Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования Московский государственный технический университет имени Н.Э. Баумана

Лабораторная работа «Реализация методов Банча-Парлетта и Банча-Кауфмана для решения системы $Ax = b$» по курсу «Вычислительные методы линейной алгебры»

Студент группы ИУ9-72 Белогуров А.А.

Преподаватель Голубков А.Ю.

Москва, 2017

Содержание

1 Постановка задачи

Дано: $Ax = b$, где $A \in M_n(\mathbb{C})$ является *самосопряженной* матрицей (т.е. $A = A^*$) и $x, b \in \mathbb{C}^n$. В вещественном случае понятие *самосопряженной* эквивалентно понятию *симметрической* матрицы (т.е. $A = A^t$).

Реализовать методы Банча-Парлетта и Банча-Кауфмана для решения системы $Ax = b$ и построение с их помощью разложения $PAP^t = LTL$.

2 Теоретические сведения

Алгоритмы Банча-Парлетта и Банча-Кауфмана предполагают построение на базе матрицы $A = A^* = A^{(0)}$ матриц $A^{(1)}, ..., A^{(m)}$, где последняя матрица $T = A^{(m)}$ является трёхдиагональной матрицей блочно-диагональной структуры $T = diag(T_1, ..., T_m)$, где каждый блок T_i имеет размер $n_i \times n_i \in \{1, 2\}$. Матрица $A^{(k)}$ может быть представлена как блочно-диагональная матрица вида $A^{(k)} = diag(T_1, ..., T_k, \widetilde{A}^{(k)})$, где $\widetilde{A}^{(k)}$ - ведущая подматрица размера $(n - m_k) \times (n - m_k)$, $m_k = n_1 + ... + n_k$, $\widetilde{A}^{(k)} = (a_{ij}^{(k)})_{i,j=m_k+1}^{n}$. На k-ом шаге переход $A^{(k-1)} \to A^{(k)}$ выполняется по следующему правилу: в соответствии с некоторой стратегией выбирается матрица перестановка $\widetilde{P}_{(k)}$ и размер n_k нового невырожденного диагонального блока T_k, где

$$\widetilde{P}_k \widetilde{A}^{(k-1)} \widetilde{P}_k^t = \begin{pmatrix} T_k & B_k^* \\ B_k & C_k \end{pmatrix}$$

и блок $C_k = C_k^*$ имеет размер $(n - m_k) \times (n - m_k)$, $m_k = m_{k-1} + n_k$. Заетм, полагая

$$\widetilde{L}_k = \begin{pmatrix} E_{n_k} & 0 \\ -B_k T_k^{-1} & E_{n-m_k} \end{pmatrix},$$

мы выполняем блочное преобразование

$$\widetilde{L}_k \widetilde{P}_k \widetilde{A}^{(k-1)} \widetilde{P}_k^t \widetilde{L}_k^* = \begin{pmatrix} T_k & 0 \\ 0 & C_k - B_k T_k^{-1} B_k^* \end{pmatrix} = \begin{pmatrix} T_k & 0 \\ 0 & \widetilde{A}^{(k)} \end{pmatrix}.$$

Иначе говоря, осуществляемая процедура может быть записана как

$$A^{(k)} = diag(T_1, ..., T_k, \widetilde{A}^{(k)}) = L_k P_k A^{(k-1)} P_k^t L_k^*,$$

где $P_k = diag(E_{m_{k-1}}, \widetilde{P}_k)$ и $L_k = diag(E_{m_{k-1}}, \widetilde{L}_k)$. Результатом выполнения является представление $A = PLTL^*P^t$ [1].

Далее необходимо рассмотреть стратегии выбора $\widetilde{P}_k$ и n_k на k-ом шаге в алгоритмах Банча-Парлетта и Банча-Кауфмана.

