МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«МИРЭА - Российский технологический университет»
РТУ МИРЭА

Институт Информационных Технологий

Кафедра Вычислительной Техники

ОТЧЕТ ПО ПРАКТИЧЕСКИМ РАБОТАМ

по дисциплине

«Математическое обеспечение систем поддержки принятия решений»

Студент группы:ИКБО-04-22 __Кликушин В.И._
(Ф. И.О. студента)

Преподаватель __Семенов Р.Э.__
(Ф.И.О. преподавателя)

Москва 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

История методов оптимизации уходит корнями в древние времена, когда люди начали искать способы минимизации затрат и максимизации результатов в повседневной жизни. Однако математическая теория оптимизации начала формироваться в XVII–XVIII веках, благодаря работам таких ученых, как Исаак Ньютон и Готфрид Лейбниц, которые заложили основы дифференциального исчисления. Именно дифференциальное исчисление стало основой для разработки методов поиска экстремумов функций.

В XIX веке развитие методов оптимизации продолжилось благодаря работам Коши, который предложил метод градиентного спуска, и Вейерштрасса, который сформулировал теорему о существовании минимума непрерывной функции на компактном множестве. В XX веке с развитием вычислительной техники и появлением компьютеров методы оптимизации получили новый импульс. Были разработаны численные методы, которые позволили решать задачи оптимизации в многомерных пространствах.

С развитием машинного обучения и искусственного интеллекта в конце XX – начале XXI века методы безусловной оптимизации стали еще более востребованными. Они используются для обучения нейронных сетей, настройки параметров моделей и решения задач регрессии и классификации. Сегодня безусловная оптимизация является неотъемлемой частью современных технологий, таких как глубокое обучение, обработка больших данных и автоматизация процессов принятия решений.

Безусловная оптимизация является важным разделом математики, который находит применение в самых разных областях. Её методы позволяют решать сложные задачи, связанные с поиском экстремумов функций, и являются основой для более сложных методов оптимизации с ограничениями. Несмотря на трудности, связанные с многомерностью, нелинейностью и вычислительной сложностью, современные методы безусловной оптимизации продолжают развиваться, что делает их мощным инструментом для решения задач.

1 МЕТОД ХУКА-ДЖИВСА

1.1 Описание алгоритма

Метод Хука – Дживса (метод конфигураций, метод пробных шагов) относится, с одной стороны, к классу прямых методов оптимизации, а с другой стороны – к классу детерминированных методов оптимизации. Метод предназначен для решения многомерных задач локальной безусловной оптимизации.

В методе Хука-Дживса поиск минимума состоит из последовательности шагов исследующего поиска относительно базисной точки и поиска по образцу.

Исследующий поиск состоит в следующем. Задаются некоторой начальной (базисной) точкой $\overline{x^{(0)}}$ и величиной шага h для каждого координатного направления $(i = \overline{1,n})$. Обследуют окрестность данной точки, изменяя по очереди компоненты вектора $\overline{x^{(0)}}$ вдоль каждого координатного направления. Для этого вычисляется значение целевой функции $f(\overline{x^{(0)}} + h_i\overline{e_i})$ в пробной точке $\overline{x^{(0)}} + h_i\overline{e_i}$, где $\overline{e_i}$ – единичный вектор в направлении оси x_i. Если значение целевой функции в пробной точке меньше значения целевой функции в базисной точке $\overline{x^{(0)}}$, то выбранный шаг h_i считается удачным, и точка $\overline{x^{(0)}}$ заменяется на $\overline{x^{(0)}} + h_i\overline{e_i}$. В противном случае вычисляется величина $\overline{x^{(0)}} - h_i\overline{e_i}$ и если значение целевой функции уменьшается, то $\overline{x^{(0)}}$ заменяется на $\overline{x^{(0)}} - h_i\overline{e_i}$. После перебора всех координатных направлений $(i = \overline{1,n})$ исследующий поиск завершается. Полученную в результате точку $\overline{x^{(1)}}$ называют новым базисом. Если в точке нового базиса $\overline{x^{(1)}}$ уменьшение значения целевой функции не было достигнуто, то исследующий поиск повторяется вокруг той же базисной точки $\overline{x^{(0)}}$ но с меньшей величиной шага. Поиск завершается, когда все текущие величины шага будут меньше заданной точности. Если исследующий поиск был удачен, т.е. $f(\overline{x^{(1)}}) < f(\overline{x^{(0)}})$, то производится поиск по образцу.

Поиск по образцу производится из точки $\overline{x^{(1)}}$ в направлении вектора $(\overline{x^{(1)}} - \overline{x^{(0)}})$, поскольку это направление привило к уменьшению значения целевой функции. Координаты новой точки определяются в соответствии с Формулой 1.1.1.

$$\overline{x^{(2)}} = \overline{x^{(1)}} + \alpha\left(\overline{x^{(1)}} - \overline{x^{(0)}}\right), \tag{1.1.1}$$

где α – ускоряющий множитель, $\alpha > 0$.

Если в результате получена точка с меньшим значением целевой функции, то она рассматривается как новая базисная точка. Если поиск по образцу был неудачен, то происходит возврат в новый базис $\overline{x^{(1)}}$, где продолжается исследующий поиск с целью выявления нового направления минимизации.

Алгоритм метода Хука-Дживса:

1. Задать размерность задачи оптимизации n, координаты начальной базисной точки $\overline{x^{(0)}} = (x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)})$, шаг h, для каждой переменной $(i = \overline{1,n})$, коэффициент уменьшения шага d $(d > 1)$, ускоряющий множитель m $(m > 0)$, точность поиска ε;
2. Ввести в рассмотрение текущую точку $\overline{x^{(1)}} = \{x_1^{(1)}, x_2^{(1)}, \dots, x_n^{(1)}\}$;
3. Положить $\overline{x^{(1)}} = \overline{x^{(0)}}$. Вычислить значение функции $f(\overline{x^{(1)}})$ в точке $\overline{x^{(1)}}$;
4. Зафиксировать первое координатное направление $i = 1$;
5. Провести исследующий поиск вдоль оси x_i. Сделать шаг h_i в положительном направлении координатной оси $\overline{x_i^{(1)}} = \overline{x_i^{(1)}} + h_i$ и вычислить значение функции $f(\overline{x^{(1)}})$ в полученной точке $\overline{x^{(1)}}$;
6. Если $f(\overline{x^{(1)}}) < f(\overline{x^{(0)}})$, то шаг в положительном направлении оси x_i считается удачным, и осуществляется переход к пункту 8. В противном случае происходит возраст в исходную точку и делается шаг в отрицательном направлении координатной оси $\overline{x_i^{(1)}} = \overline{x_i^{(1)}} -$

$2h_i$;

7. Если $f(\overline{x^{(1)}}) < f(\overline{x^{(0)}})$, то шаг в отрицательном направлении оси x_i считается удачным. Если условие не выполнено, то происходит возврат в исходную точку $\overline{x_i^{(1)}} = \overline{x_i^{(1)}} + h_i$. И в том и другом случае осуществляется переход к следующему пункту 8;
8. Проверить условие окончания исследующего поиска. Если $i < n$, то положить $i = i + 1$ и перейти к пункту 5 для продолжения исследующего поиска по оставшимся координатным направлениям. Если $i = n$ перейти к пункту 9;
9. Проверить успешность исследующего поиска. Если $\overline{x^{(1)}} = \overline{x^{(0)}}$, т.е. исследующий поиск был неудачным, то необходимо уменьшить величину шага $h_i = \frac{h_i}{d} (i = \overline{1,n})$ и проверить поиск с пункта 4;
10. Если $\overline{x^{(1)}} \neq \overline{x^{(0)}}$, то проводится поиск по образцу $\overline{x^{(p)}} = \overline{x^{(1)}} + \alpha\left(\overline{x^{(1)}} - \overline{x^{(0)}}\right)$ и вычисляется значение функции в точке образца $f(\overline{x^{(p)}})$;
11. Проверить удачность поиска по образцу. Если $f\left(\overline{x^{(p)}}\right) < f(\overline{x^{(1)}})$, то поиск по образцу удачен, и точка $\overline{x^{(0)}} = \overline{x^{(p)}}$ становится новой базисной точкой. В противном случае за базис применяется точка $\overline{x^{(0)}} = \overline{x^{(1)}}$.
12. Проверить условие окончания поиска. Если длины шагов $h_i \leq \varepsilon$ $(i = \overline{1,n})$, то поиск завершен $\overline{x^*} = \overline{x^{(1)}}, f_{min} = f(\overline{x^*})$. В противном случае осуществляется переход к пункту 3 и проводится исследующий поиск вокруг новой базисной точки $\overline{x^{(0)}}$.

