Министерство науки и высшего образования РФ

Федеральное государственное автономное

образовательное учреждение высшего образования

«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

|  |
| --- |
| Институт космических и информационных технологий |
| институт |
| Программная инженерия |
| кафедра |

**ОТЧЁТ О ПРАКТИЧЕСКОЙ РАБОТЕ №1**

|  |
| --- |
| Численные методы нулевого порядка для поиска безусловного экстремума |
| тема |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Преподаватель |  |  |  | B. B. Тынченко |
|  |  | подпись, дата |  | инициалы, фамилия |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Студент | КИ23-16/1б, 032322546 |  |  |  | Е. А. Гуртякин |
|  | номер группы, зачетной книжки |  | подпись, дата |  | инициалы, фамилия |

Красноярск 2025

1. **Цель**

* Программно реализовать численные методы безусловной оптимизации, указанные в варианте задания. Язык программирования можно использовать любой.

1. **Задачи**

Поставленные задачи:

* Программно реализовать численные методы безусловной оптимизации, указанные в варианте задания. Язык программирования можно использовать любой.
* Протестировать работу методов на функциях из примеров, решение которых пошагово рассмотрено в учебнике Пантелеева и Летовой «Методы оптимизации в примерах и задачах», раздел 2 «Численные методы».
* Провести сравнительный анализ влияния параметров методов на точность и скорость нахождения экстремума, варьируя параметры в заданных пределах с выбранным шагом. Для выполнения вычислительных экспериментов использовать целевые функции, указанные в варианте задания.
* Для реализованных методов необходимо построить графики зависимости точности и скорости вычислений от каждого из исследуемых параметров, каждый график надо сопроводить кратким комментарием. Все результаты вычислительных экспериментов следует представить компактно в виде сводной таблицы.
* По результатам исследования требуется сделать обоснованный развернутый вывод об особенностях работы исследуемых алгоритмов и их эффективности на заданных целевых функциях.

1. **Ход выполнения**
   1. **Метод Розенброка**

Первый метод, который мне предстояло реализовать, был алгоритм равномерного поиска.

Суть метода равномерного поиска состоит в следующем. Задается интервал поиска [a, b] и количество вычислений функции N. На этом интервале формируется множество из N+1 равноотстоящих пробных точек. В каждой из этих точек вычисляется значение целевой функции. Точка, в которой функция принимает наименьшее значение, объявляется приближенным решением задачи минимизации.

Задача, завязанная на реализации этого алгоритма, изложена ниже.

Необходимо взять целевую функцию (1), и найти ее минимум.

|  |  |
| --- | --- |
| f(x) = x1^2 + 4 \* x1 – x2 + 18 \* x2^2, | (1) |

Далее нужно провести исследование, выполнив несколько итераций для параметров, используемых в методе оптимизации. На каждой итерации вычисляется результат выполнения алгоритма для нашей целевой функции. При этом значение текущего параметра должно изменяться на каждой итерации. По результатам исследований мы строим графики зависимостей скорости и точности алгоритмов от значений параметров. На их основании мы делаем дальнейшие умозаключения. Также в конце проводится сравнение со вторым алгоритмом.

Нахождение минимума указано на рисунках 1 и 2.

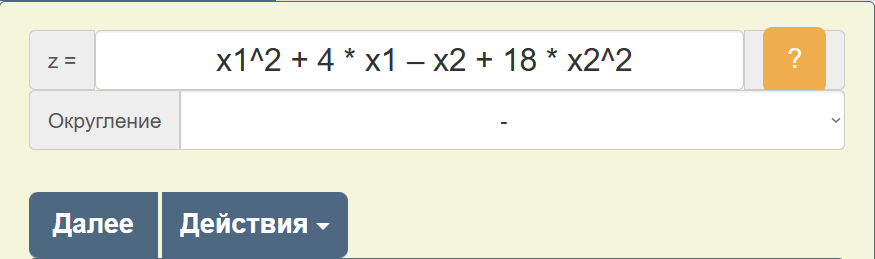


Рисунок 1 — Поле ввода функции.



Рисунок 2 — Результат.

Для данного метода были проведены измерения зависимости абсолютной погрешности и кол-ва вызовов целевой функции от параметров (рисунки 3–12). По их результатам были сделаны следующие далее выводы.

Во-первых, при увеличении шага вдоль одного направления до определенного момента абсолютная погрешность имеет тенденцию к увеличению. Кол-во вычисления функции так же падает при увеличении шага. Это может быть связано с тем, что, чем меньше шаг, тем меньше вероятность попасть «мимо» минимального значения. Скорость же увеличивается, ведь чем больше шаги, тем меньше их требуется для достижения приемлемой области.



Рисунок 3 — График зависимости величины абс. погрешности от величины шага по 1-му направлению.

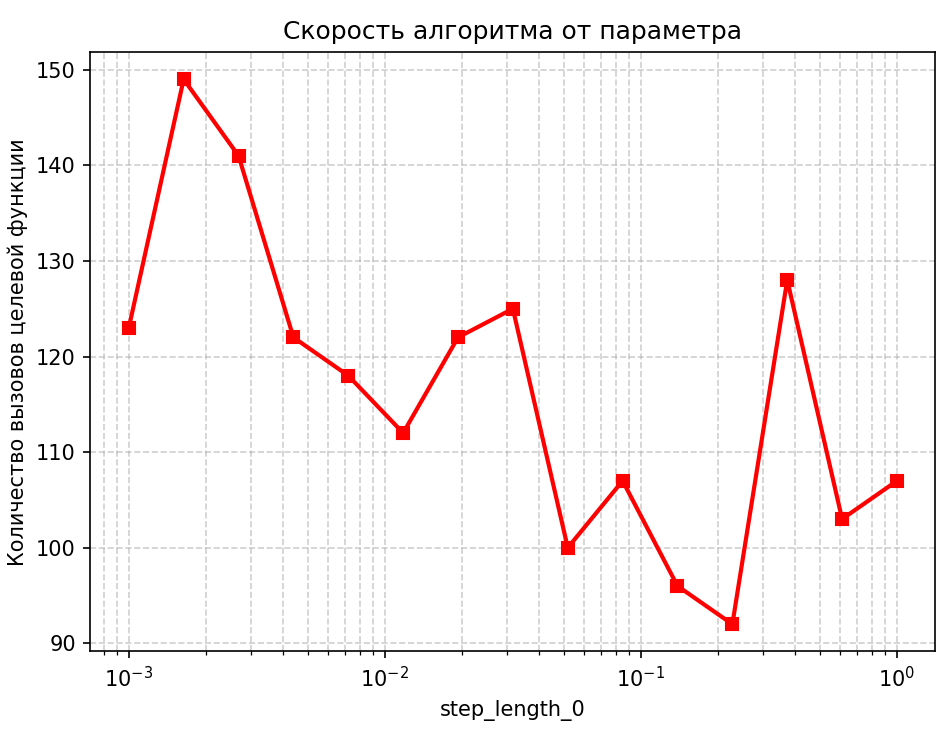


Рисунок 4 — График зависимости величины скорости алгоритма от величины шага по 1-му направлению.

Рассмотрим тенденцию для шага вдоль второго направления. Здесь мы видим, что при увеличении шага вдоль второго направления абсолютная погрешность чаще начинала «подскакивать». Скорость алгоритма уменьшалась, но изменение было незначительное. Между минимальным и максимальным значением разница в 5 вызовов. Просто в определенный момент шаг мог стать настолько большим, что стал «переступать» подходящую область минимального значения.



