

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

Методы оптимизации
Отчет по лабораторной работе №1

Работу выполнили:
Володько А.А., М32321
Ярунина К.А., М32371
Преподаватель:
Шохов М.Е.



Санкт-Петербург
2023

Постановка задачи:

1. Реализуйте градиентный спуск с постоянным шагом (learning rate).
2. Реализуйте метод одномерного поиска (метод дихотомии, метод Фибоначчи, метод золотого сечения) и градиентный спуск на его основе.
3. Проанализируйте траекторию градиентного спуска на примере квадратичных функций. Для этого придумайте две-три квадратичные функции от двух переменных, на которых работа методов будет отличаться.
4. Для каждой функции:
 - (a) исследуйте сходимость градиентного спуска с постоянным шагом, сравните полученные результаты для выбранных функций;
 - (b) сравните эффективность градиентного спуска с использованием одномерного поиска с точки зрения количества вычислений минимизируемой функции и ее градиентов;
 - (c) исследуйте работу методов в зависимости от выбора начальной точки;
 - (d) исследуйте влияние нормализации (scaling) на сходимость на примере масштабирования осей плохо обусловленной функции;
 - (e) в каждом случае нарисуйте графики с линиями уровня и траекториями методов;
5. Реализуйте генератор случайных квадратичных функций n переменных с числом обусловленности k .
6. Исследуйте зависимость числа итераций $T(n, k)$, необходимых градиентному спуску для сходимости в зависимости от размерности пространства $2 \leq n \leq 10^3$ и числа обусловленности оптимизируемой функции $1 \leq k \leq 10^3$.
7. Для получения более корректных результатов проведите множественный эксперимент и усредните полученные значения числа итераций.
8. Реализуйте одномерный поиск с учетом условий Вольфе и исследуйте его эффективность. Сравните полученные результаты с реализованными ранее методами.

1. Градиентный спуск с постоянным шагом

Решаем задачу нахождения минимума дифференцируемой функции f . Идея метода градиентного спуска с постоянным шагом заключается в том, чтобы осуществлять оптимизацию в направлении наискорейшего спуска, а это направление задаётся антиградиентом (градиентом $\cdot(-1)$). Тогда на n -ой итерации алгоритма X_n (n -ое приближение X) = $X_{n-1} - lr \cdot \nabla f_{n-1}$, где lr – размер постоянного шага.

В данной реализации мы ставим алгоритму фиксированное количество итераций (epoch), то есть останавливаемся на epoch-ом приближении.

```
def grad_f(func, X: np.array) -> np.array:
    return np.array(nd.Gradient(func)(X))

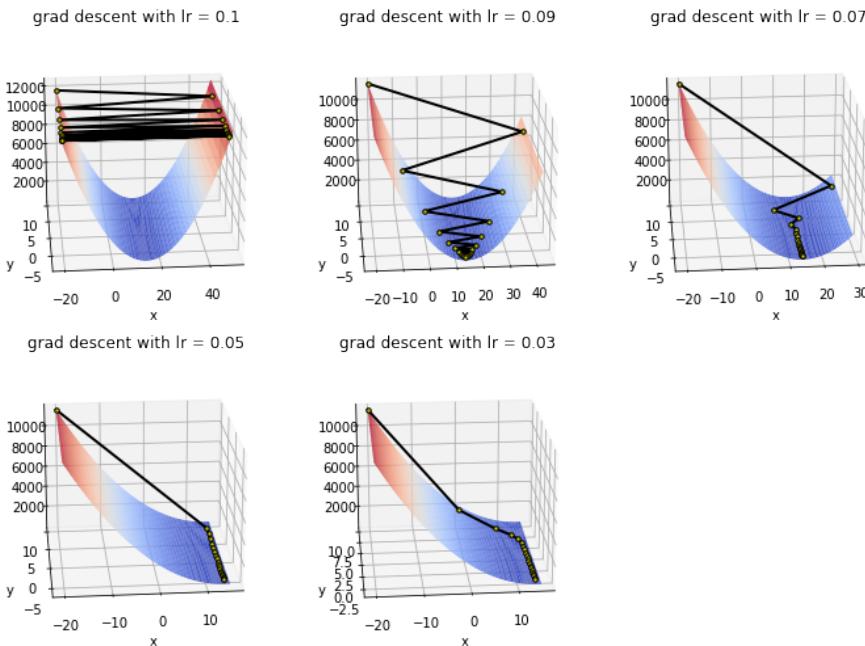
def grad_descent(f, X: np.array, lr=0.09, epoch=20) -> np.array:
    history = [np.hstack((X, f(X)))]
    for _ in range(epoch):
        X = X - lr * grad_f(f, X)
        history.append(np.hstack((X, f(X))))
    return np.array(history)

def fun(X: np.array) -> float:
    return 10 * (X[0] - 14)**2 + (X[1] + 7)**2

X = np.array([-20, 10])
grad_points = grad_descent(fun, X)
```

Рассмотрим работу градиентного спуска с различным заданным learning rate на примере функции $f(x, y) = 10(x - 14)^2 + (y + 7)^2$.

Несложно заметить, что минимум функции достигается в точке $(14, -7)$



По графикам видно, что learning rate существенно влияет на сходимость градиентного спуска. Для большей наглядности рассмотрим точки, в которых заканчивает работу градиентный спуск с различным параметром lr.

lr	end point
0.1	(-20, -6.8)
0.09	(13.6, -6.68)
0.07	(13.9, -6.17)
0.05	(14, -4.93)
0.03	(13.9, -2)

Как нетрудно заметить по таблице, худший результат сходимости у градиентного спуска с $lr = 0.1$. Анализируя график, становится ясно, что в данном случае из-за слишком большого lr градиентный спуск делает слишком большой шаг в сторону антиградиента из-за чего ходит зигзагом, колебаясь примерно на одной высоте, не успевая тем самым за 20 итераций спуститься ближе к минимуму.

При меньших значениях lr картина меняется: мы видим, что по оси Ox к точке минимума удалось приблизиться намного сильнее, чем по Oy . Это связано с коэффициентами перед квадратами переменных функции. По Ox функция как бы растянута 10кой, из-за чего у градиента значение для x по модулю в 10 раз больше, чем для y , поэтому и приближаемся по Ox быстрее.

Из-за этого слишком маленький lr тоже не лучший выбор: при $lr = 0.01$ мы оказались в точке $(13.9, -2)$, очень далеки от минимума по Oy .

2. Одномерный метод дихотомии и градиентный спуск на его основе
(сделали сразу многомерный, чтобы была возможность сравнивать с градиентным спуском из пункта 1, на одной переменной также работает)

Идея метода градиентного спуска на основе дихотомии отличается от спуска с постоянным шагом только выбором значения шага. В данной реализации мы используем не постоянное значение, а значение, вычисленное с помощью метода дихотомии.

Сперва рассмотрим, как устроен одномерный метод дихотомии:

```
def dichotomy_1d(f, x: float, eps=1e-5):
    grad = grad_f(f, x)
    a, b = 0, 1
    while abs(a - b) / 2 > eps:
        x1 = (a + b - eps) / 2
        x2 = (a + b + eps) / 2
        if f(x1) < f(x2):
            b = x2
        else:
            a = x1
    return (a + b) / 2
```

Здесь мы ищем минимум одномерной функции на отрезке $[0, 1]$: делим его на две части, и проверяем, на какой из двух половин функция f принимает меньшее значение. Затем мы повторяем этот процесс для подинтервала, на котором функция принимает меньшее значение, и так далее, пока не достигнем необходимой точности. Но зачем? В данной реализации градиентного спуска мы используем метод дихотомии для нахождения оптимального шага – коэффициента перед градиентом. Получается, что на каждой итерации градиентного спуска мы смотрим на градиент функции в текущем значении X_n (вектор), рассматриваем прямую, с направлением вектора и пересекающую точку X_n , и дихотомией выбираем X_{n+1} так, чтобы он соответствовал минимуму f на этой прямой.

