

МИНОБРНАУКИ РОССИИ  
Федеральное государственное автономное образовательное  
учреждение высшего образования  
«Южный федеральный университет»

Институт компьютерных технологий и информационной безопасности

Кафедра интеллектуальных и многопроцессорных систем  
Направление 01.04.02 Прикладная математика и информатика

**МАТЕМАТИЧЕСКИЕ МОДЕЛИ ПРОЦЕССОВ И СИСТЕМ**  
Отчет по практической работе №2  
«Итерационные методы решения систем линейных алгебраических  
уравнений»

Выполнил:  
магистрант группы КТмо1-1  
\_\_\_\_\_ Р.С. Булычев  
подпись

Защита отчета состоялась  
\_\_\_\_\_

Количество баллов \_\_\_\_\_  
Проверил: д.т.н., профессор кафедры ИМС  
\_\_\_\_\_ / А.В. Никитина /  
подпись

Таганрог 2023

Вариант выполнения работы №3.

Задание 1.

Итерационные методы решения систем линейных алгебраических уравнений. Итерационный метод позволяет найти последовательность, сходящуюся к точному решению при  $n \rightarrow \infty$ , т.е.  $\lim_{n \rightarrow \infty} x_n = x$ . Поскольку бесконечные процессы нереализуемы на практике, то обычно выполняется конечное число итераций, т.е. строится конечное множество векторов  $x^{(1)}, x^{(2)}, \dots, x^{(n)}$ . Причем, задаваясь некоторым малым числом  $\varepsilon > 0$  (погрешностью решения), добиваются, чтобы  $\|x^{(n)} - x\| < \varepsilon$ , где  $\|\cdot\|$  – некоторая норма вектора.

Решите систему уравнений методом Якоби, Зейделя, наименьших невязок и методом скорейшего спуска

Постановка задачи (вариант 3 по условию)

$$\begin{cases} 4x_1 + 2x_2 - x_3 = -1, \\ 2x_1 + 5x_2 + x_3 = -1, \\ -x_1 + x_2 + 3x_3 = -8 \end{cases}$$

## ОПИСАНИЕ МЕТОДА ЯКОБИ

Для решения систем линейных алгебраических уравнений (СЛАУ) большой размерности, а также систем, имеющих разреженные матрицы, применение точных методов (например, метод Гаусса) не является целесообразным, так как сказывается ограниченность разрядной сетки и накапливается погрешность округления.

Система с разреженной матрицей получается, например, при составлении системы уравнений теплового баланса технологической схемы или же при численном расчете потенциалов из дифференциального уравнения Гельмгольца на равномерной сетке. Есть и другие примеры, которые приводят нас к необходимости решать СЛАУ с разреженной матрицей.

Рассмотрим здесь простейший метод решения линейных систем - метод простых итераций.

Допустим, у вас есть следующая система уравнений:

$$\begin{cases} a_{11} \cdot x_1 + a_{12} \cdot x_2 + a_{13} \cdot x_3 = b_1 \\ a_{21} \cdot x_1 + a_{22} \cdot x_2 + a_{23} \cdot x_3 = b_2 \\ a_{31} \cdot x_1 + a_{32} \cdot x_2 + a_{33} \cdot x_3 = b_3 \end{cases}$$

Здесь  $x_1, x_2, x_3$  - неизвестные, а  $a[i][j]$  и  $b[i]$  - известные постоянные

В матричном виде эту систему можно было бы записать так:

$$\begin{cases} a_{11} \cdot x_1 + a_{12} \cdot x_2 + a_{13} \cdot x_3 = b_1 \\ a_{21} \cdot x_1 + a_{22} \cdot x_2 + a_{23} \cdot x_3 = b_2 \\ a_{31} \cdot x_1 + a_{32} \cdot x_2 + a_{33} \cdot x_3 = b_3 \end{cases} \rightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Выразим из 1-го уравнения системы первую неизвестную  $x_1$

Выразим из 2-го уравнения системы вторую неизвестную  $x_2$

Выразим из 3-го уравнения системы третью неизвестную  $x_3$

$$\begin{cases} x_1 = \frac{b_1}{a_{11}} - \frac{1}{a_{11}}(a_{12} \cdot x_2 + a_{13} \cdot x_3) \\ x_2 = \frac{b_2}{a_{22}} - \frac{1}{a_{22}}(a_{21} \cdot x_1 + a_{23} \cdot x_3) \\ x_3 = \frac{b_3}{a_{33}} - \frac{1}{a_{33}}(a_{31} \cdot x_1 + a_{32} \cdot x_2) \end{cases}$$

Заметим, что элементы, лежащие на главной диагонали матрицы не должны быть нулевыми. Ну а если такое случается, то надо переставлять местами


уравнения, так, чтобы выполнялось условие неравенства нулю диагональных элементов.

Итак, все эти три уравнения мы можем записать общей формулой, если перейдем к индексам: первый индекс будет пробегать по строкам, а второй по столбцам. В нашей теории нумерация начинается с 1.

$$\begin{cases} x_1 = \frac{b_1}{a_{11}} - \frac{1}{a_{11}}(a_{12} \cdot x_2 + a_{13} \cdot x_3) \\ x_2 = \frac{b_2}{a_{22}} - \frac{1}{a_{22}}(a_{21} \cdot x_1 + a_{23} \cdot x_3) \rightarrow x_k = \frac{b_k}{a_{kk}} - \frac{1}{a_{kk}} \left( \sum_{\substack{i=1 \\ i \neq k}}^{column} a_{ki} \cdot x_i \right) \\ x_3 = \frac{b_3}{a_{33}} - \frac{1}{a_{33}}(a_{31} \cdot x_1 + a_{32} \cdot x_2) \end{cases}$$

Обратите внимание, что в скобках, где находится сумма произведений, не встречается элемента с совпадающими индексами (тот  $a[k][k]$  который лежит на диагонали), поэтому при суммировании мы должны его пропустить.

В принципе, этот пропуск можно реализовать конструкцией continue при программировании алгоритма. Но если вам это не нравится, то общую формулу суммы можно переписать в другом виде:

$$x_k = \frac{b_k}{a_{kk}} - \frac{1}{a_{kk}} \left( \sum_{\substack{i=1 \\ i \neq k}}^{column} a_{ki} \cdot x_i \right)$$


$$x_k = \frac{b_k}{a_{kk}} - \frac{1}{a_{kk}} \left( \sum_{i=1}^{k-1} a_{ki} \cdot x_i + \sum_{i=k+1}^{columns} a_{ki} \cdot x_i \right)$$

Таким образом мы тоже можем пропустить ненужный элемент, не используя дополнительных конструкций if при программировании.

Постараюсь сделать схему решения еще более понятной для программирования:

$$x[k] = \frac{b[k]}{a[k][k]} - \frac{1}{a[k][k]} \left( \sum_{\substack{i=1 \\ i \neq k}}^{columns} a[k][i] \cdot x[i] \right)$$

$$x[k] = \frac{b[k]}{a[k][k]} - \frac{1}{a[k][k]} \left( \sum_{i=1}^{k-1} a[k][i] \cdot x[i] + \sum_{i=k+1}^{columns} a[k][i] \cdot x[i] \right)$$

**columns** - количество столбцов

**k** - текущий корень системы, совпадает с номером строки матрицы

**i** - номер столбца, индекс, по которому проводится суммирование

**x[i]** - старые корни, полученные на предыдущей итерации

**x[k]** - новый корень, который считается на базе корней, полученных на предыдущей итерации

Важные уточнения:

columns - количество столбцов

k - текущий корень системы, совпадает с номером строки матрицы

i - номер столбца, индекс, по которому проводится суммирование

x[i] - старые корни, полученные на предыдущей итерации

x[k] - новый корень, который считается на базе корней, полученных на предыдущей итерации

Ещё хочется обратить ваше внимание на то, что во многих языках программирования принято начинать нумерацию с нуля (!). Поэтому в наших формулах произойдет сдвиг индекса, это нужно не забыть исправить, чтобы не выйти за диапазон изменения допустимых значений индекса при записи решения в массив / список / вектор.

Иногда, в книгах по численным методам еще используется верхний индекс (чаще всего в скобках, чтоб не спутали со степенью), чтобы разграничить где решение на новой итерации, а где решение на предыдущей итерации.

У нас получается рекуррентная формула:

$$x_{[k]}^{(n+1)} = \frac{b_{[k]}}{a_{[k][k]}} - \frac{1}{a_{[k][k]}} \left( \sum_{\substack{i=1 \\ i \neq k}}^{columns} a_{[k][i]} \cdot x_{[i]}^{(n)} \right)$$

k-й корень на новой (n+1)-й итерации
i-й корень на старой n-й итерации

$$x_{[k]}^{(n+1)} = \frac{b_{[k]}}{a_{[k][k]}} - \frac{1}{a_{[k][k]}} \left( \sum_{i=1}^{k-1} a_{[k][i]} \cdot x_{[i]}^{(n)} + \sum_{i=k+1}^{columns} a_{[k][i]} \cdot x_{[i]}^{(n)} \right)$$

На мой взгляд, не нужно особо обращать внимание на верхние индексы, показывающие где новая итерация, а где старая. Так как начинающих это только сбивает с толку. В реальной программе это будет перезапись одной и той же переменной, БЕЗ как либо верхних индексов.