Рисунок 5 — График зависимости величины абс. погрешности от величины шага по 2-му направлению.



Рисунок 6 — График зависимости величины скорости алгоритма от величины шага по 2-му направлению.

Очевидно, что при увеличении величины числа для окончания алгоритма будет увеличиваться кол-во значений, которые могут быть приняты за минимум. Алгоритм станет быстрее добираться до области с этими значениями и иногда брать «крайние» значения из этой области.

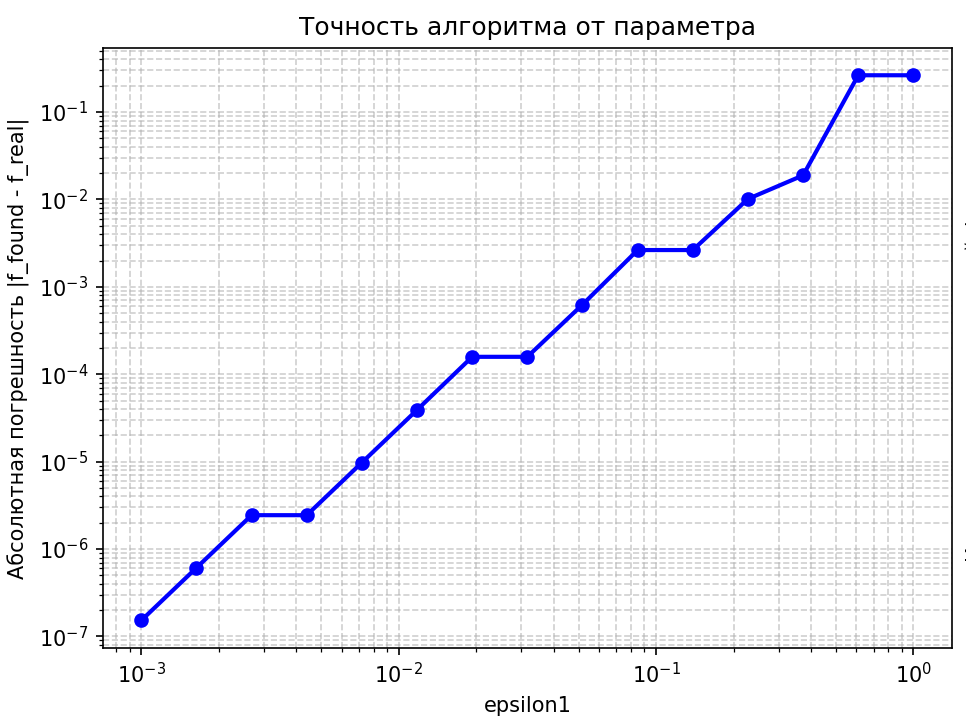


Рисунок 7 — График зависимости величины абс. погрешности от величины числа для окончания алгоритма.

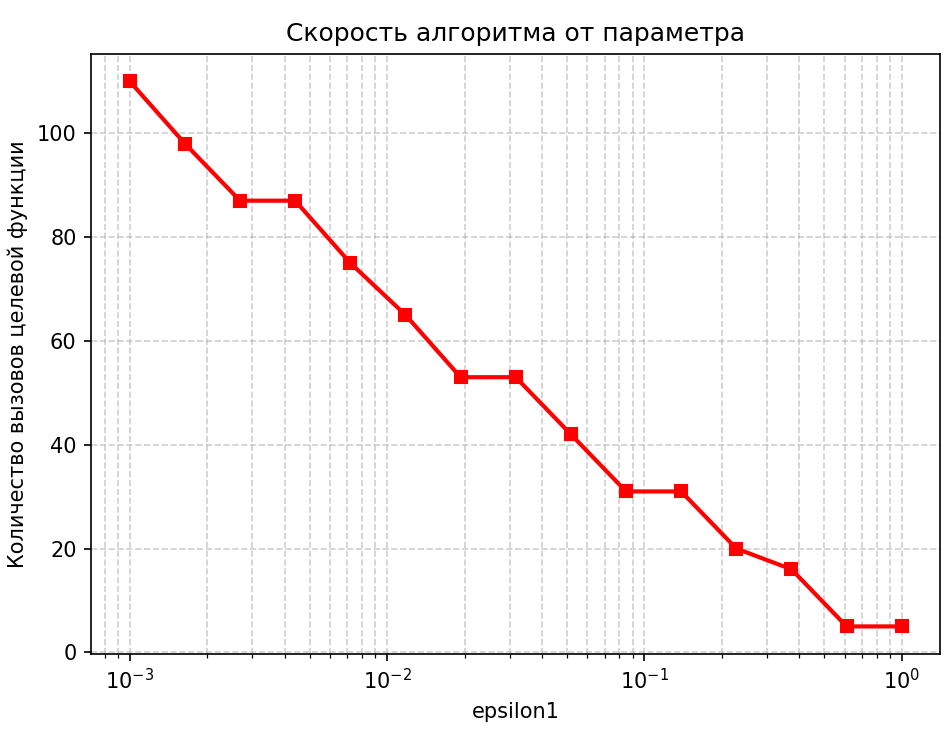


Рисунок 8 — График зависимости величины скорости алгоритма от величины числа для окончания алгоритма.

При увеличении коэффициента для успешного шага начала уменьшаться погрешность (рисунок 9). Скорее всего, шаг начал «перепрыгивать» крайние значения из области минималных значений и пробираться ближе к настоящему минимуму. Мы видим, что советуемым значением для коэффициента для успешного шага не зря является 2, т.к. при приближении к этому значению точность резко увеличилась. Скорость, однако, уменьшилась (рисунок 10). При слишком больших alpha и beta алгоритм слишком агрессивен, часто пропускает минимум, много неудачных попыток. Такая же логика применима к коэфиценту для неудачных значений. Как мы видим, рекомендация брать оптимальный коэфицент beta равный 0.5 вполне оправдана (рисунки 11, 12).

