Министерство науки и высшего образования РФ
ФГАОУ ВПО
Национальный исследовательский технологический университет «МИСИС»

Институт компьютерных наук (ИКН)

Кафедра Инфокоммуникационных технологий (ИКТ)

Отчет по контрольной работе №2
по дисциплине «Методы оптимизации»

на тему «Методы многомерного поиска»

Выполнил:
студент группы БИСТ-22-3

Котов С. С.

Проверил:
доц. каф. ИКТ

Мокрова Н. В.

Москва, 2025

Цель работы: ознакомиться с методами многомерного поиска (метод Хука-Дживса, метод наискорейшего спуска и метод сопряженных градиентов). Сравнить различные алгоритмы по эффективности на тестовых примерах.

Задание:

1. Исследовать функцию $f(x) = (3 + y^2)^2 + (x^2 - 25)^2$ (11 вариант), графически найти решение задачи поиска экстремума.
2. Реализовать численные методы решения задачи поиска безусловного экстремума функции согласно вариантам.
3. Обосновать выбор параметров алгоритмов.
4. Сделать выводы об эффективности методов оптимизации.

Теоретические сведения

Метод Хука-Дживса.

Название: Метод Хука-Дживса, также известный как метод конфигураций (pattern search method).

Категория: Метод прямого поиска (Direct Search Method). Это означает, что метод не использует производные (градиенты или Гессианы) функции для поиска минимума. Он опирается исключительно на сравнение значений функции в разных точках.

Применение: Для безусловной оптимизации многомерных функций f(x) (где x - вектор переменных).

Основная Идея:

Метод Хука-Дживса имитирует интуитивный подход к поиску "дна долины": сначала совершаются "исследовательские" шаги вокруг текущей лучшей точки по каждой координате, чтобы найти ближайшее локальное улучшение. Если исследование оказалось успешным, то предполагается, что направление от предыдущей лучшей точки к новой лучшей точке является перспективным, и делается более "смелый" "шаблонный" шаг (или шаг по образцу) в этом направлении. Если исследование не привело к улучшению, шаг исследования уменьшается, и процесс повторяется в более узкой окрестности.

Метод использует две основные концепции:

1. **Базовая точка (xb, base point):** Лучшая точка, найденная на данный момент в процессе поиска шаблона.
2. **Исследуемая точка (xt, trial point):** Точка, используемая для выполнения исследовательских шагов.

Алгоритм:

Метод включает две основные фазы: фазу исследования (Exploratory Move) и фазу поиска шаблона (Pattern Move).

1. Инициализация:

- Выбирается начальная точка x0. Она становится первой базовой точкой: xb = x0.
- Исследуемая точка также инициализируется как базовая точка: xt = xb.
- Выбирается начальный размер шага исследования delta > 0.
- Выбирается коэффициент сжатия шага rho (обычно 0 < rho < 1, часто 0.5).
- Устанавливается критерий останова (например, когда delta становится меньше некоторого малого значения delta_tolerance).

2. Основной цикл (пока не выполнен критерий останова):

- Фаза Исследования (Exploratory Move):
  - Начинается из текущей исследуемой точки xt.
  - Для каждой координаты i от 1 до n (где n - размерность задачи):
    - Попытка движения в положительном направлении: Вычисляется точка x_plus = xt с i-й координатой увеличенной на delta. Если f(x_plus) < f(xt), то xt обновляется до x_plus.
    - Если улучшение не найдено в положительном направлении, попытка движения в отрицательном направлении: Вычисляется точка x_minus = xt с i-й координатой уменьшенной на delta. Если f(x_minus) < f(xt), то xt обновляется до x_minus.
    - Если ни одно из направлений не привело к улучшению, i-я координата xt остается прежней.
  - По завершении цикла по всем координатам, xt содержит лучшую точку, найденную в ходе исследования вокруг начальной xt с шагом delta.
- Фаза Поиска Шаблона (Pattern Move) и Обновление/Сжатие Шага:
  - Сравнивается значение функции в новой исследуемой точке xt (полученной после фазы исследования) со значением функции в базовой точке xb (с начала текущей итерации основного цикла).
  - Если f(xt) < f(xb) (Исследование успешно привело к точке лучше базовой):
    - Произошло улучшение.
    - Старая базовая точка xb_old = xb.

        - Новая базовая точка становится результатом исследования: xb = xt.
        - Делается "шаблонный шаг" в предполагаемом направлении улучшения. Направление шаблона — это вектор от старой базовой точки к новой базовой точке: p = xb - xb_old.
        - Новая исследуемая точка для следующей итерации определяется как "шаблонный шаг" от новой базовой точки: xt = xb + p.
        - Размер шага исследования delta остается прежним.
    - Если f(xt) >= f(xb) (Исследование не привело к точке лучше базовой):
        - Улучшения относительно базовой точки не найдено.
        - Базовая точка xb остается прежней.
        - Шаблонный шаг не делается.
        - Размер шага исследования delta сжимается: delta = delta * rho.
        - Исследуемая точка для следующей итерации сбрасывается к текущей базовой точке: xt = xb.

3. Критерий Останова:

Цикл завершается, когда размер шага исследования delta становится меньше заданной малой величины delta_tolerance. Предполагается, что в этом случае алгоритм локализовал минимум в области, определяемой размером delta. Также обычно добавляют критерии по максимальному числу итераций или максимальному числу вызовов функции.

Преимущества:

- **Простота:** Концептуально прост и легко реализуем.
- **Не требует производных:** Это главное преимущество. Метод может быть применен к функциям, для которых производные сложно или невозможно вычислить (например, негладкие, с разрывами, "шумные" функции).
- **Робастность:** Относительно устойчив к шумам в вычислениях функции по сравнению с методами, основанными на производных.

Недостатки:

- **Медленная сходимость:** Скорость сходимости обычно гораздо медленнее, чем у градиентных методов или методов второго порядка, особенно вблизи минимума. Сходимость, как правило, линейная.

- **Неэффективен на "гребнях":** Может испытывать трудности при поиске минимума в узких, вытянутых долинах или на "гребнях" из-за своих координатных исследовательских шагов.
- **Чувствительность к параметрам:** Выбор начального delta и rho может влиять на производительность.

Применимость:

Метод Хука-Дживса хорошо подходит для задач оптимизации, где вычисление градиента затруднено или невозможно, а также как простой и надежный метод для начала исследования функции. Однако для гладких функций, где градиент легко доступен, градиентные методы или методы второго порядка (такие как Ньютон или BFGS) обычно сходятся значительно быстрее.

Метод Наискорейшего спуска.

Название: Метод наискорейшего спуска.

Категория: Градиентный метод первого порядка (Gradient Method, First-order method). Это означает, что метод использует информацию о первых производных функции (градиент) для определения направления поиска минимума.

Применение: Для безусловной оптимизации дифференцируемых многомерных функций f(x).

Основная Идея:

Основной принцип метода наискорейшего спуска основан на том, что антиградиент (-∇f(x)) в точке x указывает направление наибольшего убывания функции в этой точке. Интуитивно, чтобы спуститься к минимуму как можно быстрее, нужно двигаться именно в этом направлении.

Метод итеративно движется от текущей точки в направлении, противоположном градиенту, совершая при этом шаг некоторой длины, который находится путем минимизации функции вдоль выбранного направления.

Математическая Формулировка:

Итеративная процедура метода описывается формулой:

- x_{k+1} = x_k - α_k ∇f(x_k)
- Где:
- x_k - текущая точка на k-й итерации.
- x_{k+1} - следующая точка на (k+1)-й итерации.
- ∇f(x_k) - градиент функции f в точке x_k.

- -∇f(x_k) - направление наискорейшего спуска (обозначим его как p_k = -∇f(x_k)).
- α_k > 0 - длина шага (шаг спуска) на k-й итерации. Это скалярное значение, которое определяет, насколько далеко мы двигаемся в направлении p_k.

Определение Длины Шага (α_k): Поиск Шага (Line Search)

Выбор оптимальной длины шага α_k на каждой итерации является критически важным для эффективности метода. Идеальный α_k - это тот, который минимизирует функцию вдоль направления p_k:

α_k = argmin_{α >= 0} f(x_k + α * p_k)

Задача поиска α_k называется задачей одномерной минимизации или поиском шага (line search).