Банч-Парлетт:

1. действуя в рамках ведущей подматрицы $\widetilde{A}^{(k-1)}$, находим

$$\mu_0(k) = \max_{i,j=m_{k-1}+1,\ldots,n} |a_{ij}^{(k-1)}|, \quad \mu_1(k) = \max_{i=m_{k-1}+1,\ldots,n} |a_{ii}^{(k-1)}|$$

2. если $\mu_1(k) \geq \alpha\mu_0(k)$, где выбор параметра $0 < \alpha < 1$ оговаривается дополнительно, тогда мы полагаем $n_k = 1$ и подбираем $\widetilde{P}_k$ таким образом, что

$$|(\widetilde{P}_k \widetilde{A}^{(k-1)} \widetilde{P}_k^t)_{m_{k-1}+1 m_{k-1}+1}| = \mu_1(k),$$

т.е. если $a_{i_k i_k}^{(k-1)}$ - элемент диагонали матрицы $\widetilde{A}^{(k-1)}$ с наибольшим модулем, то $P_k = P_{m_{k-1}+1 i_k}$; в противном случае $n_k = 2$ и матрица $\widetilde{P}_k$ подбирается с тем, чтобы

$$|(\widetilde{P}_k \widetilde{A}^{(k-1)} \widetilde{P}_k^t)_{m_{k-1}+2 m_{k-1}+1}| = \mu_0(k),$$

точнее в данном случае наибольший по модулю элемент $a_{i_k j_k}^{(k-1)}$ матрицы $\widetilde{A}^{(k-1)}$ лежит вне диагонали (можно считать, что $j_k < i_k$) и для того, чтобы его перевести на позицию $(m_{k-1}+2, m_{k-1}+1)$ достаточно взять $P_k = P_{m_{k-1}+2 i_k} P_{m_{k-1}+1 j_k}$.

Банч-Кауфман:

1. будем считать, что элемент $a_{m_{k-1}+1 m_{k-1}+1}^{(k-1)}$ матрицы $\widetilde{A}^{(k-1)}$ имеет наибольший модуль среди всех её диагональных элементов (при необходимости этого можно добиться сопряжением (подобием) с подходящей матрицей перестановок);

2. находим элемент λ с наибольшим модулем в первом столбце подматрицы $\widetilde{A}^{(k-1)}$, т.е. определим i_k, $m_{k-1} + 1 \leq i_k \leq n$,

$$\lambda = |a_{i_k m_{k-1}+1}^{(k-1)}| = \max_{i=m_{k-1}+1,\ldots,n} |a_{i m_{k-1}+1}^{(k-1)}|$$

(для простоты предполагаем, что матрица невырождена и $\lambda > 0$);

3. если $|a^{(k-1)}_{m_{k-1}+1m_{k-1}+1}| \geq \alpha\lambda$, то полагаем $n_k = 1$ и $P_k = E$; в противном случае мы находим j_k, $m_{k-1}+1 \leq j_k \leq n$,

$$\sigma = |a^{(k-1)}_{j_k i_k}| = \max_{j=m_{k-1}+1,\ldots,n,\quad j\neq i_k} |a^{(k-1)}_{j i_k}|$$

(наибольший по модулю внедиагональный элемент i_k-столбца матрицы $\widetilde{A}^{(k-1)}$), и в ситуации $\sigma|a^{(k-1)}_{m_{k-1}+1m_{k-1}+1}| \geq \alpha\lambda^2$ вновь полагаем $n_k = 1$ и $P_k = E$, а иначе

- при $|a^{(k-1)}_{i_k i_k}| \geq \alpha\sigma$ положим $n_k = 1$ и возьмём $P_k = P_{m_{(k-1)}+1 i_k}$, обеспечив тем самым перестановку первого и i_k-го столбцов матрицы $\widetilde{A}^{(k-1)}$ и вместе с тем перемещение её i_k-го диагонального элемента на первую позицию:

$$(\widetilde{P}_k \widetilde{A}^{(k-1)} \widetilde{P}_k^t)_{m_{k-1}+1m_{k-1}+1} = a^{(k-1)}_{i_k i_k};$$

- при $|a^{(k-1)}_{i_k i_k}| < \alpha\sigma$ положим $n_k = 2$ и возьмём $P_k = P_{m_{k-1}+2 i_k} P_{m_{k-1}+1 j_k}$, что даёт

$$(\widetilde{P}_k \widetilde{A}^{(k-1)} \widetilde{P}_k^t)_{m_{k-1}+2m_{k-1}+1} = a^{(k-1)}_{i_k j_k} = a^{(k-1)}_{j_k i_k};$$

Исходя из соображений уменьшения оценки роста модулей в L-составляющей получаемых разложений, в обоих алгоритмах предполагается использовать $\alpha = (1+\sqrt{17})/8$.

