Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №1 по курсу «Численные методы»
Вариант №1

Студент: А. О. Дубинин
Преподаватель: И. Э. Иванов
Группа: М8О-306Б
Дата:
Оценка:
Подпись:

Москва, 2020

Лабораторная работа №1

Задача: Разработать методы решения задач линейной алгебры.

Вариант 1

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

1.2 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.

1.3 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.

1.4 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.

1.5 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.

1 Решение

1.1 LUP – разложение.
Согласно, [1], идея разложения состоит в поиске трех матриц L, U, P размером nxn таких, что:

$$PA = LU$$

где

- L – единичная нижнетреугольная матрица
- R – верхнетреугольная матрица
- P – матрица перестановок

преимущество вычисления LUP-разложения матрица A основано на том, что система линейных уравнений решается гораздо легче, если её матрица треугольна, что и выполняется в случае матриц L и U.
Решим уравние $Ax = b$, умножим обе части уравнения на P, получим уравнение $PAx = Pb$, используя наше разложение получим итоговое уравнение:

$$LUx = Pb$$

Теперь можно решить полученное уравнение, решив две треугольные системы линейных уравнений. Обозначим $y = Ux$. Решим сначала нижнетреугольную систему линейных уравнений

$$Ly = Pb$$

найдя неизвестный y с помощью метода прямой подстановки. После этого решим верхнетреугольную систему линейных уравнений

$$Ux = y$$

с помощью метода обратной подстановки.

Прямая подстановка позволяет решить нижнетреугольную систему линейных уравнений для данных L, P и b за время (n^2).
Уравнения выглядят вот так:

$$\begin{aligned} y_1 &= b_{\pi[1]} \\ l_{21}y_1 + y_2 &= b_{\pi[2]} \\ l_{31}y_1 + l_{32}y_2 + y_3 &= b_{\pi[2]} \end{aligned}$$

Вычисляем по очереди элементы

$$y_1 = b_{\pi[1]}$$

$$y_2 = b_{\pi[2]} - l_{21}y_1$$

и тд. В общем виде формула выглядит так:

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij}y_j$$

Обратная подстановка решает верхнетреугольную-похожим образом, только вычисляется сначала x_n, общая формула:

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij}x_j \right) / u_{ii}$$

Разложение A на LU, разобьем A на четыре части.

$$A = \left(\begin{array}{c|c} a_{11} & a_{12}...a_{1n} \\ \hline a_{21} & a_{22}...a_{2n} \\ ... & ... \\ a_{n1} & a_{n2}...a_{nn} \end{array} \right) = \begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix}$$

где v — вектор-столбец размером $n-1$, w^T — вектор-строка размером n - 1, а A' — матрица размером (n - 1) (n - 1). Используя матричную алгебру (проверить полученный результат можно при помощи умножения), разложим матрицу A следующим образом:

$$A = \begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{pmatrix} =$$

$$= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & L'U' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & U' \end{pmatrix} = LU$$

1.2 Метод прогонки.
Согласно, [2], метод прогонки является частным случаем метода Гаусса. Он применянется для решения СЛАУ с трехдиагональными матрицами.

$$\begin{aligned} b_1x_1 + c_1x_2 &= d_1 \\ a_2x_1 + b_2x_2 + c_2x_3 &= d_2 \\ a_3x_1 + b_3x_2 + c_3x_3 &= d_3 \end{aligned}$$

.....

$$a_{n-1}x_{n-2} + b_{n-1}x_{n-1} + c_{n-1}x_n = d_{n-1}$$

$$a_n x_{n-1} + b_n x_n \qquad\qquad = d_n$$

при этом будем полагать, что

$$a_1 = 0$$

$$c_n = 0$$

Решение будем искать в виде

$$x_i = P_i x_{i+1} + Q_i, \qquad i = 1, 2, ..., n$$

где P_i, Q_i — прогоночные коэффициенты, подлежащие определению. Для этого выразим x_1 из первого уравнения системы через x_2 и получим

$$x_1 = \frac{-c_1}{b_1} x_2 + \frac{d_1}{b_1} = P_1 x_2 + Q_1$$

откуда следует

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

Из второго уравнения системы выразим x_2 через x_3 , получим

$$x_2 = \frac{-c_2}{b_2 + a_2 P_1} x_3 + \frac{d_2 - a_1 Q_1}{b_2 + a_2 P_1} = P_2 x_3 + Q_2$$

откуда следует

$$P_2 = \frac{-c_2}{b_2 + a_2 P_1}, Q_2 = \frac{d_2 - a_2 Q_1}{b_2 + a_2 P_1}$$

Продолжая этот процесс, получим из i-го уравнения системы

$$x_i = \frac{-c_i}{b_i + a_i P_{i-1}} x_{i+1} + \frac{d_i - a_i Q_{i-1}}{b_i + a_i P_{i-1}}$$

откуда следует

$$P_i = \frac{-c_i}{b_i + a_i P_{i-1}}, Q_i = \frac{d_i - a_i Q_{i-1}}{b_i + a_i P_{i-1}}$$

