Московский государственный университет имени М.В. Ломоносова
Казахстанский филиал
Факультет вычислительной математики и кибернетики

Отчет по практикуму Вычислительные методы линейной алгебры

Составил Шмаль А.С.

Проверил Нетесов В.В.

Нур-Султан, 2020

Содержание:

1 Метод Гаусса, его применение в решении задач

1.1 Вычисление определителя матрицы

Определение: Определитель матрицы A порядка n — число:

$$det(A) = \sum_{\alpha=(\alpha_1\alpha_2,...,\alpha_n)} (-1)^{\delta(\alpha)} \cdot a_{1\alpha_1} \cdot a_{2\alpha_2} \cdot ... \cdot a_{n\alpha_n},$$

где $\delta(\alpha)$ число инверсий в перестановке $\alpha_1, \alpha_2, ..., \alpha_n$, причем сомножители в этом произведении упорядочены в порядке возрастания номеров строк, а суммирование ведется по всевозможным перестановкам $(\alpha_1\alpha_2, ..., \alpha_n)$ из чисел 1, 2, ..., n.

Вычисление определителя матрицы по определению крайне непрактично. В связи с этим были придуманы специальные способы его нахождения, одним из них является метод Гаусса. Его идея заключается в том, чтобы привести матрицу к треугольному виду, тогда определитель такой приведенной матрицы легко вычисляется как произведение ее диагональных элементов. Как известно, при прибавлении к одной строке матрицы, умноженной на некторое ненулевое число, другой строки, также умноженной на некоторое ненулевое число, ее определитель не меняется. В ходе таких операций матрицу можно привести к более простому, треугольному виду. Учитывая, что при перестановке строк местами определитель матрицы меняет знак, после получения треугольного вида можно учесть знак определителя. Все тоже самое верно и для столбцов матрицы.

Алгоритм приведения матрицы A к треугольному виду:

```
1       int n = A.row();
2       double eps = 1.0e-10;
3       for(int k = 0; k < n - 1; k++) {
4               for(int i = k + 1; i < n; i++) {
5                       if(fabs(A(k, k)) < eps) {
6                               cout << "Error!" << endl;
7                               return;
8                       }
9                       double tmp = A(i, k) / A(k, k);
10                      A(i, k) = 0;
11                      for(int j = k + 1; j < n; j++) {
12                              A(i, j) -= tmp * A(k, j);
13                      }
```

```
14                  }
15          }
```

Так как реальные машинные вычисления производятся не с точными, а с усеченными числами, т.е. неизбежны ошибки округления, то выполнение алгоритма может прекратиться или привести к неверным результатам на каком-то этапе переменная A(k, k) окажется равной нулю или очень маленьким числом, поэтому целесообразно сравнивать эту переменную с какой-то заранее заданной очень маленькой величиной eps. В случае вырожденности матрицы алгоритм также прекратится. Данный алгоритм называется **схемой единственного деления**. Для того чтобы сделать алгоритм более устойчевым к таким ситуациям следует осуществлять частичное упорядочивание по столбцам, т.е. брать за ведущий элемент наибольший по модулю элемент среди всех на данном столбце и переставлять строку с нужным элементом со строкой, которую мы бы стали обрабатывать в обычном алгоритме. Данный алгоритм называется **метод Гаусса с постолбцовым выбором главного элемента**.

Реализация данного алгоритма с матрицей A:

```
int n = A.row();
double eps = 1.0e-10;
for(int k = 0; k < n - 1; k++) {
        double max = fabs(A(k, k));
        int pos = k;
        for (int t = k + 1; t < n; t++) {
                if (fabs(A(t, k)) > max) {
                        max = fabs(A(t, k));
                        pos = t;
                }
        }
        if (pos != k) {
                for (int j = k; j < n; j++) {
                        A.swap(A(pos, j), A(k, j));
                }
        }
        for(int i = k + 1; i < n; i++) {
                if(max < eps) {
                        cout << "Error!" << endl;
                        return;
                }
                double tmp = A(i, k) / A(k, k);
                A(i, k) = 0;
                for(int j = k + 1; j < n; j++) {
                        A(i, j) -= tmp * A(k, j);
                }
```

```
            }
}
```

Однако используя такой алгоритм, необходимо подсчитывать число перестановок строк, чтобы не потерять информацию о знаке определителя матрицы.

Пример. Пользуясь схемой единственного деления, найдем определитель матрицы

$$A = \begin{bmatrix} 2 & -1 & 1 \\ 4 & 3 & 1 \\ 6 & -13 & 6 \end{bmatrix}$$

Вычисляя коэффициенты $(1) a_{21}/a_{11} = 4/2 = 2$, $(2) a_{31}/a_{11} = 6/2 = 3$, вычитая из второй строки первую, умноженную на (1) и из третьей опять таки первую, умноженную на (2), преобразуем матрицу к виду

$$A = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 5 & -1 \\ 0 & -10 & 3 \end{bmatrix}$$

Далее, вычисляя коэффициент $a_{32}^{(1)}/a_{22}^{(1)} = -10/7 = -1.428$ и вычитая из третьей строки вторую, умноженную на этот коэффициент, приводим матрицу к треугольному виду

$$A = \begin{bmatrix} 2 & -1 & 1 \\ 0 & 5 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

Теперь, окончательно находим определитель матрицы

$$det(A) = a_{11}^{(2)} \cdot a_{22}^{(2)} \cdot a_{33}^{(2)} = 2 \cdot 5 \cdot 1 = 10.$$

1.2 Решение систем линейных уравнений

Используя алгоритмы приведения квадратной матрицы к треугольному виду, описанные в п.1.1., можно решить систему линейных уравнений с квадратной невырожденной матрицей и произвольным столбцом свободных членов той же размерности, что и у матрицы. Приведение матрицы системы к такому виду называют **прямым ходом** метода Гаусса. При этом все преобразования с матрицей необходимо осуществить и со столбцом свободных членов. Поэтому в алгоритме схемы единственного деления необходимо в цикле между 11-й и 13-й строками кода добавить фрагмент «b[i] -= tmp * b[k];», где b(n) – вектор размерности n, соответствующий столбцу свободных членов. Алгоритм с постолбцовым выбором ведущего элемента будет выгдлядеть следующим образом(на вход подаются матрица A и столбец свободных членов b):

```
int n = A.row();
double eps = 1.0e-10;
for(int k = 0; k < n - 1; k++) {
        double max = fabs(A(k, k));
        int pos = k;
        for (int t = k + 1; t < n; t++) {
                if (fabs(A(t, k)) > max) {
                        max = fabs(A(t, k));
                        pos = t;
                }
        }
        if (pos != k) {
                for (int j = k; j < n; j++) {
                        A.swap(A(pos, j), A(k, j));
                }
                double tmp = b[pos];
                b[pos] = b[k]; b[k] = tmp;
        }
        for(int i = k + 1; i < n; i++) {
                if(max < eps) {
                        cout << "Error!" << endl;
                        return;
                }
                double tmp = A(i, k) / A(k, k);
                A(i, k) = 0;
                for(int j = k + 1; j < n; j++) {
                        A(i, j) -= tmp * A(k, j);
                }
                b[i] -= tmp * b[k];
        }
}
```