Приведу очень простой пример, почему на практике не нужны загромождения верхними индексами для рекуррентной формулы:

Вот так рекуррентная формула выглядит в теории:  $x^{(n+1)} = d + x^{(n)}$

Вот так она выглядит на практике при реализации на каком-либо языке программирования:  $x = d + x$

Теперь опять вернемся к нашей задаче. Итерационный процесс можно запустить до бесконечности и далее... Шучу, нельзя. Компилятор быстро спустит вас с небес на землю! Но когда останавливаться? Теория говорит нам, что можно остановиться при достижении необходимой точности.

Точность часто обозначается буквой экспилон  $\varepsilon$ , ну а в программировании взяли за правило обозначать `eps` (сокращение от `epsilon`), так как греческими буквами нам нельзя пользоваться при написании названий переменных.

Для остановки и проверки точности, а также завершения вычислений на основе вычисления расстояния между соседними элементами в последовательности итераций можно воспользоваться Евклидовой метрикой.

В этом случае, условие завершения вычислений следующее:

$$\frac{\sqrt{\sum_{i=1}^n \left( x_{[k]}^{(n+1)} - x_{[k]}^{(n)} \right)^2}}{\sqrt{\sum_{i=1}^n \left( x_{[k]}^{(n+1)} \right)^2}} < \varepsilon$$

Евклидова метрика для определения достижения необходимой точности

По сути, это что-то вроде относительной погрешности, учитывающий все решения, а именно среднеквадратичные отклонения, чтобы исключить влияние знака. Потому что при сложении абсолютных разниц у нас могла произойти иллюзия достижения необходимой точности и иллюзия маленького разброса, если бы одна разница была бы большой со знаком "+", а другая разница примерно такой же большой, но со знаком "-". Они бы просто друг друга аннигилировали и дали бы нам ложный результат. Именно поэтому используются квадраты отклонений, а потом уже накладывается корень.

Для того, чтобы мы могли считать Евклидову метрику на каждой итерации, нам необходимо сохранять вектор решения на предыдущей итерации. То есть считая новый набор корней на новой итерации, мы должны сохранить старый набор корней на предыдущей итерации, чтобы было потом с чем сравнивать.

На практике же, я бы посоветовал помимо погрешности ограничивать выполнения цикла каким-либо предельным количеством итераций. Например, чтобы максимальное число итераций было 100~500. Для адекватной сходимости этого достаточно за глаза. Так как, если метод сходится, то система достигает нужной точности примерно за 20-50 итераций (чаще всего, если точность  $\epsilon_{ps} = 0.0001$ ).

Уже совсем другое дело, если решение расходится. Именно тогда вас спасает ограничение по числу итераций, чтобы не войти в бесконечный цикл.

Следует отметить, что скорость сходимости итерационного процесса выше для матриц, у которых элементы главной диагонали велики по сравнению с внедиагональными элементами. В связи с этим, перед началом численного решения задачи желательно преобразовать систему уравнений так, чтобы преобладали диагональные элементы.

Также, чтобы запустить рекуррентную формулу и начать любой метод итераций, нам нужно начальное приближение корней. Можно взять любое, но в разумных пределах. К примеру, можно использовать столбец свободных коэффициентов в матричном представлении системы. Или просто обнулить начальное приближение. Или везде поставить единицы. Если метод будет сходиться, то он в любом случае придет к корням с нужным приближением.

Порядок решения СЛАУ методом Якоби такой:

- Приведение системы уравнений к виду, в котором на каждой строчке выражено какое-либо неизвестное значение системы.



- Произвольный выбор нулевого решения, в качестве него можно взять вектор-столбец свободных членов.
- Производим подстановку произвольного нулевого решения в систему уравнений, полученную под пунктом 1.
- Осуществление дополнительных итераций, для каждой из которых используется решение, полученное на предыдущем этапе.

## **ОПИСАНИЕ МЕТОДА ЗЕЙДЕЛЯ**

Метод Зейделя – это итерационный метод решения систем линейных уравнений (СЛАУ), который является улучшением метода простых итераций.

В методе Зейделя для нахождения решения системы линейных уравнений используется последовательное приближение. Изначально задается начальное приближение для решения системы. Затем производятся итерации, на каждом шаге которых вычисляется новое приближение решения.

На каждой итерации метода Зейделя, значения неизвестных изменяются по очереди, начиная с первой неизвестной, и используются уже вычисленные на данной итерации значения предыдущих неизвестных. Таким образом, каждое новое приближение определяется с учетом уже вычисленных значений.

### **Условия для сходимости метода Зейделя**

Метод Зейделя сходится для симметричных, положительно определенных матриц. В случае сходимости, метод Зейделя является более быстрым, чем метод простых итераций. Однако, в некоторых случаях может потребоваться больше итераций, чем в методе простых итераций.

### **Недостатки и преимущества метода Зейделя**

Преимущества метода Зейделя:

Сходимость: метод Зейделя сходится для симметричных, положительно определенных матриц.

Более быстрый, чем метод простых итераций: метод Зейделя обычно требует меньше итераций для достижения заданной точности, чем метод простых итераций.

Эффективность на разреженных матрицах: метод Зейделя может быть эффективен для разреженных матриц, поскольку он обычно приводит к меньшему числу вычислений, чем метод простых итераций.

Недостатки метода Зейделя:

Не гарантирует сходимость для несимметричных матриц: метод Зейделя может не сходиться для несимметричных матриц.

Зависимость от начального приближения: качество решения методом Зейделя может сильно зависеть от выбора начального приближения.

Неэффективность на матрицах с диагональным преобладанием: метод Зейделя может быть неэффективным для матриц с диагональным преобладанием, так как он может сходиться медленно в этом случае.

Неустойчивость при наличии вырожденных собственных значений: метод Зейделя может быть неустойчивым, если у матрицы есть вырожденные собственные значения.

Не подходит для больших матриц: метод Зейделя может потребовать большого объема памяти для хранения матрицы, что делает его неэффективным для больших матриц.

## МЕТОД НАИМЕНЬШИХ НЕВЯЗОК

Метод минимальных невязок Метод минимальных невязок основан на таком же подходе, что и метод наискорейшего спуска, но значения параметра  $\alpha$  выбираются другим образом. Рассмотрим последующее приближение в виде:  $\vec{u}^{(u+1)} = \vec{u}^{(u)} - \tau \vec{r}^u$ . Значение  $\tau$  будем выбирать из условия  $\|\vec{r}^{u+1}\| \rightarrow \min$ . Будем минимизировать не функционал, а невязку системы уравнений. Для этого перепишем исходное уравнение в терминах невязок. Умножим обе части уравнения на матрицу  $A$ .  $A \vec{u}^{(u+1)} = A \vec{u}^{(u)} - \tau A \vec{r}^u$ ,  $A \vec{u}^{(u+1)} - \vec{f}^{u+1} = A \vec{u}^{(u)} - \vec{f}^{u+1} - \tau A \vec{r}^u$ . Откуда следует:  $\vec{r}^{u+1} = \vec{r}^u - \tau A \vec{r}^u$ .

Значение  $\tau_i$  выбирается из условия минимума нормы  $\|\vec{r}_{i+1}\|$ .  $(\vec{r}_{i+1}, \vec{r}_{i+1}) = \|\vec{r}_{i+1}\|^2 = (\vec{r}_i - \tau_i A \vec{r}_i, \vec{r}_i - \tau_i A \vec{r}_i) = (\vec{r}_i, \vec{r}_i) - \tau_i (A \vec{r}_i, \vec{r}_i) - \tau_i (\vec{r}_i, A \vec{r}_i) + \tau^2 (A \vec{r}_i, A \vec{r}_i)$ . Обозначим  $a = (A \vec{r}_i, A \vec{r}_i)$ ,  $b = -2(A \vec{r}_i, \vec{r}_i)$ . Тогда для значения параметра  $\tau_i$  получим:  $\tau_i = -b/2a = (A \vec{r}_i, \vec{r}_i) / (A \vec{r}_i, A \vec{r}_i)$ .

Преимущество данных методов – они не используют никакой дополнительной информации об операторе  $A$ , т.е.  $\gamma_1$  и  $\gamma_2$ , входящие в оценку  $\gamma_1 E \leq A \leq \gamma_2 E$  и необходимые для выбора  $\tau_0$ , здесь не требуются. Рассмотрим методы минимальных невязок и скорейшего спуска.