Из последнего уравнения системы имеем

$$x_n = 0 * x_{n+1} + Q_n$$

, т.е., так как $c_n = 0$,

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

Таким образом, прямой ход определения прогоночных коэффициентов P_i , Q_i , $i = 1, 2, ..., n$, завершен. Обратный ход метода прогонки осуществляется в соответствии с выражением

$$x_n = P_n x_{n+1} + Q_n = 0 * x_{n+1} + Q_n = Q_n$$

$$x_{n-1} = P_{n-1}x_n + Q_{n-1}$$
$$x_{n-2} = P_{n-2}x_{n-1} + Q_{n-2}$$
$$.....$$
$$x_1 = P_1x_2 + Q_1$$

Для устойчивости и корректности метода прогонки достаточно выполнения следующих условий:

$$a_i \neq 0, c_i \neq 0, \qquad i = 2, 3, ..., n-1,$$
$$|b_i| \geq |a_i| + |c_i|, \qquad i = 1, 2, ..., n,$$

причем строгое неравенство имеет место хотя бы при одном i.

1.3 Метод простых итераций и метод Зейделя.

При большом числе уравнений прямые методы решения СЛАУ (за исключением метода прогонки) становятся труднореализуемыми на ЭВМ, прежде всего из-за сложности хранения и обработки матриц большой размерности. В то же время характерной особенностью ряда часто встречающихся в прикладных задачах СЛАУ является разреженность матриц. Число ненулевых элементов таких матриц мало по сравнению с их размерностью. Для решения СЛАУ с разреженными матрицами предпочтительнее использовать итерационные методы.

Методы последовательных приближений, в которых при вычислении последующего приближения используются предыдущие, уже известные приближенные решения, называются итерационными.

Рассмотрим СЛАУ с невырожденной матрицей ($det \neq 0$)

$$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1,$$
$$a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2,$$
$$.....$$
$$a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n,$$

Приведем СЛАУ к эквивалентному виду

$$x_1 = \alpha_{11}x_1 + \alpha_{12}x_2 + ... + \alpha_{1n}x_n + \beta_1,$$
$$x_2 = \alpha_{21}x_1 + \alpha_{22}x_2 + ... + \alpha_{2n}x_n + \beta_2,$$
$$.....$$
$$x_n = \alpha_{n1}x_1 + \alpha_{n2}x_2 + ... + \alpha_{nn}x_n + \beta_n,$$

или, в векторно-матричной форме

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

Такое приведение может быть выполнено различными способами. Одним из наиболее распространенных является следующий.
Разрешим систему относительно неизвестных при ненулевых диагональных элементах $a_u \neq 0, i = 1, 2, ..., n$; если какой-либо коэффициент на главной диагонали равен нулю, достаточно соответствующее уравнение поменять местами с любым другим уравнением. Получим следующие выражения для компонентов вектора β и матрицы α эквивалентной системы:

$$\beta_i = \frac{b_i}{a_{ii}}; \alpha_{ij} = -\frac{a_{ij}}{a_{ii}}, i = 1...n, j = 1...n, i \neq j$$

$$\alpha_{ii} = 0, i = 1...n$$

В качестве нулевого приближения $x^{(0)}$ вектора неизвестных примем вектор правых частей $x^{(0)} = \beta$. Тогда метод простых итераций примет вид

$$x^{(0)} = \beta$$

$$x^{(1)} = \alpha x^0 + \beta$$

$$x^{(2)} = \alpha x^1 + \beta$$

$$.....$$

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

В вычислительном процессе участвуют только произведения матрицы на вектор. Это позволяет работать только с ненулевыми элементами матрицы, что значительно упрощает процесс хранения и обработки матриц.
Имеет место следующее достаточное условие сходимости метода простых итераций.

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

Приведем также необходимое и достаточное условие сходимости метода простых итераций.

- Для сходимости итерационного процесса необходимо и достаточно, чтобы спектр матрицы α эквивалентной системы лежал внутри круга с радиусом, равным единице.

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

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

где x^* — точное решение СЛАУ. Процесс итераций останавливается про выполнении условия $\epsilon^{(k)} \leq \epsilon$, где ϵ — задаваемая вычислителем погрешность. Из этого следует неравенство

$$||x^{(k)} - x^*|| \leq \frac{||\alpha||^k}{1 - ||\alpha||}||x^{(1)} - x^{(0)}||$$

можно получить априорную оценку необходимого для достижения заданной точности числа итераций. При использовании в качестве начального приближения вектора β такая оценка определится неравенством

$$\frac{||\alpha||^{k+1}}{1 - ||\alpha||}||\beta|| \leq \epsilon$$

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

$$x_1^{(k+1)} = \alpha_{11}x_1^{(k)} + \alpha_{12}x_2^{(k)} + ... + \alpha_{1n}x_n^{(k)} + \beta_1,$$

$$x_2^{(k+1)} = \alpha_{21}x_1^{(k+1)} + \alpha_{22}x_2^{(k)} + ... + \alpha_{2n}x_n^{(k)} + \beta_2,$$

$$.....$$

$$x_n^{(k+1)} = \alpha_{n1}x_1^{(k+1)} + \alpha_{n2}x_2^{(k+1)} + ... + \alpha_{nn}x_n^{(k)} + \beta_n,$$