1.2 Постановка задачи

Цель работы: реализовать безусловную оптимизацию функции методом нулевого порядка.

Задачи: изучить метод Хука-Дживса, выбрать функцию для оптимизации (нахождение глобального минимума), произвести ручной расчёт итерации алгоритма, разработать программную реализацию метода Хука-Дживса.

Выбранная функция для оптимизации представлена Формулой 1.2.1.

$$f(x_1, x_2) = x_1^2 + e^{x_1^2 + x_2^2} + 4x_1 + 3x_2 \quad (1.2.1)$$

График выбранной функции представлен на Рисунке 1.2.1.

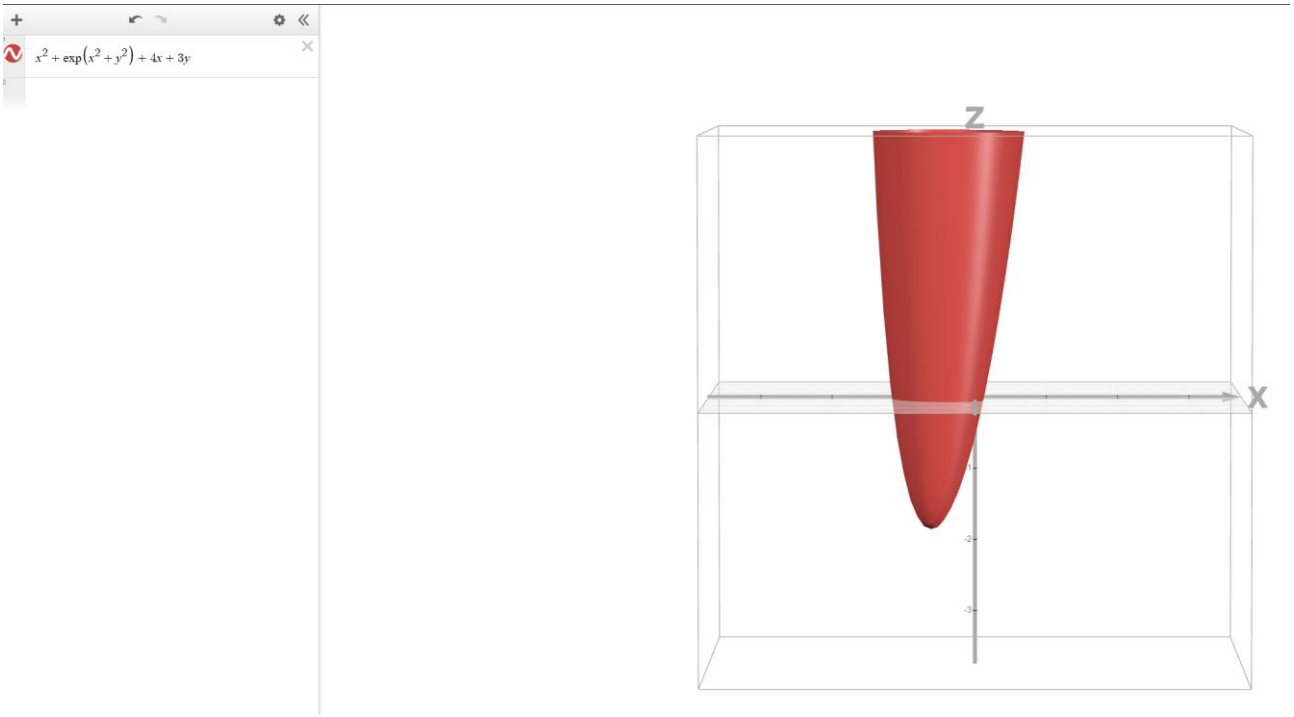

Рисунок 1.2.1 – График выбранной функции

Глобальный минимум приблизительно равен -1.8.

1.3 Ручной расчёт

Точность поиска ε принята равной 0.0001. Начальная точка $\overline{x^{(0)}} = (1; 1)$. Шаг по координатным направлениям h равен 0.2, коэффициент уменьшения шага d равен 2.

Произведём исследующий поиск вокруг базисной точки $\overline{x^{(0)}}$, значение целевой функции $f\left(\overline{x^{(0)}}\right) = 15.3891$. Фиксируя координату $x_2^{(0)}$ и делая шаг в

положительном направлении координатной оси x_1, получим пробную точку $\left(\overline{x_1^{(0)}} + h;\ \overline{x_2^{(0)}}\right) = (1 + 0.2; 1) = (1.2; 1)$. Так как $f(1.2; 1) = 20.71 > 15.3891$, то шаг в этом направлении считается неудачным. Возвращаемся на исходную точку и делаем шаг в отрицательном направлении оси x_1: $(1 - h; 1) = (0.8; 1)$. Так как $f(0.8; 1) = 11.995 < 15.3891$, то шаг в этом направлении считается удачным.

Далее делаем пробный шаг в направлении оси x_2, получим пробную точку $(0{,}8;\ 1 + h) = (0{,}8;\ 1{,}2)$. Так как $f(0.8; 1.2) = 15.9 > 11.995$, то шаг в этом направлении считается неудачным. Возвращаемся на исходную точку и делаем шаг в отрицательном направлении оси x_2: $(0.8;\ 1 - h) = (0.8;\ 0.8)$. Так как $f(0.8; 0.8) = 9.83 < 11.995$, то шаг в этом направлении считается удачным.

Таким образом, рассмотрены все координатные направления и найдена новая базисная точка $\overline{x^{(1)}} = (0.8; 0.8)$, которой соответствует значение целевой функции $f\left(\overline{x^{(1)}}\right) = 9.83$. Учитывая, что исследующий поиск был удачен $\overline{x^{(0)}} \neq \overline{x^{(1)}}$, то произведем поиск по образцу. В результате поиска получена точка $\overline{x^{(p)}} = (0.4; 0.4)$. Значение функции в найденной точке равно $f\left(\overline{x^{(p)}}\right) = 4.33$. Так как $f\left(\overline{x^{(p)}}\right) < f(\overline{x^{(1)}})$, то шаг по образцу считается удачным, и точка $\overline{x^{(p)}} = \overline{x^{(2)}}$ становится новой базисной точкой.

1.4 Программная реализация

Написан класс HookeJeevesMethod, который реализует рассматриваемый метод Хука-Дживса. Конструктор класса принимает в качестве параметров ссылку на функцию, начальную точку, величину шага для исследующего поиска, коэффициент уменьшения шага, ускоряющий множитель для поиска по образцу, точность поиска.

Код реализации метода Хука-Дживса представлен в Приложении А.

