**Московский авиационный институт**
**(национальный исследовательский университет)**

**Факультет компьютерных наук и прикладной математики**

**Кафедра математической кибернетики**

**Лабораторные работы по курсу «Численные методы»**

**Лабораторная работа №1**

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

**Москва, 2024**

# 1 Вычислительные методы линейной алгебры

## 1 LU-разложение матриц. Метод Гаусса

### 1.1 Постановка задачи

Реализовать алгоритм LU-разложения матриц (с выбором главного элемента) в виде программы. Используя разработанное программное обеспечение, решить систему линейных алгебраических уравнений (СЛАУ). Для матрицы СЛАУ вычислить определитель и обратную матрицу.

### 1.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/3.in
4
-7 3 -4 7
8 -1 -7 6
9 9 3 -6
-7 -9 -8 -5
-126 29 27 34
$ ./solution <tests/3.in
Решение системы:
x1 = 8.000000
x2 = -9.000000
x3 = 2.000000
x4 = -5.000000
Определитель матрицы: 16500.000000
Обратная матрица:
-0.054545,0.054545,0.006061,-0.018182
0.086000,-0.016000,0.082667,0.002000
-0.059818,-0.050182,-0.058909,-0.073273
0.017273,0.032727,-0.063030,-0.060909
```

### 1.3 Исходный код

```
#ifndef  LU_HPP
#define  LU_HPP

#include <algorithm>
#include  <cmath>
#include <utility>

#include "../matrix.hpp"

template  <class  T>
class lu_t {
   private:
    using matrix = matrix_t<T>;
    using vec = std::vector<T>;
    using pii = std::pair<size_t, size_t>;

    const T EPS = 1e-6;

    matrix l;
    matrix u;
    T det;
    std::vector<pii> swaps;

    void  decompose() {
        size_t  n = u.rows();
        for (size_t i = 0; i < n; ++i) {
            size_t max_el_ind = i;
            for (size_t j = i + 1; j < n; ++j) {
                if (abs(u[j][i]) > abs(u[max_el_ind][i])) {
                    max_el_ind = j;
                }
            }
            if (max_el_ind != i) {
                pii perm = std::make_pair(i, max_el_ind);
                swaps.push_back(perm);
                u.swap_rows(i, max_el_ind);
                l.swap_rows(i, max_el_ind);
                l.swap_cols(i, max_el_ind);
            }
            for (size_t j = i + 1; j < n; ++j) {
                if (abs(u[i][i]) < EPS) {
                    continue;
                }
                T mu = u[j][i] / u[i][i];
                l[j][i] = mu;
                for (size_t k = 0; k < n; ++k) {
                    u[j][k] -= mu * u[i][k];
```

```
48                  }
49              }
50          }
51          det = (swaps.size() & 1 ? -1 : 1);
52          for (size_t i = 0; i < n; ++i) {
53              det *= u[i][i];
54          }
55      }
56
57      void do_swaps(vec& x) {
58          for (pii elem : swaps) {
59              std::swap(x[elem.first], x[elem.second]);
60          }
61      }
62
63      public:
64      lu_t(const matrix& matr) {
65          if (matr.rows() != matr.cols()) {
66              throw std::invalid_argument("Matrix is not square");
67          }
68          l = matrix::identity(matr.rows());
69          u = matrix(matr);
70          decompose();
71      }
72
73      friend std::ostream& operator<<(std::ostream& out, const lu_t<T>& lu) {
74          out << "Matrix L:\n" << lu.l << "Matrix U:\n" << lu.u;
75          return out;
76      }
77
78      T get_det() { return det; }
79
80      vec solve(vec b) {
81          int n = b.size();
82          do_swaps(b);
83          vec z(n);
84          for (int i = 0; i < n; ++i) {
85              T summary = b[i];
86              for (int j = 0; j < i; ++j) {
87                  summary -= z[j] * l[i][j];
88              }
89              z[i] = summary;
90          }
91          vec x(n);
92          for (int i = n - 1; i >= 0; --i) {
93              if (abs(u[i][i]) < EPS) {
94                  continue;
95              }
96              T summary = z[i];
```