### 1. Метод минимальных невязок.

$$\frac{x^{k+1} - x^k}{\tau_{k+1}} + Ax^k = f, \quad k = 0, 1, \dots \quad (4.13)$$

Для  $r^k = f - Ax^k$  получим равенство, умножив обе части равенства (4.13) на матрицу  $A$ :

$$\frac{Ax^{k+1} - Ax^k}{\tau_{k+1}} + AAx^k = Af$$

Меняя знаки и группируя слагаемые соответствующим образом, получаем:

$$\frac{(-Ax^{k+1} + f) - (-Ax^k + f)}{\tau_{k+1}} + A(-Ax^k + f) + Af = Af$$

или

$$\frac{r^{k+1} - r^k}{\tau_{k+1}} + Ar^k = 0.$$

Параметр  $\tau_{k+1}$  будем выбирать из условия минимума невязки  $r^{k+1}$  по норме

$$r^{k+1} = r^k - \tau_{k+1} Ar^k.$$

$$\begin{aligned} \phi(\tau_{k+1}) &\equiv \|r^{k+1}\|^2 = \|r^k - \tau_{k+1} Ar^k\|^2 = ((r^k - \tau_{k+1} Ar^k), (r^k - \tau_{k+1} Ar^k)) = \\ &= \|r^k\|^2 - 2\tau_{k+1} (Ar^k, r^k) + (\tau_{k+1})^2 \|Ar^k\|^2. \end{aligned}$$

Продифференцируем  $\phi(\tau_{k+1})$  по  $\tau_{k+1}$ , получим

$$-2(Ar^k, r^k) + 2\tau_{k+1} \|Ar^k\|^2 = 0,$$

$$\tau_{k+1} = \frac{(Ar^k, r^k)}{\|Ar^k\|^2} = \frac{(Ar^k, r^k)}{(Ar^k, Ar^k)} \quad (4.14)$$

## 2. Метод скорейшего спуска.

Получается из условия минимума энергетической нормы погрешности  $\|Z^{k+1}\|_A^2 = (AZ^{k+1}, Z^{k+1})$ , где  $Z^{k+1} = Z^k - \tau X$ ,  $X$  – точное решение исходной системы. Поскольку  $AZ^k = AX^k - AX = r^k$ , и учитывая, что

$$\frac{Z^{k+1} - Z^k}{\tau_{k+1}} + AZ^k = 0 \quad \text{или} \quad Z^{k+1} = Z^k - \tau_{k+1} AZ^k, \quad \text{получим}$$

$$\begin{aligned} \|Z^{k+1}\|_A^2 &= (AZ^{k+1}, Z^{k+1}) = (AZ^k - \tau_{k+1} A^2 Z^k, Z^k - \tau_{k+1} AZ^k) = \\ &= (r^k - \tau_{k+1} Ar^k, Z^k - \tau_{k+1} r^k) = \\ &= (r^k, Z^k) - 2\tau_{k+1} (r^k, r^k) + \tau_{k+1}^2 (Ar^k, r^k) = \phi(\tau_{k+1}). \end{aligned}$$

Дифференцируя  $\|Z^{k+1}\|_A^2$  по  $\tau_{k+1}$ , получим

$$\phi'(\tau_{k+1}) = -2(r^k, r^k) + 2\tau_{k+1} (Ar^k, r^k), \quad \text{откуда}$$

$$\tau_{k+1} = \frac{(r^k, r^k)}{(Ar^k, r^k)}. \quad (4.15)$$

## МЕТОД НАЙСКОРЕЙШЕГО СПУСКА

### Метод наискорейшего спуска

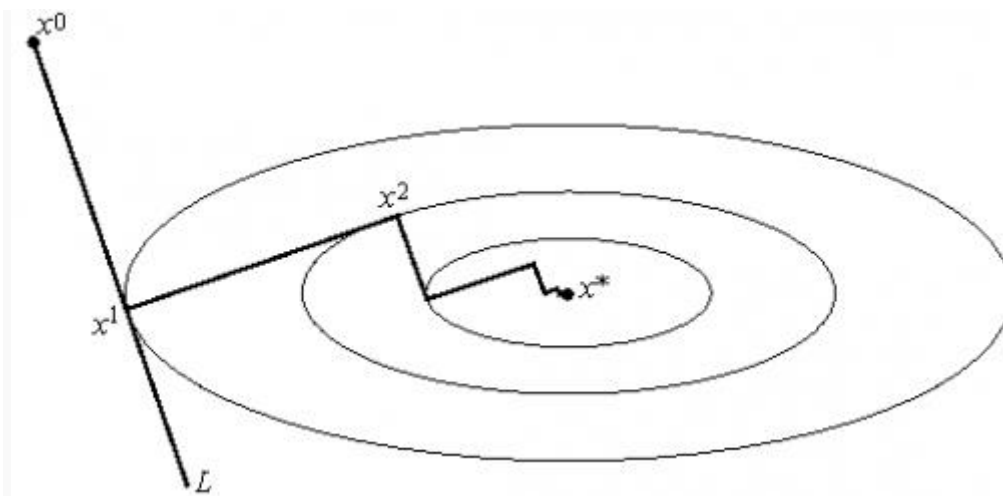


Рис.3 Геометрическая интерпретация метода наискорейшего спуска. На каждом шаге  $\lambda^{[k]}$  выбирается так, чтобы следующая итерация была точкой минимума функции  $f$  на луче  $L$ .

Этот вариант градиентного метода основывается на выборе шага из следующего соображения. Из точки  $x^{[k]}$  будем двигаться в направлении антиградиента до тех пор пока не достигнем минимума функции  $f$  на этом направлении, т. е. на луче  $L = \{x = x^{[k]} - \lambda f'(x^{[k]}); \lambda \geq 0\}$ :

$$\lambda^{[k]} = \arg \min_{\lambda \in [0, \infty)} f(x^{[k]} - \lambda f'(x^{[k]}))$$

Другими словами,  $\lambda^{[k]}$  выбирается так, чтобы следующая итерация была точкой минимума функции  $f$  на луче  $L$  (см. рис. 3). Такой вариант градиентного метода называется методом наискорейшего спуска. Заметим, кстати, что в этом методе направления соседних шагов ортогональны.

Метод наискорейшего спуска требует решения на каждом шаге задачи одномерной оптимизации. Практика показывает, что этот метод часто требует меньшего числа операций, чем градиентный метод с постоянным шагом.

В общей ситуации, тем не менее, теоретическая скорость сходимости метода наискорейшего спуска не выше скорости сходимости градиентного метода с постоянным (оптимальным) шагом

Метод наискорейшего спуска (в англ. литературе «method of steepest descent») - это итерационный численный метод (первого порядка) решения оптимизационных задач, который позволяет определить экстремум (минимум или максимум) целевой функции:

значения аргумента функции (управляемые параметры) на вещественной области.

В соответствии с рассматриваемым методом экстремум (максимум или минимум) целевой функции определяют в направлении наиболее быстрого возрастания (убывания) функции, т.е. в направлении градиента (антиградиента) функции. Градиентом функции в точке называется вектор, проекциями которого на координатные оси являются частные производные функции по координатам:

где  $i, j, \dots, n$  - единичные векторы, параллельные координатным осям.

Градиент в базовой точке строго ортогонален к поверхности, а его направление показывает направление наискорейшего возрастания функции, а противоположное направление (антиградиент), соответственно, показывает направление наискорейшего убывания функции.

Метод наискорейшего спуска является дальнейшим развитием метода градиентного спуска. В общем случае процесс нахождения экстремума функции является итерационной процедурой, которая записывается следующим образом:

где знак «+» используется для поиска максимума функции, а знак «-» используется для поиска минимума функции;

- единичный вектор направления, который определяется по формуле:

- модуль градиента определяет скорость возрастания или убывания функции в направлении градиента или антиградиента:

- константа, определяющая размеры шага и одинаковая для всех  $i$ -х направлений.

Величина шага выбирается из условия минимума целевой функции  $f(x)$  в направлении движения, т. е. в результате решения задачи одномерной оптимизации в направлении градиента или антиградиента:

Другими словами, величину шага определяют при решении данного уравнения:

## Методика расчета

**1 шаг:** Определение аналитических выражения (в символьном виде) для вычисления градиента функции  $(\nabla f(x_1, \dots, x_n))$

$$\nabla f(x_1, \dots, x_n) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x_1, \dots, x_n) \\ \dots \\ \frac{\partial}{\partial x_n} f(x_1, \dots, x_n) \end{bmatrix}$$

**2 шаг:** Задаем начальное приближение  $X = \{x_1, \dots, x_n\}$

*Далее выполняется итерационный процесс.*

**3 шаг:** Определяется необходимость рестарта алгоритмической процедуры для обнуления последнего направления поиска. В результате рестарта поиск осуществляется заново в направлении скорейшего спуска.

