МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

Кафедра компьютерных технологий и систем

Лабораторная работа №1 по курсу «ВМА»
«Методы решения СЛАУ»

Вариант 2

Выполнил
Статкевич Захар Дмитриевич
студент 3-го курса 6 группы

Преподаватель:
Будник А. М.

Минск, 2023

ОГЛАВЛЕНИЕ

Входные данные.

Дана матрица A:

$$\begin{bmatrix} 0.6444 & 0 & -0.1683 & 0.1184 & 0.1973 \\ -0.0395 & 0.4208 & 0 & -0.0802 & 0.0263 \\ 0.0132 & -0.1184 & 0.7627 & 0.0145 & 0.046 \\ 0.0395 & 0 & -0.096 & 0.7627 & 0 \\ 0.0263 & -0.0395 & 0.1907 & -0.0158 & 0.5523 \end{bmatrix}$$

и вектор b:

$$\begin{bmatrix} [1.2677 \\ 1.6819 \\ -2.3657 \\ -6.5369 \\ 2.8351] \end{bmatrix}$$

Так же в методе встречной прогонки будем использовать 3-диагональную матрицу:

$$\begin{bmatrix} 0.6444 & 0 & 0 & 0 & 0 \\ -0.0395 & 0.4208 & 0 & 0 & 0 \\ 0 & -0.1184 & 0.7627 & 0.0145 & 0 \\ 0 & 0 & -0.096 & 0.7627 & 0 \\ 0 & 0 & 0 & -0.0158 & 0.5523 \end{bmatrix}$$

Глава 1

Метод Гаусса (выбор главного элемента по строке)

1.1 Постановка задачи.

Пусть дана система

$$Ax = b.$$

Нам необходимо:

1. найти решение системы указанным методом;
2. вычислить вектор невязки;
3. вычислить определитель матрицы системы;
4. найти матрицу, обратную к матрице системы;
5. найти число обусловленности матрицы системы.

1.2 Алгоритм.

Выбор главного элемента по строке.

На каждом k-ом шаге прямого хода в качестве ведущего выбирается максимальный по модулю элемент в неприведенной части строки. После этого столбец расширенной матрицы соотвествующий главному элементу переставляется с k-ым столбцом и производится перенумерация коэффициентов при неизвестных. Для этого заводим массив индексов, в котором будем производить перестановки. Запишем систему в координатном виде:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2, \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n. \end{cases} \qquad (1)$$

Предположим, что $a_{11} \neq 0$. Разделим первое уравнение системы (1) на a_{11} и получим:

$$x_1 + c_{11}x_2 + \ldots + c_{1n}x_n = q_1 \quad (2),$$

где $c_{1j} = \dfrac{a_{1j}}{a_{11}}$, $j = \overline{2,n}$, $q_1 = \dfrac{b_1}{a_{11}}$. С помощью уравнения (2) исключим x_1 из всех остальных уравнений, начиная со 2-го. Таким образом, первое уравнение не поменяется, а все остальные примут вид

$$\begin{cases} a^1_{22}x_2 + a^1_{23}x_3 + \ldots + a^1_{2n}x_n = b^1_2, \\ \vdots \\ a^1_{n2}x_2 + a^1_{n3}x_3 + \ldots + a^1_{nn}x_n = b^1_n, \end{cases} \qquad (3)$$

где $a^1_{ij} = a_{ij} - c_{1j}a_{i1}$, $b^1_i = b_i - q_1a_{i1}$, $i,j = \overline{2,n}$. Мы завершили первый шаг прямого хода.

• Элемент a_{11} называется *ведущим* элементом шага.

Далее считаем, что a_{22} ведущий элемент. Аналогично делим на него и исключаем x_2 и так далее до x_{nn}. Окончательно приходим к системе с верхней треугольной матрицей следующего вида

$$\begin{cases} x_1 + c_{12}x_2 + \ldots + c_{1n} = q_1, \\ x_2 + \ldots + c_{2n} = q_2, \\ \vdots \\ x_n = q_n. \end{cases} \qquad (4)$$

Прямой ход.