Построив разложение $A = PLTL^*P^t$ или $P^tAP = LTL^*$ можно свести линейную систему $Ax = b$ к системам

$$\begin{cases} Lz &= P^t b \\ Tw &= z \\ L^* &= w \\ x &= Py \end{cases} \quad \Leftrightarrow \quad \begin{cases} z &= L^{-1}P^t b \\ w &= T^{-1}z \\ y &= (L^*)^{-1}w \\ x &= Py \end{cases}$$

3 Практическая реализация

Программа написана на языке программирования Python3 с использованием библиотек *numpy, math* для работы с матрицами и *argparse* для работы с аргументами командной строки. Далее преполагается, что все матрицы осуществляют операции только с вещественными числами.

На вход подаётся имя метода (в данном случае *'Bunch-Parlett'* или *'Bunch-Kaufman'*) и два файла текстового типа с матрицами. Первый из них *'matrix_ a'* - содержит симметричную матрицу A, элементы которой разделены пробелами, а каждая строка матрицы записывается в новой строке файла, и второй *'matrix_ b'* - содержит матрицу b. Также можно посмотреть эти параметры при запуске скрипта с флагом *-h* (Листинг 1).

```
~ $ python MatrixSolver.py -h
usage: MatrixSolver.py [-h] method_name matrix_a matrix_b

positional arguments:
  method_name  Name of LTL-method: 'Bunch-Kaufman' or
   ↪  'Bunch-Parlett'
  matrix_a     Name of file where exists symmetry matrix A
  matrix_b     Name of file where exists matrix B

optional arguments:
  -h, --help   show this help message and exit
```

Выше были описаны два метода для выбора $\widetilde{P}_k$ и n_k на k-ом шаге алгоритма, они соответственны представлены в методах *'bucnh-parlett()'* (Листинг 2) и *'bucnh-kaufman()'* (Листинг 3).

```
def bunch_parlett(k, matrix):
    # matrix_size - размерность матрицы
    matrix_size = len(matrix)
```

```
4
5       # max_element - max элемент по модулю матрицы A
6       # idx_max_element - его индекс
7       (max_element, idx_max_element) = (None, None)
8
9       # max_diag_element - max диагональый элемент по модулю матрицы
         ↪  A
10      # idx_max_diag_element - его индекс
11      (max_diag_element, idx_max_diag_element) = (None, None)
12
13      # 1 шаг - находим элементы, определенные выше
14      for j in range(matrix_size):
15          for i in range(j, matrix_size):
16              if max_element is None or max_element < abs(matrix[i,
                 ↪  j]):
17                  (max_element, idx_max_element) = (abs(matrix[i,
                     ↪  j]), (i, j))
18
19      for i in range(matrix_size):
20          if max_diag_element is None or max_diag_element <
             ↪  abs(matrix[i, i]):
21              (max_diag_element, idx_max_diag_element) =
                 ↪  (abs(matrix[i, i]), (i, i))
22
23      # 2 шаг - опрделеяем размерность блока n[k] и матрицу
         ↪  перастановок P
24      if k > 1:
25          row_index = m[k - 1] - calc_block_size(blocks)
26      else:
27          row_index = m[k - 1]
28
29      if max_diag_element >= max_element * alpha:
30          # n[k] = 1
31          n.append(1)
32          swap_numbers = [(row_index, idx_max_diag_element[0])]
33      else:
34          # n[k] = 2
35          n.append(2)
36          swap_numbers = [(row_index + 1, idx_max_element[0]),
             ↪  (row_index, idx_max_element[1])]
37
38      p = np.eye(len(matrix))
```

```
    while len(swap_numbers) != 0:
        numbers = swap_numbers.pop(0)

        p_new = np.eye(len(matrix))
        swap_columns(p_new, numbers[0], numbers[1])

        p = np.dot(p, p_new)

    return p
```

```
def bunch_kaufman(k, matrix):
    # matrix_size - размерность матрицы
    matrix_size = len(matrix)

    # max_diag_element - max диагональный элемент по модулю матрицы
     ↪  A
    # idx_max_diag_element - его индекс
    (max_diag_element, idx_max_diag_element) = (None, None)

    # 1 шаг - ищем наибольший диагональный элемент,
    # и при необходимости перемещаем его на позицию [0, 0]
    for i in range(matrix_size):
        if max_diag_element is None or max_diag_element <
         ↪  abs(matrix[i, i]):
            (max_diag_element, idx_max_diag_element) =
             ↪  (abs(matrix[i, i]), (i, i))

    if idx_max_diag_element != (0, 0):
        p = np.eye(matrix_size)

        swap_columns(p, 0, idx_max_diag_element[0])

        matrix = np.dot(p, matrix)
        matrix = np.dot(matrix, p.T)

        idx_max_diag_element = (0, 0)

    # 2 шаг - находим наибольший элемент в первом столбце
    (elem_lambda, idx_elem_lambda) = (None, None)