**4 шаг:** Вычисление координат единичного вектора по представленным формулам

$$S_k = \frac{\nabla f(X^k)}{\|\nabla f(X^k)\|}$$

$$\|\nabla f(x_1, x_2, \dots, x_n)_k\| = \sqrt{\sum_{k=1}^n \left( \frac{\partial}{\partial x_k} f(x_1, x_2, \dots, x_n)_k \right)^2}$$

**5 шаг:** определяем шаг расчета из условия поиска экстремума для следующей функции (решения задачи одномерной оптимизации).

$$\lambda_k \Rightarrow f(x_k \pm \lambda \cdot S_k(x_k)) \rightarrow \text{extr}$$

**6 шаг:** Определяем новые значения аргументов функции после выполнения k-го шага расчета:

$$X_{k+1} = X_k \pm \lambda_k \cdot P_k$$

где знак «+» используется для поиска максимума функции, а знак «-» используется для поиска минимума функции;

**7 шаг:** проверяем критерии останова итерационного процесса. Вычислительный процесс заканчивается, когда будет достигнута точка, в которой оценка градиента будет равна нулю (коэффициенты функции отклика становятся незначимыми). В противном случае возвращаемся к шагу 3 и продолжаем итерационный расчет

Решение метода Якоби для СЛАУ программная реализация Mathcad

Решение СЛАУ программного модуля методом Якоби.

$$\underline{A} := \begin{pmatrix} 4 & 2 & -1 \\ 2 & 5 & 1 \\ -1 & 1 & 3 \end{pmatrix} \quad \underline{f} := \begin{pmatrix} -1 \\ -1 \\ -8 \end{pmatrix} \quad \underline{x} := \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

$\text{eps} := 0.000001$



```

Yakobi(A,f,x,eps) :=
  n ← rows(A) - 1
  num ← 0
  while |A · x - f| > eps
    
$$y_0 \leftarrow \frac{f_0 - \sum_{j=1}^n (A_{0,j} \cdot x_j)}{A_{0,0}}$$

    for i ∈ 1..n - 1
      
$$y_i \leftarrow \frac{f_i - \left[ \sum_{j=0}^{i-1} (A_{i,j} \cdot x_j) + \sum_{j=i+1}^n (A_{i,j} \cdot x_j) \right]}{A_{i,i}}$$

    
$$y_n \leftarrow \frac{f_n - \sum_{j=0}^{n-1} (A_{n,j} \cdot x_j)}{A_{n,n}}$$

    x ← y
    num ← num + 1
  ( x
    num )

```

rez := Yakobi(A,f,x,eps)

X := rez<sub>0</sub>

num := rez<sub>1</sub>

$$f - A \cdot X = \begin{pmatrix} -4.905 \times 10^{-7} \\ 5.363 \times 10^{-7} \\ -3.429 \times 10^{-7} \end{pmatrix} \blacksquare$$

# Решение метода Зейделя для СЛАУ программная реализация Mathcad

```

Zeidel (A,f ,x,eps) :=
    n ← rows(A) - 1
    num ← 0
    while |A · x - f| > eps
        
$$x_0 \leftarrow \frac{f_0 - \sum_{j=1}^n (A_{0,j} \cdot x_j)}{A_{0,0}}$$

        for i ∈ 1..n - 1
            
$$x_i \leftarrow \frac{f_i - \left[ \sum_{j=0}^{i-1} (A_{i,j} \cdot x_j) + \sum_{j=i+1}^n (A_{i,j} \cdot x_j) \right]}{A_{i,i}}$$

        
$$x_n \leftarrow \frac{f_n - \sum_{j=0}^{n-1} (A_{n,j} \cdot x_j)}{A_{n,n}}$$

        num ← num + 1
    
$$\begin{pmatrix} x \\ num \end{pmatrix}$$

```

rez := Zeidel(A,f ,x ,eps)

X := rez<sub>0</sub>

num := rez<sub>1</sub>

$$X = \begin{pmatrix} -1.8 \\ 1.257 \\ -3.686 \end{pmatrix} \quad num = 21$$

$$f - A \cdot X = \begin{pmatrix} -6.524 \times 10^{-7} \\ 1.953 \times 10^{-7} \\ 0 \end{pmatrix}$$

```
import numpy as np

def jacoby(A, b, max_iter, eps):
    # Создаем начальное приближение
    x0 = np.array([0, 0, 0, 0])

    # Получаем размеры матрицы A
    n, m = A.shape

    # Создаем диагональную матрицу D и матрицу R
    D = np.zeros((n, n))
    R = np.zeros((n, n))

    # Заполняем матрицы D и R
    for i in range(n):
        for j in range(n):
            if i == j:
                D[i][j] = A[i][j]
            else:
                R[i][j] = A[i][j]
    # Вычисляем обратную диагональную матрицу D_inv
    D_inv = np.linalg.inv(D)

    # Создаем матрицу T и вектор c
    T = -D_inv.dot(R)
    c = D_inv.dot(b)

    # Начинаем итерационный процесс
    x = x0
    for i in range(max_iter):
        x_new = T.dot(x) + c
        if np.linalg.norm(x_new - x) < eps:
            break
        x = x_new

    return x

# Создаем матрицу коэффициентов
A = np.array([[4, -1, 0, 3],
              [1, 15.5, 3, 8],
              [0, -1.3, -4, 1.1],
              [14, 5, -2, 30]])

# Создаем вектор правой части
b = np.array([1, 1, 1, 1])

# Максимальное число итераций
max_iter = 1000

# Задаем точность вычислений
eps = 0.01

x = jacoby(A, b, max_iter, eps)

print("Solution:")
```

```
print(x)
```

Решение программного модуля методом Зейделя.

```
1. import math
2. import copy
3.
4. # Пробные данные для уравнения  $A \cdot X = B$ 
5. #  $x = [1.10202, 0.99091, 1.01111]$ 
6.
7. a = [[10, 1, -1],
8.      [1, 10, -1],
9.      [-1, 1, 10]]
10.
11. b = [11, 10, 10]
12.
13. # Проверка матрицы коэффициентов на корректность
14. def isCorrectArray(a):
15.     for row in range(0, len(a)):
16.         if( len(a[row]) != len(b) ):
17.             print('Не соответствует размерность')
18.             return False
19.
20.     for row in range(0, len(a)):
21.         if( a[row][row] == 0 ):
22.             print('Нулевые элементы на главной диагонали')
23.             return False
24.     return True
25.
26.
27. # Условие завершения программы на основе вычисления
28. # расстояния между соответствующими элементами соседних
29. # итераций в методе решения
30. def isNeedToComplete(x_old, x_new):
31.     eps = 0.0001
32.     sum_up = 0
33.     sum_low = 0
34.     for k in range(0, len(x_old)):
35.         sum_up += ( x_new[k] - x_old[k] ) ** 2
36.         sum_low += ( x_new[k] ) ** 2
37.
38.     return math.sqrt( sum_up / sum_low ) < eps
39.
40. # Процедура решения
41. def solution(a, b):
42.     if( not isCorrectArray(a) ):
43.         print('Ошибка в исходных данных')
44.     else:
45.         count = len(b) # количество корней
46.
47.         x = [1 for k in range(0, count) ] # начальное приближение корней
48.
49.         numberOfIter = 0 # подсчет количества итераций
50.         MAX_ITER = 100 # максимально допустимое число итераций
51.         while( numberOfIter < MAX_ITER ):
52.
53.             x_prev = copy.deepcopy(x)
54.
55.             for k in range(0, count):
56.                 S = 0
57.                 for j in range(0, count):
58.                     if( j != k ): S = S + a[k][j] * x[j]
59.                 x[k] = b[k]/a[k][k] - S / a[k][k]
60.
61.             if isNeedToComplete(x_prev, x) : # проверка на выход
62.                 break
63.
```

```

64.         numberOfIter += 1
65.
66.         print('Количество итераций на решение: ', numberOfIter)
67.
68.         return x
69.
70.
71. # MAIN - блок программы
72. print( 'Решение: ', solution(a, b) ) # Вызываем процедуру решение

```

Метод невязок реализация на языке программирования Python.

```

import numpy as np
import math

def max(n, m):
    maximum = a1[n][0]
    iconst = 0
    for i in range(1, m - 1):
        if abs(a1[n][i]) > abs(maximum):
            maximum = a1[n][i]
            iconst = i
    return iconst

def print_matrix(a):
    for i in range(len(a)):
        for j in range(len(a[i])):
            print("%2.4f" % (a[i][j]), end=' ')
        print()

a = np.array([[1.28, 0.42, 0.54, 1.00, 1.34],
              [2.11, 3.01, 4.02, 0.22, 1.56],
              [0.18, 3.41, 0.15, 1.43, 1.78],
              [2.14, 0.17, 0.26, 0.18, 1.91]], dtype=np.float32)
a1 = a.copy()
a2 = np.array([[1.28, 0.42, 0.54, 1.00],
              [2.11, 3.01, 4.02, 0.22],
              [0.18, 3.41, 0.15, 1.43],
              [2.14, 0.17, 0.26, 0.18]], dtype=np.float32)
x = np.zeros((4, 1))
x1 = np.zeros((4, 1))
permutations = np.array([1, 2, 3, 4])
t = 0
sum_coefficient = 1

# прямой ход
while (t != a1.shape[0] - 1):
    y = max(t, a1.shape[1])
    if (y != t):
        for i in range(0, a1.shape[0]):
            a1[i][t], a1[i][y] = a1[i][y], a1[i][t]
            permutations[t], permutations[y] = permutations[y], permutations[t] #
запоминаем перестановки
        coefficient = 0
        for i in range(t + 1, a1.shape[0]):
            coefficient = a1[t][t] / a1[i][t]
            sum_coefficient *= coefficient
            for j in range(t, a1.shape[1]):
                a1[i][j] = a1[i][j] * coefficient - a1[t][j]
        t += 1

