Санкт-Петербургский национальный исследовательский
университет информационных технологий, механики и оптики

ФИТиП

ПМИ

Лабораторная работа #1

Методы нулевого порядка

Ученики группы М3235:

Ефименко В.В.
Клементьев Т.А.
Лыженков А.Д.

Санкт-Петербург
2024 г.

1 Метод градиентного спуска с постоянным шагом

Суть метода - нахождение локального минимума (в нашем случае) функции с помощью движения вдоль антиградиента.

В начале выбирается стартовая точка работы алгоритма. Мы будем рассматривать функции двух переменных, а точки будем обозначать как (x, y). Также на данном этапе задаем желаемую точность работы ϵ.

Теперь мы планируем каким-то способом дойти от начальной точки до точки минимума. Для этого на каждой итерации будем находить следующую точку по формуле: $x_k = x_{k-1} - learning\_rate * gradient(x_{k-1})$

Градиент в данном случае - это направление наискорейшего увеличения функции. Именно поэтому мы идем вдоль антиградиента - в направлении наискорейшего спуска. Его мы считаем для каждой точки отдельно. $learning\_rate$ в данном случае постоянный. Он определяет длину шага вдоль выбранного нами направления.

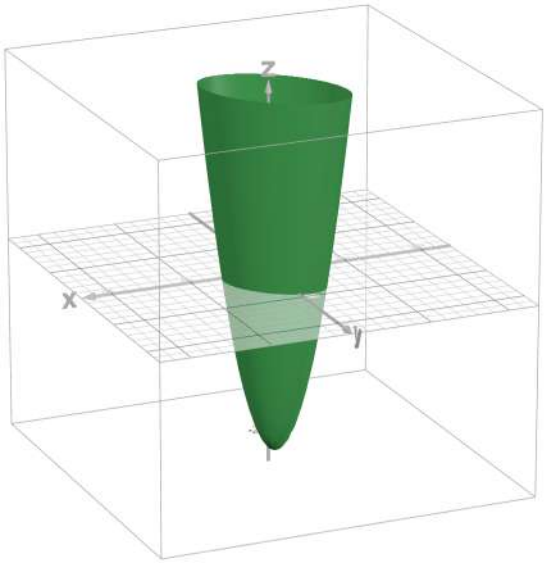
Существует несколько различных условий останова. Мы будем использовать условие малого расстояния между точками x_k и x_{k-1}. Для этого нам и понадобится заданное в начале ϵ.

Теперь давайте посмотрим работу данного метода на различных квадратичных функциях двух переменных.

Зададим $learning\_rate = 0.5, \epsilon = 10^{-3}$

1.1 $f(x, y) = x^2 + x * y + y^2 - 6 * x - 9 * y$

Возьмем квадратичную функцию, график которой представляет собой чашку.

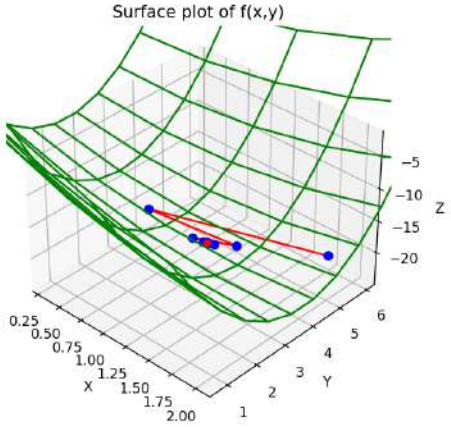


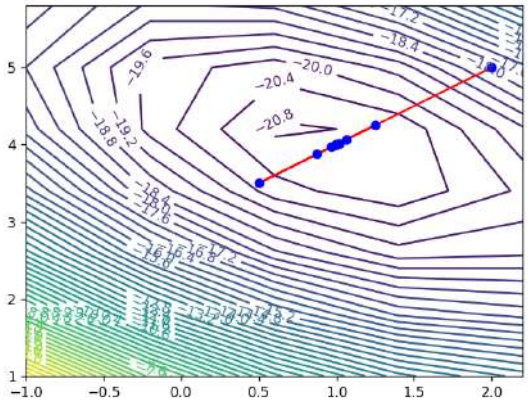
Минимум данной функции находится в точке (1; 4), а значение в этой точке равно -21.

Попробуем запуститься от разных точек и понаблюдаем за результатами:

- (x; y) = (2, 5)

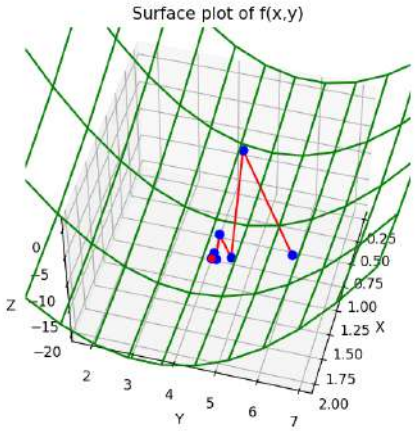
```
function: x**2 + x*y + y**2 - 6*x - 9*y
start points: x = 2, y = 5
learning_rate: 0.5, eps: 0.001

x: 1.000244140625, y: 4.000244140625
f: -20.999999284744263, iterations: 12
work time: 0.0 ms
```

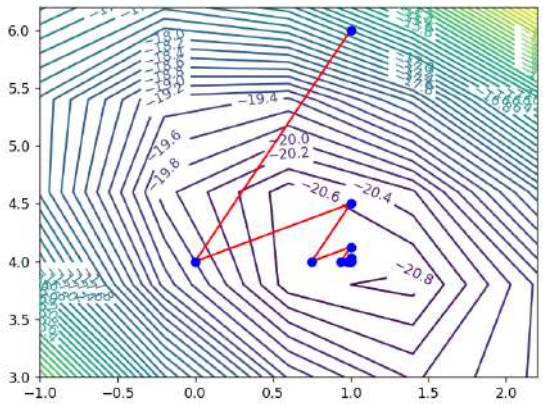




- (x; y) = (1; 6)

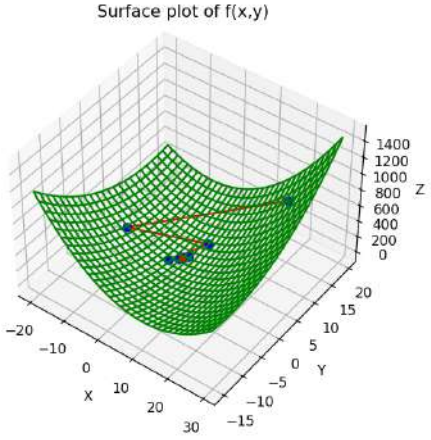
```
function: x**2 + x*y + y**2 - 6*x - 9*y
start points: x = 1, y = 6
learning_rate: 0.5, eps: 0.001

x: 1.0, y: 4.00048828125
f: -20.999999046325684, iterations: 12
work time: 0.0 ms
```

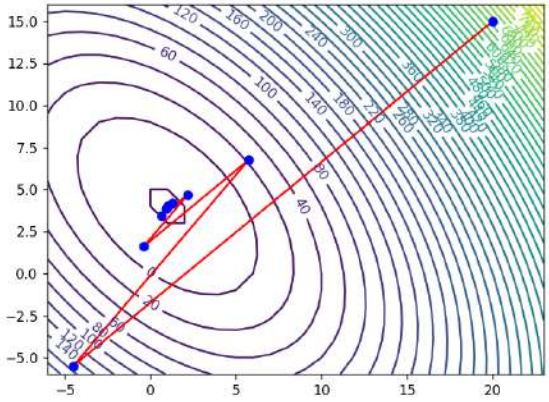




- (x; y) = (20; 15)

```
function: x**2 + x*y + y**2 - 6*x - 9*y
start points: x = 20, y = 15
learning_rate: 0.5, eps: 0.001

x: 0.9996643066406625, y: 3.999420166015625
f: -20.999997425824404, iterations: 15
work time: 0.0 ms
```





- (x; y) = (1000; 1000)

```
function: x**2 + x*y + y**2 - 6*x - 9*y
start points: x = 1000, y = 1000
learning_rate: 0.5, eps: 0.001

x: 1.0002381801605225, y: 4.000237464904785
f: -20.999999321284804, iterations: 22
work time: 0.0 ms
```

Какой вывод можно сделать из полученных результатов? Можно заметить, что количество итераций алгоритма растет гораздо медленнее, чем расстояние между стартовой точкой и точкой минимума. Тут можно предположить, что причиной этому является довольно высокий антиградиент (скорость уменьшения функции). А отсюда можно предположить, что на функциях, медленно убывающих, алгоритм будет работать медленно из-за маленького антиградиента. Давайте проверим наши предположения на какой-нибудь медленно убывающей функции.

