

Национальный исследовательский университет ИТМО
Факультет информационных технологий и программирования
Прикладная математика и информатика

Методы оптимизации
Отчёт по лабораторной работе №1

Работу выполнили:
Барсукова Ольга М3233
Зызлаев Артем М3233
Ермольев Михаил М3233
Жунусов Данияр М3233

Преподаватель:
Андреев Юрий Александрович

ИТМО

Санкт-Петербург
2025

Постановка задачи и цели работы:

Задача:

Сравнить работу разных методов градиентного спуска на разных функциях в зависимости от выбора начальной точки, стратегии выбора шага или значений гиперпараметров, числа обусловленности.

План:

1. Реализовать метод градиентного спуска с различными стратегиями выбора шага (learning rate scheduling)
2. Реализовать любой метод одномерного поиска и градиентный спуск на его основе;
3. Изучить возможности библиотеки `scipy.optimize`. Найти и использовать для сравнения библиотечные аналоги реализованных нами методов.
4. Применить реализованные нами и библиотечные методы на квадратичных функциях двух переменных
5. Реализовать ещё один метод одномерного поиска и градиентный спуск на его основе и исследовать его на эффективность в сравнении с другими методами.
6. Подобрать мультимодальную функцию, провести исследование для неё.
Подобрать функцию с зашумленными значениями, провести исследование для неё.

Цели:

Разобраться и изучить различные методы градиентного спуска

Изучить библиотеку `scipy.optimize`

Работа выполнена при помощи следующих библиотек:

1. numpy (нужен для арифметических операций с векторами)
2. matplotlib.pyplot (нужен для визуализации, создания графиков и показывает путь градиентного спуска)
3. scipy.optimize (нужен для анализа получившегося результата и сравнения с реализованными градиентными спусками)

Описание кода

Стратегия с выбором шага заключается в том, что мы, находясь в какой-то точке, анализируем направление градиента $\nabla f(x_k)$, указывающее направление роста, и вследствие анализа выполняем шаг вдоль антиградиента $-\nabla f(x_k)$ с оптимально выбранной длиной a_k .

Метод *find_min* необходим для итераций градиентного спуска и в дальнейшем будет использоваться для всех градиентов, реализованных в нашей лабораторной.

Внутри этой функции на каждой итерации мы рассчитываем градиент для новой позиции, а после рассчитываем новую позицию с учетом градиентного шага, добавляем в историю нашего градиентного спуска новый шаг и сравниваем его с прошлым в соответствии с точностью ϵ , также мы добавляем зашумление функции.

```
def find_min(self, func, init: np.array, learning_rate: float = 0.01, eps: float = 0.0001,
             max_iterations: int = 10000, noise_scale: float = 0.1) -> np.array:
    result_min = [init]
    pos = init.copy()

    for i in range(max_iterations):
        direction = self.grad(func, pos)

        direction += np.random.normal(0, noise_scale,
                                      direction.shape)
        direction = -direction

        next_pos = self.gradient_step(func, pos, direction,
                                      learning_rate, eps, max_iterations)
        if np.linalg.norm(next_pos - pos) < eps:
            break
        pos = next_pos
        result_min.append(next_pos)

    return result_min[-1]
```

```

        result_min.append(next_pos)

        if np.linalg.norm(next_pos - pos) < eps:
            break
        pos = next_pos

    return np.array(result_min)

```

Мы реализуем собственное вычисление градиента, для градиентных спусков:

```

@staticmethod
def grad(f, x: np.array, h=1e-5) -> np.array:
    return (f(x[:, np.newaxis] + h * np.eye(x.size)) -
            f(x[:, np.newaxis] - h * np.eye(x.size))) / (2 *
h)

```

Метод *gradient_step* переопределяется всеми классами, наследующимися от *GradientDescending*, и описывает нахождение следующей точки по направлению антиградиента. По умолчанию он изменяет темп обучения в зависимости от выбранной стратегии и продвигает точку на расстояние, зависящее от темпа обучения:

```

def gradient_step(self, func, x: np.array, dir: np.array,
learning_rate: float = 0.5, eps: float = 0.001, max_iterations:
int = 10000) -> np.array:
    self.iter_count +=1
    lr = learning_rate
    if self.step_strategy == "fixed":
        lr = learning_rate
    elif self.step_strategy == "decay":
        lr = learning_rate / (1 + self.iter_count * 0.01)
    elif self.step_strategy == "sqrt_decay":
        lr = learning_rate / np.sqrt(1 + self.iter_count)
    elif self.step_strategy == "exponential":
        lr = learning_rate * (0.95 ** self.iter_count)

    return x + dir * lr

```

Чтобы найти минимум функции по выбранному направлению для квазивыпуклой функции, можно использовать метод дихотомии. Выберем две точки близко к центру отрезка, на котором ищется минимум. Тогда, вычислив функцию в двух точках, можно понять, в какой половине отрезка находится минимум. Соответственно, чтобы уменьшить отрезок, содержащий точку минимума, в два раза, мы должны будет посчитать функцию дважды.

```
def gradient_step(self, func, x: np.array, dir: np.array,
learning_rate: float = 0.5, eps: float = 0.001, max_iterations:
int = 10000) -> np.array:
    left = x
    right = x + dir * learning_rate
    for i in range(max_iterations):
        if np.linalg.norm(func(left) - func(right)) < eps:
            return left
        middle = (left + right) / 2
        step = dir * h
        derive = (func(x + step) - func(x - step)) / 2
        if derive < 0:
            left = middle
        else:
            right = middle
    return left
```

Попробуем минимизировать количество вычислений функции. Для этого воспользуемся следующей идеей – будем выбирать две точки внутри отрезка так, чтобы при сужении отрезка одна из точек, в которых вычисляется значение функции на следующей итерации, совпала с одной из выбранных точек на этой итерации. То есть, если на отрезке $[a, d]$ выбраны точки $b, c: a < b < c < d$, то при сужении отрезка к отрезку $[a, c]$ точка b будет совпадать с правой точкой, выбранной на отрезке $[a, c]$, а при сужении к отрезку $[b, d]$ точка c будет совпадать с левой точкой, выбранной на отрезке $[b, d]$. Оказывается, что для выполнение этого условия отношение каждого следующего отрезка к предыдущему должно быть равно $\frac{\sqrt{5} - 1}{2}$. Это отношение также известно как "золотое сечение". Благодаря этому трюку, на каждой итерации кроме первой потребуется вычислить функцию в одной точке вместо двух. Оказывается, что в таком случае нам удастся приблизиться к точке минимума на расстояние, меньшее заранее выбранного eps , используя меньшее количество вычислений

