

Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной
математики

Кафедра вычислительной математики и программирования

Лабораторная работа №1 по курсу «Численные методы»

Студент: Ветренко П. С.
Преподаватель: Иванов И. Э.
Группа: М8О-306Б
Вариант: 2
Дата:
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №1.1

Задача: 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.

$$\begin{cases} 2 \cdot x_1 + 7 \cdot x_2 - 8 \cdot x_3 + 6 \cdot x_4 = -39 \\ 4 \cdot x_1 + 4 \cdot x_2 - 7 \cdot x_4 = 41 \\ -x_1 - 3 \cdot x_2 + 6 \cdot x_3 + 3 \cdot x_4 = 4 \\ 9 \cdot x_1 - 7 \cdot x_2 - 2 \cdot x_3 - 8 \cdot x_4 = 113 \end{cases}$$

1 Описание метода решения

LU – разложение матрицы - представление матрицы A в виде произведения двух матриц,

$$A = L \cdot U,$$

где L - нижняя треугольная матрица, U - верхняя треугольная матрица.

LU – разложение может быть построено с использованием метода Гаусса. Рассмотрим k -й шаг метода Гаусса, на котором осуществляется обнуление поддиагональных элементов k -го столбца матрицы $A^{(k-1)}$. С этой целью используется следующая операция:

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

В терминах матричных операций такая операция эквивалентна умножению $A^{(k)} = M_k A^{(k-1)}$, где элементы матрицы M_k определяются следующим образом

$$m_{ij}^{(k)} = \begin{cases} 1, i = j \\ 0, i \neq j, j \neq k \\ -\mu_{k+1}^{(k)}, i \neq j, j = k \end{cases}.$$

Т.е. матрица M_k имеет вид

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -\mu_{k+1}^{(k)} & 1 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & -\mu_n^{(k)} & 0 & 0 & 1 \end{pmatrix}$$

При этом выражение для обратной операции запишется в виде $A^{(k-1)} = M_k^{-1} A^{(k)}$, где

$$M_k^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \mu_{k+1}^{(k)} & 1 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \mu_n^{(k)} & 0 & 0 & 1 \end{pmatrix}$$

В результате прямого хода метода Гаусса получим $A^{(n-1)} = U$,

$$A = A^{(0)} = M_1^{-1}A^{(1)} = M_1^{-1}M_2^{-1}A^{(2)} = M_1^{-1}M_2^{-1} \dots M_{n-1}^{-1}A^{(n-1)},$$

где $A^{(n-1)} = U$ - верхняя треугольная матрица, а $L = M_1^{-1}M_2^{-1} \dots M_{n-1}^{-1}$ - нижняя

треугольная матрица, имеющая вид $L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ \mu_2^{(1)} & 1 & 0 & 0 & 0 & 0 \\ \mu_3^{(1)} & \mu_3^{(2)} & 1 & 0 & 0 & 0 \\ \dots & \dots & \mu_{k+1}^{(k)} & 1 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \mu_n^{(1)} & \mu_n^{(2)} & \mu_n^{(k)} & \mu_n^{(k+1)} & \dots & \mu_n^{(n-1)} & 1 \end{pmatrix}$

Таким образом, искомое разложение $A = LU$ получено.

В дальнейшем LU-разложение может быть эффективно использовано при решении систем линейных алгебраических уравнений вида $A \cdot x = b$. Действительно, подставляя LU-разложение в СЛАУ, получим $L \cdot U \cdot x = b$, или $U \cdot x = L^{-1} \cdot b$. Т.е. процесс решения СЛАУ сводится к двум простым этапам.

На первом этапе решается СЛАУ $L \cdot z = b$. Поскольку матрица системы — нижняя треугольная, решение можно записать в явном виде:

$$z_1 = b_1, z_i = b_i - \sum_{j=1}^{i-1} l_{ij} \cdot z_j, i = \overline{2, n}$$

На втором этапе решается СЛАУ $U \cdot x = z$ с верхней треугольной матрицей. Здесь, как и на предыдущем этапе, решение представляется в явном виде:

$$x_n = \frac{z_n}{u_{nn}}, x_i = \frac{1}{u_{ii}} \cdot (z_i - \sum_{j=i+1}^n u_{ij} \cdot x_j, i = \overline{n-1, 1}.$$

Отметим, что второй этап эквивалентен обратному ходу методу Гаусса, тогда как первый соответствует преобразованию правой части СЛАУ в процессе прямого хода.

2 Входные и выходные данные

Входные данные:

```
[psvetrenko@MacBook-Pro-Polina NM1 % cat input_lab1_1.txt
2 7 -8 6
4 4 0 -7
-1 -3 6 3
9 -7 -2 -8
-39 41 4 113%
psvetrenko@MacBook-Pro-Polina NM1 %
```

Исходные данные для решения СЛАУ хранятся в файле `input_lab1_1.txt`, программа считывает данные построчно.

Выходные данные:

Лабораторная работа 1
Вариант 2

Выберите метод:

- 1 – Алгоритм LU-разложения матрицы
 - 2 – Метод прогонки
 - 3 – Метод простых итераций и метод Зейделя
 - 4 – Метод вращений
 - 5 – QR-алгоритм
 - 0 – выход
- 1

Алгоритм LU-разложения матрицы

A= [[2. 7. -8. 6.]
[4. 4. 0. -7.]
[-1. -3. 6. 3.]
[9. -7. -2. -8.]]
b= [-39.0, 41.0, 4.0, 113.0]

Решение СЛАУ:

x= [8. -3. 2. -3.]

