

# Лабораторная работа №1

Методы нулевого и первого порядка

Выполнili: [Акименко Иван, Глызина Кристина, Токарева Ульяна]

Группа : M3233

## Описание методов

---

### *Не библиотечные методы*

- Функции, на которых находится минимум

```
def f1(args):  
    return args[0] ** 2 + args[1] ** 2  
  
def f1_1(args):  
    return (args[0] + 2) ** 2 + args[1] ** 2  
  
def f1_2(args):  
    return args[0] ** 2 + (args[1] - 3) ** 2  
  
def f1_3(args):  
    return (args[0] - 2) ** 2 + (args[1] + 1) ** 2  
  
-2 <= B <= 2  
  
def f3(args):  
    return args[0] ** 2 + B * args[0] * args[1] + args[1] ** 2
```

- Мультимодальная функция с минимумом в точке (0; 0)

```
def multimodal_f(args):  
    return 20 + args[0] ** 2 + args[1] ** 2 - 10 * np.cos(2*np.pi * args[0]) - 10 *  
np.cos(2 * np.pi * args[1])
```

- Функция генерирует двумерную поверхность с несколькими локальными экстремумами и случайным шумом, величина которых регулируется параметрами, позволяя исследовать влияние шума и сложности поверхности на работу методов оптимизации.

```
def noisy_multimodal_f(args, noise_level=0.2, modes=3):  
    x, y = args[0], args[1]  
    value = 0  
    for m in range(1, modes + 1):  
        value += np.sin(m * x) * np.cos(m * y)  
    noise = np.random.normal(0, noise_level)  
    return value + noise
```

- Функция для получения численного значения градиента в переданной точке способом symmetric

```
def grad_f(f, x, y):  
    eps = np.finfo(float).eps  
    delta_x = np.sqrt(eps) * max(1.00, abs(x))  
    delta_y = np.sqrt(eps) * max(1.00, abs(y))  
    df_dx = (f(x + delta_x, y) - f(x - delta_x, y)) / (2 * delta_x)  
    df_dy = (f(x, y + delta_y) - f(x, y - delta_y)) / (2 * delta_y)  
  
    return np.array([df_dx, df_dy])
```

- Функция выполняет поиск оптимального шага по направлению спуска с использованием условий Голдштейна. Она вычисляет новое положение  $(x_k, y_k)$  вдоль направления антиградиента и оценивает линейные аппроксимации функции с параметрами  $(c1, c2)$ . Итеративно корректируя границы шага, функция возвращает координаты, обеспечивающие достаточное уменьшение значения функции.

```
def goldstein(f, x, y, grad, c1, c2, a0, iterations, log_file, counters):
    p = -grad
    a_l = 0.0
    a_r = a0
    direction = np.dot(grad, p)

    fxy = f([x, y])
    counters[0] += 1

    for _ in range(iterations):
        counters[2] += 1
        a = 0.5 * (a_r - a_l)
        x_k = x + a * p[0]
        y_k = y + a * p[1]
        log_file.write(str(x_k) + ' ' + str(y_k) + '\n')
        l_a_c1 = fxy + c1 * a * direction
        l_a_c2 = fxy + c2 * a * direction
        func_value = f([x_k, y_k])
        counters[0] += 1

        if func_value > l_a_c1:
            a_r = a
        elif func_value < l_a_c2:
            a_l = a
        else:
            break

    return x_k, y_k
```

- Функция реализует метод поиска шага по правилу Армijo для градиентного спуска. Она итеративно уменьшает шаг, пока не будет достигнуто достаточное снижение значения функции. По завершении функция возвращает обновлённые координаты, вычисленные с использованием найденного шага

```
def armijo_gradient_descent(f, x, y, grad, c1, a0, q, log_file, counters):
    p = -grad
    a = a0
    direction = np.dot(grad, p)
    fxy = f([x, y])
    counters[0] += 1

    while True:
        counters[2] += 1
        x_k = x + a * p[0]
        y_k = y + a * p[1]
        log_file.write(str(x_k) + ' ' + str(y_k) + '\n')

        func_value = f([x_k, y_k])
        counters[0] += 1
        l_a = fxy + c1 * a * direction

        if func_value <= l_a:
            break
        else:
            a *= q

    return x_k, y_k
```

- Функция реализует поиск оптимального шага вдоль направления антиградиента методом золотого сечения. Итеративно сужает интервал, сравнивая значения функции в двух точках, определённых золотыми коэффициентами, до достижения заданной точности. По завершении возвращает найденный оптимальный размер шага, при котором функция достигает минимального значения вдоль данного направления.

```
def golden_section(f, x, y, grad, l, r, counters, stop=np.finfo(float).eps):
    c_k_coeff = np.float64(0.382)
    d_k_coeff = np.float64(0.618)
    a_l = l + c_k_coeff * (r - l)
    a_r = l + d_k_coeff * (r - l)
    f_l_val = f([x - a_l * grad[0], y - a_l * grad[1]])
    f_r_val = f([x - a_r * grad[0], y - a_r * grad[1]])
    counters[0] += 2
    while (r - l) > stop:
        counters[2] += 1
        if f_l_val > f_r_val:
            l = a_l
            a_l = a_r
            f_l_val = f_r_val
            a_r = l + d_k_coeff * (r - l)
            f_r_val = f([x - a_r * grad[0], y - a_r * grad[1]])
            counters[0] += 1
        else:
            r = a_r
            a_r = a_l
            f_r_val = f_l_val
            a_l = l + c_k_coeff * (r - l)
            f_l_val = f([x - a_l * grad[0], y - a_l * grad[1]])
            counters[0] += 1
    return l, l + c_k_coeff * (r - l), r
```

- **Метод параболической интерполяции.** Используется после того, как методом золотого сечения был найден достаточно малый интервал, считая, что на этом интервале функция ведет себя как парабола. Решает систему из 3 уравнений, используя встроенные в numpy методы и возвращает значение найденного коэффициента, при условии, что его значение не выходит за границы параболы, иначе возвращает значение, при котором функция принимает минимальное значение.

```
def parabolic(f, x, y, grad, alpha_vals):
    def phi(alpha):
        return f([x - alpha * grad[0], y - alpha * grad[1]])

    alphas = np.array(alpha_vals)
    phis = np.array([phi(alpha) for alpha in alphas])

    A = np.vstack([alphas ** 2, alphas, np.ones_like(alphas)]).T
    a, b, c = np.linalg.solve(A, phis)

    if a <= 0:
        res = alphas[np.argmin(phis)]
    else:
        res = -b / (2 * a)
        if res < min(alphas) or res > max(alphas):
            res = alphas[np.argmin(phis)]

    return res
```

- Функция является вспомогательной, используется в функции дихотомии для получения коэффициентов в заданном интервале, которые далее используются в методе дихотомии.

```
def get_points_for_dihotomiya(l, r):
    c_k = l + ((r - l) / 2)
    d_k = l + ((c_k - l) / 2)
    t_k = c_k + ((r - c_k) / 2)
    return c_k, d_k, t_k
```

- Функция dihotomiya ищет минимум функции  $f$  вдоль направления антиградиента, используя метод дихотомии на отрезке  $[l, r]$ . На каждом шаге вычисляются три вспомогательные точки ( $c_k, d_k, t_k$ ) с помощью функции `get_points_for_dihotomiya`. На основе сравнения значений функции  $f$  в этих точках обновляются границы интервала поиска. Итерации продолжаются до сужения интервала до заданного порога точности, после чего возвращается оптимальный размер шага вдоль направления антиградиента, соответствующий минимальному значению функции на исследуемом направлении.

```
def dihotomiya(f, x, y, grad, l, r, counters, stop=np.finfo(float).eps):
    c_k, d_k, t_k = get_points_for_dihotomiya(l, r)
    while (r - l) > stop:
        counters[2] += 1
        f_c_k = f([x - c_k * grad[0], y - c_k * grad[1]])
        counters[0] += 1
        if f_c_k > f([x - d_k * grad[0], y - d_k * grad[1]]):
            counters[0] += 1
            r = c_k
            c_k, d_k, t_k = get_points_for_dihotomiya(l, r)
        elif f_c_k > f([x - t_k * grad[0], y - t_k * grad[1]]):
            counters[0] += 2
            l = c_k
            c_k, d_k, t_k = get_points_for_dihotomiya(l, r)
        else:
            counters[0] += 2
            l = d_k
            r = t_k
            c_k, d_k, t_k = get_points_for_dihotomiya(l, r)
    return c_k
```

- Функция для вычисления шага с использованием встроенного метода `scipy.optimize.line_search`, которая использует в своей реализации для вычисления шага правила Wolfe. Найденные минимумы с данным вычислением шага сравниваются с реализованными нами методами вычисления шага по правилам Armijo и Goldstein.

```
def l_search(f, x, y, grad, a_0, c1, c2):
    alpha = line_search(
        f=f,
        myfprime=lambda args: grad_f(f, args[0], args[1]),
        xk=np.array([x, y]),
        pk=-grad,
        amax=a_0,
        c1=c1,
        c2=c2,
    )

    if alpha[0] is None:
        return -1, -1

    return x - alpha[0] * grad[0], y - alpha[0] * grad[1]
```

- Функция для вычисления шага с использованием встроенного метода `scipy.optimize.minimize_scalar`, которая использует в своей реализации методы одномерного поиска, в зависимости от параметра `method`: “golden” - метод золотого сечения и “brent” - метод параболической интерполяции. Найденные минимумы с данным вычислением шага сравниваются с реализованными нами методами градиентного спуска на основе одномерного поиска “golden\_section” и “parabolic” соответственно.

```
def s_minimize(f, x, y, grad, method):
    alpha = minimize_scalar(
        lambda a: f([x - a * grad[0], y - a * grad[1]]),
        method=method,
    )

    return x - alpha.x * grad[0], y - alpha.x * grad[1]
```

- Вспомогательная функция, которая в зависимости от параметра method выбирает один из алгоритмов обновления: стандартное  $x - h \cdot \nabla f$ , адаптивное с уменьшением шага  $h/\sqrt{k+1}$  или методы поиска оптимального шага (Армихо, Гольдштейна, золотого сечения, дихотомии и параболической интерполяции). Результатом работы функции являются новые координаты, которые позволяют итеративно приближаться к минимуму  $f(x,y)$

```

def make_step(
    f, x, y, h,
    l_s_x,
    l_s_y,
    grad,
    grad_l_s,
    method,
    iteration,
    log_file,
    c1, c2,
    a_0,
    stop,
    counters
):
    match method:
        case "default":
            return x - h * grad[0], y - h * grad[1], -1, -1
        case "decreasing_lr":
            return x - (h / np.sqrt((iteration + 1))) * grad[0], y - (h /
np.sqrt((iteration + 1))) * grad[1], -1, -1
        case "Armijo":
            a_x, a_y = armijo_gradient_descent(f, x, y, grad, c1, a_0, 0.5, log_file,
counters)
            ls_x, ls_y = l_search(f, l_s_x, l_s_y, grad_l_s, a_0, c1, 0.9)
            return a_x, a_y, ls_x, ls_y
        case "Goldstein":
            g_x, g_y = goldstein(f, x, y, grad, c1, c2, a_0, 100, log_file, counters)
            ls_x, ls_y = l_search(f, l_s_x, l_s_y, grad_l_s, a_0, c1, c2)
            return g_x, g_y, ls_x, ls_y
        case "golden_section":
            _, a2, _ = golden_section(f, x, y, grad, 0.0, a_0, counters, stop)
            sm_x, sm_y = s_minimize(f, l_s_x, l_s_y, grad_l_s, "golden")

            return x - a2 * grad[0], y - a2 * grad[1], sm_x, sm_y
        case "dihotomiya":
            alpha = dihotomiya(f, x, y, grad, 0.0, a_0, counters, stop)
            return x - alpha * grad[0], y - alpha * grad[1], -1, -1
        case "parabolic":
            a1, a2, a3 = golden_section(f, x, y, grad, 0.0, a_0, counters, stop)
            alpha = parabolic(f, x, y, grad, [a1, a2, a3])
            counters[0] += 3
            counters[2] += 1
            s_x, s_y = s_minimize(f, l_s_x, l_s_y, grad_l_s, "brent")
            return x - alpha * grad[0], y - alpha * grad[1], s_x, s_y
    
```

