Решение систем линейных уравнений

С.В. Лемешевский (sergey.lemeshevsky@gmail.com)

Институт математики НАН Беларуси

Oct 23, 2018

Содержание

Проблема решения линейной системы

$$Ax = b \tag{1}$$

является центральной в научных вычислениях. В этой главе мы остановимся на методах решения систем вида (1). Сначала остановимся на методе исключения Гаусса, а затем рассмотрим некоторые итерационные методы.

1. Прямые методы линейной алгебры

Одной из основных задач вычислительной математики является проблема решения систем линейных алгебраических уравнений с вещественными ко- эффициентами. Для нахождения приближенного решения систем уравнений используются прямые и итерационные методы. Математический аппарат ли- нейной алгебры базируется на понятиях нормы вектора и матрицы, числа обусловленности. Рассматриваются классические методы исключения неиз- вестных, отмечаются особенности решения задач с симметричной веществен- ной матрицей.

1.1. Метод исключения Гаусса

Начнем с обсуждения того, как можно легко решать треугольные системы. Затем опишем приведение системы общего вида к треугольной форме при помощи преобразований Гаусса. И, наконец, учитывая то, что полученный метод ведет себя очень плохо на нетривиальном классе задач, рассмотрим концепцию выбора ведущих элементов.

Треугольные системы. Рассмотрим следующую треугольную 2×2-систему:

$$\begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Если $l_{11}, l_{22} \neq 0$, то неизвестные могут быть определены последовательно:

$$\begin{cases} x_1 &= b_1/l_{11}, \\ x_2 &= (b_2 - l_{21}x_1)/l_{22} \end{cases}$$

Это 2×2-версия алгоритма, известного как *прямая подстановка*. Общую процедуру получаем, разрешая i-е уравнение системы $Lx = b$ относительно x_i:

$$x_i = \left(b_i - \sum_{j=1}^{i-1} l_{ij} x_j \right) / l_{ii}.$$

Если вычисления выполнить для i от 1 до n, то будут получены все компоненты вектора x. Заметим, что на i-м шаге необходимо скалярное произведение векторов $L(i, 1:i-1)$ и $x(1:i-1)$. Так как b_i содержится только в формуле для x_i, мы можем записать x_i на месте b_i.

Прямая подстановка.

Предположим, что $L \in \mathbb{R}^{n\times n}$ — нижняя треугольная матрица и $b \in \mathbb{R}^n$. Следующий код Python заменяет b на решение системы $Lx = b$. Матрица L должна быть невырождена.

```
b[0] = b[0]/L[0,0]
for i in range(1,len(b)):
    b[i] = (b[i] - np.dot(L[i,:i], b[:i]))/L[i,i]
```

Аналогичный алгоритм для верхней треугольной системы $Ux = b$ называется *обратная подстановка*. Вот формула для x_i:

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

и снова x_i можно записать на месте b_i.

Обратная подстановка.

Если матрица $U \in \mathbb{R}^{n\times n}$ верхняя треугольная и $b \in \mathbb{R}^n$, то следующий код Python заменяет b на решение системы $Ux = b$. Матрица U должна быть невырождена.

```
b[-1] = b[-1]/U[-1,-1]
for i in range(len(b)-2, -1, -1):
    b[i] = (b[i] - np.dot(U[i,i+1:], b[i+1:]))/U[i,i]
```

Отметим, что при реализации формул прямой и обратной подстановки мы использовали срезы массивов (см. раздел **??**). В первом алгоритме `L[i,:i]` означает, что берется из строки двумерного массива с индексом `i` все элементы с нулевого до `i-1`-го включительно, а `b[:i]` — элементы массива `b` с индексами от `0` до `i-1` включительно. Во втором алгоритме используются срезы `U[i,i+1:]`, содержащий от `i+1`-го до последнего (включительно) элементы `i`-той строки, и `b[i+1:]` с элементами от `i+1`-го до по-

следнего (включительно). Кроме того использовалась функция `dot` модуля `numpy`, которая вычисляет скалярное произведение двух векторов. Таким образом, мы здесь использовали векторизованные вычисления.

LU-разложение. Как мы только что видели, треугольные системы решаются «легко». Идея метода Гаусса — это преобразование системы (1) в эквивалентную треугольную систему. Преобразование достигается соответствующих линейных комбинаций уравнений. Например, в системе

$$\begin{aligned} 3x_1 + 5x_2 &= 9, \\ 6x_1 + 7x_2 &= 4, \end{aligned}$$

умножая ее первую строку на 2 и вычитая ее из второй части, мы получим

$$\begin{aligned} 3x_1 + 5x_2 &= 9, \\ -3x_2 &= -14. \end{aligned}$$

Это и есть метод исключений Гаусса при $n = 2$. Дадим полное описание этой важной процедуры, причем опишем ее выполнение на языке матричных разложений. Данный пример показывает, что алгоритм вычисляет нижнюю треугольную матрицу L и верхнюю треугольную матрицу U так, что $A = LU$, т.е.

$$\begin{bmatrix} 3 & 5 \\ 6 & 7 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 3 & 5 \\ 0 & -3 \end{bmatrix}$$