функции, чем при использовании метода дихотомии. Поэтому приближение методом "золотого сечения" особенно полезно, если вычисление функции занимает значительное количество времени (в случае с нашими квадратичными функциями, которые вычисляются относительно быстро, выгода от использования этого метода не так высока).

```
class GradientGoldenSearchSection(GradientDescending):
    def one_dimension_method(self,
                             func,
                             x: np.array,
                             direction: np.array,
                             learning_rate: float = 0.5,
                             eps: float = 0.001,
                             max_iterations: int = 10000,
                             h=1e-5) -> np.array:
        golden_ratio = (5 ** 0.5 - 1) / 2
        left = x
        right = x + direction * learning_rate
        l, r = (right - left) * (1 - golden_ratio) + left, \
                (right - left) * golden_ratio + left
        func_l = func(l)
        func_r = func(r)
        saved_right = False
        if func_l < func_r:
            right = r
            func_r = func_l
            saved_right = True
        else:
            left = l
            func_l = func_r
            saved_right = False

        for i in range(max_iterations):
            if np.linalg.norm(right - left) < eps:
                return right

            l = (right - left) * (1 - golden_ratio) + left
            r = (right - left) * golden_ratio + left
            if saved_right:
                func_l = func(l)
            else:
```

```

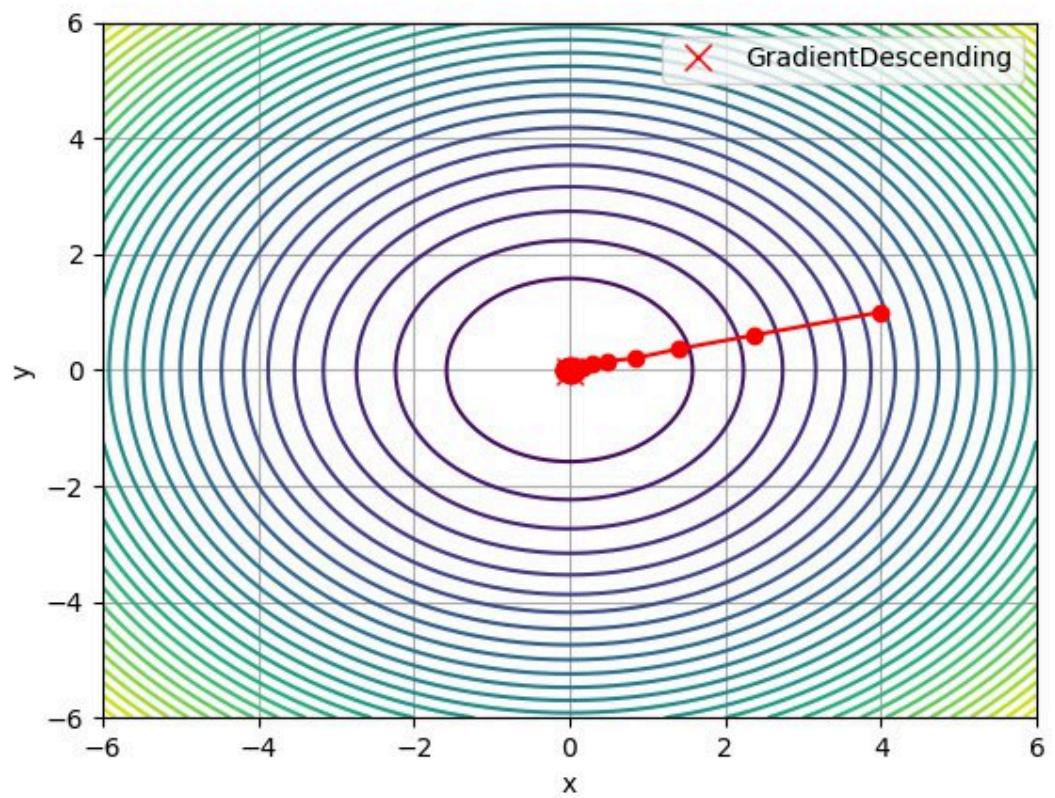
        func_r = func(r)
if func_l < func_r:
    right = r
    func_r = func_l
    saved_right = True
else:
    left = l
    func_l = func_r
    saved_right = False
return right

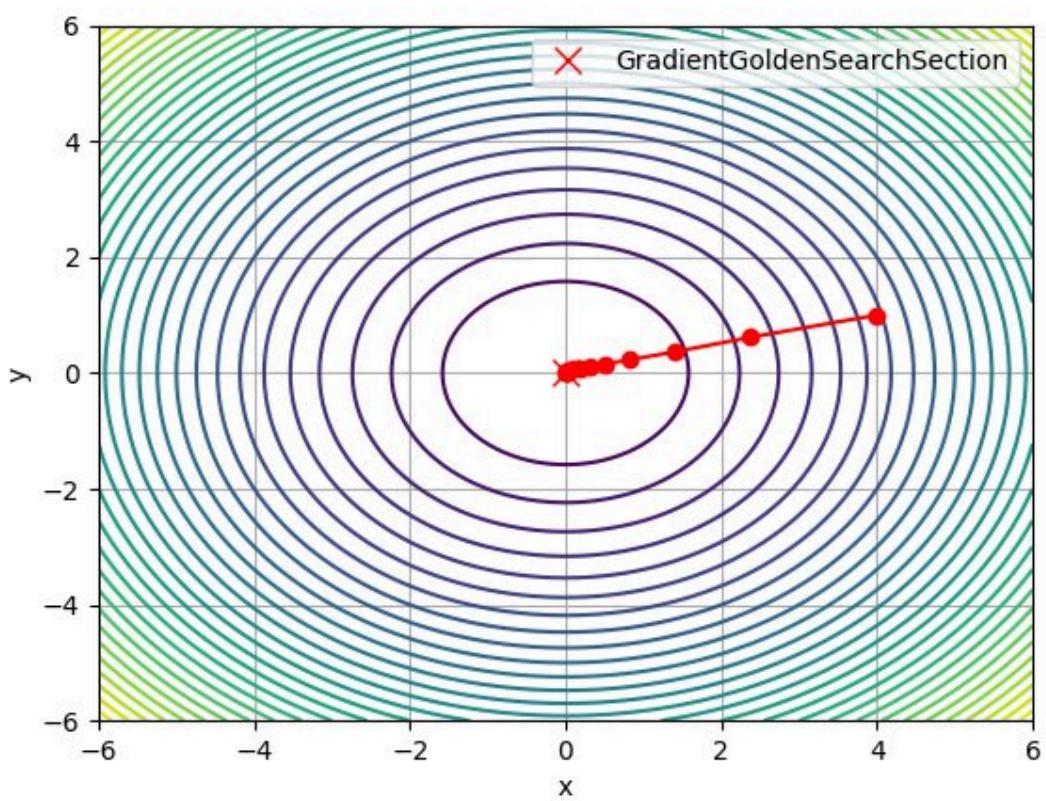
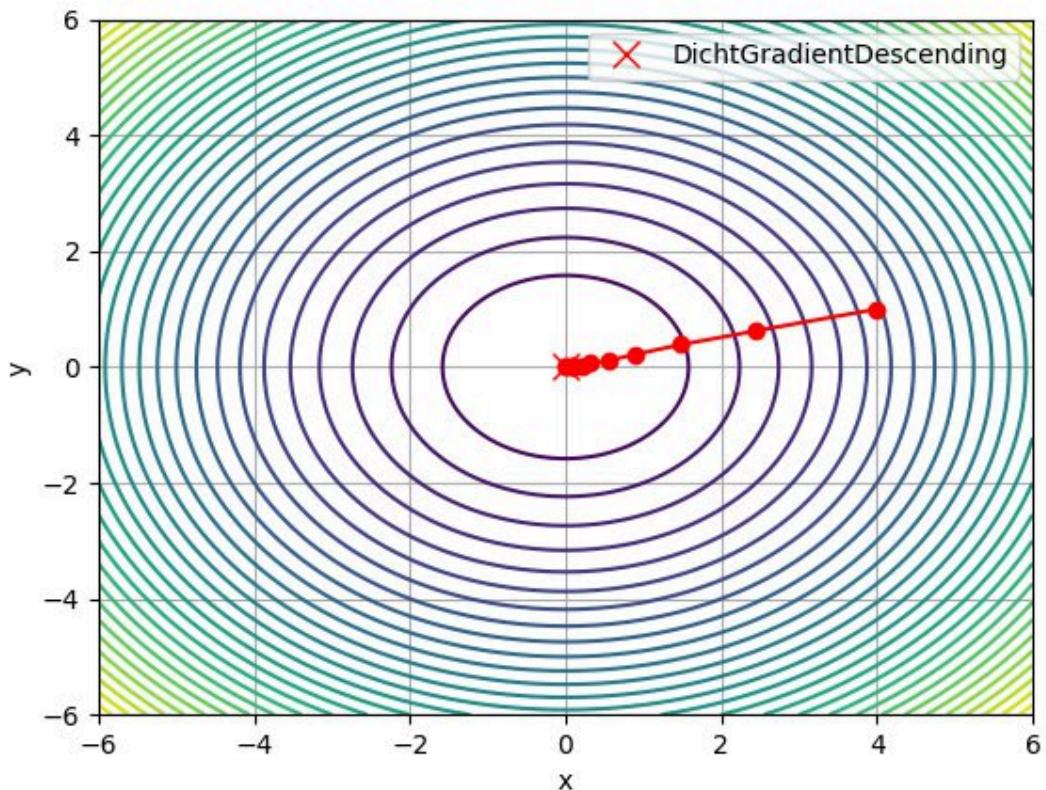
```