Для того, чтобы получить общие формулы вычисления матрицы, введем следующие обозначения:

1. $a^0_{kj} = a_{kj}, k,j = \overline{1,n}$ - исходные элементы матрицы;

2. $b^0_k = b_k$, $k = \overline{1,n}$ - компоненты вектора правых частей исходной матрицы.

Тогда

$$c_{kj} = \frac{a^{k-1}_{kj}}{a^{k-1}_{kk}},\ j = \overline{k+1,n},\ k = \overline{1,n-1},$$

$$a^k_{ij} = a^{k-1}_{ij} - a^{k-1}_{ik}c_{kj},\ i,j = \overline{k+1,n},\ k = \overline{1,n-1},$$

$$q_k = \frac{b_k^{k-1}}{a_{kk}^{k-1}},\ k = \overline{1,n},$$

$$b_i^k = b_i^{k-1} - a_{ik}^{k-1} q_k,\ i = \overline{k+1,n},\ k = \overline{1,n-1}.$$

Метод Гаусса завершен, числа a_{kk}^{k-1}, $k = \overline{1,n}$ - ведущие элементы.

Обратный ход

Обратный ход состоит в последовательном нахождении неизвестных

$$x_i = \frac{q_i - \sum_{j=i+1}^{n} c_{ij} x_j}{a_{ii}},\ i = \overline{n-1,1}.$$

$$x_n = \frac{q_n}{a_{nn}}.$$

Вычисление вектора невязки

После применения обратного хода мы получаем вектор решений x^*. Вектор невязки определяется формулой

$$r = Ax^* - b,$$

где A и b - матрица и вектор исходной системы.

Вычисление определителя

$$|A| = (-1)^m \cdot a_{11}^0 \cdot a_{22}^1 \cdot \ldots \cdot a_{nn}^{n-1},$$

где m - количество перестановок, осуществленных при прямом ходе метода Гаусса.

Вычисление обратной матрицы

Задача нахождения матрицы обратной матрице A эквивалентна задаче решения матричного уравнения

$$AX = E,$$

где $X = A^{-1}$ - искомая матрица. Если обозначить $x^{(1)}, \ldots, x^{(n)}$ - столбцы матрицы X, то эта матрица может быть найдена по столбцам решения системы вида

$$Ax^{(j)} = \delta^{(j)},\ \delta^{(j)} = (\delta_{1j}, \ldots, \delta_{nj})^T,\ x^{(j)} = (x_{1j}, \ldots, x_{nj})^T.\ j = \overline{1, n} \qquad (1)$$

Решение всех n систем доставит нам все столбцы обратной матрицы. Таким образом, задача нахождения обратной матрицы сводится к применению метода Гаусса n раз к системе вида (1).

Вычисление числа обусловленности

Для вычисления числа обусловленности воспользуемся формулой, определяющей его,

$$\nu(A) = ||A|| \cdot ||A^{-1}||.$$

При решении будет вычислять его, используя сферическую норму матрицы.