Решение исходной задачи $Ax = b$ находится посредством последовательного решения двух треугольных систем:

$$Ly = b, \quad Ux = y \quad \Rightarrow Ax = LUx = Ly = b$$

Матрица преобразования Гаусса. Чтобы получить разложение, описывающее исключение Гаусса, нам нужно иметь некоторое матричное описание процесса обнуления матрицы. Пусть $n = 2$, тогда как $x_1 \neq 0$ и $\tau = x_2/x_1$, то

$$\begin{bmatrix} 1 & 0 \\ -\tau & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_1 \\ 0 \end{bmatrix}$$

В общем случае предположим, что $x \in \mathbb{R}^n$ и $x_k \neq 0$. Если

$$\tau^{(k)T} = [\underbrace{0, \ldots, 0}_{k}, \tau_{k+1}, \ldots, \tau_n], \quad \tau_i = \frac{x_i}{x_k} \quad i = k+1, k+2, \ldots, n$$

и мы обозначим

$$M_k = I - \tau^{(k)} e_k^T, \tag{2}$$

где

$$e_k^T = [\underbrace{0, \ldots, 0}_{k-1}, 1, \underbrace{0, \ldots, 0}_{n-k}],$$
$$I = [e_1, e_2 \ldots, e_n]$$

то

$$M_k x = \begin{bmatrix} 1 & \ldots & 0 & 0 & \ldots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \ldots & 1 & 0 & \ldots & 0 \\ 0 & \ldots & -\tau_{k+1} & 1 & \ldots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \ldots & -\tau_n & 0 & \ldots & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ x_{k+1} \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Матрица M_k — это матрица *преобразования Гаусса*. Она является нижней унитреугольной. Компоненты $\tau_{k+1}, \tau_{k+2}, \ldots, \tau_n$ — это *множители Гаусса*. Вектор $\tau^{(k)}$ называется *вектором Гаусса*.

Для реализации данных идей имеется функция, которая вычисляет вектор множителей. Если `x` — массив из `n` элементов и `x[0]` ненулевой, функция `gauss` возвращает вектор длины $n-1$, такой, что если `M` — матрица преобразования Гаусса, причем `M[1:,1] = -gauss(x)` и `y = dot(M,x)`, то `y[1:] = 0`:

```
def gauss(x):
    x = np.array(x, float)
    return x[1:]/x[0]
```

Применение матриц преобразовния Гаусса. Умножение на матрицу преобразования Гаусса выполняется достаточно просто. Если матрица $C \in \mathbb{R}^{n \times r}$ и $M_k = I - \tau^{(k)} e_k^T$, тогда преобразование вида

$$M_k C = (I - \tau^{(k)} e_k^T) C = C - \tau^{(k)} (e_k^T C)$$

осуществляет одноранговую модификацию. Кроме того, поскольку элементы вектора $\tau^{(k)}$ равны нулю от первого до k-го равны нулю, то в каждой k-ой строке матрицы C задействованы лишь элементы, начиная с $k+1$-го. Следовательно, если “C“ — двумерный массив, задающий матрицу C, и “M“ задает $n \times n$-преобразование Гаусса M_1, причем “M[1:,1] = -t“, “t“ — множитель Гаусса, соответствующий $\tau^{(1)T}$, тогда следующая функция заменяет C на $M_1 C$:

```
def gauss_app(C, t):
    C = np.array(C, float)
```

```
    t = np.array([[t[i]] for i in range(len(t))], float)
    C[1:, :] = C[1:, :] - t*C[0, :]
    return C
```

Отметим, что если матрица `M[k+1:,k] = -t`, тогда обращение вида `C[k:,:] = gauss_app(C[k:,:], t)` заменяет C на M_kC

Матрицы преобразовния Гаусса $M_1, M_2, \ldots, M_{n-1}$, как правило, можно подобрать так, что матрица $M_{n-1}M_{n-2}\ldots M_1A = U$ является верхней треугольной. Легко убедиться, что если $M_k = I - \tau^{(k)}e_k^T$, тогда обратная к ней задается следующим выражением $M_k^{-1} = I + \tau^{(k)}e_k^T$ и поэтому

$$A = LU, \tag{3}$$

где

$$L = M_1^{-1}M_2^{-1}\ldots M_{n-1}^{-1}.$$

Очевидно, что L — это нижняя унитреугольная матрица. Разложение (3) называется *LU-разложением* матрицы A. Необходимо проверять *ведущие элементы* матрицы A (a_{kk}) на нуль, чтобы избежать деления на нуль в функции `gauss`. Это говорит о том, что LU-разложение может не существовать. Известно, что LU-разложение матрицы A существует, если главные миноры матрицы A не равны нулю при этом оно единственно и $\det A = u_{11}u_{22}\cdots u_{nn}$.