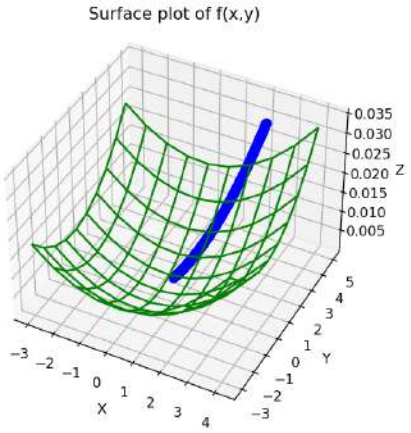
1.2 $f(x, y) = x^2/1000 + y^2/1000$

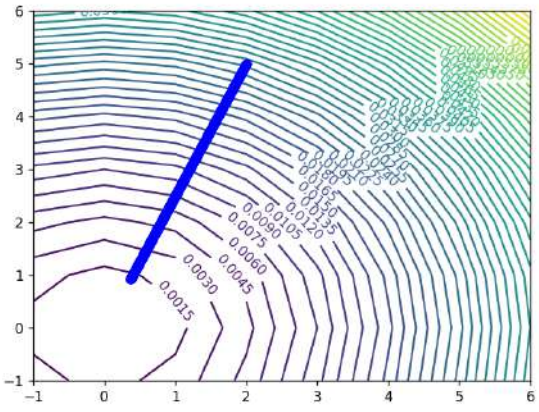
График данной функции так же является чашкой, но убывает гораздо медленне, чем предыдущая.

Минимум функции находится в точке (0; 0) и равен 0.

- (x; y) = (2, 5)

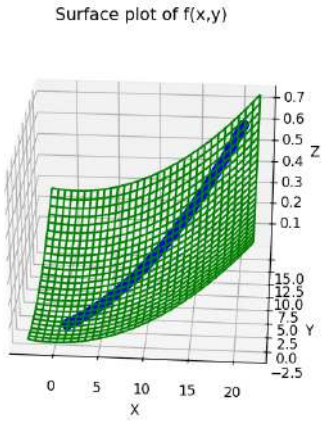
```
function: x**2 / 1000 + y**2 / 1000
start points: x = 2, y = 5
learning_rate: 0.5, eps: 0.001

x: 0.37131857917988376, y: 0.9282964479497057
f: 0.0010016140089240473, iterations: 1683
work time: 2.9990673065185547 ms
```

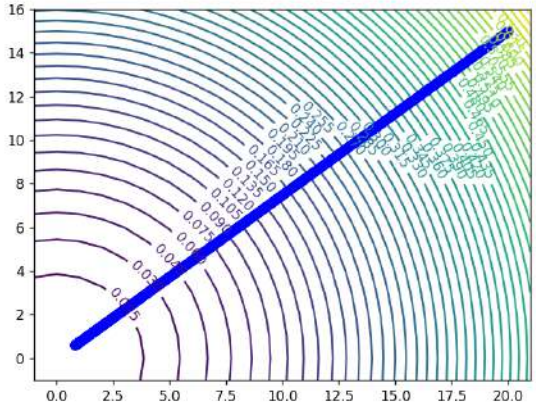




- (x; y) = (20; 15)

```
function: x**2 / 1000 + y**2 / 1000
start points: x = 20, y = 15
learning_rate: 0.5, eps: 0.001

x: 0.7994128167127444, y: 0.5995596125345606
f: 0.0010005326452650813, iterations: 3218
work time: 5.001306533813477 ms
```





- (x; y) = (1000; 1000)

```
function: x**2 / 1000 + y**2 / 1000
start points: x = 1000, y = 1000
learning_rate: 0.5, eps: 0.001

x: 0.7068953528357056, y: 0.7068953528357056
f: 0.0010014038860897269, iterations: 7251
work time: 12.619256973266602 ms
```

Как мы видим, количество итераций алгоритма увеличилось в сотни раз. А это значит, что мы были правы, предположив, что на медленно убывающих функциях алгоритм будет работать медленно. Понятно, почему это происходит. Все дело в шаге. Если для предыдущей функции шаг был приемлем (хоть он и мог проскочить точку минимума, алгоритм сходился к ней), то сейчас шаг настолько маленький, что алгоритму приходится тратить основные усилия на то, чтобы просто преодолеть этот путь от стартовой точки до конечной. Это можно заметить и на графике: синих точек настолько много, что красной траектории движения просто не видно. К тому же, точность тоже ухудшилась и довольно сильно. Конечно, можно поменять критерий останова, например, на малое приращение функции. Но тогда количество итераций возрастет многократно, так как каждый шаг теперь двигает точку не более, чем на ϵ, а это уже кажется неразумным.

Хорошо, мы разобрались, что происходит нормально и медленно убывающей функцией. Давайте теперь рассмотрим быстро убывающую функцию.

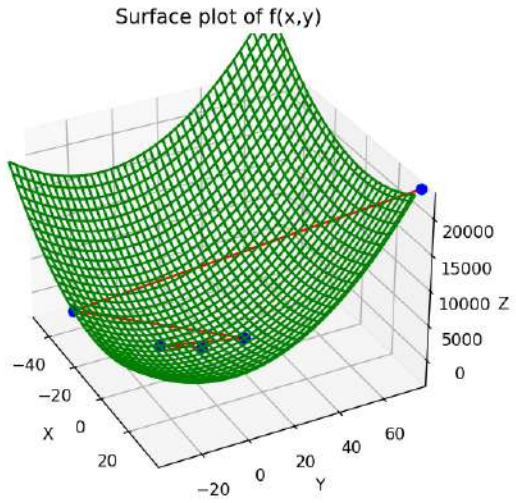
1.3 $f(x, y) = x^2 * 3 + y^2 * 3$

Данная функция убывает в несколько раз быстрее, чем первая. Посомтрим на работу алгоритма.

И что мы тут видим? Результат работы алгоритма уходит в бесконечность. Как такое могло произойти? Все дело опять в неверно выбранном шаге. Только если раньше шаг был слишком маленьким, то сейчас он слишком большой. Настолько, что алгоритм не просто проскакивает точку минимума, но и с каждым разом поднимается все выше.

То есть результаты на каждом шаге и график получаются примерно такие (начало на графике в нижней точке):

```
step 1: x = 2, y = 5
step 2: x = -4.0, y = -10.0
step 3: x = 8.0, y = 20.0
step 4: x = -16.0, y = -40.0
step 5: x = 32.0, y = 80.0
step 6: x = -64.0, y = -160.0
step 7: x = 128.0, y = 320.0
```



Из данных и графика видно, что метод расходится на данной функции. И это не потому, что метод работает неправильно. Это значит, что для каждой функции нужно подбирать приемлимый шаг (делать это вручную).

Из плюсов метода можно отметить довольно хорошую точность и скорость (при условии оптимально выбранного шага). Скорость работы обеспечивает малое количество вызовов основной функции и ее градиентов - по 1 разу за итерацию. А если правильно подобрать шаг, то он будет, во-первых,

достаточным, чтобы быстро проходить большие расстояния, а во-вторых, алгоритм не будет расходиться и давать приемлемую точность.

Из минусов - для каждой функции нужно отдельно высчитывать оптимальный шаг, что может быть трудной задачей, так как при выборе необходимо соблюсти идеальный баланс между временем работы и сходимостью алгоритма.