- Функция `gradient_descent` реализует алгоритм градиентного спуска для оптимизации заданной функции  $f$ , начиная с начальных координат  $x_0$  и  $y_0$  с указанным шагом  $h$ . На каждой итерации вычисляется градиент функции с помощью `grad_f`, и если его норма становится меньше порога `stop`, итерационный процесс прерывается. Иначе, обновляет значения переменных с помощью `make_step` и продолжает процесс до достижения максимального количества итераций или условия останова.

```

def gradient_descent(
    f, x0, y0,
    method="default",
    h=0.01,
    iterations=2000,
    stop=np.finfo(float).eps,
    c1=0.3, c2=0.7,
    a_0=2.0
):
    x, y = x0, y0
    l_s_x, l_s_y = x0, y0
    counters = [0, 0, 0]
    with open(f.__name__ + "_" + method + ".txt", "w") as log_file:
        for i in range(iterations):
            counters[2] += 1
            log_file.write(f"{x} {y}" + "\n")
            grad = grad_f(f, x, y)
            grad_l_s = grad_f(f, l_s_x, l_s_y)
            counters[1] += 1
            counters[0] += 4
            if np.linalg.norm(grad) < stop:
                break
            x, y, l_s_x, l_s_y = make_step(
                f, x, y, h,
                l_s_x, l_s_y,
                grad,
                grad_l_s,
                method, i, log_file, c1, c2, a_0, stop, counters,
            )
    return [x, y, l_s_x, l_s_y, counters]

```

---

## Библиотечные методы

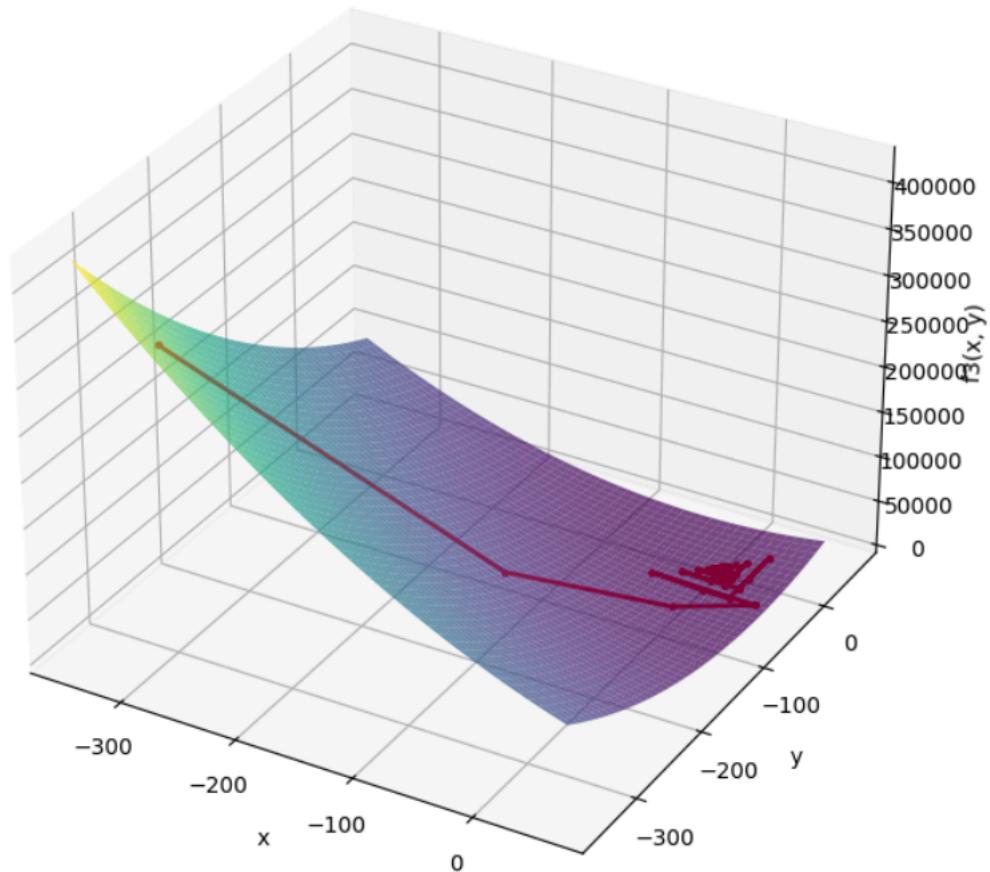
Используется библиотека “*numpy*”

- `cos()` - вычисление cosinus
- `finfo()` - информация о числовых характеристиках типа с плавающей запятой,мин и макс значение, eps и точность представления
- `sqrt()` - нахождение значения корня
- `array()` - преобразование последовательности данных в объект массива
- `dot()` - вычисление скалярного произведения векторов/матричное произведение матриц
- `float()` - число с плавающей запятой двойной точности(64бит)
- `vstack()` - формирует матрицу
- `linalg.solve()` - решает систему уравнений в матричном виде
- `argmin()` - возвращает индекс минимального элемента