Реализация. Рассмотрим пример при $n = 3$:

```
In [1]: import numpy as np

In [2]: A = np.array([[1, 4, 7], [2, 5, 8], [3, 6, 10]])

In [3]: A
Out[3]:
array([[ 1,  4,  7],
       [ 2,  5,  8],
       [ 3,  6, 10]])

In [4]: M1 = np.array([[1, 0, 0], [-2, 1, 0], [-3, 0, 1]])

In [5]: M1
Out[5]:
array([[ 1,  0,  0],
       [-2,  1,  0],
       [-3,  0,  1]])

In [6]: np.dot(M1, A)
Out[6]:
array([[  1,   4,   7],
       [  0,  -3,  -6],
       [  0,  -6, -11]])

In [7]: M2 = np.array([[1, 0, 0], [0, 1, 0], [0, -2, 1]])
```

```
In [8]: M2
Out[8]:
array([[ 1,  0,  0],
       [ 0,  1,  0],
       [ 0, -2,  1]])

In [9]: np.dot(M2,np.dot(M1,A))
Out[9]:
array([[ 1,  4,  7],
       [ 0, -3, -6],
       [ 0,  0,  1]])
```

Функция `numpy.dot`.

Обратите внимание, что в приведенном примере мы использовали функцию `dot` модуля `numpy`, которая выполняет умножение матриц в "правильном смысле", в то время как выражение `M1*A` производит поэлементное умножение.

Обобщение этого примера позволяет представить k-й шаг следующим образом:

- Мы имеем дело с матрицей $A^{(k-1)} = M_{k-1} \cdots M_1 A$, которая с 1-го по $(k-1)$-й столбец является верхней треугольной.
- Поскольку мы уже получили нули в столбцах с 1-го по $(k-1)$-й, то преобразование Гаусса можно применять только к столбцам с k-го до n-го. На самом деле нет необходимости применять преобразование Гаусса также и k-му столбцу, так как мы знаем результат.
- Множители Гаусса, задающие матрицу M_k получаются по матрице $A(k:n,k)$ и могут храниться в позициях, в которых получены нули.

С учетом сказанного выше мы можем написать следующую функцию:

```
def lu(A):
    LU = np.array(A, float)
    for k in range(LU.shape[0]-1):
        t = gauss(LU[k:, k])
        LU[k+1:, k] = t
        LU[k:, k+1:] = gauss_app(LU[k:, k+1:], t)

    return LU
```

Эта функция возвращает LU-разложение матрицы A. Где же храниться матрица L? Дело в том, что если $L = M_1^{-1} M_2^{-1} \dots M_{n-1}^{-1}$, то элементы с $(k+1)$-го до n-го в k-том столбце матрицы L равны множителям Гаусса $\tau_{k+1}, \tau_{k+2}, \dots, \tau_n$ соответственно. Этот факт очевиден, если посмотреть на произведение, задающее матрицу L:

$$L = (I + \tau^{(1)} e_1^T \cdots (I + \tau^{(n-1)} e_{n-1}^T)) = I + \sum_{k=1}^{n-1} \tau^{(k)} e_k^T.$$

Поэтому элементы $l_{ik} = lu_{ik}$ для всех $i > k$. Здесь lu_{ik} — элементы матрицы возвращаемой функцией `lu`.

После разложения матрицы A с помощью функции `lu` в возвращаемом массивы будут храниться матрицы L и U. Поэтому мы можем решить систему $Ax = b$, используя прямую и обратную подстановки описанные в разделе 1.1:

```
def solve_lu(A, b):
    LU = lu(A)
    b = np.array(b, float)
    for i in range(1, len(b)):
        b[i] = b[i] - np.dot(LU[i, :i], b[:i])
    for i in range(len(b)-1, -1, -1):
        b[i] = (b[i] - np.dot(LU[i, i+1:], b[i+1:]))/LU[i, i]
    return b
```

Замечание.

Отметим, что во всех представленных функциях мы выполняли явное преобразование входных параметров в массивы NumPy с элементами типа `float`. Это позволит правильно работать функциям в случае, если мы по ошибке создадим входные параметры не как массивы, а как списки.

Тестирование. Как известно метод Гаусса является прямым, т.е. дает точное решение системы линейных уравнений. Для проверки реализации решения системы линейных уравнений методом Гаусса мы можем написать следующую функцию:

```
def test_solve_lu():
    A = np.array([[1, 4, 7], [2, 5, 8], [3, 6, 10]])
    expected = np.array([-1./3, 1./3, 0])
    b = np.dot(A, expected)
    computed = solve_lu(A, b)
    tol = 1e-14
    success = np.linalg.norm(computed - expected) < tol
```

```
    msg = 'x_exact = ' + str(expected) + '; x_computed = ' + str(computed)
    assert success, msg
```

Замечание.

Здесь мы задали матрицу `A` системы и точное решение `expected`, на основе которых получили вектор правой части `b = np.dot(A,x)`. Для сравнения численного решения с точным используется функция `np.linalg.norm`. В случае вызова с одним аргументом вычисляется l_2-норма: $\|v\|_2 = \sqrt{\sum_{i=1}^{n} v_i^2}$.

Выбор ведущего элемента. Как уже упоминалось, LU-разложение может не существовать. В методе Гаусса с выбором ведущего элемента на очередном шаге исключается неизвестное, при котором коэффициент по модулю является наибольшим. В этом случае метод Гаусса применим для любых невырожденных матриц ($\det A \neq 0$).