Существуют разные стратегии поиска шага:

1. Точный поиск шага (Exact Line Search): Находится α_k, который точно минимизирует f вдоль p_k. Это может быть вычислительно затратно.
2. Неточный поиск шага (Inexact Line Search): Находится α_k, который удовлетворяет некоторым условиям, гарантирующим достаточное уменьшение функции и прогресс, но без необходимости точной минимизации. Наиболее распространенные условия:
    - **Условие Армихо (Правило достаточного убывания):** f(x_k + α * p_k) <= f(x_k) + c1 * α * ∇f(x_k)^T p_k (с 0 < c1 < 1, обычно очень малое, например, 1e-4). Это гарантирует, что мы спустились достаточно.
    - **Условие Вульфа (Curvature Condition):** ∇f(x_k + α * p_k)^T p_k >= c2 * ∇f(x_k)^T p_k (с c1 < c2 < 1). Это гарантирует, что шаг не слишком мал и мы прошли достаточно далеко от "плоского" места.
    - **Правило сильного Вульфа:** Дополнительное условие для предотвращения слишком больших шагов.
    - **Backtracking Line Search (Поиск с возвратом):** Простая и популярная неточная процедура, основанная на условии Армихо. Начинаем с некоторого начального шага alpha_init (часто 1.0) и уменьшаем его (alpha = alpha * rho, где 0 < rho < 1, часто 0.5) до тех пор, пока не выполнится условие Армихо.

Алгоритм Метода Наискорейшего Спуска (с неточным поиском шага):

1. **Инициализация:** Выбрать начальную точку x0, допустимую погрешность tolerance (например, по норме градиента), максимальное число итераций max_iter, параметры для поиска шага (alpha_init, c, rho).
2. **Цикл Итераций (k = 0, 1, 2, ... до max_iter):**

   a. Вычислить градиент g_k = ∇f(x_k).

   b. Вычислить норму градиента: norm_g_k = ||g_k||.

   c. **Проверка критерия останова:** Если norm_g_k < tolerance, остановить алгоритм. Текущая точка x_k является найденным локальным минимумом (или стационарной точкой).

   d. Определить направление поиска: p_k = -g_k.

   e. Найти длину шага α_k с помощью процедуры поиска шага (например, Backtracking Line Search), которая приблизительно минимизирует f(x_k + α * p_k) по α.

   f. **Обновление точки:** Вычислить новую точку x_{k+1} = x_k + α_k * p_k.
3. **Завершение:** Если цикл завершился по max_iter, алгоритм не сошелся за отведенное количество итераций.

Сходимость:

- Метод наискорейшего спуска гарантированно сходится к локальному минимуму для широкого класса функций (например, непрерывно дифференцируемых функций с липшицевым градиентом) при использовании подходящей процедуры поиска шага (например, удовлетворяющей условиям Вульфа).
- Скорость сходимости метода наискорейшего спуска является линейной. Это означает, что ошибка уменьшается примерно в постоянное число раз на каждой итерации.
- Сходимость может быть очень медленной (медленной линейной) для плохо обусловленных задач, где линии уровня функции сильно вытянуты (например, в узких, глубоких долинах). В таких случаях метод имеет тенденцию "зигзагообразно" двигаться по долине, делая шаги, почти перпендикулярные оси долины, вместо быстрого спуска вдоль нее. Это связано с тем, что последовательные направления наискорейшего спуска (при использовании точного поиска шага) оказываются ортогональными друг другу.

Вычислительная Стоимость:

- На каждой итерации требуется вычислить градиент $\nabla f(x_k)$. Если функция n-мерная, градиент имеет n компонент.
- Процедура поиска шага требует нескольких вызовов функции f. Количество этих вызовов зависит от выбранной стратегии поиска шага и свойств функции.

Преимущества:

- **Простота:** Один из самых простых градиентных методов для понимания и реализации.
- **Требует только первых производных:** Не нужна матрица Гессе или ее аппроксимации.
- **Низкая стоимость итерации (по сравнению с методами второго порядка):** Вычисление градиента обычно дешевле, чем вычисление или аппроксимация Гессиана.
- **Гарантия локальной сходимости:** Подходит для нахождения локальных минимумов.

Недостатки:

- **Медленная сходимость:** Линейная скорость сходимости, может быть очень медленной для плохо обусловленных задач.
- **Эффект "зигзага":** Неэффективное движение в узких долинах.
- Может остановиться в стационарной точке (где градиент равен нулю), которая не является минимумом (например, седловая точка).

Сравнение:

Метод наискорейшего спуска является базовым градиентным методом. Он является хорошей отправной точкой для понимания градиентных подходов, но на практике часто превосходится более продвинутыми методами, такими как метод сопряженных градиентов (который избегает эффекта зигзага для квадратичных функций) или квазиньютоновские методы (например, BFGS), которые аппроксимируют информацию второго порядка для ускорения сходимости, но при этом не требуют явного вычисления Гессиана. Однако из-за своей простоты он остается важным для теоретического анализа и может быть достаточным для простых задач или как часть более сложных алгоритмов.

Метод сопряженных градиентов

Название: Метод сопряженных градиентов.

Категория: Градиентный метод первого порядка (Gradient Method, First-order method). Как и метод наискорейшего спуска, он использует информацию о градиенте функции. Однако он строит направления поиска более эффективно.

Применение:

- В первую очередь, для решения систем линейных алгебраических уравнений вида Ax = b, где A - симметричная и положительно определенная (СПД) матрица. В этом случае метод сходится к точному решению за конечное число итераций (не более n, где n - размерность задачи, в идеальных условиях без ошибок округления).
- Расширения метода используются для безусловной оптимизации нелинейных многомерных функций f(x). Это то, что мы реализовали в коде.

Основная Идея (для оптимизации нелинейных функций):

Метод сопряженных градиентов для нелинейных задач можно рассматривать как улучшение метода наискорейшего спуска. В то время как наискорейший спуск всегда движется в направлении антиградиента (которое на каждой итерации ортогонально предыдущему направлению, если используется точный поиск шага), метод сопряженных градиентов строит набор сопряженных направлений поиска.

Что такое сопряженные направления?

Для задачи минимизации квадратичной функции f(x) = 1/2 x^T A x - b^T x, где A - СПД матрица, два вектора p_i и p_j считаются сопряженными по отношению к матрице A, если p_i^T A p_j = 0 при i != j.

Ключевое свойство сопряженных направлений для квадратичных функций: если мы выполняем поиск шага вдоль набора n A-сопряженных направлений в n-мерном пространстве, мы гарантированно найдем точный минимум за не более чем n шагов.

Метод сопряженных градиентов для квадратичных функций строит именно такой набор направлений и, по сути, выполняет минимизацию последовательно вдоль каждого из этих направлений. Примечательно, что эти сопряженные направления могут быть построены только с использованием текущего градиента и предыдущего направления поиска, без необходимости использования матрицы A явно (только через произведение A*p, что в оптимизации соответствует вычислению

Гессиана*направление, или в случае нелинейных функций - аппроксимациям).

Переход к Нелинейной Оптимизации:

Для нелинейных функций f(x), у нас нет фиксированной матрицы A. Однако вблизи локального минимума любая гладкая функция может быть аппроксимирована квадратичной (с матрицей A, равной Гессиану в минимуме). Методы сопряженных градиентов для нелинейных задач адаптируют идеи из квадратичного случая, используя текущий градиент для построения направлений, которые частично сохраняют свойство сопряженности, по крайней мере, локально или в некотором смысле.

Существует несколько формул для вычисления коэффициента beta_k, который определяет, насколько новое направление p_{k+1} "наследует" от предыдущего направления p_k. Наиболее известные формулы:

1. **Флетчер-Ривс (Fletcher-Reeves, FR):** beta_k = (∇f_{k+1}^T ∇f_{k+1}) / (∇f_k^T ∇f_k) = ||g_{k+1}||² / ||g_k||²
2. **Полак-Рибьер-Полиак (Polak-Ribière-Polyak, PRP):** beta_k = (∇f_{k+1}^T (∇f_{k+1} - ∇f_k)) / (∇f_k^T ∇f_k) = g_{k+1}^T (g_{k+1} - g_k) / ||g_k||² (Часто используется не beta_k а max(0, beta_k) для гарантии сходимости)
3. **Хестенс-Стифель (Hestenes-Stiefel, HS):** beta_k = (∇f_{k+1}^T (∇f_{k+1} - ∇f_k)) / (p_k^T (∇f_{k+1} - ∇f_k)) = g_{k+1}^T (g_{k+1} - g_k) / p_k^T y_k (где y_k = g_{k+1} - g_k)

Наша реализация использует формулу Флетчера-Ривса.