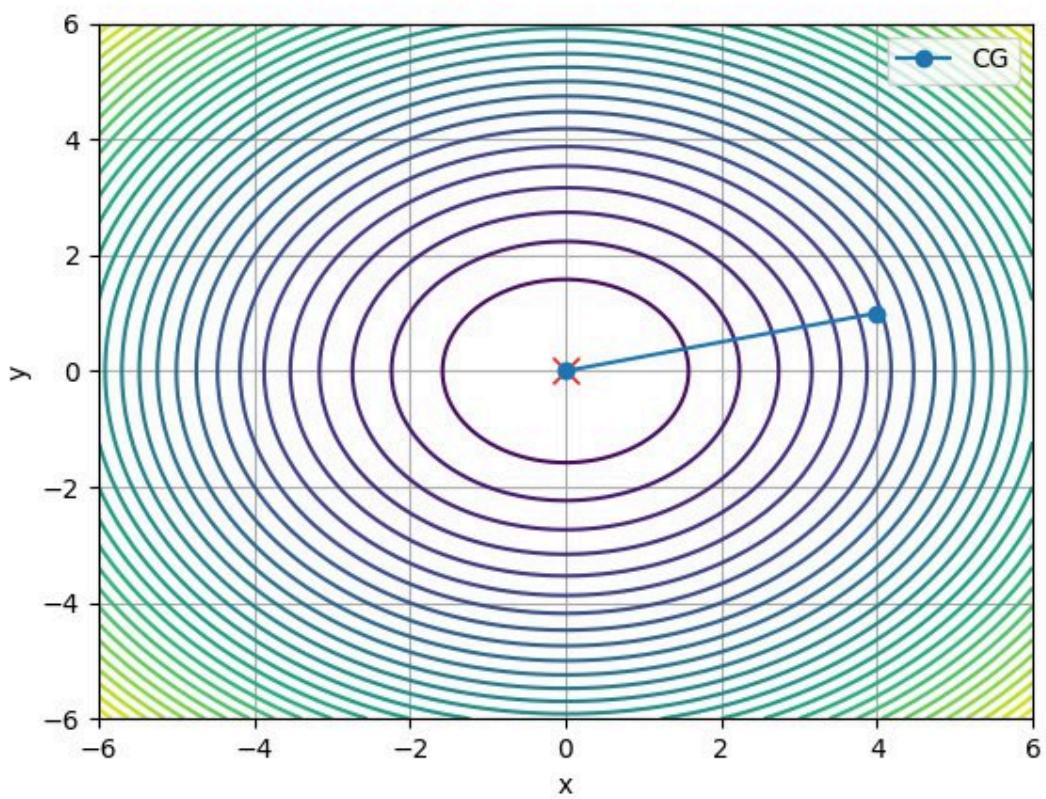
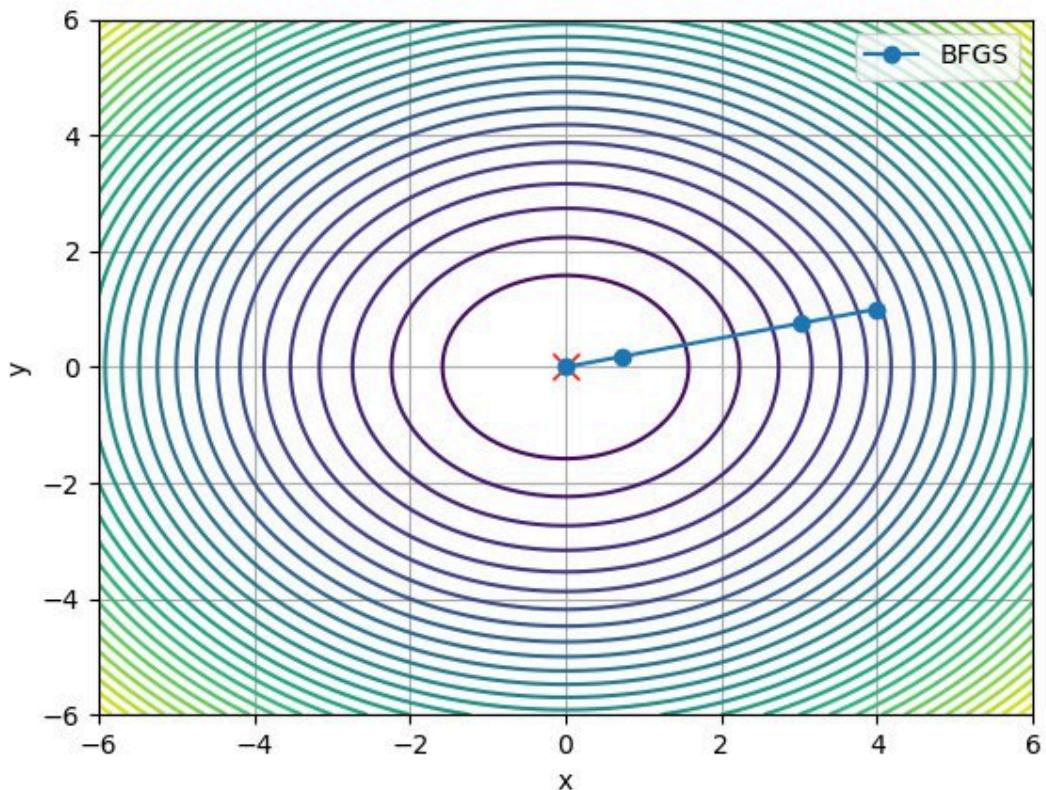
Статистика исполнения различных методов и графическая демонстрация

Пример 1: $x^2 + y^2$

	Total iterations count:	Objective function calls:	Gradient computations:	Result Minimum:
GradientDescending: (fixed)	539	1078	539	5.556735556 709032e-05
GradientDescending: (decay)	43	86	43	0.000760371 6053719853
GradientDescending: (sqrt_decay)	48	96	48	0.001395172 6359427976
GradientDescending: (exponential)	50	100	50	0.000106104 23116288881
DichtGradientDescending	81	162	9	0.006024495 17450469
GradientGoldenSearch	114	144	10	0.000133279 4639044591 5
scipy.optimize('BFGS')	3	12	4	1.251203233 8084958e-14
scipy.optimize('CG')	1	9	3	3.719357486 204178e-13