Такая стратегия предполагает переупорядочивание данных в виде перестановки двух матричных строк. Для этого используются понятие перестановочной матрицы. *Перестановочная матрица* (или *матрица перестановок*) — это матрица, отличающаяся от единичной лишь перестановкой строк, например

$$P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

Перестановочную матрицу нет необходимости хранить полностью. Гораздо более эффективно перестановочную матрицу можно представить в виде целочисленного вектора p длины n. Один из возможных способов такого представления — это держать в p_k индекс столбца в k-й строке, содержащий единственный элемент равный 1. Так вектор $p = [4, 1, 3, 2]$ соответствует кодировке приведенной выше матрицы P. Также возможно закодировать P указанием индекса строки в k-ом столбце, содержащего 1, например, $p = [2, 4, 3, 1]$.

Если P — это матрица перестановок, а A — некоторая матрица, тогда матрица AP является вариантом матрицы A с переставленными столбцами, а PA — вариантом матрицы A с переставленными строками.

Перестановочные матрицы ортогональны, и поэтому если P — перестановочная матрица, то $P^{-1} = P^T$.

В этом разделе особый интерес представляют *взаимные перестановки*. Такие перестановки осуществляют матрицы, получаемые простой переменой мест двух строк единичной матрицы, например

$$E = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Взаимные перестановки могут использоваться для описания перестановок строк и столбцов матрицы. В приведенном примере порядка 4×4 матрица EA отличается от матрицы A перестановкой 1-й и 4-й строк. Аналогично матрица AE отличается от матрицы A перестановкой 1-го и 4-го столбцов.

Если $P = E_n E_{n-1} \cdots E_1$ и каждая матрица E_k является единичной с переставленными k-й и p_k-й строками, то вектор $p = [p_1, p_2, \ldots, p_n]$ содержит всю необходимую информацию о матрице P. Действительно, вектор x может быть замещен на вектор Px следующим образом:

$$\textbf{for } k = 1 : n$$
$$x_k \leftrightarrow x_{p_k}$$

Здесь символ $\leftrightarrow$ обозначает «выполнение перестановки»:

$$x_k \leftrightarrow x_{p_k} \;\Leftrightarrow\; r = x_k,\; x_k = x_{p_k},\; x_{p_k} = r.$$

Поскольку каждая матрица E_k является симметричной и $P^T = E_1 E_2 \cdots E_n$, то также можно выполнить замещение вектора x на вектор $P^T x$:

$$\textbf{for } k = n : 1 : -1$$
$$x_k \leftrightarrow x_{p_k}$$

Существуют разные стратегии выбора ведущего элемента. Мы остановимся на стратегии частичного выбора. Пусть матрица

$$A = \begin{bmatrix} 3 & 17 & 10 \\ 2 & 4 & -2 \\ 6 & 18 & -12 \end{bmatrix}.$$

Чтобы добиться наименьших множителей в первой матрице разложения по Гауссу с помощью взаимных перестановок строк, надо сделать элемент a_{11} наибольшим в первом столбце. Если E_1 — матрица взаимных перестановок, тогда

$$E_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

Поэтому

$$E_1 A = \begin{bmatrix} 6 & 18 & -12 \\ 2 & 4 & -2 \\ 3 & 17 & 10 \end{bmatrix}$$

и

$$M_1 = \begin{bmatrix} 1 & 0 & 0 \\ -1/3 & 1 & 0 \\ -1/2 & 0 & 1 \end{bmatrix} \Rightarrow M_1 E_1 A = \begin{bmatrix} 6 & 18 & -12 \\ 0 & -2 & 2 \\ 0 & 8 & 16 \end{bmatrix}.$$

Теперь, чтобы получить наименьший множитель в матрице M_2, необходимо переставить 2-ю и 3-ю строки и т.д.

Пример иллюстрирует общую идею, основанную на перестановке строк. Обобщая эту идею, получим следующий алгоритм:

LU-разложение с частичным выбором.

Если матрица $E \in \mathbb{R}^{n \times n}$, то данный алгоритм вычисляет матрицы преобразования Гаусса $M_1, M_2 \ldots, M_{n-1}$ и матрицы взаимных перестановок $E_1, E_2, \ldots, E_{n-1}$, такие что матрица $M_{n-1} E_{n-1} \cdots M_1 E_1 A = U$ является верхней треугольной. При этом нет множителей, превосходящих 1 по абсолютной величине. Подматрица $[a_{ik}]_{i=1}^{k}$ замещается на матрицу $[u_{ik}]_{i=1}^{k}$, $k = 1, 2, \ldots, n$. Подматрица $[a_{ik}]_{i=k+1}^{n}$ замещается на матрицу $[m_{k;ik}]_{i=k+1}^{n}$, $k = 1, 2, \ldots, n-1$. Целочисленный вектор piv размера $n-1$ задает взаимные перестановки. В частности, матрица E_k переставляет строки k и piv_k, $k = 1, 2, \ldots, n-1$.

for $k = 1 : n$

1. Зададим μ, такое что $k \le \mu \le n$ и $|a_{\mu k}| = \max_{k \le i \le n} |a_{ik}|$
2. $a_{k,k:n} \leftrightarrow a_{\mu,k:n}$; $piv_k = \mu$

if $a_{kk} \neq 0$

$t =$ `gauss` $(A_{k:n,k})$; $A_{k+1:n,k} = t$

$A_{k:n,k+1:n} =$ `gauss_app` $(A_{k:n,k+1:n}, t)$

end if

end for

Чтобы решить линейную систему $Ax = b$ после вызова последнего алгоритма, мы должны

1. Вычислить вектор $y = M_{n-1} E_{n-1} \cdots M_1 E_1 b$. 2. Решить верхнюю треугольную систему $Ux = y$.