После применения какого-либо из этих двух алгоритмов можно рекурсивно вычислить неизвестные и таким образом найти решение системы(этот процесс носит название **обратный ход** метода Гаусса) :

```
Vector<double> x(n);
for (int i = n - 1; i >= 0; i--) {
        x[i] = b[i];
        double sum = 0;
        for(int j = n - 1; j > i; j--) sum += A(i, j) * x[j];
        x[i] = (x[i] - sum) / A(i, i);
}
```

Пример. Используя метод Гаусса, решим систему

$$\begin{aligned} 2x_1 - 9x_2 + 5x_3 &= -4, \\ 1.2x_1 - 5.3999x_2 + 6x_3 &= 0.6001, \\ x_1 - x_2 - 7.5x_3 &= -8.5. \end{aligned}$$

Прямой ход.
1-й шаг. Вычислим множители $a_{21}/a11 = 0.6$ и $a_{31}/a11 = 0.5$ и преобразуем систему к виду

$$\begin{aligned} 2x_1 - 9x_2 + 5x_3 &= -4, \\ 0.0001x_2 + 3x_3 &= 3.0001, \\ 3.5x_2 - 10x_3 &= -6.5. \end{aligned}$$

2-й шаг. Среди элементов $a_{22}^{(1)} = 0.0001$ и $a_{32}^{(1)} = 3.5$ матрицы полученной системы максимальный по модулю принадлежит третьему уравнению. Меняя местами второе и третье уравнения, получаем систему

$$\begin{aligned} 2x_1 - 9x_2 + 5x_3 &= -4, \\ 3.5x_2 - 10x_3 &= -6.5, \\ 0.0001x_2 + 3x_3 &= 3.0001. \end{aligned}$$

После вычисления $a_{32}^{(2)}/a_{22}^{(2)} = 0.0001/3.5 \approx 2.85714 \cdot 10^{-5}$ система уже преобразуется к виду

$$\begin{aligned} 2x_1 - 9x_2 + 5x_3 &= -4, \\ 3.5x_2 - 10x_3 &= -6.5, \\ 3.00029x_3 &= 3.00029. \end{aligned}$$

Обратный ход. Из последнего уравнения находим $x_3 = 1$. Далее, имеем $x_2 = (-6.5 + 10x_3)/3.5 = 1$, $x_1 = (-4 + 9x_2 - 5x_3)/2 = (-4 + 9 - 5)/2 = 0$.

1.3 Вычисление обратной матрицы

Справедливо утверждение: квадратная матрица обратима тогда и только тогда, когда она невырождена. Метод Гаусса решения линейных систем с квадратной невырожденной матрицей можно использовать и в нахождении обратной матрицы для заданной. Действительно, пусть нам дана квадратная невырожденная матрица порядка n, а X – это неизвестная обратная к ней матрица. Тогда, если I – это единичная матрица того же порядка, то из определения обратной матрицы имеем

$$AX = I.$$

Теперь приведем матрицу к треугольному виду и осуществим те же преобразования с единичной матрицей. Тогда получим уже новое матричное уравнение

$$A^{'}X = I^{'}. \tag{1}$$

Последнее матричное уравнение можно определить как n систем уравнений с одной и той же матрицей $A^{'}$ и соответственно со столбцами свободных членов матрицы $I^{'}$. Решая каждую систему, находим столбцы искомой матрицы как столбцы неизвестных. При решении этих систем и применяется метод Гаусса. Поэтому для реализации данного алгоритма достаточно, например, в алгоритме схемы единственного деления после 13-й строки кода добавить фрагмент с циклом «for (int s = 0; s < n; s++) I(i, s) -= tmp * I(k, s);», где I – это единичная матрица порядка n. И затем уже находим непосредственно элементы самой обратной матрицы обратным ходом:

```
for (int k = 0; k < n; k++) {
        for (int i = n - 1; i >= 0; i--) {
                X(i, k) = I(i, k);
                double sum = 0;
                for (int j = n - 1; j > i; j--) {
                        sum += A(i, j) * X(j, k);
                }
                X(i, k) = (X(i, k) - sum) / A(i, i);
        }
}
```

2 LU-разложение матрицы и его приложение в задачах

2.1 Два способа LU-разложения матрицы

Пусть A – данная матрица порядка n, а L и U – соответсвтенно нижняя и верхняя треугольные матрицы. Найдем представление матрицы A в виде произведения матриц-сомножителей L и U, перемножая эти матрицы с неизвестными элементами l_{ij} и u_{ij} и приравнивая полученные элементы матрицы-результата соотвтествующим элементам a_{ij} данной матрицы. Чтобы найти однозначное решение поставленной задачи, наложим n дополнительных условий: положим $l_{ij} = 1$. В результате получим n^2 следующих уравнений

$$u_{11} = a_{11}, \qquad u_{12} = a_{12}, \ ..., \qquad u_{1n} = a_{1n},$$

$$l_{21}u_{11} = a_{21}, \quad l_{21}u_{12} + u_{22} = a_{22}, \ ..., \quad l_{21}u_{1n} + u_{2n} = a_{2n},$$

.....

$$l_{n1}u_{11} = a_{n1}, \ l_{n1}u_{12} + l_{n2}u_{22} = a_{n2}, \ ..., \ l_{n1}u_{1n} + ... + u_{nn} = a_{nn}.$$

Таким образом, все неизвестные элементы матриц L и U можно вычислить по формулам:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj} \qquad (\text{где } i \leq j),$$

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj} \right) \quad (\text{где } i > j).$$

Во избежание деления на нулевой элемент u_{jj} будем как и в методе Гаусса сравнивать его с заранее заданным числом близким к нулю.

Алгоритм LU-разложения матрицы A:

```
int n = A.row();
double eps = 1.0e-10;
Matrix<double> L(n), U(n);
for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
                double sum = 0;
                if (i <= j) {
                        for (int k = 0; k < i; k++) {
                                sum += L(i, k) * U(k, j);
                        }
                        U(i, j) = A(i, j) - sum;
                }
                else {
                        if (U(j, j) < eps) {
                                cout << "Error!" << endl;
                                exit(1);
                        }
                        for (int k = 0; k < j; k++) {
                                sum += L(i, k) * U(k, j);
                        }
                        L(i, j) = (A(i, j) - sum) / U(j, j);
                }
                L(i, i) = 1;
        }
}
```

Пример. Выполним LU-разложение матрицы

$$\begin{bmatrix} 2 & -1 & 1 \\ 4 & 3 & 1 \\ 6 & -13 & 6 \end{bmatrix}$$

Вычислим элементы:

$$u_{11} = a_{11} = 2, u_{12} = a_{12} = -1, u_{13} = a_{13} = 1;$$

$$l_{21} = \frac{a_{21}}{u_{11}} = \frac{4}{2} = 2, l_{31} = \frac{a_{31}}{u_{11}} = \frac{6}{2} = 3;$$

$$u_{22} = a_{22} - l_{21}u_{12} = 3 - 2 \cdot (-1) = 5, u_{23} = a_{23} - l_{21}u_{13} = 1 - 2 \cdot 1 = 1;$$

$$l_{32} = \frac{1}{u_{22}}(a_{32} - l_{31}u_{12}) = \frac{1}{5}(-13 - 3 \cdot (-1)) = -2;$$

$$u_{33} = a_{33} - l_{31}u_{13} - l_{32}u_{23} = 6 - 3 \cdot 1 - (-2) \cdot (-1) = 1.$$