Теперь к многомерной реализации: делаем все то же самое. Рассматриваем коэффициенты на отрезке от 0 до 1, но в качестве X_i выбираем $X - \alpha_i \nabla f(X)$. Таким образом в градиентном спуске в $X_{n+1} = X_n - Lr \nabla f(X_n)$ мы получаем точку с минимальным значением f на направлении градиента, которая принадлежит промежутку от X до $X - \nabla f(X)$

```
def dichotomy(f, X: np.array, eps=1e-5):
    grad = grad_f(f, X)
    a, b = 0, 1
    function_calls = 0
    while abs(a - b) / 2 > eps:
        alpha1 = (a + b - eps) / 2
        alpha2 = (a + b + eps) / 2
        X1, X2 = X - alpha1 * grad, X - alpha2 * grad
        if f(X1) < f(X2):
            b = alpha2
        else:
            a = alpha1
        function_calls += 2
    return (a + b) / 2, function_calls

def dichotomy_grad_descent(f, X: np.array, epoch=20) -> np.array:
    history = [np.hstack((X, f(X)))]
    for _ in range(epoch):
        lr, _ = dichotomy(f, X)
        X = X - lr * grad_f(f, X)
        history.append(np.hstack((X, f(X))))
    return np.array(history)

def f(X: np.array) -> float:
    return (X[0] - 14.678) ** 2 + (X[1] - 7.5748) ** 2 + X[0]*X[1]

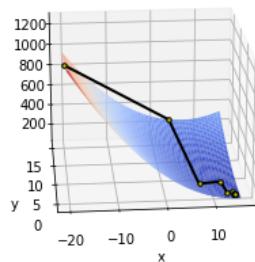
X = np.array([-20, 10])
```

```

points = dichotomy_grad_descent(f, X)

gradient descent with dichotomy for f(x,y) = (x-14.678)^2 + (y-7.5748)^2 + x*y

```



3. Анализ работы градиентного спуска с постоянным шагом и на основе метода дихотомии на примере квадратичных функций

Рассмотрим 3 функции (их запомним, будем пользоваться ими же еще и в №4):

```

def fun1(X: np.array) -> float:
    x, y = X[0], X[1]
    return 10 * (x - 14) ** 2 + (y - 7) ** 2

def fun2(X: np.array):
    x, y = X[0], X[1]
    return 0.5 * x ** 2 + 3 * y ** 2 + x*y

def fun3(X: np.array):
    x, y = X[0], X[1]
    return 45.51 * x ** 2 + 28.42 * x * y + 4.98 * y ** 2

```

Ниже показаны графики, как выглядят эти функции

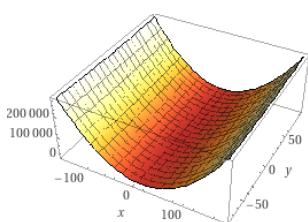


Рис. 1: A really Awesone Image

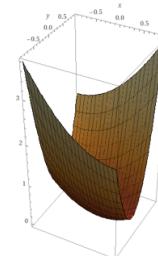


Рис. 2: A really Awesone Image

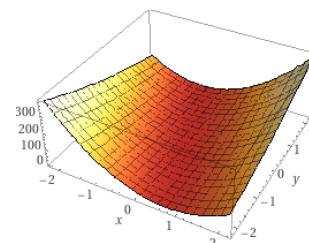


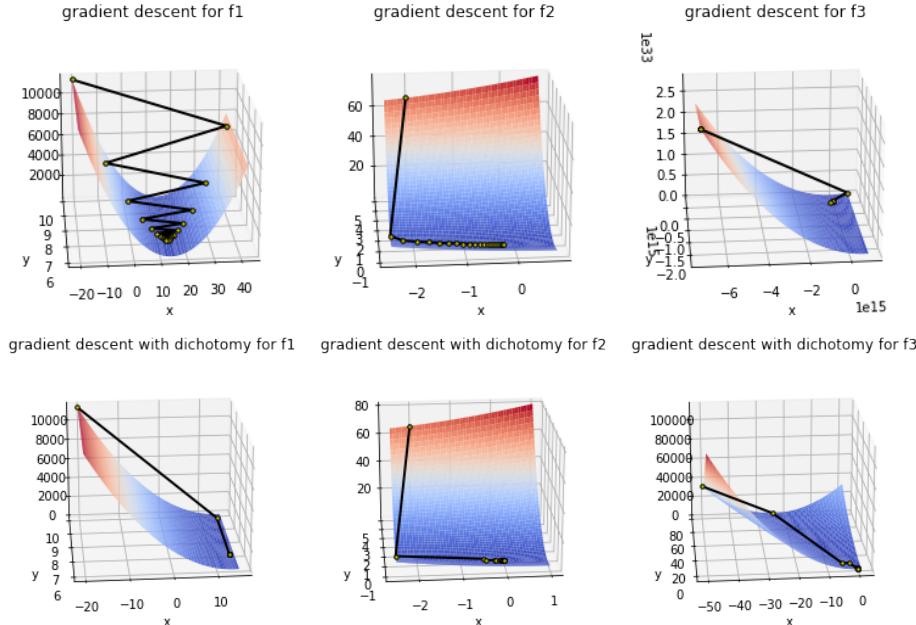
Рис. 3: A really Awesone Image

Несмотря на то, что первая и третья функции достаточно похожи, первая функция более приятная, ее число обусловленности = 10, и все должно неплохо работать. У второй число обусловленности 6, а у третьей оно около 100, и здесь уже могут возникать некоторые проблемы

Приведем так же реальные значения точек, в которых достигаются минимумы к функций

$$\begin{aligned}
 f_1: & (14, 7) \\
 f_2: & (0, 0) \\
 f_3: & (0, 0)
 \end{aligned}$$

Выбранные для каждой функции начальные точки: для $f_1: (-20, 10)$, $f_2: (-2, 5)$, $f_3: (-50, 70)$
 $lr=0.09$; epoch=20



Во первых, по данным картинкам видно, что метод на основе дихотомии приближается к минимуму значительно быстрее на всех трех функциях, так как шаг все таки выбирается не константно, а обоснованно. При данном значении epoch=20, приведем последнее приближение в каждом случае:

f_1 lr:	(13.60800669, 7.05667588)
f_1 dich:	(14,7)
f_2 lr:	(-0.2160566, 0.0416087)
f_2 dich:	(-0.00029494, 0.00015232)
f_3 lr:	(-7.17743562e+15, -2.28307566e+15)
f_3 dich:	(-1.87284535e-09, 2.62203278e-09)

Можем заметить, что f_1 во втором случае действительно нашла минимум за 20 итераций, и в целом, у всех функций значения на выходе метода градиентного спуска на основе дихотомии получились сильно ближе к действительным, чем у метода с постоянным шагом: f_3 вообще расходится в первом методе, но сходится во втором. Расходимость f_3 связана напрямую с ее числом обусловленности, сильное изменение значения функции при небольшом изменении аргумента влечет большой градиент в точке, из-за чего мы прыгнули от правильного ответа на квадрилион, так как константа из Lr не смогла сбалансировать значение градиента. Более осмысленный выбор шага в методе на основе дихотомии как раз сбалансировал позволил, благодаря чему у нас таки получилось приблизиться к минимуму.

Сравнивая f_1 и f_2 , можно отметить, что у f_2 оба метода сразу смогли выбрать оптимальное направление, а у f_1 из-за плохой обусловленности функции метод с постоянным шагом ходит зигзагом, это увеличивает количество итераций, необходимое для приближения.