1.2. Методы решения систем с симметричными матрицами

Здесь мы опишем методы, использующие специфику при решении задачи $Ax = b$. В случае, когда A — симметричная невырожденная матрица, т.е. $A = A^T$ и $\det(A) \neq 0$, существует разложение вида

$$A = LDL^T, \tag{4}$$

где L — нижняя унитреугольная матрица, D — диагональная матрица. В связи с этим работа связанная с получением разложения :eq:$sles - ldl$, составляет половину от того, что требуется для исключения Гаусса. Когда разложение :eq:$sles - ldl$ получено, решение системы $Ax = b$ может быть найдено посредством решения систем $Ly = b$ (прямая подстановка), $Dz = y$ и $L^T x = z$.

LDL^T-разложение. Разложение (4) может быть найдено при помощи исключения Гаусса, вычисляющего $A = LU$, с последующим определением D из уравнения $U = DL^T$. Тем не менее можно использовать интересный альтернативный алгоритм непосредственного вычисления L и D.

Допустим, что мы знаем первые $j-1$ столбцов матрицы L, диагональные элементы $d_1, d_2, \ldots, d_{j-1}$ матрицы D для некоторого j, $1 \leq j \leq n$. Чтобы получить способ вычисления l_{ij}, $i = j+1, j+2, \ldots, n$, и d_j приравняем j-е столбцы в уравнении $A = LDL^T$. В частности,

$$A(1:j, j) = Lv, \tag{5}$$

где

$$v = DL^T e_j = \begin{bmatrix} d_1 l_{j1} \\ \vdots \\ d_{j-1} l_{jj-1} \\ d_j \end{bmatrix}.$$

Следовательно, компоненты v_k, $k = 1, 2, \ldots, j-1$ вектора v могут быть получены простым масштабированием элементов j-й строки матрицы L. Формула для j-й компоненты вектора v получается из j-го уравнения системы $L(1:j, 1:j)v = A(1:j, j)$:

$$v_j = a_{jj} - \sum_{k=1}^{j-1} l_{ik} v_k,$$

Когда мы знаем v, мы вычисляем $d_j = v_j$. «Нижняя» половина формулы (5) дает уравнение

$$L(j+1:n, 1:j)v(1:j) = A(j+1:n, j),$$

откуда для вычисления j-го столбца матрицы L имеем:

$$L(j+1:n,j) = (A(j+1:n,j) - L(j+1:n,1:j-1)v(1:j-1))/v_j.$$

Реализация. Для получения LDL^T-разложения матрицы A можем написать функцию (сценарий ld.py):

```
def ld(A):
    """
    Для симметричной матрицы A вычисляет нижнюю треугольную
    матрицу L и диагональную матрицу D, такие
    что A = LDL^T. Элементы a_{ij} замещаются    на l_{ij}, если i > j,
    и на d_i, если i = j
    """

    n = len(A)
    LD = np.array(A, float)
    for j in range(n):
        v = np.zeros(j+1)
        v[:j] = LD[j,:j]*LD[range(j),range(j)]
        v[j] = LD[j,j] - np.dot(LD[j,:j],v[:j])
        LD[j,j] = v[j]
        LD[j+1:,j] = (LD[j+1:,j] - np.dot(LD[j+1:,:j],v[:j]))/v[j]

    return LD
```

В этой реализации мы использовали векторизованные вычисления. Разберем некоторые выражения. Строка

```
v[:j] = LD[j,:j]*LD[range(j),range(j)]
```

можно заменить следующим циклом:

```
for i in range(j):
    v[i] = LD[j,i]*LD[i,i]
```

В нашей программе доступ к `j` диагональным элементам массива `A` осуществляется выражением `A[range(j),range(j)]`.

При вычислении `v[j]` использовалась функция `np.dot`, которая вычисляет скалярное произведение векторов.

Отметим также строку

```
LD[j+1:,j] = (LD[j+1:,j] - np.dot(LD[j+1:,:j],v[:j]))/v[j]
```

в которой используется срез `L[j+1:,j]`, т.е. элементы с `j+1`-го до последнего в `j`-ом столбце.

Для решения системы $Ax = b$‘ с использованием LDL^T-разложения можно написать следующую функцию

```
def ld_solve(A, b):
    """
    Решает систему Ax = b с использованием LDL^T-разложения
    """
    LD = ld(A)
    b = np.array(b,float)
    for i in range(1, len(b)):
        b[i] = b[i] - np.dot(LD[i,:i],b[:i])
    b[:] = b[:]/LD[range(len(b)),range(len(b))]
    for i in range(len(b)-1, -1, -1):
        b[i] = (b[i] - np.dot(LD[i+1:,i],b[i+1:]))
    return b
```

Разложение Холецкого. Известно, что в случае симметричной положительно определенной матрицы разложение (4) существует и устойчиво. Тем не менее в этом случае можно использовать другое разложение:

$$A = GG^T \tag{6}$$

известное как *разложение Холецкого*, а матрицы G называются *треугольниками Холецкого*.