2 Метод градиентного спуска на основе одномерного метода поиска (золотое сечение)

Давайте немного модифицируем алгоритм грандиентного спуска. Вместо постоянного шага $learning\_rate$ мы теперь на каждой итерации будем выбирать оптимальный шаг из отрезка $[0; learning\_rate]$. Делать это мы будем с помощью метода одномерного поиска - золотого сечения.

Для этого на каждой итерации наш отрезок $[left; right]$ (который, напомним, изначально задается как $[0; learning\_rate]$) будем разбивать на 3 отрезка введением двух дополнительных точек x_1 (ближе к левому концу) и x_2 (ближе к правому концу), которые будут делить внутренние отрезки в отношении золотого сечения. Теперь левый (правый) конец отрезка сдвигаем в x_1 (x_2), если $f(x_1) >= f(x_2)$ ($f(x_2) >= f(x_1)$). Получили новый отрезок с уже отмеченной одной внутренней точкой. Находим еще одну внутреннюю точку по правилам золотого сечения и повторяем процесс до достижение заданной нами точности.

Поймем, как изменился наш изначалный алгоритм с постоянным шагом.

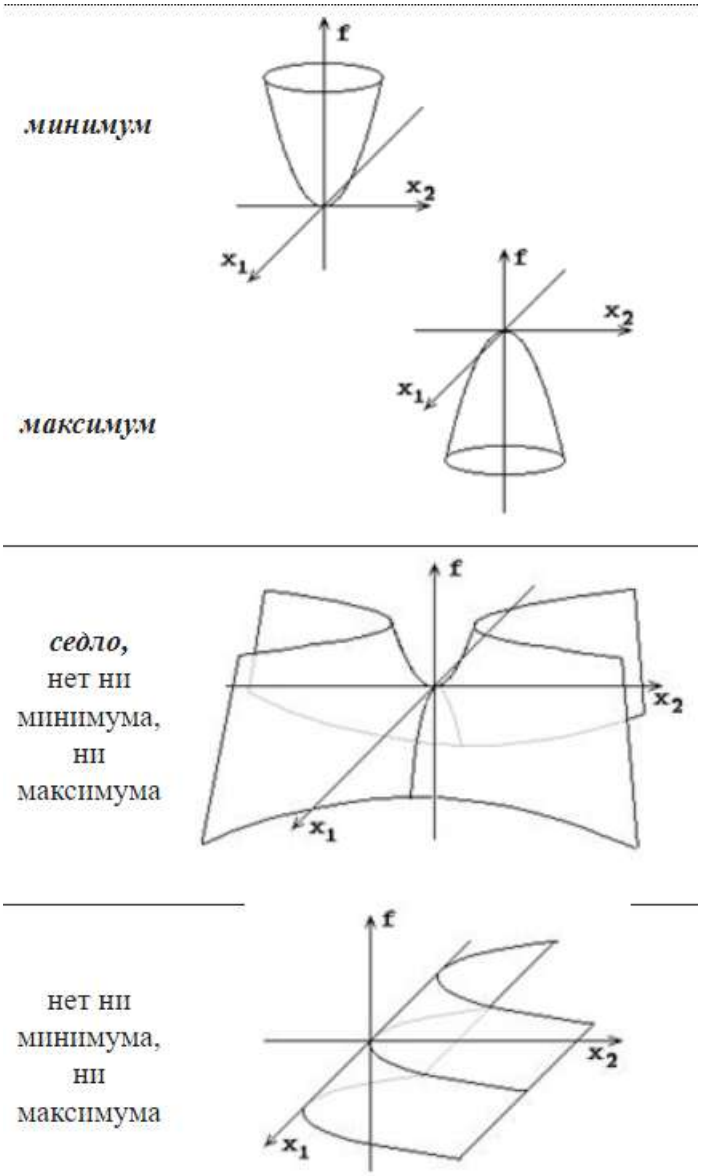
Так как сейчас шаг не фиксированный, мы перейдем в какое-то состояние из отрезка $[x_k; x_k - learning\_rate * gradient(x_{k-1})]$.

Если функция на заданном отрезке монотонно (нестрого) убывает, то алгоритм всегда будет переходить в точку $x_k - learning\_rate * gradient(x_{k-1})$, так как она будет оптимальной на дрнном шаге. Получается, пока мы находимся на монотонных убывающих участках функции, наш алгоритм ведет себя так же, как и градиентный спуск с постоянным шагом.

Если же функция ведет себя по разному на отрезке, то ситуация становится неочевидной, и алгоритм золотого сечения постарается выдать нам наиболее оптимальную точку для перехода.

Так как мы рассматриваем квадратичные функции двух переменных, давайте обратим на них дополнительное внимание.

Квадратичные функции имеют вид:



А это означает, что на нащем отрезке $[x_k; x_k - learning\_rate * gradient(x_{k-1})]$ функция всегда монотонно возрастает/убывает, кроме точки экстремума. Только если в наш отрезок попадает точка экстремума, функция на отрезке не будет монотонно убывать/возрастать. А это значит, что на всей области определения функции, кроме окрестности минимума, алгоритм с постоянным шагом и алгоритм с одномерным поиском будут вести себя одинаково, а именно - использовать в качестве шага значение $learning\_rate$.

Теперь посмотрим на окрестность точки минимума (если точка минимума

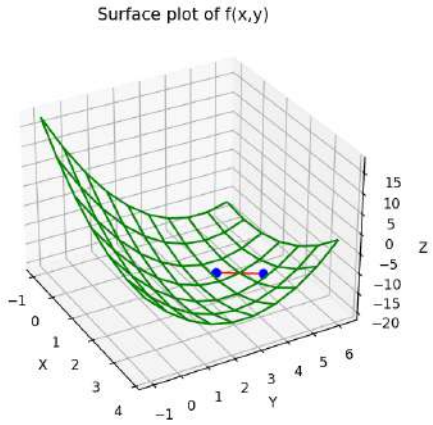
попадает в отрезок $[x_k; x_k - learning\_rate * gradient(x_{k-1})]$). В отличие от метода с постоянным шагом, где нам нужно убедиться в сходимости и подобрать правильный $learning\_rate$, алгоритм одномерного поиска сразу выдаст нам точку минимума (с погрешностью в ϵ), так как именно эта точка будет являться оптимальной на данной итерации.

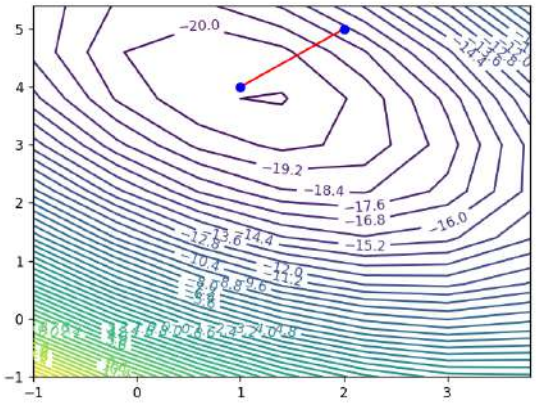
В случае, когда точки минимума нет, данный метод расходится, как и метод с постоянным шагом. Оно и понятно, ведь алгоритм каждый раз будет спускаться все ниже по функции до бесконечности.