4. Исследование методов градиентного спуска с постоянным шагом/на основе дихотомии в зависимости от различных параметров

Пара вспомогательных функций: реализация градиентных спусков с постоянным шагом и на основе дихотомии, возвращающие помимо истории приближений X количество вызовов функции f и ее градиента за период работы алгоритма, а так же работающие не до фиксированного количества итераций, а до момента, когда X_n отличается от X_{n-1} меньше, чем на эпсилон.

```
def dichotomy_grad_descent_calls(f, X: np.array, eps=1e-5):
    f_calls, grad_calls = 0, 0
    history = [np.hstack((X, f(X)))]
    while True:
        grad = grad_f(f, X)
        grad_calls += 1
        lr, f_call = bivariate_dichotomy(f, X)
        f_calls += f_call
        step = lr * grad
        if np.linalg.norm(step) < eps:
            break
    return f_calls, grad_calls, history
```

```

X = X - step
history.append(np.hstack((X, f(X))))
return np.array(history), f_calls, grad_calls

def grad_descent_calls(f, X: np.array, eps=1e-5, lr=0.09):
    grad_calls = 0
    history = [np.hstack((X, f(X)))]
    while True:
        grad = grad_f(f, X)
        step = lr * grad
        if np.linalg.norm(step) < eps:
            break
        X = X - step
        history.append(np.hstack((X, f(X))))
        grad_calls += 1
    return np.array(history), grad_calls # returns only calls on the gradient cz gradient descent with learning rate
→ doesn't call f

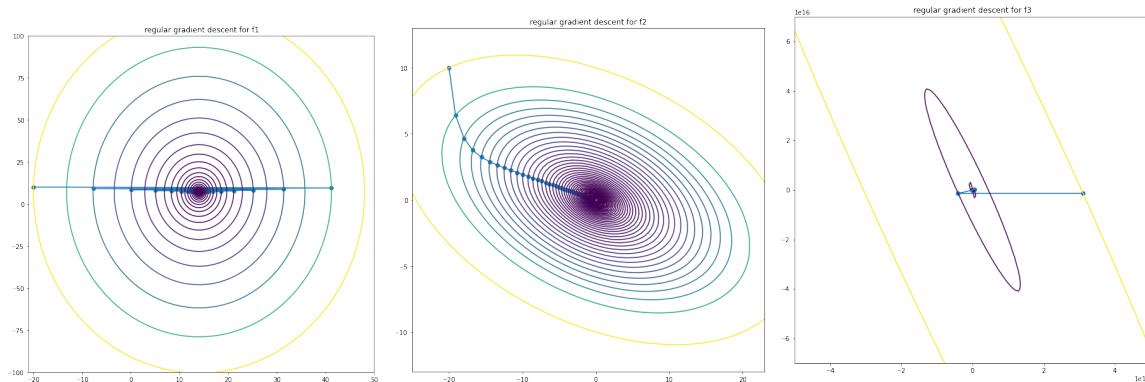
```

- (a) Рассмотрим сходимость градиентного спуска с постоянным шагом.

Ниже представлены графики с линиями уровня и траекториями метода градиентного спуска с постоянным шагом для функций f_1 , f_2 и f_3 . Здесь для всех функций начальная точка $(x, y) = (-20, 10)$, и мы не ограничиваем количество итераций алгоритма, считаем, пока X_n отличается от X_{n-1} больше, чем на эпсилон

Вот полученные значения b и графики с линиями уровня и траекториями метода с постоянным шагом:

$$\begin{aligned}f_1 \text{ lr: } & (14.00000448, 7.00000228) \\f_2 \text{ lr: } & (-1.16710423e-05, 2.24763739e-06) \\f_3 \text{ lr: } & (3.10437164e+16, -1.37176924e+15)\end{aligned}$$



Здесь мы можем заметить отчетливую зависимость близости полученного значения к реальному от числа обусловленности функции, чем меньше число обусловленности, тем лучше у метода градиентного спуска получилось приблизится к реальному значению. Благодаря хорошей обусловленности, f_2 градиент задает направление минимума функции более приближенно к правде. Даже с константным шагом, так как здесь небольшие значения градиента близ минимума, ей удалось лучше всего приблизиться к минимуму (мы умножаем градиент на константу, так что чем меньше градиент, тем более четкий шаг мы можем делать). У первой ситуация чуть хуже за счет больших значений градиента у минимума, а третьей совсем грустно, она минимум не нашла, так как значения градиента были слишком большими, и константа lr не справилась с тем, чтобы достаточно уменьшить шаг.

- (b) Сравните эффективность градиентного спуска на основе дихотомии с точки зрения количества вычислений минимизируемой функции и ее градиентов

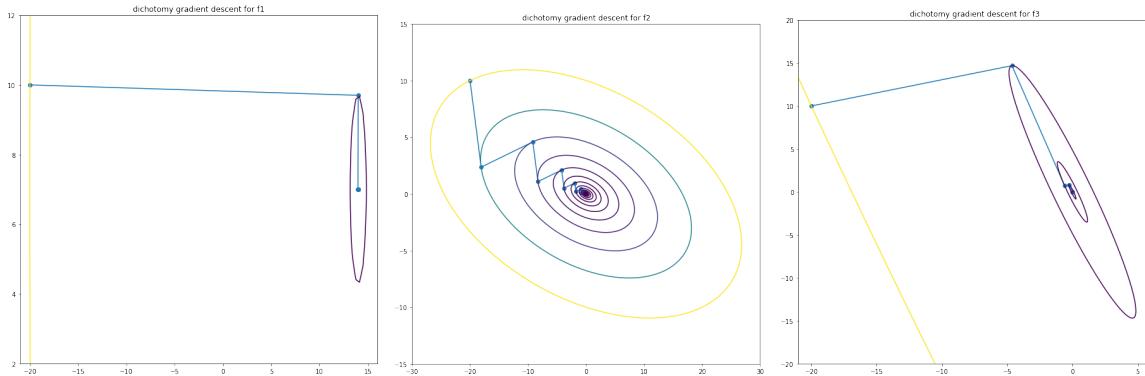
Начальная точка все так же $(x, y) = (-20, 10)$, и опять итерируемся пока X_n отличается от X_{n-1} больше, чем на эпсилон

Полученные значения:

f_1 lr:	(14.00000448, 7.00000228)
f_1 dich:	(14, 7.00000122)
f_2 lr:	(-1.30489485e-04, 2.51299787e-05)
f_2 dich:	(-1.62195004e-05, 8.10996894e-06)
f_3 lr:	(3.10437164e+16, -1.37176924e+15)
f_3 dich:	(-5.71735541e-06, 3.68307274e-06)

	f_1	f_2	f_3
lr grad calls	71	159	17
dich f calls	204	1258	306
dich grad calls	6	37	9

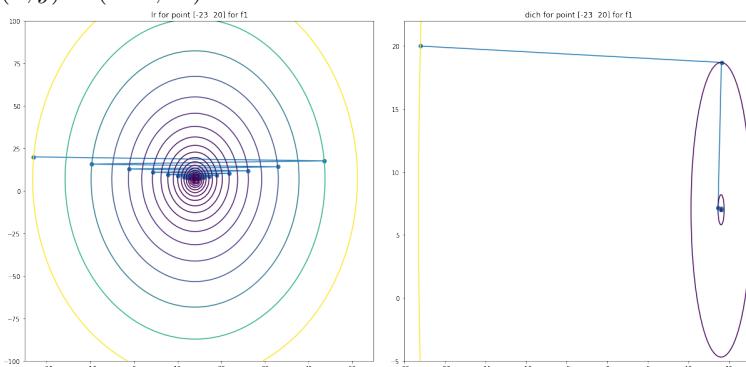
Графики с линиями уровня и траекториями метода на основе дихотомии:



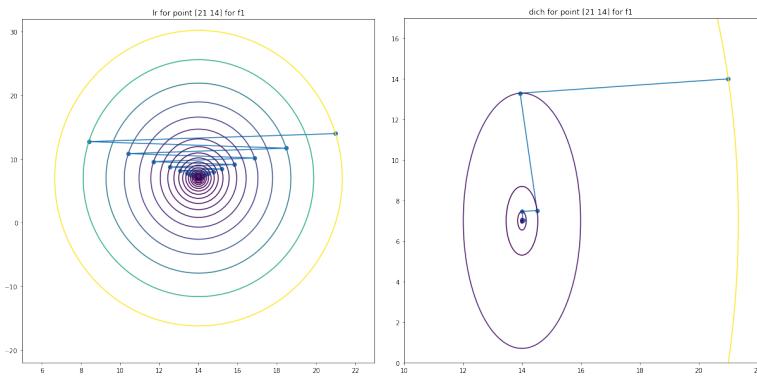
Первое, что можно заметить – f_3 наконец то сходится, причем не то чтобы число вызовов функции/итераций сильно отличалось от той же f_1 , несмотря на числа обусловленности. Здесь сыграли другие факторы – удачный для f_3 выбор начальной точки, например. И в целом, градиентный спуск на основе дихотомии добивается куда более точного результата на всех функциях за счет осознанного выбора шага. Здесь из интересных результатов так же можно выделить то, что, несмотря на наименьшее число обусловленности f_2 , она работает дольше всего в плане итераций / вызовов функции, так же, из-за небольшой значений градиента близ минимума, шаги очень осторожные выходят.