Следовательно, матрица раскладывается на множители:

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & -2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & -1 & 1 \\ 0 & 5 & -1 \\ 0 & 0 & 1 \end{bmatrix}.$$

LU-разложение можно получить и другим способом. В прямом ходе метода Гаусса приведение матрицы A к треугольному виду равносильно умножению на нее слева некоторых матриц-перестановок(умножение слева, т.к. происходят преобразования строк):

$$M_{n-1}...M_2M_1A = A^{(n-1)} \tag{2}$$

Обнуление всех элементов на k-том столбце матрицы, расположенных ниже диагонального(ведущего) элемента, равносильно умножению матрицы слева на матрицу вида

$$M_k = \begin{pmatrix} 1 & 0 & .... & 0 & .... & .... & ...0 \\ 0 & 1 & ..... & 0 & .... & .... & ...0 \\ 0 & 0 & ..... & 0 & .... & .... & ...0 \\ . & . & ..... & . & .... & .... & .... \\ 0 & 0 & ..... & 1 & .... & .... & .... \\ 0 & 0 & ..... & -tmp_{(k+1)k} & .... & .... & ...0 \\ . & . & ..... & . & .... & .... & .... \\ 0 & 0 & ..... & -tmp_{(n-1)k} & .... & .... & ...0 \\ 0 & 0 & ..... & -tmp_{nk} & .... & .... & ...1 \end{pmatrix},$$

где $tmp_{ik} = a_{ik}^{k-1}/a_{kk}^{k-1}$(т.е. коэффициент tmp из алгоритма схемы единственного деления на стр.3).

Из сотношения 2 получаем разложение для матрицы A

$$A = M_1^{-1}M_2^{-1}...M_{n-1}^{-1}A^{(n-1)}.$$

Введя обозначения $L = M_1^{-1}M_2^{-1}...M_{n-1}^{-1}$ и $U = A^{(n-1)}$, получаем разложение для исходной матрицы

$$A = LU.$$

При этом легко убедиться, что матрица M_k^{-1} имеет вид

$$\begin{pmatrix} 1 & 0 & .... & 0 & .... & .... & ...0 \\ 0 & 1 & ..... & 0 & .... & .... & ...0 \\ 0 & 0 & ..... & 0 & .... & .... & ...0 \\ . & . & ..... & . & .... & .... & .... \\ 0 & 0 & ..... & 1 & .... & .... & .... \\ 0 & 0 & ..... & tmp_{(k+1)k} & .... & .... & ...0 \\ . & . & ..... & . & .... & .... & .... \\ 0 & 0 & ..... & tmp_{(n-1)k} & .... & .... & ...0 \\ 0 & 0 & ..... & tmp_{nk} & .... & .... & ...1 \end{pmatrix}.$$

Выполнив перемножение матриц $M_1^{-1}, M_2^{-1}, ..., M_{n-1}^{-1}$, получаем вид матрицы L

$$\begin{pmatrix} 1 & 0 & .... & 0 & .... & .... & ...0 \\ tmp_{21} & 1 & ..... & 0 & .... & .... & ...0 \\ tmp_{31} & tmp_{32} & ..... & 0 & .... & .... & ...0 \\ . & . & ..... & . & .... & .... & .... \\ tmp_{k1} & tmp_{k2} & ..... & 1 & .... & .... & .... \\ tmp_{(k+1)1} & tmp_{(k+1)2} & ..... & tmp_{(k+1)k} & .... & .... & ...0 \\ . & . & ..... & . & .... & .... & .... \\ tmp_{(n-1)1} & tmp_{(n-1)2} & ..... & tmp_{(n-1)k} & .... & .... & ...0 \\ tmp_{n1} & tmp_{n2} & ..... & tmp_{nk} & .... & .... & ...1 \end{pmatrix}.$$

Если в алгоритме схемы единственного деления при приведении матрицы к треугольному виду вместо обнуления ведущего элемента A(k, k) хранить в нем тут же полученное значение коэффициента tmp, то в итоге, в полученной матрице все элементы, расположенные ниже диагонали, и будут соответственными элементами матрицы L(диагональные же элементы этой матрицы известны, они равны единице), а все остальные элементы этой матрицы представляют собой соответственные элементы матрицы U. Таким образом, все искомые элементы матриц L и U можно хранить на месте исходной мтрицы.

2.2 Решение систем линейных уравнений LU-разложением

LU-разложение матрицы коэффициентов системы линейных уравнений можно применить для решения этой сиcетмы. Пусть дана система

$$Ax = b.$$

Выполним подстановку $A = LU$:

$$LUx = b.$$

Положив $Ux = y$, приводим поставленную задачу к решению двух систем с треугольными матрицами

$$\begin{cases} Ly = b, \\ Ux = y. \end{cases}$$

Преимущество данного способа в том, что, если требуется решить некоторое число систем с одной и той же матрицей коэффициентов, но с разными столбцами свободных членов, то достаточно один раз найти LU-разложение матрицы A, а затем решать вышеприведенную систему с разными столбцами свободных членов b. Пользуясь уже полученным LU-разложением, опишем этот алгоритм решения сиcетмы линейных уравнений:

```
int n = A.row();
Vector<double> y(n);
for (int i = 0; i < n; i++) {
        y[i] = b[i];
        double sum = 0;
        for (int j = 0; j < i; j++) sum += L(i, j) * y[j];
        y[i] -= sum;
}
Vector<double> x(n);
for (int i = n - 1; i >= 0; i--) {
        x[i] = y[i];
        double sum = 0;
        for(int j = n - 1; j > i; j--) sum += U(i, j) * x[j];
        x[i] = (x[i] - sum) / U(i, i);
}
```

2.3 Вычисление определителя и обратной матрицы LU-разложением

Определитель матрицы легко вычисляется, если она уже разложена на матрицы-сомножители L и U. Пусть матрица A представлена в виде этого разложения:

$$A = LU.$$

Согласно свойствам определителя матрицы из этого равенства вытекает:

$$det(A) = det(L)det(U),$$

откуда, учитывая, что диагональные элементы матрицы L равны единице, получаем

$$det(A) = 1 \cdot det(U) = det(U) = \prod_{i=1}^{n} u_{ii}$$

С помощью LU-разложения можно найти и обратную матрицу, если в обычном алгоритме вычисления обратной матрицы выполнить подстановку $A = LU$, то матричное уравнение будет выглядеть следующим образом:

$$LUX = I,$$

что равносильно

$$\begin{cases} LY = I, \\ UX = Y. \end{cases}$$

Последние два матричных уравнения решаются в полной аналогии с уравнением 1 на стр.9.

