Белорусский государственный университет
Факультет прикладной математики и информатики

Методы решения СЛАУ

Отчет по лабораторной работе
студента 2 курса 3 группы
Аквуха Джеймса

Преподаватель:
Будник А. М.

Минск 2015

Постановка задачи

Дана СЛАУ

$$Ax = b,\ A = \begin{pmatrix} a_{11} & \cdots & a_{1,n} \\ \cdots & \cdots & \cdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix},\ b = \begin{pmatrix} b_1 \\ \cdots \\ b_n \end{pmatrix},\ x = \begin{pmatrix} x_1 \\ \cdots \\ x_n \end{pmatrix}.$$

Требуется решить систему, оценить ее устойчивость и погрешность полученного решения. В итерационных методах также необходимо исследовать вопрос сходимости.

Теоретические сведения

1. Метод Гаусса с выбором главного элемента по строкам и столбцам

Эквивалентные системы уравнений

Две системы линейных уравнений называются эквивалентными, если каждое решение одной из них является решением другой. Процесс решения системы линейных уравнений состоит в последовательном преобразовании её в эквивалентную систему с помощью элементарных преобразований, которыми являются:

- перестановка любых двух уравнений системы;
- умножение обеих частей любого уравнения системы на отличное от нуля число;
- прибавление к любому уравнению другого уравнения, умноженного на любое число;
- вычёркивание уравнения, состоящего из нулей, т.е. уравнения вида $0 \cdot x_1 + 0 \cdot x_2 + \ldots + 0 \cdot x_n = 0$.

Прямой ход метода Гаусса

Рассмотрим систему m линейных уравнений с n неизвестными:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2, \\ \ldots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m. \end{cases}$$

$$a_{kj}^{(0)} = a_{kj}, \quad k, j = \overline{1, n}$$

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{a_{kj}^{(k-1)}}{a_{kk}^{(k-1)}}, \quad i, j = \overline{k+1, n}, k = \overline{1, n-1}$$

$$b_k^{(0)} = b_k, \quad k = \overline{1,n}$$

$$b_i^{(k)} = b_i^{(k-1)} - a_{ik}^{(k-1)} * \frac{b_k^{(k-1)}}{a_{kk}^{(k-1)}} \quad i = \overline{k+1,n}, k = \overline{1,n-1}$$

Обратный ход метода Гаусса

Полученная система уравнений

$$\begin{cases} a_{11}^0 x_1 + a_{12}^0 x_2 + a_{13}^0 x_3 + \ldots + a_{1n}^0 x_n = b_1^0, \\ a_{22}^0 x_2 + a_{23}^0 x_3 + \ldots + a_{2n}^0 x_n = b_2^0, \\ a_{33}^0 x_3 + \ldots + a_{3n}^0 x_n = b_3^0, \\ \ldots \\ a_{nn}^0 x_n = b_n^0 \end{cases}$$

$$x_i = b_i^{(n-1)} - \sum_{j=i+1}^{n} a_{ij}^{(n-1)} x_j, i = \overline{n-1,1}, x_n = b_n^{(n-1)}$$

2. Метод квадратного корня

Суть метода квадратного корня состоит в следующем. Пусть дана линейная система, где $A=(a_{ij})$ - симметрическая квадратная матрица порядка n, т.е. $A^T=(a_{ij})=A$, $\bar{b}=(b_1,\ldots,b_n)^T$ - вектор правых частей системы, $\bar{x}=(x_1,\ldots,x_n)^T$ - вектор-столбец неизвестных.
Решение исходной системы проведем в два этапа.

I этап (прямой ход)

Согласно следствию из $LU-теоремы$ представим матрицу A в виде $A=G*G^T$

где G – нижняя треугольная матрица, G^T – транспонированная по отношению к G матрица. Пусть

$$G = \begin{pmatrix} g_{11} & 0 & 0 & \cdots & 0 \\ g_{12} & g_{22} & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ g_{1n} & g_{2n} & g_{3n} & \cdots & g_{nn} \end{pmatrix}$$

Тогда, производя перемножение матриц G и G^T получим следующие уравнения для определения элементов g_{ij} матрицы G :

$$g_{1i}g_{1j} + g_{2i}g_{2j} + \ldots + g_{ii}g_{ij} = a_{ij}, \quad i < j$$
$$g_{1i}^2 + g_{2i}^2 + \ldots + g_{ii}^2 = a_{ii}.$$

$$g_{11} = \sqrt{a_{11}}, \quad g_{1j} = \frac{a_{1j}}{g_{11}}, \quad j > 1;$$
$$g_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} g_{ki}^2}, \quad 1 < i \le n;$$
$$g_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} g_{ki}g_{kj}}{g_{ii}}, \quad i < j;$$
$$g_{ij} = 0, \quad i > j.$$

$$\det(A) = \det(G) \cdot \det(G^T) = (g_{11} \cdot g_{22} \cdot \ldots \cdot g_{nn})^2 \neq 0$$

при действительных a_{ij} могут получиться чисто мнимые g_{ij} . Формально метод применим и в этом случае. Если же матрица A является положительно определенной, то мнимых g_{ij} не будет.