```

```

# Обратный ход
for i in range(a1.shape[0] - 1, -1, -1):
    x[i] = (a1[i][-1] - sum([a1[i][j] * x[j] for j in range(i + 1, a1.shape[1] - 1)])) / a1[i][i]

# восстанавливаем порядок
for i in range(0, 4):
    x1[permutations[i] - 1] = x[i]

for i in range(0, 4):
    print("x", i + 1, "=", "%.4f" % (x1[i]))

# невязка
temp = np.zeros((4, 1))
discrepancy = np.zeros((4, 1))
for i in range(a.shape[0]):
    discrepancy[i] = a[i][-1] - sum([a[i][j] * x1[j] for j in range(0, a.shape[1] - 1)])
print("Вектор невязки")
for i in range(0, 4):
    print("d", i + 1, "=", "%.16f" % (discrepancy[i]))

# определитель
print("Определитель матрицы")
det = 1
for i in range(0, 4):
    det *= a1[i][i]
print("det=", "%.4f" % abs(det / (sum_coefficient)))

# обратная матрица
inv = np.zeros((4, 4))
for k in range(0, 4):
    a3 = a.copy()
    for j in range(0, 4):
        if (k == j):
            a3[j][4] = 1
        else:
            a3[j][4] = 0
    t = 0
    permutations = np.array([1, 2, 3, 4])

# прямой ход
while (t != a3.shape[0] - 1):
    y = max(t, a3.shape[1])
    if (y != t):
        for i in range(0, a3.shape[0]):
            a3[i][t], a3[i][y] = a3[i][y], a3[i][t]
        permutations[t], permutations[y] = permutations[y], permutations[t] #
запоминаем перестановки
    coefficient = 0
    for i in range(t + 1, a3.shape[0]):
        coefficient = a3[t][t] / a3[i][t]
        for j in range(t, a.shape[1]):
            a3[i][j] = a3[i][j] * coefficient - a3[t][j]
    t += 1

# обратный ход
for i in range(a3.shape[0] - 1, -1, -1):
    x[i] = (a3[i][-1] - sum([a3[i][j] * x[j] for j in range(i + 1, a3.shape[1] - 1)])) / a3[i][i]

# восстанавливаем порядок

```

```

for i in range(0, 4):
    x1[permutations[i] - 1] = x[i]
for i in range(0, 4):
    inv[i][k] = x1[i]
print("Обратная матрица")
print_matrix(inv)
print("Проверка")
c = a2.dot(inv)
print_matrix(c)

```

Метод скорейшего спуска (градиентный) реализация на языке программирования Python.

```

import random
1. import numpy as np
2. import matplotlib.pyplot as plt
3.
4.
5. """
6. Метод наискорейшего спуска
7.  Функция Розенброка
8.  Функция  $f(x) = 100 * (x(2) - x(1)^2)^2 + (1 - x(1))^2$ 
9.  Градиент  $g(x) = (-400 * (x(2) - x(1)^2) * x(1) - 2 * (1 - x(1)), 200 * (x(2) - x(1)^2))$ 
10.  """
11.
12.
13.  def goldsteinsearch(f,df,d,x,alpham,rho,t):
14.      '''
15.          Функция линейного поиска
16.          Число f, производная df, текущая точка итерации x и
            текущее направление поиска d
17.      '''
18.      flag = 0
19.
20.      a = 0
21.      b = alpham
22.      fk = f(x)
23.      gk = df(x)
24.
25.      phi0 = fk
26.      dphi0 = np.dot(gk, d)
27.      # print(dphi0)
28.      alpha=b*random.uniform(0,1)
29.
30.      while(flag==0):

```

```

31.         newfk = f(x + alpha * d)
32.         phi = newfk
33.         # print(phi,phi0,rho,alpha ,dphi0)
34.         if (phi - phi0) <= (rho * alpha * dphi0):
35.             if (phi - phi0) >= ((1 - rho) * alpha * dphi0):
36.                 flag = 1
37.             else:
38.                 a = alpha
39.                 b = b
40.                 if (b < alphas):
41.                     alpha = (a + b) / 2
42.                 else:
43.                     alpha = t * alpha
44.         else:
45.             a = a
46.             b = alpha
47.             alpha = (a + b) / 2
48.     return alpha
49.
50.
51.     def rosenbrock(x):
52.         # Функция:  $f(x) = 100 * (x(2) - x(1)^2)^2 + (1 - x(1))^2$ 
53.         return 100*(x[1]-x[0]**2)**2+(1-x[0])**2
54.
55.
56.     def jacobian(x):
57.         # Градиент  $g(x) = (-400 * (x(2) - x(1)^2) * x(1) - 2 * (1 - x(1)), 200 * (x(2) - x(1)^2))$ 
58.         return np.array([-400*x[0]*(x[1]-x[0]**2)-2*(1-x[0]),200*(x[1]-x[0]**2)])
59.
60.
61.
62.     def steepest(x0):
63.
64.         print ('Начальная точка:')
65.         print(x0,'\n')
66.         imax = 20000
67.         W = np.zeros((2, imax))
68.         epo=np.zeros((2, imax))
69.         W[:, 0] = x0
70.         i = 1
71.         x = x0
72.         grad = jacobian(x)
73.         delta = sum (grad ** 2) # начальная ошибка
74.

```



```

75.         f = open ("Самый быстрый.txt", 'w')
76.
77.     while i < imax and delta > 10 ** (-5):
78.         p = -jacobian(x)
79.         x0 = x
80.         alpha = goldsteinsearch(rosenbrock, jacobian, p, x,
1.     0.1, 2)
81.         x = x + alpha * p
82.         W[:, i] = x
83.         if i % 5 == 0:
84.
85.             epo[:,i] =np.array((i,delta))
86.             f.write(str(i)+"          "+str(delta)+"\n")
87.             print(i,np.array((i,delta)))
88.         grad = jacobian(x)
89.         delta = sum(grad ** 2)
90.         i = i + 1
91.
92.         print ("Количество итераций:", i)
93.         print ("Приблизительное оптимальное решение:")
94.     print(x, '\n')
95.         W = W[:, 0: i] # Запись точки итерации
96.
97.     return [W,epo]
98.
99.     if __name__=="__main__":
100.         X1 = np.arange(-1.5, 1.5 + 0.05, 0.05)
101.         X2 = np.arange(-3.5, 4 + 0.05, 0.05)
102.         [x1, x2] = np.meshgrid(X1, X2)
103.         f = 100 * (x2-x1 ** 2) ** 2 + (1-x1) ** 2 #
    заданная функция
104.         plt.contour (x1, x2, f, 20) # рисует 20 контурных
    линий функции
105.
106.         x0 = np.array([-1.2, 1])
107.         list_out = steepest(x0)
108.         W=list_out[0]
109.
110.         epo=list_out[1]
111.
112.         plt.plot (W [0,:], W [1,:], 'g * -') # Рисуем
    траекторию схождения точки итерации
113.
114.         plt.show()

```

Работающий код с графическим выражением значений и ответом.

```
import numpy as np
import matplotlib.pyplot as plot

radius = 8                                # working plane radius
global_epsilon = 0.000000001              # argument increment for
derivative
centre = (global_epsilon, global_epsilon)  # centre of the working circle
arr_shape = 100                            # number of points processed /
360
step = radius / arr_shape                  # step between two points

def differentiable_function(x, y):
    return np.sin(x) * np.exp((1 - np.cos(y)) ** 2) + \
           np.cos(y) * np.exp((1 - np.sin(x)) ** 2) + (x - y) ** 2

def rotate_vector(length, a):
    return length * np.cos(a), length * np.sin(a)

def derivative_x(epsilon, arg):
    return (differentiable_function(global_epsilon + epsilon, arg) -
            differentiable_function(epsilon, arg)) / global_epsilon

def derivative_y(epsilon, arg):
    return (differentiable_function(arg, epsilon + global_epsilon) -
            differentiable_function(arg, epsilon)) / global_epsilon

def calculate_flip_points():
    flip_points = np.array([0, 0])
    points = np.zeros((360, arr_shape), dtype=bool)
    cx, cy = centre

    for i in range(arr_shape):
        for alpha in range(360):
            x, y = rotate_vector(step, alpha)
            x = x * i + cx
            y = y * i + cy
            points[alpha][i] = derivative_x(x, y) + derivative_y(y, x) > 0
            if not points[alpha][i - 1] and points[alpha][i]:
                flip_points = np.vstack((flip_points, np.array([alpha, i -
1])))

    return flip_points

def pick_estimates(positions):
    vx, vy = rotate_vector(step, positions[1][0])
    cx, cy = centre
    best_x, best_y = cx + vx * positions[1][1], cy + vy * positions[1][1]

    for index in range(2, len(positions)):
        vx, vy = rotate_vector(step, positions[index][0])
        x, y = cx + vx * positions[index][1], cy + vy * positions[index][1]
        if differentiable_function(best_x, best_y) >
differentiable_function(x, y):
```

```

        best_x = x
        best_y = y

    for index in range(360):
        vx, vy = rotate_vector(step, index)
        x, y = cx + vx * (arr_shape - 1), cy + vy * (arr_shape - 1)
        if differentiable_function(best_x, best_y) >
differentiable_function(x, y):
            best_x = x
            best_y = y

    return best_x, best_y

def gradient_descent(best_estimates, is_x):
    derivative = derivative_x if is_x else derivative_y
    best_x, best_y = best_estimates
    descent_step = step
    value = derivative(best_y, best_x)

    while abs(value) > global_epsilon:
        descent_step *= 0.95
        best_y = best_y - descent_step \
            if derivative(best_y, best_x) > 0 else best_y + descent_step
        value = derivative(best_y, best_x)

    return best_y, best_x

def find_minimum():
    return
gradient_descent(gradient_descent(pick_estimates(calculate_flip_points()),
False), True)

def get_grid(grid_step):
    samples = np.arange(-radius, radius, grid_step)
    x, y = np.meshgrid(samples, samples)
    return x, y, differentiable_function(x, y)

def draw_chart(point, grid):
    point_x, point_y, point_z = point
    grid_x, grid_y, grid_z = grid
    plot.rcParams.update({
        'figure.figsize': (4, 4),
        'figure.dpi': 200,
        'xtick.labelsize': 4,
        'ytick.labelsize': 4
    })
    ax = plot.figure().add_subplot(111, projection='3d')
    ax.scatter(point_x, point_y, point_z, color='red')
    ax.plot_surface(grid_x, grid_y, grid_z, rstride=5, cstride=5, alpha=0.7)
    plot.show()

if __name__ == '__main__':
    min_x, min_y = find_minimum()
    minimum = (min_x, min_y, differentiable_function(min_x, min_y))
    draw_chart(minimum, get_grid(0.05))

```

Вывод:

В практической работе было рассмотрено несколько новых способов работы с СЛАУ итерационными методами: метод Якоби, метод Зейделя, метод скорейших невязок, метод скорейшего спуска. Выяснилось, что максимальное число итераций в задаче 100~500. Для адекватной сходимости этого достаточно.