Реализация вышеприведенного алгоритма:

```
int n = A.row();
Matrix<double> Y(n);
for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
                Y(i, k) = I(i, k);
                double sum = 0;
                for (int j = 0; j < i; j++) {
                        sum += L(i, j) * Y(j, k);
                }
                Y(i, k) -= sum;
        }
}
Matrix<double> X(n);
for (int k = 0; k < n; k++) {
```

```
        for (int i = n - 1; i >= 0; i--) {
                X(i, k) = Y(i, k);
                double sum = 0;
                for (int j = n - 1; j > i; j--) {
                        sum += U(i, j) * X(j, k);
                }
                X(i, k) = (X(i, k) - sum) / U(i, i);
        }
}
```

2.4 Метод Гаусса и разложение матрицы на множители

Используя LU-разложение, метод Гаусса решения систем линейных уравнений можно разбить на два основных этапа. Первый этап состоит в сохранении LU-разложения матрицы на ее месте, а второй – в преобразовании столбца свободных членов, пользуясь значениями коэффициентов из матрицы L, т.е. в коде можно реализовать две функции, описывающие эти два этапа, причем первый независим. Эту же идею можно использовать при реализации алгоритма, содержащего метод Гаусса с постолбцовым выбором главного элемента и разложение исходной матрицы на множители. Также как и во втором способе LU-разложения матрицы A все преобразования с этой матрицей эквивалентны умножению на нее слева матрицы вида M_k на стр.11, но перед этим еще происходит умножение слева некоторой матрицы перестановок P_k, отвечающую за перестановку строк в матрице A. Все это можно описать соотношением

$$M_{n-1}P_{n-1}...M_2P_2M_1P_1A = A^{(n-1)},$$

откуда получаем

$$A = P_1^{-1}M_1^{-1}P_2^{-1}M_2^{-1}...P_{n-1}^{-1}M_{n-1}^{-1}U,$$

где $U = A^{(n-1)}$. Введя обозначение $\widetilde{L} = P_1^{-1}M_1^{-1}P_2^{-1}M_2^{-1}...P_{n-1}^{-1}M_{n-1}^{-1}$, имеем

$$A = \widetilde{L}U.$$

Матрица $\widetilde{L}$ отличается от матрицы L перестановкой множителей в столбцах, поэтому последнее разложение является разложением не самой матрицы A, а матрицы PA, полученной из матрицы A перестановкой ее строк. Следовательно, решить систему можно получив сначала разложение матрицы $\widetilde{A}$(первый этап), при этом номера переставленных строк

при выборе ведущего элемента следует хранить в каком-нибудь векторе-перестановок размерности n. В худшем случае понадобиться $n-1$ перестановок, поэтому в качестве последней компоненты этого вектора целесообразно хранить число перестановок, чтобы не терять информацию о знаке определителя исходной матрицы. Во втором этапе осуществляются все те преобразования со столбцом свободных членов(для этого есть все необходимые данные, полученные в первом этапе).

Реализация данного алгоритма:

Первый этап

```
void LU_decomp_PM(Matrix<double>& A, Vecotr<int>& p) {
        int n = A.row();
        double eps = 1.0e-10;
        for (int k = 0; k < n - 1; k++) {
                double max = fabs(A(k, k));
                int pos = k;
                for (int t = k + 1; t < n; t++) {
                        if (fabs(A(t, k)) > max) {
                                max = fabs(A(t, k));
                                pos = t;
                        }
                }
                if (pos != k) {
                        p[n - 1]++;
                        for (int j = k; j < n; j++) {
                                A.swap(A(pos, j), A(k, j));
                        }
                }
                for (int i = k + 1; i < n; i++) {
                        if (max < eps) {
                                cout << "Error!" << endl;
                                exit(1);
                        }
                        double tmp = A(i, k) / A(k, k);
                        A(i, k) = tmp;
                        for (int j = k + 1; j < n; j++) {
                                A(i, j) -= tmp * A(k, j);
                        }
                }
                p[k] = pos;
        }
}
```

Второй этап

```
void solve_by_LU_decomp_with_permutations
(Matrix<double> A, Vecotr<int>& p, Vector<double> b) {
        int m = A.row();
        int n = A.col();
        int l = b.size();
        if(m != n || n != l) {
                cout << "Error!" <<endl;
                return;
        }
        LU_decomp_PM(A, p);
        for(int k = 0; k < n - 1; k++) {
                int pos = p[k];
                if (k!= pos) A.swap(b[k], b[pos]);
                for(int i = k + 1; i < n; i++) {
                        b[i] -= A(i, k) * b[k];
                }
        }
        Vector<double> x(n);
        for (int i = n - 1; i >= 0; i--) {
                x[i] = b[i];
                double sum = 0;
                for(int j = n - 1; j > i; j--) {
                        sum += A(i, j) * x[j];
                }
                x[i] = (x[i] - sum) / A(i, i);
        }
        cout << "x: " << x << endl;
}
```

3 Другие методы решения систем линейных уравнений

3.1 Метод прогонки

Возникают ситуации, когда матрица системы линейных уравнений имеет какую-то особенно удобную структуру. В таких случаях применяют более быстрые методы решения систем. Одним из таких является **метод прогонки**. Его применяют при решении систем с ленточными матрицами

$$\begin{aligned}
b_1x_1 + c_1x_2 &= f_1,\\
a_2x_1 + b_2x_2 + c_2x_3 &= f_2,\\
&\cdots\cdots\cdots\cdots\\
a_ix_{i-1} + b_ix_i + c_ix_{i+1} &= f_i,\\
&\cdots\cdots\cdots\cdots\\
a_{n-1}x_{n-2} + b_{n-1}x_{n-1} + c_{n-1}x_n &= f_{n-1},\\
a_nx_{n-1} + b_nx_n &= f_n.
\end{aligned}$$

Преобразуем первое уравнение данной системы к виду

$$x_1 = \alpha_1x_2 + \beta_1,$$

где $\alpha_1 = -c_1/b_1, \beta_1 = f_1/b_1$.

Подставим полученное для x_1 выражение во второе уравнение:

$$a_2(\alpha_1x_2 + \beta_1) + b_2x_2 + c_2x_3 = f_2.$$

Запишем это уравнение в виде

$$x_2 = \alpha_2x_3 + \beta_2,$$

где $\alpha_2 = -c_2/(b_2 + a_2\alpha_1), \beta_2 = (f_2 - a_2\beta_1)/(b_2 + a_2\alpha_1)$.

Полученное выражение для x_2 подставляем в третье уравнение системы и т.д.

Таким образом, i-е уравнение системы преобразуется к виду

$$x_i = \alpha_ix_{i+1} + \beta_i, \tag{3}$$

где $\alpha_i = -c_i/(b_i + a_i\alpha_{i-1}), \beta_i = (f_i - a_i\beta_{i-1})/(b_i + a_i\alpha_{i-1})$.

На n-ном шаге подстановка выражения $x_{n-1} = \alpha_{n-1}x_n + \beta_{n-1}$ дает:

$$a_n(\alpha_{n-1}x_n + \beta_{n-1}) + b_nx_n = f_n.$$

Отсюда можно найти значение x_n:

$$x_n = \beta_n = (f_n - a_n\beta_{n-1})/(b_n + a_n\alpha_{n-1}).$$

Остальные неизвестные значения x_i для $i = n-1, n-2, ..., 1$ легко определяются по формуле 3.
В этом методе процесс вычисления **прогоночных коэффициентов** α_i и $\beta_i (i = \overline{1,n})$ называют **прямым ходом** метода прогонки(**прямой прогонкой**), а процесс нахождения неизвестных называют **обратным ходом** метода прогонки(**обратной прогонкой**).

