cases} x^{(0)} = \beta \\ x^{(1)} = \beta + \alpha x^{(0)} \\ x^{(2)} = \beta + \alpha x^{(1)} \\ \cdots\cdots\cdots\cdots \\ x^{(k)} = \beta + \alpha x^{(k-1)} \end{cases}$$

Видно преимущество итерационных методов по сравнению, например, с рассмотренным выше методом Гаусса. В вычислительном процессе участвуют только произведения матрицы на вектор, что позволяет работать только с ненулевыми элементами матрицы, значительно упрощая процесс хранения и обработки матриц. Имеет место следующее достаточное условие сходимости метода простых итераций. Метод простых итераций сходится к единственному решению СЛАУ (а, следовательно, и к решению исходной СЛАУ) при любом начальном приближении, если какая-либо норма матрицы эквивалентной системы меньше единицы. Приведем также необходимое и достаточное условие сходимости метода простых итераций. Для сходимости итерационного процесса необходимо и достаточно, чтобы спектр матрицы эквивалентной системы лежал внутри круга с радиусом, равным единице.

Метод Зейделя

Метод Зейделя для известного вектора итерации имеет вид:

$$\begin{cases} x_1^{k+1} = \beta_1 + \alpha_{11}x_1^k + \alpha_{12}x_2^k + \ldots + \alpha_n x_n^k \\ x_2^{k+1} = \beta_2 + \alpha_{21}x_1^{k+1} + \alpha_{22}x_2^k + \ldots + \alpha_n x_n^k \\ x_3^{k+1} = \beta_3 + \alpha_{31}x_1^{k+1} + \alpha_{32}x_2^{k+1} + \alpha_{33}x_3^k + \ldots + \alpha_{3n}x_n^k \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ x_n^{k+1} = \beta_n + \alpha_{n1}x_1^{k+1} + \alpha_{n2}x_2^{k+1} + \ldots + \alpha_{nn-1}x_{n-1}^{k+1} + \alpha_{nn}x_n^k \end{cases}$$

Из этой системы видно $x^{k+1} = \beta + Bx^{k+1} + Cx^k$, что где B - нижняя треугольная матрица с диагональными элементами, равными нулю, а C - верхняя треугольная матрица с диагональными элементами, отличными от нуля $\alpha = B+C$. Следовательно

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

Реализовать метод вращений в виде программы, задавая в качестве входных данных матрицу и точность вычислений. Используя разработанное программное обеспечение, найти собственные значения и собственные векторы симметрических матриц. Проанализировать зависимость погрешности вычислений от числа итераций.

Вариант: 3

$$\begin{pmatrix} 5 & 5 & 3 \\ 5 & -4 & 1 \\ 3 & 1 & 2 \end{pmatrix}$$

Теория

Метод вращений Якоби применим только для симметрических матриц A_{nxn} ($A = A^T$) и решает полную проблему собственных значений и собственных векторов таких матриц. Он основан на отыскании с помощью итерационных процедур матрицы U в преобразовании подобия $\Lambda = U^{-1}AU$, а поскольку для симметрических матриц A матрица преобразования подобия U является ортогональной ($U^{-1} = U^T$), то $\Lambda = U^T AU$, где Λ - диагональная матрица с собственными значениями на главной диагонали

Пусть дана симметрическая матрица A. Требуется для нее вычислить с точностью ε все собственные значения и соответствующие им собственные векторы. Алгоритм метода вращения следующий:

Пусть известна матрица $A^{(k)}$ на k–й итерации, при этом для $k = 0$ $A^{(0)} = A$.

1. Выбирается максимальный по модулю недиагональный элемент $a(k)$ матрицы

$$A^{(k)} \left(|a_{ij}^{(k)}| = \max_{l<m} |a_{lm}^{(k)}| \right)$$

2. Ставится задача найти такую ортогональную матрицу $U^{(k)}$, чтобы в результате преобразования подобия $A^{(k+1)} = U^{(k)T} A^{(k)} U^{(k)}$ произошло обнуление элемента $a_{ij}^{(k+1)}$ матрицы $A^{(k+1)}$.

3. Строится матрица $A^{(k+1)}$

   $A^{(k+1)} = U^{(k)T} A^{(k)} U^{(k)}$, где $a_{ij}^{(k+1)} \approx 0$.