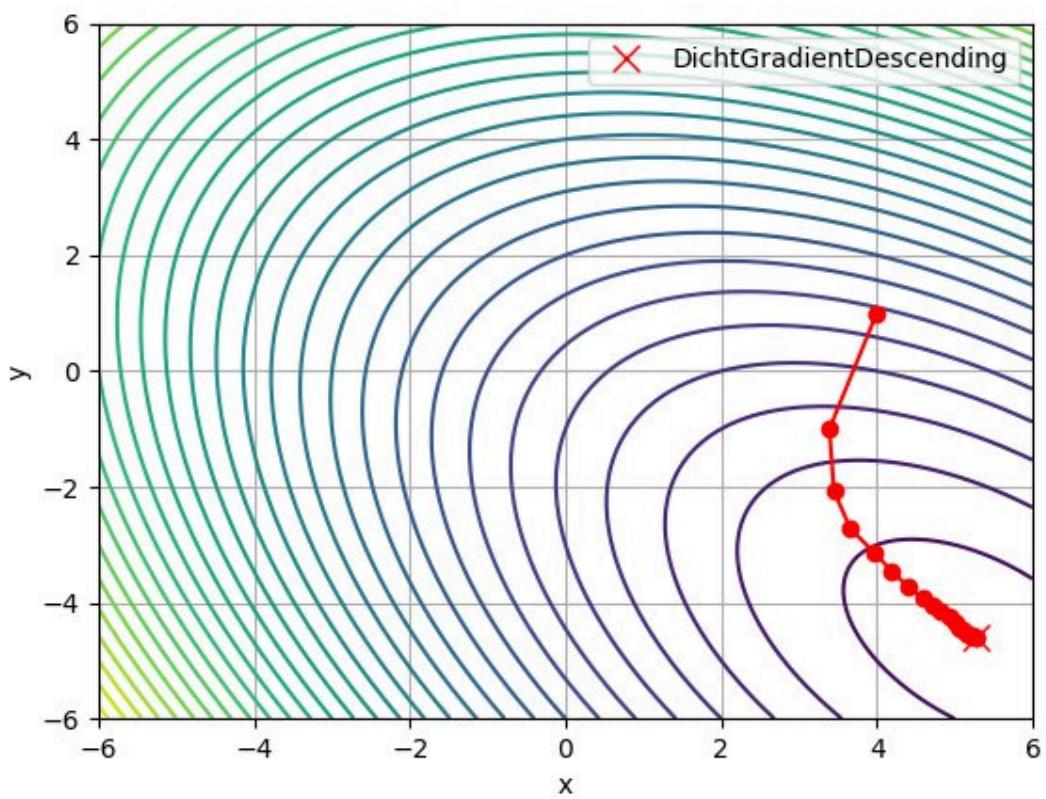
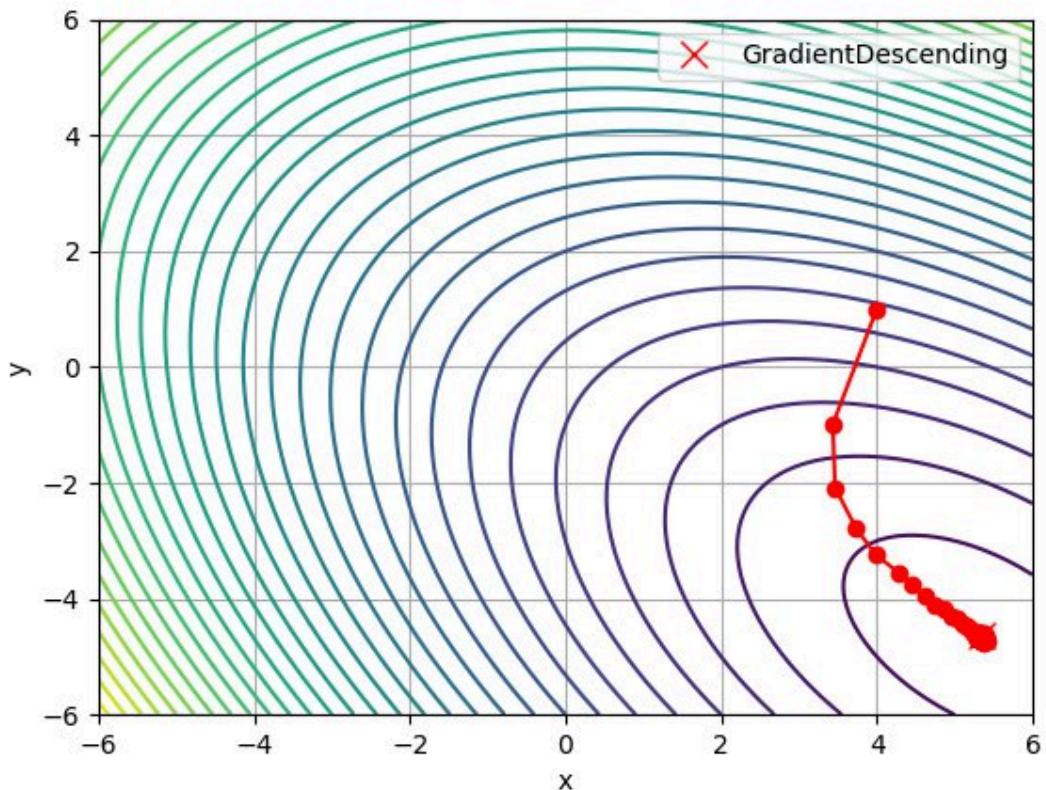


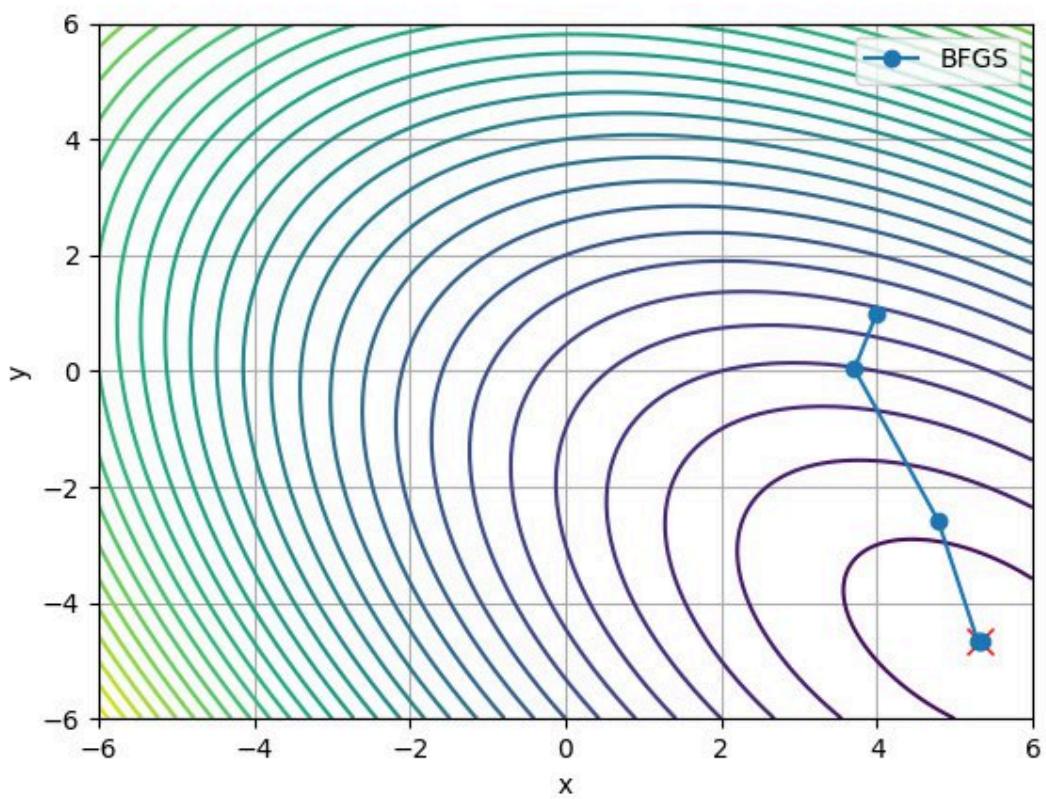
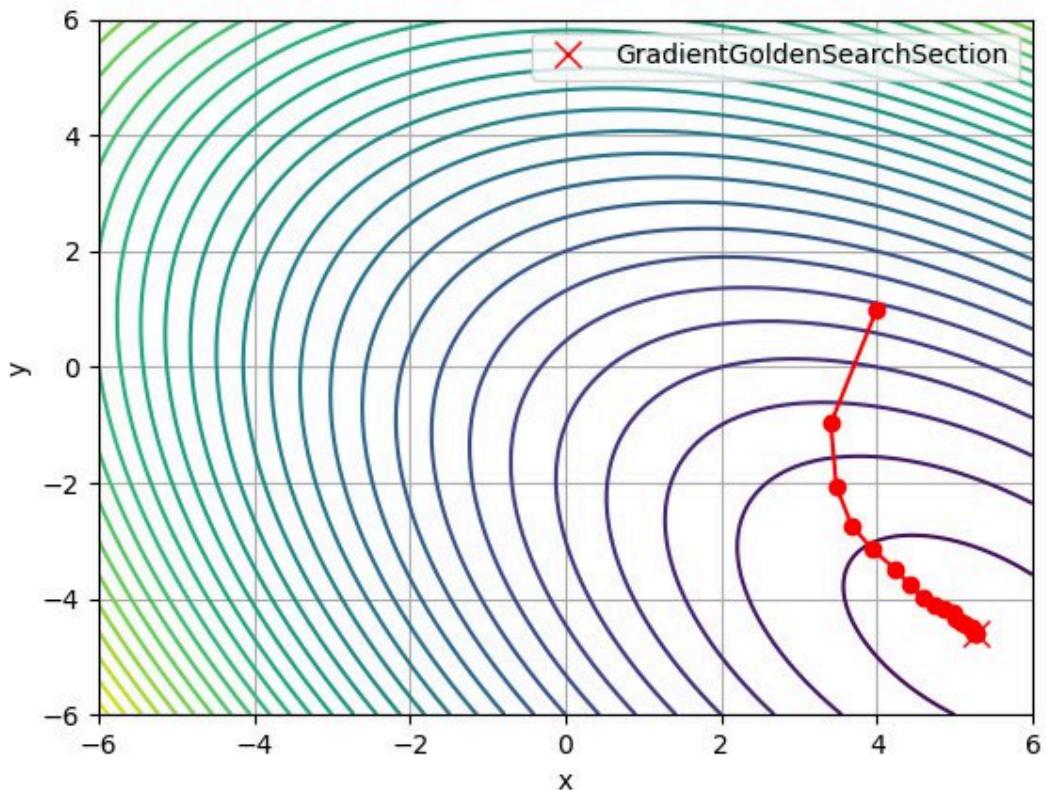