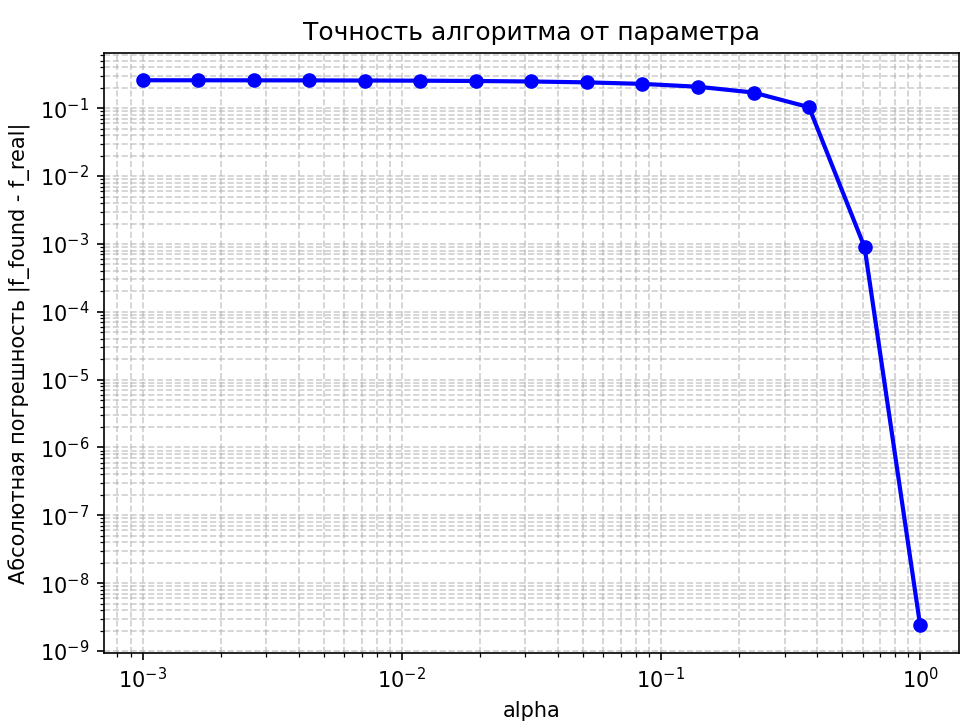


Рисунок 9 — График зависимости величины абс. погрешности от коэффициента для успешного шага.



Рисунок 10 — График зависимости величины скорости алгоритма от коэффициента для успешного шага.

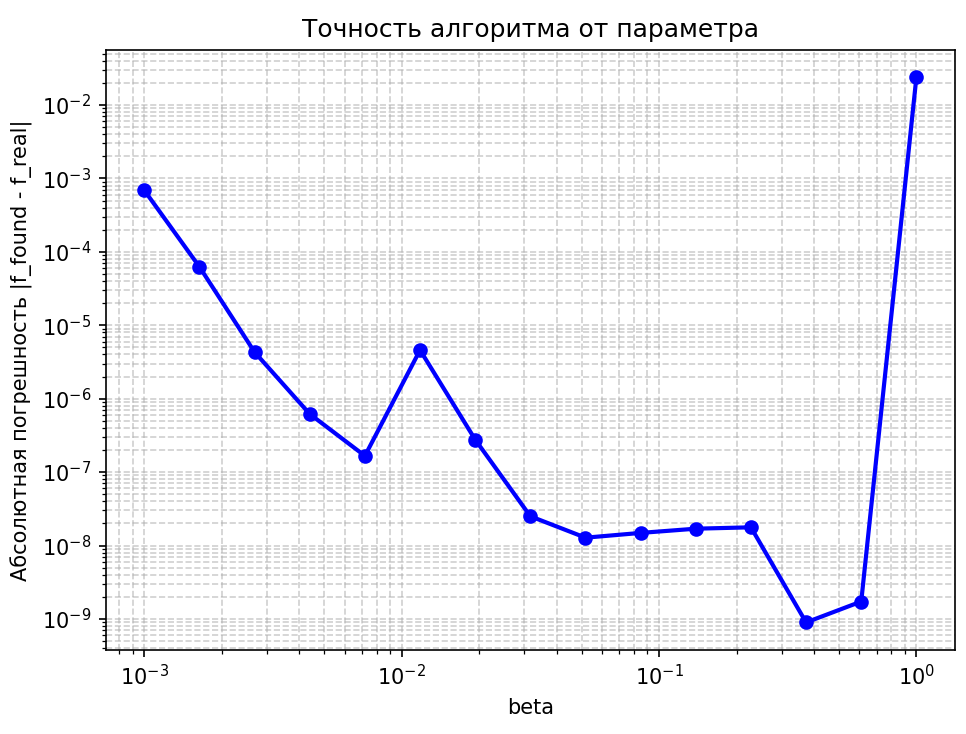


Рисунок 11 — График зависимости величины абс. погрешности от коэффициента для неудачного шага.



Рисунок 12 — График зависимости величины скорости алгоритма от коэффициента для неудачного шага.

Сам код программы состоит из следующих составных частей**:**

* intCount() – счетчик вызовов целевой функции;
* fun(x) - вызов целевой функции;
* norm(v) – вычисление длины вектора через евклидову норму, чтобы далее сделать его единичным, используется при построении новых ортогональных векторов;
* gram\_schmidt(directions, displacement) – метод для перестройки направлений(ортогонализации);
* rozenbrock(...) – основной алгоритм;
* plot\_accuracy\_and\_speed\_vs\_parameter(...) – метод для постройки графиков.

Код для алгоритма Розенбока представлен в листинге 1.

Листинг 1 — Код для метода Розенбока

"""

алгоритм Розенброка

"""

Продолжение листинга 1

import numpy as np

import math

import matplotlib.pyplot as plt

# Глобальный счётчик вызовов функции

count = 0

def incCount():

    global count

    count += 1

def fun(x):

    """

    Целевая функция для минимизации.

    Можно заменить на любую другую.

    """

    incCount()

    # return (1 - x[0])\*\*2 + 100 \* (x[1] - x[0]\*\*2)\*\*2 # Функция Розенброка

    return x[0] \*\* 2 + 4 \* x[0] - x[1] + 18 \* x[1] \*\* 2 # Ваша текущая функция

    # return x[0] \*\* 2 + 45 \* x[0] + x[1] + 30 \* x[1] \*\* 2

def norm(v):

    """Евклидова норма вектора"""

    return math.sqrt(sum(vi\*\*2 for vi in v))

def gram\_schmidt(directions, displacement):

    """

    Перестройка направлений по методу Грама-Шмидта.

    Первое направление — нормированный вектор перемещения.

    Остальные — ортогонализуются относительно него.

    """

    n = len(directions)

    new\_directions = []

    # Первое направление — нормированный вектор перемещения

    d0 = np.array(displacement, dtype=float)

    d0 /= norm(d0)

    new\_directions.append(d0)

    # Остальные — ортогонализуем

    for i in range(1, n):

        v = np.array(directions[i], dtype=float)

        for j in range(len(new\_directions)):

            proj = np.dot(v, new\_directions[j]) \* new\_directions[j]

            v -= proj

        if norm(v) > 1e-10:  # избегаем нулевых векторов

            v /= norm(v)

Продолжение листинга 1

        else:

            # если вектор стал нулевым — генерируем случайный ортогональный

            v = np.random.randn(n)

            for j in range(len(new\_directions)):

                v -= np.dot(v, new\_directions[j]) \* new\_directions[j]

            v /= norm(v)

        new\_directions.append(v)

    return [d.tolist() for d in new\_directions]

import numpy as np

import matplotlib.pyplot as plt

# --- Добавляем новую функцию для экспериментов и построения графиков ---

def plot\_accuracy\_and\_speed\_vs\_parameter(

    param\_name="step\_length\_0",

    param\_values=np.logspace(-3, 0, 15),  # от 0.001 до 1.0

    fixed\_params=None,

    x0=[-1.2, 0.0],

    max\_iter=1000,

    fuilure\_count\_border=10,

    real\_min=(-2, -1 / 36),

    real\_fmin=-60751 / 120,

):

    """

    Эксперимент: запуск алгоритма Розенброка при различных значениях одного параметра.

    Параметры:

   - param\_name: имя параметра для варьирования ('step\_length\_0', 'step\_length\_1', 'epsilon1', 'alpha', 'beta')

    - param\_values: список значений параметра для перебора

    - fixed\_params: словарь фиксированных параметров (если None — используются стандартные)

    - x0: начальная точка

    - max\_iter: макс. итераций

    - fuilure\_count\_border: порог неудачных циклов

    - real\_min: реальный минимум (x0, x1)

    - real\_fmin: реальное значение функции в минимуме

    Вывод: два графика — точность и скорость (вызовы функции) в зависимости от параметра.