Результат работы программы представлен на Рисунках 1.4.1–1.4.2.

```
(.venv) PS C:\python_projects\MIREA\Математическое обеспечение систем поддержки принятия решений\практические работы\методы нулевого порядка\метод хука-дживса> python Hooke_Jeeves_method.py
Итерация №0:
| Вершина |  x1 |  x2 |  Значение функции |
|---|---|---|---|
| X0 |  1 |  1 |  15.3891 |
Итерация №1:
| Вершина |  x1 |  x2 |  Значение функции |
|---|---|---|---|
| X0 |  1 |  1 |  15.3891 |
| X1 |  0.4 |  0.4 |  4.33713 |
Итерация №2:
| Вершина |  x1 |  x2 |  Значение функции |
|---|---|---|---|
| X0 |  1 |  1 |  15.3891 |
| X1 |  0.4 |  0.4 |  4.33713 |
| X2 |  -0.2 |  -0.2 |  -0.276713 |
Итерация №3:
| Вершина |  x1 |  x2 |  Значение функции |
|---|---|---|---|
| X0 |  1 |  1 |  15.3891 |
| X1 |  0.4 |  0.4 |  4.33713 |
| X2 |  -0.2 |  -0.2 |  -0.276713 |
| X3 |  -0.8 |  -0.8 |  -1.36336 |
Итерация №4:
| Вершина |  x1 |  x2 |  Значение функции |
|---|---|---|---|
| X0 |  1 |  1 |  15.3891 |
| X1 |  0.4 |  0.4 |  4.33713 |
| X2 |  -0.2 |  -0.2 |  -0.276713 |
| X3 |  -0.8 |  -0.8 |  -1.36336 |
| X4 |  -0.6 |  -0.6 |  -1.78557 |
Итерация №5:
| Вершина |  x1 |  x2 |  Значение функции |
|---|---|---|---|
| X0 |  1 |  1 |  15.3891 |
| X1 |  0.4 |  0.4 |  4.33713 |
| X2 |  -0.2 |  -0.2 |  -0.276713 |
| X3 |  -0.8 |  -0.8 |  -1.36336 |
| X4 |  -0.6 |  -0.6 |  -1.78557 |
```

Рисунок 1.4.1 – Начальные итерации работы метода Хука-Дживса

```
Итерация №20:
| Вершина |  x1 |  x2 |  Значение функции |
|---|---|---|---|
| X0 |  1 |  1 |  15.3891 |
| X1 |  0.4 |  0.4 |  4.33713 |
| X2 |  -0.2 |  -0.2 |  -0.276713 |
| X3 |  -0.8 |  -0.8 |  -1.36336 |
| X4 |  -0.6 |  -0.6 |  -1.78557 |
| X5 |  -0.6 |  -0.7 |  -1.80035 |
| X6 |  -0.6 |  -0.65 |  -1.80307 |
| X7 |  -0.625 |  -0.65 |  -1.80443 |
| X8 |  -0.6125 |  -0.6625 |  -1.80529 |
| X9 |  -0.614063 |  -0.6625 |  -1.80529 |
| X10 |  -0.613281 |  -0.663281 |  -1.80529 |
Найденный минимум: [-0.6132812500000002, -0.6632812500000003]
Значение функции в минимуме: -1.8052924440555334
```

Рисунок 1.4.2 – Последняя итерация метода Хука-Дживса

2 МЕТОД НАИСКОРЕЙШЕГО ГРАДИЕНТНОГО СПУСКА

2.1 Описание алгоритма

Метод наискорейшего спуска отличается от метода градиентного спуска способом определения величины шага h_k. Величина шага задается не произвольно, а выбирается так, чтобы на каждой итерации достигалось максимально возможное уменьшение целевой функции $f(\bar{x})$ вдоль направления ее антиградиента $-\nabla f(\overline{x^{(k)}})$, вычисленного в точке $\overline{x^{(k)}}$. Величина шага h_k определяется из решения вспомогательной одномерной задачи минимизации, которая может быть решена аналитически или численно (Формула 2.1.1).

$$\varphi(h_k) = f\left(\overline{x^{(k)}} - h_k \nabla f\left(\overline{x^{(k)}}\right)\right) \to min, h_k > 0 \qquad (2.1.1)$$

При квадратичной интерполяции целевой функции величину шага можно определить по Формуле 2.1.2.

$$h_k = \frac{(\nabla f\left(\overline{x^{(k)}}\right), \nabla f\left(\overline{x^{(k)}}\right))}{(H\left(\overline{x^{(k)}}\right)\nabla f\left(\overline{x^{(k)}}\right), \nabla f\left(\overline{x^{(k)}}\right))}, \qquad (2.1.2)$$

где $H\left(\overline{x^{(k)}}\right)$ – матрица Гессе, вычисленная в точке $\overline{x^{(k)}}$.

Алгоритм метода минимизации целевой функции $f(\bar{x})$ методом наискорейшего спуска заключается в следующем:

1. Задать размерность задачи оптимизации n, координаты начальной точки $\overline{x^{(0)}} = (x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)})$, точности поиска ε.
2. Положить счетчик числа $k = 0$.
3. Определить направление вектора градиента целевой функции $f(\bar{x})$:

$\nabla f(\overline{x^{(k)}}) = (\frac{\partial f(\overline{x^{(k)}})}{\partial x_1}, \frac{\partial f(\overline{x^{(k)}})}{\partial x_2}, \dots, \frac{\partial f(\overline{x^{(k)}})}{\partial x_n})$ в точке $\overline{x^{(k)}} = (x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)})$. Для вычисления координат вектора градиента использовать разностную формулу первых частных производных.

4. Проверить условие окончания поиска (Формула 2.1.3).

$$\left\|\nabla f(\overline{x^{(k)}})\right\| = \sqrt{\sum_{i=1}^{n}(\frac{\partial f(\overline{x^{(k)}})}{\partial x_i})^2} \leq \varepsilon \qquad (2.1.3)$$

Если условие выполнено, то расчет окончен $\overline{x^*} = \overline{x^{(k)}}$, иначе прейти к пункту 5.

5. Вычислить шаг h_k по Формуле 2.1.2, используя результаты вычислений пункта 3 и разностные формулы вторых и смешанных производных.
6. Определить координаты точки $\overline{x^{(k+1)}} = \overline{x^{(k)}} - h_k \nabla f(\overline{x^{(k)}})$, положить $k = k + 1$ и перейти к пункту 3.

2.2 Постановка задачи

Цель работы: реализовать безусловную оптимизацию функции методом первого порядка.

Задачи: изучить методы градиентного спуска с постоянным шагом, наискорейшего градиентного спуска, выбрать функцию для оптимизации (нахождение глобального минимума), произвести ручной расчёт итерации алгоритма, разработать программную реализацию метода наискорейшего градиентного спуска.

Выбранная функция для оптимизации представлена Формулой 1.2.1.

График выбранной функции представлен на Рисунке 1.2.1.

Глобальный минимум приблизительно равен -1.8.

2.3 Ручной расчёт

Точность поиска ε принята равной 0.0001. Начальная точка $\overline{x^{(0)}} = (1; 1)$.

Найдем градиент функции в произвольной точке $\overline{x^{(k)}} = (x_1^{(k)}, x_2^{(k)})$:

$$\nabla f\left(\overline{x^{(k)}}\right) = \left(\frac{\partial f\left(\overline{x^{(k)}}\right)}{\partial x_1}; \frac{\partial f\left(\overline{x^{(k)}}\right)}{\partial x_2}\right) = (2x_1 * e^{x_1^2+x_2^2} + 2x_1 + 4; 2x_2 * e^{x_1^2+x_2^2} + 3)$$

Вычислим градиент функции $\nabla f\left(\overline{x^{(k)}}\right)$ в начальной точке $\overline{x^{(0)}}$:

$$\nabla f\left(\overline{x^{(0)}}\right) = (20.77; 17.77)$$

Вычислим матрицу Гессе в точке $\overline{x^{(0)}}$:

$$H\left(\overline{x^{(0)}}\right) = \begin{pmatrix} 46.33 & 29.55 \\ 29.55 & 44.33 \end{pmatrix}$$

Определим координаты точки $\overline{x^{(1)}}$ по Формуле из пункта 6:

$$h_k = 0.013$$

$$\overline{x^{(1)}} = \overline{x^{(0)}} - h_0 \nabla f\left(\overline{x^{(0)}}\right) = (1; 1) - 0.013 * (20.77; 17.77) = (0.721; 0.761)$$

Вычислим значение функции в найденной точке: $f\left(\overline{x^{(1)}}\right) = 8.70$.