Определитель матрицы:

det A= -4923.999999999998

Обратная матрица:

A*= [[0.10235581 0.07189277 0.16125102 0.07432981]
[0.03980504 0.11129163 0.03493095 -0.05442729]
[-0.00365556 0.08671812 0.15495532 -0.02051178]
[0.08123477 -0.03818034 0.11210398 0.01137287]]

U= [[2. 7. -8. 6.]
[0. -10. 16. -19.]
[0. 0. 2.8 5.05]
[0. 0. 0. 87.92857143]]

L= [[1. 0. 0. 0.]
[2. 1. 0. 0.]
[-0.5 -0.05 1. 0.]
[4.5 3.85 -9.85714286 1.]]

Проверка:

L*U= [[2. 7. -8. 6.]
[4. 4. 0. -7.]
[-1. -3. 6. 3.]
[9. -7. -2. -8.]]

3 Выводы

При выполнении лабораторной работы я изучила метод решения СЛАУ с помощью LU-разложения. LU – разложение удобно использовать при решении систем линейных алгебраических уравнений вида $A \cdot x = b$, процесс решения СЛАУ сводится к двум простым этапам. На первом этапе решается СЛАУ $L \cdot z = b$. На втором этапе решается СЛАУ $U \cdot x = z$ с верхней треугольной матрицей.

Лабораторная работа №1.2

Задача: 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.

$$\begin{cases} 10 \cdot x_1 - 5 \cdot x_2 = -120 \\ 3 \cdot x_1 + 10 \cdot x_2 - 2 \cdot x_3 = -91 \\ 2 \cdot x_2 - 9 \cdot x_3 - 5 \cdot x_4 = 5 \\ 5 \cdot x_3 + 16 \cdot x_4 - 4 \cdot x_5 = -74 \\ -8 \cdot x_4 + 16 \cdot x_5 = -56 \end{cases}$$

1 Описание метода решения

Метод прогонки является частным случаем метода Гаусса и используется для решения систем линейных уравнений вида $Ax = b$, где A - трёхдиагональная матрица.

Трёхдиагональной матрицей называется матрица такого вида, где во всех остальных местах, кроме главной диагонали и двух соседних с ней, стоят нули. Метод прогонки состоит из двух этапов: прямой прогонки и обратной прогонки. На первом этапе определяются прогоночные коэффициенты, а на втором – находят неизвестные x .

СЛАУ имеет вид:

$$\begin{cases} b_1 \cdot x_1 + c_1 \cdot x_2 = d_1, a_1 = 0 \\ a_2 \cdot x_1 + b_2 \cdot x_2 + c_2 \cdot x_3 = d_2 \\ a_3 \cdot x_2 + b_3 \cdot x_3 + c_3 \cdot x_4 = d_3 \\ \dots\dots\dots \\ a_{n-1} \cdot x_{n-2} + b_{n-1} \cdot x_{n-1} + c_{n-1} \cdot x_n = d_{n-1} \\ a_n \cdot x_{n-1} + b_n \cdot x_n = d_n, c_n = 0 \end{cases}$$

Решение СЛАУ будем искать в виде:

$$x_i = P_i \cdot x_{i+1} + Q_i, \overline{1, n},$$

где P_i, Q_i - прогоночные коэффициенты. Прямой ход метода прогонки состоит в вычислении этих коэффициентов P_i, Q_i , при $i = \overline{1, n}$

Прогоночные коэффициенты вычисляются по следующим формулам:

$$P_i = \frac{-c_i}{b_i + a_i \cdot P_{i-1}}, Q_i = \frac{d_i - a_i \cdot Q_{i-1}}{b_i + a_i \cdot P_{i-1}}, i = \overline{2, n-1};$$

$$P_1 = \frac{-c_1}{b_1}, Q_1 = \frac{d_1}{b_1};$$

$$P_n = 0, Q_n = \frac{d_n - a_n \cdot Q_{n-1}}{b_n + a_n \cdot P_{n-1}}, i = n.$$

Таким образом, прямой ход метода прогонки по определению прогоночных коэффициентов $P_i, Q_i, i = \overline{1, n}$ завершен.

Обратный ход метода прогонки заключается в поиске самих значений x_i , согласно уже полученной формуле $x_i = P_i \cdot x_{i+1} + Q_i, \overline{1, n}$. Значения x_i будем искать в обратном порядке:

[illegible]

Общее число операций в методе прогонки равно $8 \cdot n + 1$, т.е. пропорционально числу уравнений. Такие методы решения СЛАУ называют экономичными. В то же время, число операций в методе Гаусса пропорционально n^3 . Условия устойчивости метода прогонки:

$$a_i \neq 0, c_i \neq 0, i = \overline{2, n-1},$$

$$|b_1| \geq |a_i| + |c_i|, i = \overline{1, n},$$

причем строгое неравенство имеет место хотя бы при одном i .

2 Входные и выходные данные

Входные данные:

```
psvetrenko@MacBook-Pro-Polina NM1 % cat input_lab1_2.txt
0 3 2 5 -8
10 10 -9 16 16
5 -2 -5 -4 0
-120 -91 5 -74 -56%
psvetrenko@MacBook-Pro-Polina NM1 % █
```

Исходные данные для решения СЛАУ хранятся в файле input_lab1_2.txt, программа считывает данные по столбцам.