    """

    global count  # Используем глобальный счётчик

Продолжение листинга 1

    accuracies = []  # |f\_found - f\_real|

    speeds = []  # число вызовов fun()

    print(f"Проведение экспериментов по параметру '{param\_name}'...")

    for param\_val in param\_values:

        # Устанавливаем параметры

        if fixed\_params is None:

            step\_lengths = [0.3, 0.1]

            epsilon1 = 1e-4

            alpha = 2.0

            beta = 0.5

        else:

            step\_lengths = fixed\_params.get("step\_lengths", [0.3, 0.1])

            epsilon1 = fixed\_params.get("epsilon1", 1e-4)

            alpha = fixed\_params.get("alpha", 2.0)

            beta = fixed\_params.get("beta", 0.5)

        # Применяем текущее значение параметра

        if param\_name == "step\_length\_0":

            step\_lengths[0] = param\_val

        elif param\_name == "step\_length\_1":

            step\_lengths[1] = param\_val

        elif param\_name == "epsilon1":

            epsilon1 = param\_val

        elif param\_name == "alpha":

            alpha = param\_val

        elif param\_name == "beta":

            beta = param\_val

        else:

            raise ValueError(f"Неизвестный параметр: {param\_name}")

        # Сбрасываем счётчик вызовов функции

        global count

        count = 0

        # Запускаем алгоритм

        try:

            x\_best, f\_best, points, iterations = rozenbrock(

                x0=x0,

                step\_lengths=step\_lengths,

                epsilon1=epsilon1,

                alpha=alpha,

                beta=beta,

                max\_iter=max\_iter,

Продолжение листинга 1

                fuilure\_count\_border=fuilure\_count\_border,

            )

        except Exception as e:

            print(f"Ошибка при param={param\_val:.4f}: {e}")

            accuracies.append(np.nan)

          speeds.append(np.nan)

            continue

        # Вычисляем точность как |f\_found - f\_real|

        accuracy = abs(f\_best - real\_fmin)

        speed = count  # количество вызовов fun()

        print("Точность:", accuracy, "Скорость:", speed)

        accuracies.append(accuracy)

        speeds.append(speed)

        print(f"param={param\_val:.4f}, accuracy={accuracy:.2e}, calls={speed}")

    # --- Графики ---

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

    # График 1: Точность

    ax1.plot(param\_values, accuracies, "o-", color="blue", linewidth=2, markersize=6)

    ax1.set\_xscale(

        "log"

    )  # Логарифмическая шкала по оси X — удобно для таких параметров

    ax1.set\_xlabel(f"{param\_name}")

    ax1.set\_ylabel("Абсолютная погрешность |f\_found - f\_real|")

    ax1.set\_title("Точность алгоритма от параметра")

    ax1.grid(True, which="both", linestyle="--", alpha=0.6)

    ax1.set\_yscale(

        "log"

    )  # Логарифмическая по Y — так как погрешности могут быть очень малы

    # График 2: Скорость (количество вызовов функции)

    ax2.plot(param\_values, speeds, "s-", color="red", linewidth=2, markersize=6)

    ax2.set\_xscale("log")

    ax2.set\_xlabel(f"{param\_name}")

ax2.set\_ylabel("Количество вызовов целевой функции")

    ax2.set\_title("Скорость алгоритма от параметра")

    ax2.grid(True, which="both", linestyle="--", alpha=0.6)

    plt.tight\_layout()

Продолжение листинга 1

    plt.show()

    print("------")

    return param\_values, accuracies, speeds

def rozenbrock(

    x0,

    step\_lengths,

    epsilon1=1e-4,

    alpha=2.0,

    beta=0.5,

    max\_iter=1000,

    fuilure\_count\_border=10,

):

    """

    Алгоритм Розенброка (метод вращающихся координат)

    Параметры:

    - x0: начальная точка [x1, x2, ...]

    - step\_lengths: начальные длины шагов по каждому направлению [Δ1, Δ2, ...]

    - epsilon1: порог по изменению координат

    - epsilon2: порог по изменению функции

    - alpha: коэффициент увеличения шага (>1)

    - beta: коэффициент уменьшения шага (0 < beta < 1)

    - max\_iter: максимальное число итераций

    Возвращает:

    - x\_best: найденная точка минимума

    - f\_best: значение функции в точке минимума

    - points: траектория поиска

    - iterations: число итераций

    """

    consecutive\_failures = 0 # Счётчик неудачных циклов подряд

    ever\_improved = False

    epsilon\_step = epsilon1  # Порог для длин шагов (переименуем для ясности)

    # Шаг 1: Инициализация

    dim = len(x0)  # кол-во ортогональных направлений

    x\_current = np.array(x0, dtype=float)  #

    steps = np.array(step\_lengths, dtype=float)  # делаем массив с длинами шагов

    # создаем единичную матрицу dim x dim, где каждая строка - нормлаьный питоновский массив

Продолжение листинга 1

    directions = [

        np.eye(dim)[i].tolist() for i in range(dim)

    ]  # Начальные направления — оси координат

    points = [x\_current.copy().tolist()]

    iteration = 0

    f\_current = fun(x\_current)

    print(f"Начало поиска: x0 = {x\_current}, f(x0) = {f\_current:.6f}")

    while iteration < max\_iter:

        points.append(x\_current)

        iteration += 1

        improved = False  # Флаг: было ли улучшение на этой итерации

        x\_start\_cycle = (

            x\_current.copy()

        )  # Сохраняем начало цикла для проверки остановки

        # Шаг 2: Циклический поиск по всем направлениям

        for i in range(dim):

            direction = np.array(directions[i])

            step = steps[i]

            # Пробуем шаг в положительном направлении

            x\_trial = x\_current + step \* direction

            f\_trial = fun(x\_trial)

            if f\_trial < f\_current:

                # Успех: принимаем шаг, увеличиваем длину шага

                x\_current = x\_trial

                f\_current = f\_trial

                steps[i] \*= alpha

                improved = True

                print(

                    f"Итерация {iteration}, направление {i}: УСПЕХ, шаг увеличен до {steps[i]:.6f}"

                )

            else:

                # Неудача: пробуем шаг в отрицательном направлении

                x\_trial = x\_current - step \* direction

                f\_trial = fun(x\_trial)

                if f\_trial < f\_current:

                    # Успех в обратном направлении

                    x\_current = x\_trial

Продолжение листинга 1

                  f\_current = f\_trial

                    steps[i] \*= alpha

                    improved = True

                    print(

                        f"Итерация {iteration}, направление {i}: УСПЕХ (обратное), шаг увеличен до {steps[i]:.6f}"

                    )

                else:

                    # Неудача: уменьшаем шаг

                    steps[i] \*= beta

                    print(

                        f"Итерация {iteration}, направление {i}: НЕУДАЧА, шаг уменьшен до {steps[i]:.6f}"