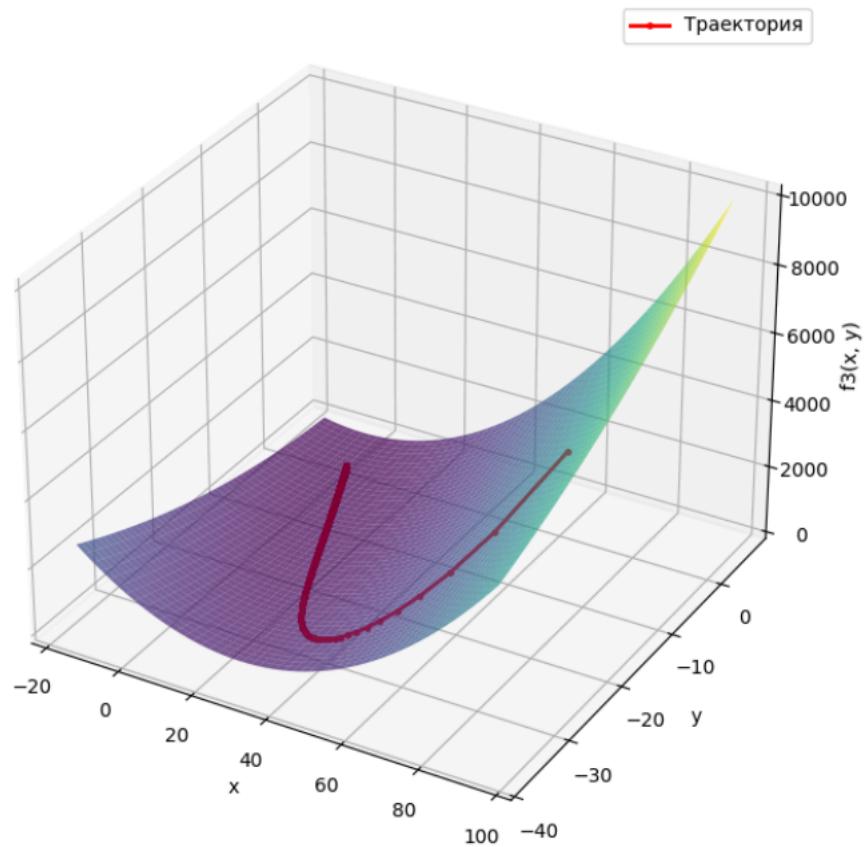
## Графики

3D график  $f_3$ \_Armijo

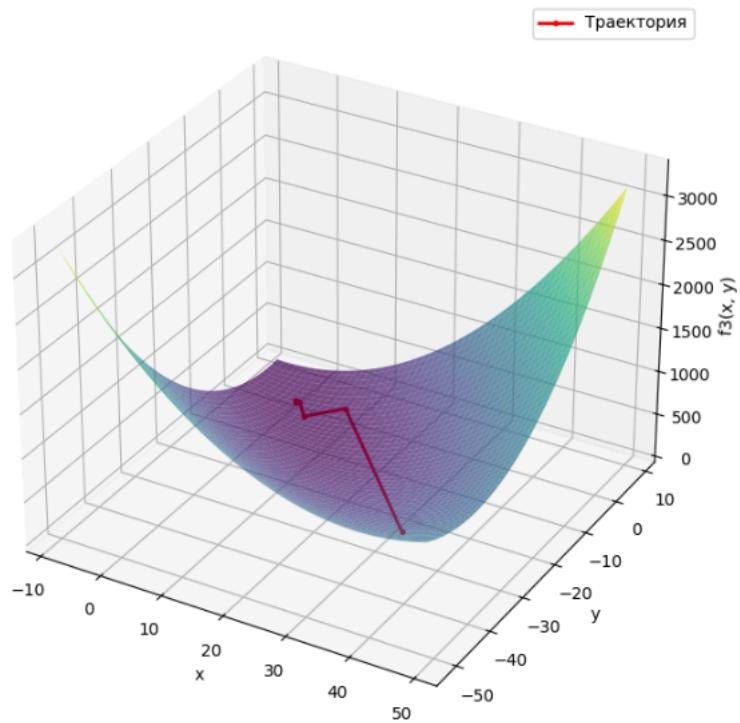
 Траектория



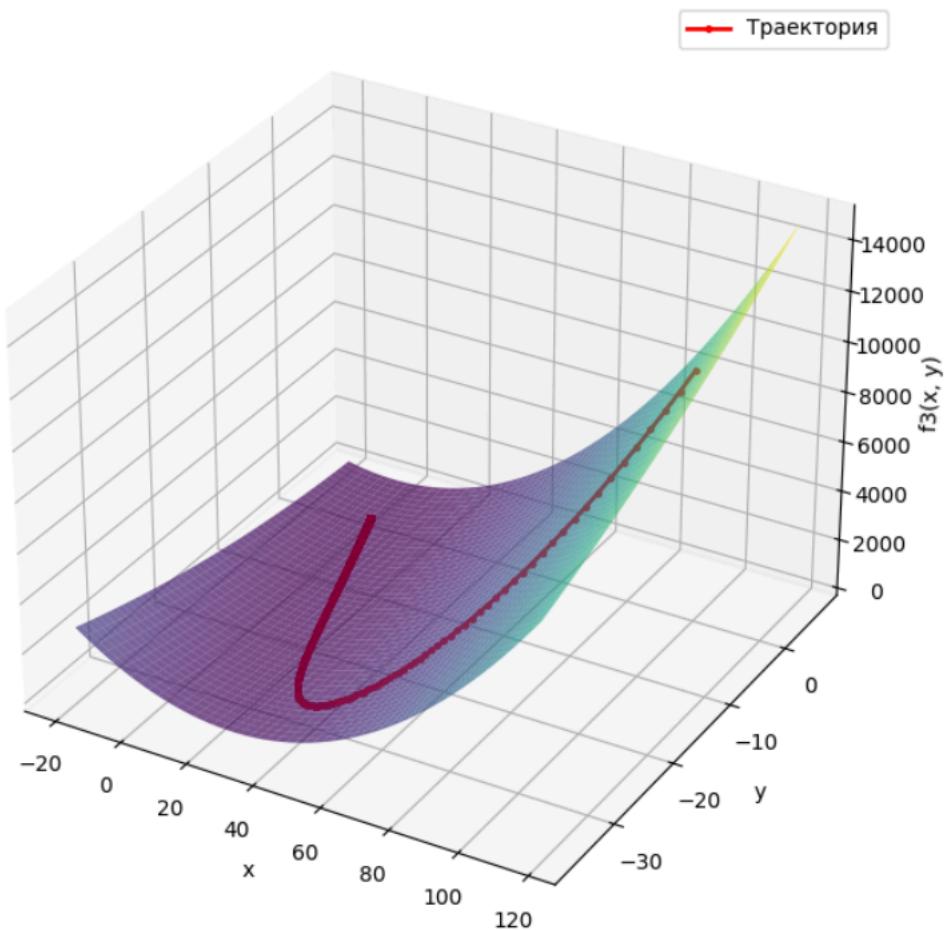
3D график f3\_decreasing\_lr



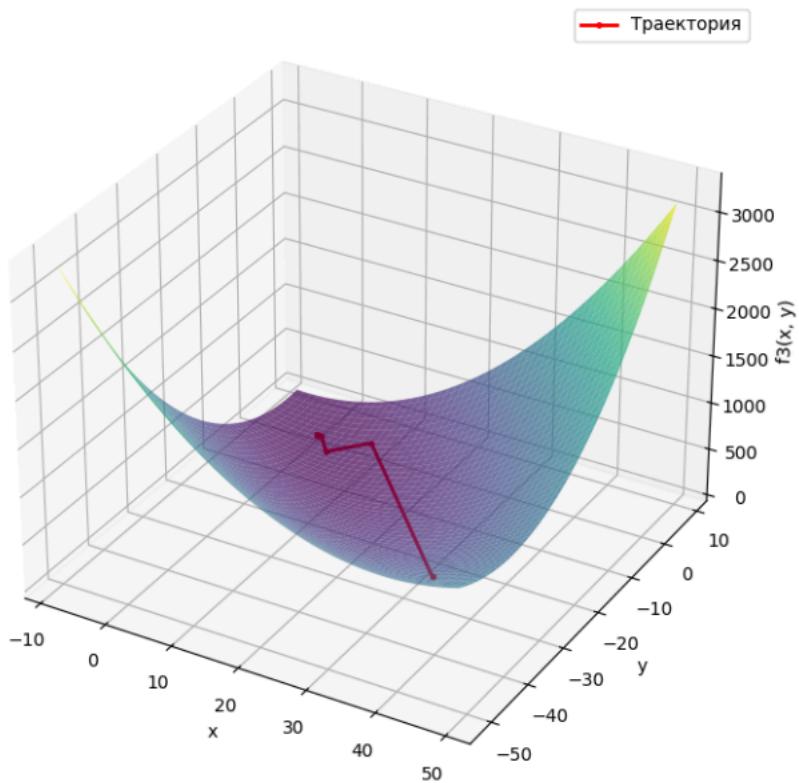
3D график f3\_golden\_section



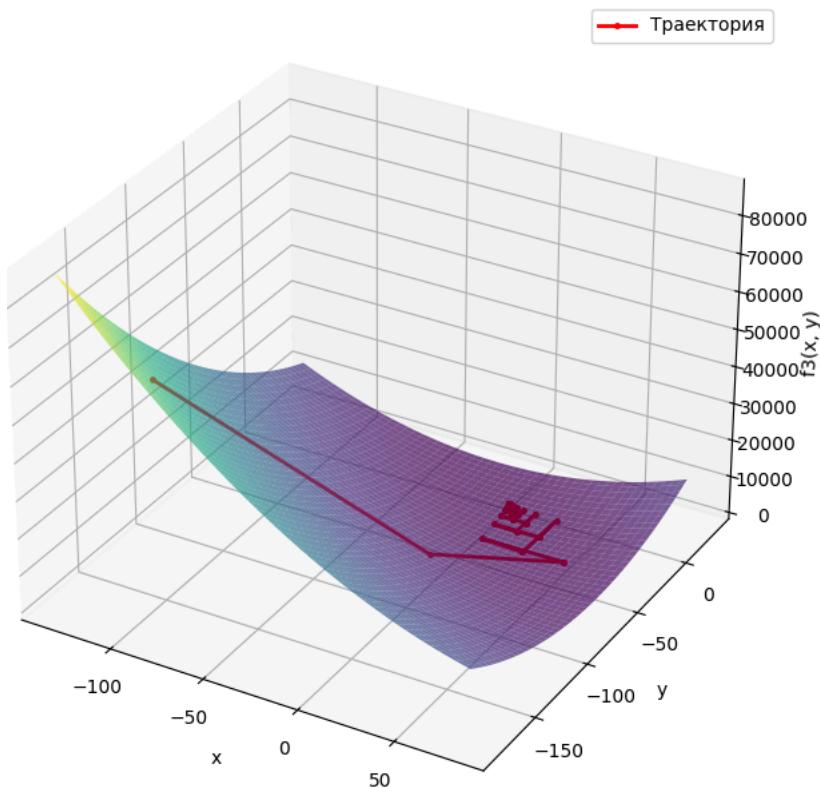
3D график f3\_default



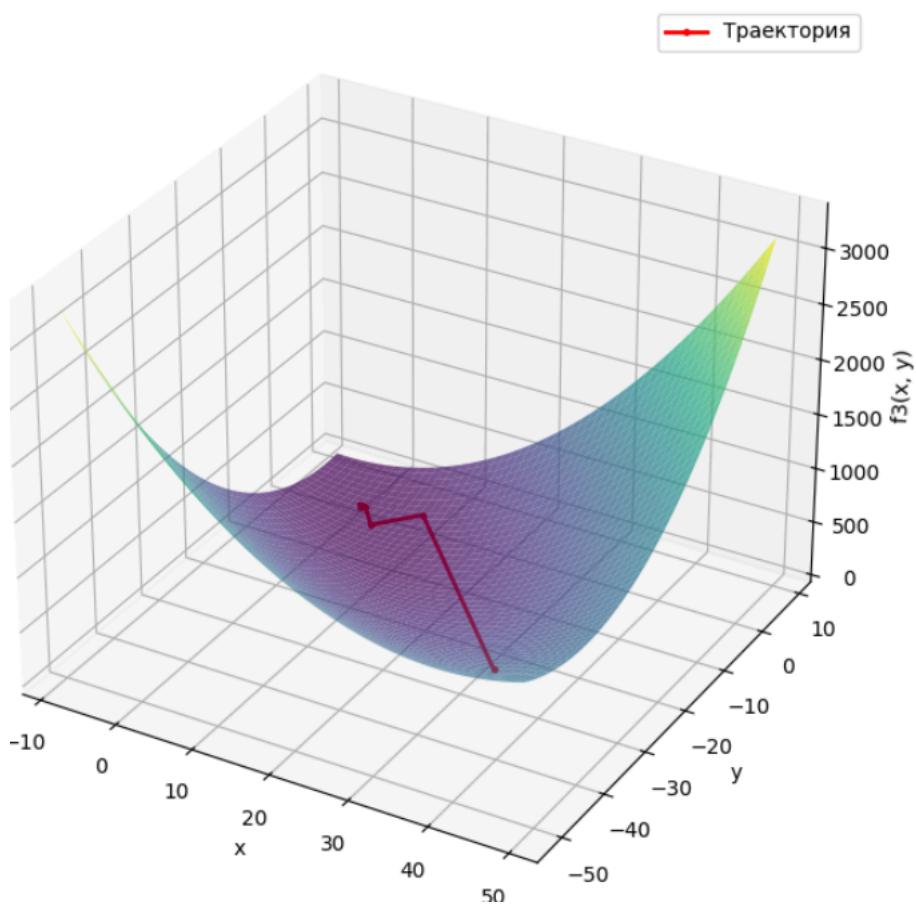
3D график f3\_dihotomiya