Это легко показать, исходя из существования LDL^T разложения. Так как для симметричной положительно определенной матрицы существует $A = LDL^T$ и диагональные элементы матрицы D положительны, то $G = L\mathrm{diag}(\sqrt{d_{11}}, \sqrt{d_{22}}, \ldots, \sqrt{d_{nn}})$.

2. Итерационные методы решения систем линейных алгебраических уравнений

2.1. Стандартные итерационные методы

В разделах 1.1 и 1.2 процедуры решения систем алгебраических уравнений были связаны с разложением матрицы коэффициентов A. Методы такого типа называются *прямыми методами*. Противоположностью прямым методам являются *итерационные методы*. Эти методы порождают последовательность приближенных решений $\{x^{(k)}\}$. При оценивании качества итерационных методов в центре внимания вопрос от том, как быстро сходятся итерации $x^{(k)}$.

Итерации Якоби и Гаусса — Зейделя. Простейшей итерационной схемой, возможно, являются *итерации Якоби*. Они определяются для матриц с ненулевыми диагональными элементами.

Идею метода можно представить, используя запись 3×3-системы $Ax = b$ в следующем виде:

$$x_1 = (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11},$$
$$x_2 = (b_2 - a_{21}x_1 - a_{23}x_3)/a_{22},$$
$$x_3 = (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}.$$

Предположим, что $x^{(k)}$ — какое-то приближение к $x = A^{-1}b$. Чтобы получить новое приближение $x^{(k+1)}$, естественно взять:

$$x_1^{(k+1)} = (b_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)})/a_{11},$$
$$x_2^{(k+1)} = (b_2 - a_{21}x_1^{(k)} - a_{23}x_3^{(k)})/a_{22},$$
$$x_3^{(k+1)} = (b_3 - a_{31}x_1^{(k)} - a_{32}x_2^{(k)})/a_{33}.$$

Эти формулы и определяют итерации Якоби в случае $n = 3$. Для произвольных n мы имеем

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)} \right) /a_{ii}, \quad i = 1, 2, \ldots, n. \qquad (7)$$

Заметим, что в итерациях Якоби при вычислении $x_i^{(k+1)}$ не используется информация, полученная в самый последний момент. Например, при вычислении $x_2^{(k+1)}$ используется $x_1^{(k)}$, хотя уже известна компонента $x_1^{(k+1)}$. Если мы пересмотрим итерации Якоби с тем, чтобы всегда использовать самые последние оценки для x_i, то получим:

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)} \right) /a_{ii}, \quad i = 1, 2, \ldots, n. \qquad (8)$$

Так определяется то, что называется *итерациями Гаусса — Зейделя*.

Для итераций Якоби и Гаусса — Зейделя переход от $x^{(k)}$ к $x^{(k+1)}$ в сжатой форме описывается в терминах матриц L, D и U, опреде-

ляемых следующим образом:

$$L = \begin{bmatrix} 0 & 0 & \cdots & \cdots & 0 \\ a_{21} & 0 & \cdots & \cdots & 0 \\ a_{31} & a_{32} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn-1} & 0 \end{bmatrix},$$

$$D = \operatorname{diag}(a_{11}, a_{12}, \ldots, a_{nn}),$$

$$U = \begin{bmatrix} 0 & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & 0 & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & a_{n-1n} \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix}.$$

Шаг Якоби имеет вид $M_J x^{(k+1)} = N_J x^{(k)} + b$, где $M_J = D$ и $N_J = -(L+U)$. С другой стороны, шаг Гаусса — Зейделя определяется как $M_G x^{(k+1)} = N_G x^{(k)} + b$, где $M_G = (D+L)$ и $N_G = -U$.

Процедуры Якоби и Гаусса — Зейделя — это типичные представители большого семейства итерационных методов, имеющих вид

$$Mx^{(k+1)} = Nx^{(k)} + b, \tag{9}$$

где $A = M - N$ — расщепление матрицы A. Для практического применения итераций (9) должна «легко» решаться система с матрицей M. Заметим, что для итераций Якоби и Гаусса — Зейделя матрица M соответственно диагональная и нижняя треугольная.

Сходятся ли итерации (9) к $x = A^{-1}b$, зависит от собственных значений матрицы $M^{-1}N$. Определим *спектральный радиус* произвольной $n \times n$-матрицы G как

$$\rho(G) = \max\{|\lambda| : \ \lambda \in \lambda(G)\},$$

тогда если матрица M невырожденная и $\rho(M^{-1}N) < 1$, то итерации $x^{(k)}$, определенные согласно $M^{(k+1)} = Nx^{(k)} + b$, сходятся к $x = A^{-1}b$ при любом начальном векторе $x^{(0)}$.