                    )

        if improved:

            ever\_improved = True

      # Шаг 3: Проверка условий остановки

        if all(step < epsilon\_step for step in steps):

            print(f"\nСходимость достигнута на итерации {iteration}")

            print(f"Все шаги меньше {epsilon\_step}: {[f'{s:.2e}' for s in steps]}")

            print(f"Точка минимума: x\* = {x\_current}")

            print(f"f(x\*) = {f\_current:.6f}")

            break

        # Обновляем счётчик неудач

        if not improved:

            consecutive\_failures += 1

        else:

            consecutive\_failures = 0

        # Дополнительный критерий: защита от зацикливания

        if consecutive\_failures > fuilure\_count\_border:

            print(f"\n{consecutive\_failures} циклов без улучшения — останавливаемся")

            break

        # Шаг 4: Перестройка направлений, если не было улучшения

        if not improved and ever\_improved:  # <-- ИСПРАВЛЕНА УСЛОВИЕ

            print(

                f"\nИтерация {iteration}: Нет улучшения, но ранее были успехи — перестраиваем направления"

            )

            displacement = x\_current - x\_start\_cycle

Окончание листинга 1

            # Проверка, что вектор перемещения не нулевой (защита от деления на 0)

            if norm(displacement) > 1e-12:

                new\_directions = gram\_schmidt(directions, displacement.tolist())

                directions = new\_directions

            else:

                print("Вектор перемещения нулевой — пропускаем перестройку")

        elif not improved and not ever\_improved:

            print(

                f"\nИтерация {iteration}: Нет улучшений с самого начала — возможно, шаги слишком велики или начальная точка в минимуме"

            )

    else:

        print(f"\nДостигнуто максимальное число итераций ({max\_iter})")

    return x\_current, f\_current, points, iteration

def main():

    print("=== Запуск алгоритма Розенброка ===")

    p\_names = ["step\_length\_0", "step\_length\_1", "epsilon1", "alpha", "beta"]

    for param\_name in p\_names:

        param\_vals, accs, spds = plot\_accuracy\_and\_speed\_vs\_parameter(

            param\_name=param\_name,

            param\_values=np.logspace(-3, 0, 15),  # 15 точек от 0.001 до 1.0

            x0=[-1.2, 0.0],

            real\_min=(-2, 1 / 36),

            real\_fmin=-289 / 72,

        )

if \_\_name\_\_ == "\_\_main\_\_":

    main()

* 1. **Метод сопряженных направлений**

Второй алгоритм, который мне простояло реализовать, был метод сопряженных векторов.

Задаётся начальная точка и система из *n* линейно независимых направлений — обычно собственных векторов матрицы Гессе целевой функции. На каждой итерации выполняется последовательная одномерная минимизация вдоль каждого из направлений. После завершения цикла по всем направлениям формируется новое сопряжённое направление как разность между текущей и предыдущей точками, полученной после полного прохода. Это направление добавляется к системе, а старые направления заменяются на ортогонализованную систему, основанную на всех предыдущих перемещениях. Для минимизации вдоль каждого направления применяется метод дихотомии или другой одномерный поиск. Поскольку для квадратичной функции любые *n* сопряжённых направлений обеспечивают достижение глобального минимума за ровно n итераций, метод гарантирует точное нахождение минимума (в пределах точности одномерного поиска) без необходимости вычисления градиентов.

Задача, завязанная на реализации этого алгоритма, изложена ниже.

Необходимо взять целевую функцию (2), и найти ее минимум.

|  |  |
| --- | --- |
| f(x) = 2\*x1\*\*2 + 2\*x2\*\*2 + x1\*x2 + 4\*x2\*\*2, | (2) |

Далее нужно провести исследование, выполнив несколько итераций для параметров, используемых в методе оптимизации. На каждой итерации вычисляется результат выполнения алгоритма для нашей целевой функции. При этом значение текущего параметра должно изменяться на каждой итерации. По результатам исследований мы строим графики зависимостей скорости и точности алгоритмов от значений параметров. На их основании мы делаем дальнейшие умозаключения. Также в конце проводится сравнение со вторым алгоритмом.

Для начала найдем минимум функции через онлайн-сервис.

Нахождение минимума указано на рисунках 13 и 14.

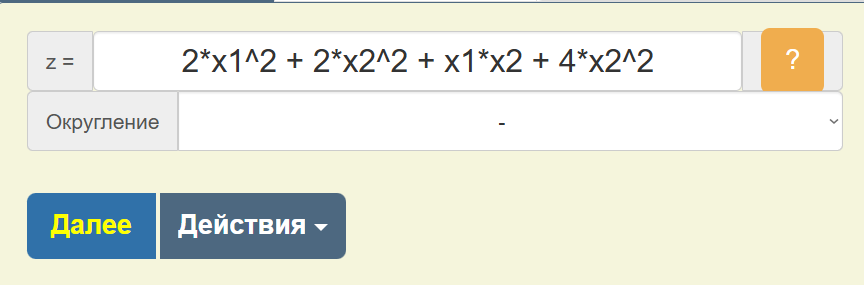


Рисунок 13 — Поле ввода функции.

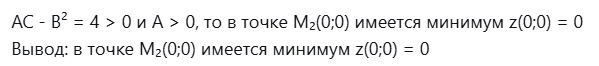


Рисунок 14 — Вывод результата.

Для данного алгоритма были проведены измерения зависимости абсолютной погрешности и кол-ва вызова целевой функции от параметров (рисунки 15–18).

Возможно, равные скачки происходят из-за того, что всегда берется один эпсилон, а значения скачут туда-сюда, так как при увеличении диапазона минимум может появиться то с одной стороны границ ограниченной области, то с другой (рисунок 15).

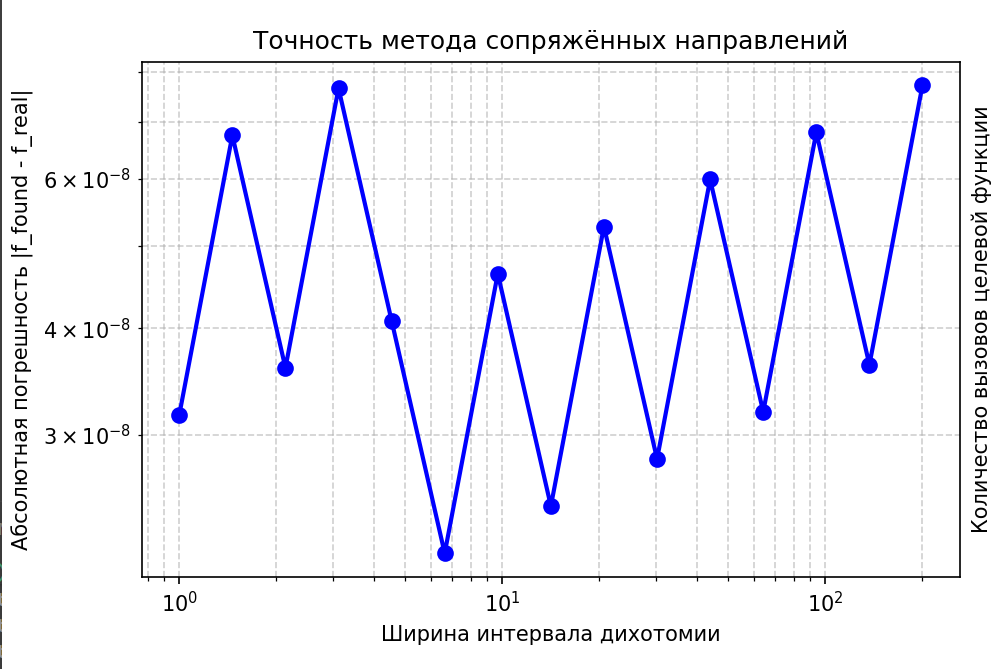


Рисунок 15 — График зависимости величины абсолютной погрешности алгоритма отширины диапазона дихотомии.