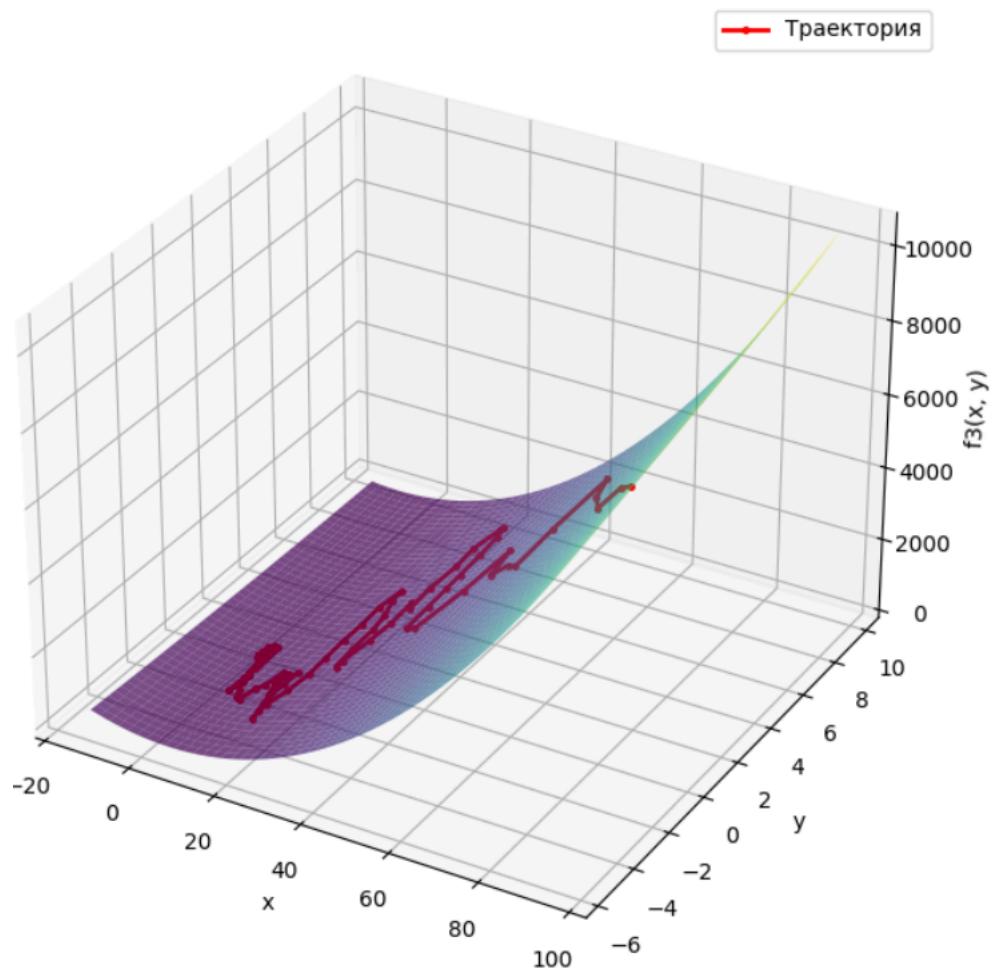
3D график  $f3_{\_Goldstein}$



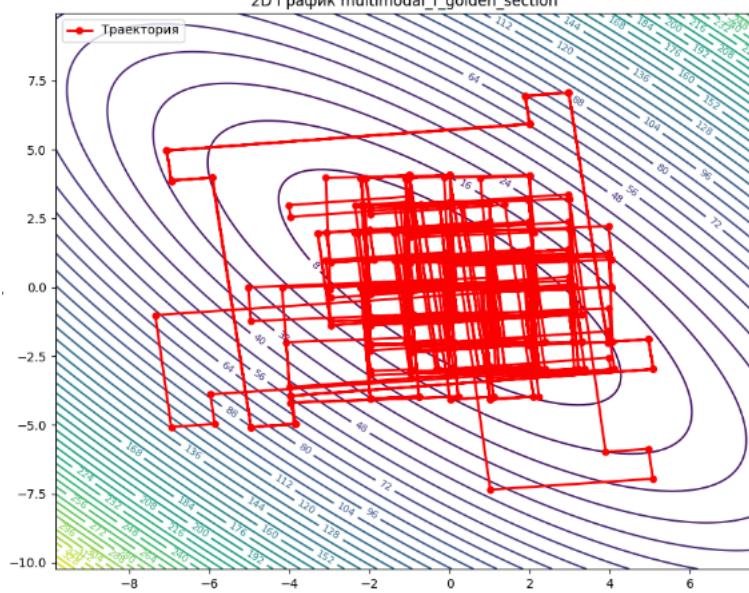
3D график  $f3_{\_parabolic}$



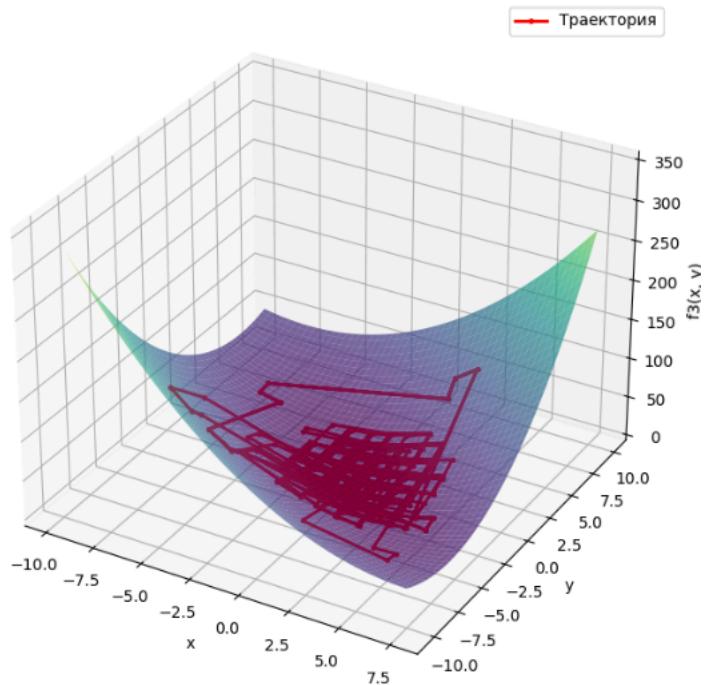
3D график multimodal\_f\_decreasing\_lr



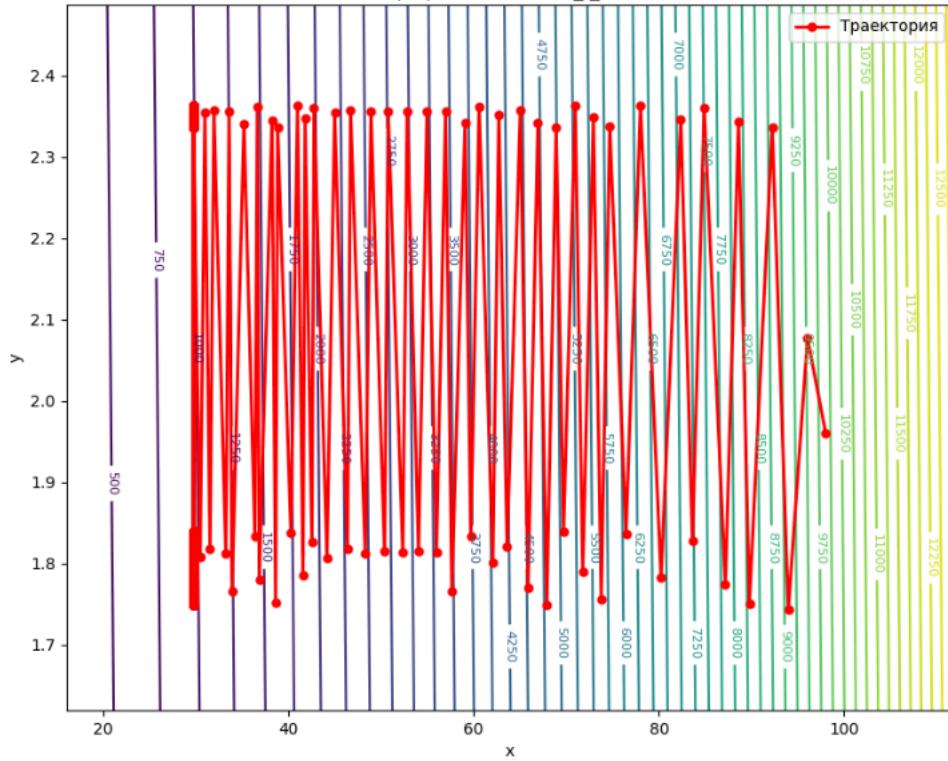
2D График multimodal\_f\_golden\_section



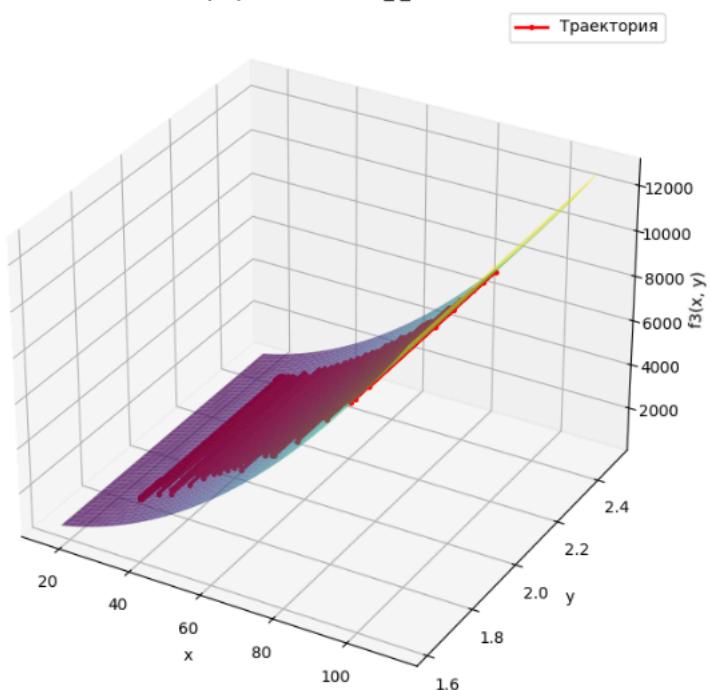
3D график multimodal\_f\_golden\_section