2.1 $f(x, y) = x^2 + x * y + y^2 - 6 * x - 9 * y$

- (x; y) = (2, 5)

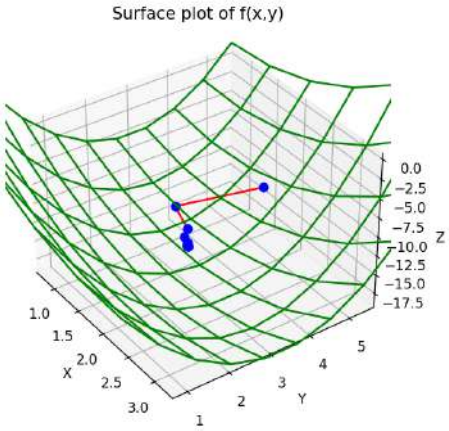
```
function: x**2 + x*y + y**2 - 6*x - 9*y
start points: x = 2, y = 5
learning_rate: golden(0.5), eps: 0.001

x: 1.0001132759634463, y: 4.000113275963447
f: -20.999999961505665, iterations: 1
work time: 0.0 ms
```

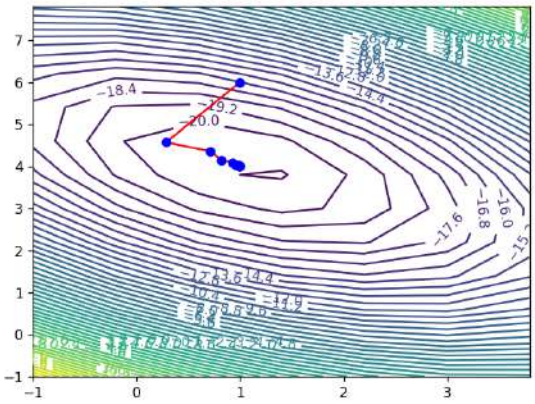




- (x; y) = (1; 6)

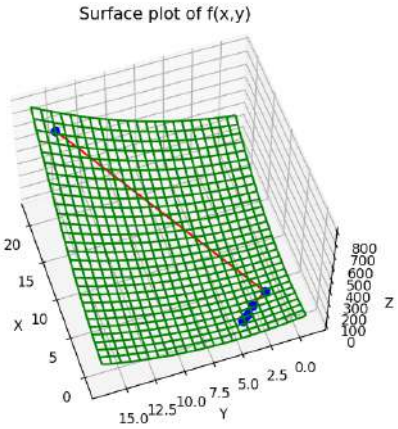
```
function: x**2 + x*y + y**2 - 6*x - 9*y
start points: x = 1, y = 6
learning_rate: golden(0.5), eps: 0.001

x: 0.998869095786297, y: 4.001402220636029
f: -20.999998340610176, iterations: 10
work time: 0.0 ms
```

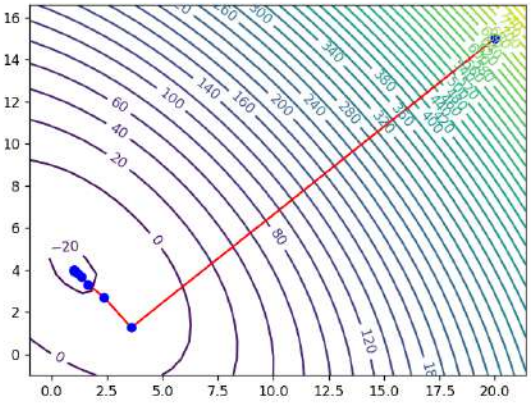




- (x; y) = (20; 15)

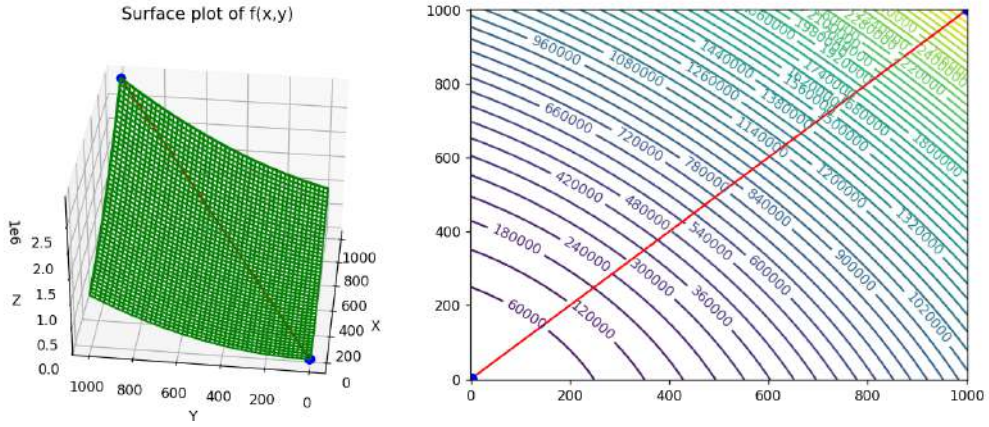
```
function: x**2 + x*y + y**2 - 6*x - 9*y
start points: x = 20, y = 15
learning_rate: golden(0.5), eps: 0.001

x: 1.0013451550312298, y: 3.9987193459135617
f: -20.99999827316134, iterations: 12
work time: 0.0 ms
```





- (x; y) = (1000; 1000)

```
function: x**2 + x*y + y**2 - 6*x - 9*y
start points: x = 1000, y = 1000
learning_rate: golden(0.5), eps: 0.001

x: 1.0010932393443857, y: 3.999121177569151
f: -20.99999899326213, iterations: 11
work time: 0.0 ms
```

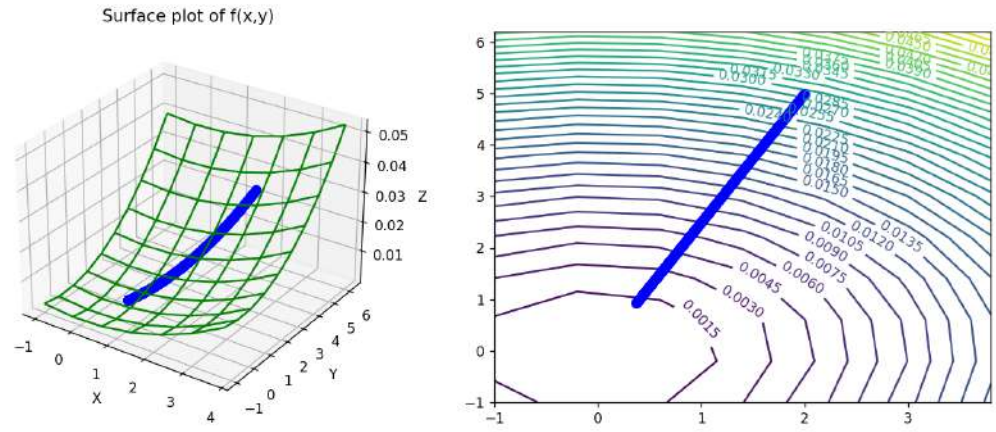


Заметим, что первый случай выполнился за 1 итерацию, так как точка минимума лежала на нашем отрезке шага. В остальных случаях метод работает быстрее метода с постоянным шагом из-за того, что он не проскакивает мимо точки минимума, а сразу попадает в нее.

2.2 $f(x, y) = x^2/1000 + y^2/1000$

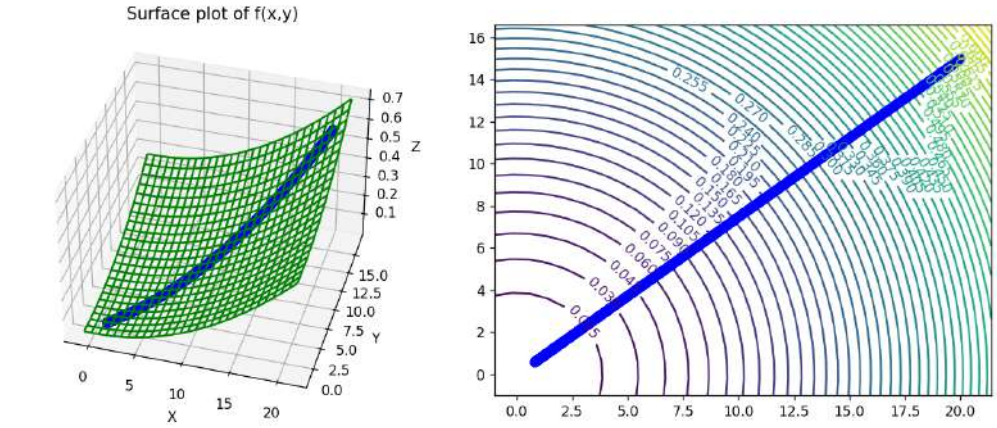
- (x; y) = (2, 5)

```
function: x**2 / 1000 + y**2 / 1000
start points: x = 2, y = 5
learning_rate: golden(0.5), eps: 0.001

x: 0.37154784019667497, y: 0.9288696004916848
f: 0.001000846532272396, iterations: 1684
work time: 22.668838500976562 ms
```