Проверим условие окончания процесса поиска. Для этого вычислим градиент целевой функции $\nabla f\left(\overline{x^{(1)}}\right)$ в точке $\overline{x^{(1)}}$: $\nabla f\left(\overline{x^{(1)}}\right) = (9.78; 7.58)$.

Норма вектора градиента равна 12.38, больше, чем заданная точность поиска, поэтому начинается вторая итерация.

2.4 Программная реализация

Написан класс RapidGradientDescent, который реализует метод наискорейшего градиентного спуска. Конструктор класса принимает в качестве параметров ссылку на функцию, начальную точку, точность поиска.

Код реализации метода наискорейшего градиентного спуска представлен в Приложении Б.

Результат работы программы представлен на Рисунках 2.4.1–2.4.2.

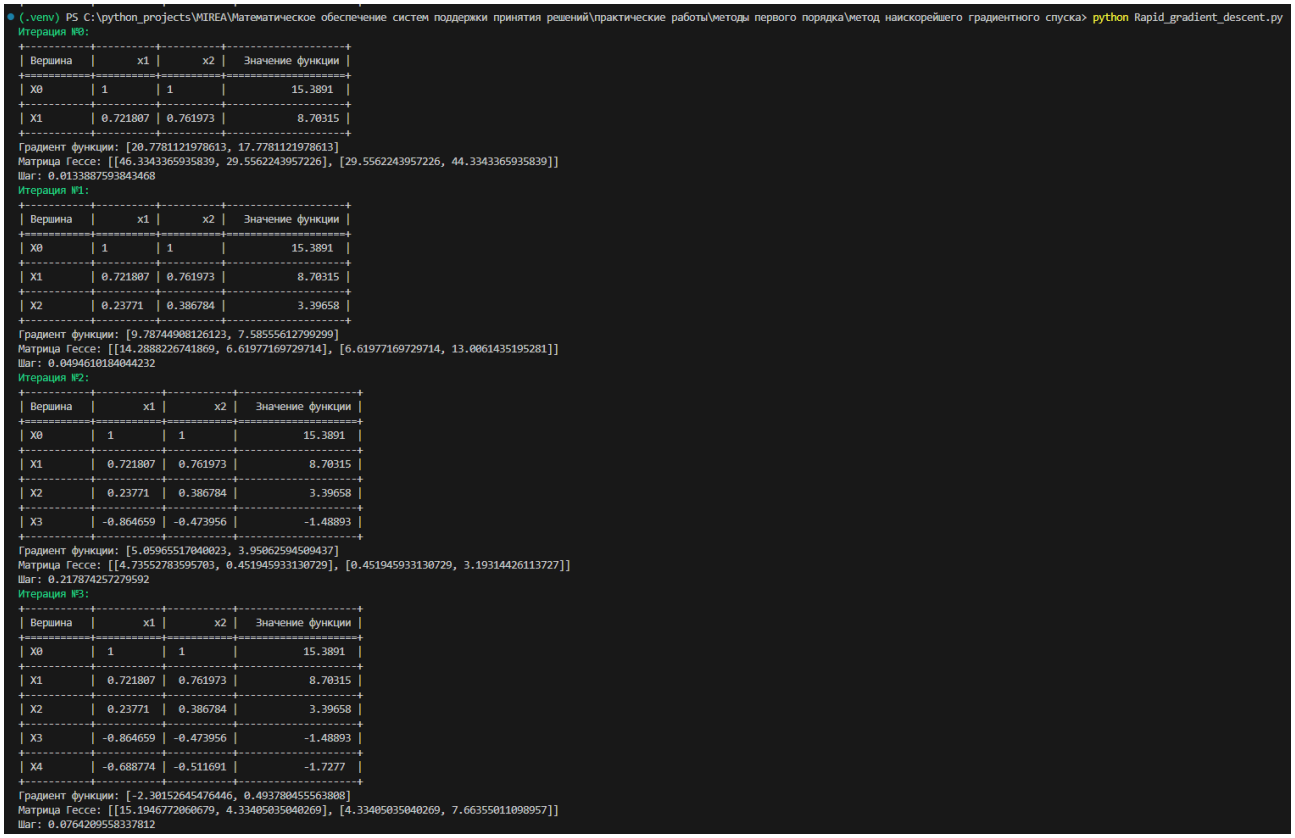

Рисунок 2.4.1 – Начальные итерации работы метода наискорейшего градиентного спуска

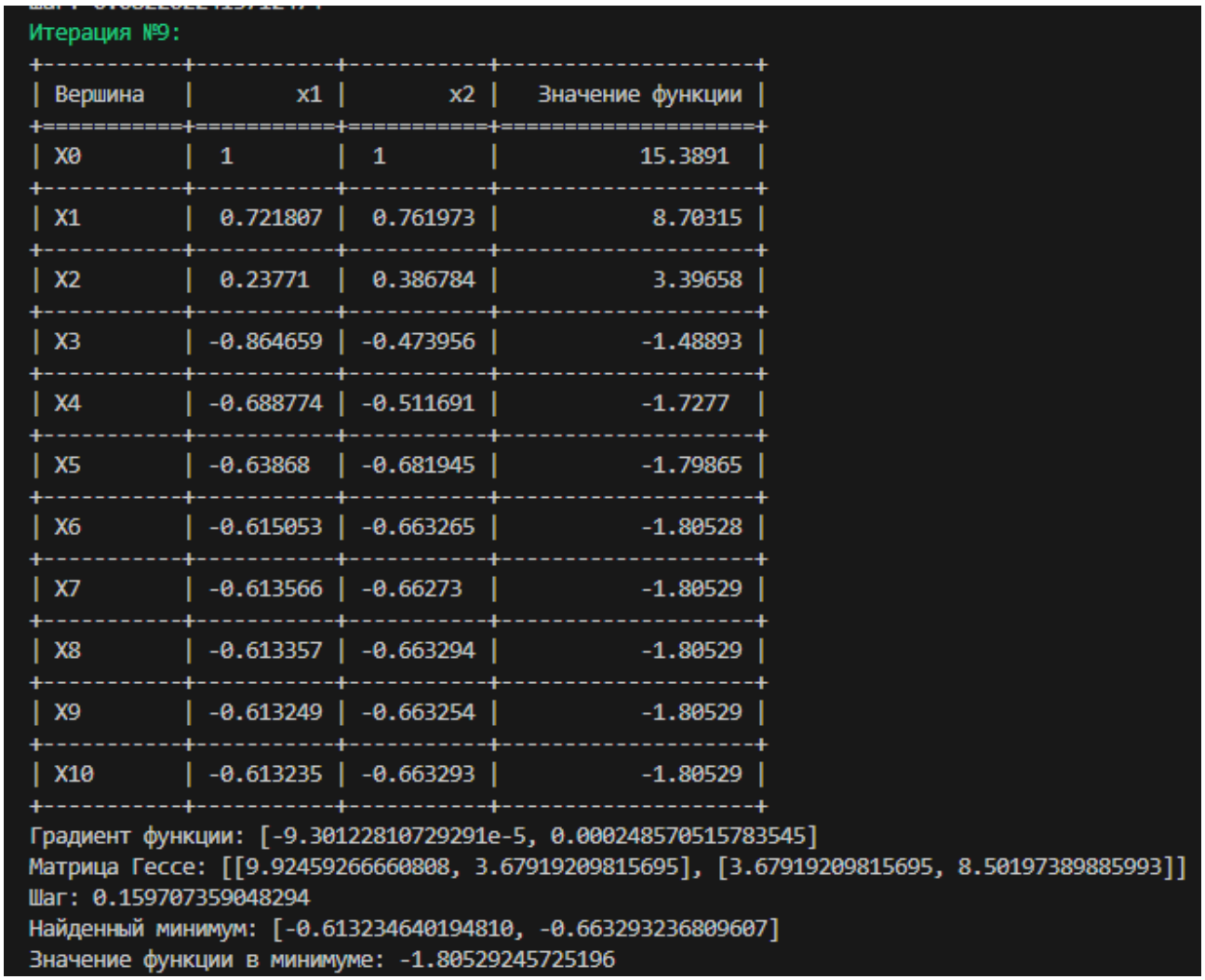

Рисунок 2.4.2 – Последняя итерация метода наискорейшего градиентного спуска

3 МЕТОД НЬЮТОНА

3.1 Описание алгоритма

В основе метода Ньютона лежит квадратичная аппроксимация целевой функции. Последовательность итераций строится таким образом, чтобы во вновь получаемой точке градиент аппроксимирующей функции обращался в нуль.