Очевидно, что чем больше интервал поиска, тем дольше придется искать на нем минимум и больше раз придется вызвать целевую функцию (рисунок 16).



Рисунок 16 — График зависимости величины скорости алгоритма от ширины диапазона дихотомии.

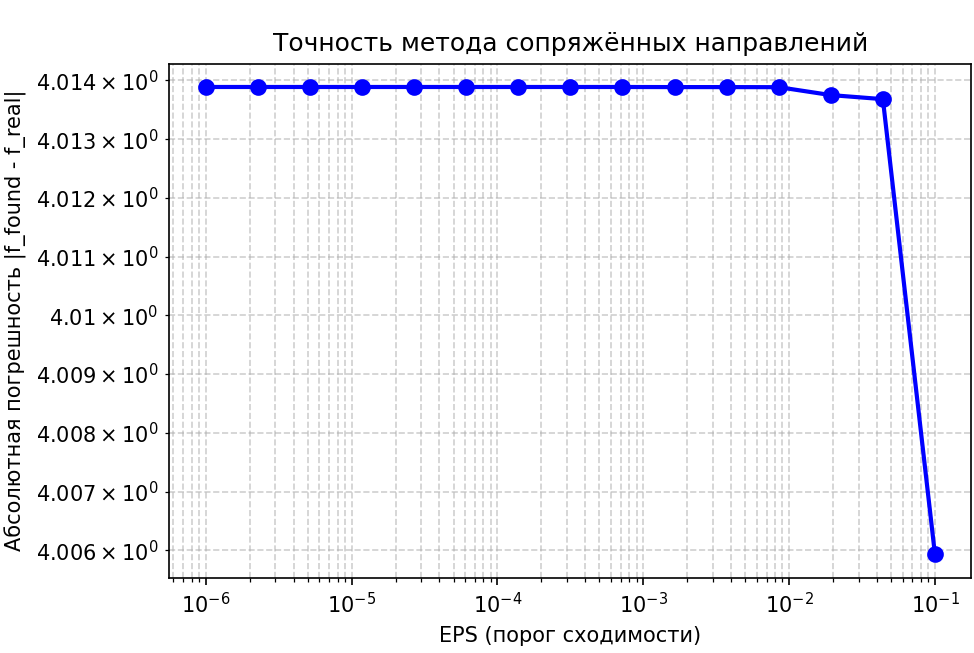


Рисунок 17 — График зависимости величины точности алгоритма от величины порога сходимости.

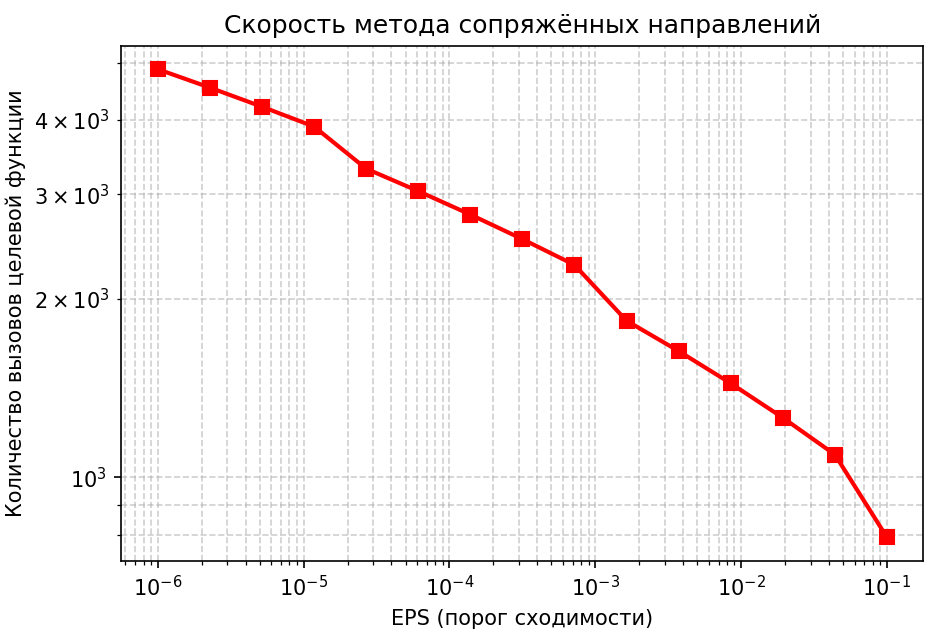


Рисунок 18 — График зависимости величины скорости алгоритма от величины порога сходимости.

В коде использовались следующие методы:

* f(\*args) - Обёртка над целевой функцией, которая автоматически считает количество вызовов функции — для оценки "скорости" алгоритма;
* dichotomy(f\_sym, dichotomy\_range=[-50, 50]) - Находит минимум одномерной функции f\_sym(s) на заданном интервале методом дихотомии (точный линейный поиск);
* conjdirmethod(dichotomy\_range=[-50, 50]) - Реализация метода сопряжённых направлений для минимизации квадратичной функции. Использует собственные векторы гессиана как начальные направления;
* analyze\_conjdirmethod\_by\_EPS() – функция для построения графика зависимостей от EPS;
* analyze\_conjdirmethod\_by\_dichotomy\_width() – функция для построения графика зависимостей от ширины интервала метода Дихотомии.

Код для алгоритма сопряженных векторов представлен в листинге 2.

Листинг 2 — Код для метода сопряженных векторов

import numpy as np

import matplotlib.pyplot as plt

from sympy import symbols, parse\_expr, hessian, Matrix

from sympy.abc import s

from sympy.utilities.lambdify import lambdify

from functools import reduce

# --- Настройки (должны быть доступны в глобальной области) ---

ROUNDING\_NUMBER = 3

MAX\_ITER = 20

NUMBER\_OF\_VARS = 2

x\_vars = symbols("x1:%d" % (NUMBER\_OF\_VARS + 1))

loss\_function\_str = "2\*x1\*\*2 + 2\*x2\*\*2 + x1\*x2 + 4\*x2\*\*2"

loss\_function = parse\_expr(loss\_function\_str)

real\_min\_point = [0, 0]  # Реальный минимум

real\_fmin = 0  # f(0,0) = 0

# Создаём отслеживаемую функцию

call\_count = 0

Продолжение листинга 2

def f(\*args):

    global call\_count

    call\_count += 1

    return f\_original(\*args)

f\_original = lambdify(x\_vars, loss\_function)

init\_vector = [0, 0]

init\_vector = Matrix(init\_vector)

listmerge = lambda s: reduce(lambda d, el: d.extend(el) or d, s, [])

def rounding\_vectors(vector):

    vector = list(vector.evalf())

    return [round(el, ROUNDING\_NUMBER) for el in vector]

def dichotomy(f\_sym, dichotomy\_range=[-50, 50]):

    """

    Метод дихотомии с настраиваемым диапазоном поиска.