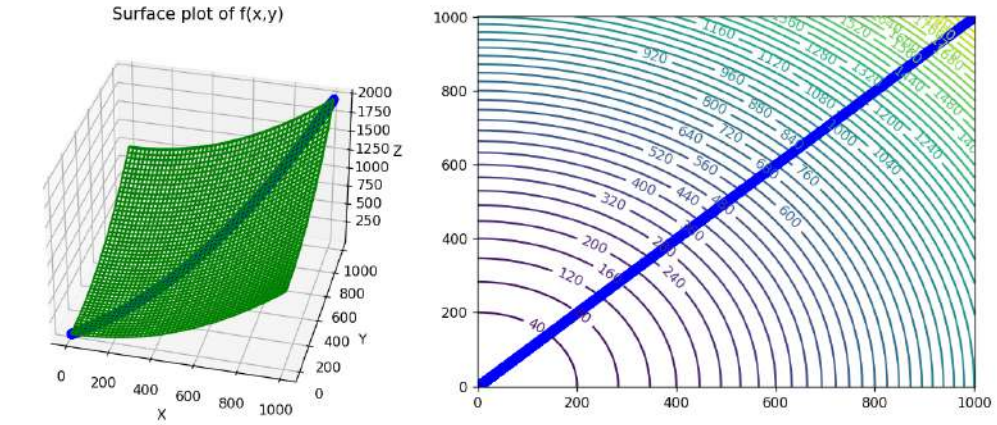
- (x; y) = (20; 15)

```
function: x**2 / 1000 + y**2 / 1000
start points: x = 20, y = 15
learning_rate: golden(0.5), eps: 0.001

x: 0.8002864851114545, y: 0.6002148638335942
f: 0.0010007163410188265, iterations: 3220
work time: 34.70444679260254 ms
```



- (x; y) = (1000; 1000)

```
function: x**2 / 1000 + y**2 / 1000
start points: x = 1000, y = 1000
learning_rate: golden(0.5), eps: 0.001

x: 0.7075802643609245, y: 0.7075802643609245
f: 0.0010013396610261515, iterations: 7257
work time: 75.07157325744629 ms
```



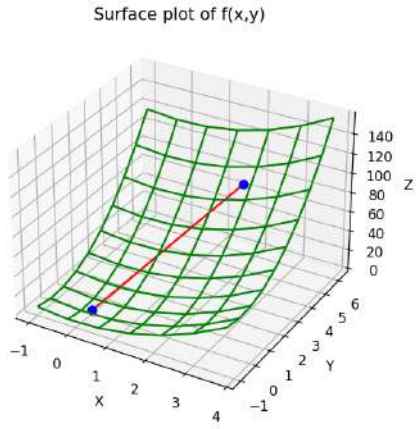
На медленно убывающих функциях алгоритмы работают почти одинаково, включая проблемы с точностью вычислений. Это происходит как раз из-

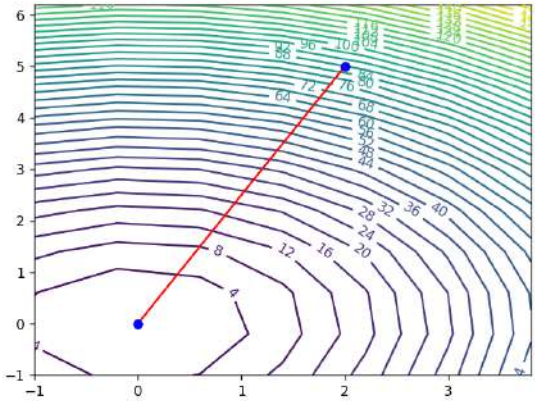
за того, что на отрезке монотонного убывания функции методы работают одинаково (используют в качестве шага $learning\_rate$). Также стоит обратить внимание на время работы программы именно в миллисекундах. Метод с одномерным поиском работает медленнее метода с постоянным шагом, хотя и делает меньше итераций. Это объясняется тем, что на каждой итерации метода с одномерным поиском значение функции вычисляется по несколько раз в отличие от 1 раза за итерацию у метода с постоянным шагом.

2.3 $f(x, y) = x^2 * 3 + y^2 * 3$

- (x; y) = (2, 5)

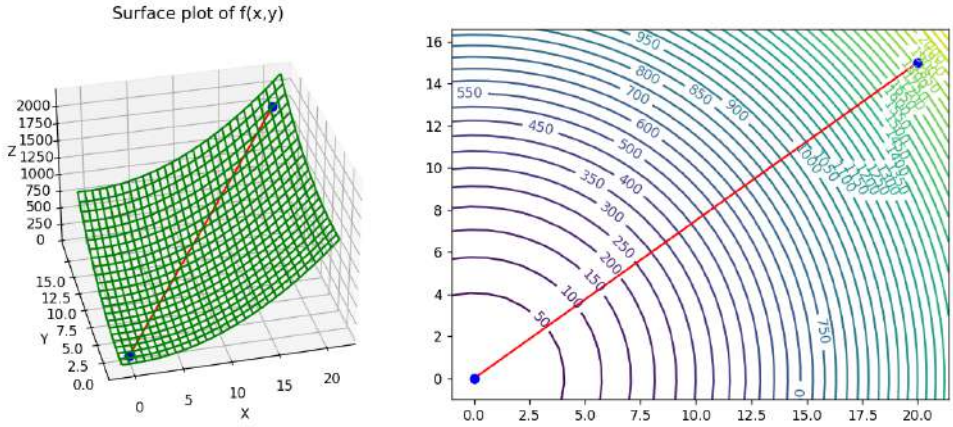
```
function: x**2 * 3 + y**2 * 3
start points: x = 2, y = 5
learning_rate: golden(0.5), eps: 0.001

x: 1.0265155115735331e-07, y: 2.5662887789324774e-07
f: 2.2918716577127454e-13, iterations: 2
work time: 0.0 ms
```





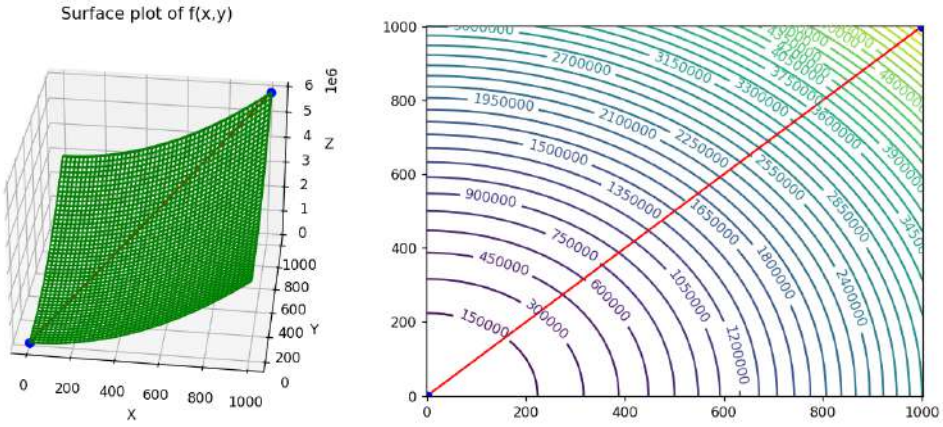
- (x; y) = (20; 15)

```
function: x**2 * 3 + y**2 * 3
start points: x = 20, y = 15
learning_rate: golden(0.5), eps: 0.001

x: 1.026515511572991e-06, y: 7.698866336799601e-07
f: 4.939378572657061e-12, iterations: 2
work time: 0.0 ms
```



- (x; y) = (1000; 1000)

```
function: x**2 * 3 + y**2 * 3
start points: x = 1000, y = 1000
learning_rate: golden(0.5), eps: 0.001

x: 5.132577557867557e-05, y: 5.132577557867557e-05
f: 1.5806011432515417e-08, iterations: 2
work time: 0.0 ms
```



Как и ожидалось, независимо от выбранного шага и скорости убывания функции, метод не будет расходиться при наличии точки минимума у функции. Еще один любопытный момент заключается в том, что для быстро убывающих функций метод работает за гораздо меньшее количество итераций, чем для нормально убывающей функции. Это происходит потому, что градиент быстроубывающей функции гораздо больше. А значит, алгоритм, не проскакивая точку минимума, сможет быстрее оказаться в окрестности конечной точки.