Выходные данные:

```
Выберите метод:
1 - Алгоритм LU-разложения матрицы
2 - Метод прогонки
3 - Метод простых итераций и метод Зейделя
4 - Метод вращений
5 - QR-алгоритм
0 - выход
2
```

```
Метод прогонки
p= [-0.5      0.23529412 -0.5862069  0.3060686  0.      ]
q= [-12.      -6.47058824 -2.10344828 -4.85751979 -7.      ]
```

```
Решение СЛАУ: x= [-9. -6.  2. -7. -7.]
```

3 Выводы

При выполнении второго задания я познакомилась с еще одним методом решения СЛАУ - методом прогонки. Как и в методе Гаусса здесь есть прямой и обратный ход. Во время прямого хода мы ищем необходимые коэффициенты, с помощью которых во время обратного хода мы найдём корни уравнений. Этот метод мне показался более простым как в понимании, так и в реализации.

Лабораторная работа №1.3

Задача: 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.

$$\begin{cases} 24 \cdot x_1 + 2 \cdot x_2 + 4 \cdot x_3 - 9 \cdot x_4 = -9 \\ -6 \cdot x_1 - 27 \cdot x_2 - 8 \cdot x_3 - 6 \cdot x_4 = -76 \\ -4 \cdot x_1 + 8 \cdot x_2 + 19 \cdot x_3 + 6 \cdot x_4 = -79 \\ 4 \cdot x_1 + 5 \cdot x_2 - 3 \cdot x_3 - 13 \cdot x_4 = -70 \end{cases}$$

частей $x^{(0)} = \beta$ или $x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)}$.

Тогда метод простых итераций имеет вид:

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

Достаточное условие сходимости метода простых итераций:

Метод простых итераций сходится к единственному решению эквивалентной СЛАУ, а следовательно, и к решению исходной СЛАУ при любом начальном приближении $x^{(0)}$, если какая-либо норма матрицы α эквивалентной системы меньше единицы $\|\alpha\| < 1$.

Если используется метод Якоби для эквивалентной СЛАУ, то достаточным условием сходимости является диагональное преобладание матрицы A , т.е.

$$|\alpha_{ii}| > \sum_{j=1, i \neq j}^n |a_{ij}|, \forall i$$

для каждой строки матрицы A модули элементов, стоящих на главной диагонали, больше суммы модулей недиагональных элементов.

При выполнении достаточного условия сходимости оценка погрешности решения на k -й итерации дается выражением:

$$\|x^{(k)} - x^*\| \leq \epsilon^{(k)} = \frac{\|\alpha\|}{1 - \|\alpha\|} \cdot \|x^{(k)} - x^{(k-1)}\|,$$

где x^* - точное решение СЛАУ.

Процесс итераций останавливается при выполнении условия $\epsilon^{(k)} \leq \epsilon$, где ϵ — задаваемая точность.

Метод Зейделя

Метод простых итераций довольно медленно сходится. Для его ускорения существует метод Зейделя, заключающийся в том, что при вычислении компонента x_i^{k+1} вектора неизвестных на $(k+1)$ -й итерации используются $x_1^{k+1}, x_2^{k+1}, \dots, x_{i-1}^{k+1}$, уже вычисленные на $(k+1)$ -й итерации.

Значения остальных компонент $x_{i+1}^{k+1}, x_{i+2}^{k+1}, \dots, x_n^{k+1}$ берутся из предыдущей итерации. Так же, как и в методе простых итераций, строится эквивалентная СЛАУ и за начальное приближение принимается вектор правых частей $x^{(0)} = (\beta_1 \beta_2 \dots \beta_n)$.

Тогда метод Зейделя для известного вектора $(x_1^k, x_2^k, \dots, x_n^k)^T$ на k -й итерации имеет вид:

[illegible]

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

Следовательно,

$$(E - B) \cdot x^k + 1 = C \cdot x^k + \beta,$$

откуда

$$x^{k+1} = (E - B)^{-1} \cdot C \cdot x^k + (E - B)^{-1} \cdot \beta.$$

Таким образом, метод Зейделя является методом простых итераций с матрицей правых частей $\alpha = (E - B)^{-1} \cdot C$ и вектором правых частей $(E - B)^{-1} \cdot \beta$ и, следовательно, сходимость и погрешность метода Зейделя можно исследовать с помощью формул, выведенных для метода простых итераций, в которых вместо матрицы α подставлена матрица $(E - B)^{-1} \cdot C$, а вместо вектора правых частей — вектор $(E - B)^{-1} \cdot \beta$.

Для практических вычислений важно, что в качестве достаточных условий сходимости метода Зейделя могут быть использованы условия, приведенные для метода простых итераций ($\|\alpha\| < 1$ или диагональное преобладание матрицы A).

В случае выполнения этих условий для оценки погрешности на k -й итерации можно использовать выражение:

$$\epsilon^k = \frac{\|C'\|}{1 - \|\alpha\|} \cdot \|x^{(k)} - x^{(k-1)}\|.$$

2 Входные и выходные данные

Входные данные:

```
psvetrenko@MacBook-Pro-Polina NM1 % cat input_lab1_3.txt
24. 2. 4. -9.
-6. -27. -8. -6.
-4. 8. 19. 6.
4. 5. -3. -13.
-9. -76. -79. -70.
psvetrenko@MacBook-Pro-Polina NM1 %
```

Исходные данные для решения СЛАУ хранятся в файле `input_lab1_3.txt`, программа считывает данные построчно.