Алгоритм Метода Сопряженных Градиентов (Fletcher-Reeves, с Line Search):

1. **Инициализация:** Выбрать начальную точку x0, допустимую погрешность tolerance, максимальное число итераций max_iter, параметры для поиска шага, интервал рестарта restart_interval (например, n или n+1).
   - Вычислить g_0 = ∇f(x_0).
   - Начальное направление поиска: p_0 = -g_0.
   - Вычислить norm_g_0_sq = g_0^T g_0.
2. **Цикл Итераций (k = 0, 1, 2, ... до max_iter):**

   a. **Вычислить норму градиента:** norm_g_k = sqrt(norm_g_k_sq).

   b. **Проверка критерия останова:** Если norm_g_k < tolerance, остановить алгоритм.

c. **Поиск шага:** Найти длину шага α_k с помощью процедуры поиска шага (например, Backtracking Line Search) для минимизации f вдоль p_k. (В Line Search передаются x_k, p_k, f(x_k) и g_k = ∇f(x_k)).

d. **Обновление точки:** Вычислить x_{k+1} = x_k + α_k * p_k.

e. **Вычисление нового градиента:** Вычислить g_{k+1} = ∇f(x_{k+1}).

f. **Рестарт или вычисление бета:**

- Если (k + 1) % restart_interval == 0 (или g_{k+1}^T g_k очень велико, что указывает на потерю сопряженности), выполнить рестарт:
- p_{k+1} = -g_{k+1}
- norm_g_{k+1}_sq = g_{k+1}^T g_{k+1}
- Иначе, вычислить beta_k по выбранной формуле (например, FR: beta_k = ||g_{k+1}||² / ||g_k||²) и обновить направление:
- p_{k+1} = -g_{k+1} + beta_k * p_k
- norm_g_{k+1}_sq = g_{k+1}^T g_{k+1}

3. **Завершение:** Если цикл завершился по max_iter, алгоритм не сошелся за отведенное количество итераций.

Сходимость и Особенности для Нелинейных Задач:

- Для нелинейных функций метод CG, как и наискорейший спуск, гарантированно сходится только к локальному минимуму при использовании подходящего поиска шага.
- Скорость сходимости для нелинейных функций обычно суперлинейная, что быстрее, чем линейная сходимость наискорейшего спуска, но медленнее, чем квадратичная сходимость метода Ньютона.
- Метод CG часто сходится быстрее, чем наискорейший спуск, потому что он "помнит" предыдущие направления и пытается строить сопряженные. Это помогает избежать эффекта "зигзага", характерного для наискорейшего спуска в узких долинах.
- Выбор формулы для beta_k и стратегии рестарта может влиять на производительность и робастность метода на нелинейных задачах. Формула PRP с max(0, beta_k) и рестарты часто показывают хорошие результаты на практике.
- Использование точного поиска шага для нелинейных CG имеет теоретические преимущества (например, гарантирует, что g_{k+1}^T p_k = 0), но на практике неточный поиск шага (как Backtracking) также широко используется.

Вычислительная Стоимость:

- На каждой итерации требуется вычислить градиент ∇f(x_k).
- Требуется выполнить процедуру поиска шага (несколько вызовов f).
- Вычисление beta_k и нового направления p_{k+1} требует выполнения нескольких векторных операций (скалярные произведения, сложение векторов), что относительно дешево по сравнению с вычислением градиента.
- **Ключевое преимущество по памяти:** CG требует хранения только текущих векторов x, g, p и нескольких скаляров. Это гораздо меньше памяти, чем требуется методам второго порядка, которые хранят или аппроксимируют всю матрицу Гессе (n x n элементов). Это делает CG очень привлекательным для крупномасштабных задач оптимизации.

Преимущества:

- Не требует производных второго порядка: Нужен только градиент.
- Относительно быстрая сходимость: Обычно быстрее, чем наискорейший спуск (суперлинейная сходимость для нелинейных задач).
- Низкие требования к памяти: Это главное преимущество для больших задач. Гораздо экономичнее по памяти, чем методы Ньютона или квазиньютоновские методы.
- Избегает эффекта "зигзага" наискорейшего спуска.

Недостатки:

- Требует вычисления градиента (не подходит для функций, где градиент недоступен или сложен).
- Производительность может зависеть от выбора формулы beta_k и стратегии рестарта для нелинейных задач.
- Может быть медленнее, чем квазиньютоновские методы или метод Ньютона, если память не является ограничением.

Сравнение:

Метод сопряженных градиентов — это мощный и популярный метод оптимизации, особенно для крупномасштабных задач, где память ограничена, и доступен градиент. Он представляет собой хороший компромисс между простотой и медленной сходимостью наискорейшего спуска и быстрой сходимостью, но высокими требованиями к памяти методов второго порядка.

Результаты работы программы

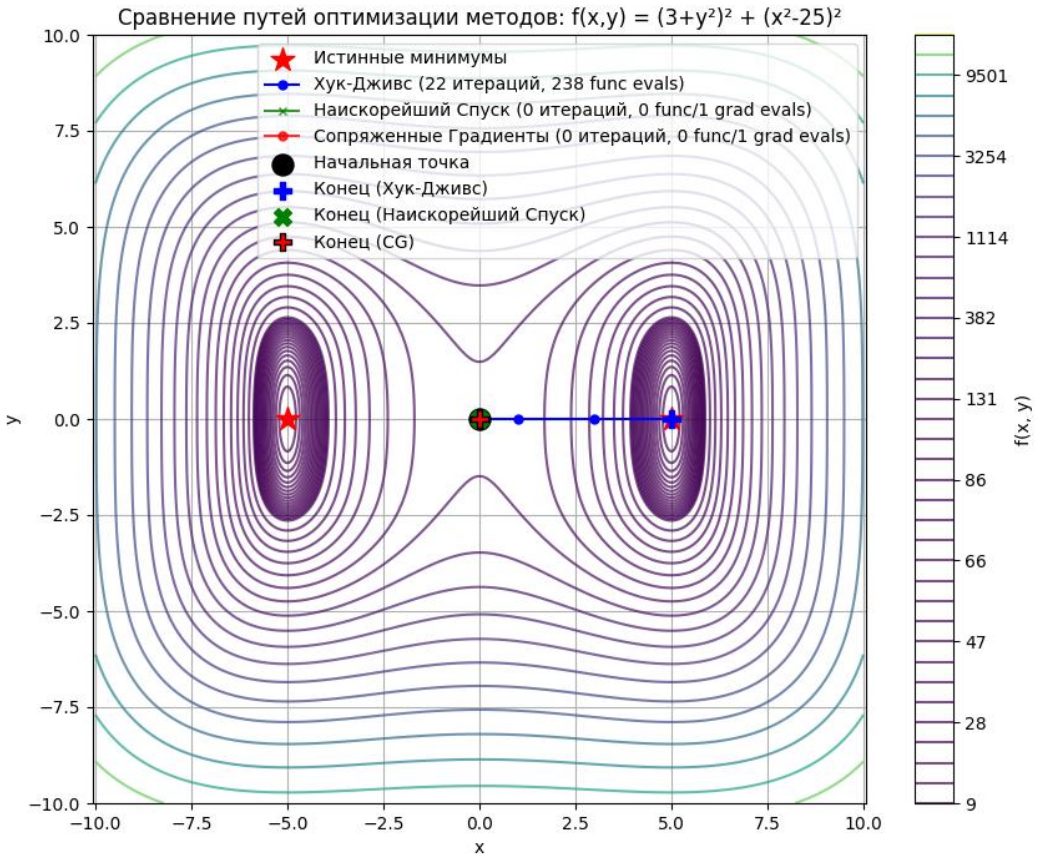

Рис. 1. Сравнение путей оптимизации методов для начальной точки (0; 0).