II этап (обратный ход)

Исходная система эквивалентна двум системам с треугольной матрицей:

$$G\overline{y} = \overline{b}, \; G^T\overline{x} = \overline{y}$$

Или в развернутом виде:

$$\begin{cases} g_{11}y_1 = b_1, \\ g_{12}y_1 + g_{22}y_2 = b_2, \\ \ldots\ldots\ldots\ldots\ldots\ldots \\ g_{1n}y_1 + g_{2n}y_2 + \ldots + g_{nn}y_n = b_n \end{cases}$$

$$\begin{cases} g_{11}x_1 + g_{12}x_2 + \ldots + g_{1n}x_n = y_1, \\ \qquad g_{22}x_2 + \ldots + g_{2n}x_n = y_2, \\ \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \qquad\qquad\qquad g_{nn}x_n = y_n. \end{cases}$$

$$y_1 = \frac{b_1}{g_{11}},$$
$$y_i = \frac{b_i - \sum_{k=1}^{i-1} g_{ki}y_k}{g_{ii}}, \quad i > 1.$$

и

$$x_n = \frac{y_n}{g_{nn}},$$

$$x_i = \frac{y_i - \sum_{k=i+1}^{n} g_{ik} x_k}{g_{ii}}, \quad i < n.$$

3. Метод прогонки

Метод прогонки является частным случаем метода Гаусса и применяется к системам с трех-пятидиагональной матрицей. Такие системы часто встречаются при численном решении краевых задач для дифференциальных уравнений второго порядка, при моделировании некоторых инженерных задач. Если при решении таких систем применять метод Гаусса, то расчет можно организовать таким образом, чтобы не включать нулевые элементы матрицы. Этим самым экономится требуемая память и уменьшается объем вычислений. Указанное ускорение вычислений допускают системы линейных алгебраических уравнений с ленточными, блочными, квазитреугольными, почти треугольными и другими матрицами.
Запишем систему в каноническом виде:

$$a_i x_{i-1} - b_i x_i + c_i x_{i+1} = d_i, 1 \le i \le n, \; a1 = c_n = 0 \;\;,$$

где

$$A = \begin{pmatrix} c_1 & -b_1 & 0 & \dots & 0 & 0 & 0 \\ -a_2 & c_2 & -b_2 & \dots & 0 & 0 & 0 \\ 0 & -a_3 & c_3 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & -a_{n-1} & c_{n-1} & -b_{n-1} \\ 0 & 0 & 0 & \dots & 0 & -a_n & c_n \end{pmatrix}$$

Предполагается, что $b_i \neq 0, \; i = \overline{1,n}$
Метод прогонки состоит из двух этапов: прямой прогонки и обратной прогонки. На первом этапе определяются прогоночные коэффициенты, а на втором – находят неизвестные:

$$\xi_i = \frac{a_i}{c_i - \xi_{i+1} b_i}, \quad i = \overline{n-1, 1}, \quad \xi_n = \frac{a_n}{c_n}$$

$$\eta_i = \frac{f_i + \eta_{i+1} b_i}{c_i - \xi_{i+1} b_i}, \quad i = \overline{n-1, 0}, \quad \eta_n = \frac{f_n}{c_n}$$

$$x_{i+1} = \xi_{i+1} x_i + \eta_{i+1}, \quad i = \overline{0, n-1}, \quad x_0 = \eta_0$$

$$a_i \neq 0, \quad b_i \neq 0, |c_i| \geq |a_i| + |b_i|, \quad i = \overline{1, n-1}, \quad |c_0| \geq |b_0|, \quad |c_n| \geq |a_n|,$$

Для вычисления определителя исходной матрицы достаточно выполнения прямого хода метода прогонки:

$$det(A) = c_n \left(\prod_{i=0}^{n-1} c_i - \xi_{i+1} b_i \right)$$

4. Метод простой итерации (метод Якоби)

Если в системе линейных уравнений много неизвестных, прямые методы решения СЛАУ, позволяющие получить более точное решение, становятся неэффективными. В таких случаях для нахождения корней системы более удобно пользоваться приближёнными итерационными методами. К простейшим итерационным методам решения СЛАУ относятся метод простой итерации и метод Зейделя.
Предположим, что диагональные коэффициенты исходной матрицы не равны 0.
Разрешим первое уравнение системы относительно x_1, другое – относительно x_2 и т. д.
Получим эквивалентную систему уравнений: $\bar{x} = \alpha \bar{x} + \bar{\beta}$, где

$$a_{ij} = -\frac{a_{ij}}{a_{ii}}, i \neq j; \; a_{ij} = 0, i = j; \; \beta_i = \frac{b_i}{a_{ii}}$$

Систему такого вида называют приведенной.
Последовательность приближений строится согласно правилу

$$\bar{x}^{(k+1)} = \alpha \bar{x}^{(k)} + \bar{\beta}$$

Начальное приближение выбирается, вообще говоря, произвольно.
Процесс простой итерации для приведенной системы сходится к единственному решению независимо от выбора начального приближения, если какая-либо норма матрицы α меньше единицы.
Таким образом, для итерационного процесса достаточным условием сходимости является условие:

$$\|\alpha\| < 1$$

Можно доказать, что при выполнении достаточного признака сходимости итерационного процесса оценка метода простой итерации будет следующая:

$$\|\bar{x} - \bar{x}^{(k+1)}\| < \frac{\|\alpha\|^{k+1}}{1 - \|\alpha\|} \|\beta\|$$

В качестве начального приближения можно выбрать вектор свободных членов.
На практике процесс итераций прерывается, если $max|\bar{x}^{(k+1)} - \bar{x}^{(k)}| < \epsilon$, где ε - заданная точность.

5. Метод Гаусса-Зейделя

Метод Зейделя представляет собой модификацию метода простой итерации. Основная его идея заключается в том, что при вычислении $(k+1)$ – ого приближения неизвестной x_i, используются вычисленные ранее $(k+1)$ -ые приближения неизвестных $x_{1,} x_{2,} \ldots, x_{i-1}$.
Теорема сходимости для метода простой итерации справедлива и для метода Зейделя.
Как правило, метод Зейделя даёт лучшую сходимость, чем метод простой итерации

6. Метод минимальных невязок

Метод минимальных невязок относится к итерационным методам вариационного типа. Функционалом в методе минимальных невязок является функционал невязки:

$$\left(r^{0,}r^{0}\right)=\left\|r^{0}\right\|^{2}$$

В итерационном процессе:

$$x^{k+1}=x^{k}-\tau_{k+1}*r^{k}$$

Рассматриваемый функционал достигает минимума при:

$$\tau_{k+1}=\frac{\left(Ar^{k},r^{k}\right)}{\left(Ar^{k},Ar^{k}\right)}$$

Для того, чтобы рассматриваемый метод сходился, достаточно чтобы матрица А являлась симметричной и положительно определенной. При этом для погрешности метода справедлива оценка:

$$\left\|A\left(x^{k}-x\right)\right\| \le \rho_{0}^{k}\left\|A\left(x^{0}-x\right)\right\|,\ \rho_{0}=\frac{1-\xi}{1+\xi},\ \xi=\frac{\lambda_{min}}{\lambda_{max}}$$