   В качестве критерия окончания итерационного процесса используется условие малости суммы квадратов внедиагональных элементов:

$$t\left(A^{(k+1)}\right) = \sqrt{\left(\sum_{l,m;l<m} \left(a_{lm}^{(k+1)}\right)^2 \right)}$$

   .

Если $t\left(A^{(k+1)}\right) > \varepsilon$, то итерационный процесс

$$A^{(k)} = U^{(k)T} A^{(k)} U^{(k)} = U^{(k)T} U^{(k-1)T} \dots U^{(0)} A^{(0)} U^{(0)} U^{(1)} \dots U^{(k)}$$

продолжается. Если $t\left(A^{(k+1)}\right) < \varepsilon$, то итерационный процесс останавливается, и в качестве искомых собственных значений принимаются $\lambda_1 \approx a_{11}, \lambda_2 \approx a_{22}, \dots, \lambda_n \approx a_{nn}$. Координатными столбцами собственных векторов матрицы A в единичном базисе будут столбцы матрицы $U^{(1)} = U^{(0)}U^{(1)}...U^{(k)}$, т.е.

$$x^1 = \begin{pmatrix} u_{11} \\ u_{21} \\ \vdots \\ u_{n1} \end{pmatrix}, \quad x^2 = \begin{pmatrix} u_{12} \\ u_{22} \\ \vdots \\ u_{n2} \end{pmatrix}, \quad \dots, \quad x^n = \begin{pmatrix} u_{1n} \\ u_{2n} \\ \vdots \\ u_{nn} \end{pmatrix}$$

причем эти собственные векторы будут ортогональны между собой, т.е.
$(x_l, x_m) \approx 0$, $l \neq m$.

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

Реализовать алгоритм QR – разложения матриц в виде программы. На его основе разработать программу, реализующую QR – алгоритм решения полной проблемы собственных значений произвольных матриц, задавая в качестве входных данных матрицу и точность вычислений. С использованием разработанного программного обеспечения найти собственные значения матрицы.

Вариант: 3

$$\begin{pmatrix} 5 & -5 & -6 \\ -1 & -8 & -5 \\ 2 & 7 & -3 \end{pmatrix}$$

Теория

В основе QR-алгоритма лежит представление матрицы в виде, где $A = QR$ - ортогональная матрица, R - верхняя треугольная. Такое разложение существует для любой квадратной матрицы. Одним из возможных подходов к построению QR разложения является использование преобразования Хаусхолдера, позволяющего обратить в нуль группу поддиагональных элементов столбца матрицы. Преобразование Хаусхолдера осуществляется с использованием матрицы Хаусхолдера, имеющей следующий вид:

$$H = E - \frac{2}{\nu^T \nu} \nu \nu^T$$

где ν - произвольный ненулевой вектор-столбец, E-единичная матрица, $\nu\nu^T$-квадратная матрицаого же размера

Матрица Хаусхолдера H_1 вычисляется

$$\nu_1^1 = a_0 + sign(a_{11}^0)\sqrt{\left(\sum_{j=1}^{n}(a_{j1}^0)\right)}$$

$$\nu_i^1 = a_{i1}^0, \qquad i = \overline{1,n}$$

$$H_1 = E - 2\frac{\nu^1 \nu^{1T}}{\nu^{1T} \nu^1}$$

Процедура QR - разложения многократно используется в QR - алгоритме вычисления собственных значений. Строится следующий итерационный процесс:

$$\begin{aligned}
&A^{(0)} = A,\\
&A^{(0)} = Q^{(0)} R^{(0)} - \text{производится QR-разложение}\\
&A^{(1)} = R^{(0)} Q^{(0)} - \text{перемножение матриц}\\
&\ldots\ldots\ldots\ldots\\
&A^{(k)} = Q^{(k)} R^{(k)} - \text{производится QR-разложение}\\
&A^{(k+1)} = R^{(k)} Q^{(k)} - \text{перемножение матриц}
\end{aligned}$$

Таким образом, каждому вещественному собственному значению будет соответствовать столбец со стремящимися к нулю поддиагональными элементами и в качестве критерия сходимости итерационного процесса для таких собственных значений мож-

но использовать следующее неравенство:

$$\left(\sum_{l=m+1}^{n} (a_{lm}^{k})^2 \right)^{1/2}$$

При этом соответствующее собственное значение принимается равным диагональному элементу данного столбца.

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:

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