Возможно, если число уравнений до 300, а если их, скажем, 50000.

Лучше использовать, метод сопряженных градиентов (не надо ничего перенумеровывать, хранить матрицу в памяти и прочее). Единственное условие – положительная определенность матрицы и симметричность.

Методы градиентного спуска являются достаточно мощным инструментом решения задач оптимизации. Главным недостатком методов является ограниченная область применимости. В данном методе поиск происходит более крупными шагами, и градиент функции вычисляется в меньшем числе точек.

Для наискорейшего спуска, шаг расчета выбирается такой величины, что движение выполняется до тех пор, пока происходит улучшение функции, достигая, таким образом, экстремума в некоторой точке. В этой точке вновь определяют направление поиска (с помощью градиента) и ищут новую точку оптимума целевой функции и т.д. Таким образом, в данном методе поиск происходит более крупными шагами, и градиент функции вычисляется в меньшем числе точек.

## ПРИЛОЖЕНИЕ 1

### Результат метода Якоби

```
метод Якоби x
C:\Users\User\anaconda3\envs\pdalpy\python.exe "C:/1/,/python/fractal-master/метод Якоби.py"
Solution:
[-2.77717461e+180  3.67486445e+180  3.24161015e+180  1.38734521e+181]

Process finished with exit code 0
```

### Результат работы метода Зейделя

```
метод Зейделя x
C:\Users\User\anaconda3\envs\pdalpy\python.exe "C:/1/,/python/fractal-master/метод Зейделя.py"
Количество итераций на решение: 2
Решение: [1.1020231, 0.99090979, 1.011111331]

Process finished with exit code 0
```

## Метод скорейших невязок результат

```
.pytest_cache
метод невязок X
C:\Users\User\anaconda3\envs\pdaipy\python.exe "C:/1/,/python/fractal-master/метод невязок.py"
x 1 = 0.8873
x 2 = 0.3882
x 3 = -0.3818
x 4 = 0.2475
Вектор невязки
d 1 = 0.0000000000000002
d 2 = -0.00000000069059931
d 3 = -0.0000000594036695
d 4 = 0.0000000272731422
Определитель матрицы
det= 23.2761
Обратная матрица
-0.1005 -0.0222 0.0046 0.5489
-0.4891 0.0395 0.3073 0.2277
0.3565 0.2372 -0.2319 -0.4276
1.1415 -0.1163 -0.0097 -0.5673
Проверка
1.0000 0.0000 -0.0000 0.0000
0.0000 1.0000 0.0000 -0.0000
0.0000 0.0000 1.0000 -0.0000
0.0000 0.0000 -0.0000 1.0000

Process finished with exit code 0
```

## Метод скорейшего спуска ответ

```
[0.99664783 0.99329498]
```

## ОТВЕТЫ НА ВОПРОСЫ ПО ПРАКТИЧЕСКОЙ РАБОТЕ 2

### 1. КАКИЕ МЕТОДЫ РЕШЕНИЯ СЛАУ ВЫ ЗНАЕТЕ?

Метод Гаусса, метод Крамера, метод Гаусса-Жордана, матричный метод, метод прогонки, разложение Холецкого; Среди них итерационные методы: метод Якоби (метод простой итерации), метод Гаусса-Зейделя, метод релаксации, метод релаксации, многосеточный метод.

### 2. ЗАПИШИТЕ КАНОНИЧЕСКУЮ ФОРМУ ОДНОШАГОВЫХ (ДВУХСЛОЙНЫХ) ИТЕРАЦИОННЫХ СХЕМ.

**Каноническая форма одношаговых методов.** Наиболее употребительные одношаговые итерационные методы решения СЛАУ укладываются в единую схему, согласно которой итерационная последовательность  $\{x^N\}$ , построенная по такому методу, подчиняется уравнению общего вида

$$B_{N+1} \frac{x^{N+1} - x^N}{\tau_{N+1}} + Ax^N = b, \quad (11.5)$$

в котором  $\det B_{N+1} \neq 0$ ,  $N = 0, 1, \dots$

Уравнение (11.5) называют *канонической формой одношагового итерационного метода*. Выбор невырожденных матриц  $B_{N+1}$  характеризует конкретный метод, итерационный параметр  $\tau_{N+1}$  не является обязательным (его можно было бы учесть в матрице  $B_{N+1}$ ) и вводится в уравнение из соображений удобства. Его используют для поиска путей усиления конкретного метода с точки зрения сходимости итерационной последовательности и скорости этой сходимости.

В канонической форме одношагового итерационного метода изменение  $x^{N+1} - x^N$  текущего столбца связывается с невязкой  $b - Ax^N$ . Этот столбец в рассматриваемой выше СЛАУ. Чем меньше невязка, тем меньше изменение текущего столбца. Матрица  $B_{N+1}$  реализующая эту связь на  $N$ -м шаге, должна быть достаточно простой, так как, согласно канонической форме метода, для получения изменения  $x^{N+1} - x^N$  требуется вычисление обратной матрицы для  $B_{N+1}$ . Если это не так, то более простым может быть обращение матрицы  $A$ , и тогда прямой метод решения СЛАУ окажется более предпочтительным.

Если в канонической форме (11.5) одношагового итерационного метода матрица  $B_{N+1}$  является единичной, то итерационный метод называют *явным*, а иначе — *неявным*. Если матрицы  $B_{N+1}$  и итерационный параметр  $\tau_{N+1}$  от

номера итерации не зависят:  $B_{N+1} = B$ ,  $T_{N+1} = T$ , то метод называют стационарным.

**Методы Якоби и Зейделя.** Представим невырожденную матрицу  $A$  в виде суммы трех матриц

$$A = A_- + D + A_+, \quad (11.6)$$

где  $D$  — диагональная матрица;  $A_-$  и  $A_+$  — соответственно нижняя и верхняя треугольные матрицы с нулевыми элементами на диагонали. Такое представление существует и единственно, так как в каждой позиции только одна из складываемых матриц имеет ненулевой элемент, равный соответствующему элементу матрицы  $A$ . Матрица  $A_-$  содержит элементы  $A$ , расположенные под главной диагональю, матрица  $A_+$  — элементы над главной диагональю, а матрица  $D$  — диагональные.

**Пример 11.1.** Для квадратной матрицы

$$A = \begin{pmatrix} 1 & -3 & 1 & 2 \\ 2 & 0 & 6 & -1 \\ 3 & -3 & -2 & -7 \\ -1 & -2 & 4 & 5 \end{pmatrix}$$

четвертого порядка в представлении (11.6) имеем

$$A_- = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 3 & -3 & 0 & 0 \\ -1 & -2 & 4 & 0 \end{pmatrix}, \quad D = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & 5 \end{pmatrix},$$

$$A_+ = \begin{pmatrix} 0 & -3 & 1 & 2 \\ 0 & 0 & 6 & -1 \\ 0 & 0 & 0 & -7 \\ 0 & 0 & 0 & 0 \end{pmatrix}. \quad \#$$

В СЛАУ  $Ax = b$  вместо матрицы  $A$  подставим ее представление (11.6). Получим  $(A_- + D + A_+)x = b$ , откуда

$$Dx = -(A_- + A_+)x + b. \quad (11.7)$$

Если диагональные элементы матрицы  $A = (a_{ij})$  не равны нулю, то диагональная матрица  $D = \text{diag}(a_{11}, a_{22}, \dots, a_{nn})$  имеет обратную матрицу  $D^{-1} = \text{diag}(a_{11}^{-1}, a_{22}^{-1}, \dots, a_{nn}^{-1})$  — в этом случае систему (11.7) можно записать в виде

$$x = -D^{-1}(A_- + A_+)x + D^{-1}b. \quad (11.8)$$



Систему (11.8) можно использовать для построения итерационной последовательности

$$x^{N+1} = -D^{-1}(A_- + A_+)x^N + D^{-1}b, \quad N = 0, 1, \dots, \quad (11.9)$$

где  $x^0$  — произвольное начальное приближение. Вычисление решения СЛАУ при помощи указанной последовательности называют методом Якоби. Матричные соотношения (11.9) несложно записать в координатной форме. Обозначая номера строк и столбцов индексами внизу, получим

$$x_i^{N+1} = \frac{1}{a_{ii}} \left( - \sum_{j=1}^{i-1} a_{ij} x_j^N - \sum_{j=i+1}^n a_{ij} x_j^N + b_i \right), \quad i = \overline{1, n}. \quad (11.10)$$

Здесь и далее мы условимся считать равными нулю суммы, у которых верхний предел суммирования меньше нижнего.