```
97              for (int j = n - 1; j > i; --j) {
98                  summary -= x[j] * u[i][j];
99              }
100             x[i] = summary / u[i][i];
101         }
102         return x;
103     }
104
105     matrix inv_matrix() {
106         size_t n = l.rows();
107         matrix res(n);
108         for (size_t i = 0; i < n; ++i) {
109             vec b(n);
110             b[i] = T(1);
111             vec x = solve(b);
112             for (size_t j = 0; j < n; ++j) {
113                 res[j][i] = x[j];
114             }
115         }
116         return res;
117     }
118
119     ~lu_t() = default;
120 };
121
122 #en if /* LU_HPP */
```

# 2 Метод прогонки

## 2.1 Постановка задачи

Реализовать метод прогонки в виде программы, задавая в качестве входных данных ненулевые элементы матрицы системы и вектор правых частей. Используя разработанное программное обеспечение, решить СЛАУ с трехдиагональной матрицей.

## 2.2 Консоль

```
$ cat tests/3.in
5
-7 -6
6 12 0
-3 5 0
-9 21 8
-5 -6
-75 126 13 -40 -24
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/3.in
5
-7 -6
6 12 0
-3 5 0
-9 21 8
-5 -6
-75 126 13 -40 -24
$ ./solution <tests/3.in
Решение системы:
x1 = 3.000000
x2 = 9.000000
x3 = 8.000000
x4 = 0.000000
x5 = 4.000000
```

## 2.3 Исходный код

```
#ifndef  TRIDIAG_HPP
#define  TRIDIAG_HPP

#include <exception>
#include <iostream>
#include <vector>

template <class T>
class tridiag_t {
   private:
    using vec = std::vector<T>;

    const double EPS = 1e-9;

    int n;
    vec a;
    vec b;
    vec c;

   public:
    tridiag_t(const int& _n) : n(_n), a(n), b(n), c(n) {}

    tridiag_t(const vec& _a, const vec& _b, const vec& _c) {
        if (!(_a.size() == _b.size() and _a.size() == _c.size())) {
            throw std::invalid_argument("Sizes of a, b, c are invalid");
        }
        n = _a.size();
        a = _a;
        b = _b;
        c = _c;
    }

    friend std::istream& operator>>(std::istream& in, tridiag_t<T>& tridiag)    {
        in >> tridiag.b[0] >> tridiag.c[0];
        for (int i = 1; i < tridiag.n - 1; ++i) {
             in >> tridiag.a[i] >> tridiag.b[i] >> tridiag.c[i];
        }
        in >> tridiag.a.back() >> tridiag.b.back();
        return in;
    }

    vec solve(const vec& d) {
        int m = d.size();
        if (n != m) {
            throw std::invalid_argument("Size of vector d is invalid");
        }
        vec p(n);
```

```
48          p[0] = -c[0] / b[0];
49          vec q(n);
50          q[0] = d[0] / b[0];
51          for (int i = 1; i < n; ++i) {
52              p[i] = -c[i] / (b[i] + a[i] * p[i - 1]);
53              q[i] = (d[i] - a[i] * q[i - 1]) / (b[i] + a[i] * p[i - 1]);
54          }
55          vec x(n);
56          x.back() = q.back();
57          for (int i = n - 2; i >= 0; --i) {
58              x[i] = p[i] * x[i + 1] + q[i];
59          }
60          return x;
61      }
62
63      ~tridiag_t() = default;
64  };
65
66  #endif /* TRIDIAG_HPP */
```

# 3 Итерационные методы решения СЛАУ

## 3.1 Постановка задачи

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.

## 3.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/3.in
4 0.000000001
28 9 -3 -7
-5 21 -5 -3
-8 1 -16 5
0 -2 5 8
-159 63 -45 24
$ ./solution <tests/3.in
Метод простых итераций
Решени получено за 53 итераций
Решение системы:
x1 = -6.000000
x2 = 3.000000
x3 = 6.000000
x4 = 0.000000
Метод Зейделя
Решени получено за 20 итераций
Решение системы:
x1 = -6.000000
x2 = 3.000000
x3 = 6.000000
x4 = 0.000000
```