    """

    local\_calls = 0

    f\_lambdified = lambdify(s, f\_sym)

    a, b = dichotomy\_range  # Используем переданный диапазон

    delta = EPS / 2

    while abs(b - a) > EPS:

        x1 = (a + b - delta) / 2

        x2 = (a + b + delta) / 2

        val1 = f\_lambdified(x1)

      val2 = f\_lambdified(x2)

        local\_calls += 2

        global call\_count

        call\_count += local\_calls

        if val1 <= val2:

            b = x2

        else:

            a = x1

    return (a + b) / 2

def conjdirmethod(dichotomy\_range=[-50, 50]):

    """

    Метод сопряжённых направлений.

    Возвращает найденную точку минимума.

Продолжение листинга 2

    """

    global call\_count, EPS, MAX\_ITER

    H = hessian(loss\_function, x\_vars)

    v = [el.evalf() for el in listmerge([el[2] for el in H.eigenvects()])]

    x\_0 = init\_vector

    i = 0

    while True:

        i += 1

        # Первый собственный вектор, скорректированный под текущую точку (хотя для константных EV subs не меняет)

        d = [v[0].subs({x\_vars[k]: x\_0[k] for k in range(NUMBER\_OF\_VARS)})]

        # Первый шаг: движение вдоль первого направления

        x = x\_0 + s \* d[0]

        line\_expr = loss\_function.subs({x\_vars[k]: x[k] for k in range(NUMBER\_OF\_VARS)})

        S = dichotomy(line\_expr, dichotomy\_range=dichotomy\_range)

        x = x.subs(s, S)

        for j in range(1, NUMBER\_OF\_VARS):

            # Начало нового направления

            y0 = x + v[j].subs({x\_vars[k]: x\_0[k] for k in range(NUMBER\_OF\_VARS)})

            y = y0

            # Коррекция вдоль предыдущих направлений

            for k in range(j):

                y = y + s \* d[k]

                line\_expr = loss\_function.subs({x\_vars[k]: y[k] for k in range(NUMBER\_OF\_VARS)})

                S = dichotomy(line\_expr, dichotomy\_range=dichotomy\_range)

                y = y.subs(s, S)

            # Новое направление

            d.append(x - y)

            x = x + s \* d[j]

            line\_expr = loss\_function.subs({x\_vars[k]: x[k] for k in range(NUMBER\_OF\_VARS)})

            S = dichotomy(line\_expr, dichotomy\_range=dichotomy\_range)

            x = x.subs(s, S)

        # Проверка сходимости

        if (x - x\_0).norm() <= EPS:

            break

Продолжение листинга 2

      if i > MAX\_ITER:

            break

        x\_0 = x

    return x

# Существующая функция для EPS (оставлена без изменений для полноты)

def analyze\_conjdirmethod\_by\_EPS():

    """

    Проводит вычислительный эксперимент: анализ влияния параметра EPS

    на точность и скорость метода сопряжённых направлений.

    Точность: |f\_found - f\_real|, где f\_real = f(0,0) = 0

    Скорость: количество вызовов целевой функции f()

    Использует фиксированные настройки:

      - Начальная точка [0, 0]

      - Макс. итераций = 20

      - Целевая функция: 2\*x1\*\*2 + 2\*x2\*\*2 + x1\*x2 + 4\*x2\*\*2

    """

    global call\_count, EPS, init\_vector, MAX\_ITER

    # --- Константы ---

    param\_name = "EPS"

    param\_values = np.logspace(-6, -1, 15)  # от 1e-6 до 0.1 (15 значений)

    real\_fmin = 0.0  # f(0,0) = 0

    init\_vector = Matrix([0, 0])  # фиксированная начальная точка

    MAX\_ITER = 20  # фиксированное макс. число итераций

    accuracies = []  # |f\_found - 0|

    speeds = []  # количество вызовов f()

    print(

        f"🔍 Эксперимент: Влияние параметра {param\_name} на метод сопряжённых направлений"

    )

    print(f"   Целевая функция: {loss\_function\_str}")

    print(f"   Реальный минимум: f(0,0) = {real\_fmin}")

    print(

        f"   Диапазон {param\_name}: [{param\_values[0]:.2e}, {param\_values[-1]:.1f}] ({len(param\_values)} значений)\n"

    )

Продолжение листинга 2

    for i, eps\_val in enumerate(param\_values):

        # Сброс глобальных переменных перед каждым запуском

        call\_count = 0

        EPS = eps\_val

        try:

            # Запуск метода

            min\_x = conjdirmethod()

            min\_x\_list = [float(min\_x[i]) for i in range(NUMBER\_OF\_VARS)]

            f\_found = f(\*min\_x\_list)  # ← через tracked\_f → увеличивает call\_count

            accuracy = abs(f\_found - real\_fmin)

            speed = call\_count

            accuracies.append(accuracy)

            speeds.append(speed)

            print(

                f"EPS={eps\_val:.2e}: f\_found={f\_found:.8f}, "

                f"accuracy={accuracy:.2e}, calls={speed}"

            )

        except Exception as e:

            print(f"EPS={eps\_val:.2e}: ошибка — {e}")

            accuracies.append(np.nan)

            speeds.append(np.nan)

            continue

    # --- Построение графиков ---

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

    # График 1: Точность

    ax1.plot(param\_values, accuracies, "o-", color="blue", linewidth=2, markersize=7)

    ax1.set\_xscale("log")

    ax1.set\_yscale("log")

    ax1.set\_xlabel("EPS (порог сходимости)")

    ax1.set\_ylabel("Абсолютная погрешность |f\_found - f\_real|")

    ax1.set\_title("Точность метода сопряжённых направлений")

    ax1.grid(True, which="both", linestyle="--", alpha=0.6)

    # График 2: Скорость (вызовы функции)

    ax2.plot(param\_values, speeds, "s-", color="red", linewidth=2, markersize=7)

Продолжение листинга 2

  ax2.set\_xscale("log")

    ax2.set\_yscale("log")

    ax2.set\_xlabel("EPS (порог сходимости)")

    ax2.set\_ylabel("Количество вызовов целевой функции")

    ax2.set\_title("Скорость метода сопряжённых направлений")

    ax2.grid(True, which="both", linestyle="--", alpha=0.6)

    plt.suptitle(

        f"Влияние параметра EPS на метод сопряжённых направлений\n"

        f"Целевая функция: {loss\_function\_str}",

        fontsize=14,

        fontweight="bold",

        y=1.02,

    )

    plt.tight\_layout()

    plt.show()

    return param\_values, accuracies, speeds

# НОВАЯ ФУНКЦИЯ: Анализ влияния ширины интервала дихотомии

def analyze\_conjdirmethod\_by\_dichotomy\_width():

    """

    Проводит вычислительный эксперимент: анализ влияния ширины начального интервала дихотомии

    на точность и скорость метода сопряжённых направлений.