Используя представление (11.6) матрицы  $A$ , запишем СЛАУ  $Ax = b$  в следующем виде:

$$(D + A_-)x = -A_+x + b. \quad (11.11)$$

Если диагональные элементы матрицы  $A$  не равны нулю, то нижняя треугольная матрица  $D + A_-$  имеет обратную матрицу  $(D + A_-)^{-1}$  и систему (11.11) эквивалентна следующей:

$$x = (D + A_-)^{-1}(-A_+x + b). \quad (11.12)$$

Соотношение (11.12) можно использовать для построения итерационной последовательности

$$x^{N+1} = (D + A_-)^{-1}(-A_+x^N + b). \quad (11.13)$$

использование которой для решения СЛАУ  $Ax = b$  называют *методом Зейделя*.

Для вычисления итерационной последовательности в методе Зейделя требуется обращение нижней треугольной матрицы  $D + A_-$ , но оказывается, что такое обращение является формальным. Преобразуем соотношение (11.13):

$$Dx^{N+1} = -A_-x^{N+1} - A_+x^N + b.$$

Исходя из этой формы уравнения, получаем

$$x_i^{N+1} = \frac{1}{a_{ii}} \left( - \sum_{j=1}^{i-1} a_{ij} x_j^{N+1} - \sum_{j=i+1}^n a_{ij} x_j^N + b_i \right), \quad i = \overline{1, n}. \quad (11.14)$$

В уравнениях (11.14) элементы  $x$  столбца  $x^{N+1}_i$  находятся и в левой, и в правой части. Однако если их вычислять последовательно для  $i = 1, i = 2, i = n$ , мы увидим, что в формуле для очередного элемента  $x^{N+1}_i$  используются только уже найденные элементы  $x^{N+1}_1, \dots, x^{N+1}_{i-1}$ . Обращение матрицы  $D + A_-$  естественным образом встроено в вычислительную схему и не требует отдельных усилий.

Вычислительные схемы методов Зейделя и Якоби, которые описываются уравнениями (11.10) и (11.14), очень похожи. Различие лишь в том, что при вычислении каждого элемента столбца  $x^{N+1}$  в методе Якоби используются только элементы предыдущего столбца  $x^N$ , а в методе Зейделя более новые уже найденные элементы текущего столбца.

И метод Якоби, и метод Зейделя могут быть получены в рамках канонической формы (11.5) одношаговых итерационных методов. Итерационная последовательность метода Якоби подчиняется уравнению  $Dx^{N+1} = -(A_- + A_+)x^N + b$ , которое эквивалентно (11.9). Это уравнение легко преобразуется в форму  $Dx^{N+1} = -(A - D)x^N + b$ , откуда получаем

$$D(x^{N+1} - x^N) + Ax^N = b.$$

Видим, что для получения метода Якоби в канонической форме надо положить  $B_{N+1} = D$ ,  $T_{N+1} = 1$ . Таким образом, метод Якоби можно квалифицировать как одношаговый неявный стационарный итерационный метод.

Из уравнения (11.13) находим  $(D + A_-)x^{N+1} = -A_+x^N + b$ , или, заменяя матрицу  $A_+$  через матрицу  $A$ ,  $(D + A_-)x^{N+1} = -(A - D - A_-)x^N + b$ . Следовательно,

$$(D + A_-)(x^{N+1} - x^N) + Ax^N = b. \quad (11.15)$$

Для представления метода Зейделя в канонической форме мы должны положить  $B_{N+1} = D + A_-$ ,  $T_{N+1} = 1$ ,  $N = 0, 1, \dots$ . Таким образом, метод Зейделя также относится к одношаговым стационарным неявным методам.

### 3. ОТ ЧЕГО ЗАВИСИТ СКОРОСТЬ СХОДИМОСТИ ИТЕРАЦИОННЫХ МЕТОДОВ?

Сходимость итерационной последовательности стационарного метода означает, что к нулю сходится последовательность  $\{v^N\}$ , определенная рекуррентным соотношением (11.25). Согласно равенству (11.26), указанная сходимость определяется сжимающими свойствами матрицы  $T$  [I-Д8.2]. Бели отображение  $y = Tx$  в пространстве  $R^n$  с заданной нормой является сжимающим, т.е. для некоторой постоянной  $q < 1$  и любых  $x^1, x^2 \in R^n$  выполняется неравенство

$$\|Tx^1 - Tx^2\| = \|T(x^1 - x^2)\| \leq q\|x^1 - x^2\|,$$

то последовательность  $\{v^N\}$  сходится. Постоянная  $q$  определяется свойствами матрицы  $T$ , и от величины этой постоянной зависит, насколько быстро сходится последовательность  $\{v^N\}$ . Необходимое и достаточное условие сходимости к нулю последовательности ошибок дает теорема 11.4, но проверка условия теоремы достаточно сложна. Поэтому часто прибегают к другим условиям, более простым, но имеющим лишь достаточный характер. Эти условия обычно связаны с нормой матрицы  $T$ .

Как и выше, будем рассматривать столбцы высоты  $n$  в качестве векторов линейного арифметического пространства  $R^n$ , в котором задана некоторая норма  $\|\cdot\|$ . Для матриц будем использовать норму, индуцированную заданной нормой в  $R^n$ , и для индуцированной нормы используем то же обозначение, что и для нормы в  $R^n$ . Из (11.26) следует, что

$$\|v^N\| \leq \|T^N\| \|v^0\|.$$

Отсюда видим, что последовательность  $\{v^N\}$  сходится к нулю, если  $\|T^N\| \rightarrow 0$  при  $N \rightarrow \infty$ . Взяв  $\varepsilon > 0$ , найдем такой номер  $N(\varepsilon)$ , что при  $N \geq N(\varepsilon)$  будет  $\|T^N\| < \varepsilon$ . Тогда для  $N = N(\varepsilon)$  получаем  $\|v^N\| < \varepsilon \|v^0\|$ , т.е. за  $N$  итераций начальное приближение уменьшилось в  $1/\varepsilon$  раз. Число  $N(\varepsilon)$  следует выбирать наименее возможным, и тогда оно будет определять минимальное число итераций для заданной точности  $\varepsilon$ , а обратную величину  $1/N(\varepsilon)$  естественно рассматривать как скорость сходимости итерационного метода. Отметим, что минимальное число итераций и скорость сходимости метода зависят от того, какая выбрана норма в линейном арифметическом пространстве.

Вычислить нормы  $\|T^N\|$  очень сложно. Чтобы установить сходимость последовательности таких норм и выяснить скорость сходимости, надо использовать для таких норм различные оценки. Например, если  $\|T\| = q < 1$ , то, согласно неравенству  $\|AB\| \leq \|A\| \|B\|$ , верному для любой кольцевой нормы, имеем неравенство  $\|T^N\| \leq \|T\|^N = q^N$ , откуда сразу следует сходимость к нулю последовательности  $\{\|T^N\|\}$ . Более того, записав неравенства  $\|T^N\| \leq$

$q^N < \epsilon$ , получаем при помощи логарифмирования оценку минимального числа итераций:

$$N(\epsilon) \geq \frac{\ln \epsilon}{\ln q} = \frac{\ln(1/\epsilon)}{\ln(1/\|T\|)}.$$

Скорость сходимости итерационной последовательности в стационарном методе зависит, вообще говоря, от выбранного значения итерационного параметра  $r$ . То значение  $r$  параметра, при котором скорость сходимости метода наивысшая, называют оптимальным итерационным параметром. Это значение, конечно, зависит от используемой нормы. Остановимся на случае, когда в  $R^n$  рассматривается евклидова норма, порожденная стандартным скалярным произведением.

Если  $A$  — симметрическая положительно определенная матрица, то для метода простой итерации (11.16)

$$\tau_0 = \frac{2}{\lambda_{\max}(A) + \lambda_{\min}(A)},$$

где  $\lambda_{\max}(A)$  и  $\lambda_{\min}(A)$  — соответственно минимальное и максимальное собственные значения матрицы  $A$ . При этом верна следующая оценка ошибки  $N$ -й итерации:  $\|v^N\| \leq \rho^N \|v^0\|$ , где

$$\rho = \frac{\lambda_{\max}(A) - \lambda_{\min}(A)}{\lambda_{\max}(A) + \lambda_{\min}(A)}.$$

Параметр  $\rho$  определяется отношением  $\kappa = \lambda_{\max}(A)/\lambda_{\min}(A)$ , представляющим собой число обусловленности симметрической положительно определенной матрицы  $A$ . Чем меньше число обусловленности, тем меньше  $\rho$  и тем выше скорость сходимости метода. При плохо обусловленной матрице  $A$  параметр  $\rho$  близок к единице и скорость сходимости метода простой итерации мала. В такой ситуации целесообразно ориентироваться на неявные итерационные методы (11.22).