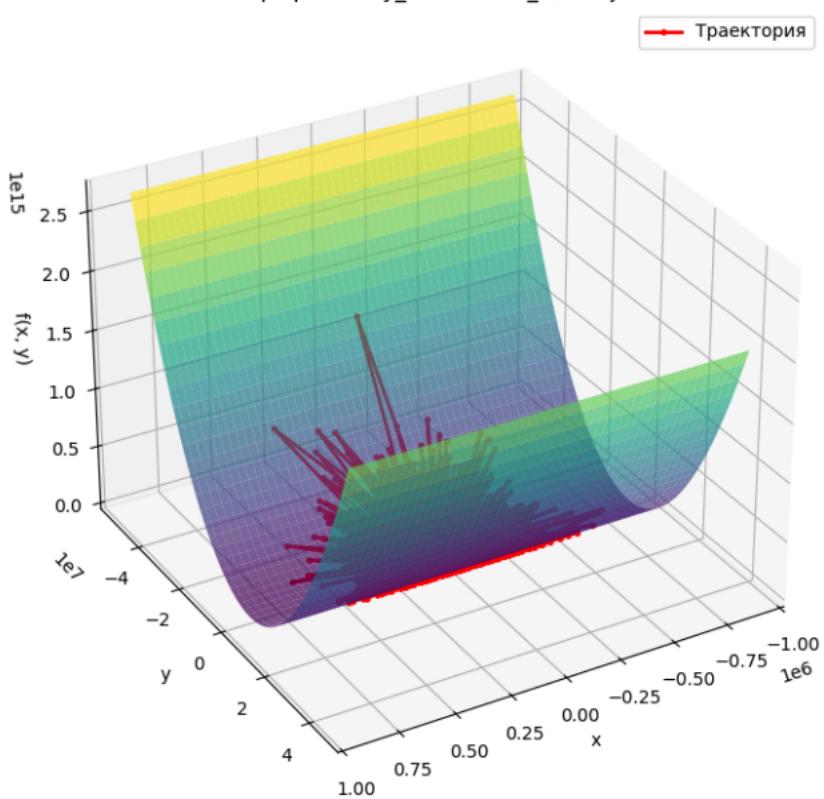
2D График multimodal\_f\_default

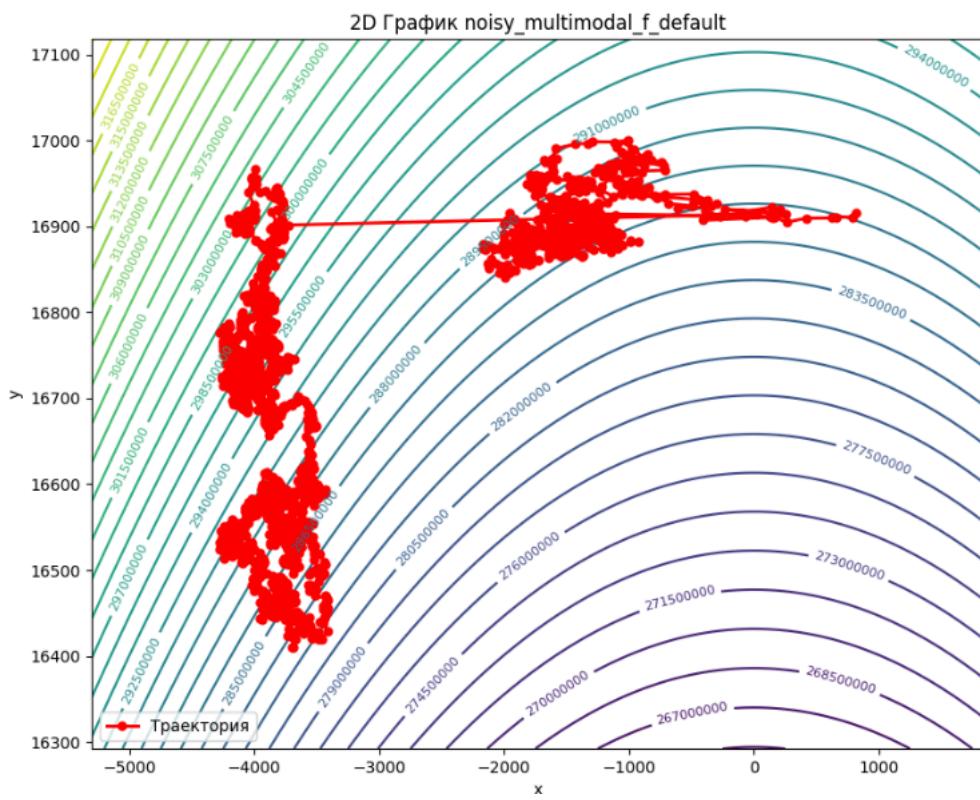
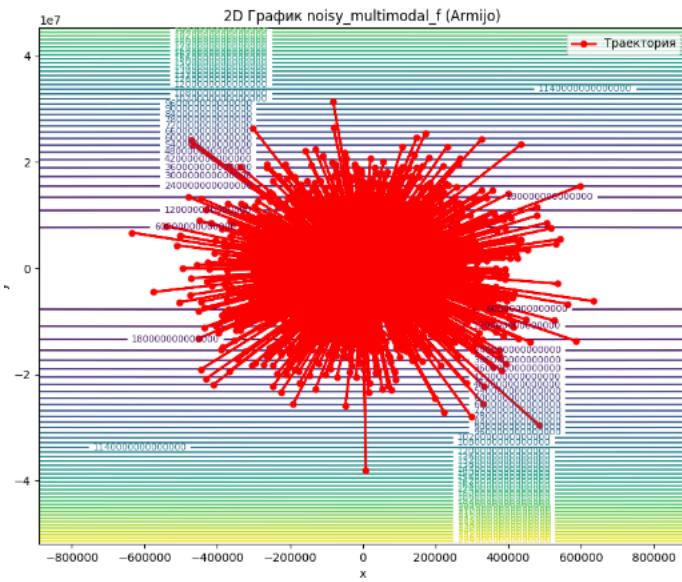


3D график multimodal\_f\_default

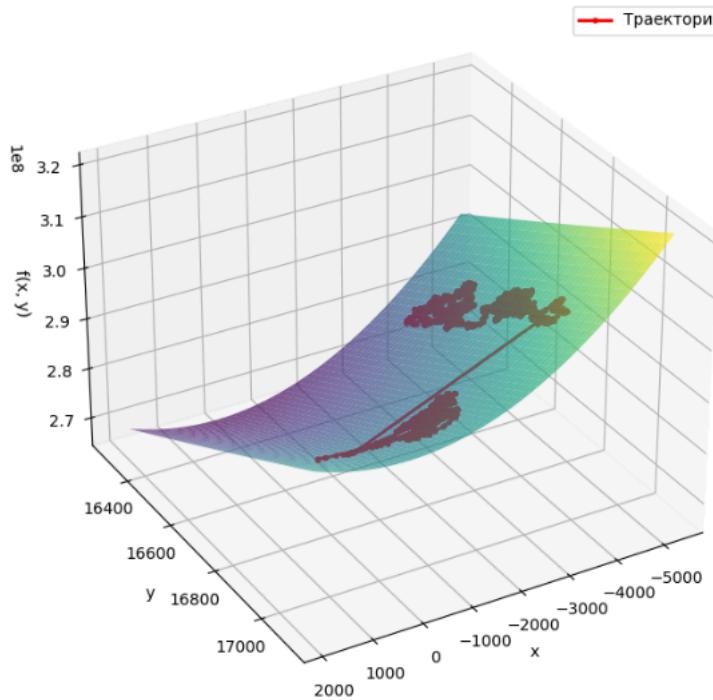


3D график noisy\_multimodal\_f (Armijo)

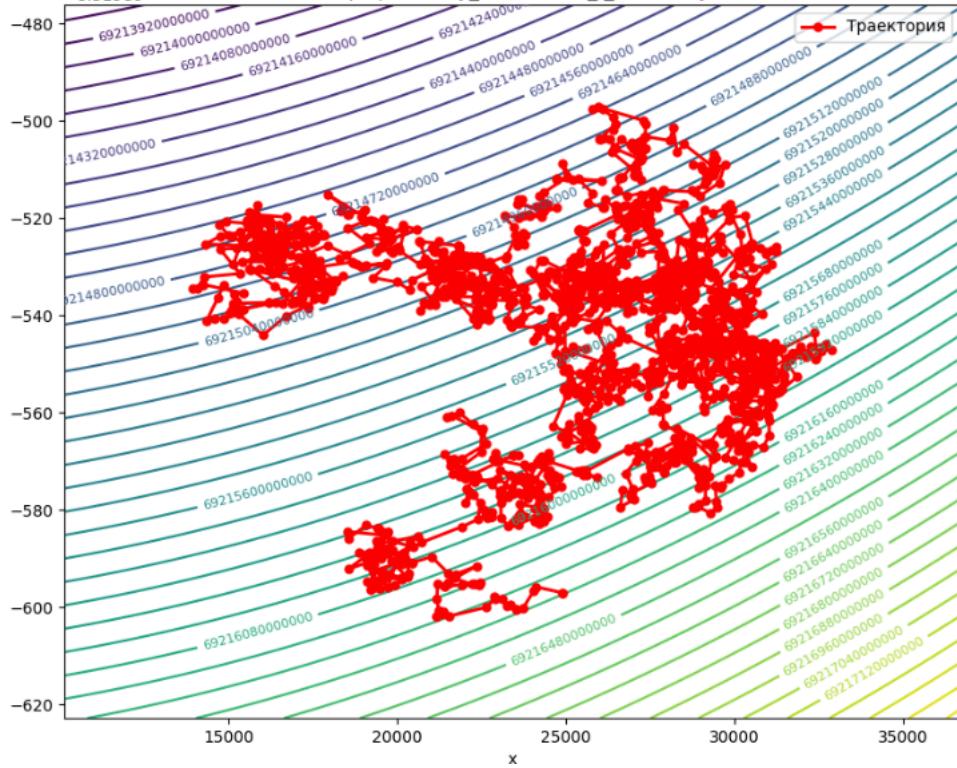




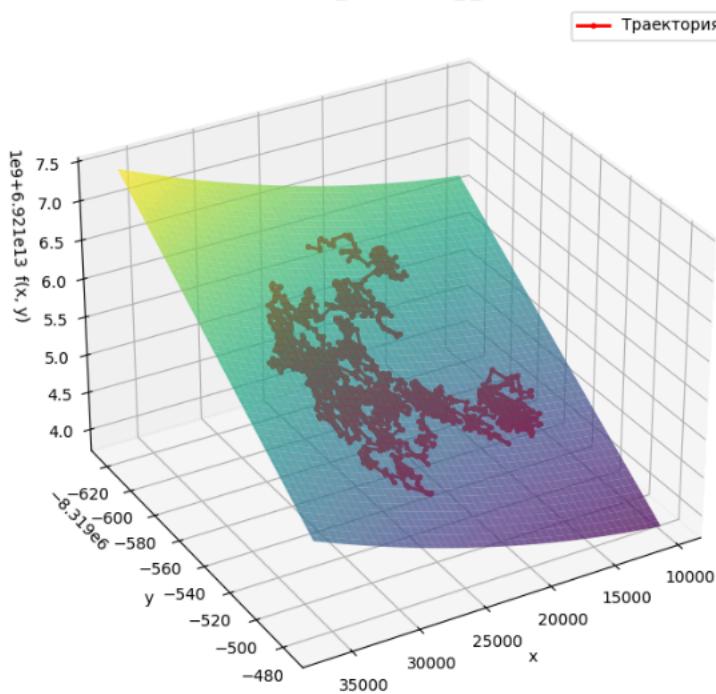
3D график noisy\_multimodal\_f\_default



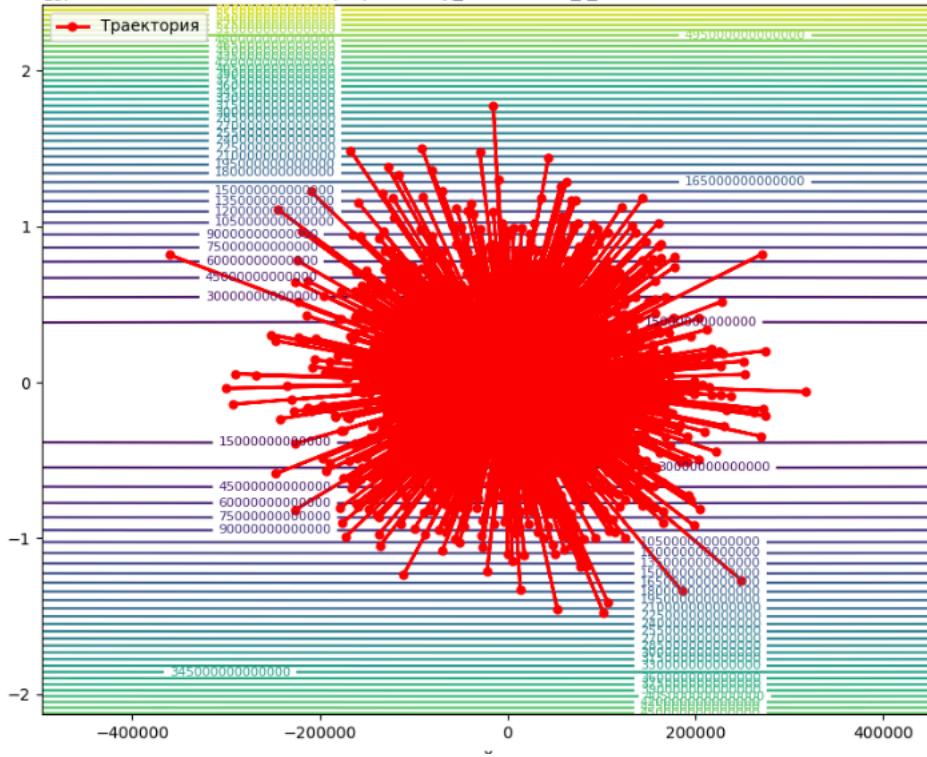
2D График noisy\_multimodal\_f\_dihotomiya