Последовательность приближений строится в соответствии с Формулой 3.1.1.

$$\overline{x^{(k+1)}} = \overline{x^{(k)}} + \overline{p^{(k)}}, \tag{3.1.1}$$

где k – номер итерации ($k = 0, 1, \dots$);

$\overline{x^{(0)}}$ – начальное приближение;

$\overline{p^{(k)}} = -H^{-1}\overline{x^{(k)}}\nabla f\left(\overline{x^{(k)}}\right)$ – вектор направления спуска.

Направление спуска $\overline{p^{(k)}}$ ведет к убыванию целевой функции только при положительной определенности матрицы Гессе $H\left(\overline{x^{(k)}}\right) > 0$. В тех итерациях, в которых матрица Гессе отрицательно определена $H\left(\overline{x^{(k)}}\right) < 0$, последовательность приближений к точке минимума строится по методу наискорейшего градиентного спуска. С этой целью проводится замена вектора направления спуска на антиградиентное $\overline{p^{(k)}} = -h_k\nabla f\left(\overline{x^{(k)}}\right)$.

Величина шага h_k вычисляется по Формуле 3.1.2.

$$h_k = \frac{\left(\nabla f\left(\overline{x^{(k)}}\right), \overline{p^{(k)}}\right)}{\left(H\left(\overline{x^{(k)}}\right)\overline{p^{(k)}}, \overline{p^{(k)}}\right)}, \tag{3.1.2}$$

где $H\left(\overline{x^{(k)}}\right)$ – матрица Гессе, вычисленная в точке $\overline{x^{(k)}}$.

Алгоритм метода Ньютона:

1. Задать размерность задачи оптимизации n, координаты начальной точки $\overline{x^{(0)}} = (x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)})$, точности поиска ε.
2. Положить счетчик числа $k = 0$.
3. Определить направление вектора градиента целевой функции $f(\bar{x})$: $\nabla f\left(\overline{x^{(k)}}\right) = \left(\frac{\partial f\left(\overline{x^{(k)}}\right)}{\partial x_1}, \frac{\partial f\left(\overline{x^{(k)}}\right)}{\partial x_2}, \dots, \frac{\partial f\left(\overline{x^{(k)}}\right)}{\partial x_n}\right)$ в точке $\overline{x^{(k)}} = (x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)})$. Для вычисления координат вектора градиента использовать разностную формулу первых частных производных.
4. Проверить условие окончания поиска (Формула 3.1.3).

$$\left\|\nabla f\left(\overline{x^{(k)}}\right)\right\| = \sqrt{\sum_{i=1}^{n}\left(\frac{\partial f\left(\overline{x^{(k)}}\right)}{\partial x_i}\right)^2} \le \varepsilon \qquad (3.1.3)$$

   Если условие выполнено, то расчет окончен $\overline{x^*} = \overline{x^{(k)}}$, иначе прейти к пункту 5.
5. Сформировать матрицу Гессе $H\left(\overline{x^{(k)}}\right)$, используя разностные формулы вычисления вторых и смешанных производных.
6. Проверить положительную определенность матрицы Гессе $H\left(\overline{x^{(k)}}\right)$. Если матрица положительно определена $H\left(\overline{x^{(k)}}\right) > 0$, то перейти к пункту 7, иначе — к пункту 8.
7. Определить координаты точки $\overline{x^{(k+1)}} = \overline{x^{(k)}} - H^{-1}\overline{x^{(k)}}\nabla f\left(\overline{x^{(k)}}\right)$ и перейти к пункту 10.
8. Вычислить шаг h_k по Формуле 3.1.2, используя результаты вычислений пункта 3 и разностные формулы вторых и смешанных производных.
9. Определить координаты точки $\overline{x^{(k+1)}} = \overline{x^{(k)}} - h_k\nabla f\left(\overline{x^{(k)}}\right)$ по методу наискорейшего градиентного спуска.
10. Положить $k = k + 1$ и перейти к пункту 3.

3.2 Постановка задачи

Цель работы: реализовать безусловную оптимизацию функции методом второго порядка.

Задачи: изучить метод Ньютона, выбрать функцию для оптимизации (нахождение глобального минимума), произвести ручной расчёт итерации алгоритма, разработать программную реализацию метода наискорейшего градиентного спуска.

Выбранная функция для оптимизации представлена Формулой 1.2.1.

График выбранной функции представлен на Рисунке 1.2.1.

Глобальный минимум приблизительно равен -1.8.

3.3 Ручной расчёт

Точность поиска ε принята равной 0.0001. Начальная точка $\overline{x^{(0)}} = (1; 1)$.

Найдем градиент функции в произвольной точке $\overline{x^{(k)}} = (x_1^{(k)}, x_2^{(k)})$:

$$\nabla f\left(\overline{x^{(k)}}\right) = \left(\frac{\partial f\left(\overline{x^{(k)}}\right)}{\partial x_1}; \frac{\partial f\left(\overline{x^{(k)}}\right)}{\partial x_2}\right) = (2x_1 * e^{x_1^2+x_2^2} + 2x_1 + 4; 2x_2 * e^{x_1^2+x_2^2} + 3)$$

Вычислим градиент функции $\nabla f\left(\overline{x^{(k)}}\right)$ в начальной точке $\overline{x^{(0)}}$:

$$\nabla f\left(\overline{x^{(0)}}\right) = (20.77; 17.77)$$

Вычислим матрицу Гессе в точке $\overline{x^{(0)}}$:

$$H\left(\overline{x^{(0)}}\right) = \begin{pmatrix} 46.33 & 29.55 \\ 29.55 & 44.33 \end{pmatrix}$$

Так как знаки угловых миноров строго положительны, то согласно критерию Сильвестра, матрица Гессе положительна определена. Следовательно, направление спуска определяем по Формуле Ньютона из пункта 7.

Вычислим координаты новой точки:

$$\overline{x^{(1)}} = \overline{x^{(0)}} - H^{-1}\overline{x^{(0)}}\nabla f\left(\overline{x^{(0)}}\right) = \binom{1}{1} - \begin{pmatrix} 0.037 & -0.025 \\ -0.025 & 0.039 \end{pmatrix}\binom{20.77}{17.77}$$
$$= (0.66; 0.82)$$

Значение функции в найденной точке:

$$f\left(\overline{x^{(1)}}\right) = 8.62$$

Норма вектора градиента в точке равна $\overline{x^{(1)}}$ равна 12.38, больше, чем заданная точность поиска, поэтому осуществляется переход ко второй итерации.

3.4 Программная реализация

Написан класс NewtonMethod, который реализует метод Ньютона. Конструктор класса принимает в качестве параметров ссылку на функцию, начальную точку, точность поиска.

Код реализации метода Ньютона представлен в Приложении В.

Результат работы программы представлен на Рисунках 3.4.1–3.4.2.