Последовательная верхняя релаксация. Метод Гаусса — Зейделя очень привлекателен в силу своей простоты. К несчастью, если спектральный радиус для $M_G^{-1}N_G$ близок к единице, то метод может оказаться непозволительно медленным из-за того, что ошибки стремятся к нулю как $\rho(M_G^{-1}N_G)^k$. Чтобы исправить это,

возьмем $\omega \in \mathbb{R}$ и рассмотрим следующую модификацию шага Гаусса — Зейделя:

$$x_i^{(k+1)} = \omega \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) / a_{ii} + (1-\omega) x_i^{(k)}. \qquad (10)$$

Так определяется метод *последовательной верхней релаксации* (SOR — Successive Over Relaxation). В матричных обозначениях шаг SOR выглядит как

$$M_\omega x^{(k+1)} = N_\omega x^{(k)} + \omega b,$$

где $M_\omega = D + \omega L$ и $N_\omega = (1-\omega)D - \omega U$. Для небольшого числа специфических задач значения реалксационного параметра ω, минимизирующего $\rho(M_\omega^{-1} N_\omega)$, является известным. В более сложных задачах, однако, для того чтобы определить подходящее ω, может возникнуть необходимость в выполнении весьма трудного анализа собственных значений.

2.2. Метод сопряженных градиентов

Трудность, связанная с SOR и такого же типа методами, заключается в том, что они зависят от параметров, правильный выбор которых иногда бывает затруднителен. Например, для того чтобы чебышевское ускорение было успешным, нам нужны хорошие оценки для наибольшего и наименьшего собственных значений соответствующей итерационной матрицы ^{-1}N. Если эта матрица не устроена по-особому, то получение их в аналитическом виде, скорее всего, невозможно, а вычисление дорого.

Наискорейший спуск. Вывод метода связан с минимизацией функционала:

$$\varphi(x) = \frac{1}{2} x^T A x - x^T b,$$

где $b \in \mathbb{R}^n$ и матрица A предполагается положительно определенной и симметричной. Минимальное значение φ равно $-b^T A^{-1} b/2$ и достигается при $x = A^{-1} b$. Таким образом, минимизация φ и решение системы $Ax = b$ — эквивалентные задачи.

Одной из самых простых стратегий минимизации функционала φ является *метод наискорейшего спуска*. В текущей точке x_c функция φ убывает наиболее быстро в направлении антиградиента $\nabla\varphi(x_c) = b - Ax_c$. Мы называем $r_c = b - Ax_c$ *невязкой* вектора x_c. Если невязка ненулевая, то $\varphi(x_c + \alpha r_c) < \varphi(x_c)$ для некоторого положительного α (будем называть этот параметр *поправкой*). В

методе наискорейшего спуска (с точной минимизацией на прямой) мы берем поправку

$$\alpha = \frac{r_c^T r_c}{r_c^T A r_c},$$

дающую минимум для $\varphi(x_c + \alpha r_c)$. Итерационный процесс запишется следующим образом

$$\begin{aligned} \alpha_k &= \frac{r_{k-1}^T r_{k-1}}{r_{k-1}^T A r_{k-1}}, \\ x_k &= x_{k-1} + \alpha_k r_{k-1}, \\ r_k &= b - A x_k, \quad k = 1, 2, \ldots \end{aligned}$$

при начальных векторах $x_0 = 0$, $r_0 = b$.

К несчастью, скорость сходимости может быть недопустимо медленной, если число обусловленности $\kappa(A) = \lambda_1(A)/\lambda_2(A)$ большое. В этом случае линии уровня для φ являются сильно вытянутыми гиперэллипсоидами, а минимизация соответствует поиску самой нижней точки на относительно плоском дне крутого оврага. При наискорейшем спуске мы вынуждены переходить с одной стороны оврага на другую вместо того, чтобы спуститься к его дну. Направления градиента, возникающие при итерациях, являются слишком близкими; это и замедляет продвижение к точке минимума.

Произвольные направления спуска. Чтобы избежать ловушек при наискорейшем спуске, мы рассмотрим последовательную минимизацию φ вдоль какого-либо множества направлений $\{p_1, p_2, \ldots\}$, которые не обязаны соответствовать невязкам $\{r_0, r_1, \ldots\}$. Легко показать, что минимум $\varphi(x_{k-1} + \alpha p_k)$ по α дает

$$\alpha_k = \frac{p_k^T r_{k-1}}{p_k^T A p_k}.$$

Для того, чтобы обеспечить уменьшение функционала φ, мы должны потребовать, чтобы p_k не был ортогонален к r_{k-1}. Проблема состоит в том, как выбирать эти векторы, чтобы гарантировать глобальную сходимость и в то же время обойти ловушки наискорейшего спуска.

Метод сопряженных градиентов. Как было сказано выше направления спуска p_k нужно выбирать так, чтобы они не были ортогональны к невязкам r_{k-1}, т.е. $p_k r_{k-1} \neq 0$. Кроме того, метод сопряженных градиентов основан на том, что требуется, чтобы направление p_k было A-сопряженным по отношению к $p_1, p_2, \ldots, p_{k-1}$, т.е. $p_m^T A p_k = 0$ для $m = 1, 2, \ldots, k-1$.

Поскольку наша цель — осуществить быстрое сокращение величины невязок, естественно выбирать в качестве p_k вектор, который ближе всего к r_{k-1} среди векторов, A-сопряженных с $p_1, p_2, \ldots, p_{k-1}$.