Выходные данные:

```

Выберите метод:
1 – Алгоритм LU-разложения матрицы
2 – Метод прогонки
3 – Метод простых итераций и метод Зейделя
4 – Метод вращений
5 – QR-алгоритм
0 – выход
3

A= [[ 24.  2.  4. -9.]
     [-6. -27. -8. -6.]
     [-4.  8. 19.  6.]
     [ 4.  5. -3. -13.]]

b= [-9.0, -76.0, -79.0, -70.0]

Введите точность вычислений:
0.1
eps= 0.1

Метод простых итераций

alfa= [[ 0.          -0.08333333 -0.16666667  0.375      ]
        [-0.22222222  0.          -0.2962963  -0.22222222]
        [ 0.21052632 -0.42105263  0.          -0.31578947]
        [ 0.30769231  0.38461538 -0.23076923  0.          ]]

beta= [-0.375      2.81481481 -4.15789474  5.38461538]

Условие выполнено:
||alfa||= 0.9473684210526315 <1

10 итераций
[ 3.99766442  1.99940062 -6.99780807  8.99709506]

Метод Зейделя

alfa= [[ 0.          -0.08333333 -0.16666667  0.375      ]
        [-0.22222222  0.          -0.2962963  -0.22222222]
        [ 0.21052632 -0.42105263  0.          -0.31578947]
        [ 0.30769231  0.38461538 -0.23076923  0.          ]]

beta= [-0.375      2.81481481 -4.15789474  5.38461538]

Условие выполнено:
||alfa||= 0.9473684210526315 <1

6 итераций
[ 3.99963344  2.00004142 -6.99989529  8.99987898]

```

3 Выводы

Выполнив третье задание, я убедилась, что метод Зейделя работает быстрее, чем метод простых итераций. Также метод Зейделя дает немного большую точность за меньшее число шагов, чем метод простых итераций. И хоть метод простых итераций

получается более экономичным, так как мы производим только произведение матрицы на вектор, но метод Зейделя работает быстрее за счёт того, что на каждой итерации учитывает уже вычисленные данные.

Лабораторная работа №1.4

Задача: 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.

$$\begin{pmatrix} -9 & 7 & 5 \\ 7 & 8 & 9 \\ 5 & 9 & 8 \end{pmatrix}$$

1 Описание метода решения

Метод вращений Якоби численного решения задач на собственные значения и собственные векторы матриц

Метод вращений Якоби применим только для симметрических матриц $A_{n \times n}$ ($A = A^T$) и решает полную проблему собственных значений и собственных векторов таких матриц.

Он основан на отыскании с помощью итерационных процедур матрицы U в преобразовании подобия $\Lambda = U^{-1} \cdot A \cdot U$, а поскольку для симметрических матриц A матрица преобразования подобия U является ортогональной $U^{-1} = U^T$, то $\Lambda = U^T \cdot A \cdot U$, где Λ - диагональная матрица с собственными значениями на главной диагонали.

$$\Lambda = \begin{pmatrix} \lambda_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \lambda_n \end{pmatrix};$$

U - матрица преобразования, столбцы которой являются собственными векторами матрицы A , соответствующие ее собственным значениям.

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

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

1) Выбирается максимальный по модулю недиагональный элемент $a_{ij}^{(k)}$ матрицы $A^{(k)}$ $|a_{ij}^{(k)}| = \max_{l < m} |a_{lm}^{(k)}|$.

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

В матрице вращения на пересечении i -й строки и j -го столбца находится элемент $u_{ij}^{(k)} = -\sin \varphi^{(k)}$, где $\varphi^{(k)}$ - угол вращения, подлежащий определению. Симметрично относительно главной диагонали (j -я строка, i -й столбец) расположен элемент $u_{ji}^{(k)} = \sin \varphi^{(k)}$.

Диагональные элементы $u_{ii}^{(k)}, u_{jj}^{(k)}$ равны соответственно $u_{ii}^{(k)} = u_{jj}^{(k)} \cos \varphi^{(k)}$; другие диагональные элементы $u_{mm}^{(k)} = 1, m = \overline{1, n}, m \neq i \neq j$; остальные элементы в матрице

вращения $U^{(k)}$ равны нулю.

Угол вращения $\varphi^{(k)}$ определяется из условия $a_{ij}^{k+1} = 0$:

$$\varphi^{(k)} = \frac{1}{2} \cdot \arctan \frac{2 \cdot a_{ij}^{(k)}}{a_{ii}^{(k)} - a_{jj}^{(k)}},$$

причём если $a_{ii}^{(k)} = a_{jj}^{(k)}$, то $\varphi^{(k)} = \frac{\pi}{4}$.

3) Строится матрица $A^{(k+1)}$:

$$A^{(k+1)} = U^{(k)T} \cdot A^{(k)} \cdot U^{(k)},$$

в которой элемент $a_{ij}^{(k+1)} \approx 0$.

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

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

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

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

продолжается.

Если $t(A^{(k+1)}) < \epsilon$, то итерационный процесс останавливается, и в качестве искоемых собственных значений принимаются $\lambda_1 \approx a_{11}^{k+1}$, $\lambda_2 \approx a_{22}^{k+1}$, \dots , $\lambda_n \approx a_{nn}^{k+1}$.