### 3.3 Исходный код

```cpp
#ifndef ITERATION_HPP
#define ITERATION_HPP

#include <cmath>

#include "../matrix.hpp"

class iter_solver {
  private:
    using matrix = matrix_t<double>;
    using vec = std::vector<double>;

    matrix a;
    size_t n;
    double eps;

    static constexpr double INF = 1e18;

  public:
    int iter_count;

    iter_solver(const matrix& _a, double _eps = 1e-6) {
        if (_a.rows() != _a.cols()) {
            throw std::invalid_argument("Matrix is not square");
        }
        a = matrix(_a);
        n = a.rows();
        eps = _eps;
    }

    static double norm(const matrix& m) {
        double res = -INF;
        for (size_t i = 0; i < m.rows(); ++i) {
            double s = 0;
            for (double elem : m[i]) {
                s += std::abs(elem);
            }
            res = std::max(res, s);
        }
        return res;
    }

    static double norm(const vec& v) {
        double res = -INF;
        for (double elem : v) {
            res = std::max(res, std::abs(elem));
        }
```

```
48          return res;
49      }
50
51      std::pair<matrix, vec> precalc_ab(const vec& b, matrix& alpha, vec& beta) {
52          for (size_t i = 0; i < n; ++i) {
53              beta[i] = b[i] / a[i][i];
54              for (size_t j = 0; j < n; ++j) {
55                  if (i != j) {
56                      alpha[i][j] = -a[i][j] / a[i][i];
57                  }
58              }
59          }
60          return std::make_pair(alpha, beta);
61      }
62
63      vec solve_simple(const vec& b) {
64          matrix alpha(n);
65          vec beta(n);
66          precalc_ab(b, alpha, beta);
67          double eps_coef = 1.0;
68          if (norm(alpha) - 1.0 < eps) {
69              eps_coef = norm(alpha) / (1.0 - norm(alpha));
70          }
71          double eps_k = 1.0;
72          vec x(beta);
73          iter_count = 0;
74          while (eps_k > eps) {
75              vec x_k = beta + alpha * x;
76              eps_k = eps_coef * norm(x_k - x);
77              x = x_k;
78              ++iter_count;
79          }
80          return x;
81      }
82
83      vec zeidel(const vec& x, const matrix& alpha, const vec& beta) {
84          vec x_k(beta);
85          for (size_t i = 0; i < n; ++i) {
86              for (size_t j = 0; j < i; ++j) {
87                  x_k[i] += x_k[j] * alpha[i][j];
88              }
89              for (size_t j = i; j < n; ++j) {
90                  x_k[i] += x[j] * alpha[i][j];
91              }
92          }
93          return x_k;
94      }
95
96      vec solve_zeidel(const vec& b) {
```

```
97          matrix alpha(n);
98          vec beta(n);
99          precalc_ab(b, alpha, beta);
100         matrix c(n);
101         for (size_t i = 0; i < n; ++i) {
102             for (size_t j = i; j < n; ++j) {
103                 c[i][j] = alpha[i][j];
104             }
105         }
106         double eps_coef = 1.0;
107         if (norm(alpha) - 1.0 < eps) {
108             eps_coef = norm(c) / (1.0 - norm(alpha));
109         }
110         double eps_k = 1.0;
111         vec x(beta);
112         iter_count = 0;
113         while (eps_k > eps) {
114             vec x_k = zeidel(x, alpha, beta);
115             eps_k = eps_coef * norm(x_k - x);
116             x = x_k;
117             ++iter_count;
118         }
119         return x;
120     }
121
122     ~iter_solver() = default;
123 };
124
125 #endif /* ITERATION_HPP */
```

# 4 Метод вращений

## 4.1 Постановка задачи

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.