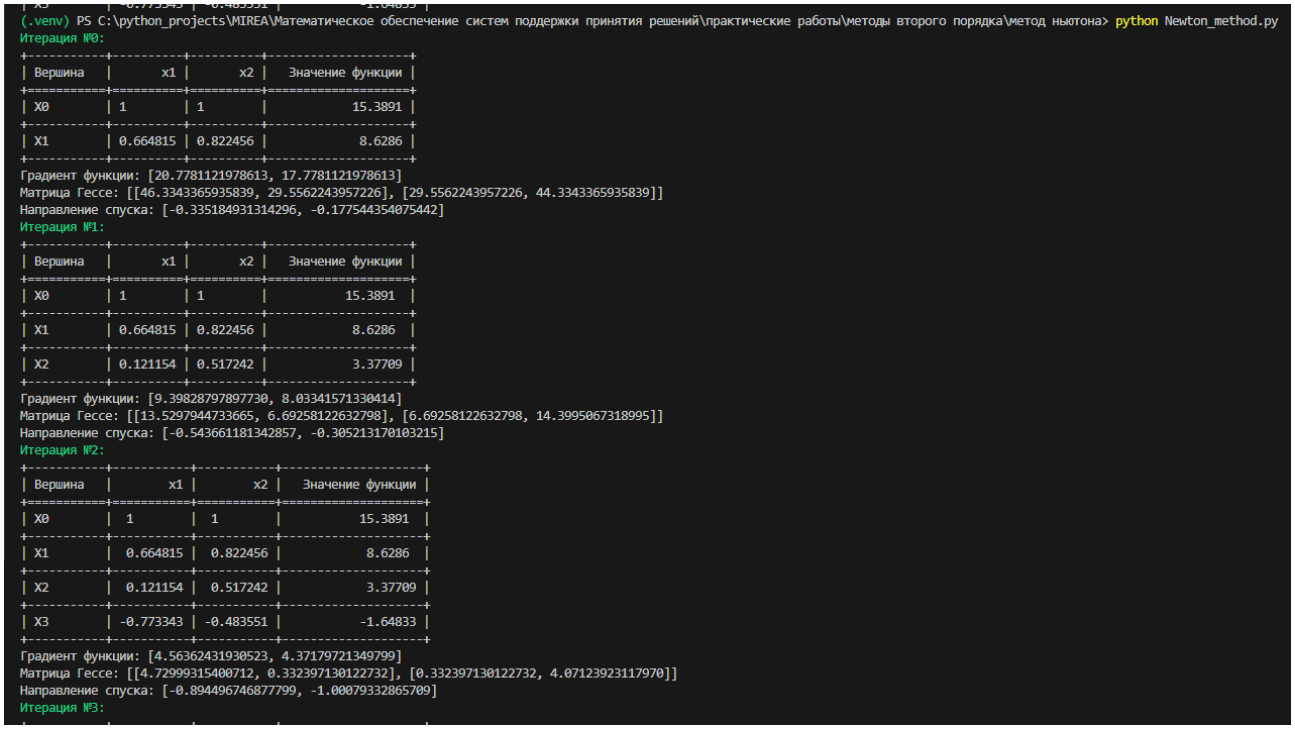

Рисунок 3.4.1 – Начальные итерации работы метода Ньютона

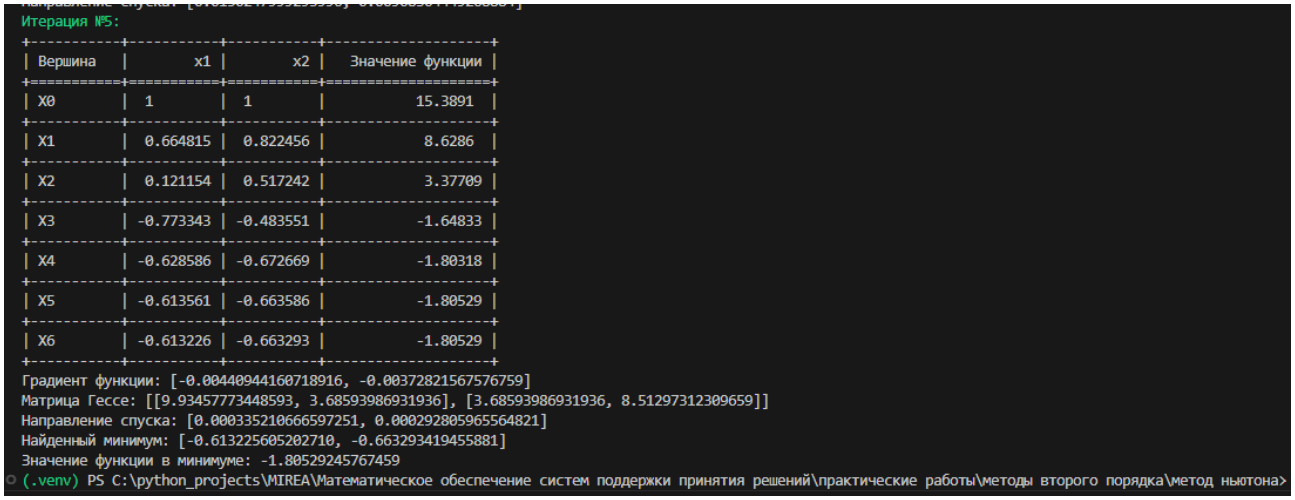

Рисунок 3.4.2 – Последняя итерация метода Ньютона

ЗАКЛЮЧЕНИЕ

В рамках практических работ изучены и реализованы методы нулевого, первого и второго порядка, которые являются основой для решения задач многомерной оптимизации.

Методы нулевого порядка, такие как метод Хука-Дживса, симплексный метод и метод Нелдера-Мида, позволяют находить минимум целевой функции, используя только значения самой функции, без необходимости вычисления её производных. Эти методы особенно полезны в случаях, когда функция не является гладкой или её производные трудно вычислить.

Методы первого порядка, такие как градиентный спуск, метод наискорейшего спуска и метод покоординатного спуска, используют информацию о первых производных целевой функции. Эти методы позволяют более эффективно находить минимум функции, особенно в случаях, когда функция имеет сложный рельеф. В ходе выполнения практических работ изучены различные модификации градиентных методов, включая метод Флетчера-Ривса, который использует сопряженные направления для ускорения сходимости.

Методы второго порядка, такие как метод Ньютона и его модификации, используют информацию о вторых производных целевой функции, что позволяет достичь более высокой скорости сходимости по сравнению с методами первого порядка. Эти методы особенно эффективны для задач с квадратичными функциями, где они могут найти минимум за конечное число итераций. В рамках курса рассмотрены как классический метод Ньютона, так и его модификации, такие как метод Ньютона-Рафсона, который обеспечивает более устойчивую сходимость.

Написаны программные реализации метода Хука-Дживса, наискорейшего градиентного спуска, Ньютона. С ростом порядка метода уменьшается количество итераций, необходимых для сходимости метода к минимуму.

СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Сорокин, А. Б. Введение в роевой интеллект: теория, расчеты и приложения [Электронный ресурс] : Учебно-методическое пособие / А. Б. Сорокин – Москва: Московский технологический университет (МИРЭА), 2019.
2. Сорокин, А. Б. Безусловная оптимизация. [Электронный ресурс] : учебно-метод. пособие / А. Б. Сорокин, О. В. Платонова, Л. М. Железняк — М. РТУ МИРЭА , 2020.
3. Сорокин, А. Б. Введение в генетические алгоритмы: теория, расчеты и приложения. [Электронный ресурс] : учебно-метод. пособие / А. Б. Сорокин — М. МИРЭА , 2018.
4. Хоботов, Е. Н. Математическое обеспечение систем поддержки принятия решений. [Электронный ресурс] : учебное пособие / Е. Н. Хоботов, Л. М. Железняк, Р. Э. Семенов .— М. : РТУ МИРЭА , 2024.

ПРИЛОЖЕНИЯ

Приложение А — Код реализации метода Хука-Дживса.

Приложение Б — Код реализации метода наискорейшего градиентного спуска.

Приложение В — Код реализации метода Ньютона.