Итак, из плюсов метода можно выделить хорошую скорость и точность алгоритма. Данный метод очень похож на предыдущий, но есть и серьезные

отличия.

Во-первых, данный метод не расходится при наличие точки минимума у функции, чего не гарантирует метод с постоянным шагом. Это происходит потому, что метод с одномерным поиском не проскочит точку минимума, так как она попадет в нужный нам отрезок $[x_k; x_k - learning\_rate * gradient(x_{k-1})]$, а значит будет выбрана в качестве оптимальной точки (с точностью до ϵ)

Во-вторых, время работы алгоритмов различается при одинаковых значениях $learning\_rate$. Это происходит потому, что в методе с постоянным шагом на каждой итерации происходит только по одному вычислению функции и ее градиентов, в то время, как в методе с золотым сечением на каждой итерации в процессе поиска оптимальной точки на отрезке вычисление функции происходит несколько раз. А именно (примерно): $\log_\phi(\frac{right-left}{\epsilon}) = \log_\phi(\frac{right}{\epsilon}$, где $\phi = \frac{1+\sqrt{5}}{2}$

В-третьих, Нам не нужно для каждой функции стараться как-то вручную подбирать значения $learning\_rate$. Вместо этого мы можем оценить, сколько времени мы готовы пожертвовать на каждую итерацию алгоритма. А время работы каждой итерации примерно: $time\_f * \log_\phi(\frac{right}{\epsilon} + time\_gradient$, где $time\_f$ и $time\_gradient$ - время вычисления функции и ее градиента соответственно. С помощью таких несложных вычислений мы узнаем значение $right$, а алгоритм на каждой итерации сам будет выбирать оптимальное значение шага.

Дополнительное задание 1

Была разработана собственная реализация метода Нелдера-Мида на языке Python. Функция *my_nelder_mead* принимает на вход функцию для оптимизации, а также параметры сходимости и другие параметры. Реализация включает в себя проверку сходимости по итерациям и по значению функции.

Метод Нелдера-Мида (симплекс-метод) является итерационным алгоритмом оптимизации, который не требует вычисления градиента целевой функции. Он работает на основе формирования и обновления симплекса - многогранника в пространстве параметров. Алгоритм включает в себя шаги рефлексии, экспансии, констракции и сжатия, которые позволяют двигаться к оптимуму.

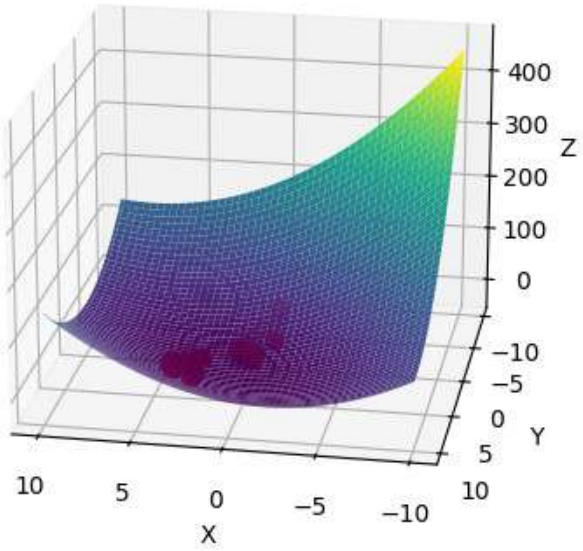
Вводится бесконечный цикл, который продолжается до достижения критерия сходимости. Точки и их значения сортируются для определения наихудшей, лучшей и второй наихудшей точек.
Шаги метода Нелдера-Мида:
Отражение:
1. Вычисляется точка отражения *xr* относительно центроида симплекса.
2. Определяется значение функции в точке отражения *xr*.
3. Если значение в *xr* лучше, чем во второй наихудшей точке и хуже, чем в лучшей точке, то заменяется вторая наихудшая точка на *xr*.
Растяжение:
1. Если значение в *xr* лучше, чем в лучшей точке, то вычисляется точка растяжения *xe*.
2. Если значение в *xe* лучше, чем в *xr*, то *xe* становится новой точкой, иначе - *xr*.
Сжатие или уменьшение:
1. Если значение в *xr* хуже, чем во второй наихудшей точке, то вычисляется точка сжатия *xc*.
2. Если значение в *xc* лучше, чем значение во второй наихудшей точке, то *xc* становится новой точкой, иначе точки сжимаются к лучшей точке.
Критерий остановки:
Метод завершает работу, если достигнут критерий сходимости, заданный по количеству итераций или по достижению достаточно малого значения изменения функции.
Возвращаемые результаты:
В конце работы метода возвращается объект *Result*, содержащий оптимальную точку, значение функции в этой точке, время выполнения метода, количество итераций, название метода и количество вычислений функции.

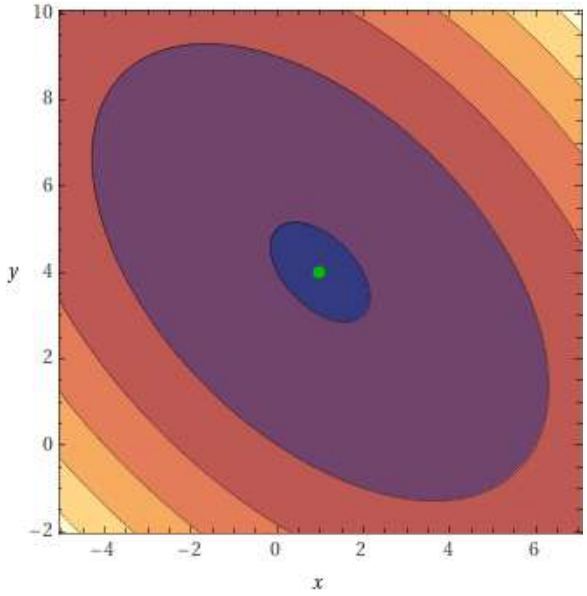
Сравним результаты с библиотечной версией алгоритма. Возьмем следующие функции:
Функция 1: $f(x,y)=x^2+xy+y^2-6x-9y$
Функция 2: $g(x,y)=3x^2+3y^2$
Функция 3: $h(x,y)=(x^2+y-11)^2+(x+y^2-7)^2$

	Функция 1	Функция 2	Функция 1 (библиотечная)	Функция 2 (библиотечная)
Итерации	24	43	74	43
Вычислений функции	42	75	138	81
Аргументы	1.00324857 4.00213812	1.51161496e-05 - 4.35538213e-06	0.99997922 4.0000079	2.05649609e-05 2.46854500e-05
Значение	- 20.999977929382965	7.42402000200541e-10	-20.99999999966979	3.0968671679740567e-09
Потраченное время (секунд)	**0.0034196376800**	**0.0047807693481**	**0.0048146247863**	**0.0014140605926**