## 4.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/3.in
3 0.000001
-7 -6 8
-6 3 -7
8 -7 4
$ ./solution <tests/3.in
Собственные значения:
l_1 = -11.607818
l_2 = 15.020412
l_3 = -3.412593
Собственные векторы:
0.905671,-0.412378,0.098514
0.190483,0.603339,0.774402
-0.378784,-0.682588,0.624977
Решение получено за 7 итераций
```

### 4.3 Исходный код

```
#ifndef ROTATION_HPP
#define ROTATION_HPP

#include <cmath>

#include "../matrix.hpp"

class rotation {
   private:
    using matrix = matrix_t<double>;
    using vec = std::vector<double>;

    static constexpr double GLOBAL_EPS = 1e-9;

    size_t n;
    matrix a;
    double eps;
    matrix v;

    static double norm(const matrix& m) {
        double res = 0;
        for (size_t i = 0; i < m.rows(); ++i) {
            for (size_t j = 0; j < m.cols(); ++j) {
                if (i == j) {
                    continue;
                }
                res += m[i][j] * m[i][j];
            }
        }
        return std::sqrt(res);
    }

    double calc_phi(size_t i, size_t j) {
        if (std::abs(a[i][i] - a[j][j]) < GLOBAL_EPS) {
            return std::atan2(1.0, 1.0);
        } else {
            return 0.5 * std::atan2(2 * a[i][j], a[i][i] - a[j][j]);
        }
    }

    matrix create_rotation(size_t i, size_t j, double phi) {
        matrix u = matrix::identity(n);
        u[i][i] = std::cos(phi);
        u[i][j] = -std::sin(phi);
        u[j][i] = std::sin(phi);
        u[j][j] = std::cos(phi);
        return u;
```

```
48      }
49
50      void build() {
51          iter_count = 0;
52          while (norm(a) > eps) {
53              ++iter_count;
54              size_t i = 0, j = 1;
55              for (size_t ii = 0; ii < n; ++ii) {
56                  for (size_t jj = 0; jj < n; ++jj) {
57                      if (ii == jj) {
58                          continue;
59                      }
60                      if (std::abs(a[ii][jj]) > std::abs(a[i][j])) {
61                          i = ii;
62                          j = jj;
63                      }
64                  }
65              }
66              double phi = calc_phi(i, j);
67              matrix u = create_rotation(i, j, phi);
68              v = v * u;
69              a = u.t() * a * u;
70          }
71      }
72
73      public:
74       int iter_count;
75
76       rotation(const matrix& _a, double _eps) {
77          if (_a.rows() != _a.cols()) {
78              throw std::invalid_argument("Matrix is not square");
79          }
80          a = matrix(_a);
81          n = a.rows();
82          eps = _eps;
83          v = matrix::identity(n);
84          build();
85       };
86
87       matrix get_eigen_vectors() { return v; }
88
89       vec get_eigen_values() {
90          vec res(n);
91          for (size_t i = 0; i < n; ++i) {
92              res[i] = a[i][i];
93          }
94          return res;
95       }
96
```

```
 97 ||      ~rotation() = default;
 98 || };
 99 ||
    ||
100  #endif /* ROTATION_HPP */
```

# 5 QR алгоритм

## 5.1 Постановка задачи

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.

## 5.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/3.in
3 0.000001
-1 4 -4
2 -5 0
-8 -2 0
$ ./solution <tests/3.in
Решение получено за 32 итераций
Собственные значения:
l_1 = -7.547969
l_2 = 5.664787
l_3 = -4.116817
```