Для неявного итерационного метода (11.22), т.е. при  $B \neq E$ , в случае симметрических матриц  $A$  и  $B$

$$\tau_0 = \frac{2}{\lambda_{\max}(B^{-1}A) + \lambda_{\min}(B^{-1}A)},$$

где  $\lambda_{\min}(B^{-1}A)$  и  $\lambda_{\max}(B^{-1}A)$  — соответственно минимальное и максимальное собственные значения матрицы  $B^{-1}A$ . Для ошибки итерации верна оценка  $\|v^N\| \leq \rho^N \|v^0\|$ , где

$$\rho = \frac{\lambda_{\max}(B^{-1}A) - \lambda_{\min}(B^{-1}A)}{\lambda_{\max}(B^{-1}A) + \lambda_{\min}(B^{-1}A)}.$$

Величина  $\rho$  определяется числом обусловленности

$$\xi = \frac{\lambda_{\max}(B^{-1}A)}{\lambda_{\min}(B^{-1}A)}$$

матрицы  $B^{-1}A$ . Невырожденную симметрическую матрицу  $B$  следует выбирать так, чтобы  $\xi$  было мало и во всяком случае было меньше числа обусловленности матрицы  $A$  (в противном случае использование неявного метода не имеет смысла).

#### 4. КАКИЕ ПРЕИМУЩЕСТВА У ВАРИАЦИОННО-ИТЕРАЦИОННЫХ МЕТОДОВ?

Преимущество данных методов — они не используют никакой дополнительной информации об операторе  $A$ , т.е.  $\gamma_1$  и  $\gamma_2$ , входящие в оценку  $\gamma_1 E \leq A \leq \gamma_2 E$  и необходимые для выбора  $\tau_0$ , здесь не требуются. Рассмотрим методы минимальных невязок и скорейшего спуска.

- 1) имеют простую вычислительную процедуру;
- 2) не требуют сложных специальных процедур для экономии памяти ЭВМ под нулевые элементы матрицы коэффициентов, как метод Гаусса;
- 3) самоисправление ошибок.

#### 5. КАКИМ ОБРАЗОМ ОПРЕДЕЛЯЕТСЯ ОКОНЧАНИЕ ИТЕРАЦИЙ?

Критерий окончания в методе Якоби:

$$\|x^{(n+1)} - x^{(n)}\| < \varepsilon, \text{ где } \varepsilon = \frac{1 - \rho}{1 + \rho} \cdot \varepsilon_0$$

В случае если  $\rho < 1/2$ , то можно применить более простой критерий окончания итераций:

$$\|x^{(n+1)} - x^{(n)}\| < \varepsilon$$

За условия сходимости метода Зейделя и критерий окончания итераций можно принять такие же значения, как и в методе Якоби.

## 6. ЗАПИШИТЕ МЕТОД ЯКОБИ В ВЕКТОРНОЙ ФОРМЕ

Метод Якоби еще можно записать так

$$D(x^{k+1} - x^k) + Ax^k = f.$$

## 7. ЗАПИШИТЕ МЕТОД ЗЕЙДЕЛЯ В ВЕКТОРНОЙ ФОРМЕ.

Можно получить представление метода Зейделя в векторной форме

$$(D + A^-)(x^{k+1} - x^k) + Ax^k = f.$$

## 8. ЗАПИШИТЕ ФОРМУЛУ РАСЧЕТА ИТЕРАЦИОННОГО ПАРАМЕТРА СОГЛАСНО МЕТОДУ СКОРЕЙШЕГО СПУСКА.

В общем случае параметр  $\lambda$  может быть найден из уравнения

$$\frac{d}{d\lambda} U(\bar{x}^{(k)} - \lambda \nabla U(\bar{x}^{(k)})) = 0$$

Если считают  $\lambda$  малой величиной и не учитывают членов, содержащих  $\lambda$  во второй и высших степенях, то приближенно искомое решение можно найти из матричных равенств  $\bar{x}^{(k+1)} = \bar{x}^{(k)} - \mu_k W_k^T \bar{f}^{(k)}$  ( $k = 0, 1, \dots$ ),

$$\mu_k = 2\lambda_k = \frac{(\bar{f}^{(k)}, W_k W_k^T \bar{f}^{(k)})}{(W_k W_k^T \bar{f}^{(k)}, W_k W_k^T \bar{f}^{(k)})}, \quad (k = 0, 1, \dots), \text{ где}$$

$$\bar{f}^{(k)} = \bar{f}(\bar{x}^{(k)}) = \begin{pmatrix} f_1(x_1^{(k)}, \dots, x_n^{(k)}) \\ f_2(x_1^{(k)}, \dots, x_n^{(k)}) \\ \dots \\ f_n(x_1^{(k)}, \dots, x_n^{(k)}) \end{pmatrix},$$

$$W_k = W(\bar{x}^{(k)}) = \begin{pmatrix} \frac{\partial f_1(\bar{x}^{(k)})}{\partial x_1} & \dots & \frac{\partial f_1(\bar{x}^{(k)})}{\partial x_n} \\ \dots & \dots & \dots \\ \frac{\partial f_n(\bar{x}^{(k)})}{\partial x_1} & \dots & \frac{\partial f_n(\bar{x}^{(k)})}{\partial x_n} \end{pmatrix},$$

$$W_k^T \bar{f}^{(k)} = W^T(\bar{x}^{(k)}) \cdot \bar{f}(\bar{x}^{(k)}) = \begin{pmatrix} \sum_{i=1}^n \frac{\partial f_1(\bar{x}^{(k)})}{\partial x_1} \bar{f}_i(\bar{x}^{(k)}) \\ \dots \\ \sum_{i=1}^n \frac{\partial f_n(\bar{x}^{(k)})}{\partial x_n} \bar{f}_i(\bar{x}^{(k)}) \end{pmatrix}.$$

### 9\*. УСЛОВИЕ ПРИМЕНИМОСТИ МЕТОДА МИНИМАЛЬНЫХ НЕВЯЗОК.

Методом минимальных невязок называется итерационный метод (4), в котором параметр  $\tau_{k+1}$  выбирается из условия минимума  $\|r_{k+1}\|$  при заданной норме  $\|r_j\|$ . Получим явное выражение для итерационного параметра  $\tau_{k+1}$ . Из (5) получаем и, следовательно,  $Ax_{k+1} = Ax_k - \tau_{k+1} r_k$ ,  $x_{k+1} = x_k + \tau_{k+1} \Gamma^{-1} r_k$ , (6) т. е. невязка  $r_k$  удовлетворяет тому же уравнению, что и погрешность  $Z_k = XA - X$ . Возводя обе части уравнения (6) скалярно в квадрат, получим  $\|r_{k+1}\|^2 = \|r_k\|^2 - 2\tau_{k+1} (r_k, Ar_k) + \tau_{k+1}^2 \Pi Ar_k f$ . (7) Отсюда видно, что  $\|r_{k+1}\|$  достигает минимума, если  $1 - (4'fe, r_k) \Pi Ar_k f$  (8) Таким образом, в методе минимальных невязок переход от  $k$ -н итерации к  $(k+1)$ -й осуществляется следующим образом. По найденному значению  $x_k$  вычисляется вектор невязки  $r_k = Ax_k - f$  и по формуле (8) находится параметр  $\tau_{k+1}$ . Затем по формуле (5) досчитывается вектор  $x_{k+1}$ . Метод минимальных невязок (5), (8) сходится с той же скоростью, что и метод простой итерации с оптимальным параметром  $\tau$ .

## 10\*. УСЛОВИЕ ПРИМЕНИМОСТИ МЕТОДА СКОРЕЙШЕГО СПУСКА.

Метод скорейшего спуска. Рассмотрим явный метод и выберем итерационный параметр  $\tau_{k+1}$  из условия минимума  $\|z_{k+1}\|_A$  при заданном векторе  $z_k$ , где  $z_{k+1} = x_{k+1} - x$ . Поскольку погрешность  $z_k$  удовлетворяет уравнению  $z_{k+1} = z_k - \tau_{k+1} A^* z_k$

имеем

$$\|z_{k+1}\|_A^2 = \|z_k\|_A^2 - 2\tau_{k+1} (Az_k, Az_k) + \tau_{k+1}^2 (A^2 z_k, Az_k).$$

Следовательно,  $\|z_{k+1}\|_A^2$  будет минимальной, если положить

$$\tau_{k+1} = \frac{(Az_k, Az_k)}{(A^2 z_k, Az_k)}.$$

Метод скорейшего спуска сходится с той же скоростью, что и метод простой итерации с оптимальным параметром  $\tau = \tau_0$ . Для погрешности метода скорейшего спуска справедлива оценка

$$\|x_n - x\|_A \leq \rho_0^n \|x_0 - x\|_A, \quad n = 0, 1, \dots,$$

$$\text{где } \rho_0 = \frac{1 - \xi}{1 + \xi}, \quad \xi = \frac{\lambda_{\min}(A)}{\lambda_{\max}(A)}.$$