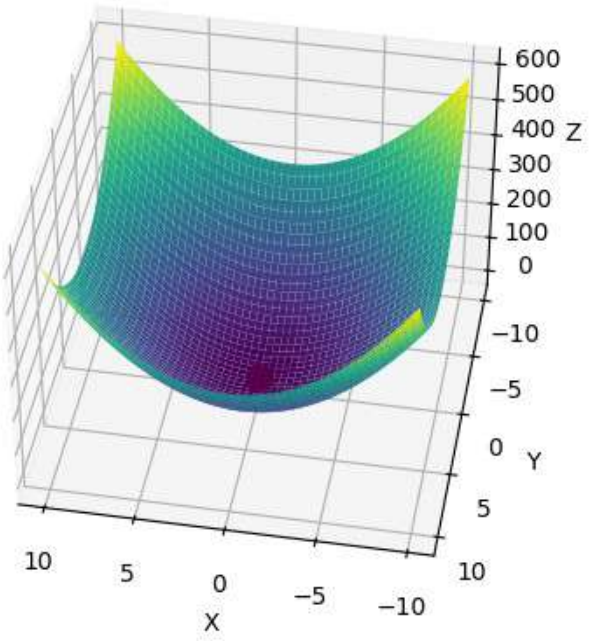
	Функция 3	Функция 3 (библиотечная)
Итерации	55	80
Вычислений функции	98	152
Аргументы	3.00000386 1.99999902	3.58446454 -1.84816402
Значение	4.930559734872751e-10	7.983653886202804e-08
Потраченное время (секунд)	**0.0038664340972**	**0.0030114650726**

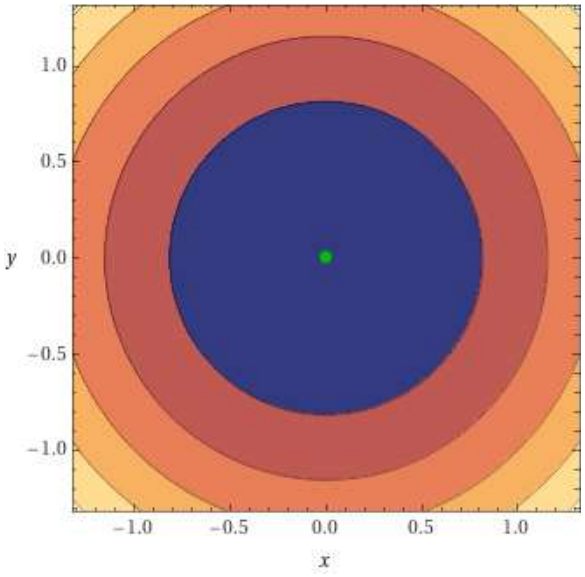
f(x, y) = x^2 + xy + y^2 - 6x - 9y



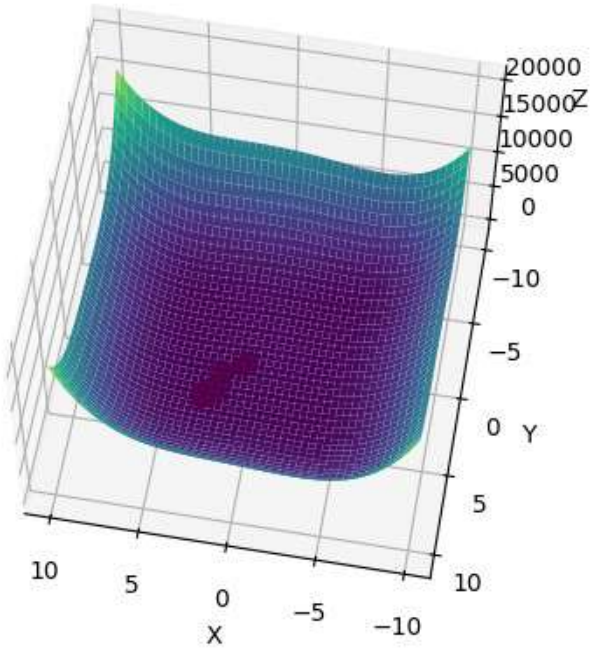


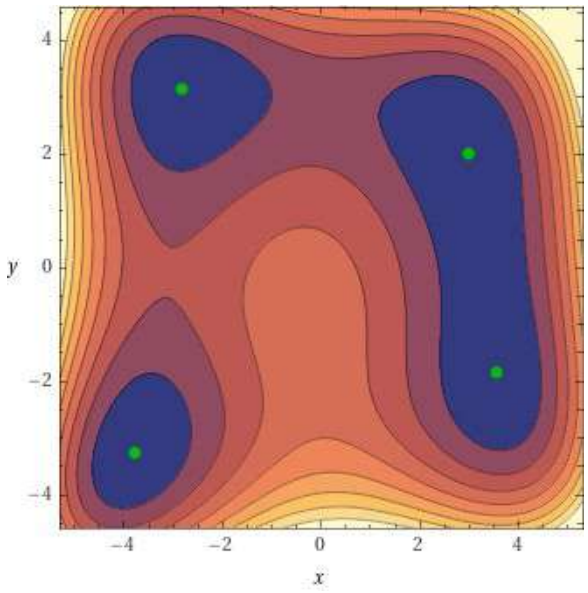
g(x, y) = 3x^2 + 3y^2


600
500
400
300
200
100
0
Z
−10
−5
0
5
10
Y
10
5
0
−5
−10
X


1.0
0.5
0.0
−0.5
−1.0
y
−1.0
−0.5
0.0
0.5
1.0
x

h(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2





Анализ результатов:
Собственная реализация метода Нелдера-Мида показывает хорошие результаты, сходясь к оптимальному значению функции за меньшее количество итераций и требуя меньшего количества вычислений функции, чем библиотечная реализация. Однако время выполнения собственной реализации в целом немного меньше или сравнимо с библиотечной. Это может указывать на то, что данная реализация хоть и более оптимальна в использовании ресурсов, но может потреблять больше времени из-за более сложного алгоритма или реализации.

Преимущества и недостатки:
Главное преимущество заключается в простоте и эффективности в реализации и использовании. В добавок к этому метод не требует градиентной информации. Из недостатков можно выделить отсутствие теории сходимости, метод может приводить к неверному ответу даже для гладких функций.

Заключение:
Таким образом была реализована собственная версия метода Нелдера-Мида, которая демонстрировала хорошие результаты сходимости. В целом, метод является полезным инструментом для решения задач оптимизации.

1 Дополнительная часть 2.1

1) Исследуйте эффективность методов на функциях n переменных, в зависимости от размерности пространства n:

$fR^n(x) = factor \cdot \sum_{i=1}^{n} x_i^2$, где $x = (x_1, x_2, ..., x_n)$

МЕТОД ГРАДИЕНТНОГО СПУСКА: $\epsilon = 0.00001, step = 0.005$

N	Factor	AVGIterations	AVGExecution
3	1	1542	11.026620864868164 ms
3	100	1	0.0 ms
3	1/100	40998	373.6245632171631 ms
30	1	1802	13.192415237426758 ms
30	100	1	0.0 ms
30	1/100	48906	474.61462020874023 ms
150	1	1944	15.31362533569336 ms
150	100	1	0.0 ms
150	1/100	58481	614.3414974212646 ms

Из проведённых тестов можно увидеть, что размерность пространства является лишь одним из многих факторов, влияющих на производительность градиентного спуска. Тем не менее он может оказывать сильное влияние на время исполнения программы. Легче всего это увидеть на примере медленно убывающей функции ($factor = 1/100$). Помимо того, что точка движется в противоположном направлении градиента на очень маленькое значение, так ещё и количество точек (фиксация одной из координат) увеличено, что увеличивает затраты на вычисление градиента.

МЕТОД ГРАДИЕНТНОГО СПУСКА НА ОСНОВЕ ЗОЛОТОГО СЕЧЕНИЯ: $\epsilon = 0.00001, startstep = 0.005$

N	*Factor*	*AVGIterations*	*AVGExecution*
3	1	1-2	1.609086990356445 3 ms or 0.0ms
3	100	2	1.000881195068359 4 ms
3	1/100	869	113.8427257537841 8 ms
30	1	2	0.99945068359375 ms or 0.0 ms
30	100	2	0.998258590698242 2 ms
30	1/100	947	116.2104606628418 ms
150	1	2	0.972032546997070 3 ms or 0.0 ms
150	100	2	0.972032546997070 3 ms
150	1/100	1044	202.2392749786377 ms

Вполне следовало ожидать, что данный метод будет более эффективен. Трудности с вычислением медленно убывающей функции остаются теми же, что и ранее.