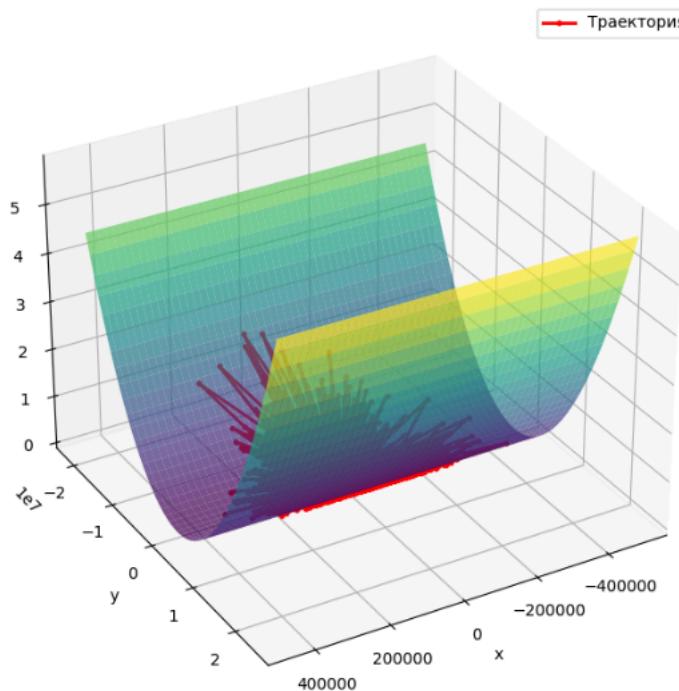
3D график noisy\_multimodal\_f\_dihotomiya



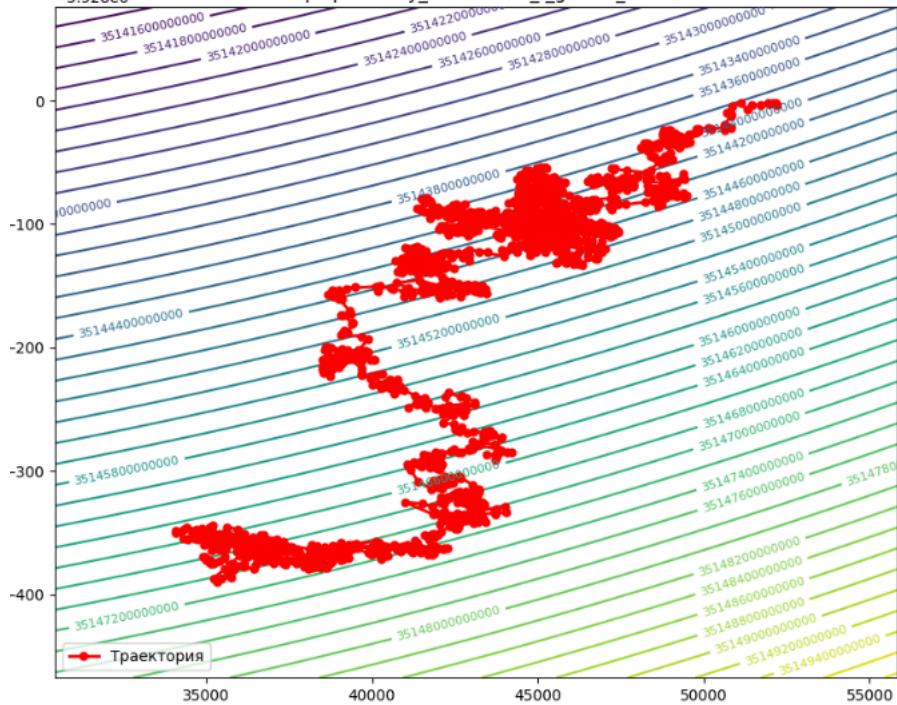
2D График noisy\_multimodal\_f\_Goldstein



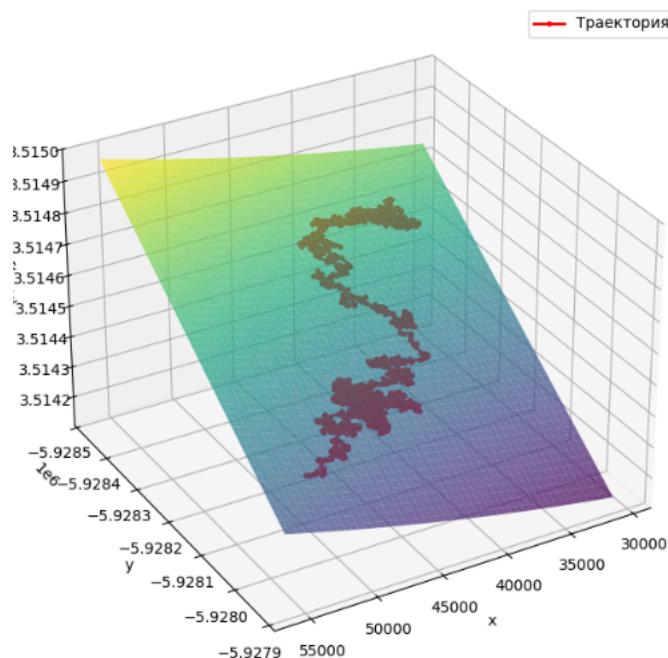
3D график noisy\_multimodal\_f\_Goldstein



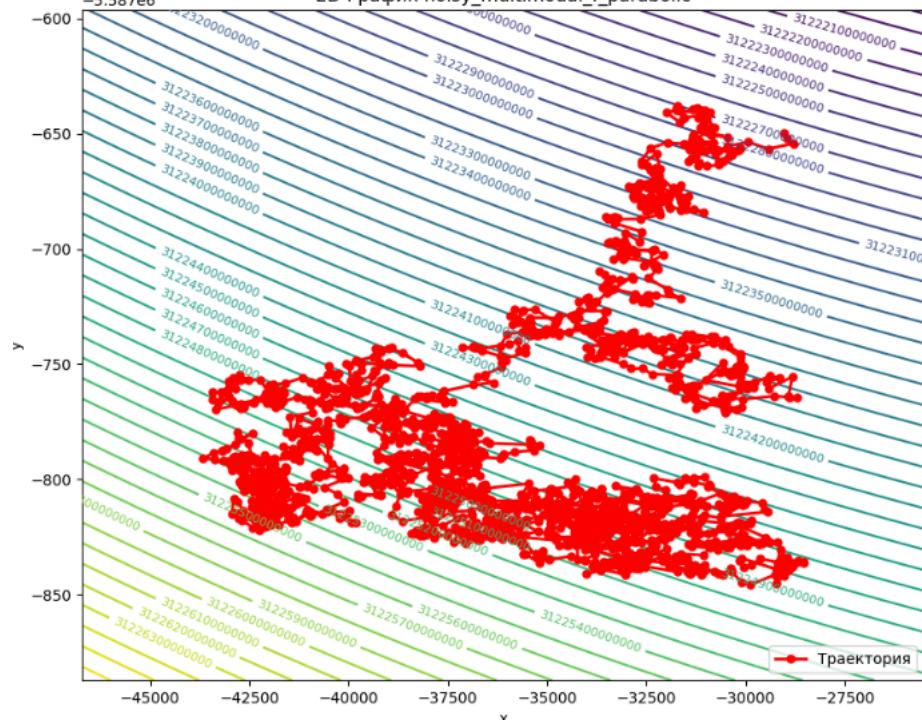
2D График noisy\_multimodal\_f\_golden\_section



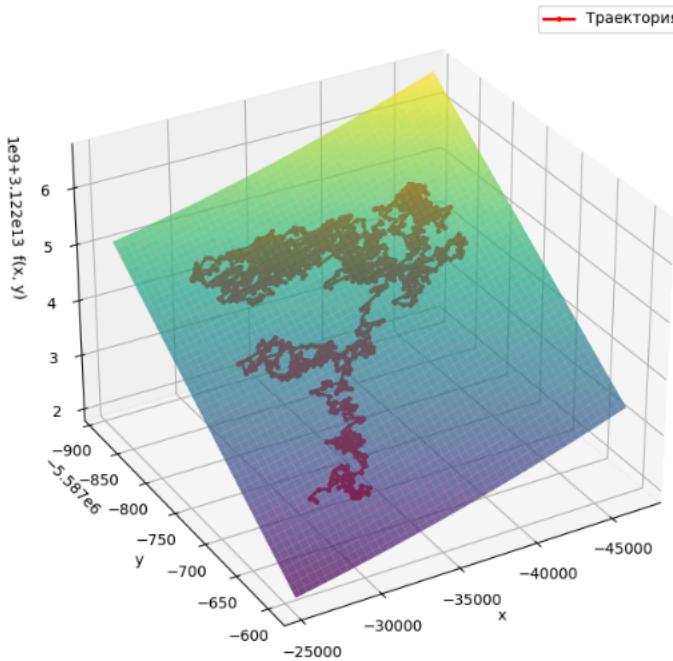
3D график noisy\_multimodal\_f\_golden\_section



2D График noisy\_multimodal\_f\_parabolic



3D график noisy\_multimodal\_f\_parabolic



## Описание результатов

$$\mathbf{f1: f1(x,y)=x^2 + y^2 \text{ (минимум в (0,0))}}$$

Параметры: params = { "x0": 100, "y0": 2, "stop": 1e-6, "h": 0.1, "iterations": 2000, "decreasing\_lr\_iterations: 50000" }

Метод	Число итераций	Вызовы функции	Вызовы градиента	$(x, y)$	Результат SciPy $(x, y)$
Default (GD)	948	3792	948	$(4.910 \times 10^{-7}, 9.820 \times 10^{-9})$	—
Decreasing LR	10728	42912	10728	$(1.110 \times 10^{-16}, 2.220 \times 10^{-18})$	—
Armijo	5	12	2	(0.0, 0.0)	(0.0, 0.0)
Goldstein	4	11	2	(0.0, 0.0)	(0.0, 0.0)
Golden Section	69	82	3	$(5.0945 \times 10^{-12}, 1.0189 \times 10^{-13})$	$(6.8789 \times 10^{-18}, 1.3758 \times 10^{-19})$
Dihotomiya	23	70	2	(0.0, 0.0)	—
Parabolic (Brent)	36	46	2	$(1.0345 \times 10^{-8}, 2.0689 \times 10^{-10})$	$(4.2633 \times 10^{-14}, 8.8818 \times 10^{-16})$

Параметры: params = { "x0": 75, "y0": 3, "stop": 1e-7, "h": 0.05, "iterations": 2000, "decreasing\_lr\_iterations: 30000" }

Метод	Количество итераций	Количество вызовов функции	Количество вычислений градиента	( $x, y$ )	Результат SciPy ( $x, y$ )
Default (GD)	1047	4188	1047	$(4.98 \times 10^{-8}, 1.99 \times 10^{-9})$	—
Decreasing LR	30000	120000	30000	$(7.41 \times 10^{-14}, 2.96 \times 10^{-15})$	—
Armijo	5	12	2	(0.0, 0.0)	(0.0, 0.0)
Goldstein	4	11	2	(0.0, 0.0)	(0.0, 0.0)
Golden Sect.	73	86	3	$(5.11 \times 10^{-12}, 2.05 \times 10^{-13})$	$(5.16 \times 10^{-18}, 2.06 \times 10^{-19})$
Dihotomiya	27	82	2	(0.0, 0.0)	—
Parabolic	75	92	3	$(2.98 \times 10^{-12}, 1.19 \times 10^{-13})$	$(-3.16 \times 10^{-30}, -1.97 \times 10^{-31})$

### f1\_1: $f1_1(x,y) = (x + 2)^2 + y^2$ (минимум в (-2,0))

Параметры: params = { "x0": 100, "y0": 2, "stop": 1e-6, "h": 0.1, "iterations": 2000, "decreasing\_lr\_iterations: 50000" }