1.4 Метод вращений Якоби численного решения задач на собственные значения и собственные векторы матриц.
Метод вращений Якоби применим только для симметрических матриц $A_{n*n}(A = A^T)$ и решает полную проблему собственных значений и собственных векторов таких матриц. Он основан на отыскании с помощью итерационных процедур матрицы U в преобразовании подобия $\Delta = U^{-l}AU$, а поскольку для симметрических матриц A матрица преобразования подобия U является ортогональной $U^{-l} = UT$, имеем $\Delta = U^T AU$, здесь Δ — диагональная матрица с собственными значениями на главной диагонали

$$\Delta = \begin{pmatrix} \lambda_1 & ... & 0 \\ ... & ... & ... \\ 0 & ... & \lambda_n \end{pmatrix}$$

Пусть дана симметрическая матрица A. Требуется вычислить для нее с погрешностью ϵ все собственные значения и соответствующие им собственные векторы. Приведем алгоритм метода вращения. Пусть известна матрица $A^{(k)}$ на k-й итерации, при этом k = 0 для $A^{(0)}$ = A.

Выбирается максимальный по модулю недиагональный элемент $a_{ij}^{(k)}$ матрицы $A^{(k)}$. Ставится задача найти ортогональную матрицу $U^{(k)}$ такую, чтобы в результате преобразования подобия $A^{(k+1)} = U^{(k)T}A^{(k)}U^{(k)}$ произошло обнуление элемента $a_{ij}^{(k+1)}$ матрицы $A^{(k+1)}$. В качестве ортогональной матрицы выбирается матрица вращения, имеющая следующий вид:

$$U^k = \begin{pmatrix} 1 & ... & ... & ... & ... & ... & ... & ... & ... \\ ... & ... & ... & ... & ... & ... & ... & 0 & ... \\ ... & ... & 1 & ... & ... & ... & ... & ... & ... \\ ... & ... & ... & cos\phi^{(k)} & ... & -sin\phi^{(k)} & ... & ... & ... \\ ... & ... & ... & 1 & ... & ... & ... & ... & ... \\ ... & ... & ... & ... & ... & ... & ... & ... & ... \\ ... & ... & ... & ... & ... & 1 & ... & ... & ... \\ ... & ... & ... & sin\phi^{(k)} & ... & cos\phi^{(k)} & ... & ... & ... \\ ... & ... & ... & ... & ... & ... & 1 & ... & ... \\ ... & 0 & ... & ... & ... & ... & ... & ... & ... \\ ... & ... & ... & ... & ... & ... & ... & ... & 1 \end{pmatrix}$$

В матрице вращения на пересечении i-й строки и j-го столбца находится элемент $u_{ij}^{(k)} = -sin\phi^{(k)}$, где $\phi^{(k)}$ – угол вращения, подлежащий определению. Симметрично относительно главной диагонали (j-я строка, i-й столбец) расположен элемент $u_{ji}^{(k)} = sin\phi^{(k)}$. Диагональные элементы $u_{ii}^{(k)}$ и $u_{jj}^{(k)}$ равны соответсвенно $u_{ii}^{(k)} = cos\phi^{(k)}$ и $u_{jj}^{(k)} = cos\phi^{(k)}$; другие диагональные элементы равны 1. Остальные элементы в матрице вращения $U^{(k)}$ равны нулю.
Угол вращения $\phi^{(k)}$ определяется из условия $a_{ij}^{(k+1)} = 0$

$$\phi^{(k)} = \frac{1}{2} arctg \frac{2a_{ij}^{(k)}}{a_{ii}^{(k)} - a_{jj}^{(k)}}$$

Строится матрица $A^{(k+1)}$

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

в которой элемент $a_{ij}^{(k+1)} \approx 0$ В качестве критерия окончания итерационного процесса используется условие малости суммы квадратов внедиагональных элементов:

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

Если $t(A^{(k+1)}) > \epsilon$, то итерационный процесс продолжается. Если нет, то итерационный процесс останавливается, и в качестве искомых собственных значений принимаются

$$\lambda_1 \approx a_{11}^{(k+1)}, \lambda_2 \approx a_{22}^{(k+1)}, ..., \lambda_n \approx a_{nn}^{(k+1)}.$$

Координатными столбцами собственных векторов матрицы A в единичном базисе будут столбцы матрицы U.

1.5 QR-алгоритм нахождения собственных значений матриц.

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 = QR, где Q — ортогональная матрица, $Q^{-1} = Q^T$, а R — верхняя треугольная.

Такое разложение существует для любой квадратной матрицы. Одним из возможных подходов к построению QR-разложения является использование преобразования Хаусхолдера, позволяющего обратить в нуль группу поддиагональных элементов столбца матрицы. Преобразование Хаусхолдера осуществляется с использованием матрицы Хаусхолдера, имеющей вид

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

где v — произвольный ненулевой вектор-столбец, E — единичная матрица, vv^T — квадратная матрица того же размера. Легко убедиться, что любая матрица такого вида является симметричной и ортогональной. При этом произвол в выборе вектора v дает возможность построить матрицу, отвечающую некоторым дополнительным требованиям.