1.3 Реализация.

```
import math
import numpy as np
def gaussian(matrix_, inhomogeneity):
    matrix = matrix_.copy()
    insertions = 0
    indexes = [i for i in range (matrix.shape[0])]
    unit_matrix = np.eye(matrix.shape[0])


    for k in range(matrix.shape[0]):
        leading_column = k
        for i in range(k, matrix.shape[0]):
            if math.fabs(matrix[k][i]) > math.fabs(matrix[leading_column][k]):
                leading_column = i
        for j in range(k, matrix.shape[1]):
            matrix[j][leading_column], matrix[j][k] =  matrix[j][k], matrix[j][
               leading_column]
        insertions+=1
        indexes[leading_column], indexes[k] = indexes[k], indexes[
           leading_column]

        q = inhomogeneity[k] / matrix[k][k]
        for j in range(matrix.shape[0] - 1, k - 1, -1):
            c = matrix[k][j] / matrix[k][k]
            for i in range(matrix.shape[0] - 1, k, -1):
                matrix[i][j] = matrix[i][j] - matrix[i][k]*c
```

```
                if j == matrix.shape[0] - 1:
                    inhomogeneity[i] = inhomogeneity[i] - matrix[i][k]*q

    results_with_insertions = np.zeros(matrix.shape[0])
    for i in range(matrix.shape[0]-1, -1, -1):
        summary = 0
        for j in range(i+1, matrix.shape[0]):
            summary += matrix[i][j]*results_with_insertions[j]
        results_with_insertions[i] = (inhomogeneity[i] - summary) / matrix[i][i
            ]

    results = np.zeros(matrix.shape[0])
    for i in range(matrix.shape[0]):
        results[indexes[i]] = results_with_insertions[i]

    discrepancy_vector = np.zeros(matrix.shape[0])
    for i in range(matrix.shape[0]):
        for j in range(matrix.shape[0]):
            discrepancy_vector[i] += matrix[i][j]*results[j]; # r += Ax
        discrepancy_vector[i] -= inhomogeneity[i] # r -= b

    determinant = (-1)**insertions
    for k in range(matrix.shape[0]):
        determinant *= matrix[k][k]
    return results, discrepancy_vector, determinant;
def inverse_A(A):
    n = 5
    E = np.eye(n)
    A_inv = np.empty((0, 5))
    for i in range(n):
        A_inv = np.vstack((A_inv, gaussian(A, E[i])[0]))
        # A_inv.append(gaussian(A, E[i])[0])
    return A_inv.T
    # print(*A_inv, sep='\n')
def condition_N(A):
    nu = np.linalg.norm(A, 2) * np.linalg.norm(inverse_A(A), 2)
    return nu
```

1.4 Выводы по полученным результатам.

Решение:

$$\begin{bmatrix} 0.99821505 \\ 1.99986528 \\ -2.99975971 \\ -9.00000843 \\ 6.00705353 \end{bmatrix}$$

Вектор невязки:

$$\begin{bmatrix} 2.22044605 \times 10^{-16} \\ 0 \\ 0 \\ -8.88178420 \times 10^{-16} \\ 0 \end{bmatrix}$$

Норма вектора невзяки (Евклидова норма)

$$8.950 \times 10^{-16}$$

Определитель:

$$-0.0833942433132514$$

Обратная матрица:

$$A^{-1} = \begin{bmatrix} 1.58764 & 0.07502 & 0.47008 & -0.26014 & -0.60988 \\ 0.1376 & 2.38888 & 0.10145 & 0.22436 & -0.17136 \\ -0.00045 & 0.36621 & 1.34465 & 0.01034 & -0.12927 \\ -0.08228 & 0.04221 & 0.1449 & 1.32591 & 0.01531 \\ -0.06796 & 0.04204 & -0.47527 & 0.0628 & 1.87247 \end{bmatrix}$$

Проверка обратной матрицы:

$$A \cdot A^{-1} - E =$$

$$\begin{bmatrix} 0 & -2.6 \times 10^{-18} & 2.5 \times 10^{-17} & -5.0 \times 10^{-17} & 6.8 \times 10^{-17} \\ 1.5 \times 10^{-17} & 0 & -9.4 \times 10^{-18} & -1.0 \times 10^{-17} & -9.8 \times 10^{-18} \\ -3.1 \times 10^{-18} & 2.0 \times 10^{-17} & 0 & -4.3 \times 10^{-18} & 7.2 \times 10^{-18} \\ 8.4 \times 10^{-18} & -6.9 \times 10^{-19} & 1.2 \times 10^{-17} & 0 & -2.0 \times 10^{-18} \\ -2.2 \times 10^{-18} & 3.3 \times 10^{-18} & 4.9 \times 10^{-17} & 9.1 \times 10^{-18} & 0 \end{bmatrix}$$

Число обусловленности:

$$2.260282522619793$$

Вывод.

Решение было получено с точностью 10^{-16}, что приемлемо для выбранного метода. Так же стоит отметить высокую точность нахождения обратной матрицы и малое значение вектора невязки.

Глава 2

Метод встречной прогонки

2.1 Постановка задачи.

Необходимо найти решение системы линейных алгебраических уравнений вида $Ax = b$, где A - трёхдиагональная, квадратная матрица n-ого порядка, и b – столбцы размеров n.