### 5.3 Исходный код

```
#ifndef  QR_ALGO_HPP
#define  QR_ALGO_HPP

#include  <cmath>
#include  <complex>

#include "../matrix.hpp"

class qr_algo {
   private:
     using matrix = matrix_t<double>;
     using vec = std::vector<double>;
     using complex = std::complex<double>;
     using pcc = std::pair<complex, complex>;
     using vec_complex = std::vector<complex>;

     static constexpr double INF = 1e18;
     static constexpr complex COMPLEX_INF = complex(INF, INF);

     size_t n;
     matrix a;
     double eps;
     vec_complex eigen;

     double vtv(const vec& v) {
         double res = 0;
         for (double elem : v) {
             res += elem * elem;
         }
         return res;
     }

     double norm(const vec& v) { return std::sqrt(vtv(v)); }

     matrix vvt(const vec& b) {
         size_t n_b = b.size();
         matrix res(n_b);
         for (size_t i = 0; i < n_b; ++i) {
             for (size_t j = 0; j < n_b; ++j) {
                 res[i][j] = b[i] * b[j];
             }
         }
         return res;
     }

     double sign(double x) {
         if (x < eps) {
```

```
48              return -1.0;
49          } else if (x > eps) {
50              return 1.0;
51          } else {
52              return 0.0;
53          }
54      }
55
56      matrix householder(const vec& b, int id) {
57          vec v(b);
58          v[id] += sign(b[id]) * norm(b);
59          return matrix::identity(n) - (2.0 / vtv(v)) * vvt(v);
60      }
61
62      pcc solve_sq(double a11, double a12, double a21, double a22) {
63          double a = 1.0;
64          double b = -(a11 + a22);
65          double c = a11 * a22 - a12 * a21;
66          double d_sq = b * b - 4.0 * a * c;
67          if (d_sq > eps) {
68              complex bad(NAN, NAN);
69              return std::make_pair(bad, bad);
70          }
71          complex d(0.0, std::sqrt(-d_sq));
72          complex x1 = (-b + d) / (2.0 * a);
73          complex x2 = (-b - d) / (2.0 * a);
74          return std::make_pair(x1, x2);
75      }
76
77      bool check_diag() {
78          for (size_t i = 0; i < n; ++i) {
79              double col_sum = 0;
80              for (size_t j = i + 2; j < n; ++j) {
81                  col_sum += a[j][i] * a[j][i];
82              }
83              double norm = std::sqrt(col_sum);
84              if (!(norm < eps)) {
85                  return false;
86              }
87          }
88          return true;
89      }
90
91      void calc_eigen() {
92          for (size_t i = 0; i < n; ++i) {
93              if (i < n - 1 and !(abs(a[i + 1][i]) < eps)) {
94                  auto [l1, l2] = solve_sq(a[i][i], a[i][i + 1], a[i + 1][i],
95                                           a[i + 1][i + 1]);
96                  if (std::isnan(l1.real())) {
```

```
                    eigen[i] = COMPLEX_INF;
                    ++i;
                    eigen[i] = COMPLEX_INF;
                    continue;
                }
                eigen[i] = l1;
                eigen[++i] = l2;
            } else {
                eigen[i] = a[i][i];
            }
        }
    }

    bool check_eps() {
        if (!check_diag()) {
            return false;
        }
        vec_complex prev_eigen(eigen);
        calc_eigen();
        for (size_t i = 0; i < n; ++i) {
            bool bad = (std::norm(eigen[i] - COMPLEX_INF) < eps);
            if (bad) {
                return false;
            }
            double delta = std::norm(eigen[i] - prev_eigen[i]);
            if (delta > eps) {
                return false;
            }
        }
        return true;
    }

    void build() {
        iter_count = 0;
        while (!check_eps()) {
            ++iter_count;
            matrix q = matrix::identity(n);
            matrix r(a);
            for (size_t i = 0; i < n - 1; ++i) {
                vec b(n);
                for (size_t j = i; j < n; ++j) {
                    b[j] = r[j][i];
                }
                matrix h = householder(b, i);
                q = q * h;
                r = h * r;
            }
            a = r * q;
        }
```

```
146     }
147
148   public:
149     int iter_count;
150
151     qr_algo(const matrix& _a, double _eps) {
152         if (_a.rows() != _a.cols()) {
153             throw std::invalid_argument("Matrix is not square");
154         }
155         n = _a.rows();
156         a = matrix(_a);
157         eps = _eps;
158         eigen.resize(n,  COMPLEX_INF);
159         build();
160     };
161
162     vec_complex get_eigen_values() {
163         calc_eigen();
164         return eigen;
165     }
166
167     ~qr_algo() = default;
168 };
169
170 #endif /* QR_ALGO_HPP */
```