Результат выполнения

```
Gauss elimination method with complete pivoting:
Given matrix:
    4.9970e-01   -6.5800e-02    1.3200e-02    2.6300e-02    9.2100e-02   -2.8141e+00
    6.8400e-02    7.8240e-01    0.0000e+00   -5.2600e-02    5.2600e-02    2.4104e+00
    3.9500e-02    0.0000e+00    6.2860e-01   -1.8410e-01    1.0520e-01    2.2828e+00
   -7.8900e-02    1.6570e-01    0.0000e+00    6.1810e-01   -2.6300e-02   -1.6332e+00
    3.2880e-01    0.0000e+00    1.1840e-01    1.3200e-02    7.3640e-01    1.8936e+00
Triangulized matrix:
    1.0000e+00    8.7423e-02    6.7229e-02   -6.7229e-02    0.0000e+00    3.0808e+00
    0.0000e+00    1.0000e+00    1.9096e-01    4.3281e-02    2.6115e-02   -5.1664e+00
    0.0000e+00    0.0000e+00    1.0000e+00   -1.5301e-03    1.6302e-01    5.3329e+00
    0.0000e+00    0.0000e+00    0.0000e+00    1.0000e+00    8.8986e-03   -3.9820e+00
    0.0000e+00    0.0000e+00   -1.3878e-17    0.0000e+00    1.0000e+00    2.0003e+00
Inverse matrix:
    2.1520e+00    1.9501e-01    8.8113e-03   -6.6225e-02   -2.8670e-01
   -1.0454e-01    1.2446e+00    1.6126e-02    1.1675e-01   -7.3960e-02
    1.0562e-01   -9.2360e-02    1.6305e+00    4.7805e-01   -2.2247e-01
    2.6092e-01   -3.1159e-01   -1.4509e-02    1.5749e+00    4.7943e-02
   -9.8252e-01   -6.6636e-02   -2.6583e-01   -7.5522e-02    1.5209e+00
Solution:
   -6.0005e+00    3.0003e+00    2.0003e+00   -3.9998e+00    5.0007e+00
Matrix residual:
    4.4409e-16   -7.8063e-18    3.4694e-18   -8.6736e-19   -2.7756e-17
    1.3878e-17    0.0000e+00   -1.7347e-18    3.4694e-18    0.0000e+00
    0.0000e+00   -1.1276e-17    0.0000e+00   -1.0408e-16    0.0000e+00
   -6.9389e-17    4.9656e-17    8.6736e-19    0.0000e+00    6.9389e-18
   -1.1102e-16   -1.3878e-17   -2.7756e-17    0.0000e+00    0.0000e+00
Solution residual:
    0.0000e+00    0.0000e+00   -4.4409e-16   -8.8818e-16    0.0000e+00
Matrix residual norm:
4.8399e-16
Solution residual norm:
8.8818e-16
Condition number:
3.4843e+00

Sweeping method for tridiagonal matrix:
Given matrix:
    4.9970e-01   -6.5800e-02    0.0000e+00    0.0000e+00    0.0000e+00   -2.8141e+00
    6.8400e-02    7.8240e-01    0.0000e+00    0.0000e+00    0.0000e+00    2.4104e+00
    0.0000e+00    0.0000e+00    6.2860e-01   -1.8410e-01    0.0000e+00    2.2828e+00
    0.0000e+00    0.0000e+00    0.0000e+00    6.1810e-01   -2.6300e-02   -1.6332e+00
    0.0000e+00    0.0000e+00    0.0000e+00    1.3200e-02    7.3640e-01    1.8936e+00
Sufficient condition test: Passed
Solution:
   -5.1664e+00    3.5324e+00    2.8903e+00   -2.5309e+00    2.6168e+00
Matrix residual:
    0.0000e+00   -1.3878e-17    0.0000e+00    0.0000e+00    0.0000e+00
    2.7756e-17    0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
    0.0000e+00    0.0000e+00   -1.1102e-16    0.0000e+00    1.7347e-18
    0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00   -6.9389e-18
    0.0000e+00    0.0000e+00    0.0000e+00   -3.4694e-18    0.0000e+00
Solution residual:
    4.4409e-16    0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
Matrix residual norm:
1.1276e-16
Solution residual norm:
4.4409e-16
Condition number:
1.8248e+00

Square root method for symmetrical matrix:
Given matrix:
    3.7027e-01    7.5622e-03    7.0356e-02   -4.2156e-02    2.9798e-01   -3.9969e-01
```

```
    7.5622e-03    6.4394e-01   -8.6856e-04    5.9534e-02    3.0736e-02    1.8004e+00
    7.0356e-02   -8.6856e-04    4.0933e-01   -1.1382e-01    1.5453e-01    1.6220e+00
   -4.2156e-02    5.9534e-02   -1.1382e-01    4.1957e-01   -2.6247e-02   -1.6055e+00
    2.9798e-01    3.0736e-02    1.5453e-01   -2.6247e-02    5.6529e-01    1.5452e+00
Sufficient condition test: Passed
Solution:
   -6.0005e+00    3.0003e+00    2.0003e+00   -3.9998e+00    5.0007e+00
Matrix residual:
    0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
    0.0000e+00    0.0000e+00   -3.4694e-18   -3.4694e-18    0.0000e+00