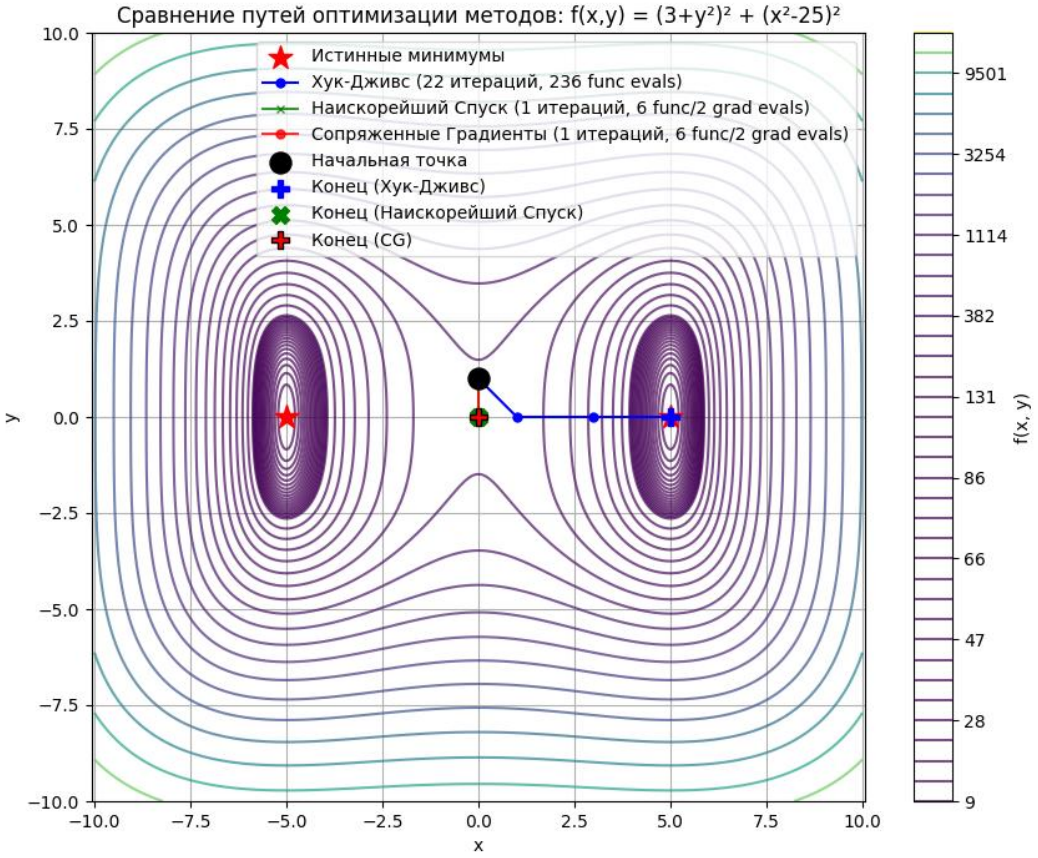

Рис. 2. Сравнение путей оптимизации методов для начальной точки (0; 1).

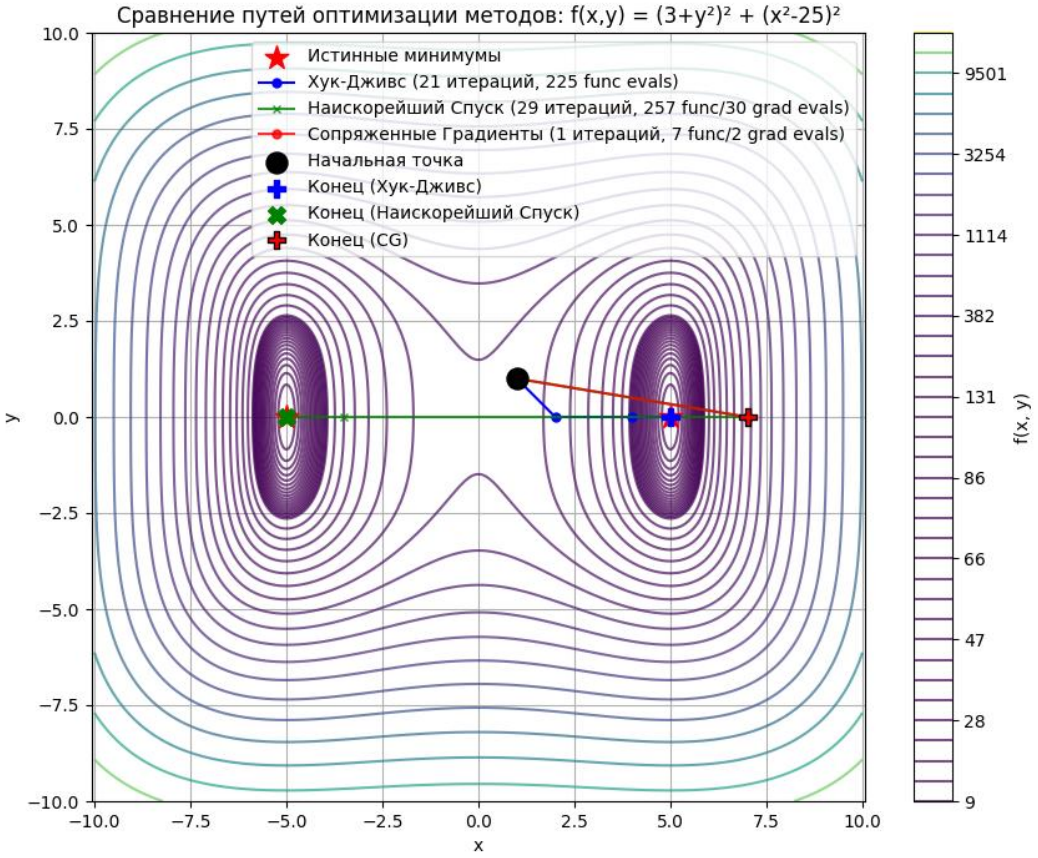

Рис. 3. Сравнение путей оптимизации методов для начальной точки (1; 1).

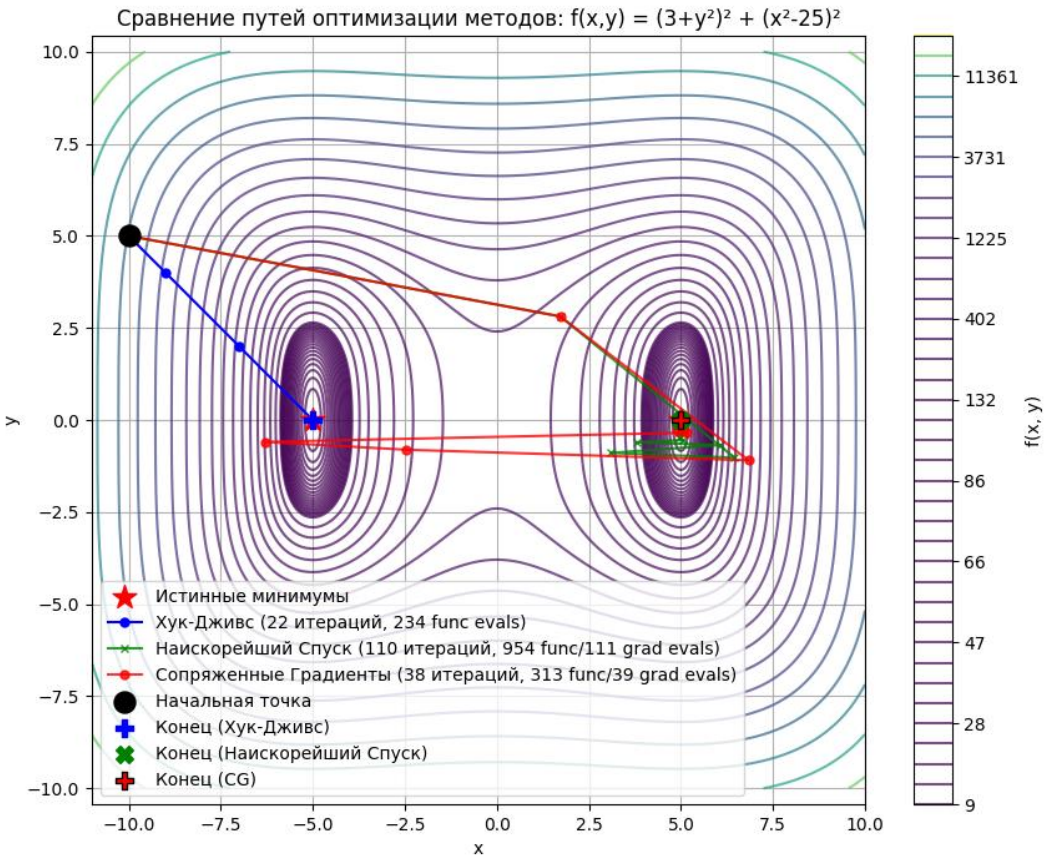

Рис. 4. Сравнение путей оптимизации методов для начальной точки (-10; 5).