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

$$\widetilde{b} = Hb, b = (b_1, b_2, ..., b_n)^T, \widetilde{b} = (\widetilde{b_1}, 0, ..., 0)^T.$$

Тогда вектор v определится следующим образом:

$$v = b + sign(b_1)||b_2||e_1$$

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 этого процесса.

Положим $A_0 = A$ и построим преобразование Хаусхолдера H_1, переводящее матрицу A_0 в матрицу A_1 с нулевыми элементами первого столбца под главной диагональю

$A_1 = H_1 A_0$

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

$$v^1_1 = a^0_{11} + sign(a^0_{11}))\Big(\sum_{j=1}^{n}(a^0_{j1})^2\Big)^{\frac{1}{2}},$$

$$v^1_i = a^0_{i1}, i = 2, 3, ..., n.$$

Матрица Хаусхолдера H_1 вычисляется

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

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

$$v^2_1 = 0,$$

$$v^2_2 = a^1_{22} + sign(a^1_{22}))\Big(\sum_{j=2}^{n}(a^1_{j2})^2\Big)^{\frac{1}{2}},$$

$$v^2_i = a^1_{i1}, i = 3, 4, ..., n.$$

Повторяя процесс (n - 1) раз, получим искомое разложение A = QR, где

$$Q = (H_{n-1} H_{n-2} ... H_1)^T, R = A_{n-1}$$

Следует отметить определенное сходство рассматриваемого процесса с алгоритмом Гаусса. Отличие заключается в том, что здесь обнуление поддиагональных элементов соответствующего столбца осуществляется с использованием ортогонального преобразования. Процедура QR-разложения многократно используется в QR-алгоритме вычисления собственных значений. Строится следующий итерационный процесс:
$A^{(0)} = A$,
$A^{(0)} = Q^{(0)} R^{(0)}$ – производится QR-разложение,
$A^{(1)} = R^{(0)} Q^{(0)}$ – производится перемножение матриц,
.....
$A^{(k)} = Q^{(k)} R^{(k)}$ – разложение,

$A^{(k+1)} = R^{(k+1)}Q^{(k)}$ – перемножение

Таким образом, каждая итерация реализуется в два этапа. На первом этапе осуществляется разложение матрицы $A^{(k)}$; в про- изведение ортогональной $Q^{(k)}$ и верхней треугольной $R^{(k)}$ матриц, а на втором – полученные матрицы перемножаются в обратном порядке.
Нетрудно показать подобие матриц $A^{(k+1)}$ и $A^{(k)}$. Действительно, учитывая ортогональность $Q^{(k)}$, $Q^{(k)T}Q^{(k)} = \mathrm{E}$, можно записать:

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

Аналогично можно показать, что любая из матриц $A^{(k)}$ ортогонально подобна матрице A. При отсутствии у матрицы кратных собственных значений последовательность $A^{(k)}$ сходится к верхней треугольной матрице (в случае, когда все собственные значения вещественны) или к верхней квазитреугольной матрице (если имеются комплексно-сопряженные пары собственных значений).
Таким образом, каждому вещественному собственному значению будет соответствовать столбец со стремящимися к нулю поддиагональными элементами и в качестве критерия сходимости итерационного процесса для таких собственных значений можно использовать 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 x 2, т.е. матрица $A^{(k)}$ имеет блочнодиагональную структуру. Принципиально то, что элементы этих блоков изменяются от итерации к итерации без видимой закономерности, в то время как комплексно-сопряженные собственные значения, определяемые каждым блоком, имеют тенденцию к сходимости. Это обстоятельство необходимо учитывать при формировании критерия выхода из итерационного процесса. Если в ходе итераций прослеживается комплексно-сопряженная пара собственных значений, соответствующая блоку, образуемому элементами j-го и (j + 1)-го столбцов $a_{jj}^{(k)}, a_{jj+1}^{(k)}, a_{j+1j}^{(k)}, a_{j+1j+1}^{(k)}$, то, несмотря на значительное изменение в ходе итераций самих этих элементов, собственные значения, соответствующие данному блоку и определяемые из решения квадратного уравнения

$$(a_{jj}^{(k)} - \lambda^{(k)})(a_{j+1j+1}^{(k)} + \lambda^{(k)}) = a_{jj+1}^{(k)}a_{j+1j}^{(k)}$$

начиная с некоторого k отличаются незначительно. В качестве критерия окончания итераций для таких блоков может быть использовано условие $|\lambda^{(k)}) - \lambda^{(k-1)})| \leq \epsilon$