```

```
    for i in range(matrix_size):
        if elem_lambda is None or elem_lambda < abs(matrix[i, 0]):
            (elem_lambda, idx_elem_lambda) = (abs(matrix[i, 0]),
             ↪  (i, 0))

    # 3 шаг - определяем размерность блока n[k] и матрицу
     ↪  перестановок P
    if k > 1:
        row_index = m[k - 1] - calc_block_size(blocks)
    else:
        row_index = m[k - 1]

    if max_diag_element >= alpha * elem_lambda:
        # n[k] = 1
        n.append(1)
        swap_numbers = [(0, 0)]
    else:
        (elem_sigma, idx_elem_sigma) = (None, None)

        for i in range(1, matrix_size):
            if elem_sigma is None or elem_sigma < abs(matrix[i,
             ↪  0]):
                (elem_sigma, idx_elem_sigma) = (abs(matrix[i, 0]),
                 ↪  (i, 0))

        if elem_sigma * max_diag_element >= alpha * (elem_lambda
         ↪  ** 2):
            # n[k] = 1
            n.append(1)
            swap_numbers = [(0, 0)]
        else:
            if max_diag_element >= alpha * elem_sigma:
                # n[k] = 1
                n.append(1)
                swap_numbers = [(row_index,
                 ↪  idx_max_diag_element[0])]
            else:
                # n[k] = 2
                n.append(2)
                swap_numbers = [(row_index + 1,
                 ↪  idx_max_diag_element[0]), (row_index,
                 ↪  idx_max_diag_element[1])]
```

```
62
63      p = np.eye(len(matrix))
64      while len(swap_numbers) != 0:
65          numbers = swap_numbers.pop(0)
66
67          p_new = np.eye(len(matrix))
68          swap_columns(p_new, numbers[0], numbers[1])
69
70          p = np.dot(p, p_new)
71
72      return p
```