(c) Сравнение работы методов в зависимости от выбора начальной точки

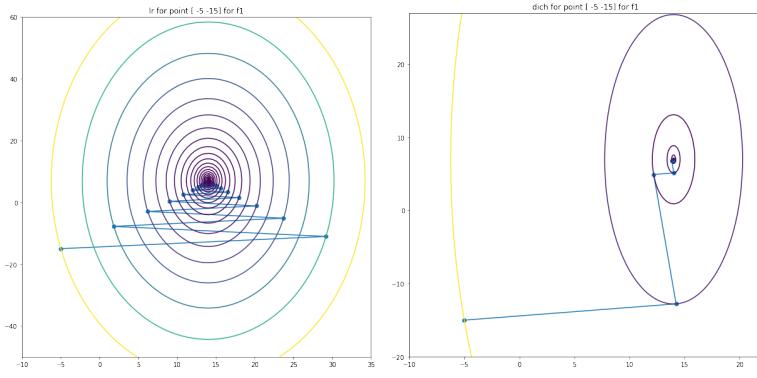
$f_1: (x, y) = (-23, 20)$



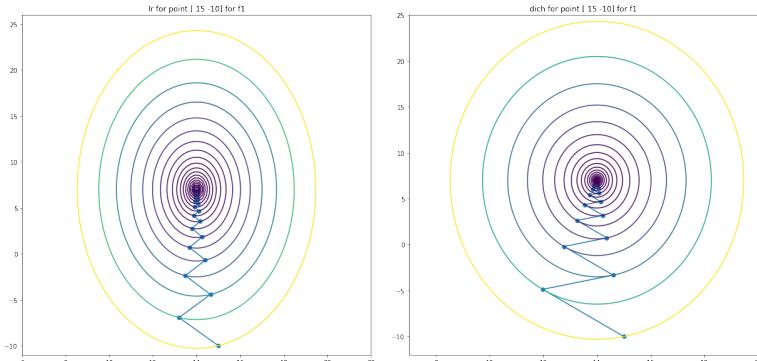
$(x, y) = (21, 14)$



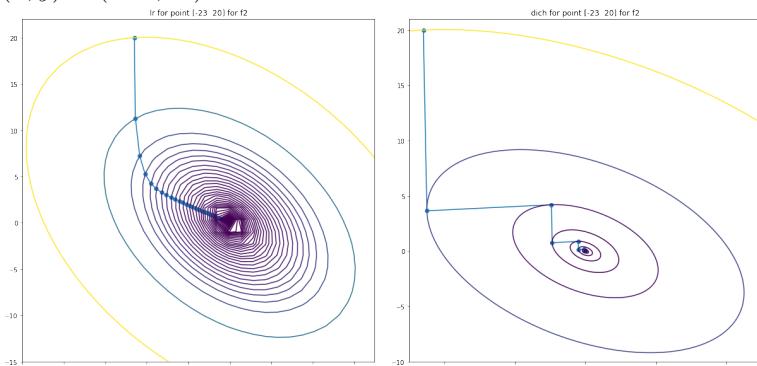
$(x, y) = (-5, -15)$



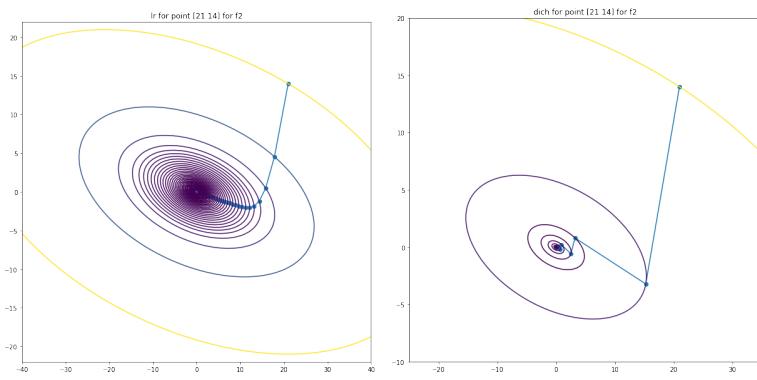
$(x, y) = (15, -10)$



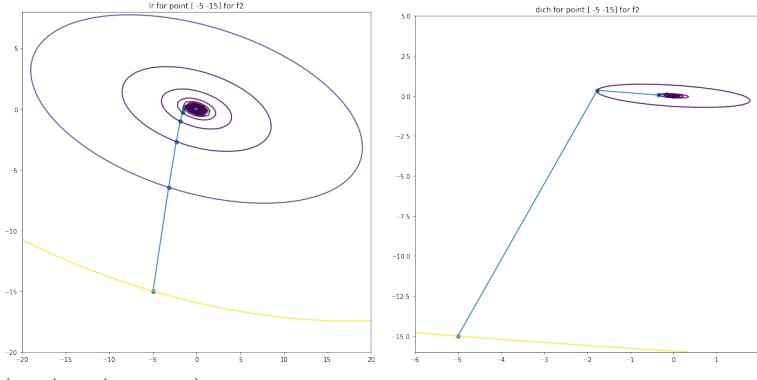
$f_2: (x, y) = (-23, 20)$



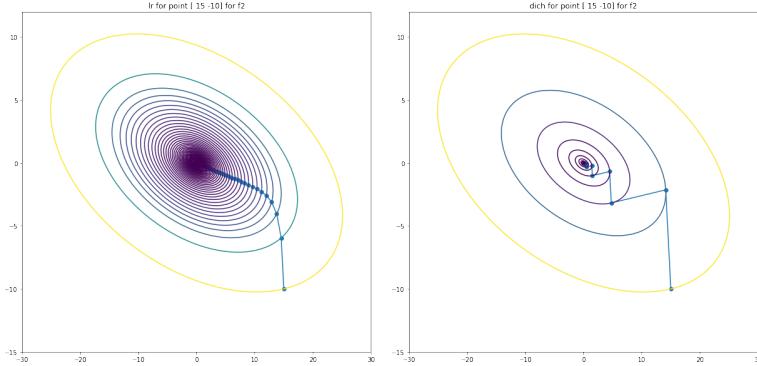
$(x, y) = (21, 14)$



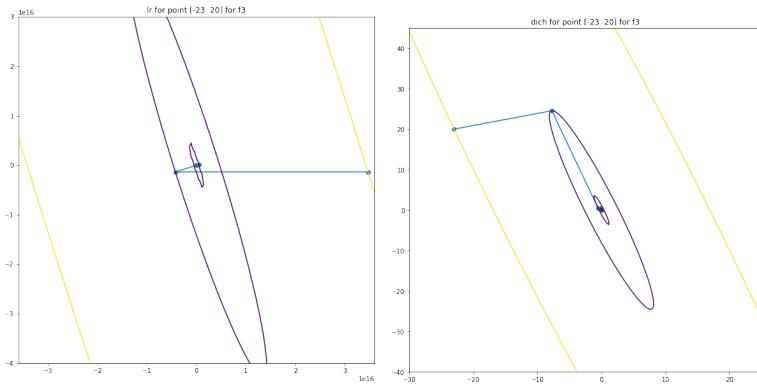
$(x, y) = (-5, -15)$



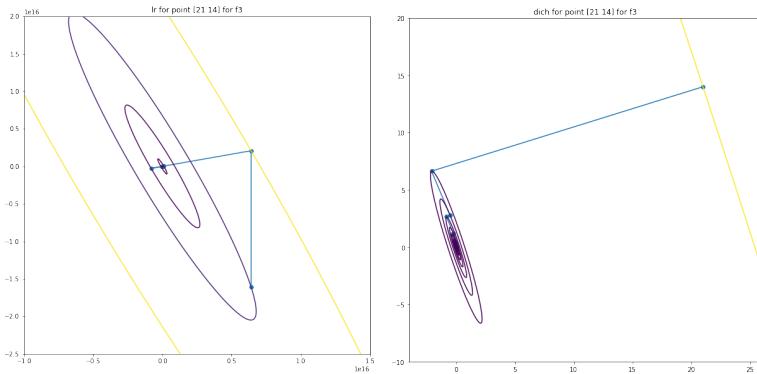
$(x, y) = (15, -10)$



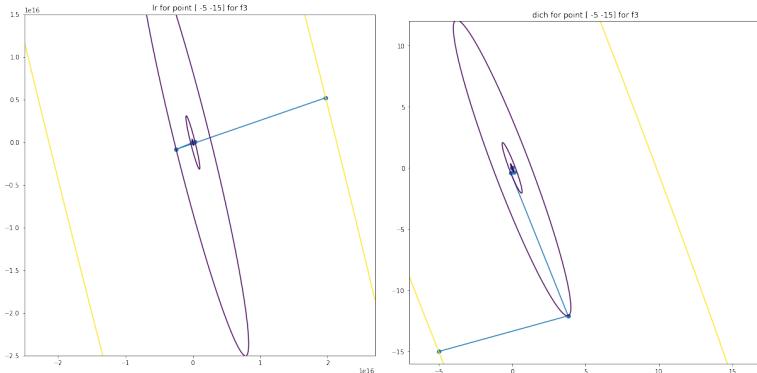
$f_3: (x, y) = (-23, 20)$



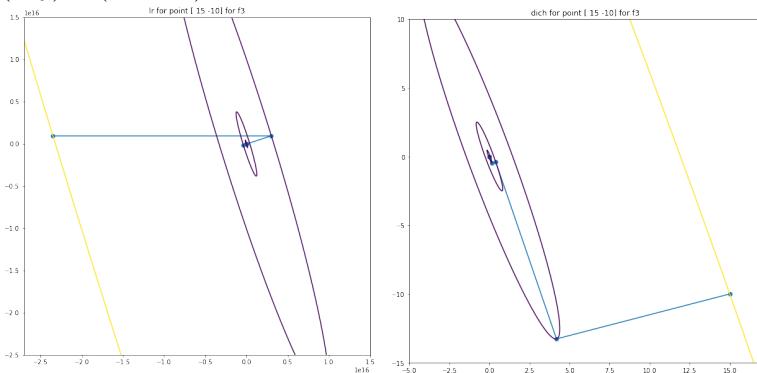
$(x, y) = (21, 14)$



$(x, y) = (-5, -15)$



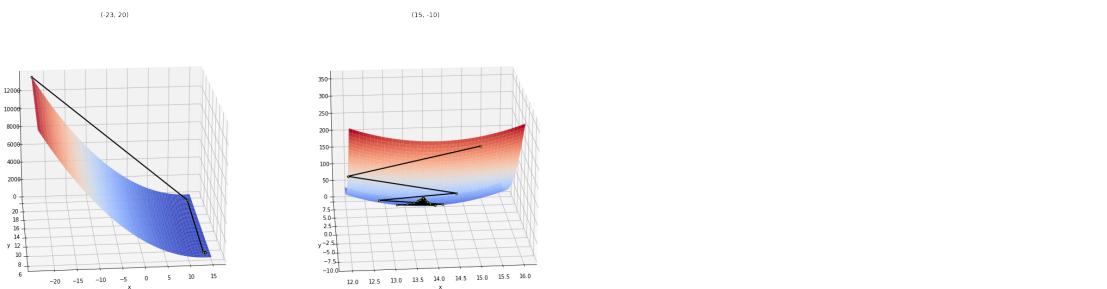
$(x, y) = (15, -10)$



+-----+ f1 (-23, 20) (21, 14) (-5, -15) (15, -10) avg +-----+					
lr grad 71 64 69 64 67.0					
dich f 306 442 510 1836 773.5					
dich grad 9 13 15 54 22.75					
+-----+ f2 (-23, 20) (21, 14) (-5, -15) (15, -10) avg +-----+					
lr grad 162 157 128 156 150.75					
dich f 714 782 884 850 807.5					
dich grad 21 23 26 25 23.75					
+-----+ f3 (-23, 20) (21, 14) (-5, -15) (15, -10) avg +-----+					
lr grad 17 17 17 17 17.0					
dich f 306 918 306 306 459.0					
dich grad 9 27 9 9 13.5					

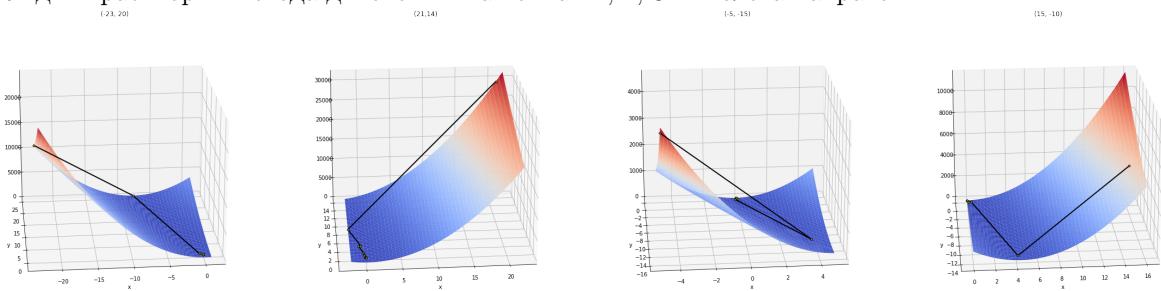
У f_1 количество вызовов функции и градиента достаточно стабильны в точках 1, 2 и 3, но метод градиентного спуска на дихотомии сильно поплыл на точке 4

Для сравнения, графики траекторий для точек 1 и 4:



Очевидно, получившийся из-за неидеального числа обусловленности (а оно =10) и неудачной точки старта зигзаг сильно увеличил число итераций спуска (в примерно 4 раза), как и количество вызовов функции и градиента, причем последнее аж на 1300 раз, что ну очень дорого! f_2 тем временем ведет себя максимально стабильно – хорошее число обусловленности явно этому способствует.

Результаты f_3 выглядят подозрительно, но мы их честно не придумали, вот, например, графики, как выглядят траектории метода дихотомии на точках 1, 2, 3 и 4 слева направо:

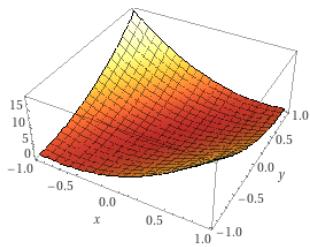


Графики для 1, 3 и 4 имеют очень схожие траектории, при этом ходят по разным направлениям и точкам, это может быть связано с четностью функции. Однако у f_3 не всегда будет такой результат – во второй точке количество итераций в спуске на основе дихотомии сильно увеличивается – одновременно из-за неудачного выбора старта.