   -5.5511e-17    0.0000e+00    2.2204e-16    0.0000e+00   -1.1102e-16
    0.0000e+00    1.9516e-17   -8.6736e-17    0.0000e+00    0.0000e+00
    0.0000e+00    6.9389e-18    0.0000e+00   -2.7756e-17    0.0000e+00
Solution residual:
    5.5511e-16    0.0000e+00    0.0000e+00    2.2204e-16    0.0000e+00
Matrix residual norm:
3.8858e-16
Solution residual norm:
5.5511e-16
Condition number:
8.5835e+00

Jacobi method:
Given matrix:
    4.9970e-01   -6.5800e-02    1.3200e-02    2.6300e-02    9.2100e-02   -2.8141e+00
    6.8400e-02    7.8240e-01    0.0000e+00   -5.2600e-02    5.2600e-02    2.4104e+00
    3.9500e-02    0.0000e+00    6.2860e-01   -1.8410e-01    1.0520e-01    2.2828e+00
   -7.8900e-02    1.6570e-01    0.0000e+00    6.1810e-01   -2.6300e-02   -1.6332e+00
    3.2880e-01    0.0000e+00    1.1840e-01    1.3200e-02    7.3640e-01    1.8936e+00
Sufficient condition test: Passed
Solution:
   -6.0005e+00    3.0003e+00    2.0003e+00   -3.9998e+00    5.0007e+00
Precision:
4.2678e-06 (13 iterations)
Matrix residual:
    4.4409e-16   -7.8063e-18    3.4694e-18   -8.6736e-19   -2.7756e-17
    1.3878e-17    0.0000e+00   -1.7347e-18    3.4694e-18    0.0000e+00
    0.0000e+00   -1.1276e-17    0.0000e+00   -1.0408e-16    0.0000e+00
   -6.9389e-17    4.9656e-17    8.6736e-19    0.0000e+00    6.9389e-18
   -1.1102e-16   -1.3878e-17   -2.7756e-17    0.0000e+00    0.0000e+00
Solution residual:
    2.4181e-07   -1.6753e-07    8.6460e-08    5.4276e-08   -1.8967e-06
Matrix residual norm:
4.8399e-16
Solution residual norm:
1.8967e-06
Condition number:
3.4843e+00

Gauss-seidel method:
Given matrix:
    4.9970e-01   -6.5800e-02    1.3200e-02    2.6300e-02    9.2100e-02   -2.8141e+00
    6.8400e-02    7.8240e-01    0.0000e+00   -5.2600e-02    5.2600e-02    2.4104e+00
    3.9500e-02    0.0000e+00    6.2860e-01   -1.8410e-01    1.0520e-01    2.2828e+00
   -7.8900e-02    1.6570e-01    0.0000e+00    6.1810e-01   -2.6300e-02   -1.6332e+00
    3.2880e-01    0.0000e+00    1.1840e-01    1.3200e-02    7.3640e-01    1.8936e+00
Sufficient condition test: Passed
Solution:
   -6.0005e+00    3.0003e+00    2.0003e+00   -3.9998e+00    5.0007e+00
Precision:
1.4820e-06 (8 iterations)
Matrix residual:
    4.4409e-16   -7.8063e-18    3.4694e-18   -8.6736e-19   -2.7756e-17
    1.3878e-17    0.0000e+00   -1.7347e-18    3.4694e-18    0.0000e+00
    0.0000e+00   -1.1276e-17    0.0000e+00   -1.0408e-16    0.0000e+00
   -6.9389e-17    4.9656e-17    8.6736e-19    0.0000e+00    6.9389e-18
   -1.1102e-16   -1.3878e-17   -2.7756e-17    0.0000e+00    0.0000e+00
```

```
Solution residual:
    8.1454e-08    3.1102e-08    4.8991e-08   -1.9956e-08   -8.8818e-16
Matrix residual norm:
4.8399e-16
Solution residual norm:
8.1454e-08
Condition number:
3.4843e+00

Minimal residual method:
Given matrix:
    3.7027e-01    7.5622e-03    7.0356e-02   -4.2156e-02    2.9798e-01   -3.9969e-01
    7.5622e-03    6.4394e-01   -8.6856e-04    5.9534e-02    3.0736e-02    1.8004e+00
    7.0356e-02   -8.6856e-04    4.0933e-01   -1.1382e-01    1.5453e-01    1.6220e+00
   -4.2156e-02    5.9534e-02   -1.1382e-01    4.1957e-01   -2.6247e-02   -1.6055e+00
    2.9798e-01    3.0736e-02    1.5453e-01   -2.6247e-02    5.6529e-01    1.5452e+00
Sufficient condition test: Passed
Solution:
   -6.0005e+00    3.0003e+00    2.0003e+00   -3.9998e+00    5.0007e+00
Precision:
8.4928e-06 (34 iterations)
Matrix residual:
    0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
    0.0000e+00    0.0000e+00   -3.4694e-18   -3.4694e-18    0.0000e+00
   -5.5511e-17    0.0000e+00    2.2204e-16    0.0000e+00   -1.1102e-16
    0.0000e+00    1.9516e-17   -8.6736e-17    0.0000e+00    0.0000e+00
    0.0000e+00    6.9389e-18    0.0000e+00   -2.7756e-17    0.0000e+00
Solution residual:
    5.1975e-06   -1.8329e-07    5.9736e-07    2.0424e-06   -8.4928e-06
Matrix residual norm:
3.8858e-16
Solution residual norm:
8.4928e-06
Condition number:
8.5835e+00
```