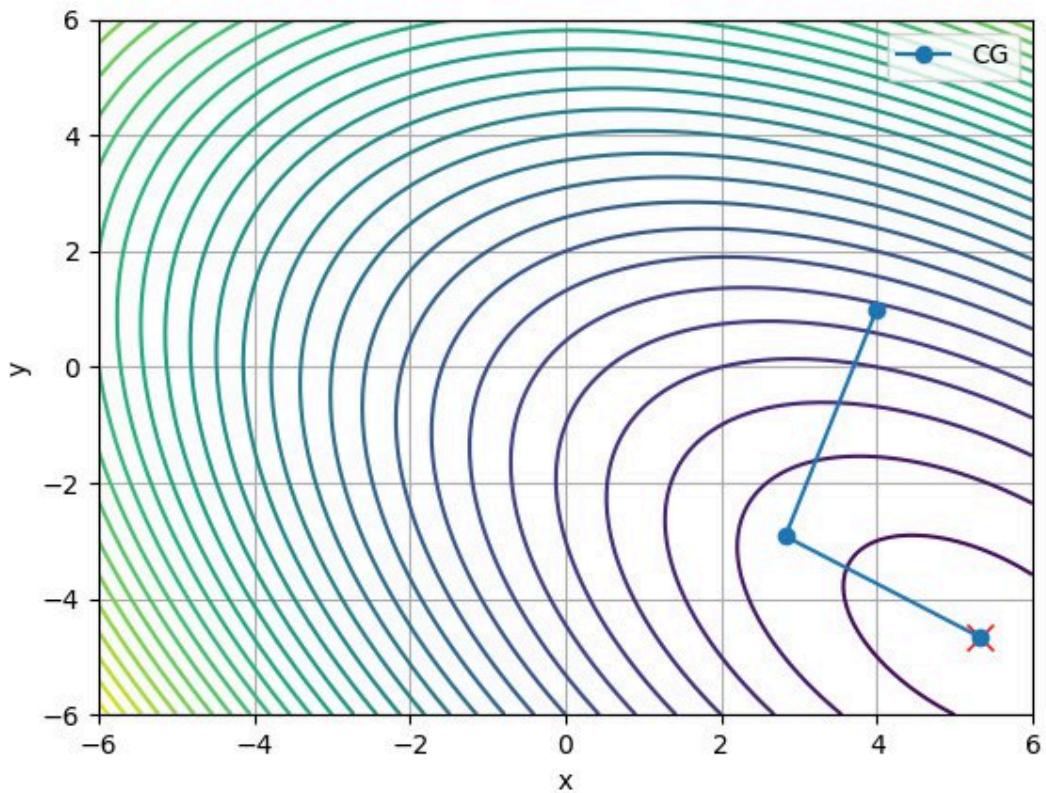


Пример 2: $(x - 3)^2 + (y + 2)^2 + xy$

	Total iterations count:	Objective function calls:	Gradient computations:	Result Minimum:
GradientDescending: (fixed)	347	694	347	-12.3320876 28116017
GradientDescending: (decay)	453	906	453	-12.3332551 07464232
GradientDescending: (sqrt_decay)	90	180	90	-12.3234595 98890867
GradientDescending: (exponential)	51	102	51	-12.3258958 35937644
DichtGradientDescending	163	326	18	-12.3293981 54009443
GradientGoldenSearch	207	264	19	-12.3282713 31883368
scipy.optimize('BFGS')	5	18	6	-12.3333333 33329735
scipy.optimize('CG')	2	15	5	-12.3333333 3333255