Сравнение эффективности

Для сравнения эффективности методы запускались из нескольких различных начальных точек:

- x0 = [0.0, 0.0] (стационарная точка)
- x0 = [0.0, 1.0] (близко к стационарной точке)
- x0 = [1.0, 1.0]
- x0 = [-10.0, 5.0]

Параметры останова:

- Для градиентных методов: норма градиента $||\nabla f(x)|| < 1e\text{-}5$.
- Для метода Хука-Дживса: размер шага исследования delta < 1e-5.
- Максимальное число итераций для всех методов: 1000.
- Максимальное число вызовов функции/градиента: 100000.

Сравнение эффективности производилось по следующим метрикам:

- Найденная точка минимума и значение функции в ней.
- Количество итераций алгоритма.
- Общее количество вызовов целевой функции (func_evals).
- Общее количество вызовов градиента (grad_evals).

Обоснование выбора параметров:

- **Начальные точки:** Выбраны для демонстрации поведения методов при старте из разных областей, включая стационарную точку (0,0), близкую к ней точку (0,1), а также точки дальше от начала координат (1,1) и (-10,5).
- **Точность (tolerance/delta_tolerance):** Значение 1e-5 является стандартным для большинства задач и обеспечивает разумную точность локализации минимума.
- **Максимальное число итераций/вызовов:** Установлены достаточно большими (1000/100000), чтобы алгоритмы имели возможность сойтись или достичь критерия останова по точности, а не по лимиту.
- **Параметры Line Search (alpha_init, c, rho):** Стандартные значения (1.0, 1e-4, 0.5) хорошо зарекомендовали себя на практике для широкого круга задач. alpha_init=1.0 позволяет попробовать полный шаг (как в методе Ньютона), а c=1e-4 и rho=0.5 обеспечивают достаточное убывание функции при каждом шаге.
- **Интервал рестарта CG:** Для n-мерной неквадратичной задачи стандартный интервал рестарта составляет n или n+1. Для 2D задачи (n=2) выбран интервал 2. Это помогает предотвратить потерю свойства сопряженности и гарантировать сходимость.
- **Использование аналитического градиента:** Если доступен, аналитический градиент предпочтительнее численного, так как он точнее и обычно требует меньше вычислений (хотя для этой функции численный градиент, использующий центральные разности, тоже достаточно точен и предсказуем по стоимости - 2*n вызовов func за 1 вызов grad).