Координатными столбцами собственных векторов матрицы A в единичном базисе будут столбцы матрицы $U = U^{(0)} \cdot U^{(1)} \dots U^{(k)}$, т.е.

$$(x^1)^T = (u_{11}, u_{21}, \dots, u_{n1}), (x^2)^T = (u_{12}, u_{22}, \dots, u_{n2}), \dots, (x^n)^T = (u_{1n}, u_{2n}, \dots, u_{nn}),$$

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

2 Входные и выходные данные

Входные данные:

```
psvetrenko@MacBook-Pro-Polina NM1 % cat input_lab1_4.txt
-9. 7. 5.
7. 8. 9.
5. 9. 8.
psvetrenko@MacBook-Pro-Polina NM1 %
```

Исходные данные хранятся в файле input_lab1_4.txt, программа считывает данные построчно.

Выходные данные:

```
Выберите метод:
1 - Алгоритм LU-разложения матрицы
2 - Метод прогонки
3 - Метод простых итераций и метод Зейделя
4 - Метод вращений
5 - QR-алгоритм
0 - выход
4
```

Метод вращений

```
A= [[-9. 7. 5.]
     [ 7. 8. 9.]
     [ 5. 9. 8.]]
```

```
Введите точность вычислений:
0.1
eps= 0.1
4 итераций
```

```
Собственные значения:
-11.695959528598651
19.53203443529618
-0.8360749066975255
```

```
Собственные векторы:
[ 0.95088381 -0.28891762 -0.11111519]
[0.28744058 0.69090574 0.66334544]
[-0.11488207 -0.66270346 0.74001773]
```

3 Выводы

В результате выполнения четвёртого задания я узнала, как можно находить собственные значения и собственные вектора симметрических матриц с помощью метода вращений Якоби. Довольно интересно, что метод Якоби производит вращения до тех пор, пока матрица не станет почти диагональной. Тогда элементы на диагонали аппроксимируют собственные значения матрицы A .

Лабораторная работа №1.5

Задача: 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.

$$\begin{pmatrix} -6 & 4 & 0 \\ -7 & 6 & -7 \\ -2 & -6 & -7 \end{pmatrix}$$

1 Описание метода решения

QR-алгоритм

При решении полной проблемы собственных значений для несимметричных матриц эффективным является подход, основанный на приведении матриц к подобным, имеющим треугольный или квазитреугольный вид. Одним из наиболее распространенных методов этого класса является QR-алгоритм, позволяющий находить как вещественные, так и комплексные собственные значения.

В основе QR-алгоритма лежит представление матрицы в виде $A = Q \cdot R$, где Q - ортогональная матрица ($Q^{-1} = Q^T$), а R - верхняя треугольная матрица. Такое разложение существует для любой квадратной матрицы. Одним из возможных подходов к построению QR-разложения является использование преобразования Хаусхолдера, позволяющего обратить в нуль группу поддиагональных элементов столбца матрицы.

Преобразование Хаусхолдера осуществляется с использованием матрицы Хаусхолдера, имеющей следующий вид:

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

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

Рассмотрим случай, когда необходимо обратить в нуль все элементы какого-либо вектора кроме первого, т.е. построить матрицу Хаусхолдера такую, что

$$\tilde{b} = H \cdot b, b = (b_1, b_2, \dots, b_n)^T, \tilde{b} = (\tilde{b}_1, 0, \dots, 0)^T.$$

Тогда вектор ν определится следующим образом:

$$\nu = b + \text{sign}(b_1) \cdot \|b_2\|_2 \cdot e_1,$$

где $\|b_2\|_2 = (\sum_i b_i^2)^{\frac{1}{2}}$ - евклидова норма вектора, $e_1 = (1, 0, \dots, 0)^T$.

Применяя описанную процедуру с целью обнуления поддиагональных элементов каждого из столбцов исходной матрицы, можно за фиксированное число шагов получить ее QR-разложение.

Рассмотрим подробнее реализацию данного процесса. Положим $A_0 = A$ и построим преобразование Хаусхолдера $H_1(A_1 = H_1 \cdot A_0)$, переводящее матрицу A_0 в матрицу A_1 с нулевыми элементами первого столбца под главной диагональю:

$$A_0 = \begin{pmatrix} a_{11}^0 & a_{12}^0 & \dots & a_{1n}^0 \\ a_{21}^0 & a_{22}^0 & \dots & a_{2n}^0 \\ \dots & \dots & \dots & \dots \\ a_{n1}^0 & a_{n2}^0 & \dots & a_{nn}^0 \end{pmatrix} \xrightarrow{H_1} A_1 = \begin{pmatrix} a_{11}^1 & a_{12}^1 & \dots & a_{1n}^1 \\ 0 & a_{22}^1 & \dots & a_{2n}^1 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & a_{nn}^1 \end{pmatrix}$$

Ясно, что матрица Хаусхолдера H_1 должна определяться по первому столбцу матрицы A_0 , т.е. в качестве вектора b в формуле ν берется вектор $(a_{11}^0, a_{21}^0, \dots, a_{n1}^0)^T$. Тогда компоненты вектора ν вычисляются следующим образом:

$$\nu_1^1 = a_{11}^0 + \text{sign}(a_{11}^0) \cdot \left(\sum_{j=1}^n (a_{j1}^0)^2 \right)^{\frac{1}{2}}$$

$$\nu_i^1 = a_{i1}^0, i = \overline{2, n}.$$

Матрица Хаусхолдера H_1 вычисляется согласно формуле:

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

На следующем, втором, шаге рассматриваемого процесса строится преобразование Хаусхолдера $H_2(A_2 = H_2 \cdot A_1)$, обнуляющее расположенные ниже главной диагонали элементы второго столбца матрицы A_1 . Взяв в качестве вектора b вектор $(a_{22}^1, a_{32}^1, \dots, a_{n2}^1)^T$ размерности $n - 1$, получим следующие выражения для компонентов вектора ν :

$$\nu_2^1 = 0,$$

$$\nu_2^2 = a_{22}^1 + \text{sign}(a_{22}^1) \cdot \left(\sum_{j=2}^n (a_{j2}^1)^2 \right)^{\frac{1}{2}}$$

$$\nu_i^2 = a_{i2}^1, i = \overline{3, n}.$$

Повторяя процесс $n - 1$ раз, получим искомое разложение $A = Q \cdot R$, где $Q = (H_{n-1} \cdot H_{n-2} \cdot \dots \cdot H_0)^T = H_1 \cdot H_2 \cdot \dots \cdot H_{n-1}$, $R = A_{n-1}$.

Процедура QR-разложения многократно используется в QR-алгоритме вычисления собственных значений.

Строится следующий итерационный процесс.

$$A^{(0)} = A,$$

$$A^{(0)} = Q^{(0)} \cdot R^{(0)} - \text{QR-разложение},$$

$$A^{(1)} = R^{(0)} \cdot Q^{(0)} - \text{перемножение матриц},$$

.....

$$A^{(k)} = Q^{(k)} \cdot R^{(k)}$$

$$A^{(k+1)} = R^{(k)} \cdot Q^{(k)}$$

Таким образом, каждая итерация реализуется в два этапа. На первом этапе осуществляется разложение матрицы $A^{(k)}$ в произведение ортогональной и верхней треугольной матриц, а на втором - полученные матрицы перемножаются в обратном порядке.

Каждому вещественному собственному значению будет соответствовать столбец со стремящимися к нулю поддиагональными элементами и в качестве критерия сходимости итерационного процесса для таких собственных значений можно использовать следующее неравенство:

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

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

Каждой комплексно-сопряженной паре соответствует диагональный блок размерностью 2×2 , т.е. матрица $A^{(k)}$ имеет блочно-диагональную структуру. В качестве критерия окончания итераций для таких блоков может быть использовано следующее условие

$$|\lambda^{(k)} - \lambda^{(k-1)}| \leq \epsilon.$$

2 Входные и выходные данные

Входные данные:

```
psvetrenko@MacBook-Pro-Polina NM1 % cat input_lab1_5.txt
-6 -4 0
-7 6 -7
-2 -6 -7
psvetrenko@MacBook-Pro-Polina NM1 %
```

Исходные данные хранятся в файле input_lab1_5.txt, программа считывает данные построчно.

Выходные данные:

```
Выберите метод:
1 - Алгоритм LU-разложения матрицы
2 - Метод прогонки
3 - Метод простых итераций и метод Зейделя
4 - Метод вращений
5 - QR-алгоритм
0 - выход
5

QR-алгоритм

A= [[-6. -4. 0.]
    [-7. 6. -7.]
    [-2. -6. -7.]]

Введите точность вычислений:
0.1

QR-разложение:

Q= [[-9.99164926e-01  4.08589111e-02 -4.47979085e-08]
    [-4.08589111e-02 -9.99164926e-01  6.90746515e-07]
    [-1.65373484e-08  6.92000084e-07  1.00000000e+00]]
R= [[ 1.13904204e+01 -2.31425346e+00 -2.68164855e+00]
    [ 7.83580323e-19 -9.96717086e+00 -1.09069150e+00]
    [-3.25685550e-23  1.06398111e-21 -5.67249618e+00]]

Собственные векторы:

x0 = -11.286350602972368
x1 = 9.95884705565746
x2 = -5.6724964526851025
```

3 Выводы

После выполнения пятого задания я увидел различия между методами Якоби и QR-алгоритмом. С помощью QR-разложения можно решить проблему с нахождением комплексных собственных значений, однако этот алгоритм на каждой итерации производит намного больше операций, что делает его более медленным. Также для реализации этого алгоритма было необходимо реализовать преобразование Хаусхолдера.

Программный код

```
from numpy import *
import numpy as np
import math
import cmath

def LU(a,b):
    u=a
    l=np.zeros([len(b),len(b)])
    for k in range(1,len(b)):
        for i in range(k-1,len(b)):
            for j in range(i,len(b)):
                l[j,i]=u[j,i]/u[i,i]
            for i in range(k,len(b)):
                for j in range(k-1,len(b)):
                    u[i,j]=u[i,j]-l[i,k-1]*u[k-1,j]
    result=(u,l)
    return result

def prog(a,b,c,d):
    n=len(d)
    x=np.zeros(n)
    p=np.zeros(n)
    q=np.zeros(n)
    i=0
    p[0]=-c[0]/b[0]
    q[0]=d[0]/b[0]
    for i in range(1,n-1):
        p[i]=-c[i]/(a[i]*p[i-1]+b[i])
        print('p= ',p)
        i=0
        for i in range(1,n):
            q[i]=(d[i]-a[i]*q[i-1])/(a[i]*p[i-1]+b[i])
            print('q= ',q)
        i=3
        x[n-1]=q[n-1]
        while i>=0:
            x[i]=p[i]*x[i+1]+q[i]
            i=i-1
    return x
```

```

def norma_matrix(a,count):
norm=0.
for i in range(0,count-1):
sum_a=0
for j in range(0,count-1):
sum_a+=math.fabs(a[i,j])
if sum_a>norm:
norm=sum_a
return norm

def norma_vector(b):
normv=b[0]
for i in range(1,len(b)-1):
if math.fabs(b[i])>normv:
normv=math.fabs(b[i])
return normv

def method_yakobi(a,b,count):
alfa=np.zeros((count,count))
beta=np.zeros(count)
for i in range(0,count):
beta[i]=b[i]/a[i,i]
for j in range(0,count):
if i==j:
alfa[i,j]=0
else:
alfa[i,j]=-a[i,j]/a[i,i]
res=(alfa,beta)
return res

def matr_vect(a,b):
res=np.zeros(len(b))
for i in range(0,len(b)):
for j in range(0,len(b)):
res[i]+=a[i,j]*b[j]
return res

def vect_p_vect(b,c):
res=np.zeros(len(b))
for i in range(0,len(b)):