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & \cdots & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & \cdots & \cdots & 0 \\ 0 & a_{32} & a_{33} & \cdots & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & a_{n-1n-2} & a_{n-1n-1} & a_{n-1n} \\ 0 & \cdots & \cdots & 0 & a_{nn-1} & a_{nn} \end{bmatrix}, x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Нам необходимо:

1. найти решение системы методом встречной прогонки;

2. вычислить вектор невязки, оценить его величину;

2.2 Алгоритм.

Элементы правой части и матрицы системы обозначим и пронумеруем по-другому, СЛАУ будет иметь вид ($N = n - 1$)

$$A = \begin{bmatrix} c_0 & -b_0 & 0 & \cdots & \cdots & 0 \\ -a_1 & c_1 & -b_1 & \cdots & \cdots & 0 \\ 0 & -a_2 & c_2 & \cdots & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & -a_{N-1} & c_{N-1} & -b_{N-1} \\ 0 & \cdots & \cdots & 0 & -a_N & c_N \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_N \end{bmatrix}$$

m здесь - номер уравнения, на котором "встречаются"две ветви прямого хода - "сверху"и "снизу".

В приведенных обозначениях во встречной прогонке сначала выполняют её прямой ход - вычисляют коэффициенты.

Сверху:

$$\alpha_1 = b_0/c_0,$$
$$\beta_1 = f_0/c_0,$$
$$\alpha_{i+1} = b_i/(c_i - a_i\alpha_i), \quad i = 1, 2, \cdots, m-1,$$
$$\beta_{i+1} = (f_i + a_i\beta_i)/(c_i - a_i\alpha_i), \quad i = 1, 2, \cdots, m-1.$$

Снизу:

$$\xi_N = a_N/c_N,$$
$$\eta_N = f_N/c_N,$$
$$\xi_i = a_i/(c_i - b_i\xi_{i+1}), \quad i = N-1, N-2, \cdots,$$
$$\eta_i = (f_i + b_i\eta_{i+1})/(c_i - b_i\xi_{i+1}), \quad i = N-1, N-2, \cdots, m.$$

Обратный ход:

$$y_m = (\eta_m + \xi_m\beta_m)/(1 - \xi_m\alpha_m),$$
$$y_{m-1} = (\beta_m + \alpha_m\eta_m)/(1 - \xi_m\alpha_m),$$
$$y_i = \alpha_{i+1}y_{i+1} + \beta_{i+1}, \quad i = m-2, \cdots, 1, 0,$$
$$y_{i+1} = \xi_{i+1}y_i + \eta_{i+1}, \quad i = m, m+1, \cdots, N-1.$$

2.3 Реализация.

Для хранения коэффицентов α, η и β, ξ завели два массива *alpha* и *beta*. Так же задали точность $eps = 10^{-20}$, для которой можно привести аналогию с машинным нулем. При каждом вычислении коэффицента, сравниваем с *eps*, если результат меньше заданной точности, то полностью обнуляем значение.

```
import numpy as np
from decimal import Decimal
eps = 10e-20
def sweepMet(A, b):
    n = len(A)
    m = (n + 1) // 2
    diagonalC = np.zeros(n)
    diagonalA = np.zeros(n)
```

```
    diagonalB = np.zeros(n)
    # np.set_printoptions(precision=25, suppress=True)
    for i in range(n):
        diagonalC[i] = A[i][i]

    for i in range(n - 1):
        diagonalA[i + 1] = -A[i + 1][i]
        diagonalB[i] = -A[i][i + 1]
    alpha = np.zeros(n)
    beta = np.zeros(n)

    alpha[0] = diagonalB[0] / diagonalC[0]
    beta[0] = b[0] / diagonalC[0]

    for i in range(1, m):
        alpha[i] = diagonalB[i] / (diagonalC[i] - diagonalA[i] * alpha[i-1])
        beta[i] = (b[i] + diagonalA[i] * beta[i - 1]) / (diagonalC[i] -
           diagonalA[i] * alpha[i-1])
        # Check for machine zero
        alpha[i] = alpha[i] if abs(alpha[i]) > eps else 0.0
        beta[i] = beta[i] if abs(beta[i]) > eps else 0.0

    alpha[n - 1] = diagonalA[n - 1] / diagonalC[n - 1]
    beta[n - 1] = b[n - 1] / diagonalC[n - 1]

    for i in range(n - 2, m - 1, -1):
        alpha[i] = diagonalA[i] / (diagonalC[i] - diagonalB[i] * alpha[i + 1])
        beta[i] = (beta[i + 1] * diagonalB[i] + b[i]) / (diagonalC[i] -
           diagonalB[i] * alpha[i + 1])
        alpha[i] = alpha[i] if abs(alpha[i]) > eps else 0.0
        beta[i] = beta[i] if abs(beta[i]) > eps else 0.0
    massX = np.zeros(n)
    massX[m - 1] = ((b[m - 1] + diagonalB[m - 1]*beta[m] + diagonalA[m - 1]*
       beta[m - 2])/
    (diagonalC[m - 1] - diagonalA[m - 1]*alpha[m - 2] - diagonalB[m - 1]*alpha[
       m]))

    for i in range(1, m):
        massX[m - i - 1] = alpha[m - i - 1] * massX[m - i] + beta[m - i - 1]
        massX[m + i - 1] = alpha[m + i - 1] * massX[m + i - 2] + beta[m + i -
           1]
    checker = np.zeros(n)
    for i in range(0, n):
        checker[i] = diagonalC[i] - alpha[i]* diagonalA[i]
    return massX
```