Итого из проведенных экспериментов понимаем, что чем ниже число обусловленности, тем стабильнее ведет себя метод на основе дихотомии с разными начальными точками. Про постоянный шаг такое тоже можно сказать, однако здесь он ведет себя относительно стабильнее вне зависимости от обусловленности функции.

- (d) Влияние нормализации на сходимость на примере масштабирования осей плохо обусловленной функции
Рассмотрим следующую функцию (ее число обусловленности =20)

```
def f(X: np.array, x_scale=1, y_scale=1):
    x, y = X[0] * x_scale, X[1] * y_scale
    return 6.30117 * x ** 2 + 4.198827 * y ** 2 - 9.264456 * x * y
```



В таблице ниже представлены значения количества итераций для градиентного спуска на основе дихотомии с начальной точкой $(x, y) = (-0.5, 2.1)$ для различных значений `x_scale` и `y_scale`. (Все, что уходит дальше, чем на 5000 итераций, считаем за `inf`).

$y \setminus x$	0.2	0.4	0.6	0.8	1.0	2.0	3.0	4.0	5.0	10.0	20.0	30.0	50.0
0.2	211	191	311	651	602	336	591	4351	inf	4032	inf	inf	976
0.4	113	92	130	100	210	836	85	413	3233	inf	inf	inf	2948
0.6	144	79	54	122	175	381	839	253	325	inf	1252	inf	696
0.8	204	119	68	103	63	11	17	9	27	inf	712	inf	132
1.0	275	98	68	49	9	11	11	19	7	4851	344	inf	44
2.0	409	59	17	5	6	9	11	9	21	423	2284	inf	inf
3.0	2905	247	398	89	15	8	9	11	7	9	1378	inf	inf
4.0	1850	1656	535	162	265	5	9	9	15	77	1630	790	inf
5.0	inf	1586	1056	691	424	11	10	7	5	13	13	2332	inf
10.0	3472	102	238	258	2007	16	291	131	9	29	169	702	1798
20.0	inf	inf	inf	4328	2904	718	32	68	164	204	77	127	176
30.0	inf	4037	inf	inf	inf	74	516	352	817	304	182	178	184
50.0	inf	inf	inf	3381	2655	inf	4906	74	20	83	272	191	194

Заметим, что функция сходится наиболее быстро для x_scale и y_scale близких к единице, то есть без scaling'a, и чем ближе друг к другу значения x_scale и y_scale , тем меньше scaling влияет на количество необходимых для сходимости итераций. Так как функция состоит исключительно из слагаемых второй степени, scaling, при котором x_scale и y_scale равны, эквивалентен домножению всей функции на константу.

Итак, вспомним как вычисляется число обусловленности функции:

$$cond_f = \sup_X \frac{\|J\| \cdot \|X\|}{\|f(X)\|}$$

для нашей изначальной функции $f(x, y) = \alpha_1 x^2 + \alpha_2 y^2 + \alpha_3 xy$ получаем отскайленную функцию $\approx g(x, y) = \beta(\alpha_1 x^2 + \alpha_2 y^2 + \alpha_3 xy)$, и тогда якобиан g будет равен β - якобиану f . Тогда

$$cond_g = \sup_X \frac{\|\beta \cdot J\| \cdot \|X\|}{\|\beta \cdot f(X)\|} = cond_f$$

а значит, число обусловленности функции в данном случае почти не меняется, поэтому и количество итераций изменяется не так сильно.

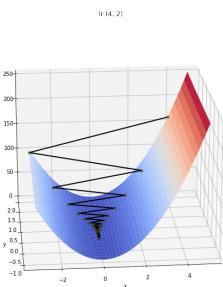
Для объяснения резкого увеличения числа необходимых методу итераций в данном случае вспомним о том, что число обусловленности функции как раз отражает, насколько чувствителен ее результат к небольшим изменениям аргумента. При масштабировании функции по одной оси, изменяется форма функции и ее чувствительность. В частности, масштабирование одной оси может растянуть или сжать функцию вдоль этой оси, что и приводит к изменениям числа обусловленности, меняя ее чувствительность к небольшим изменениям аргумента, и (об этом будем говорить в следующем пункте) может повлиять на сходимость.

Разумеется, не всегда scaling будет ухудшать сходимость функции: вот пример функции (с числом обусловленности все так же = 20), когда существует scaling = (0.4, 1.8), при котором значительно уменьшается количество итераций метода на основе дихотомии (и по результатам предыдущий экспериментов логично предположить, что и с постоянным шагом итераций будет значительно меньше).

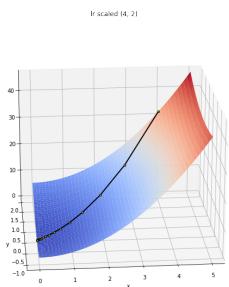
```
def f(X: np.array, x_scale=1, y_scale=1):
    x, y = X[0] * x_scale, X[1] * y_scale
    return 10 * x ** 2 + 0.5 * y ** 2
```

Здесь значение начальной точки $(x, y) = (4, 2)$

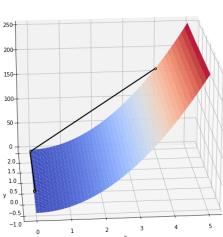
$y \setminus x$	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0	2.2	2.4	2.6	2.8
0.2	221	371	445	933	1967	1173	1081	485	inf	inf	3875	inf	inf	inf
0.4	60	102	207	219	342	247	461	1099	237	263	149	2179	3289	3573
0.6	26	45	64	58	78	115	443	773	81	73	191	293	479	507
0.8	12	26	60	108	164	207	249	435	51	47	113	69	105	115
1.0	8	9	9	7	7	7	7	6	9	9	7	27	39	41
1.2	11	9	9	9	7	7	7	5	7	9	7	17	21	23
1.4	15	8	9	9	9	8	9	7	7	13	7	11	13	13
1.6	19	6	9	9	9	9	8	7	7	11	6	13	13	13
1.8	21	3	9	9	9	9	9	7	8	7	6	8	9	9
2.0	21	7	8	9	9	9	9	8	7	7	6	7	15	8
2.2	21	9	7	9	9	9	9	9	9	9	9	7	11	7
2.4	21	12	6	9	10	9	9	9	9	9	8	7	9	7
2.6	21	15	5	9	10	10	9	9	9	9	8	10	9	7
2.8	19	17	5	8	10	10	10	9	9	9	9	9	8	7



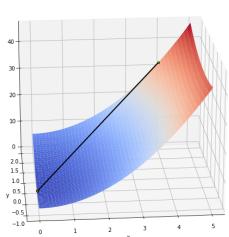
lr (4, 2)



lr scaled (4, 2)



dch (4, 2)



dch scaled (4, 2)

Это мы и видим, дихотомия работает на 6 итераций меньше, а спуск с постоянным шагом снизил количество итераций со 104х до 35и, то есть и там и там scaling позволил сократить количество необходимых итераций в 3 раза. Это происходит во многом потому, что после scaling'а число обусловленности функции $= \frac{81}{80} \approx 1$, мы растягиваем обе оси так, что градиенты функции более плавно изменяются в зависимости от аргумента, что позволяет градиенту лучше соответствовать направлению минимума. Напротив, у функций с резкими или прерывистыми изменениями градиентов (как и было изначально) градиентный спуск может застрять в локальных минимумах или колебаться зигзагом, что приведет к более медленной сходимости или нестабильности.