Вывод

1. Метод Гаусса целесообразно применять только для решения систем линейных уравнений небольшого порядка. Причина этого заключается в том, что вычислительная сложность метода Гаусса растет пропорционально n^3, поэтому при относительно небольших n решение системы может занимать большое количество времени. Как видно из полученных результатов, норма вектора невязки имеет порядок 10^{-16}. Учитывая, что программно вещественный тип имеет в своем представлении 17 значащих цифр, вышесказанное позволяет сделать вывод о высокой точности метода Гаусса. Иными словами, точность метода Гаусса определяется точностью представления используемого типа данных.
Стоит отметить, что полученная точность достигнута несмотря на относительно небольшое число обусловленности исходной матрицы.

2. Метод квадратного корня, как и метод Гаусса, целесообразно применять только для решения систем линейных уравнений небольшого порядка. Несмотря на то, что вычислительная сложность метода квадратного корня также пропорциональна n^3, его выполнение требует в 2 раза меньше операций по сравнению с методом Гаусса. Это является преимуществом метода квадратного корня в том случае, если исходная матрица является симметрической. Если исходная матрица не является симметрической, то перед применением метода квадратного корня ее необходимо привести к симметрическому виду домножением слева на A^T. Для этого потребуется совершить n^3 операций, и таким образом метод квадратного корня потеряет свое преимущество перед методом Гаусса.
Как видно из полученных результатов, метод квадратного корня также обеспечивает высокую точность, которая зависит от точности представления используемого типа данных.

3. Метод прогонки применяется для матриц с ленточной структурой. Если исходная матрица является ленточной, то вычислительная сложность решения соответствующей системы будет пропорциональна n. Таким образом метод прогонки можно применять к матрицам произвольной размерности, так как его сложность растет линейно с увеличением размерности матрицы.
Как видно из полученных результатов, несмотря на плохую обусловленность исходной матрицы, решение, полученное с помощью метода прогонки, имеет наибольшую точность по сравнению с методом Гаусса и методом квадратного корня.

4. На практике целесообразно использовать приближенные методы решения систем линейных уравнений, так как часто решение необходимо получить с определенной точностью. Одним из таких методов является метод простых итераций, или метод Якоби. Вычислительная сложность одного шага итерационного процесса Якоби пропорциональна n^2. Таким образом решение с наперед заданной точностью может быть получено за $k n^2$ операций, где k - число итераций, необходимых для достижения заданной точности.
Одним из достаточных условий сходимости метода Якоби является диагональное преобладание матрицы. Если матрица не является диагонально преобладающей, то вопрос сходимости метода необходимо исследовать отдельно.

Для решения исходной системы линейных уравнений с точностью 10^{-6} потребовалось 13 итераций.

5. Метод Гаусса-Зейделя, как и следовало ожидать, обладает лучшей сходимостью по сравнению с методом Якоби. При использовании метода Гаусса-Зейделя для решения исходной системы линейных уравнений с точностью 10^{-6} потребовалось лишь 8 итераций.

6. Для решения линейных систем уравнений можно применять различные методы поиска экстремумов. Одним из примеров таких методов является метод минимальных невязок. Итерационный процесс метода минимальных невязок сводится к минимизации сферической нормы вектора невязок.
Для решения исходной системы линейных уравнений методом минимальных невязок с точностью 10^{-6} потребовалось 34 итерации.