2.4 Выводы по полученным результатам.

Коэффициенты Alpha:

$$\begin{bmatrix} -0 & 0 & -0.019 & 0.13 & 0.03 \end{bmatrix}$$

Коэффициенты Beta:

$$\begin{bmatrix} 1.97 & 4.18 & -2.45 & -8.57 & 5.13 \end{bmatrix}$$

Вектор решений X:

$$\begin{bmatrix} 1.97 & 4.18 & -2.28 & -8.86 & 4.88 \end{bmatrix}$$

Вектор оценок (для обоснования корректности):

$$\begin{bmatrix} 0.64 & 0.42 & 0.76 & 0.75 & 0.55 \end{bmatrix}$$

Вектор невязки:

$$\begin{bmatrix} 9.29 \times 10^{-17} & -1.93 \times 10^{-16} & 2.89 \times 10^{-16} & 5.86 \times 10^{-16} & 4.53 \times 10^{-16} \end{bmatrix}$$

Норма невязки:

$$8.23 \times 10^{-16}$$

Вывод.

Решение было получено с точностью 10^{-16}, что приемлемо для выбранного метода. Так же стоит отметить малое значение вектора невязки, что подтверждает высокую точность метода. Для выполнения встречной прогонки в трёхдиагональной СЛАУ из n уравнений с n неизвестными требуется: $2n + 2$ делений, $3n - 2$ сложений/вычитаний, $3n - 2$ умножений, что свидетельствует о линейной сложности алгоритма. Так же было проверено достаточное условие корректности метода и построен вектор оценок.

Глава 3

Метод Якоби

3.1 Постановка задачи.

Пусть дана система

$$Ax = b.$$

Предполагается, что $\det A = |A| \neq 0$. Тогда решение системы существует и оно единственно. Нам необходимо:

1. найти решение системы методом Якоби с точностью 10^{-5};
2. вычислить вектор невязки, оценить его норму;
3. определить количество итераций для достижения необходимой точности;
4. обосновать сходимость метода;

3.2 Алгоритм.

Метод Якоби состоит в последовательном вычислении значения вектора x^k, до тех пор, пока не выполнится условие:

$$\left\|x^{k+1} - x^k\right\| \leq \varepsilon$$

Приведём $Ax = b$ к каноническому виду $x = Bx + g$, и g найдём по методу Якоби, как:

$$B = \begin{bmatrix} 0 & -\frac{a_{12}}{a_{11}} & -\frac{a_{12}}{a_{11}} & \cdots & -\frac{a_{1n}}{a_{11}} \\ -\frac{a_{21}}{a_{22}} & 0 & -\frac{a_{13}}{a_{22}} & \cdots & \\ \vdots & & & \ddots & \vdots \\ -\frac{a_{12}}{a_{nn}} & -\frac{a_{n2}}{a_{nn}} & -\frac{a_{n2}}{a_{nn}} & \cdots & 0 \end{bmatrix}, g = \begin{bmatrix} \frac{b_{11}}{a_{11}} \\ \frac{b_{22}}{a_{22}} \\ \vdots \\ \frac{b_{nn}}{a_{nn}} \end{bmatrix}$$