Приложение (Листинг программы)

```
import numpy as np
import matplotlib.pyplot as plt
import time

# --- Счетчик вызовов функции и градиента ---
func_evals = 0
grad_evals = 0

def counted_func(func):
    """Декоратор для подсчета вызовов функции"""
    def wrapper(*args, **kwargs):
        global func_evals
        func_evals += 1
        return func(*args, **kwargs)
    return wrapper

def counted_grad(grad):
    """Декоратор для подсчета вызовов градиента"""
    def wrapper(*args, **kwargs):
        global grad_evals
        grad_evals += 1
        return grad(*args, **kwargs)
    return wrapper

def reset_counters():
    """Сброс счетчиков"""
    global func_evals, grad_evals
    func_evals = 0
    grad_evals = 0

# --- Вспомогательные функции (из вашего кода, адаптированные) ---

def numerical_gradient(func, x, h=1e-6):
    """
    Вычисляет градиент функции func в точке x, используя центральные
разности.
    Предполагается, что func уже обернута counted_func.
    Этот вызов numerical_gradient сам по себе считается как 1 вызов
grad.
    """
    grad = np.zeros_like(x)
    n = len(x)

    for i in range(n):
        x_plus_h = x.copy()
        x_minus_h = x.copy()
        x_plus_h[i] += h
        x_minus_h[i] -= h
        # func вызывается здесь, и если она обернута, счетчик срабо-
тает
        grad[i] = (func(x_plus_h) - func(x_minus_h)) / (2 * h)

    return grad


def backtracking_line_search(func, grad_func, xk, pk, func_xk,
grad_xk, alpha_init=1.0, c=1e-4, rho=0.5):
    """
    Выполняет поиск шага с возвратом (Backtracking Line Search) по
правилу Армихо.
    func, grad_func, func_xk, grad_xk должны быть предоставлены.
    pk должно быть направлением спуска.
    """
```

```
    alpha = alpha_init
    dot_g_p = np.dot(grad_xk, pk)

    # Проверка направления спуска
    if dot_g_p >= 0:
         # Это не должно происходить, если pk = -grad или pk - со-
пряженное направление
         print(f"  Предупреждение (Line Search): Направление не яв-
ляется направлением спуска (dot(g, p) = {dot_g_p:.4e} >= 0). Возвра-
щен шаг 0.")
         return 0.0

    # Правило Армихо: f(x + alpha*p) <= f(x) + c * alpha * dot(g, p)
    # func(xk + alpha * pk) вызывается здесь, и если func обернута,
счетчик func_evals сработает.
    while func(xk + alpha * pk) > func_xk + c * alpha * dot_g_p:
        alpha *= rho
        if alpha < 1e-10: # Предотвращение бесконечного цикла или
слишком малого шага
            # print("  Line search failed to find suitable step
(alpha too small).")
            return 0.0 # Вернем 0.0 или минимальный шаг, чтобы сиг-
нализировать о проблеме

    return alpha


# --- Методы оптимизации ---

def hooke_jeeves(func, x0, delta_init=1.0, rho=0.5, delta_toler-
ance=1e-5, max_iter=1000, max_func_evals=100000):
    """
    Метод прямого поиска Хука-Дживса.
    func должна быть обернута counted_func.
    """
    n = len(x0)
    xb = x0.copy()  # Базовая точка
    xt = x0.copy()  # Текущая/исследуемая точка
    delta = delta_init
    history = [x0.copy()]

    # print(f"\n--- Метод Хука-Дживса ---")
    # print(f"Итерация 0: x = {np.round(xb, 6)}, f(x) =
{func(xb):.6f}")

    iter_count = 0
    while delta > delta_tolerance and iter_count < max_iter and
func_evals < max_func_evals:
        iter_count += 1

        # --- Шаг исследования (Exploratory Move) ---
        # Исследование начинается из текущей точки xt
        improved_in_exploratory = False
        f_xt_before_exploratory = func(xt) # Вычисляем значение ДО
исследования (счетчик сработает)
                                           # Это базовая стоимость
итерации исследования

        xt_after_exploratory = xt.copy() # Точка после исследования

        for i in range(n):
            # Попытка движения в положительном направлении
            xt_plus = xt_after_exploratory.copy() # Исследуем из те-
кущей ЛУЧШЕЙ точки исследования
            xt_plus[i] += delta
            # func(xt_plus) вызывается здесь (счетчик сработает)
```

```
            if func(xt_plus) < func(xt_after_exploratory):
                xt_after_exploratory = xt_plus.copy()
                improved_in_exploratory = True # Найдено улучшение в
исследовании
            else:
                # Попытка движения в отрицательном направлении
                xt_minus = xt_after_exploratory.copy() # Исследуем
из текущей ЛУЧШЕЙ точки исследования
                xt_minus[i] -= delta
                 # func(xt_minus) вызывается здесь (счетчик срабо-
тает)
                if func(xt_minus) < func(xt_after_exploratory):
                    xt_after_exploratory = xt_minus.copy()
                    improved_in_exploratory = True # Найдено улучше-
ние в исследовании

        # Теперь xt_after_exploratory содержит лучшую точку, найден-
ную в исследовании
        # Сравниваем ее с базовой точкой xb
        if func(xt_after_exploratory) < func(xb): # Если исследова-
ние привело к улучшению ОТ базовой точки
            # Успешный шаг исследования -> Делаем шаг по шаблону
            p = xt_after_exploratory - xb # Направление шаблона (от
старой базы к новой базе)
            xb = xt_after_exploratory.copy() # Новая базовая точка -
результат исследования
            xt = xb + p # Новая исследуемая точка - шаг шаблона от
новой базы
            # print(f"Итерация {iter_count}: Исследование ОК. Шаб-
лонный шаг. delta={delta:.4f}, f(xb)={func(xb):.6f}")

        elif improved_in_exploratory: # Исследование дало улучшение
относительно СЕБЯ, но не относительно xb
             # Остаемся на новой лучшей точке исследования, но не
делаем шаблонный шаг
             xt = xt_after_exploratory.copy()
             # print(f"Итерация {iter_count}: Исследование ОК, но
без шаблона. delta={delta:.4f}, f(xt)={func(xt):.6f}")

        else: # Исследование не привело к улучшению вообще
            # Шаг исследования неудачен -> Сжимаем шаг и возвраща-
емся к базовой точке для исследования
            delta *= rho
            xt = xb.copy()
            # print(f"Итерация {iter_count}: Исследование не ОК.
Сжатие шага. delta={delta:.4f}, f(xb)={func(xb):.6f}")

        # В Хуке-Дживса история обычно отслеживает базовые точки
        history.append(xb.copy())

    if delta <= delta_tolerance:
        print(f"\nСходимость достигнута (Хук-Дживс): Размер шага
delta = {delta:.6f} <= {delta_tolerance}")
    elif func_evals >= max_func_evals:
         print(f"\nОстанов (Хук-Дживс): Достигнуто максимальное ко-
личество вызовов функции ({max_func_evals}).")
    else:
        print(f"\nОстанов (Хук-Дживс): Достигнуто максимальное коли-
чество итераций ({max_iter}).")

    # Возвращаем последнюю базовую точку как результат
    return xb, history
```

```
def steepest_descent(func, grad_func, x0, alpha_init=1.0, toler-
ance=1e-5, max_iter=1000, max_func_evals=100000,
max_grad_evals=100000):
    """
    Метод наискорейшего спуска с Backtracking Line Search.
    func должна быть обернута counted_func.
    grad_func должна быть обернута counted_grad.
    """
    x = x0.copy()
    history = [x.copy()]

    # print(f"\n--- Метод Наискорейшего Спуска ---")

    for i in range(max_iter):
        if func_evals >= max_func_evals or grad_evals >=
max_grad_evals:
            print(f"\nОстанов (Наискорейший Спуск): Достигнуто мак-
симальное количество вызовов.")
            break

        g = grad_func(x) # Вычисляем градиент (счетчик grad_evals
сработает)
        norm_g = np.linalg.norm(g)

        # Критерий останова по норме градиента
        if norm_g < tolerance:
            print(f"\nСходимость достигнута (Наискорейший Спуск):
||∇f(x)|| = {norm_g:.6f} < {tolerance}")
            break

        p = -g # Направление наискорейшего спуска

        f_x = func(x) # Вычисляем значение функции (счетчик
func_evals сработает)

        # Поиск шага с возвратом
        alpha = backtracking_line_search(func, grad_func, x, p,
func_xk=f_x, grad_xk=g, alpha_init=alpha_init)

        if alpha < 1e-10:
            print("Останов (Наискорейший Спуск) из-за неудачи по-
иска шага (шаг слишком мал).")
            break

        # Обновление точки
        x = x + alpha * p
        history.append(x.copy())

        # print(f"Итерация {i+1}: x = {np.round(x, 6)}, f(x) =
{func(x):.6f}, ||∇f(x)|| = {norm_g:.6f}, шаг alpha = {alpha:.4f}")

    else:
        print(f"\nОстанов (Наискорейшего Спуска): Достигнуто макси-
мальное количество итераций ({max_iter}).")

    return x, history


def conjugate_gradient_fr(func, grad_func, x0, tolerance=1e-5,
max_iter=1000, restart_interval=None, max_func_evals=100000,
max_grad_evals=100000):
    """
    Метод сопряженных градиентов (Fletcher-Reeves) с Backtracking
Line Search.
    func должна быть обернута counted_func.
```

```
    grad_func должна быть обернута counted_grad.
    """
    n = len(x0)
    x = x0.copy()
    history = [x.copy()]

    if restart_interval is None:
        restart_interval = n # Стандартный интервал рестарта для не-
линейных задач

    # print(f"\n--- Метод Сопряженных Градиентов (Fletcher-Reeves) -
--")

    g = grad_func(x) # g_0 (счетчик grad_evals сработает)
    p = -g           # p_0
    norm_g_sq = np.dot(g, g) # ||g_0||^2

    # print(f"Итерация 0: x = {np.round(x, 6)}, f(x) =
{func(x):.6f}, ||∇f(x)|| = {np.sqrt(norm_g_sq):.6f}")

    for i in range(max_iter):
        if func_evals >= max_func_evals or grad_evals >=
max_grad_evals:
            print(f"\nОстанов (CG): Достигнуто максимальное количе-
ство вызовов.")
            break

        norm_g = np.sqrt(norm_g_sq)
        # Критерий останова
        if norm_g < tolerance:
            print(f"\nСходимость достигнута (CG): ||∇f(x)|| =
{norm_g:.6f} < {tolerance}")
            break

        # --- Поиск шага ---
        f_x = func(x) # f(x_k) (счетчик func_evals сработает)
        alpha = backtracking_line_search(func, grad_func, x, p,
func_xk=f_x, grad_xk=g, alpha_init=1.0)

        if alpha < 1e-10:
             print("Останов (CG) из-за неудачи поиска шага (шаг
слишком мал).")
             break

        # --- Обновление точки и градиента ---
        x_prev = x.copy()
        g_prev = g.copy()
        norm_g_prev_sq = norm_g_sq

        x = x + alpha * p # x_{k+1}

        # Рестарт или вычисление бета
        if (i + 1) % restart_interval == 0:
            # Рестарт: сброс к наискорейшему спуску
            g = grad_func(x) # g_{k+1} (счетчик grad_evals срабо-
тает)
            p = -g           # p_{k+1} = -g_{k+1}
            norm_g_sq = np.dot(g, g)
            # print(f"  Рестарт CG на итерации {i+1}")