Приложение А. Листинг программы

```
 * /**
 * @author James Akwuh <@jakwuh>
 * @description The purpose of the code is only to get calculation results.
 * Optimization, memory handling, etc. are ignored for simplicity.
 * (However there should be no memory leaks because of using vectors)
 */
#include <iostream>
#include <fstream>
#include <istream>
#include <ostream>
#include <iomanip>
#include <vector>
#include <algorithm>
#include <functional>
#include <initializer_list>

#define DEBUG false
#define EPS 0.00001
#undef minor

using namespace std;

typedef vector<double> Vector;
typedef vector<vector<double>> Matrix;

namespace operators {
	Vector operator+ (Vector a, Vector b) {
		for (int i = 0; i < a.size(); ++i) a[i] += b[i];
		return a;
	}
	Vector operator- (Vector a, Vector b) {
		for (int i = 0; i < a.size(); ++i) a[i] -= b[i];
		return a;
	}
	double operator* (Vector a, Vector b) {
		double x = 0;
		for (int i = 0; i < a.size(); ++i) x += a[i] * b[i];
		return x;
	}

	Matrix operator+ (Matrix a, Matrix b) {
		for (int i = 0; i < a.size(); ++i) a[i] = a[i] + b[i];
		return a;
	}
	Matrix operator- (Matrix a, Matrix b) {
		for (int i = 0; i < a.size(); ++i) a[i] = a[i] - b[i];
		return a;
	}
	Matrix operator*(Matrix a, Matrix b) {
		int n = a.size(), m = b.size(), k = b[0].size();
		if (m != a[0].size()) {throw "Matrix:operator*:dimensions differ";}
		Matrix c(n, Vector(k, 0));
		for (int i = 0; i < n; ++i) {
			for (int j = 0; j < k; ++j) {
				for (int l = 0; l < m; ++l) c[i][j] += a[i][l] * b[l][j];
			}
		}
		return c;
	}
	Matrix operator*(Matrix a, double c) {
		for (int i = 0; i < a.size(); ++i) {
			for (int j = 0; j < a[0].size(); ++j) {
				a[i][j] *= c;
			}
		}
		return a;
	}
}

using namespace operators;
namespace generators {
	Matrix identity_matrix(int n) {
		Matrix a(n, Vector(n, 0));
		for (int i = 0; i < n; ++i) a[i][i] = 1;
		return a;
	}

	Matrix transpose(Matrix a) {
		Matrix b(a[0].size(), Vector(a.size(), 0));
		for (int i = 0; i < a.size(); ++i) {
			for (int j = 0; j < a[0].size(); ++j) b[j][i] = a[i][j];
		}
		return b;
	}

	Matrix transpose(Vector v) {
		Matrix a(1, v);
		return transpose(a);
	}

	Matrix inverse(Matrix a) {
		Matrix b = a;
		for (int i = 0; i < a.size(); ++i) {
			b[i].resize(a.size() * 2);
			b[i][i + a.size()] = 1;
		}
		for (int i = 0; i < b.size(); ++i) {
			for (int k = (i == 0 ? 1 : 0); k < b.size(); k += (k == i - 1 ? 2 : 1)) {
				double c = - b[k][i] / b[i][i];
				for (int j = 0; j < b[0].size(); ++j) b[k][j] += c * b[i][j];
			}
			for (int j = 0; j < b[0].size(); ++j) if (j != i) b[i][j] /= b[i][i];
			b[i][i] = 1;
		}
		for (auto it = b.begin(); it != b.end(); ++it) it->erase(it->begin(), it->begin() + a.size());
		return b;
	}

	Matrix minor(Matrix a, int i, int j) {
		a.erase(a.begin() + i);
```

```
                for (auto it = a.begin(); it != a.end(); ++it) it->erase(it->begin() + j);
                return a;
        }

        Matrix quad(Matrix a) {
                int n = min(a.size(), a[0].size());
                for(int i = n; i < a.size(); ++i) a.erase(a.begin() + i);
                a = transpose(a);
                for(int i = n; i < a.size(); ++i) a.erase(a.begin() + i);
                return transpose(a);
        }
}

using namespace generators;
namespace helpers {
        double sqr(double a){ return a*a;}

        double norm(Matrix a) {
                double x = 0;
                for (int i = 0; i < a.size(); ++i) {
                        double tmp = 0;
                        for (int j = 0; j < a[0].size(); ++j) {
                                tmp += fabs(a[i][j]);
                        }
                        x = max(tmp, x);
                }
                return x;
        }

        double norm(Vector x) {
                double tmp = fabs(x[0]);
                for (int i = 0; i < x.size(); ++i) tmp = max(tmp, fabs(x[i]));
                return tmp;
        }

        double det(Matrix a) {
                if (a.size() == 1) return a[0][0];
                double x = 0;
                for (int j = 0, k = 1; j < a.size(); ++j, k *= -1) {
                        Matrix m = minor(a, 0, j);
                        x += k * a[0][j] * det(m);
                }
                return x;
        }

        bool silvester(Matrix a) {
                if (a.size() != a[0].size()) return false;
                for (int i = 0; i < a.size(); ++i) {
                        for (int j = i + 1; j < a.size(); ++j) {
                                if (a[i][j] != a[j][i]) return false;
                        }
                }
                while (a.size() > 0) {
                        if (det(a) < 0) return false;
                        a.erase(a.begin() + a.size() - 1);
                        for (auto it = a.begin(); it != a.end(); ++it) {it->erase(it->begin() + it->size() - 1);}
                }
                return true;
        }

        Matrix column(Matrix a, int j) {
                return transpose(transpose(a)[j]);
        }

        Matrix swap_rows(Matrix a, int i, int j) {
                swap(a[i], a[j]);
                return a;
        }

        Matrix swap_cols(Matrix a, int i, int j) {
                return transpose(swap_rows(transpose(a), i, j));
        }
}

using namespace helpers;
namespace io {

        void print(Vector v) {
                if (DEBUG)cout << "{";
                bool first = true;
                for (auto it : v) {
                        if (DEBUG)if (!first) cout << ",";
                        cout << (DEBUG ? std::fixed : std::scientific) << setprecision(4) << setw(DEBUG && first ? 13 : 14) << it;
                        first = false;
                }
                cout << (DEBUG ? "}" : "") << endl;
        }

        void print(Matrix a) {
                int c = a.size();
                if (DEBUG) cout << "{";
                for (auto it : a) {
                        print(it);
                        if (DEBUG) cout << (--c > 0 ? "," : "}");
                }
                if (DEBUG) cout << endl;
        }

        void check(Matrix a, Vector x) {
                Matrix aq = quad(a);
                Vector r = transpose(aq * transpose(x) - column(a, a.size()))[0];
                Matrix E = identity_matrix(a.size());
                Matrix R = aq * inverse(aq) - E;

                cout << "Matrix residual:" << endl;
                print(R);
                cout << "Solution residual:" << endl;
                print(r);
                cout << "Matrix residual norm:" << endl;
                cout << norm(R) << endl;
                cout << "Solution residual norm:" << endl;
                cout << norm(r) << endl;
                cout << "Condition number:" << endl;
```

```
                cout << norm(aq) * norm(inverse(aq)) << endl;
        }
}

using namespace io;
namespace methods {
        /**
         * Gauss elimination method with complete pivoting.
         * Solves & prints: given extended matrix, solution, vector residual,
         * vector residual norm, matrix residual, matrix residual norm,