- Реализация генератора случайных квадратичных функций от n переменных с числом обусловленности k . Генерируем матрицу n на n из нулей – S , и заполняем ее диагональные элементы числами от 1 до k , причем так, что значения 1 и k обязательно достигаются. В таком случае ее число обусловленности – отношение максимального и минимального собственного числа – как раз = k . Матрицу ответа A строим по следующему правилу (чтобы функция была поинтереснее): $A = I.T \cdot S \cdot I$ где I – рандомная невырожденная матрица (это не влияет собственные числа матрицы). В качестве невырожденной матрицы генерируем ортогональную (ее определитель = ± 1 по определению), и тогда сгенерированная квадратичная функция – $f(X) = X.T \cdot A \cdot X$. Заметим, что уже на этом этапе мы можем достать из нее производную – $f'(X) = X.T \cdot A$, ее тоже возвращаем (будем использовать ее в задачах 6, 7)

```
def generate_matrix(dimension: int, condition: float):
```

```

S = np.zeros((dimension, dimension))
S[:dimension, :dimension] = np.diag(np.linspace(1, condition, dimension))
I = np.random.randn(dimension, dimension)
I = scipy.linalg.orth(I)
A = I.T @ S @ I

def generate_quadratic_function_by_cond_number(amount_of_variables: int, condition: float):
    A = generate_matrix(amount_of_variables, condition)

    def f(X: np.array):
        return X.T @ A @ X

    def df(X: np.array):
        return 2 * X.T @ A

    return f, df

```

6. Исследуйте зависимость числа итераций $T(n, k)$, необходимых градиентному спуску для сходимости в зависимости от размерности пространства $2 \leq n \leq 10^3$ и числа обусловленности оптимизируемой функции $1 \leq k \leq 10^3$.

Для получения данных просто проходимся циклом по n и k в данных диапазонах, генерируем для данных n и k функцию от n переменных с числом обусловленности k используя функцию из задания 5, а та же начальную точку X , и запускаем на данной функции градиентный спуск на основе дихотомии (слегка видоизмененный, так как нам не нужны данные о последовательных приближениях точки X или полученном значении, нам достаточно получить количество итераций градиентного спуска)

```

def dich_iters(f, df, X: np.array, eps=1e-5):
    grad = df(X)
    a, b = 0, 1
    while abs(a - b) / 2 > eps:
        alpha1 = (a + b - eps) / 2
        alpha2 = (a + b + eps) / 2
        X1, X2 = X - alpha1 * grad, X - alpha2 * grad
        if f(X1) < f(X2):
            b = alpha2
        else:
            a = alpha1
    return (a + b) / 2

def grad_descent_with_dich_iters(f, df, X: np.array, max_iter=30000, eps=1e-5):
    iter = 0
    while True:
        grad = df(X)
        if np.linalg.norm(grad) < eps:
            break
        lr = bivariate_dichotomy(f, df, X)
        X = X - lr * grad
        iter += 1
        if iter > max_iter:
            break
    return iter

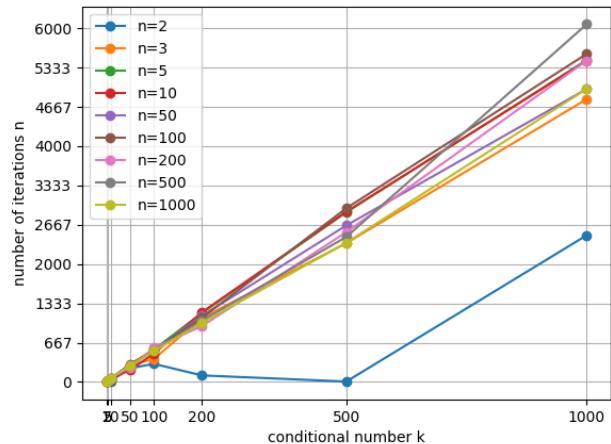
n_vals = [2, 3, 5, 10, 50, 100, 200, 500, 1000]
k_vals = [1, 5, 10, 50, 100, 200, 500, 1000]

def make_table():
    table = []
    for n in n_vals:
        X = np.random.rand(n)
        small = []
        for k in k_vals:
            f, df = generate_quadratic_function_by_cond_number(n, k)
            iterations_amount = grad_descent_with_dich_iters(f, df, X)
            small.append(np.hstack((n, k, iterations_amount)))
        table.append(np.array(small))
    return np.array(table)

table = make_table()

```

Таблица и график зависимости количества итераций от n и k ниже (таблица совмещена с таблицей для №7)



n	k	iterations	iterations average	n	k	iterations	iterations average
2	1	1	2.0	50	1	2	2.0
2	5	5	15.4	50	5	27	28.3
2	10	15	11.0	50	10	58	54.8
2	50	232	9.6	50	50	273	264.6
2	100	307	22.2	50	100	543	554.0
2	200	111	123.8	50	200	1123	1089.8
2	500	7	163.8	50	500	2655	2790.0
2	1000	2489	2441.0	50	1000	4971	5340.2
3	1	1	1.0	100	1	2	2.0
3	5	25	24.2	100	5	27	29.0
3	10	50	51.4	100	10	55	56.1
3	50	280	261.8	100	50	301	268.8
3	100	391	505.9	100	100	543	538.3
3	200	1086	1019.1	100	200	1089	1003.3
3	500	2354	2331.5	100	500	2947	2707.8
3	1000	4799	4909.1	100	1000	5567	5615.1
5	1	2	2.0	200	1	2	2.0
5	5	29	27.0	200	5	29	29.4
5	10	57	53.5	200	10	59	57.1
5	50	258	262.0	200	50	249	266.6
5	100	551	514.6	200	100	579	520.4
5	200	1169	1062.9	200	200	949	1017.6
5	500	2879	2630.4	200	500	2541	2628.4
5	1000	5462	4856.3	200	1000	5453	5342.6
10	1	2	2.0	500	1	2	2.0
10	5	27	27.6	500	5	30	30.5
10	10	53	53.0	500	10	61	59.5
10	50	201	276.3	500	50	279	275.2
10	100	481	523.5	500	100	531	540.6
10	200	1183	941.0	500	200	1041	1009.4
10	500	2889	2560.5	500	500	2461	2389.0
10	1000	5450	5053.6	500	1000	6069	5417.0
n		iterations	iterations average				
1000	1	2	2.0				
1000	5	31	31.3				
1000	10	61	61.2				
1000	50	277	287.0				
1000	100	525	552.4				
1000	200	1005	1091.2				
1000	500	2361	2666.6				
1000	1000	4973	5351.6				

По оси Ох на графике величина condition number, по оси Оу - количество итераций, необходимых градиентному спуску, чтобы сойтись. На графике изображена серия из 9 экспериментов, соответствующих количеству переменных в генерированной функции. По графику видно, что количество итераций в целом возрастает с увеличением числа обусловленности

для любого количества переменных; однако они не зависят друг от друга напрямую. На количество итераций может влиять и множество других факторов, например, другие свойства функции или выбор начальной точки.

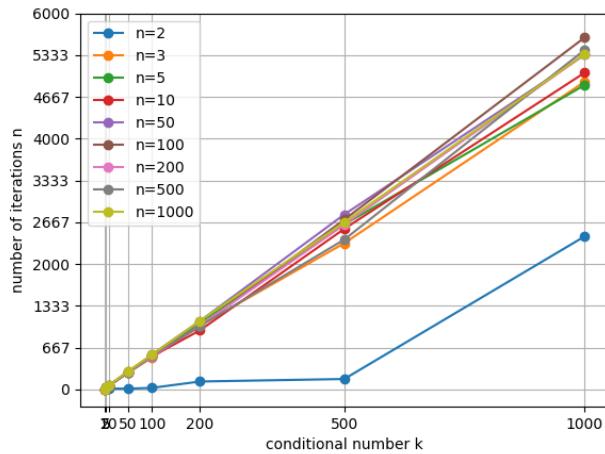
7. Для получения более корректных результатов проведите множественный эксперимент и усредните полученные значения числа итераций

Все то же самое, что и в №6, за исключением того, что здесь мы для каждого фиксированного n и k генерируем 10 различных функций и сохраняем их среднее арифметическое количество итераций для получения более корректных данных.

```
def make_table_avg(tries=10):
    table = []
    for n in n_vals:
        X = np.random.rand(n)
        small = []
        for k in k_vals:
            iterations_amount = 0
            for _ in range(tries):
                f, df = generate_quadratic_function_by_cond_number(n, k)
                iterations_amount += grad_descent_with_dich_iters(f, df, X)
            small.append(np.hstack((n, k, iterations_amount / tries)))
        table.append(np.array(small))
    return np.array(table)

table = make_table_avg()
```

Таблица приведена выше, график:



В целом, отличий от графика из №6 почти никаких, и имеющиеся вызваны скорее удачным рандомом стартовой точки и функции в №6.