Алгоритм метода прогонки(на вход подаются матрица A и столбец свободных членов b):

```
int n = A.row();
Vector<double> alpha(n - 1), beta(n + 1), x(n);
double gamma;
gamma = A(0, 0);
alpha[0] = -A(0, 1) / gamma;
beta[0] = f[0] / gamma;
for (int i = 1; i < n - 1; i++) {
        gamma = A(i, i) + A(i, i - 1) * alpha[i - 1];
        alpha[i] = -A(i, i + 1) / gamma;
        beta[i] = (f[i] - A(i, i - 1) * beta[i - 1]) / gamma;
}
gamma = A(n - 1, n - 1) + A(n - 1, n - 2) * alpha[n - 2];
beta[n - 1] = (f[n - 1] - A(n - 1, n - 2) * beta[n - 2]) / gamma;
x[n - 1] = beta[n - 1];
for (int i = n - 2; i >= 0; i--) {
        x[i] = alpha[i] * x[i + 1] + beta[i];
}
```

Пример. Пользуясь методом прогонки, решим ситсему

$$\begin{aligned} 5x_1 - \quad x_2 \qquad\qquad\quad &= 2, \\ 2x_1 + 4.6x_2 - \quad x_3 \qquad\quad &= 3.3, \\ 2x_2 + 3.6x_3 - 0.8x_4 &= 2.6, \\ 3x_3 + 4.4x_4 &= 7.2. \end{aligned}$$

Прямой ход. Вычислим прогоночные коэффициенты:

$\gamma_1 = 5$, $\alpha_1 = 1/\gamma_1 = 1/5 = 0.2$, $\beta_1 = 2/\gamma_1 = 2/5 = 0.4$;
$\gamma_2 = 4.6 + 2\alpha_1 = 4.6 + 2 \cdot 0.2 = 5$, $\alpha_2 = 1/\gamma_2 = 1/5 = 0.2$;
$\beta_2 = (3.3 - 2\beta_1)/\gamma_2 = (3.3 - 2 \cdot 0.4)/5 = 0.5$;
$\gamma_3 = 3.6 + 2\alpha_2 = 3.6 + 2 \cdot 0.2 = 4$, $\alpha_3 = 0.8/\gamma_3 = 0.8/4 = 0.2$;
$\beta_3 = (2.6 - 2\beta_2)/\gamma_3 = (2.6 - 2 \cdot 0.5)/4 = 0.4$;
$\gamma_4 = 4.4 + 3\alpha_3 = 4.4 + 3 \cdot 0.2 = 5$;
$\beta_4 = (7.2 - 3\beta_3)/\gamma_4 = (7.2 - 3 \cdot 0.4)/5 = 1.2$.

Обратный ход. Полагаем $x_4 = \beta_4 = 1.2$. Теперь окончательно находим:

$x_3 = \alpha_3 x_4 + \beta_3 = 0.2 \cdot 1.2 + 0.4 = 0.64$;
$x_2 = \alpha_2 x_3 + \beta_2 = 0.2 \cdot 0.64 + 0.5 = 0.628$;
$x_1 = \alpha_1 x_2 + \beta_1 = 0.2 \cdot 0.628 + 0.4 = 0.5256$.

3.2 Метод Холецкого(метод квадратных корней)

Пусть требуется решить систему с квадратной невырожденной матрицей

$$Ax = b, \tag{4}$$

причем матрица A – положительно определенная. Согласно критерию Сильвестра все ее угловые миноры должны быть положительными. Также потребуем симметричность от этой матрицы. Идея метода Холецкого(метода квадратных корней) состоит в том, чтобы разложить такую матрицу на матрицы-сомножители

$$A = LL^T.$$

В отличии от LU-разложения диагональные элементы матрицы L уже необязательно равны единице, но они должны быть положительными. Фактически достаточно найти элементы матрицы L, а затем чтобы получить второй сомножитель, просто транспонировать эту матрицу. При этом L^T, очевидно, также треугольная, но уже верхняя треугольная матрица.

Далее, так же, как при решении систем LU-разложением, матричное уравнение (4) разбивают на два более простых

$$\begin{cases} Ly = b, \\ L^T x = y. \end{cases}$$

Для того чтобы найти элементы матрицы L, вычислим элементы матрицы LL^T и приравняем их соответствующим элементам матрицы A. Тогда получим следующую систему уравнений

$$\begin{cases} l_{11}^2 = a_{11}^2, \\ l_{i1} l_{11} = a_{i1}, \qquad i = \overline{2,n}, \\ l_{21}^2 + l^2 22 = a_{22}, \\ l_{i1} l_{21} + l_{i2} l_{22} = a_{i2}, \qquad i = \overline{3,n}, \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ l_{k1}^2 + l_{k2}^2 + \ldots + l_{kk}^2 = a_{kk}, \\ l_{i1} l_{k1} + l_{i2} l_{k2} + l_{ik} l_{kk} = a_{ik}, \qquad i = \overline{k+1,n}, \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ l_{n1}^2 + l_{n2}^2 + l_{nn}^2 = a_{nn}. \end{cases}$$

Таким образом, все элементы матрицы L в этом разложении отличные от нуля вычисляются по формулам:

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2} \qquad (i = \overline{1,n}), \tag{5}$$

$$l_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} l_{ik} l_{jk}}{l_{ii}} \qquad (j = \overline{2,n}, \quad i > j).$$

Алгоритмически выгоднее просто каждый раз проверять, положительно ли подкоренное выражение в формуле 5, чем сначала проверить всю исходную матрицу на положтельную определенность(из последнего вытекает положительность подкоренного выражения). Этот метод также называют методом квадратных корней, т.к. для вычисления диагональных элементов матрицы L производится операция извлечения квадратного корня.

Алгоритм метода Холецкого(на вход подаются матрица A и столбец свободных членов b):

```
int m = A.row();
int n = A.col();
int l = b.size();
if(m != n || n != l) {
        cout << "Error!" <<endl;
        return;
}
Matrix<double> create;
if (A != create.transposed(A)) {
        cout << "Error!" << endl;
        return;
}
Matrix<double> L(n);
for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
                double sum = 0;
                if (i == j) {
                        for (int k = 0; k < j; k++) {
                                sum += (L(j, k) * L(j, k));
                        }
                        if (A(i, j) < sum) {
                                cout << "Error!" << endl;
                                return;
                        }
                        L(i, j) = sqrt(A(i, j) - sum);
                }
                else if (i > j) {
                        for (int k = 0; k < j; k++) {
                                sum += (L(i, k) * L(j, k));
                        }
                        L(i, j) = (A(i, j) - sum) / L(j, j);
                }
        }
}
Matrix<double> Lt = create.transposed(L);
Vector<double> y(n);
for (int i = 0; i < n; i++) {
        y[i] = b[i];
        double sum = 0;
        for (int j = 0; j < i; j++) {
                sum += L(i, j) * y[j];
        }
        y[i] = (y[i] - sum) / L(i, i);
}
Vector<double> x(n);
for (int i = n - 1; i >= 0; i--) {
```

```
        x[i] = y[i];
        double sum = 0;
        for(int j = n - 1; j > i; j--) {
                sum += Lt(i, j) * x[j];
        }
        x[i] = (x[i] - sum) / Lt(i, i);
}
cout << "x: " << x << endl;
```