Далее будем вычислять следующее значение , как: $x^{k+1} = Bx^k + g$, а коли-

чество итераций можно оценить по формуле

$$\left\|x^{k+1}-x^{k}\right\| \leq \frac{||B|^{k+1}}{1-\|B\|}\|g\| \leq \varepsilon \quad \Rightarrow \quad k=\log_{\|B\|}\left(\frac{\varepsilon *(1-||B||}{\|g\|}\right)-1$$

Для обоснования сходимости метода воспользуемся достаточным условием:

$$\|B\|<1$$

3.3 Реализация.

```
import numpy as np

epsilon = 10^(-5)

def Jacobi_method(A, b):
    B = np.zeros((5, 5))
    counter = 0
    for i in range(5):
        for j in range(5):
            if i != j:
                B[i][j] = -A[i][j]/A[i][i]
    print(B)
    g = np.zeros(5).transpose()
    for i in range(5):
        g[i] = b[i]/A[i][i]
    print(g)

    x_k = b
    x_k_1 = b.dot(1.2)

    while(np.linalg.norm(x_k_1 - x_k, ord=1) > epsilon) :
        print(f"\n{np.linalg.norm(x_k_1 - x_k, ord=1)} > {epsilon}\n")
        print(f"------------------iteration {counter}------------------\n")

        x_k  = x_k_1
        print(x_k)
        x_k_1 = B.dot(x_k) +g
        print(x_k_1)
        #if counter == 2 : break
        counter+=1

    print(f"\n-------------- Result ------------------\n")
    result = A.dot(x_k_1) - b
     k = log(epsilon*(1-np.linalg.norm(B))/np.linalg.norm(g, 1), np.linalg.norm
        (B))-1
    print(f"Solution :{x_k_1}")
    print(f"Apriory iterations:{counter}")
    print(f"Iterations:{counter}")
    print(f"Discrepancy vector:{result}")
    print(f"Discrepancy norm:{np.linalg.norm(result)}")
```

```
    print(f"B norm:{np.linalg.norm(B)}")

    return B
```

3.4 Выводы по полученным результатам.

Решение:

$$\begin{bmatrix} 0.99821559 \\ 1.99986555 \\ -2.99975951 \\ -9.00000846 \\ 6.00705329 \end{bmatrix}$$

Расчетное количество итераций: 35.

Количество итераций: 11.

Вектор невязки:

$$\begin{bmatrix} 2.66 \times 10^{-7} \\ 8.91 \times 10^{-8} \\ 1.14 \times 10^{-7} \\ -2.34 \times 10^{-8} \\ -8.99 \times 10^{-8} \end{bmatrix}$$

Норма невязки: 3.17×10^{-7}.

Норма матрицы B: $0.647 < 1$.

Вывод.

Точность решения и скорость сходимости метода Якоби зависят от заданного $\varepsilon = 10^{-5}$. Метод сходится, так как выполняется достаточное условие: норма полученной матрицы B меньше единицы. Норма невязки сильно превосходит нормы невязок точных методов, таких как: метод Гаусса и метод прогонки. Отсюда можно сделать вывод, что норма невязки напрямую зависит от заданной точности ε. Так же стоит отметить, что реальное количество итераций оказалось намного меньше предсказанного. На это расхождение главным образом влияет выбор матрицы B и вектора g.

Глава 4

Метод градиентного спуска

4.1 Постановка задачи.

Пусть дана система

$$Ax = b.$$

Предполагается, что $\det A = |A| \neq 0$. Тогда решение системы существует и оно единственно. Нам необходимо:

1. найти решение системы методом градиентного спуска с точностью 10^{-5};
2. вычислить вектор невязки, оценить его норму;
3. определить количество итераций для достижения необходимой точности;
4. обосновать сходимость метода;

4.2 Алгоритм.