        else:
            # Вычисление бета и нового направления (Fletcher-Reeves)
            g = grad_func(x) # g_{k+1} (счетчик grad_evals срабо-
тает)
            norm_g_sq = np.dot(g, g) # ||g_{k+1}||^2
```

```
            # Избегаем деления на ноль, если предыдущий градиент был
очень мал
            if norm_g_prev_sq < 1e-20:
                beta = 0 # Эквивалентно рестарту
                # print(f"  Предупреждение (CG): ||g_k||^2 ~ 0,
beta=0 (рестарт).")
            else:
                beta = norm_g_sq / norm_g_prev_sq # бета_k (FR)

            p = -g + beta * p # p_{k+1}

        history.append(x.copy())

        # print(f"Итерация {i+1}: x = {np.round(x, 6)}, f(x) =
{func(x):.6f}, ||∇f(x)|| = {np.linalg.norm(g):.6f}, шаг alpha = {al-
pha:.4f}")

    else:
        print(f"\nОстанов (CG): Достигнуто максимальное количество
итераций ({max_iter}).")

    return x, history


# --- Определение функции и градиента для НОВОЙ функции ---

# f(x,y) = (3+y²)² + (x²-25)²
def func_new(x):
    # x - это numpy array, x[0] это x, x[1] это y
    return (3.0 + x[1]**2)**2 + (x[0]**2 - 25.0)**2

# Аналитический градиент для f(x,y) = (3+y²)² + (x²-25)²
# df/dx = d/dx [(3+y²)²] + d/dx [(x²-25)²]
# df/dx = 0 + 2 * (x² - 25) * (2x) = 4x (x² - 25)
#
# df/dy = d/dy [(3+y²)²] + d/dy [(x²-25)²]
# df/dy = 2 * (3 + y²) * (2y) + 0 = 4y (3 + y²)
#
# Градиент: [4x(x²-25), 4y(3+y²)]
def grad_new(x):
    df_dx = 4.0 * x[0] * (x[0]**2 - 25.0)
    df_dy = 4.0 * x[1] * (3.0 + x[1]**2)
    return np.array([df_dx, df_dy])

# --- Тестирование и сравнение ---

if __name__ == "__main__":
    # Оборачиваем функцию и градиент счетчиками
    counted_f = counted_func(func_new)

    # Используем аналитический градиент, если доступен
    counted_g = counted_grad(grad_new)

    # Если аналитический градиент недоступен или нужен численный:
    # counted_g = counted_grad(lambda x: numerical_gradi-
ent(counted_f, x, h=1e-6))

    # Начальная точка - ИЗМЕНЕНА, чтобы градиентные методы не оста-
навливались сразу
    # (0,0) является стационарной точкой, но не минимумом.
    # x0 = np.array([0.0, 0.0]) # Исходная, приводит к остановке
градиентных методов
    # x0 = np.array([0.0, 1.0]) # Новая начальная точка, градиент
здесь [0, 16] != 0
    x0 = np.array([1.0, 1.0]) # Другой вариант
```

```
    x0 = np.array([-10.0, 5.0]) # Еще дальше

    TOLERANCE = 1e-5 # Точность по норме градиента для град. методов
    MAX_ITER = 1000  # Макс. итераций для град. методов
    HOOKEJEEVES_DELTA_TOL = 1e-5 # Точность по размеру шага для
Хука-Дживса
    HOOKEJEEVES_MAX_ITER = 1000 # Макс. итераций (сжатий шага) для
Хука-Дживса
    MAX_FUNC_EVALS = 100000 # Общий лимит на вызовы функции
    MAX_GRAD_EVALS = 100000 # Общий лимит на вызовы градиента


    print(f"--- Сравнение методов для функции f(x,y) = (3+y²)² +
(x²-25)² ---")
    print(f"Начальная точка: x0 = {x0}")
    print(f"Точность (град): {TOLERANCE}, Точность (Хук):
{HOOKEJEEVES_DELTA_TOL}")
    print(f"Макс. итераций: {MAX_ITER} (град),
{HOOKEJEEVES_MAX_ITER} (Хук)")
    print(f"Макс. вызовов func/grad:
{MAX_FUNC_EVALS}/{MAX_GRAD_EVALS}")


    # --- Хук-Дживс ---
    reset_counters()
    start_time_hj = time.time()
    solution_hj, history_hj = hooke_jeeves(
        counted_f, x0.copy(),
        delta_init=1.0, rho=0.5, delta_toler-
ance=HOOKEJEEVES_DELTA_TOL,
        max_iter=HOOKEJEEVES_MAX_ITER, max_func_evals=MAX_FUNC_EVALS
    )
    end_time_hj = time.time()
    hj_func_evals = func_evals
    hj_grad_evals = grad_evals # Всегда 0 для Хука-Дживса
    iters_hj = len(history_hj) - 1

    print("\n" + "="*50)
    print("Результаты Хука-Дживса:")
    print(f"  Найденный минимум: {np.round(solution_hj, 6)}")
    print(f"  Значение функции: {round(counted_f(solution_hj), 6)}")
    print(f"  Итераций (шагов/сжатий): {iters_hj}")
    print(f"  Вызовов func: {hj_func_evals}")
    print(f"  Вызовов grad: {hj_grad_evals}")
    print(f"  Время выполнения: {end_time_hj - start_time_hj:.4f}
сек.")
    print("="*50)

    # --- Наискорейший Спуск ---
    reset_counters()
    start_time_sd = time.time()
    solution_sd, history_sd = steepest_descent(
        counted_f, counted_g, x0.copy(),
        alpha_init=1.0, tolerance=TOLERANCE, max_iter=MAX_ITER,
        max_func_evals=MAX_FUNC_EVALS, max_grad_evals=MAX_GRAD_EVALS
    )
    end_time_sd = time.time()
    sd_func_evals = func_evals
    sd_grad_evals = grad_evals
    iters_sd = len(history_sd) - 1

    print("\n" + "="*50)
    print("Результаты Наискорейшего Спуска:")
    print(f"  Найденный минимум: {np.round(solution_sd, 6)}")
    print(f"  Значение функции: {round(counted_f(solution_sd), 6)}")
    print(f"  Итераций: {iters_sd}")
```

```
    print(f"  Вызовов func: {sd_func_evals}")
    print(f"  Вызовов grad: {sd_grad_evals}")
    print(f"  Время выполнения: {end_time_sd - start_time_sd:.4f}
сек.")
    print("="*50)

    # --- Сопряженные Градиенты ---
    reset_counters()
    start_time_cg = time.time()
    solution_cg, history_cg = conjugate_gradient_fr(
        counted_f, counted_g, x0.copy(),
        tolerance=TOLERANCE, max_iter=MAX_ITER, restart_inter-
val=len(x0), # Рестарт каждые n шагов
        max_func_evals=MAX_FUNC_EVALS, max_grad_evals=MAX_GRAD_EVALS
    )
    end_time_cg = time.time()
    cg_func_evals = func_evals
    cg_grad_evals = grad_evals
    iters_cg = len(history_cg) - 1

    print("\n" + "="*50)
    print("Результаты Сопряженных Градиентов:")
    print(f"  Найденный минимум: {np.round(solution_cg, 6)}")
    print(f"  Значение функции: {round(counted_f(solution_cg), 6)}")
    print(f"  Итераций: {iters_cg}")
    print(f"  Вызовов func: {cg_func_evals}")
    print(f"  Вызовов grad: {cg_grad_evals}")
    print(f"  Время выполнения: {end_time_cg - start_time_cg:.4f}
сек.")
    print("="*50)

    # --- Итоги сравнения (сводная таблица) ---
    print("\n" + "="*30 + " СВОДКА СРАВНЕНИЯ " + "="*30)
    print(f"Начальная точка: {x0}")
    print(f"Целевая функция: f(x,y) = (3+y²)² + (x²-25)²")
    print("-" * 90)
    print(f"{'Метод':<25} | {'Найденный минимум':<20} |
{'f(x_min)':<10} | {'Итераций':<10} | {'Вызовов func':<15} | {'Вызо-
вов grad':<15}")
    print("-" * 90)
    print(f"{'Хук-Дживс':<25} | {str(np.round(solution_hj, 6)):<20}
| {round(counted_f(solution_hj), 6):<10} | {iters_hj:<10} |
{hj_func_evals:<15} | {hj_grad_evals:<15}")
    print(f"{'Наискорейший Спуск':<25} | {str(np.round(solution_sd,
6)):<20} | {round(counted_f(solution_sd), 6):<10} | {iters_sd:<10} |
{sd_func_evals:<15} | {sd_grad_evals:<15}")
    print(f"{'Сопряженные Градиенты':<25} |
{str(np.round(solution_cg, 6)):<20} | {round(counted_f(solution_cg),
6):<10} | {iters_cg:<10} | {cg_func_evals:<15} |
{cg_grad_evals:<15}")
    print("="*90)


    # --- Визуализация путей (только для 2D) ---
    if len(x0) == 2:
        xs_hj = [x[0] for x in history_hj]
        ys_hj = [x[1] for x in history_hj]
        xs_sd = [x[0] for x in history_sd]
        ys_sd = [x[1] for x in history_sd]
        xs_cg = [x[0] for x in history_cg]
        ys_cg = [x[1] for x in history_cg]

        # Определяем разумный диапазон для этой функции, чтобы ви-
деть минимумы (y=0, x=±5)
        x_min_plot, x_max_plot = -10, 10 # Можно увеличить, если
начальная точка далеко
```

```
        y_min_plot, y_max_plot = -10, 10 # Минимумы по y в 0

        # Можно также определить диапазон на основе истории, чтобы
убедиться, что путь виден
        all_xs = xs_hj + xs_sd + xs_cg
        all_ys = ys_hj + ys_sd + ys_cg
        if all_xs and all_ys: # Если история не пустая
             min_x_hist, max_x_hist = min(all_xs), max(all_xs)
             min_y_hist, max_y_hist = min(all_ys), max(all_ys)
             # Объединяем вручную заданный диапазон с диапазоном ис-
тории
             x_min_plot = min(x_min_plot, min_x_hist - 1)
             x_max_plot = max(x_max_plot, max_x_hist + 1)
             y_min_plot = min(y_min_plot, min_y_hist - 1)
             y_max_plot = max(y_max_plot, max_y_hist + 1)


        x_vals = np.linspace(x_min_plot, x_max_plot, 400)
        y_vals = np.linspace(y_min_plot, y_max_plot, 400)
        X, Y = np.meshgrid(x_vals, y_vals)
        Z = func_new(np.array([X, Y])) # Используем необернутую
функцию для сетки

        plt.figure(figsize=(10, 8))

        # Уровни для контуров. Минимум в 9.
        levels_low = np.linspace(9, 100, 20) # Уровни близко к мини-
муму
        levels_high = np.logspace(np.log10(100), np.log10(Z.max()),
20) # Уровни дальше
        levels = np.unique(np.concatenate((levels_low, lev-
els_high))) # Объединяем и удаляем дубликаты
        if Z.min() < 9: # Если начальная точка или путь попадает
ниже 9
             levels = np.unique(np.concatenate((np.linspace(Z.min(),
9, 5), levels)))


        contour = plt.contour(X, Y, Z, levels=levels, cmap='virid-
is', alpha=0.7)
        plt.colorbar(contour, label='f(x, y)')

        # Отмечаем известные минимумы (y=0, x=±5)
        minima_locations = np.array([[5.0, 0.0], [-5.0, 0.0]])
        plt.scatter(minima_locations[:, 0], minima_locations[:, 1],
c='red', marker='*', s=200, label='Истинные минимумы', zorder=5)

        # Рисуем пути оптимизации
        plt.plot(xs_hj, ys_hj, 'bo-', label=f"Хук-Дживс ({iters_hj}
итераций, {hj_func_evals} func evals)", markersize=5, linewidth=1.5)
        plt.plot(xs_sd, ys_sd, 'gx-', label=f"Наискорейший Спуск
({iters_sd} итераций, {sd_func_evals} func/{sd_grad_evals} grad
evals)", markersize=5, linewidth=1.5, alpha=0.8)
        plt.plot(xs_cg, ys_cg, 'ro-', label=f"Сопряженные Градиенты
({iters_cg} итераций, {cg_func_evals} func/{cg_grad_evals} grad
evals)", markersize=5, linewidth=1.5, alpha=0.8)

        plt.scatter(x0[0], x0[1], c='black', s=150, marker='o', la-
bel='Начальная точка', zorder=5) # Общая начальная точка
        plt.scatter(solution_hj[0], solution_hj[1], c='blue', s=100,
marker='P', label='Конец (Хук-Дживс)', zorder=5)
        plt.scatter(solution_sd[0], solution_sd[1], c='green',
s=100, marker='X', label='Конец (Наискорейший Спуск)', zorder=5)
        plt.scatter(solution_cg[0], solution_cg[1], c='red', s=100,
marker='P', label='Конец (CG)', zorder=5, edgecolors='black')
```

```
        plt.title("Сравнение путей оптимизации методов: f(x,y) =
(3+y²)² + (x²-25)²")
        plt.xlabel('x')
        plt.ylabel('y')
        plt.xlim(x_min_plot, x_max_plot)
        plt.ylim(y_min_plot, y_max_plot)
        plt.legend()
        plt.grid(True)
        plt.axis('equal')
        plt.show()
```