Пример. Используя метод Холецкого, решим систему

$$\begin{aligned} 6.25x_1 - x_2 + 0.5x_3 &= 7.5, \\ -x_1 + 5x_2 + 2.12x_3 &= -8.68, \\ 0.5x_1 + 2.12x_2 + 3.6x_3 &= -0.24. \end{aligned}$$

Вычислим элементы матрицы L:

$$\begin{aligned} &l_{11} = \sqrt{a_{11}} = \sqrt{6.25} = 2.5, l_{21} = \frac{a_{21}}{l_{11}} = -\frac{1}{2.5} = -0.4; \\ &l_{31} = \frac{a_{31}}{l_{11}} = \frac{0.5}{2.5} = 0.2, l_{22} = \sqrt{a_{22} - l_{21}^2} = \sqrt{5 - 0.16} = 2.2; \\ &l_{32} = \frac{(a_{32} - l_{31}l_{21})}{l_{22}} = \frac{(2.12 - 0.2 \cdot (-0.4))}{2.2} = 1; \\ &l_{33} = \sqrt{a_{33} - l_{31}^2 - l_{32}^2} = \sqrt{3.6 - 0.2^2 - 1^2} = 1.6. \end{aligned}$$

Тогда матрица L имеет вид:

$$\begin{bmatrix} 2.5 & 0 & 0 \\ -0.4 & 2.2 & 0 \\ 0.2 & 1 & 1.6 \end{bmatrix}$$

Система $Ly = b$ такова:

$$\begin{aligned} 2.5y_1 &= 7.5, \\ -0.4y_1 + 2.2y_2 &= -8.68, \\ 0.2y_1 + y_2 + 1.6y_3 &= -0.24. \end{aligned}$$

Решая ее, находим $y_1 = 3, y_2 = -3.4, y_3 = 1.6$. Окончательно из системы $L^T x = y$

$$\begin{aligned} 2.5x_1 - 0.4x_2 + 0.2x_3 &= 3, \\ 2.2x_2 + x_3 &= -3.4, \\ 1.6x_3 &= 1.6, \end{aligned}$$

находим решение $x_1 = 0.8, x_2 = -2, x_3 = 1$.

4 QR-разложение матрицы

4.1 Процесс ортогонализации Грама-Шмидта

Помимо LU-разложения, есть и другие разложения матриц на множители, которыми удобно представлять исходные матрицы. Одним из таких является QR-разложение, где Q – ортогональная матрица, а R – верхняя треугольная. Одним из способов получения такого разложения является процесс ортогонализации Грама-Шмидта. Суть его состоит в том, чтобы построить ортонормированный базис, отталкиваясь от некоторого исходного базиса. Изложим этот алгоритм.

Пусть $a = (a_1, ..., a_n)$ - некоторый базис в n-мерном евклидовом пространстве E. Будем строить новый, уже ортонормированный базис $q = (q_1, ..., q_n)$. Последовательно вычисляем векторы g_1 и q_1, g_2 и q_2 и т.д. по формулам:

$$\begin{array}{ll} g_1 = a_1, & q_1 = \frac{g_1}{||g_1||_2}; \\ g_2 = a_2 - (a_2, q_1)q_1, & q_2 = \frac{g_2}{||g_2||_2}; \\ g_3 = a_3 - (a_3, q_1)q_1 - (a_3, q_2)q_2, & q_3 = \frac{g_3}{||g_3||_2}; \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & \ldots\ldots\ldots\ldots \\ g_n = a_n - (a_n, q_1)q_1 - ... - (a_n, q_{n-1})q_{n-1}, & q_n = \frac{g_n}{||g_n||_2}. \end{array} \qquad (6)$$

Вот как выглядел бы псевдокод данного алгоритма:

$for\ j = 1, ..., n$
    $gj = aj$
    $for\ i = 1, ...j - 1$
        $\alpha = (a_j, q_i)$
    $g_j = g_j - \alpha q_i$
    $if\ ||g_j||_2 = 0$
        $exit$
    $q_j = g_j / ||g_j||_2.$

Данный алгоритм можно модифицировать. Это можно сделать, если при вычислении коэффициента α находить вместо скалярного произведения векторов aj и qi, скалярное произведение векторов gj и qi. Формально результат не изменится, поскольку $(g_j, q_i) = (a_j - \sum_{k=1}^{j-1} \alpha_k q_k, q_i) = (a_j, q_i) - (\sum_{k=1}^{j-1} \alpha_k q_k, q_i) = (a_j, q_i)$(в силу попарной ортогональности векторов q_k, q_i, $(\sum_{k=1}^{j-1} \alpha_k q_k, q_i) = 0$). Данная модификация повышает численную устойчивость алгоритма. Это объясняется тем, что вектор g_j обладает минимальной нормой среди всех векторов вида $a_j - \sum_{k=1}^{j-1} \beta_k q_k$, где β_k - произвольные коэффициенты. Поэтому при скалярном умножении на g_j ошибки накапливаются существенно медленнее.

Заметим, что ситуации с делением на ноль в выражении $if\ ||g_j||_2 = 0$ возникнуть не может. Предположим, что норма вектора g_j оказалась равной нулю, тогда отсюда следует, что сам вектор g_j нулевой. Тогда заключаем, что

$$a_j = (a_j, q1)q1 + ... + (a_j, q_{j-1})q_{j-1},$$

т.е. вектор a_j является линейной комбинацией векторов $q_1, ..., q_{j-1}$, которые в силу (6) выражаются через векторы $a_1, ..., a_{j-1}$. Следовательно, этот вектор является линейной комбинацией системы векторов $a_1, ..., a_{j-1}$, а система векторов $a1, ..., a_{j-1}, a_j$ линейно зависима. Но это противоречит определению базиса $a_1, ..., a_n$. При вычислении QR-разложения матрицы роль векторов в процессе ортогонализации играют столбцы исходной матрицы, в результате матрица Q составлена из полученной ортонормированной системы вектор-столбцов исходной матрицы. Далее, полагая $A = QR$, где A имеет размеры m и n, Q – те же размеры, R имеет порядок n, и из того, что матрица Q ортогональная выходит, что

$$R = Q^{-1}A = Q^TA.$$

Отсюда получаем $$r_{ij} = \sum_{k=1}^{m} q_{ik}^t a_{kj} = \sum_{k=1}^{m} q_{ki} a_{kj}. \qquad (7)$$

Из ранее отмеченного вытекает, что для осуществления ортогонализации столбцов матрицы A необходимо, чтобы они все были линейно независимы.