Приложение А

Код реализации метода Хука-Дживса

Листинг А – Реализация метода Хука-Дживса

```
import math
from tabulate import tabulate
from typing import Callable, List

class HookeJeevesMethod:
    def __init__(self, fitness_function: Callable[..., float], x0: List[float],
h: float, d: float, m: float, epsilon: float):
        '''
        Инициализация метода Хука-Дживса для оптимизации функции.

        Параметры:
            fitness_function (Callable[..., float]): Целевая функция для
оптимизации.
            x0 (List[float]): Начальная точка (вектор).
            h (float): Начальный шаг для исследующего поиска.
            d (float): Коэффициент уменьшения шага.
            m (float): Ускоряющий множитель для поиска по образцу.
            epsilon (float): Точность для условия остановки.
        '''
        self.func = fitness_function
        self.n = len(x0)
        self.epsilon = epsilon
        self.h = h
        self.d = d
        self.m = m
        self.basis_vector = x0  # Базисная точка
        self.step = [h] * self.n  # Шаги по каждой координате
        self.arr = [[x0, self.func(*x0)]]  # Массив для хранения истории точек

    def _print_table(self) -> None:
        '''
        Вывод текущего состояния в виде таблицы.
        '''
        headers = ["Вершина"] + [f"x{i+1}" for i in range(self.n)] + ["Значение
функции"]
        table = [[f"X{i}"] + vertex[0] + [vertex[1]] for i, vertex in
enumerate(self.arr)]
        print(tabulate(table, headers=headers, tablefmt="grid"))

    def _exploratory_search(self, x: List[float]) -> List[float]:
        '''
        Исследующий поиск вокруг точки x.

        Параметры:
            x (List[float]): Текущая точка.

        Возвращает:
            List[float]: Новая точка после исследующего поиска.
        '''
        new_x = x.copy()
        for i in range(self.n):
            tmp_x = new_x.copy()
            tmp_x[i] += self.step[i]
            if self.func(*tmp_x) < self.func(*new_x):
                new_x = tmp_x
            else:
```

Продолжение Листинга А

```
                tmp_x[i] -= 2 * self.step[i]
                if self.func(*tmp_x) < self.func(*new_x):
                    new_x = tmp_x
        return new_x

    def _pattern_search(self, x: List[float], basis_vector: List[float]) ->
List[float]:
        '''
        Поиск по образцу.

        Параметры:
            x (List[float]): Текущая точка.
            basis_vector (List[float]): Базисная точка.

        Возвращает:
            List[float]: Новая точка после поиска по образцу.
        '''
        return [x[i] + self.m * (x[i] - basis_vector[i]) for i in range(self.n)]

    def optimize(self, show_iterations: bool = True) -> List[float]:
        '''
        Оптимизация целевой функции с использованием метода Хука-Дживса.

        Параметры:
            show_iterations (bool): Флаг, указывающий, нужно ли выводить
информацию о каждой итерации.

        Возвращает:
            List[float]: Координаты точки минимума.
        '''
        iteration = 0
        while True:
            if show_iterations:
                print('\033[92m' + f'Итерация №{iteration}:' + '\033[0m')
                self._print_table()

            # Исследующий поиск вокруг базисной точки
            new_x = self._exploratory_search(self.basis_vector)
            new_value = self.func(*new_x)

            if new_value >= self.func(*self.basis_vector):
                self.step = [step / self.d for step in self.step]
                if all(step < self.epsilon for step in self.step):
                    return self.basis_vector
            else:
                # Поиск по образцу
                pattern_x = self._pattern_search(new_x, self.basis_vector)
                pattern_value = self.func(*pattern_x)

                if pattern_value < new_value:
                    self.basis_vector = pattern_x
                else:
                    self.basis_vector = new_x

                self.arr.append([self.basis_vector,
self.func(*self.basis_vector)])

            iteration += 1

# def test_function(x1: float, x2: float) -> float:
#     return 2.8 * x2**2 + 1.9 * x1 + 2.7 * x1**2 + 1.6 - 1.9 * x2
```

Окончание Листинга А

```
def fitness_function(x1: float, x2: float) -> float:
    '''
    Пример целевой функции для оптимизации.

    Параметры:
        x1 (float): Первая переменная.
        x2 (float): Вторая переменная.

    Возвращает:
        float: Значение функции в точке (x1, x2).
    '''
    return x1 ** 2 + math.exp(x1 ** 2 + x2 ** 2) + 4 * x1 + 3 * x2

if __name__ == '__main__':
    optimizer = HookeJeevesMethod(fitness_function, x0 = [1, 1], h = 0.2, d = 2,
m = 2, epsilon = 0.0001)
    minimum = optimizer.optimize()
    print("Найденный минимум:", minimum)
    print("Значение функции в минимуме:", fitness_function(*minimum))
```

Приложение Б

Код реализации метода наискорейшего градиентного спуска

Листинг Б – Реализация метода наискорейшего градиентного спуска

```
import math
import sympy as sp
from tabulate import tabulate
from typing import Callable, List

class RapidGradientDescent:
    def __init__(self, fitness_function: Callable[..., float], x0: List[float],
epsilon: float):
        '''
        Инициализация метода наискорейшего градиентного спуска для оптимизации
функции.

        Параметры:
            fitness_function (Callable[..., float]): Целевая функция для
оптимизации.
            x0 (List[float]): Начальная точка (вектор).
            epsilon (float): Точность для условия остановки.
        '''
        self.func = fitness_function
        self.n = len(x0)
        self.epsilon = epsilon
        self.arr = [[x0, self.func(*x0)]]
        self._variables = sp.symbols(f"x1:{self.n + 1}")
        self._gradient_expr = [sp.diff(self.func(*self._variables), var) for var
in self._variables]
        self._hessian_expr = [[sp.diff(self._gradient_expr[i], var) for var in
self._variables] for i in range(self.n)]

    def _print_table(self) -> None:
        '''
        Вывод текущего состояния в виде таблицы.
        '''
        headers = ["Вершина"] + [f"x{i+1}" for i in range(self.n)] + ["Значение
функции"]
        table = [[f"X{i}"] + vertex[0] + [vertex[1]] for i, vertex in
enumerate(self.arr)]
        print(tabulate(table, headers=headers, tablefmt="grid"))

    def _compute_gradient(self, x: List[float]) -> List[float]:
        '''
        Вычисление градиента функции в точке x.

        Параметры:
            x (List[float]): Точка, в которой вычисляется градиент.

        Возвращает:
            List[float]: Градиент функции в точке x.
        '''
        return [g.evalf(subs={self._variables[i]: x[i] for i in range(self.n)})
for g in self._gradient_expr]

    def _compute_hessian(self, x: List[float]) -> List[List[float]]:
        '''
        Вычисление матрицы Гессе функции в точке x.

        Параметры:
```

Продолжение Листинга Б

```
            x (List[float]): Точка, в которой вычисляется матрица Гессе.

        Возвращает:
            List[List[float]]: Матрица Гессе функции в точке x.
        '''
        return [[h.evalf(subs={self._variables[i]: x[i] for i in range(self.n)})
for h in row] for row in self._hessian_expr]

    def _check_stopping_condition(self, gradient: List[float]) -> bool:
        '''
        Проверка условия остановки.

        Параметры:
            gradient (List[float]): Градиент функции в текущей точке.

        Возвращает:
            bool: True, если условие остановки выполнено, иначе False.
        '''
        norm = math.sqrt(sum(g ** 2 for g in gradient))
        return norm < self.epsilon

    def _compute_step(self, gradient: List[float], hessian: List[List[float]]) -
> float:
        '''
        Вычисление шага h_k.

        Параметры:
            gradient (List[float]): Градиент функции в текущей точке.
            hessian (List[List[float]]): Матрица Гессе функции в текущей точке.

        Возвращает:
            float: Оптимальный шаг h_k.
        '''
        numerator = sum(g ** 2 for g in gradient)

        denominator = 0.0
        for i in range(self.n):
            for j in range(self.n):
                denominator += gradient[i] * hessian[i][j] * gradient[j]

        return numerator / denominator

    def optimize(self, show_info: bool = True):
        '''
        Выполнение оптимизации методом наискорейшего градиентного спуска.

        Параметры:
            show_info (bool): Флаг для отображения информации о каждой итерации.

        Возвращает:
            List[float]: Точка минимума.
        '''
        x = self.arr[0][0]
        iteration = 0

        while True:
            gradient = self._compute_gradient(x)
            if self._check_stopping_condition(gradient):
                break

            hessian = self._compute_hessian(x)
```