```



```

res[i]=b[i]+c[i]
return res

def vect_m_vect(b,c):
res=np.zeros(len(b))
for i in range(0,len(b)):
res[i]=b[i]-c[i]
return res

def sum_razl(a,count):
b=np.zeros((count,count))
c=np.zeros((count,count))
for i in range(0,count):
for j in range(0,count):
if (j<=i):
b[i,j]=a[i,j]
if (j>i):
c[i,j]=a[i,j]
res=(b,c)
return res

def matr_m_matr(a,b,count):
for i in range(0,count-1):
for j in range(0,count-1):
a[i,j]-=b[i,j]
return a

def mpi(a,b,count,eps):
result=method_yakobi(a,b,count-1)
(alfa,beta)=result
print('\nalfa=',alfa)
print('\nbeta=',beta)
if norma_matrix(alfa,count)<1:
print('\nУсловие выполнено:')
print('||alfa||=',norma_matrix(alfa,count),'<1\n')
q=(norma_matrix(alfa,count)/(1-norma_matrix(alfa,count)))
x=beta
k=0
while True:
tmp=x
x=matr_vect(alfa,tmp)

```

```

x=vect_p_vect(beta,x)
epsk=vect_m_vect(x,tmp)
epsk=q*norma_vector(epsk)
k+=1
if epsk<eps:
break
print(k,'итераций')
print(x)
return

def zeidel(a,b,count,eps):
result=method_yakobi(a,b,count-1)
(alfa,beta)=result
print('\nalfa=',alfa)
print('\nbeta=',beta)
if norma_matrix(alfa,count)<1:
print('\nУсловие выполнено:')
print('||alfa||=',norma_matrix(alfa,count),'<1\n')
q=(norma_matrix(alfa,count)/(1-norma_matrix(alfa,count)))
x=beta
k=0
while True:
tmp=x
res=sum_razl(alfa,count-1)
(bb,cc)=res
ee=np.eye(count-1)
dd=matr_m_matr(ee,bb,count)
dd=linalg.inv(dd)
x=vect_p_vect(matr_vect(dd,matr_vect(cc,x)),matr_vect(dd,beta))
epsk=vect_m_vect(x,tmp)
epsk=norma_vector(epsk)
epsk=(norma_matrix(cc,count)/(1-norma_matrix(alfa,count)))*epsk
k+=1
if epsk<eps:
break
print(k,'итераций')
print(x)
return

def maxel(a,count):
maxelem=math.fabs(a[0,1])

```

```

l=0
m=1
for i in range(0,count):
    for j in range(i+1,count):
        if (math.fabs(a[i,j])>maxelem) and (i<j):
            maxelem=math.fabs(a[i,j])
            l=i
            m=j
    return maxelem,l,m

def multi_2matr(a,b,count):
    c=np.zeros((count,count))
    for i in range(0,count):
        for j in range(0,count):
            for k in range(0,count):
                c[i,j]+=a[i,k]*b[k,j]
    return c

def off(a,l,m,count):
    summ=0
    for l in range(0,count):
        for m in range(l+1,count):
            summ=summ+(a[l,m]*a[l,m])
    return math.sqrt(summ)

def trans(a,count):
    result=maxel(a,count)
    (maxelem,l,m)=result
    if a[l,l]==a[m,m]:
        phi=math.pi/4
    else:
        phi=0.5*(math.atan(2*a[l,m]/(a[l,l]-a[m,m])))
    c=math.cos(phi)
    s=math.sin(phi)
    u=np.eye(count)
    u[l,l]=c
    u[l,m]=-s
    u[m,m]=c
    u[m,l]=s
    ut=np.eye(count)
    ut[l,l]=c

```

```

ut[m,m]=c
ut[l,m]=s
ut[m,l]=-s
res=(u,ut,l,m)
return res

def rotation(a,count,eps):
k=0
v=np.eye(count)
while True:
result=trans(a,count)
(u,ut,l,m)=result
a=multi_2matr(multi_2matr(ut,a,count),u,count)
v=multi_2matr(v,u,count)
k+=1
res=(a,v)
if off(a,l,m,count)<eps:
break
print(k,'итераций')
return res

def housholder(A,n,eps):
A1 = np.zeros((n,n))
while abs(A[0,0]-A1[0,0]) >eps:
A1 = np.array(A)
u = 0
v = np.zeros((n,1))
H = np.zeros((n,n))
E = np.eye(n)
for i in range(1,n):
v[i,0] = A[i,0]
for j in range(0,n):
u = u + A[j,0]**2
v[0,0] = A[0,0] + np.sign(A[0,0])*sqrt(u)
H = E -2*(np.dot(v,v.transpose()) / np.dot(v.transpose(),v))
H1 = np.array(H)
A = np.dot(H,A)
v[0,0] = 0
u = A[1,1]**2 + A[2,1]**2
v[1,0] = A[1,1] + np.sign(A[1,1])*sqrt(u)
v[2,0] = A[2,1]