Реализация данного алгоритма с модификацией для матрицы A размеров m, n(m ≥n):

```
int m = A.row();
int n = A.col();
Matrix<double> R(n), G(m, n), Q(m, n);
for (int j = 0; j < n; j++) {
        for (int i = 0; i < m; i++) G(i, j) = A(i, j);
        for (int i = 0; i < j; i++) {
                double scalar = 0;
                for (int k = 0; k < m; k++) {
                        scalar += A(k, j) * Q(k, i);
                }
                R(i, j) = scalar;
                for (int k = 0; k < m; k++) {
                        G(k, j) -= Q(k, i) * scalar;
                }
        }
        double norm = 0;
```

```
        for (int k = 0; k < m; k++) norm += G(k, j) * G(k, j);
        norm = sqrt(norm);
        if (!norm) return;
        for (int k = 0; k < m; k++) Q(k, j) = G(k, j) / norm;
        R(j, j) = norm;
}
```

Пример. Выполним QR-разложение матрицы, используя процесс ортогонализации Грамма-Шмидта

$$\begin{bmatrix} 1 & 2 & 2 \\ -1 & 0 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

Положим, что $a_1 = (1, -1, 0)$, $a_2 = (2, 0, 0)$, $a_3 = (2, 2, 1)$. Тогда соглсасно формулам (6) имеем:

$g_1 = a_1$, $q_1 = \frac{g_1}{||g_1||_2} = \frac{1}{\sqrt{2}}(1, -1, 0) = (\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}, 0)$;
$g_2 = a_2 - (a_2, q_1)q_1 = (2, 0, 0) - \frac{2}{\sqrt{2}}(\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}, 0) =$
$(2, 0, 0) - (1, -1, 0) = (1, 1, 0)$,
$q_2 = \frac{g_2}{||g_2||_2} = \frac{1}{\sqrt{2}}(1, 1, 0) = (\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0)$;
$g_3 = a_3 - (a_3, q_1)q_1 - (a_3, q_2)q_2 = (2, 2, 1) - \frac{4}{\sqrt{2}}(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0)$
$= (2, 2, 1) - (2, 2, 0) = (0, 0, 1)$, $q_3 = \frac{g_3}{||g_3||_2} = (0, 0, 1)$.

С учетом формулы (7) находим элементы матрицы R:

$r_{11} = ||g_1||_2 = \sqrt{2}$; $r_{12} = (a_2, q_1) = \frac{2}{\sqrt{2}} = \sqrt{2}$;
$r_{13} = (a_3, q_1) = 0$; $r_{22} = ||g_2||_2 = \sqrt{2}$;
$r_{23} = (a_3, q_2) = \frac{4}{\sqrt{2}} = 2\sqrt{2}$; $r_{33} = ||g_3||_2 = 1$.

Следовательно, искомое разложение имеет вид

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \sqrt{2} & \sqrt{2} & 0 \\ 0 & \sqrt{2} & 2\sqrt{2} \\ 0 & 0 & 1 \end{bmatrix}.$$

4.2 Метод Хаусхолдера(отражений)

Ортогонализация Грама-Шмидта не единственный способ получить QR-разложение матрицы. Еще одним эффективным методом является метод Хаусхолдера или метод отражений, суть которого заключается в преобразованиях матриц матрицами Хаусхолдера. Для начала выявим смысл матрицы Хаусхолдера.

Пусть w – фиксированный вектор-столбец евклидова пространства R_n с единичной нормой $||w||_2 = 1$ размерности n. Матрицей Хаусхолдера называют матрицу размеров $n \times n$ вида

$$H = I - 2ww^T.$$

Пусть x – пока произвольный вектор-столбец той же размерности, что и w. Тогда его образ $y = Hx$ имеет вид

$$y = Hx = x - 2xww^T = x - 2(x, w)w \tag{8}$$

Далее, потребуем, чтобы вектор x был коллинеарен вектору w, т.е. $x = \alpha w$, где $\alpha = const(\alpha \neq 0)$. Тогда из выражения (8) для вектора y получаем

$$y = x - 2(\alpha w, w)w = x - 2\alpha(w, w)w = x - 2\alpha w = x - 2x = -x.$$

Если потребуем, чтобы вектор x был ортогонален вектору w, тогда из того же соотношения (8) имеем

$$y = x - 0 \cdot w = x.$$

Таким образом, если вектор x ортогонален вектору w, то матрица Хаусхолдера H его не изменяет, если же он ему коллинеарен, то преобразование Хаусхолдера переводит его в противоположный – отражает. Осуществляя умножение ww^T

$$ww^T = \begin{pmatrix} w_1 \\ w_2 \\ ... \\ w_n \end{pmatrix} \cdot \begin{pmatrix} w1 & w2 & ... & w_n \end{pmatrix} = \begin{pmatrix} w_1^2 & w_1w_2 & ... & w_1w_n \\ w_2w_1 & w_2^2 & ... & w_2w_n \\ ... & ... & ... & ... \\ w_nw_1 & w_nw_2 & ... & w_n^2 \end{pmatrix},$$

убеждаемся в том, что матрица ww^T симметричная, а следовательно и матрица H. Пользуясь этим получаем

$$HH^T = H^2 = I - 4ww^T + 4w(w, w)w^T = I.$$

Т.е. матрица Хаусхолдера еще и ортогональна.

Построим теперь матрицу H так, чтобы она преобразовывала вектор x в вектор коллинеарный вектору $e_1 = (1, 0, ..., 0)^T$, т.е. чтобы занулились все координаты вектора x кроме первой. Очевидно, преобразование Хаусхолдера не меняет длину(норму$_2$) вектора. Пусть $y = (y_1, 0, ..., 0)$ – это полученный образ вектора x. Тогда

$$||y||_2 = |y_1| = ||x||_2 \Rightarrow y_1 = \pm||x||_2 \Rightarrow y = \pm||x||_2 e_1$$

Подставляя последнее выражение для вектора y в равенство (8), имеем

$$\pm||x||_2 e_1 = x - 2(x, w)w \Leftrightarrow \tilde{w} = (x, w)w = x \mp ||x||_2 e_1. \tag{9}$$

Следовательно, нормируя вектор $\tilde{w}$, при построении матрицы Хаусхолдера нужно взять вектор

$$w = \frac{\tilde{w}}{||w||_2}.$$

Знак в выражении (9) для вектора $\tilde{w}$ подбирают так, чтобы не происходило вычитания(чтобы не терялись значащие цифры при вычитании близких чисел для более устойчевого процесса вычислений). Чтобы этого достигнуть, положим

$$\tilde{w} = (x_1 - \beta, x_2, ..., x_n)^T,$$

где

$$\beta = \begin{cases} ||x||_2, \ \textbf{если } x_1 \leq 0 \\ -||x||_2, \ \textbf{если } x_1 > 0. \end{cases} \tag{10}$$

Норму вектора можно вычислить теперь, пользуясь выражением для β:

$$||\tilde{w}||_2^2 = (x_1 - \beta)^2 + x_2^2 + ... + x_n^2 = 2\beta^2 - 2\beta x_1.$$

Таким образом, матрицу Хаусхолдера H для ранее описанного преобразования можно построить с помощью вектора $w = \gamma(x_1 - \beta, x_2, ..., x_n)^T$, где β определяется выражением (10), и

$$\gamma = \frac{1}{\sqrt{2\beta(\beta - x_1)}}.$$

При этом у полученного вектора-образа первая координата равна β, а все остальные нулевые.