Окончание Листинга Б

```
            h_k = self._compute_step(gradient, hessian)

            x_new = [x[i] - h_k * gradient[i] for i in range(self.n)]
            f_new = self.func(*x_new)

            self.arr.append([x_new, f_new])
            x = x_new

            if show_info:
                print('\033[92m' + f'Итерация №{iteration}:' + '\033[0m')
                self._print_table()
                print(f"Градиент функции: {gradient}")
                print(f"Матрица Гессе: {hessian}")
                print(f"Шаг: {h_k}")

            iteration += 1

        return x

def test_function(x1: float, x2: float) -> float:
    return 2.8 * x2**2 + 1.9 * x1 + 2.7 * x1**2 + 1.6 - 1.9 * x2

def fitness_function(x1: float, x2: float) -> float:
    '''
    Пример целевой функции для оптимизации.

    Параметры:
        x1 (float): Первая переменная.
        x2 (float): Вторая переменная.

    Возвращает:
        float: Значение функции в точке (x1, x2).
    '''
    return x1 ** 2 + sp.exp(x1 ** 2 + x2 ** 2) + 4 * x1 + 3 * x2

if __name__ == '__main__':
    optimizer = RapidGradientDescent(fitness_function, x0=[1, 1],
epsilon=0.0001)
    minimum = optimizer.optimize()
    print("Найденный минимум:", minimum)
    print("Значение функции в минимуме:", fitness_function(*minimum))
```

Приложение В

Код реализации метода Ньютона

Листинг В – Реализация метода Ньютона

```
import math
import sympy as sp
from tabulate import tabulate
from typing import Callable, List

class NewtonMethod:
    def __init__(self, fitness_function: Callable[..., float], x0: List[float],
epsilon: float):
        '''
        Инициализация метода Ньютона для оптимизации функции.

        Параметры:
            fitness_function (Callable[..., float]): Целевая функция для
оптимизации.
            x0 (List[float]): Начальная точка (вектор).
            epsilon (float): Точность для условия остановки.
        '''
        self.func = fitness_function
        self.n = len(x0)
        self.epsilon = epsilon
        self.arr = [[x0, self.func(*x0)]]
        self._variables = sp.symbols(f"x1:{self.n + 1}")
        self._func_expr = self.func(*self._variables)
        self._gradient_expr = [sp.diff(self._func_expr, var) for var in
self._variables]
        self._hessian_expr = [[sp.diff(g, var) for var in self._variables] for g
in self._gradient_expr]

    def _print_table(self) -> None:
        '''
        Вывод текущего состояния в виде таблицы.
        '''
        headers = ["Вершина"] + [f"x{i+1}" for i in range(self.n)] + ["Значение
функции"]
        table = [[f"X{i}"] + vertex[0] + [vertex[1]] for i, vertex in
enumerate(self.arr)]
        print(tabulate(table, headers=headers, tablefmt="grid"))

    def _compute_gradient(self, x: List[float]) -> List[float]:
        '''
        Вычисление градиента функции в точке x.

        Параметры:
            x (List[float]): Точка, в которой вычисляется градиент.

        Возвращает:
            List[float]: Градиент функции в точке x.
        '''
        return [g.evalf(subs={self._variables[i]: x[i] for i in range(self.n)})
for g in self._gradient_expr]

    def _compute_hessian(self, x: List[float]) -> List[List[float]]:
        '''
        Вычисление матрицы Гессе функции в точке x.
```

Продолжение Листинга В

```
        Параметры:
            x (List[float]): Точка, в которой вычисляется матрица Гессе.

        Возвращает:
            List[List[float]]: Матрица Гессе функции в точке x.
        '''
        return [[h.evalf(subs={self._variables[i]: x[i] for i in range(self.n)})
for h in row] for row in self._hessian_expr]

    def _check_stopping_condition(self, gradient: List[float]) -> bool:
        '''
        Проверка условия остановки.

        Параметры:
            gradient (List[float]): Градиент функции в текущей точке.

        Возвращает:
            bool: True, если условие остановки выполнено, иначе False.
        '''
        norm = math.sqrt(sum(g ** 2 for g in gradient))
        return norm < self.epsilon

    def _is_positive_definite(self, matrix: List[List[float]]) -> bool:
        '''
        Проверка положительной определённости матрицы.

        Параметры:
            matrix (List[List[float]]): Матрица для проверки.

        Возвращает:
            bool: True, если матрица положительно определена, иначе False.
        '''
        for i in range(1, self.n + 1):
            sub_matrix = [row[:i] for row in matrix[:i]]
            det = sp.Matrix(sub_matrix).det()
            if det <= 0:
                return False
        return True

    def optimize(self, show_info: bool = True):
        '''
        Выполнение оптимизации методом Ньютона.

        Параметры:
            show_info (bool): Флаг для отображения информации о каждой итерации.

        Возвращает:
            List[float]: Точка минимума.
        '''
        x = self.arr[0][0]
        iteration = 0

        while True:
            gradient = self._compute_gradient(x)
            if self._check_stopping_condition(gradient):
                break

            hessian = self._compute_hessian(x)

            if self._is_positive_definite(hessian):
                # Если матрица положительно определена, используем метод Ньютона
```

Окончание Листинга В

```
                hessian_inv = sp.Matrix(hessian).inv()
                hessian_inv = [[float(hessian_inv[i, j]) for j in range(self.n)]
for i in range(self.n)]
                direction = [-sum(hessian_inv[i][j] * gradient[j] for j in
range(self.n)) for i in range(self.n)]

                # Обновляем точку с шагом 1 (стандартный метод Ньютона)
                x_new = [x[i] + direction[i] for i in range(self.n)]
            else:
                # Если матрица не положительно определена, используем метод
наискорейшего спуска
                direction = [-g for g in gradient]

                # Вычисляем шаг h_k по формуле из шага 8
                numerator = sum(gradient[i] * direction[i] for i in
range(self.n))
                denominator = sum(sum(hessian[i][j] * direction[j] for j in
range(self.n)) * direction[i] for i in range(self.n))
                h_k = numerator / denominator

                # Обновляем точку с учетом шага h_k
                x_new = [x[i] + h_k * direction[i] for i in range(self.n)]

            f_new = self.func(*x_new)
            self.arr.append([x_new, f_new])
            x = x_new

            if show_info:
                print('\033[92m' + f'Итерация №{iteration}:' + '\033[0m')
                self._print_table()
                print(f"Градиент функции: {gradient}")
                print(f"Матрица Гессе: {hessian}")
                print(f"Направление спуска: {direction}")
                if not self._is_positive_definite(hessian):
                    print(f"Шаг h_k: {h_k}")

            iteration += 1
        return x

def test_function(x1: float, x2: float) -> float:
    return 2.8 * x2**2 + 1.9 * x1 + 2.7 * x1**2 + 1.6 - 1.9 * x2

def fitness_function(x1: float, x2: float) -> float:
    '''
    Пример целевой функции для оптимизации.

    Параметры:
        x1 (float): Первая переменная.
        x2 (float): Вторая переменная.

    Возвращает:
        float: Значение функции в точке (x1, x2).
    '''
    return x1 ** 2 + sp.exp(x1 ** 2 + x2 ** 2) + 4 * x1 + 3 * x2

if __name__ == '__main__':
    optimizer = NewtonMethod(fitness_function, x0=[1, 1], epsilon=0.0001)
    minimum = optimizer.optimize()
    print("Найденный минимум:", minimum)
    print("Значение функции в минимуме:", fitness_function(*minimum))
```