         * inverse matrix, condition number
         * @param a extended matrix
         */
        void gauss_with_pivoting(Matrix a) {
                Matrix ac(a);
                Vector xs(a.size());
                for (int i = 0; i < xs.size(); ++i) xs[i] = i;
                for (int i = 0; i < ac.size(); ++i) {
                        double max = fabs(a[i][i]), maxI = i, maxJ = i;
                        for (int l = i; l < ac.size(); ++l) {
                                for (int m = i; m < ac.size(); ++m) {
                                        if (fabs(ac[l][m]) > max) {
                                                max = fabs(ac[l][m]);
                                                maxI = l;
                                                maxJ = m;
                                        }
                                }
                        }
                        if (maxI != i) ac = swap_rows(ac, i, maxI);
                        if (maxJ != i) ac = swap_cols(ac, i, maxJ);
                        swap(xs[i], xs[maxJ]);

                        for (int k = i + 1; k < ac.size(); ++k) {
                                double c = - ac[k][i] / ac[i][i];
                                for (int j = i; j < ac[0].size(); ++j) {
                                        ac[k][j] += c * ac[i][j];
                                }
                        }
                        for (int j = ac[0].size() - 1; j >= i; --j) {
                                ac[i][j] /= ac[i][i];
                        }
                }
                Vector v(ac.size());
                for (int i = ac.size() - 1; i >= 0; --i) {
                        v[xs[i]] = ac[i][ac[0].size()-1];
                        for (int j = i + 1; j < ac.size(); ++j) {
                                v[xs[i]] -= v[xs[j]] * ac[i][j];
                        }
                }
                // solution residual
                Matrix aq = quad(a);
                Matrix r = transpose(aq * transpose(Matrix(1, v)) - transpose(Matrix(1, transpose(a)[a.size()])));
                // matrix residual
                Matrix E = Matrix(a.size(), Vector(a.size(), 0));
                for (int i = 0; i < a.size(); ++i){ E[i][i] =1; }
                Matrix R = aq * inverse(aq) - E;
                cout << "Gauss elimination method with complete pivoting:" << endl;
                cout << "Given matrix:" << endl;
                print(a);
                cout << "Triangulized matrix:" << endl;
                print(ac);
                cout << "Inverse matrix:" << endl;
                print(inverse(aq));
                cout << "Solution:" << endl;
                print(v);
                check(a, v);
                cout << endl;
        }

        /**
         * Sweeping method for tridiagonal matrix.
         * Solves & prints: given extended matrix, solution, vector residual,
         * vector residual norm, matrix residual, matrix residual norm,
         * condition number.
         * Checks method sufficient conditions.
         * @param a [description]
         */
        void tridiagonal(Matrix matrix) {
                int n = matrix.size();
                for (int i = 0; i < n; ++i) {
                        for (int j = 0; j < n; ++j) {
                                if (abs(i - j) > 1) matrix[i][j] = 0;
                        }
                }
                bool sufficient = true;
                bool sufficient_counter = 0;
                Vector f = transpose(matrix)[matrix.size()];
                Vector l(n), m(n), x(n), a(n), b(n), c(n);
                double sum;
                for (int i = 0; i < n; ++i) {
                        sum = 0;
                        if (i > 0) {a[i] = -matrix[i][i - 1]; sum += fabs(a[i]);}
                        if (i < n - 1) {b[i] = -matrix[i][i + 1]; sum += fabs(a[i]);}
                        c[i] = matrix[i][i];
                        if (fabs(c[i]) > sum) ++sufficient_counter;
                        if (fabs(c[i]) < sum) sufficient = false;
                }
                if (!sufficient_counter) sufficient = false;
                l[n - 1] = a[n - 1] / c[n - 1];
                m[n - 1] = f[n - 1] / c[n - 1];
                for (int i = n - 2; i >= 0; --i) {
                        l[i] = a[i] / (c[i] - l[i + 1] * b[i]);
                        m[i] = (f[i]+m[i + 1] * b[i]) / (c[i] - l[i + 1] * b[i]);
                }
                x[0] = m[0];
                for (int i = 1; i < n; ++i) {x[i] = l[i] * x[i - 1] + m[i];}
                // solution residual
                Matrix aq = matrix;
                for (auto it = aq.begin(); it != aq.end(); ++it){it->erase(it->begin() + it->size() -1);}
                Matrix r = transpose(aq * transpose(Matrix(1, x)) - transpose(Matrix(1, transpose(matrix)[matrix.size()])));
                // matrix residual
                Matrix E = Matrix(a.size(), Vector(a.size(), 0));
                for (int i = 0; i < a.size(); ++i){ E[i][i] =1; }
```

```
        Matrix R = aq * inverse(aq) - E;
        cout << "Sweeping method for tridiagonal matrix:" << endl;
        cout << "Given matrix:" << endl;
        print(matrix);
        cout << "Sufficient condition test:" << (sufficient ? " Passed" : " Failed") << endl;
        cout << "Solution:" << endl;
        print(x);
        check(matrix, x);
        cout << endl;
}

void square_root(Matrix a) {
        Matrix aq = quad(a);
        Matrix bq = transpose(aq) * aq;
        Matrix f = transpose(aq) * column(a, a.size());
        for (int i = 0; i < a.size(); ++i) {
                for (int j = 0; j <= a.size(); ++j) {
                        a[i][j] = (j == a.size() ? f[i][0] : bq[i][j]);
                }
        }
        aq = quad(a);
        f = column(a, a.size());
        int n = a.size();

        Matrix s(n, Vector(n, 0));
        s[0][0] = sqrt(a[0][0]);
        for (int j = 1; j < n; ++j) {s[0][j] = a[0][j] / s[0][0];}
        for (int i = 1; i < n; ++i) {
                s[i][i] = a[i][i];
                for (int k = 0; k < i; ++k) {
                        s[i][i] -= sqr(s[k][i]);
                }
                s[i][i] = sqrt(s[i][i]);

                for (int j = i + 1; j < n; ++j) {
                        s[i][j] = a[i][j];
                        for (int k = 0; k < i; ++k) {
                                s[i][j] -= s[k][i] * s[k][j];
                        }
                        s[i][j] /= s[i][i];
                }
        }
        Vector y(n, 0);
        Vector x(n, 0);
        y[0] = f[0][0] / s[0][0];
        for (int i = 1; i < n; ++i) {
                y[i] = f[i][0];
                for (int j = 0; j < i; ++j) {
                        y[i] -= s[j][i] * y[j];
                }
                y[i] /= s[i][i];
        }
        x[n - 1] = y[n - 1] / s[n - 1][n - 1];
        for (int i = n - 2; i >= 0; --i) {
                x[i] = y[i];
                for (int j = i + 1; j < n; ++j) {
                        x[i] -= s[i][j] * x[j];
                }