    Точность: |f\_found - f\_real|, где f\_real = f(0,0) = 0

    Скорость: количество вызовов целевой функции f()

    Использует фиксированные настройки:

      - Начальная точка [0, 0]

      - EPS = 1e-4 (фиксированный)

      - Макс. итераций = 20

      - Интервал: [-width/2, width/2]

      - Целевая функция: 2\*x1\*\*2 + 2\*x2\*\*2 + x1\*x2 + 4\*x2\*\*2

    """

    global call\_count, EPS, init\_vector, MAX\_ITER

    # --- Константы ---

    param\_name = "Ширина интервала дихотомии"

    param\_values = np.logspace(0, np.log10(200), 15)  # от 1 до 200 (15 значений)

    real\_fmin = 0.0  # f(0,0) = 0

    init\_vector = Matrix([0, 0])  # фиксированная начальная точка

Продолжение листинга 2

  MAX\_ITER = 20  # фиксированное макс. число итераций

    FIXED\_EPS = 1e-4  # Фиксированный EPS для всех запусков

    accuracies = []  # |f\_found - 0|

    speeds = []  # количество вызовов f()

    print(

        f"   Эксперимент: Влияние параметра {param\_name} на метод сопряжённых направлений"

    )

    print(f"   Целевая функция: {loss\_function\_str}")

    print(f"   Реальный минимум: f(0,0) = {real\_fmin}")

    print(f"   Фиксированный EPS: {FIXED\_EPS:.2e}")

    print(

        f"   Диапазон ширины: [{param\_values[0]:.1f}, {param\_values[-1]:.1f}] ({len(param\_values)} значений)\n"

    )

    for i, width\_val in enumerate(param\_values):

        # Сброс глобальных переменных перед каждым запуском

        call\_count = 0

        EPS = FIXED\_EPS

        dichotomy\_range = [-width\_val / 2, width\_val / 2]

        try:

            # Запуск метода с текущим диапазоном

            min\_x = conjdirmethod(dichotomy\_range=dichotomy\_range)

            min\_x\_list = [float(min\_x[i]) for i in range(NUMBER\_OF\_VARS)]

            f\_found = f(\*min\_x\_list)  # ← увеличивает call\_count

            accuracy = abs(f\_found - real\_fmin)

            speed = call\_count

            accuracies.append(accuracy)

            speeds.append(speed)

            print(

                f"  Width={width\_val:.1f}: f\_found={f\_found:.8f}, "

                f"accuracy={accuracy:.2e}, calls={speed}"

            )

        except Exception as e:

            print(f"  Width={width\_val:.1f}: ошибка — {e}")

            accuracies.append(np.nan)

            speeds.append(np.nan)

            continue

Окончание листинга 2

    # --- Построение графиков ---

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

    # График 1: Точность

    ax1.plot(param\_values, accuracies, "o-", color="blue", linewidth=2, markersize=7)

    ax1.set\_xscale("log")

    ax1.set\_yscale("log")

    ax1.set\_xlabel("Ширина интервала дихотомии")

    ax1.set\_ylabel("Абсолютная погрешность |f\_found - f\_real|")

    ax1.set\_title("Точность метода сопряжённых направлений")

    ax1.grid(True, which="both", linestyle="--", alpha=0.6)

    # График 2: Скорость (вызовы функции)

    ax2.plot(param\_values, speeds, "s-", color="red", linewidth=2, markersize=7)

    ax2.set\_xscale("log")

    ax2.set\_yscale("log")

    ax2.set\_xlabel("Ширина интервала дихотомии")

    ax2.set\_ylabel("Количество вызовов целевой функции")

    ax2.set\_title("Скорость метода сопряжённых направлений")

    ax2.grid(True, which="both", linestyle="--", alpha=0.6)

    plt.suptitle(

        f"Влияние ширины интервала дихотомии на метод сопряжённых направлений\n"

        f"Целевая функция: {loss\_function\_str} (EPS={FIXED\_EPS:.2e})",

        fontsize=14,

        fontweight="bold",

        y=1.02,

    )

    plt.tight\_layout()

    plt.show()

    return param\_values, accuracies, speeds

if \_\_name\_\_ == "\_\_main\_\_":

    # Запуск новой функции

    analyze\_conjdirmethod\_by\_dichotomy\_width()

    analyze\_conjdirmethod\_by\_EPS()

1. **Сравнение двух представленных методов**

Для сравнения методов проведем дополнительные исследования первой функции вторым алгоритмом и второй функции первым. В таблицах приведём результаты обоих алгоритмом для каждой из двух функций.

Результаты исследований первой функции первым алгоритмом представлены в таблице 1.

Таблица 1 — Результаты исследования первой функции первым алгоритмом

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Изменяющийся параметр** | **Шаг в первом направлении** | **Шаг во втором направлении** | **Эпсилон** | **alpha** | **beta** |
| Среднее значение  По абсолютной погрешности | 1.6929681360503916e-09 | 2.3893104976006423e-09 | 0.03753879020942629 | 0.19971139510011074 | 0.001643345776990183 |
| Среднее значение  По количеству вызовов целевой функции | 116.33333333333333 | 140.86666666666667 | 51.86666666666667 | 23.733333333333334 | 65.46666666666667 |

Результаты исследований второй функции первым алгоритмом представлены в таблице 2.

Таблица 2 — Результаты исследования второй функции первым алгоритмом

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Изменяющийся параметр** | **Шаг в первом направлении** | **Шаг во втором направлении** | **Эпсилон** | **alpha** | **beta** |
| Среднее значение  По абсолютной погрешности | 4.69e-09 | 6.16e-33 | 2.64e-01 | 2.47e-32 | 2.10e-01 |
| Среднее значение  По количеству вызовов целевой функции | 105 | 62 | 5 | 56 | 50 |

Результаты исследований первой функции вторым алгоритмом представлены в таблице 3.

Таблица 3 — Результаты исследования первой функции вторым алгоритмом

|  |  |  |
| --- | --- | --- |
| **Изменяющийся параметр** | **Ширина интервала для дихотомии (от 1 до 200)** | **Эпсилон** |
| Среднее значение  По абсолютной погрешности | 1.1437481592935228e-0 | 4.013334805793459 |
| Среднее значение  По количеству вызовов целевой функции | 2569.8 | 2633.8 |

Результаты исследований второй функции вторым алгоритмом представлены в таблице 4.

Таблица 4 — Результаты исследования второй функции вторым алгоритмом

|  |  |  |
| --- | --- | --- |
| **Изменяющийся параметр** | **Ширина интервала для дихотомии (от 1 до 200)** | **Эпсилон** |
| Среднее значение  По абсолютной погрешности | 4.662264810233137e-08 | 0.0018306519208770092 |
| Среднее значение  По количеству вызовов целевой функции | 23311 | 27645.4 |

1. **Выводы**

Метод Розенброка продемонстрировал высокую гибкость и адаптивность за счёт динамического изменения шагов и перестройки направлений. Наибольшее влияние на его работу оказывают коэффициенты растяжения (α) и сжатия (β).

Метод сопряжённых направлений показал высокую точность на квадратичных функциях, что ожидаемо благодаря его теоретическим свойствам. Однако он требует значительных вычислительных ресурсов из-за использования метода дихотомии для одномерной минимизации.

Сравнительный анализ показал, что метод Розенброка более экономен по количеству вызовов функции, тогда как метод сопряжённых направлений обеспечивает в среднем более высокую точность на квадратичных функциях, но требует больше вычислений.

Метод сопряжённых направлений теоретически идеален для квадратичных функций, но на практике показал не такой большой разрыв с розенбергом из-за большого числа вызовов функции в одномерном поиске (дихотомии). Метод сопряжённых направлений, реализованный через дихотомию для линейного поиска, непрактичен в реальных задачах из-за огромного количества вычислений. Его следует использовать только в теоретических или учебных целях, либо заменить дихотомию на более быстрый метод одномерной оптимизации (например, золотое сечение).