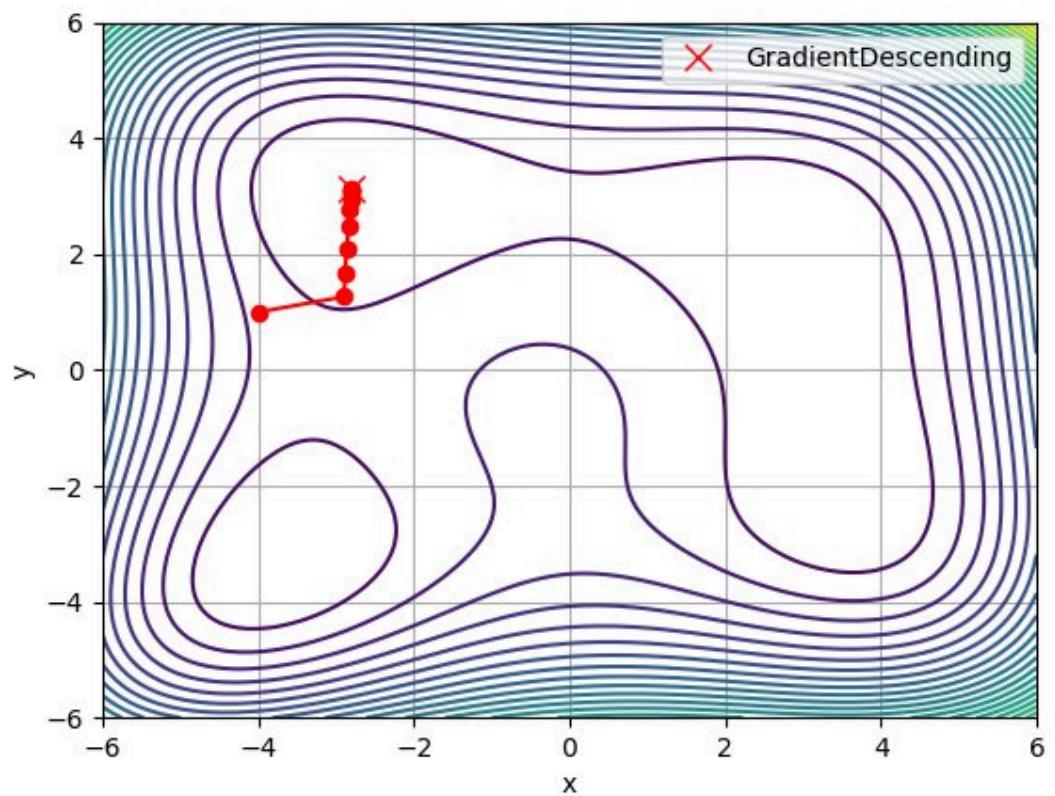


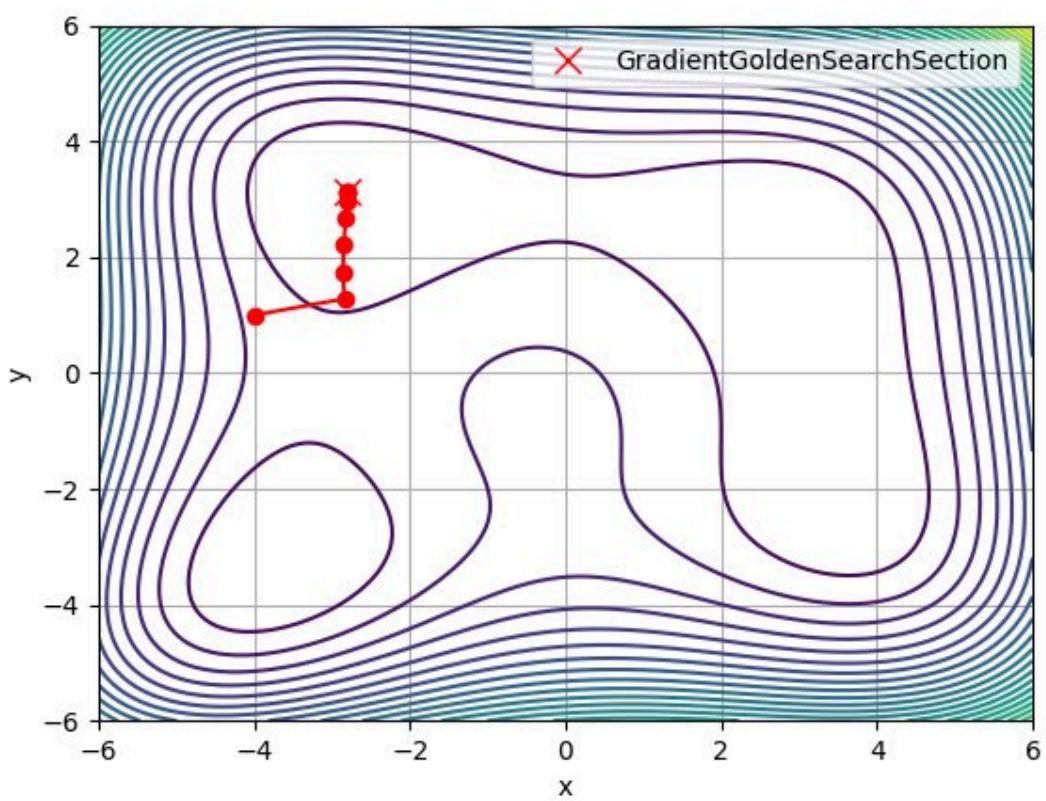
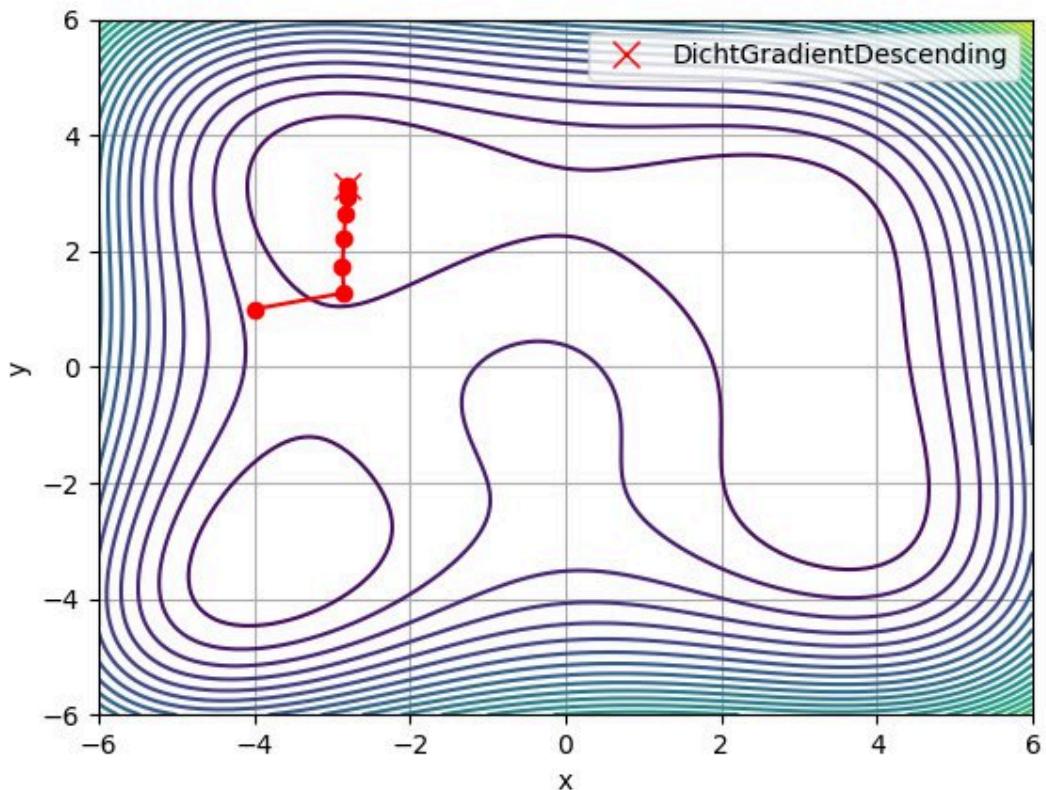


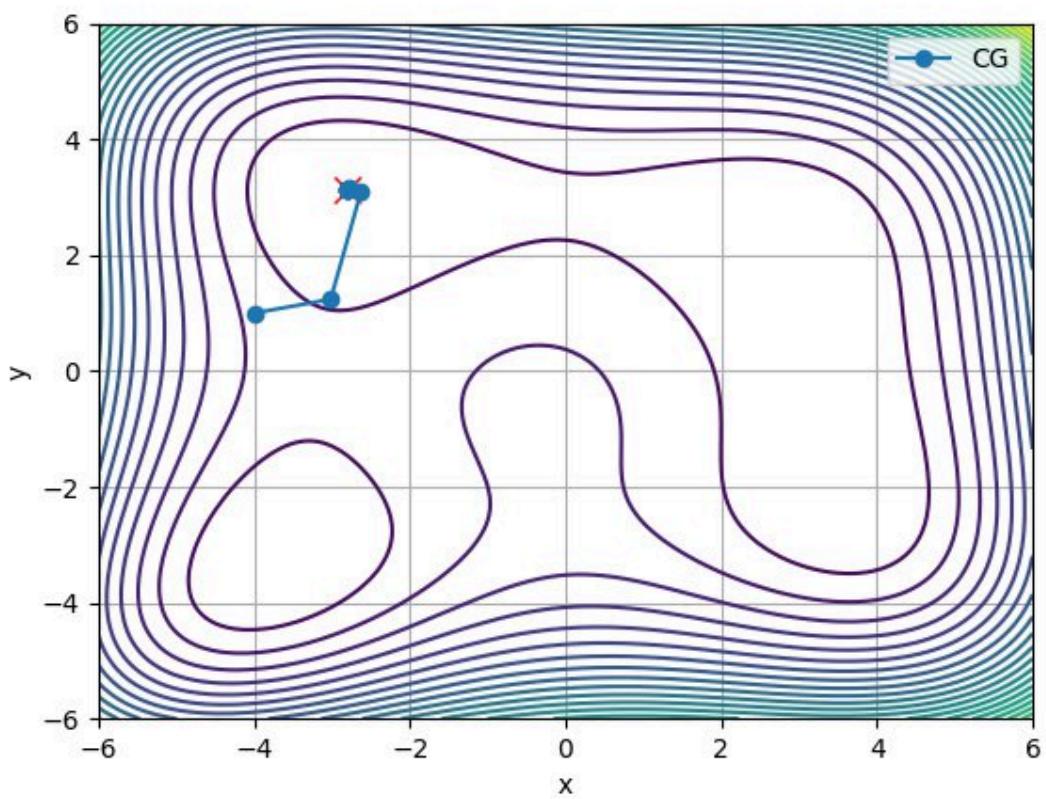
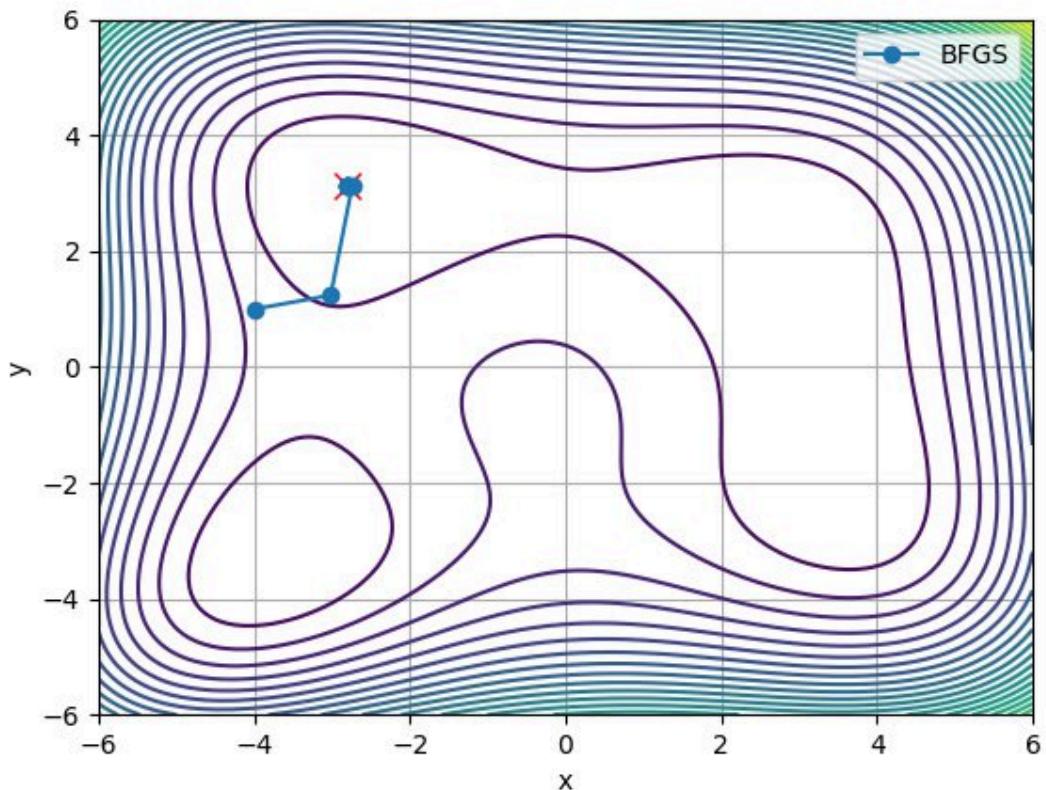
Пример 3-4: $(x^2 + y - 11)^2 + (x + y^2 - 7)^2$ (функция Химмельблау)