2 Исходный код

1.1 LUP – разложение.

```
import sys


class MatrixError(Exception):
    pass


class InverseMatrixError(MatrixError):
    pass


class DegenerateMatrixError(MatrixError):
    pass


def get_cofactor(A, row, column, m):
    n = len(A)
    i1 = 0
    j1 = 0
    temp = [[0 for _ in range(m)] for _ in range(m)]
    for i in range(n):
        for j in range(n):
            if i != row and j != column:
                temp[i1][j1] = A[i][j]
                j1 += 1
                if j1 == m:
                    j1 = 0
                    i1 += 1
    return temp


def adjoint(A):
    n = len(A)
    adjoint_A = [[0 for _ in range(n)] for _ in range(n)]
    if n == 1:
        adjoint_A[0][0] = 1
        return adjoint_A

    sign = 1
    for i in range(n):
        for j in range(n):
            temp = get_cofactor(A, i, j, n - 1)

            sign = 1 if (i + j) % 2 == 0 else -1
```

```
45
46              adjoint_A[j][i] = sign * determinant_of_matrix(temp)
47
48      return adjoint_A
49
50
51  def inverse(A):
52      n = len(A)
53      det = determinant_of_matrix(A)
54      if det == 0:
55          raise InverseMatrixError("The inverse matrix does not exist")
56
57      adjoint_A = adjoint(A)
58
59      inverse_A = [[0 for _ in range(n)] for _ in range(n)]
60      for i in range(n):
61          for j in range(n):
62              inverse_A[i][j] = adjoint_A[i][j] / det
63
64      return inverse_A
65
66
67  def determinant_of_matrix(A):
68      n = len(A)
69
70      if n == 1:
71          return A[0][0]
72
73      sign = 1
74      D = 0
75      for i in range(n):
76          temp = get_cofactor(A, 0, i, n - 1)
77          D += sign * A[0][i] * determinant_of_matrix(temp)
78          sign = -sign
79
80      return D
81
82
83  def LUP_decomposition(A):
84      n = len(A)
85      pi = list(range(0, len(A)))
86      for k in range(n):
87          p = -1000
88          for i in range(k, n):
89              if A[i][k] > p:
90                  p = A[i][k]
91                  k_ = i
92          if p == -1000:
93              raise DegenerateMatrixError("Matrix is degenerate")
```

```
94          pi[k], pi[k_] = pi[k_], pi[k]
95          for i in range(n):
96              A[k][i], A[k_][i] = A[k_][i], A[k][i]
97          for i in range(k + 1, n):
98              A[i][k] = A[i][k] / A[k][k]
99              for j in range(k + 1, n):
100                 A[i][j] = A[i][j] - A[i][k] * A[k][j]
101
102     L = list()
103     U = list()
104     for i in range(len(A)):
105         L_ = []
106         U_ = []
107         for j in range(len(A)):
108             if i > j:
109                 L_.append(round(A[i][j], 1))
110                 U_.append(0)
111             else:
112                 if i == j:
113                     L_.append(1)
114                 else:
115                     L_.append(0)
116                 U_.append(round(A[i][j], 1))
117         L.append(L_)
118         U.append(U_)
119     return L, U, pi
120
121
122 def LUP_solve(L, U, pi, b):
123     n = len(L)
124     x, y = [0 for _ in range(n)], [0 for _ in range(n)]
125     for i in range(n):
126         sum = 0
127         for j in range(i):
128             sum += L[i][j] * y[j]
129         y[i] = b[pi[i]] - sum
130
131     for i in range(n - 1, -1, -1):
132         sum = 0
133         for j in range(i + 1, n):
134             sum += U[i][j] * x[j]
135         x[i] = round((y[i] - sum) / U[i][i], 1)
136     return x
137
138
139 if __name__ == "__main__":
140
141     if len(sys.argv) != 3:
142         print("use {} <matrix_file> <b_file>")
```

```
        exit(0)

    matrix_file = sys.argv[1]
    b_file = sys.argv[2]

    A = []
    with open(matrix_file) as m:
        for line in m:
            A.append(list(map(int, line.split())))

    b = []
    with open(b_file) as m:
        b = list(map(int, m.read().split()))

    print("determinant A = {}".format(determinant_of_matrix(A)))
    print("inverse A:")
    inverse_A = inverse(A)
    for i in inverse_A:
        print(i)

    L, U, pi = LUP_decomposition(A)
    x = LUP_solve(L, U, pi, b)
    print("L:")
    for i in L:
        print(i)
    print("U:")
    for i in U:
        print(i)
    print("P:")
    for i in pi:
        for j in range(len(pi)):
            if i == j:
                print(1, end=" ")
            else:
                print(0, end=" ")
        print()
    print("x:")
    for i in x:
        print(i)
```