```

```

H = E -2*(np.dot(v,v.transpose()) / np.dot(v.transpose(),v))
A = np.dot(H,A)
R = np.array(A)
Q = np.dot(H1,H)
A = np.dot(R,Q)
return Q,R,A

def sv(A,n,eps):
Q,R,A = housholder(A,n,eps)
x0 = A[0,0]
print('\nx0 = ',x0)
a = 1
b = -A[1,1]-A[2,2]
c = A[1,1]*A[2,2]-A[1,2]*A[2,1]
discr = b ** 2 -4 * a * c
if discr >0:
x1 = (-b + math.sqrt(discr)) / (2 * a)
x2 = (-b -math.sqrt(discr)) / (2 * a)
print('x1 =',x1,'\nx2 =',x2)
elif discr == 0:
x = -b / (2 * a)
print('x ='% x)
else:
x1 = (-b + cmath.sqrt(discr)) / (2 * a)
x2 = (-b -cmath.sqrt(discr)) / (2 * a)
print('x1 =',x1,'\nx2 =',x2)
return x0,x1,x2

def main():
print('Лабораторная работа 1\nВариант 2')

while True:
print('\nВыберите метод:\n1 -Алгоритм LU-разложения матрицы\n2 -Метод прогонки\n3
-Метод простых итераций и метод Зейделя\n4 -Метод вращений\n5 -QR-алгоритм\n0
-выход')
k=int(input())
if k==0:
break
if k==1:
f = open('input_lab1_1.txt')
line = f.readlines()

```

```

count=len(line)
a=line[0].split()
a=list(map(float,a))
for i in range(1,count-1):
    tmp=line[i].split()
    tmp=list(map(float,tmp))
    a=np.row_stack((a,tmp))
b=line[count-1].split()
b=list(map(float,b))
f.close()

print('\nАлгоритм LU-разложения матрицы')
print('\nA= ',a)
print('b= ',b)
print('\nРешение СЛАУ:')
print('x= ',linalg.solve(a,b))
print('\nОпределитель матрицы:')
print('det A= ',linalg.det(a))
print('\nОбратная матрица:')
print('A*= ',linalg.inv(a))

res=LU(a,b)
(u,l)=res
print('\nU= ',u)
print('\nL= ',l)
print('\nПроверка:')
print('L*U= ',l.dot(u))
z=np.zeros(len(b))
for i in range(0,len(b)):
    sum=0
    for j in range(0,i):
        sum+=l[i,j]*z[j]
    z[i]=b[i]-sum

if k==2:
    f = open('input_lab1_2.txt')
    line = f.readlines()
    a=line[0].split()
    a=list(map(int,a))
    b=line[1].split()
    b=list(map(int,b))

```

```

c=line[2].split()
c=list(map(int,c))
d=line[3].split()
d=list(map(int,d))
f.close()

print('\nМетод прогонки')
print('\nРешение СЛАУ: x=',prog(a,b,c,d))

if k==3:
f = open('input_lab1_3.txt')
line = f.readlines()
count=len(line)
a=line[0].split()
a=list(map(float,a))
for i in range(1,count-1):
tmp=line[i].split()
tmp=list(map(float,tmp))
a=np.row_stack((a,tmp))
b=line[count-1].split()
b=list(map(float,b))
f.close()

print('\nA=',a)
print('\nb=',b)
print('\nВведите точность вычислений:')
eps=float(input())
print('eps=',eps)
print('\nМетод простых итераций')
mpi(a,b,count,eps)
print('\nМетод Зейделя')
zeidel(a,b,count,eps)

if k==4:
f = open('input_lab1_4.txt')
line = f.readlines()
count=len(line)
a=line[0].split()
a=list(map(float,a))
for i in range(1,count):
tmp=line[i].split()

```

```

tmp=list(map(float,tmp))
a=np.row_stack((a,tmp))
f.close()

print('\nМетод вращений')
print('\nA=',a)
print('\nВведите точность вычислений:')
eps=float(input())
print('eps=',eps)
result=rotation(a,count,eps)
(a,v)=result
print('\nСобственные значения:')
for i in range(0,count):
    print(a[i,i])
print('\nСобственные векторы:')
for i in range(0,count):
    sz=np.zeros(count)
    for j in range(0,count):
        sz[j]=v[j,i]
    print(sz)

if k==5:
    f = open('input_lab1_5.txt')
    line = f.readlines()
    n=len(line)
    A=line[0].split()
    A=list(map(float,A))
    for i in range(1,n):
        tmp=line[i].split()
        tmp=list(map(float,tmp))
        A=np.row_stack((A,tmp))
    f.close()

print('QR-алгоритм')
print('A=',A)
print('Введите точность вычислений:')
eps=float(input())
(q,r,a) = housholder(A,n,eps)
print('\nQR-разложение: \n\nQ= ',q,'\nR=',r)
(x0,x1,x2) = sv(A,n,eps)

```



```
return  
main()
```