Стартовые позиции: (- 4, 1) и (4, 1)

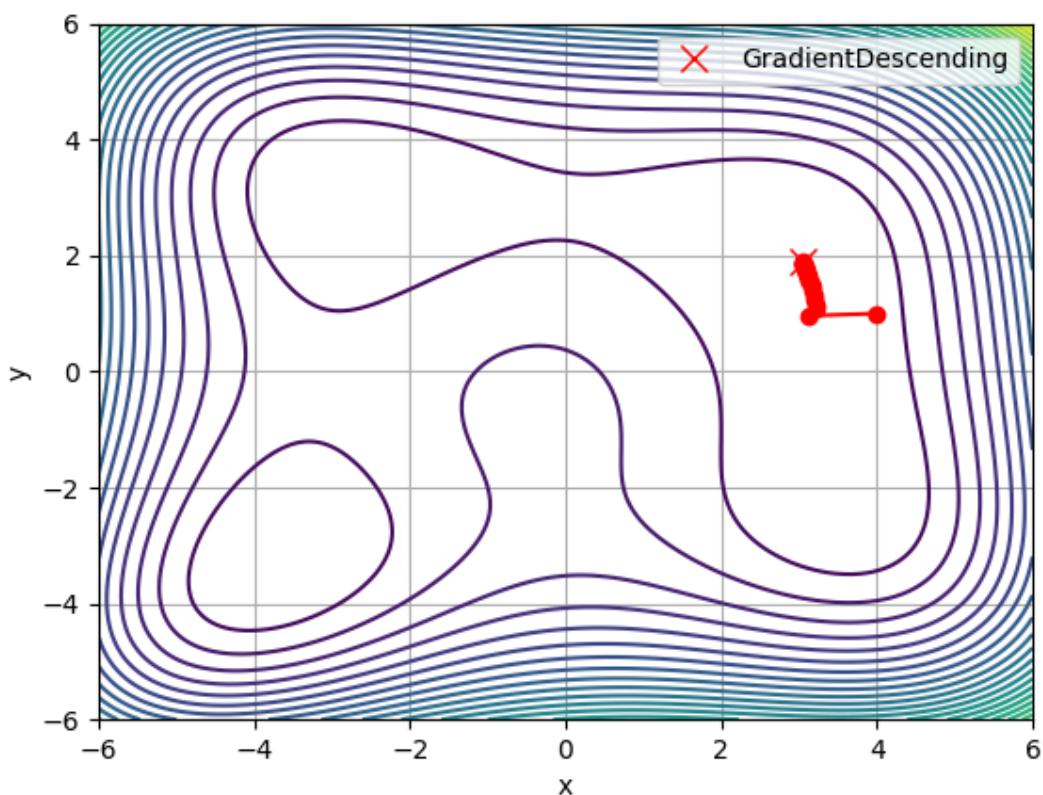
	Total iterations count:	Objective function calls:	Gradient computations:	Result Minimum:
GradientDescending: (fixed)	10	20	10	0.006512562 518942776
DichtGradientDescending	46	92	8	0.004986121 285955159
GradientGoldenSearch	53	77	8	0.000419525 2209331338 4
scipy.optimize('BFGS')	7	33	11	1.181630560 1505918e-14
scipy.optimize('CG')	8	54	18	1.449981844 5719755e-15

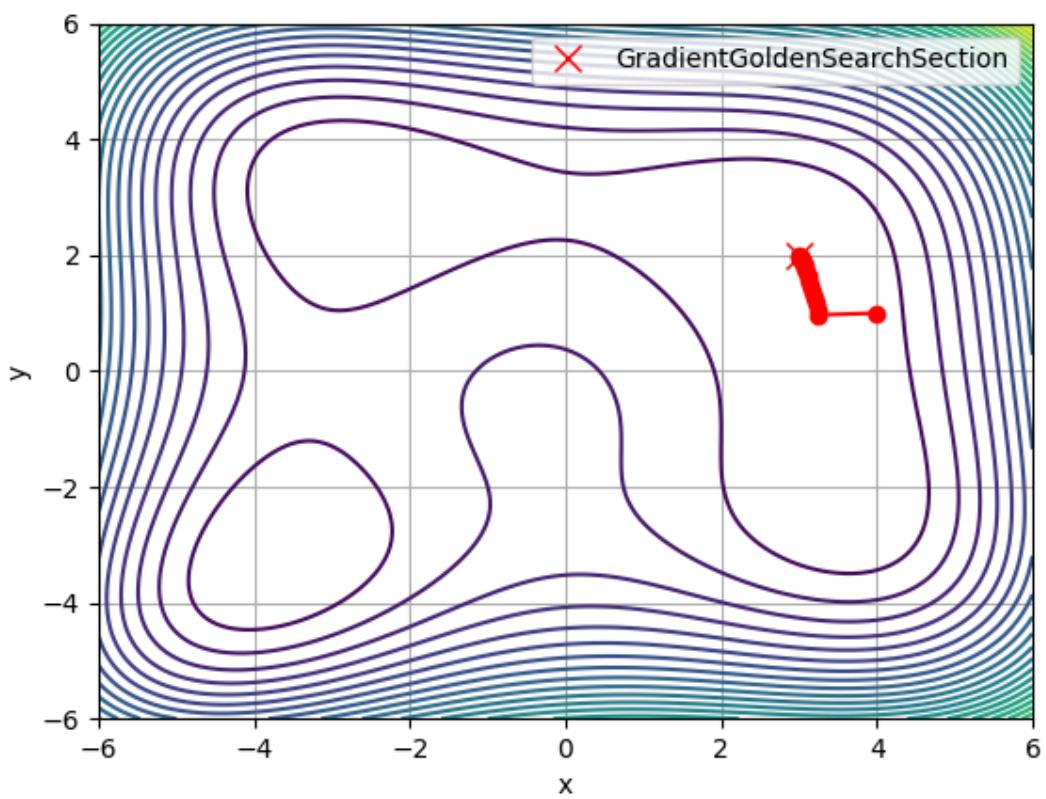
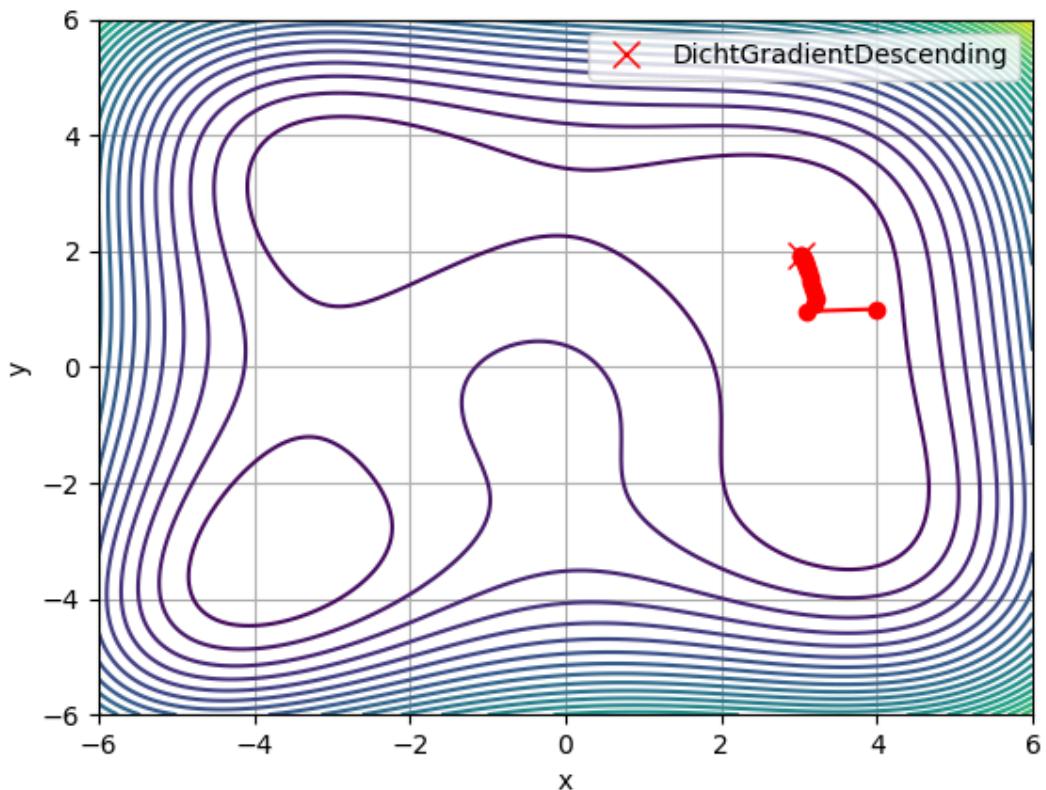