1.2 Метод прогонки.

```
import sys


class MatrixError(Exception):
    pass


```

```
8  def forward(matrix, D):
9      n = len(matrix)
10     A = [0 for _ in range(n)]
11     B = [0 for _ in range(n)]
12     b = matrix[0][0]
13     c = matrix[0][1]
14     d = D[0]
15     A[0] = -c / b
16     B[0] = d / b
17     for i in range(1, n - 1):
18         a = matrix[i][i - 1]
19         b = matrix[i][i]
20         c = matrix[i][i + 1]
21         d = D[i]
22         A[i] = -c / (b + a * A[i - 1])
23         B[i] = (d - a * B[i - 1]) / (b + a * A[i - 1])
24     A[n - 1] = 0
25     a = matrix[n - 1][n - 2]
26     b = matrix[n - 1][n - 1]
27     d = D[n - 1]
28     B[n - 1] = (d - a * B[n - 2]) / (b + a * A[n - 2])
29     return A, B
30
31
32 def back(A, B):
33     n = len(A)
34     x = [0 for _ in range(n)]
35     x[n - 1] = B[n - 1]
36     for i in range(n - 2, -1, -1):
37         x[i] = A[i] * x[i + 1] + B[i]
38     return x
39
40
41 if __name__ == "__main__":
42
43     if len(sys.argv) != 3:
44         print("use {} <matrix_file> <b_file>")
45         exit(0)
46
47     matrix_file = sys.argv[1]
48     b_file = sys.argv[2]
49
50     matrix = []
51     with open(matrix_file) as m:
52         for line in m:
53             matrix.append(list(map(int, line.split())))
54
55     b = []
56     with open(b_file) as m:
```

```
        b = list(map(int, m.read().split()))

    A, B = forward(matrix, b)
    x = list(map(int, back(A, B)))
    print(x)
```

1.3 Метод простых итераций и метод Зейделя.

```
import sys

epsilon = 0.01


class MatrixError(Exception):
    pass


class MethodError(Exception):
    pass


def zendel_method(alpha_norma, beta, alpha):
    x_old = beta.copy()
    x_new = [0 for _ in range(len(beta))]
    epsilon_k_fst = 1
    cnt_iteration = 0
    while True:
        x_new = [0 for _ in range(len(beta))]
        for i in range(len(alpha)):
            for j in range(len(alpha[0])):
                if j < i:
                    x_new[i] += alpha[i][j] * x_new[j]
                else:
                    x_new[i] += alpha[i][j] * x_old[j]
            x_new[i] += beta[i]

        diffence_of_x = 0
        for i in range(len(beta)):
            diffence_of_x = max(abs(x_new[i] - x_old[i]), diffence_of_x)
        epsilon_k = epsilon_k_fst * diffence_of_x
        if epsilon_k <= epsilon:
            break
        cnt_iteration += 1

        x_old = x_new.copy()
    print("Zendel iteration:")
    print([round(i, 2) for i in x_new])
    print("Iterations = {}".format(cnt_iteration))

```

```
42
43 def simple_iterations_aux(alpha_norma, beta, alpha):
44     x_old = beta.copy()
45     x_new = [0 for _ in range(len(beta))]
46     epsilon_k_fst = 1
47     cnt_iteration = 0
48     while True:
49         x_new = [0 for _ in range(len(beta))]
50         for i in range(len(alpha)):
51             for j in range(len(alpha[0])):
52                 x_new[i] += alpha[i][j] * x_old[j]
53             x_new[i] += beta[i]
54
55         diffence_of_x = 0
56         for i in range(len(beta)):
57             diffence_of_x = max(abs(x_new[i] - x_old[i]), diffence_of_x)
58         epsilon_k = epsilon_k_fst * diffence_of_x
59         if epsilon_k <= epsilon:
60             break
61         cnt_iteration += 1
62
63         x_old = x_new.copy()
64     print("Simple iteration:")
65     print([round(i, 2) for i in x_new])
66     print("Iterations = {}".format(cnt_iteration))
67
68
69 def simple_iteraions(matrix, b):
70     n = len(matrix)
71     alpha = [[0 for _ in range(n)] for _ in range(n)]
72     beta = [0 for _ in range(n)]
73     for i in range(n):
74         aii = matrix[i][i]
75         beta[i] = b[i] / aii
76         for j in range(n):
77             if i != j:
78                 alpha[i][j] = -(matrix[i][j] / aii)
79
80     # alpha norma
81     alpha_norma = 0
82     for i in range(len(alpha)):
83         tmp = 0
84         for j in range(len(alpha[0])):
85             tmp += abs(alpha[i][j])
86         alpha_norma = max(tmp, alpha_norma)
87
88     # if alpha_norma >= 1:
89         # raise MethodError("Alpha >= 1, alpha = {}".format(alpha_norma))
90
```

```
    simple_iterations_aux(alpha_norma, beta, alpha)
    print()
    zendel_method(alpha_norma, beta, alpha)


if __name__ == "__main__":

    if len(sys.argv) != 3:
        print("use {} <matrix_file> <b_file>")
        exit(0)

    matrix_file = sys.argv[1]
    b_file = sys.argv[2]

    matrix = []
    with open(matrix_file) as m:
        for line in m:
            matrix.append(list(map(int, line.split())))

    b = []
    with open(b_file) as m:
        b = list(map(int, m.read().split()))
    print("epsilon = {}\n".format(epsilon))
    simple_iteraions(matrix, b)
```