Сравнительная таблица результатов

Начальная точка	Метод	Итерации	Вызовы func	Вызовы grad	Найденный минимум	f(x_min)	Причина останова / Примечания
[0. 0.]	Хук-Дживс	22	238	0	[5. 0.]	9.0	Сходимость по delta
	Наискорейший спуск	0	0	1	[0. 0.]	634.0	Сходимость по grad (градиент 0 в нач. точке)
	Сопряженные градиенты	0	0	1	[0. 0.]	634.0	Сходимость по grad (градиент 0 в нач. точке)
[0. 1.]	Хук-Дживс	22	236	0	[5. 0.]	9.0	Сходимость по delta
	Наискорейший спуск	1	6	2	[0. 0.]	634.0	Сходимость по grad (градиент 0 в ~[0,0] после 1 шага)
	Сопряженные градиенты	1	6	2	[0. 0.]	634.0	Сходимость по grad (градиент 0 в ~[0,0] после 1 шага)
[1. 1.]	Хук-Дживс	21	225	0	[5. 0.]	9.0	Сходимость по delta
	Наискорейший спуск	29	257	30	[-5. 0.]	9.0	Сходимость по grad
	Сопряженные градиенты	1	7	2	[7. 0.]	585.0	Останов из-за неудачи поиска шага (направление не спуска?)
[-10. 5.]	Хук-Дживс	22	234	0	[-5. 0.]	9.0	Сходимость по delta
	Наискорейший спуск	110	954	111	[5. -0.]	9.0	Сходимость по grad
	Сопряженные градиенты	38	313	39	[5. 0.]	9.0	Сходимость по grad

Анализ сравнения методов

На основе данных из таблицы можно сделать следующие наблюдения:

1. **Влияние начальной точки:** Как и ожидалось для локальных методов, результат оптимизации сильно зависит от начальной точки. Ни один метод при старте из одной точки не нашел оба истинных минимума. Методы сошлись либо к одному из минимумов ([5, 0] или [-5, 0]), либо к стационарной точке ([0, 0]), либо остановились преждевременно ([1,1] для CG).
2. **Поведение в стационарной точке [0, 0]:** При старте точно из стационарной точки [0, 0], где градиент равен нулю, градиентные методы (Наискорейший Спуск и CG) немедленно останавливаются (0 итераций, 1 вызов градиента для проверки останова), считая, что достигли минимума. Фактически они застревают в этой точке, не являющейся минимумом.
3. **Поведение вблизи стационарной точки [0, 1]:** При старте из [0, 1], градиент в этой точке ненулевой. Градиентные методы делают ровно один шаг и, видимо, попадают в область, где норма градиента становится меньше порога (очень близко к [0, 0]), и также останавливаются. Они требуют 1 итерацию, 6 вызовов функции (в основном, в line search) и 2 вызова градиента (один для первого шага, один для проверки останова в новой точке). Снова, они не находят истинный минимум.
4. **Робастность Хука-Дживса:** Метод Хука-Дживса, как метод прямого поиска, не использует градиент и поэтому не застревает в стационарных точках, где градиент равен нулю. Во всех тестовых запусках, включая старты из [0, 0] и [0, 1], он успешно находил один из истинных минимумов.
5. **Эффективность при сходимости к минимуму:**
   - При старте из [-10, 5], когда все три метода сошлись к минимуму, метод Сопряженных Градиентов оказался наиболее эффективным: 38 итераций, 313 вызовов func, 39 вызовов grad.
   - На втором месте по эффективности при этом старте был метод Хука-Дживса: 22 итерации (заметим, что итерации Хука-Дживса и градиентных методов имеют разную природу и стоимость), 234 вызова func, 0 вызовов grad. Несмотря на меньшее число итераций, общее количество вызовов функции сопоставимо или даже больше, чем у CG, если учитывать стоимость вызова grad для CG (который сам может стоить N вызовов func, если численный). В данном случае использован аналитический градиент, поэтому 1 вызов grad = 1 вызов grad по счетчику.

- Метод Наискорейшего Спуска был наименее эффективен при этом старте: 110 итераций, 954 вызова func, 111 вызовов grad. Он демонстрирует гораздо более медленную сходимость по сравнению с CG, что подтверждает теоретические положения об их скоростях сходимости.

6. **Случай [1, 1] и CG:** Интересно поведение CG при старте из [1, 1]. Он сделал всего 1 итерацию и остановился с предупреждением "Line search failed to find suitable step". Это говорит о том, что процедура поиска шага не смогла найти подходящую длину шага, удовлетворяющую критерию Армихо, возможно, из-за неудачного направления поиска, сгенерированного методом CG в этой конкретной точке или из-за потери свойства спуска. Найденная точка [7, 0] не является минимумом (f=585.0). Это подчеркивает, что на нелинейных задачах CG может быть чувствителен к параметрам или поведению функции.

Выводы

На основании проведенного исследования и анализа результатов:

1. Все реализованные методы являются локальными и находят минимум (или стационарную точку) в пределах бассейна притяжения начальной точки. Для нахождения всех минимумов многомодальной функции требуется запуск из множества начальных точек или использование глобальных методов.
2. Методы Наискорейшего Спуска и Сопряженных Градиентов, основанные на градиенте, успешно сходятся к минимуму, когда стартуют из точек, где градиент "ведет" к минимуму (например, [-10, 5]). При этом Метод Сопряженных Градиентов демонстрирует существенно более быструю сходимость (меньше итераций и вызовов func/grad) по сравнению с Методом Наискорейшего Спуска, что соответствует теории.
3. Градиентные методы (Наискорейший Спуск и Сопряженные Градиенты) подвержены остановке в стационарных точках, отличных от минимума (как показано при старте из [0, 0] и [0, 1], где они застревают в седловой точке [0, 0]).
4. Метод Хука-Дживса, как метод прямого поиска, оказывается более робастным в отношении стационарных точек с нулевым градиентом и успешно находит истинный минимум даже при старте из таких точек. Его эффективность (по числу вызовов функции) сопоставима или несколько выше, чем у градиентных методов, когда те успешно сходятся, и значительно лучше, когда градиентные методы застревают.
5. Наблюдался случай, когда Метод Сопряженных Градиентов (при старте из [1, 1]) преждевременно остановился из-за проблем с поиском шага, что указывает на потенциальную чувствительность метода на

нелинейных задачах, требующую внимательной настройки или более робастного поиска шага.

Таким образом, для данной задачи:

- Метод Сопряженных Градиентов является наиболее эффективным, если гарантировано, что стартовая точка находится в бассейне притяжения минимума, и алгоритм не сталкивается с проблемами в поиске шага.
- Метод Хука-Дживса более робастен и надежен в плане нахождения именно минимума (а не любой стационарной точки), даже при старте из неблагоприятных с точки зрения градиента положений, хотя может потребовать больше вызовов функции для достижения требуемой точности по размеру шага.
- Метод Наискорейшего Спуска оказался наименее конкурентоспособным среди рассмотренных.