В методе градиентного спуска нахождение решения системы $Ax = b$ связано с задачей минимизации квадратичного функционала $F(x) = (Ax, x) - 2(b, x)$. Метод градиентного спуска состоит в последовательном вычислении вектора x^k:

Положим начальное приближение $x^0 = b$. Далее будем вычислять x^k по итерационной формуле $x^k = x^{k-1} - \tau_k r^{k-1}$, где $r^{k-1} = Ax^{k-1} - b, \tau_k = \frac{\left(r^{k-1}, r^{k-1}\right)}{(Ar^{k-1}, r^{k-1})}$, до тех пор, пока не выполнится условие: $\left\|x^{k+1} - x^k\right\| \leq \varepsilon$.

Для сходимости достаточно, чтобы матрица A была симметричной, для этого домножим матрицу A и вектор b на матрицу A^T слева. Решение новой системы $A^T Ax = A^T b$ будет совпадать с решением системы $Ax = b$.

4.3 Реализация.

```
    import numpy as np

def gradient_descent(A, b, tolerance):
    n = len(b)
    E = np.identity(n)
    At = A.transpose() =
    A = np.dot(At, A)
    b = np.dot(At, b)
    xk = b
    x = np.zeros(n)
    k = 0
    while True:
        rk = np.dot(A, xk) - b
        x = xk - np.dot(rk, np.dot(rk, rk) / np.dot(np.dot(A, rk), rk))
        k += 1
        if abs(np.linalg.norm(x, 1) - np.linalg.norm(xk, 1)) < tolerance:
            break
        xk = x
    r = np.dot(A, x) - b
    rnorm = np.linalg.norm(r, 1)
```

4.4 Выводы по полученным результатам.

Решение x:

$$\begin{bmatrix} 0.99825763 & 1.99994978 & -2.99974174 & -8.99999341 & 6.0070143 \end{bmatrix}$$

Количество итераций: 24

Вектор невязки:

$$\begin{bmatrix} 1.05 \times 10^{-5} & 1.38 \times 10^{-5} & -6.10 \times 10^{-6} & 8.71 \times 10^{-6} & -6.83 \times 10^{-6} \end{bmatrix}$$

Норма невязки: 4.60×10^{-5}

Вывод.

Скорость сходимости метода градиентного спуска зависит от заданного $\varepsilon = 10^{-5}$. Метод сходится, так как выполняется достаточное условие: матрица $A^T A$ – симметричная. Норма невязки сильно превосходит нормы невязок точных методов, таких как: метод Гаусса и метод прогонки. Отсюда можно сделать вывод, что норма невязки напрямую зависит от заданной точности ε. Так же стоит отметить более медленную сходимость по сравнению с методом Якоби.

Глава 5

Сравнительный анализ

	Сложность	Память	Норма вектора невязки	Количество итераций
Метод Гаусса	$O(n^3)$	$O(n^2)$	8.95×10^{-16}	
Встречная прогонка	$8n$	$3n$	8.23×10^{-16}	
Метод Якоби	$O(kn^2)$	$O(n^2)$	3.17×10^{-7}	11
Градиентный спуск	$O(kn^2)$	$O(n^2)$	4.60×10^{-5}	24

Сравнивая точные методы: метод Гаусса и метод встречной прогонки, можно сказать, что встречная прогонка имеет заметные преимущества как в точности вычислений, так и в памяти и скорости работы. Метод прогонки использует 3-диагональную структуру матрицы по максимум, тем самым показывая отличные результаты вычислений. Но вместе с этим, стоит сказать, что метод прогонки решает только узкий круг задач, в чем проигрывает методу Гаусса.

Сравнивая итерационные методы: метод Якоби и метод градиентного спуска, можем наблюдать, что метод Якоби позволил найти решение намного быстрее, чем метод градиентного спуска (11 итераций против 24). А так же сравнивая нормы векторов невязки, можем удтверждать, что найденное решение методом Якоби имеет лучшую точность. Делаем вывод, что точность (норма вектора невязки) напрямую зависит от количества итераций. Так же стоит отметить особенность применения метода Якоби - $\|B\| < 1$, что в общем-то сужает круг решаемых задач.