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

```
import sys
from math import pi, atan, cos, sin, sqrt

epsilon = 0.01


class MatrixError(Exception):
    pass


class MethodError(Exception):
    pass


def matrix_product(A, B):
    n1 = len(A)
    m1 = len(A[0])
    n2 = len(B)
    m2 = len(B[0])
    if m1 != n2:
        raise MatrixError("Matrix product error")
```

```
22      result = [[0 for _ in range(m2)] for _ in range(n1)]
23      for i in range(n1):
24          for k in range(m2):
25              for j in range(m1):
26                  result[i][k] += A[i][j] * B[j][k]
27      return result
28
29
30  def transpose_matrix(A):
31      n = len(A)
32      m = len(A[0])
33      result = [[0 for _ in range(n)] for _ in range(m)]
34      for i in range(n):
35          for j in range(m):
36              result[j][i] = A[i][j]
37      return result
38
39
40  def find_max(A):
41      n = len(A)
42      i_r, j_r = 0, 0
43      maxx = 0
44      for i in range(n):
45          for j in range(i + 1, n):
46              if abs(A[i][j]) > maxx:
47                  maxx = abs(A[i][j])
48                  i_r = i
49                  j_r = j
50      return i_r, j_r
51
52
53  def spin_method(A):
54      n = len(A)
55      A_new = A.copy()
56      eigenvectors = [[0 if i != j else 1 for j in range(n)] for i in range(n)]
57      while True:
58          i_maxx, j_maxx = find_max(A_new)
59          if A_new[i_maxx][i_maxx] == A_new[j_maxx][j_maxx]:
60              phi = pi / 4
61          else:
62              phi = 0.5 * atan((2 * A_new[i_maxx][j_maxx]) / (A_new[i_maxx][i_maxx] -
                    A_new[j_maxx][j_maxx]))
63
64          U = [[0 if i != j else 1 for j in range(n)] for i in range(n)]
65          U[i_maxx][i_maxx] = cos(phi)
66          U[j_maxx][j_maxx] = cos(phi)
67          U[i_maxx][j_maxx] = -sin(phi)
68          U[j_maxx][i_maxx] = sin(phi)
69
```

```
        eigenvectors = matrix_product(eigenvectors, U)

        UT = transpose_matrix(U)
        A_new = matrix_product(matrix_product(UT, A_new), U)
        epsilon_k = 0
        for i in range(n):
            for j in range(i):
                epsilon_k += A_new[i][j] ** 2
        epsilon_k = sqrt(epsilon_k)
        if epsilon_k < epsilon:
            break

    eigenvalues = [round(A_new[i][i], 2) for i in range(n)]
    eigenvectors = [[round(eigenvectors[i][j], 4) for j in range(n)] for i in range(n)]

    return eigenvalues, eigenvectors


if __name__ == "__main__":

    if len(sys.argv) != 3:
        print("use {} <matrix_file> <b_file>")
        exit(0)

    matrix_file = sys.argv[1]
    b_file = sys.argv[2]

    matrix = []
    with open(matrix_file) as m:
        for line in m:
            matrix.append(list(map(int, line.split())))

    print("epsilon = {}\n".format(epsilon))
    eigenvalues, eigenvectors = spin_method(matrix)
    print("eigenvalues:")
    print(eigenvalues)
    print("eigenvectors")
    for i in range(len(eigenvectors)):
        print("h{} = ".format(i), end='')
        for j in range(len(eigenvectors)):
            print("{0:.3f}".format(eigenvectors[j][i]), end=" ")
        print()
```

1.5 QR-алгоритм нахождения собственных значений матриц.

```
import sys
import numpy as np
from numpy.linalg import norm, eig
from matrix import Matrix, Vector

```

```
6  eps = 0.01
7
8  def numpy_eig(matrix, my_values):
9      print("My eigenvalues:")
10     print(my_values)
11
12     a = np.array(matrix.get_data())
13     eig_np = eig(a)
14     print("Numpy eigenvalues:")
15     print(eig_np[0].round(3))
16
17
18 def sign(x):
19     return -1 if x < 0 else 1 if x > 0 else 0
20
21
22 def householder(a, sz, k):
23     v = np.zeros(sz)
24     a = np.array(a.get_data())
25     v[k] = a[k] + sign(a[k]) * norm(a[k:])
26     for i in range(k + 1, sz):
27         v[i] = a[i]
28     v = v[:, np.newaxis]
29     H = np.eye(sz) - (2 / (v.T  v)) * (v  v.T)
30     return Matrix.from_list(H.tolist())
31
32
33 def get_QR(A):
34     sz = len(A)
35     Q = Matrix.identity(sz)
36     A_i = Matrix(A)
37
38     for i in range(sz - 1):
39         col = A_i.get_column(i)
40         H = householder(col, len(A_i), i)
41         Q = Q.multiply(H)
42         A_i = H.multiply(A_i)
43
44     return Q, A_i
45
46
47 def get_roots(A, i):
48     sz = len(A)
49     a11 = A[i][i]
50     a12 = A[i][i + 1] if i + 1 < sz else 0
51     a21 = A[i + 1][i] if i + 1 < sz else 0
52     a22 = A[i + 1][i + 1] if i + 1 < sz else 0
53     return np.roots((1, -a11 - a22, a11 * a22 - a12 * a21))
54
```

```
55
56 def finish_iter_for_complex(A, eps, i):
57     Q, R = get_QR(A)
58     A_next = R.multiply(Q)
59     lambda1 = get_roots(A, i)
60     lambda2 = get_roots(A_next, i)
61     return True if abs(lambda1[0] - lambda2[0]) <= eps and \
62                    abs(lambda1[1] - lambda2[1]) <= eps else False
63
64
65 def get_eigenvalue(A, eps, i):
66     A_i = Matrix(A)
67     while True:
68         Q, R = get_QR(A_i)
69         A_i = R.multiply(Q)
70         a = np.array(A_i.get_data())
71         if norm(a[i + 1:, i]) <= eps:
72             res = (a[i][i], False, A_i)
73             break
74         elif norm(a[i + 2:, i]) <= eps and finish_iter_for_complex(A_i, eps, i):
75             res = (get_roots(A_i, i), True, A_i)
76             break
77     return res
78
79
80 def QR_method(A, eps):
81     res = Vector()
82     i = 0
83     A_i = Matrix(A)
84     while i < len(A):
85         eigenval = get_eigenvalue(A_i, eps, i)
86         if eigenval[1]:
87             res.extend(eigenval[0])
88             i += 2
89         else:
90             res.append(eigenval[0])
91             i += 1
92         A_i = eigenval[2]
93     return res, i
94
95
96 if __name__ == '__main__':
97
98     if len(sys.argv) != 2:
99         print("use {} <matrix_file>")
100        exit(0)
101
102    matrix_file = sys.argv[1]
103
```

```
104     data = []
105     with open(matrix_file) as m:
106         for line in m:
107             data.append(list(map(int, line.split())))
108 
109     A = Matrix()
110     A.data = data
111     print("epsilon = {}\n".format(eps))
112     tmp, count_iter = QR_method(A, eps)
113     numpy_eig(A, tmp)
```