                x[i] /= s[i][i];
        }

        Matrix E = Matrix(a.size(), Vector(n, 0));
        for (int i = 0; i < n; ++i){ E[i][i] =1; }
        Matrix R = aq * inverse(aq) - E;
        cout << "Square root method for symmetrical matrix:" << endl;
        cout << "Given matrix:" << endl;
        print(a);
        cout << "Sufficient condition test:" << (silvester(aq) ? " Passed" : " Failed") << endl;
        cout << "Solution:" << endl;
        print(x);
        check(a, x);
        cout << endl;
}

void jacobi(Matrix a, double precision) {
        int n = a.size();
        bool criteria = true;
        for (int i = 0; i < n; ++i) {
                double sum = -fabs(a[i][i]);
                for (int j = 0; j < n; ++j) {
                        sum += fabs(a[i][j]);
                }
                if (a[i][i] <= sum) criteria = false;
        }
        Matrix X(1, Vector(n, 0));
        double eps = precision + 1;
        int k = 0;
        while (eps > precision) {
                X.resize(k + 2, Vector(n, 0));
                for (int i = 0; i < n; ++i) {
                        X[k + 1][i] = a[i][n] / a[i][i];
                        for (int j = 0; j < n; ++j) {
                                if (j == i) continue;
                                X[k + 1][i] -= a[i][j] / a[i][i] * X[k][j];
                        }
                }
                eps = norm(X[k + 1] - X[k]);
                ++k;
        }
        Vector x = X[k];
        Matrix aq = a;
        for (auto it = aq.begin(); it != aq.end(); ++it){it->erase(it->begin() + it->size() -1);}
        Matrix r = transpose(aq * transpose(Matrix(1, x)) - transpose(Matrix(1, transpose(a)[a.size()])));
        // matrix residual
        Matrix E = Matrix(a.size(), Vector(a.size(), 0));
        for (int i = 0; i < a.size(); ++i){ E[i][i] =1; }
        Matrix R = aq * inverse(aq) - E;
        cout << "Jacobi method:" << endl;
        cout << "Given matrix:" << endl;
        print(a);
        cout << "Sufficient condition test:" << (criteria ? " Passed" : " Failed") << endl;
        cout << "Solution:" << endl;
        print(x);
```

```
                cout << "Precision:" << endl;
                cout << eps << " (" << k << " iterations)" << endl;
                check(a, x);
                cout << endl;
        }

        void gauss_seidel(Matrix a, double precision) {
                int n = a.size();
                Matrix X(1, Vector(n, 0));
                double eps = precision + 1;
                int k = 0;
                bool criteria = true;
                for (int i = 0; i < n; ++i) {
                        double sum = -fabs(a[i][i]);
                        for (int j = 0; j < n; ++j) {
                                sum += fabs(a[i][j]);
                        }
                        if (a[i][i] <= sum) criteria = false;
                }
                while (eps > precision) {
                        X.resize(k + 2, Vector(n, 0));
                        for (int i = 0; i < n; ++i) {
                                X[k + 1][i] = a[i][n] / a[i][i];
                                for (int j = 0; j < n; ++j) {
                                        if (j == i) continue;
                                        if (j < i) {
                                                X[k + 1][i] -= a[i][j] / a[i][i] * X[k + 1][j];
                                        } else {
                                                X[k + 1][i] -= a[i][j] / a[i][i] * X[k][j];
                                        }
                                }
                        }
                        eps = norm(X[k + 1] - X[k]);
                        ++k;
                }
                Vector x = X[k];
                Matrix aq = a;
                for (auto it = aq.begin(); it != aq.end(); ++it){it->erase(it->begin() + it->size() -1);}
                Matrix r = transpose(aq * transpose(Matrix(1, x)) - transpose(Matrix(1, transpose(a)[a.size()])));
                // matrix residual
                Matrix E = Matrix(a.size(), Vector(a.size(), 0));
                for (int i = 0; i < a.size(); ++i){ E[i][i] =1; }
                Matrix R = aq * inverse(aq) - E;
                cout << "Gauss-seidel method:" << endl;
                cout << "Given matrix:" << endl;
                print(a);
                cout << "Sufficient condition test:" << (criteria ? " Passed" : " Failed") << endl;
                cout << "Solution:" << endl;
                print(x);
                cout << "Precision:" << endl;
                cout << eps << " (" << k << " iterations)" << endl;
                check(a, x);
                cout << endl;
        }

        void minimal_residual_method(Matrix a, double precision) {
                Matrix aq = quad(a);
                Matrix bq = transpose(aq) * aq;
                Matrix f = transpose(aq) * column(a, a.size());
                for (int i = 0; i < a.size(); ++i) {
                        for (int j = 0; j <= a.size(); ++j) {
                                a[i][j] = (j == a.size() ? f[i][0] : bq[i][j]);
                        }
                }
                aq = quad(a);
                f = column(a, a.size());

                Matrix x = f;
                Matrix r = aq * x - f;

                int k = 0;
                while (norm(transpose(r)[0]) > precision) {
                        ++k;
                        x = x - r * ((transpose(r) * (aq * r))[0][0] / (transpose(aq * r) * (aq * r))[0][0]);
                        r = aq * x - f;
                }
                Vector xs = transpose(x)[0];
                cout << "Minimal residual method:" << endl;
                cout << "Given matrix:" << endl;
                print(a);
                cout << "Sufficient condition test:" << (silvester(bq) ? " Passed" : " Failed") << endl;
                cout << "Solution:" << endl;
                print(xs);
                cout << "Precision:" << endl;
                cout << norm(transpose(r)[0]) << " (" << k << " iterations)" << endl;
                check(a, xs);
                cout << endl;
        }
}

using namespace methods;
int main(int argc, char *argv[]) {
        try     {
                if (argc == 2) {freopen (argv[1], "w", stdout);}
                Matrix a = {
                        {0.4997, -0.0658, 0.0132, 0.0263, 0.0921, -2.8141},
                        {0.0684, 0.7824, 0.0000, -0.0526, 0.0526, 2.4104},
                        {0.0395, 0.0000, 0.6286, -0.1841, 0.1052, 2.2828},
                        {-0.0789, 0.1657, 0.0000, 0.6181, -0.0263, -1.6332},
                        {0.3288, 0.0000, 0.1184, 0.0132, 0.7364, 1.8936}
                };
                gauss_with_pivoting(a);
                tridiagonal(a);
                square_root(a);
                jacobi(a, EPS);
                gauss_seidel(a, EPS);
                minimal_residual_method(a, EPS);
                if (argc == 2) {fclose(stdout);}
        } catch (const char* error) {
                cout << error;
        }
}
```