Метод	Количество итераций	Количество вызовов функции	Количество вычислений градиента	( $x, y$ )	Результат SciPy ( $x, y$ )
Default (GD)	949	3796	949	$(-1.99999951, 9.62 \times 10^{-9})$	—
Decr. LR	50000	200000	50000	$(-2.0, 8.27 \times 10^{-25})$	—
Armijo	5	12	2	(-2.0, 0.0)	(-2.0, 0.0)
Goldstein	4	11	2	(-2.0, 0.0)	(-2.0, 0.0)
Golden Sect.	69	82	3	$(-2.0, \sim 10^{-13})$	$(-2.0, \sim 10^{-19})$
Dihotomiya	23	70	2	(-2.0, 0.0)	—
Parabolic	36	46	2	$(-2.0, 2.07 \times 10^{-10})$	(-2.0, 0.0)

Параметры: params = { "x0": 75, "y0": 3, "stop": 1e-7, "h": 0.05, "iterations": 2000, "decreasing\_lr\_iterations": 30000 }

Метод	Количество итераций	Количество вызовов функции	Количество вычислений градиента	( $x, y$ )	Результат SciPy ( $x, y$ )
Default (GD)	1049	4196	1049	( $-1.9999999509, 1.91 \times 10^{-9}$ )	—
Decreasing LR	30000	120000	30000	( $-2.0, 2.96 \times 10^{-15}$ )	—
Armijo	5	12	2	( $-2.0, 0.0$ )	( $-2.0, 0.0$ )
Goldstein	4	11	2	( $-2.0, 0.0$ )	( $-2.0, 0.0$ )
Golden Sect.	73	86	3	( $-2.0, 2.05 \times 10^{-13}$ )	( $-2.0, -3.88 \times 10^{-18}$ )
Dihotomiya	27	82	2	( $-2.0, 0.0$ )	—
Parabolic	75	92	3	( $-2.0, 1.19 \times 10^{-13}$ )	( $-2.0, 0.0$ )

### f1\_2: $f1_2(x,y)=x^2 + (y - 3)^2$ (минимум в (0,3))

Параметры: params = { "x0": 100, "y0": 2, "stop": 1e-6, "h": 0.1, "iterations": 2000, "decreasing\_lr\_iterations": 50000 }

Метод	Число итераций	Вызовы функции	Вызовы градиента	( $x, y$ )	Результат SciPy ( $x, y$ )
Default (GD)	948	3792	948	( $4.910 \times 10^{-7}, 2.999999995$ )	—
Decreasing LR	50000	200000	50000	( $8.268 \times 10^{-25}, 3.000000000$ )	—
Armijo	5	12	2	( $0.0, 3.0$ )	( $0.0, 3.0$ )
Goldstein	4	11	2	( $0.0, 3.0$ )	( $0.0, 3.0$ )
Golden Section	69	82	3	( $5.0945 \times 10^{-12}, 2.99999999999948$ )	( $6.8789 \times 10^{-18}, 3.0$ )
Dihotomiya	23	70	2	( $0.0, 3.0$ )	—
Parabolic (Brent)	36	46	2	( $1.0345 \times 10^{-8}, 2.99999999896554$ )	( $1.4211 \times 10^{-14}, 3.0$ )

Параметры: params = { "x0": 75, "y0": 3, "stop": 1e-7, "h": 0.05, "iterations": 2000, "decreasing\_lr\_iterations: 30000" }

Метод	Количество итераций	Количество вызовов функции	Количество вычислений градиента	$(x, y)$	Результат SciPy $(x, y)$
Default (GD)	1047	4188	1047	$(4.98 \times 10^{-8}, 3.0)$	—
Decreasing LR	30000	120000	30000	$(7.41 \times 10^{-14}, 3.0)$	—
Armijo	5	12	2	$(0.0, 3.0)$	$(0.0, 3.0)$
Goldstein	4	11	2	$(0.0, 3.0)$	$(0.0, 3.0)$
Golden Sect.	73	86	3	$(5.11 \times 10^{-12}, 3.0)$	$(5.16 \times 10^{-18}, 3.0)$
Dihotomiya	27	82	2	$(0.0, 3.0)$	—
Parabolic	75	92	3	$(2.98 \times 10^{-12}, 3.0)$	$(0.0, 3.0)$

$$f_1-3: f_1-3(x,y) = (x - 2)^2 + (y + 1)^2 \text{ (минимум в } (2, -1))$$

Параметры: params = { "x0": 100, "y0": 2, "stop": 1e-6, "h": 0.1, "iterations": 2000, "decreasing\_lr\_iterations: 50000" }

Метод	Количество итераций	Количество вызовов функции	Количество вычислений градиента	$(x, y)$	Результат SciPy $(x, y)$
Default (GD)	947	3788	947	$(2.00000049, -1.0)$	—
Decr. LR	50000	200000	50000	$(2.0, -1.0)$	—
Armijo	5	12	2	$(2.0, -1.0)$	$(2.0, -1.0)$
Goldstein	4	11	2	$(2.0, -1.0)$	$(2.0, -1.0)$
Golden Sect.	69	82	3	$(2.0, -1.0)$	$(2.0, -1.0)$
Dihotomiya	23	70	2	$(2.0, -1.0)$	—
Parabolic	36	46	2	$(2.00000001, -1.0)$	$(2.0, -1.0)$

Параметры: params = { "x0": 75, "y0": 3, "stop": 1e-7, "h": 0.05, "iterations": 2000, "decreasing\_lr\_iterations": 30000 }

Метод	Количество итераций	Количество вызовов функции	Количество вычислений градиента	$(x, y)$	Результат SciPy $(x, y)$
Default (GD)	1046	4184	1046	(2.00, -1.00)	—
Decreasing LR	30000	120000	30000	(2.00, -1.00)	—
Armijo	5	12	2	(2.00, -1.00)	(2.00, -1.00)
Goldstein	4	11	2	(2.00, -1.00)	(2.00, -1.00)
Golden Sect.	73	86	3	(2.00, -1.00)	(2.00, -1.00)
Dihotomiya	27	82	2	(2.00, -1.00)	—
Parabolic (Brent)	75	92	3	(2.00, -1.00)	(2.00, -1.00)

**f3: f3(x,y)= $x^2 + B * x * y + y^2$  (при  $-2 < B < 2$  (минимум в (0,0)))**

Параметры: params = { "x0": 100, "y0": 2, "stop": 1e-6, "h": 0.1, "iterations": 2000, "decreasing\_lr\_iterations": 50000 }

Метод	Число итераций	Вызывы функции	Вызывы градиента	( $x, y$ )	Результат SciPy ( $x, y$ )
Default (GD)	2000	8000	2000	(0.00216949, -0.00216949)	—
Decreasing LR	50000	200000	50000	$(1.0104 \times 10^{-8}, -1.0104 \times 10^{-8})$	—
Armijo	254	509	64	$(8.83496 \times 10^{-7}, -6.54774 \times 10^{-7})$	$(5.25945 \times 10^{-8}, -8.02472 \times 10^{-8})$
Goldstein	191	446	64	$(8.83496 \times 10^{-7}, -6.54774 \times 10^{-7})$	$(5.25945 \times 10^{-8}, -8.02472 \times 10^{-8})$
Golden Section	466	544	16	$(1.33330 \times 10^{-6}, -1.38666 \times 10^{-6})$	$(1.35690 \times 10^{-6}, -1.41118 \times 10^{-6})$
Dihotomiya	331	921	16	$(1.35714 \times 10^{-6}, -1.41143 \times 10^{-6})$	—
Parabolic (Brent)	527	635	16	$(1.32763 \times 10^{-6}, -1.38064 \times 10^{-6})$	$(1.35691 \times 10^{-6}, -1.41119 \times 10^{-6})$

Параметры: params = { "x0": 75, "y0": 3, "stop": 1e-7, "h": 0.05, "iterations": 2000, "decreasing\_lr\_iterations": 30000 }

Метод	Количество итераций	Количество вызовов функции	Количество вычислений градиента	( $x, y$ )	Результат SciPy ( $x, y$ )
Default (GD)	2000	8000	2000	(0.00159, -0.00159)	—
Decreasing LR	30000	120000	30000	(0.00645, -0.00645)	—
Armijo	282	565	71	(6.38e-08, -8.71e-08)	(3.88e-09, -3.21e-09)
Goldstein	212	495	71	(6.38e-08, -8.71e-08)	(3.88e-09, -3.21e-09)
Golden Sect.	572	660	18	(4.82e-08, -5.00e-08)	(4.83e-08, -5.02e-08)
Dihotomiya	443	1245	18	(4.83e-08, -5.02e-08)	—
Parabolic (Brent)	612	734	18	(4.81e-08, -4.99e-08)	(4.83e-08, -5.02e-08)

На обычных гладких, выпуклых, квадратичных функциях двух переменных все методы как реализованные нами, так и библиотечные, корректно находят минимум. При чем на всех функциях, кроме f3, методы поиска оптимального шага справляются за сравнительно малое количество итераций, на функции f3 поведение спуска становится более интересным и количество итераций возрастает. Для метода “decreasing\_lr” количество итераций специально установлено в разы больше, потому что при малом количестве итераций метод не успевает сойтись.

**multimodal\_f:  $20 + x^2 + y^2 - 10\cos(2\pi x) - 10\cos(2\pi y)$  ( 1  
глобальный минимум (0,0) и множество локальных)**

Параметры: params = { "x0": 100, "y0": 2, "stop": 1e-6, "h": 0.1, "iterations": 2000, "decreasing\_lr\_iterations": 50000 }

Метод	Количество итераций	Количество вызовов функции	Количество вычислений градиента	( $x, y$ )	Результат SciPy ( $x, y$ )
Default (GD)	2000	8000	2000	(29.80, 2.35)	—
Decr. LR	1377	5508	1377	(1.99, $-4.02 \times 10^{-12}$ )	—
Armijo	5	12	2	(0.0, 0.0)	(0.0, 0.0)
Goldstein	4	11	2	(0.0, 0.0)	(0.0, 0.0)
Golden Sect.	63425	73425	2000	(-5.94, 3.98)	(2.98, $-3.46 \times 10^{-10}$ )
Dihotomiya	23	70	2	(0.0, 0.0)	—
Parabolic	65015	79015	2000	(1.05, -0.994)	$(5.77 \times 10^{-10}, -5.45 \times 10^{-11})$

Параметры: params = { "x0": 75, "y0": 3, "stop": 1e-7, "h": 0.05, "iterations": 2000, "decreasing\_lr\_iterations": 30000 }