Нахождение разложения $PAP^t = LTL^*$ выполняется методом *find_ltl()* (Листинг 4).

```
1   def find_ltl(matrix, method=bunch_parlett):
2       source_matrix_size = len(matrix)
3       source_matrix = matrix
4       source_matrix_l = np.eye(source_matrix_size)
5
6       # k - шаг алгоритма, начинаем с 1
7       k = 1
8
9       while len(matrix) > 1:
10          matrix_p = method(k, matrix)
11
12          # Определяем матрицы перестановок, на основе которых будет
              ↪ строиться конечная
13          new_matrix_p = np.eye(source_matrix_size)
14          new_matrix_p[m[k-1]:source_matrix_size,
              ↪ m[k-1]:source_matrix_size] = matrix_p
15          matrixes_p.append(new_matrix_p)
16
17          if k > 1:
18              source_matrix_l = np.dot(matrixes_p[k-1],
                  ↪ source_matrix_l)
19              source_matrix_l = np.dot(source_matrix_l,
                  ↪ matrixes_p[k-1])
20
```

```
21          matrix = np.dot(matrix_p, matrix)
22          matrix = np.dot(matrix, matrix_p.T)
23
24          m.append(m[k - 1] + n[k])
25
26          matrix_t = get_matrix_t(matrix, n[k])
27          matrix_b = get_matrix_b(matrix, n[k])
28          matrix_l = calculate_matrix_l(matrix, matrix_t, matrix_b,
             ↪  n[k])
29
30          matrix = np.dot(lg.inv(matrix_l), matrix)
31          matrix = np.dot(matrix, lg.inv(matrix_l.T))
32
33          blocks.append(matrix[0:n[k], 0:n[k]])
34
35          matrix = matrix[n[k]:len(matrix), n[k]:len(matrix)]
36
37          size_l = len(source_matrix_l) - len(matrix_l)
38          source_matrix_l[size_l:len(source_matrix_l),
             ↪  size_l:len(source_matrix_l)] = matrix_l
39
40          # переходим на следующую итерацию
41          k += 1
42
43      blocks.append(matrix)
44
45      result_matrix_t = calc_result_diagonal_matrix(blocks)
46      result_matrix_p = calc_result_matrix_p(matrixes_p,
         ↪  source_matrix_size)
47
48      return source_matrix, result_matrix_p, result_matrix_t,
         ↪  source_matrix_l
```

Кроме основного файла есть файл *Util.py* в котором реализованы методы для вычисления побочных матриц на этапе построения разложения $PAP^t = LTL^*$ и вычисление матрицы b системы $Ax = b$.

4 Тестирование

4.1 Матрица 4x4

В качестве первого теста были взяты следующие матрицы A и b:

$$A = \begin{pmatrix} 6 & 12 & 3 & -6 \\ 12 & -8 & -13 & 4 \\ 3 & -13 & -7 & 1 \\ -6 & 4 & 1 & 6 \end{pmatrix}, \quad b = \begin{pmatrix} 2 \\ 4 \\ 10 \\ -5 \end{pmatrix}.$$

Методом Банча-Парлетта были найдены матрицы разложения:

$$P = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad T = \begin{pmatrix} -8 & -13 & 0 & 0 \\ -13 & -7 & 0 & 0 \\ 0 & 0 & 5.8584 & 0 \\ 0 & 0 & 0 & -2.3202 \end{pmatrix},$$

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0.1327 & -0.3893 & 1 & 0 \\ 0.3982 & -1.1681 & -1.0967 & 1 \end{pmatrix}.$$

Аналогично для метода Банча-Кауфмана:

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad T = \begin{pmatrix} -8 & 12 & 0 & 0 \\ 12 & 6 & 0 & 0 \\ 0 & 0 & 2.7812 & 0 \\ 0 & 0 & 0 & -2.8764 \end{pmatrix},$$

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0.5938 & -0.6875 & 1 & 0 \\ -0.5 & 0 & -1.9775 & 1 \end{pmatrix}.$$

Соответственно была найдена неизвестная матрица x, которая для каждого из методов оказалась одинаковой:

$$x = \begin{pmatrix} -4.4361 \\ 1.5469 \\ -6.9375 \\ -5.1445 \end{pmatrix}$$

При проверке соотношений $Ax = b$ и $PAP^t = LTL^*$ приведённые выше матрицы показали правильные результаты. Кроме прочего они совпали с вычислениями, произведёнными вручную.

4.2 Матрица 7x7

Для второго теста были сгенерированы симметричная матрица A и матрица b, элементами которых являются числа в интервале $[-100, 100]$:

$$A = \begin{pmatrix} -94 & 78 & -72 & 21 & -21 & -10 & 21 \\ 78 & 20 & -12 & 19 & -55 & -22 & 37 \\ -72 & -12 & -42 & -33 & -11 & -20 & 83 \\ 21 & 19 & -33 & -2 & 43 & 94 & -33 \\ -21 & -55 & -11 & 43 & 20 & 0 & -29 \\ -10 & -22 & -20 & 94 & 0 & 22 & -66 \\ 21 & 37 & 83 & -33 & -29 & -66 & 2 \end{pmatrix}, \quad b = \begin{pmatrix} -6 \\ 5 \\ 4 \\ -13 \\ -27 \\ -46 \\ 31 \end{pmatrix}.$$