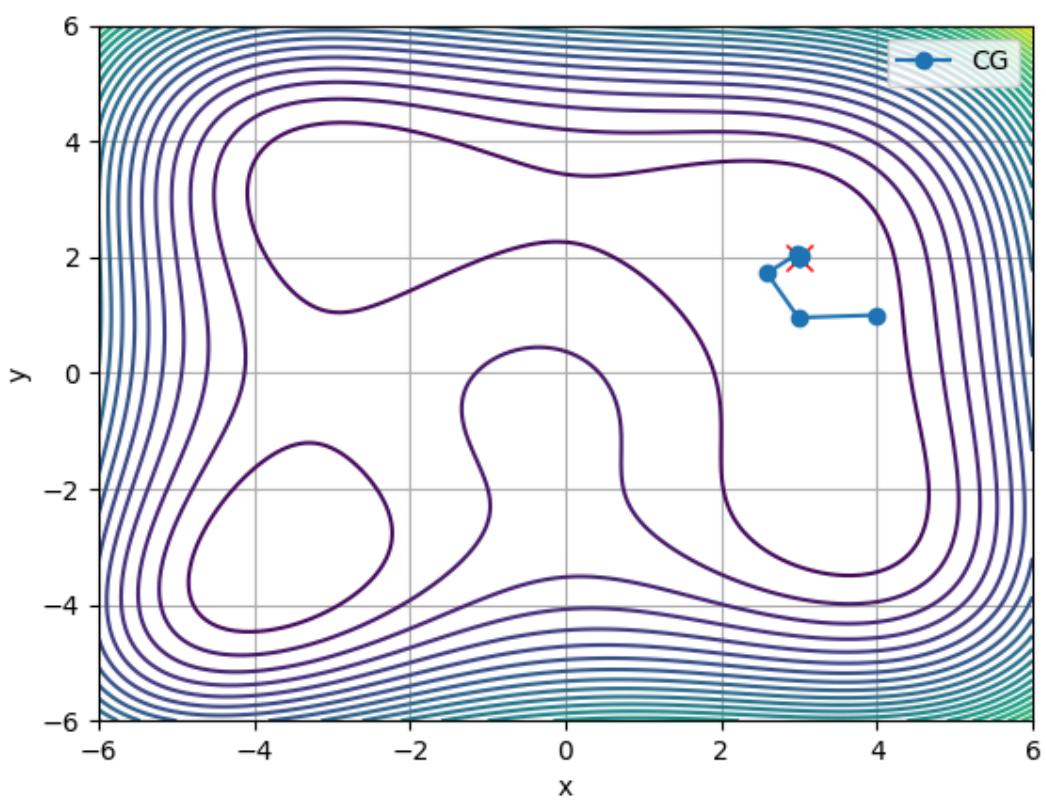
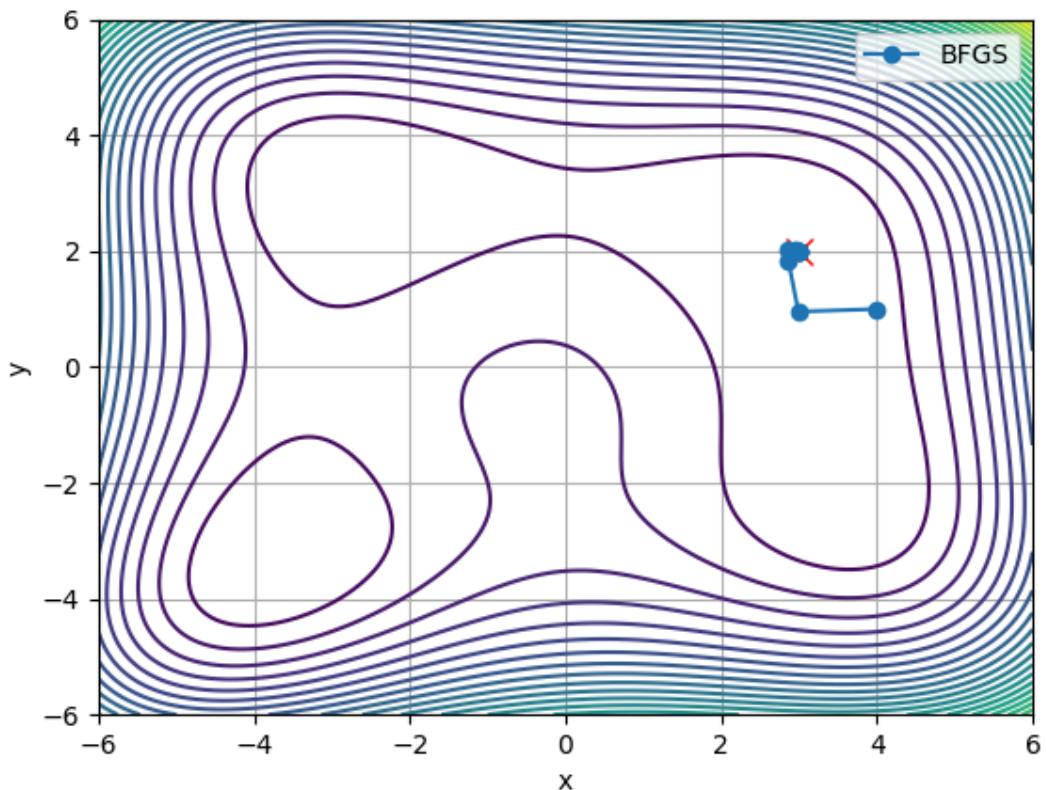




	Total iterations count:	Objective function calls:	Gradient computations:	Result Minimum:
GradientDescending: (fixed)	22	44	22	0.179886035 32126396
DichtGradientDescenting	75	150	17	0.058987269 38354686
GradientGoldenSearchSection	73	130	19	0.005051947 174465604
scipy.optimize('BFGS')	9	36	12	7.219170181 569629e-15
scipy.optimize('CG')	8	63	21	2.045634472 1687617e-14







Вывод

Анализируя результат наших методов и сравнивая их с библиотекой `scipy.optimize`, мы можем заметить, что наша реализация выполняет гораздо больше итераций, вызовов функций и подсчетов градиентов, что скорее всего является следствием не самой лучшей реализации, но результат наших методов очень похож на результаты минимумов с точностью до ϵ , что говорит о работоспособности реализованных методов. Также была приведена мультимодальная функция (примеры 3 и 4), в которой появляются локальные минимумы. Для борьбы с проблемой нахождения локального минимума мы использовали зашумление функции, но, возможно, из-за неправильного подобранных коэффициента зашумления алгоритм все равно иногда проваливается в локальные минимумы.