Для получения таких направлений спуска и нахождения приближенного решения используется *метод сопряженных градиентов*. Ниже представлен код функции, реализующий данный алгоритм (файл cg.py)

```
def cg(A, b, tol, it_max):
    it = 0
    x = 0
    r = np.copy(b)
    r_prev = np.copy(b)
    rho = np.dot(r, r)
    p = np.copy(r)
    while (np.sqrt(rho) > tol*np.sqrt(np.dot(b, b)) and it < it_max):
        it += 1
        if it == 1:
            p[:] = r[:]
        else:
            beta = np.dot(r, r)/np.dot(r_prev, r_prev)
            p = r + beta*p
            w = np.dot(A, p)
            alpha = np.dot(r, r)/np.dot(p, w)
            x = x + alpha*p
            r_prev[:] = r[:]
            r = r - alpha*w
            rho = np.dot(r, r)

    return x, it
```

3. Тестирование реализации методов

4. Задачи

Задача 1: Решение системы линейных уравнений с трехдиагональной матрицей

Написать программу, которая решает систему линейных уравнений для трехдиагональной ($a_{ij} = 0$ при $|i - j| > 1$) $n \times n$-матрицы на основе LU-разложения. Написать следующие тестовые функции:

1. Найти решение уравнения с

$$a_{ii} = 2, \quad a_{ii-1} = a_{ii+1} = -1$$

при правой части $b_i = 2h^2$, $h = 1/n$, $i = 1, 2, \ldots, n-1$, $b_n = -(n-1) * h(1 - (n-1)/h)$ и сравнить его с точным решением $x_i = ih(1 - ih)$, $i = 1, 2, \ldots, n$.

2. Вычислить определитель матрицы и сравнить его значение с точным $n+1$.

Подсказка. Трехдиагональная матрица A задается тремя диагоналями:

$$d_i = a_{ii}, \quad e_i^u = a_{ii+1}, \quad e_i^l = a_{ii-1}.$$

В модуле функция (например, `lu3`) выполняет LU-разложение матрицы A и возвращает результат в виде трех диагоналей. Для решения системы используется другая функция (например, `solve_lu3`).

Задача 2: Метод Гаусса с частичным выбором ведущего элемента

Написать модуль, который реализует идеи частичного выбора ведущего элемента из раздела 1.1. Функция для LU-разложения должна выводить, кроме самого разложения, еще и вектор, определяющий матрицу перестановок. Напишите тестовые функции для проверки выполнения LU-разложения и решения системы уравнений с матрицей

$$A = \begin{bmatrix} 3 & 17 & 10 \\ 2 & 4 & -2 \\ 6 & 18 & -12 \end{bmatrix}$$

Задача 3: Разложение Холецкого

Написать программу, реализующую разложение Холецкого $A = GG^T$ для симметричной положительно определенной матрицы A и вычисляющей определитель матрицы на основе этого разложения. Найти разложение Холецкого и определитель матрицы Гильберта, для которой

$$a_{ij} = \frac{1}{i+j-1}, \quad i = 1, 2, \ldots, n, \ j = 1, 2, \ldots, n$$

при $n = 4$.

Задача 4: Метод Якоби

Написать программу, реализующую метод Якоби с использованием циклов Python (функция `jacobi`) и с векторизованными вычислениями (функция `jacobi_vec`). Сравнить время выполнения этих функций. Написать тестовые функции, проверяющие работу функции `jacobi`.

Задача 5: Метод Зейделя

Написать программу, реализующую метод Зейделя (функция `seidel`). Написать тестовые функции, проверяющие работу функции `seidel`.

Задача 6: Сравнение методов Якоби и Зейделя

Используя функции из 4 и 5, найти решение задачи системы $Ax = b$ с трехдиагональной матрицей A, в которой

$$a_{ii} = 2, \quad a_{ii+1} = -1 - \alpha, \quad a_{ii-1} = -1 + \alpha, \quad i = 1, 2 \ldots, n-1,$$

$$a_{00} = 2, \quad a_{01} = -1 - \alpha, \quad a_{n-1n} = -1 + \alpha, \quad a_{nn} = 2,$$

а правая часть

$$b_0 = 1 - \alpha, \quad b_i = 0, \quad i = 1, 2, \ldots, n-1, \quad b_n = 1 + \alpha,$$

определяет точное решение $x_i = 1$, $i = 1, 2, \ldots, n$. Сравнить скорости сходимости (число итераций) методов Якоби и Зейделя при различных параметрах n и α при $0 \leqslant \alpha \leqslant 1$. Для этого построить график зависимости числа итераций K от n при фиксированном α, а также график зависимости числа итераций K от α при фиксированном n.

Задача 7: Метод верхней релаксации

Написать программу, реализующую приближенное решение системы линейных алгебраических уравнений методов релаксации из 2.1. Написать тестовые функции. Исследовать графически зависимость скорости сходимости этого итерационного метода от итерационного параметра ω при численном решении системы уравнений из 6 при различных параметрах n и α.

Задача 8: Метод сопряженных градиентов

С помощью метода сопряженных градиентов (файл cg.py) найти решение системы $Ax = b$ с матрицей Гильберта из задачи 5 и правой частью

$$b_i = \sum_{j=1}^{n} a_{ij}, \quad i = 1, 2, \ldots, n,$$

для которой точное решение есть $x_i = 1$, $i = 1, 2, \ldots, n$. Построить график зависимости числа итераций от n.

Предметный указатель