8. Реализация одномерного поиска с учетом условий Вольфе

Идея метода градиентного спуска с учетом условий Вольфе так же представляет собой версию градиентного спуска с условиями на шаг в каждой итерации, условия Вольфе позволяют сделать шаг "достаточно, но не избыточно точным". Как выглядят сами условия Вольфе: для f – нашей функции, x_k – текущего приближения, p_k – того направления, вдоль которого мы собираемся делать шаг, α – длины этого шага c_1 и c_2 , $c_1 < c_2$ – некоторых константы, которые находятся в диапазоне от 0 до 1, и $\nabla f(x_k)p_k < 0$

- (a) Armijo rule: $f(x_k + \alpha p_k) \leq f(x_k) + c_1 \alpha \nabla f(x_k)^T p_k$ – отвечает за то, чтобы выбранный шаг приближал f к минимуму достаточно – $f(x_k + \alpha p_k)$ и $f(x_k)$ достаточно отличались друг от друга
- (b) curvature: $\nabla f(x_k + \alpha p_k)^T p_k \geq c_2 \nabla f(x_k)^T p_k$ – мы знаем, что $p_k \cdot \nabla f(x) < 0$ так как p_k – направление убывания. Так как мы ищем минимум ($x: \nabla f(x) = 0$) нам не нужно уходить в следующей итерации в $x_{k+1} = x_k + \alpha p_k$, если градиент в направлении p_k ($p_k \cdot \nabla f(x)$) остается таким же маленьким – нам нужно его увеличивать.

В методе градиентного спуска наше направление p_k равняется градиенту $\nabla f(x_k)$

Условия Вольфе позволяют нам выбрать оптимальный шаг α , и раз нам подходит любое значение, удовлетворяющее условиям, давайте выбирать наибольшее из возможных, чтобы ускорить градиентный спуск.

В данной реализации на каждой итерации мы ставим изначально значение $\alpha = 1$, и уменьшаем его, пока оно не начнет удовлетворять условиям

```

def check_wolfe(f, X: np.array, grad, alpha, c1, c2): # в град. спуске направление = -df(X);
    grad_sq = grad @ grad
    X_next = X + alpha * -grad
    if f(X_next) > f(X) - c1 * alpha * grad_sq:
        return False
    if grad @ grad_f(f, X_next) > c2 * grad_sq:
        return False
    return True

def gradient_descent_with_wolfe(f, X: np.array, eps=1e-5, c1=1e-10, c2=0.9, wolfe_limit=1e5):
    history = []
    history.append(np.hstack((X, f(X))))
    iterations = 0
    while True:
        grad = grad_f(f, X)
        alpha = 1
        i = 0
        while not check_wolfe(f, X, grad, alpha, c1, c2):
            i += 1
            if i > wolfe_limit:
                alpha = 1
                break
            alpha /= 2
        step = alpha * grad
        if np.linalg.norm(step) < eps:
            break
        iterations += 1
        X = X - step
        history.append(np.hstack((X, f(X))))
    return (np.array(history), iterations)

```

Обратимся к уже известным нам функциям, на основе которых рассматривались метод дихотомии и метод градиентного спуска со статичным learning rate, и понаблюдаем за работой градиентного спуска с условиями Вольфе на них. Подопытные функции:

```

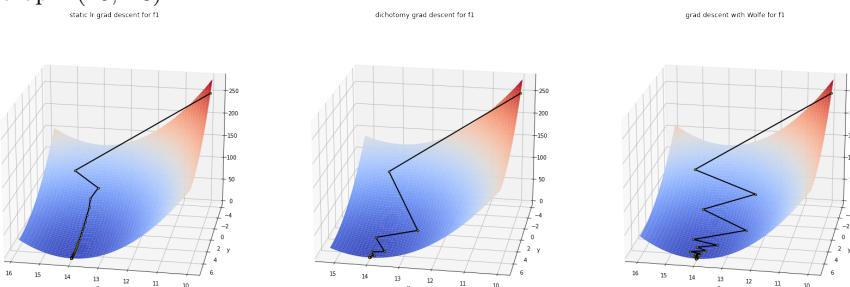
def f1(X: np.array) -> float:
    x, y = X[0], X[1]
    return 10 * (x - 14) ** 2 + (y - 7) ** 2

def f2(X: np.array) -> float:
    x, y = X[0], X[1]
    return 0.5 * x ** 2 + 3 * y ** 2 + x*y

def f3(X: np.array) -> float:
    x, y = X[0], X[1]
    return 45.51 * x ** 2 + 28.42 * x * y + 4.98 * y ** 2

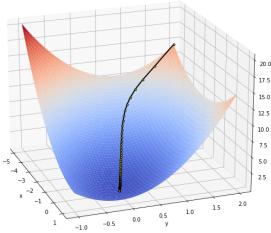
```

f_1 , старт: $(10, -3)$

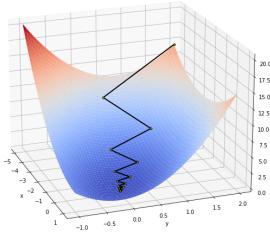


f_2 старт: $(-5, 2)$

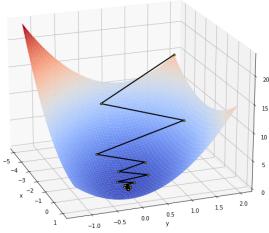
static lr grad descent for f2



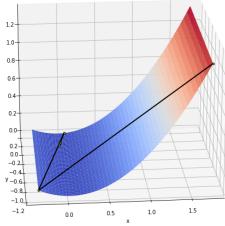
dichotomy grad descent for f2



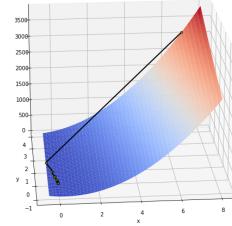
grad descent with Wolfe for f2

 f_3 старт: $(7, 4)$

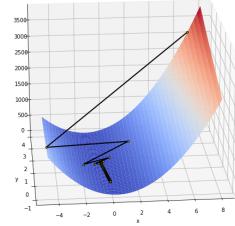
static lr grad descent for f3



dichotomy grad descent for f3



grad descent with Wolfe for f3



Здесь представлено количество итераций, необходимых для каждого метода и каждой функции до момента, когда x_{n+1} отличается от x_n менее, чем на эпсилон

iterations	f_1	f_2	f_3
lr	92	285	-
dichotomy	27	47	43
wolfe	50	43	385

Можно сказать, что дихотомия в среднем все еще обгоняет оба метода по количеству итераций, и ведет себя в этом плане гораздо стабильнее, а вот по сравнению с методом градиентного спуска с постоянным шагом Вольфе может работать значительно быстрее (видно на примерах 1 и 2, в 3 случае градиентный спуск с постоянным шагом не сопрался). Это вызвано тем, что в дихотомии мы подбираем шаг как бы бинпоиском, а при условиях Вольфе так или иначе просто делим на 2, соответственно в методе дихотомии шаг выбрать получается точнее.