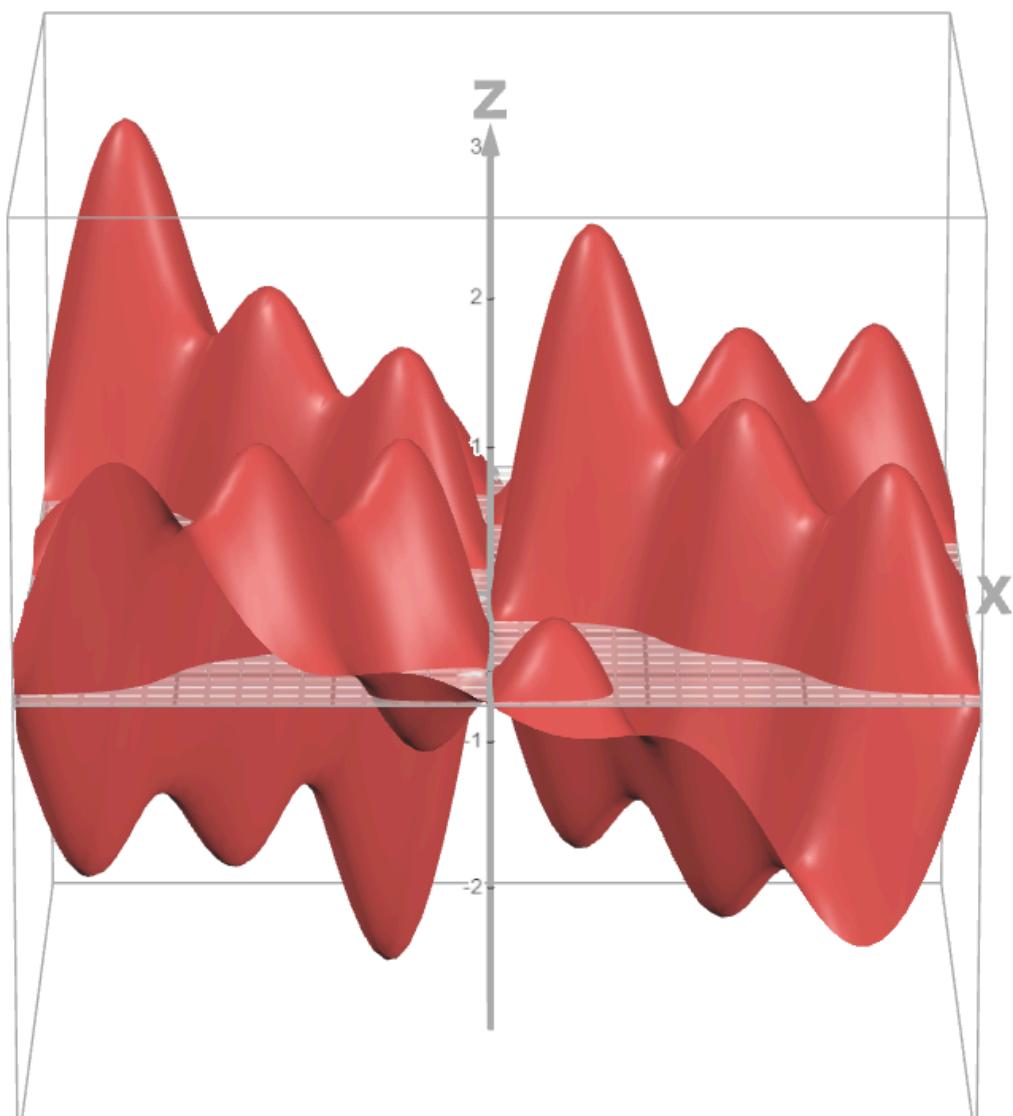
Метод	Количество итераций	Количество вызовов функции	Количество вычислений градиента	( $x, y$ )	Результат SciPy ( $x, y$ )
Default (GD)	2000	8000	2000	(27.8268, 2.8045)	—
Decreasing LR	161	644	161	(20.8843, 1.9899)	—
Armijo	5	12	2	(0.0, 0.0)	(0.0, 0.0)
Goldstein	4	11	2	(0.0, 0.0)	(0.0, 0.0)
Golden Sect.	71569	81569	2000	(-1.0482, 0.9922)	$(-0.9949586, 2.50 \times 10^{-14})$
Dihotomiya	27	82	2	(0.0, 0.0)	—
Parabolic	73470	87470	2000	(-2.0566, 2.9834)	$(0.9949586, 1.14 \times 10^{-10})$

На мультимодальной функции отлично показали себя методы Армихо, Голдстейна, а так же дихотомии, которые нашли корректный минимум функции. Результаты стандартного метода и метода с динамическим выбором шага оценить сложно, однако, при увеличении количества итераций, второй таки достиг локального минимума в точке (2; 0). Методы золотого сечения и параболической интерполяции нашли локальные минимумы.

Встроенные методы `minimize_scalar`, кажется, так же были около локальных минимумов, но при большом количестве итераций, встроенный метод параболической интерполяции достиг глобального минимума функции. Методы `line_search` также нашли глобальный минимум.

$$\text{noisy\_multimodal\_f} = \sum \sin(m \cdot x) \cdot \cos(m \cdot y), m = 1..M + N(0, \sigma)$$

Данная функция имеет бесконечное количество глобальных и локальных минимумов. Для удобства анализа результатов (методом просмотра глазами на 3D функцию), была выбрана начальная точка (0; 0).



Параметры: params = { "x0": 0, "y0": 0, "stop": 1e-6, "h": 0.1, "iterations": 2000, "decreasing\_lr\_iterations": 50000, "noise": 0.0, "multimodality": 3 }

Метод	Количество итераций	Количество вызовов функции	Количество вычислений градиента	( $x, y$ )	Результат SciPy ( $x, y$ )
Default (GD)	13	52	13	(-0.6672911092638969, 0.0)	—
Decreasing LR	25	100	25	(-0.6672910070037089, 0.0)	—
Armijo	70	113	11	(-0.6672911401838063, 0.0)	(-0.6672910715489765, 0.0)
Goldstein	60	103	11	(-0.6672911401838063, 0.0)	(-0.6672910715489765, 0.0)
Golden Section	71	84	3	(-6.950476385772965, 0.0)	(-0.6672910674904292, 0.0)
Dihotomiya	45	127	3	(-2.640083648663914, 0.0)	—
Parabolic (Brent)	40	50	2	(-6.95047637824605, 0.0)	(-2.640083649041431, 0.0)

При отсутствии шумов все реализованные нами методы нашли корректные глобальные минимумы, кроме метода дихотомии, который нашел лишь локальный минимум. Аналоги из `scipy.optimize` так же нашли глобальные минимумы, кроме метода `minimize_scalar(..., method="brent")`, реализующего метод параболической интерполяции, он так же нашел лишь локальный минимум.

Параметры: params = { "x0": 0, "y0": 0, "stop": 1e-6, "h": 0.1, "iterations": 2000, "decreasing\_lr\_iterations": 50000, "noise": 0.2, "multimodality": 3 }

Метод	Количество итераций	Количество вызовов функции	Количество вычислений градиента	( $x, y$ )	Результат SciPy ( $x, y$ )
Default (GD)	13	52	13	(-0.6672911092638972, 0.0)	—
Decreasing LR	25	100	25	(-0.6672910068178832, 0.0)	—
Armijo	70	113	11	(-0.6672911392524838, 0.0)	(-0.6672910715489764, 0.0)
Goldstein	60	103	11	(-0.6672911392524838, 0.0)	(-0.6672910715489764, 0.0)
Golden Section	71	84	3	(-6.950476383468198, 0.0)	(-0.6672910677787257, 0.0)
Dihotomiya	45	127	3	(-2.6400836486639143, 0.0)	—
Parabolic (Brent)	40	50	2	(-6.950476378267171, 0.0)	(-2.6400836490332744, 0.0)

Добавление небольшого шума не повлияло на результаты.

Сделаем чуть больше шума: params = { "x0": 0, "y0": 0, "stop": 1e-6, "h": 0.1, "iterations": 2000, "decreasing\_lr\_iterations": 50000, "noise": 0.6, "multimodality": 3 }

Метод	Количество итераций	Количество вызовов функции	Количество вычислений градиента	(x, y)	Результат SciPy (x, y)
Default (GD)	13	52	13	(-0.6672911092638972, 0.0)	—
Decreasing LR	25	100	25	(-0.6672910068178832, 0.0)	—
Armijo	70	113	11	(-0.6672911392524838, 0.0)	(-0.6672910715489764, 0.0)
Goldstein	60	103	11	(-0.6672911392524838, 0.0)	(-0.6672910715489764, 0.0)
Golden Section	71	84	3	(-6.950476383468198, 0.0)	(-0.6672910677787257, 0.0)
Dihotomiya	45	127	3	(-2.6400836486639143, 0.0)	—
Parabolic (Brent)	40	50	2	(-6.9504763782680135, 0.0)	(-2.6400836490373445, 0.0)

Результаты все так же не изменились. Сделаем еще больше шума и изменим точку старта на точку около глобального максимума функции.

Параметры: params = { "x0": 0.7, "y0": 0, "stop": 1e-6, "h": 0.1, "iterations": 2000, "decreasing\_lr\_iterations: 50000" "noise" = 5.0 "multimodality" = 3 }

Метод	Количество итераций	Количество вызовов функции	Количество вычислений градиента	(x, y)	Результат SciPy (x, y)
Default (GD)	16	64	16	(1.8151983543103618, 0.0)	—
Decreasing LR	139	556	139	(1.8151982815889542, 0.0)	—
Armijo	33	60	7	(1.8151984387184366, 0.0)	(-1.65373292538544, -0.9864455888338969)
Goldstein	228	259	8	(1.8151984197972988, 0.0)	(-1.6536896948863242, -0.9863946331989978)
Golden Section	95	113	4	(3.643101653935821, 0.0)	(1.8151984126581335, 0.0)
Dihotomiya	67	191	4	(5.6158942366531495, 0.0)	—
Parabolic (Brent)	96	120	4	(3.643101652751662, 0.0)	(1.8151984095332803, 0.0)

Теперь ситуация изменилась, только метод дихотомии нашел глобальный минимум, все остальные методы, включая методы `scipy.optimize` нашли лишь локальный минимум. Таким образом, при умеренном количестве шумов, большинство методов корректно находят глобальный минимум и даже при большом количестве шумов сходятся, пусть и в основном к локальным минимумам.

## **ВЫВОД:**

В ходе лабораторной работы мы продемонстрировали результаты работы методов с разным набором параметров и выяснили, что независимо от начальных условий и параметров для простых квадратичных функций (`f1,f1_1,f1_2, f1_3`) все методы сходятся к аналитически известному минимуму (при этом кол-во итераций и вычислений может значительно различаться); Методы Armijo, Goldstein, golden\_section, дихотомия и parabolic показывают гораздо более эффективную сходимость - они достигают критерия остановки за несколько итераций (4–5 для квадратичных функций), тогда как фиксированные стратегии требуют тысячи итераций и все равно могут быть относительно далеки; если сравнивать `default` и `decreasing_lr`, то `default` даёт быстрое приближение к минимуму со средним количеством итераций, а `decreasing_lr` выдает более высокую точность но требует для этого увеличенное число итераций.

Таким образом эксперимент ясно показывает, что адаптивный выбор шага позволяет достичь более быстрой сходимости и высокой точности найденного минимума (также подчеркивает важность корректного выбора гиперпараметров и критериев останова в алгоритмах оптимизации)

P.S все графики самих функций можно посмотреть по следующим ссылкам:

- **f1:** <https://www.desmos.com/3d/giodtu9kxu>
- **f1\_1:** <https://www.desmos.com/3d/ornullhu8i>
- **f1\_2:** <https://www.desmos.com/3d/kalf5u3nwt>
- **f1\_3:** <https://www.desmos.com/3d/g6bmltzhzq>
- **f3 (B = 1.5):** <https://www.desmos.com/3d/neht5nufl>
- **multimodal\_f:** <https://www.desmos.com/3d/atupv1wwkl>
- **noisy\_multimodal\_f (без шумов, коэффициент мультимодальности M = 3):** <https://www.desmos.com/3d/9cktbjmp3x>