Строя матрицу Хаусхолдера таким образом, можно получить QR-разложение матрицы. Пусть нам дана матрица A порядка n. Мы можем привести ее к треугольному виду, пользуясь построением необходимых матриц Хаусхолдера. Это можно сделать, если строить матрицу Хаусхолдера с помощью вектора-столбца матрицы A. На первом шаге роль вектора играет первый столбец матрицы A, затем полученную матрицу H_1 умножаем слева на матрицу A, в результате чего она преобразуется к виду

$$A^{(1)} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & ... & a_{1n}^{(1)} \\ 0 & a_{22}^{(1)} & ... & a_{2n}^{(1)} \\ 0 & a_{32}^{(1)} & ... & a_{3n}^{(1)} \\ ... & ... & ... & ... \\ 0 & a_{m2}^{(1)} & ... & a_{nn}^{(1)} \end{pmatrix}$$

Далее, на втором шаге рассматриваем матрицу, полученную из этой вычеркиванием первых строки и столбца, и в ней уже роль вектора Хаусхолдера играет снова первый столбец размерности на один меньше предыдущего. Понятно, что матрица H_2 уже будет также иметь порядок на один меньше предыдущей, поэтому необходимо достроить ее до матрицы порядка n, т.к. потом мы ее умножим слева на матрицу $A^{(1)}$, как в первом шаге. Достроим ее следующим образом:

$$P_2 = \begin{pmatrix} 1 & 0 \\ 0 & H_2 \end{pmatrix}$$

Теперь поступая как в первом шаге, получаем

$$P_2 A^{(1)} = A^{(2)} = \begin{pmatrix} a_{11}^{(2)} & a_{12}^{(2)} & ... & a_{1n}^{(2)} \\ 0 & a_{22}^{(2)} & ... & a_{2n}^{(2)} \\ 0 & 0 & ... & a_{3n}^{(2)} \\ ... & ... & ... & ... \\ 0 & 0 & ... & a_{nn}^{(2)} \end{pmatrix}$$

и т.д. На k-ом шаге матрица A преобразуется к виду

$$P_k A^{(k-1)} = A^{(k)} = \begin{pmatrix} a_{11}^{(k)} & a_{12}^{(k)} & ... & ... & a_{1k}^{(k)} & ... & a_{1n}^{(k)} \\ 0 & a_{22}^{(k)} & ... & ... & a_{2k}^{(k)} & ... & a_{2n}^{(k)} \\ 0 & 0 & ... & ... & a_{3k}^{(k)} & ... & ... \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & 0 & ... & ... & a_{kk}^{(k)} & ... & a_{kn}^{(k)} \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & 0 & ... & ... & 0 & ... & a_{nn}^{(k)} \end{pmatrix}.$$

Матрица P_k достраивается до матрицы порядка n как

$$P_k = \begin{pmatrix} I & O \\ O & H_k \end{pmatrix},$$

где матрицы I, O – соответственно единичная, нулевая матрицы $(k-1)$-го порядка. При $k = 1$ полагаем $P_k = H_k$. Обозначив $Q = P_1 P_2 ... P_{n-1}$ и $R = P_{n-1} ... P_2 P_1 A$, получаем искомое QR-разложение матрицы A.

Фактическое построение матрицы P_k с помощью выражений для β и γ, полученных на стр.28, дает матрицу

$$\begin{pmatrix} 1 & ... & 0 & 0 & 0 & ... & 0 \\ .... & ... & ... & ... & ... & ... & ... \\ 0 & ... & 1 & 0 & 0 & ... & 0 \\ 0 & ... & 0 & 1 - 2\gamma_k^2 \tilde{w}_k^2 & -2\gamma_k^2 \tilde{w}_k a_{k+1,k}^{(k-1)} & ... & -2\gamma_k^2 \tilde{w}_k a_{n,k}^{(k-1)} \\ 0 & ... & 0 & -2\gamma_k^2 \tilde{w}_k a_{k+1,k}^{(k-1)} & 1 - 2\gamma_k^2 a_{k+1,k}^{(k-1)} a_{k+1,k}^{(k-1)} & ... & -2\gamma_k^2 a_{k+1,k}^{(k-1)} a_{nk}^{(k-1)} \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & ... & 0 & -2\gamma_k^2 \tilde{w}_k a_{n,k}^{(k-1)} & -2\gamma_k^2 a_{k+1,k}^{(k-1)} a_{nk}^{(k-1)} & ... & 1 - 2\gamma_k^2 a_{n,k}^{(k-1)} a_{n,k}^{(k-1)} \end{pmatrix},$$

где $\tilde{w}_k = a_{kk}^{(k-1)} - \beta_k$.

Реализация данного алгоритма с матрицей A:

```
int m = A.row();
int n = A.col();
int end = m - 1;
if (m > n) end = n;
double beta, gamma, norm;
Matrix<double> create, I = create.I(m);
Matrix<double> Q = I, R = A;
for (int i = 0; i < end; i++) {
        norm = 0;
        for (int k = i; k < m; k++) norm += R(k, i) * R(k, i);
        if (!norm) continue;
        if (R(i, i) <= 0) beta = sqrt(norm);
        else beta = -sqrt(norm);
        gamma = 1 / (beta * (beta - R(i, i)));
        R(i, i) -= beta;
        Matrix<double> H = I;
        for (int s = i; s < m; s++) {
                for (int j = i; j < m; j++) {
                        if (j != s) {
                                H(s, j) = -gamma * R(s, i) *
                                                R(j, i);
                        }
                        else {
```

```
                                H(s, j) = 1 - gamma * R(s, i) *
                                                    R(s, i);
                        }
                        H(j, s) = H(s, j);
                    }
                }
            }
            Q *= H;
            R(i, i) += beta;
            R = H * R;
            for (int s = i + 1; s < m; s++) R(s, i) = 0;
}
```

Пример. Используя метод Хаусхолдера(отражений), найдем QR-разложение матрицы

$$\begin{bmatrix} 0 & 2 & 1 \\ 0 & 0 & 1 \\ 0 & -1 & 1 \end{bmatrix}.$$

$P_1 = I \Rightarrow A_1 = P_1 A = A;$

$\beta_2 = ||(0,0,-1)^T||_2 = 1,\ \gamma_2 = \frac{1}{\beta_2 \sqrt{2}} = \frac{1}{\sqrt{2}};$

$w_2 = \frac{1}{\sqrt{2}}(0,-1,-1)^T;$

$$P_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - 2 \cdot \frac{1}{2} \cdot \begin{bmatrix} 0 \\ -1 \\ -1 \end{bmatrix} \cdot \begin{bmatrix} 0 & -1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix};$$

$R = A_2 = P_2 A_1,\ Q = P_1 P_2.$

Следовательно, искомое разложение имеет вид

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 2 & 1 \\ 0 & 1 & -1 \\ 0 & 0 & -1 \end{bmatrix}.$$

5 Список литературы

[1] Вычислительные методы: учебное пособие, четвертое издание. А.А. Амосов, Ю.А. Дубинский, Н.В. Копченова.

[2] Основы численных методов: учебник для вузов. В.М. Вержбицкий.