3 Вывод программы

1.1 LUP – разложение.
Вывод определителя, обратной матрицы, L, U, P и решения СЛАУ.

```
art@mars: python p_1/main.py matrix.txt b.txt
determinant A = 8
inverse A:
[-11.5,-2.0,42.5,18.0]
[5.125,1.0,-18.125,-8.0]
[-0.75,0.0,2.75,1.0]
[0.125,0.0,-0.125,0.0]
L:
[1,0,0,0]
[-3.0,1,0,0]
[-2.0,0.0,1,0]
[1.0,0.0,0.0,1]
U:
[1,2,-2,6]
[0,1.0,8.0,31.0]
[0,0,1.0,22.0]
[0,0,0,-8.0]
P:
1 0 0 0
0 1 0 0
0 0 0 1
0 0 1 0
x:
2.0
4.0
2.0
3.0
```

1.2 Метод прогонки.
Вывод решения СЛАУ.

```
art@mars: python p_2/main.py matrix.txt b.txt
[7,5,4,6,4]
```

1.3 Метод простых итераций и метод Зейделя.
Точность $\varepsilon = 0.01$
Вывод решения СЛАУ и кол-во итераций понадобившись для нахождения решения.

```
art@mars: python p_3/main.py matrix.txt b.txt
epsilon = 0.01

Simple iteration:
[8.0,4.0,3.0,9.0]
Iterations = 18

Zendel iteration:
[8.0,4.0,3.0,9.0]
Iterations = 8
```

1.4 Метод вращений Якоби численного решения задач на собственные значения и собственные векторы матриц.
Точность $\varepsilon = 0.01$
Вывод собственных значений и собственных векторов.

```
art@mars: python p_4/main.py matrix.txt b.txt
epsilon = 0.01

eigenvalues:
[-3.71,2.07,-19.36]
eigenvectors
h0 = 0.887 0.346 0.305
h1 = -0.052 0.732 -0.679
h2 = -0.459 0.587 0.667
```

1.5 QR-алгоритм нахождения собственных значений матриц.
Точность $\varepsilon = 0.01$
Вывод собственных значений и сравнения с выводом тех же собственных значений библиотекой numpy.

```
art@mars: python p_5/main.py matrix.txt b.txt
epsilon = 0.01

My eigenvalues:

-13.5018
(5.250200311024764+2.8650261230841987j)
(5.250200311024764-2.8650261230841987j)

Numpy eigenvalues:
[-13.501+0.j      5.251+2.865j   5.251-2.865j]
```

4 Выводы

Благодаря этой лабораторной работе, я узнал, что можно быстро с помощью компьютера находить решения СЛАУ, собственные значения и собственные вектора, а не использовать ручку и тетрадь. Узнал несколько численных методов линейной алгебры и понял, что множество этих методов подразумевают под собой скрытый метод Гаусса, только они работают быстрее. Правда у них есть и минусы, некоторые методы нельзя применить к любой матрицы, а некоторые итерационные методы считают значения с определенной точностью.

Конечно нужно разработывать численные методы линейной алгебры, потому что огромные матрицы, например из задач экономики, все равно считаются долго, даже на хороших современных компьютерах.

Список литературы

[1] Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн
Алгоритмы: построение и ана- лиз, 3-е изд.
(ISBN 978-5-8459-1794-2 (рус.))

[2] *Численные методы. Учебник*
Пирумов Ульян Гайкович, Гидаспов Владимир Юрьевич
(ISBN 978-5-534-03141-6)