Методом Банча-Парлетта были найдены матрицы разложения:

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix},$$

$$T = \begin{pmatrix} -94 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 84.7234 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -47.6052 & 113.003 & 0 & 0 & 0 \\ 0 & 0 & 113.003 & -28.2709 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -43.5929 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 57.02392 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 7.3826 \end{pmatrix},$$

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -0.8298 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0.7660 & -0.8468 & 1 & 0 & 0 & 0 & 0 \\ -0.2234 & 0.6424 & 0 & 1 & 0 & 0 & 0 \\ -0.2234 & 0.4299 & -0.5566 & -0.3959 & 1 & 0 & 0 \\ 0.1064 & -0.3576 & -0.5765 & -0.5791 & -1.4757 & 1 & 0 \\ 0.2234 & -0.8548 & -0.0122 & -0.5029 & -0.9914 & 0.2652 & 1 \end{pmatrix}.$$

Аналогично для метода Банча-Кауфмана:

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

$$T = \begin{pmatrix} -94 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 84.7234 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -12.9691 & -18.2396 & 0 & 0 & 0 \\ 0 & 0 & -18.2396 & -47.6052 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1413.7443 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 36.9549 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 14.0942 \end{pmatrix},$$

$$L = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -0.8298 & 1 & 0 & 0 & 0 & 0 & 0 \\ -0.2234 & 0.4299 & 1 & 0 & 0 & 0 & 0 \\ 0.7660 & -0.8468 & 0 & 1 & 0 & 0 & 0 \\ 0.2234 & -0.8548 & -15.2149 & 7.0110 & 1 & 0 & 0 \\ 0.1064 & -0.3576 & -19.9557 & 8.4441 & 1.2995 & 1 & 0 \\ -0.2234 & 0.6424 & 15.88498 & -8.4500 & -1.1078 & 0.0096 & 1 \end{pmatrix}.$$

Стоит сразу отметить, что для удобства читаемости элементы всех матриц записаны как числа с 4 знаками после запятой. Сама программа оперирует числами с 16 знаками после запятой. Далее было найдено решение системы $Ax = b$, где

$$x = \begin{pmatrix} 0.0846 \\ -0.0363 \\ -0.2863 \\ -0.8244 \\ -0.4279 \\ -0.2372 \\ -0.4702 \end{pmatrix}.$$

Так как становится проблематичным проверять матрицы большой размерности, то далее для проверки результатов будет использоваться библиотека *numpy*. В данном случае вычисления показали правильные результаты.

4.3 Матрицы больших размеров

Далее будут сгенерированы несколько матриц разного размера, а также будет проанализировано время в секундах работы каждого из алгоритмов.

Шаг	Метода Банча-Парлетта (сек)	Метод Банча-Кауфмана (сек)
Матрица размера 17x17		
1	0.0080	0.0070
2	0.0065	0.0065
3	0.0075	0.0070
Матрица размера 100x100		
1	0.4081	0.1885
2	0.3571	0.1965
3	0.3376	0.1960
Матрица размера 500x500		
1	80.0489	62.0738
2	70.4752	59.2997
3	69.9380	47.3335

Оба алгоритма имееют вычислительную сложность: $n^3/3+O(n^2)$, но в случае метода Банча-Парлетта к этому времени добавляется еще от $n^3/6$ до $n^3/12$ операций сравнений, необходимых для выбора диагонального блока. В случае метода Банча-Кауфмана алгоритм при выборе нового диагонального блока оперирует не со всей ведущей подматрицей, а лишь с её диагональю и двумя столбцами. Собственно, эта разница во времени видна выше в таблице.

5 Вывод

В ходе выполнения лабораторной работы были изучены два алгоритма для нахождения разложения $PAP^t = LTL^*$: метод Банча-Кауфмана и метод Банча-Парлетта, и было найдено решение системы $Ax = b$ с помощью данного разложения, для этого реализована программа на языке Python3. Кроме этого была проанализирована работа этих методов на матрицах разного размера.

Список литературы

[1] Голубков А. Ю. Лекции по вычислительным методам линейной алгебры 2013-2016 г. 162 с.