МЕТОД НЕЛДЕРА-МИДА: $tol = 0.00001$

N	Factor	AVGIterations	AVGExecution
3	1	93	2.014636993408203 ms
3	100	84	3.607034683227539 ms
3	1/100	91	3.991365432739258 ms
5	1	235	6.514310836791992 ms
5	100	279	7.523536682128906 ms
5	1/100	239	8.602619171142578 ms
10	1	1443	37.947654724121094 ms
10	100	1436	49.20816421508789 ms
10	1/100	1432	37.29581832885742 ms

Можно заметить, что при небольших n метод Нелдера-Мида справляется лучше градиентного спуска без золотого сечения, примерно всегда выдавая одну и ту же производительность. Это объясняется отсутствием зависимости от градиента. Тем не менее при больших n алгоритм теряет точность, из-за чего значение функции в точке минимума может иметь внушительную погрешность.

2 Дополнительная часть 2.2

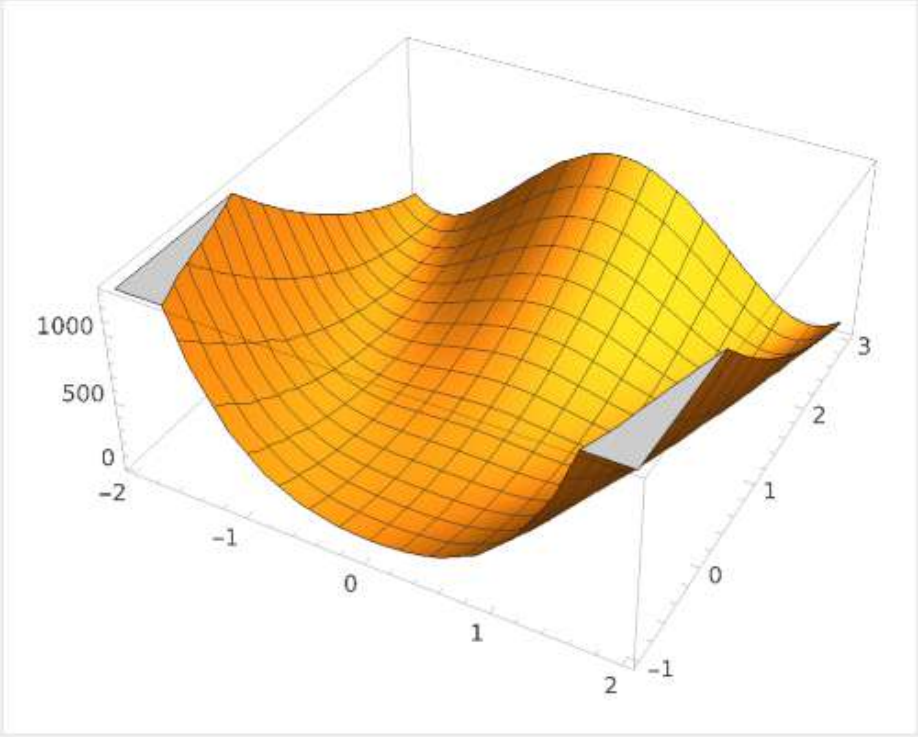
2) Исследуйте эффективность методов на плохо обусловленных функциях двух переменных:

В качестве плохо обусловленной функции возьмём $f = (1 - x)^2 + 100 \cdot (y - x^2)^2$, известную как функция Розенброка.

Обе вариации градиентного спуска не дали результатов, а точнее было получено переполнение. Если ограничить количество итераций и уровень обучения, то будет получена большая погрешность. Градиенты на большей части поверхности функции очень малы, особенно в долине, что приводит к очень медленной сходимости, если начальная точка находится далеко от минимума. Хотя у функции Розенброка единственный глобальный минимум, медленное движение вдоль долины может заставить алгоритм остановиться из-за недостаточного изменения функции или координат между итерациями,

что может быть ошибочно интерпретировано как достижение минимума. В итоге градиентному спуску не удалось приблизиться к значению в точке минимума (1,1). В свою очередь метод НЕЛДЕРА-МИДА показал следующие результаты:

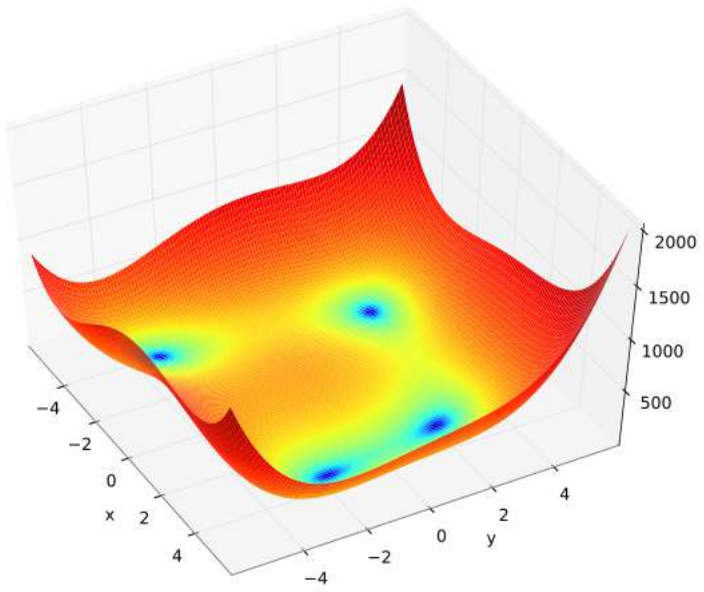
```
Calculation method: Nelder-Mead

Computed function: (1-x)**2+100*(y-x**2)**2

Iterations: 79

Number of function value calculations: 148

Result args: [1.00003147 1.00006336]

Result value: 1.0078716929461423e-09

Execution time (seconds): 0.0019900798797607422
```



Мультимодальные функции

Посмотрим работу методов на мультимодальных функциях - функциях с более, чем одним экстремумом.

Для примера возьмем функцию: $f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$ - функция Химмельблау.



Она имеет 4 локальных минимума в точках (3; 2), (-2.805..., 3.1313...), (-3.779..., -3.283...), (3.584..., -1.848...), значение функции в которых ранво 0.

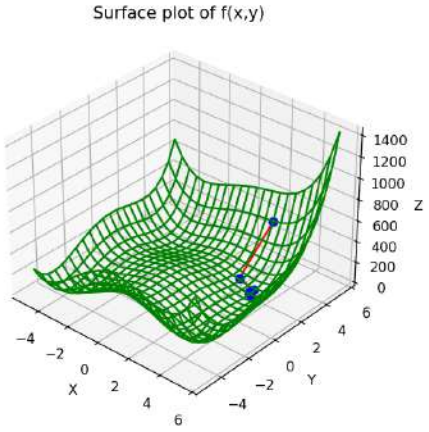
Метод градиентного спуска с постоянным шагом расходится на данной функции. В процессе вычисления получаются такие значения, и это только на 4 шаге:

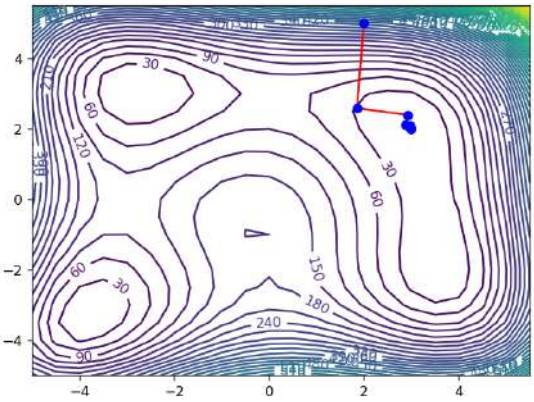
```
step 1: x = 2, y = 5, f = 404
step 2: x = -10.0, y = -193.0, f = 1386232640.0
step 3: x = -39322.0, y = 14371463.0, f = 4.265833734122783e+28
step 4: x = -83807819460778.0, y = -5.936533719208194e+21, f = 1.2420290555159123e+87
```

Теперь протестируем несколько точек на методе градиентного спуска на основе золотого сечения. Как мы уже знаем, метод не будет расходится при наличии точки минимума.

- (x, y) = (2, 5)

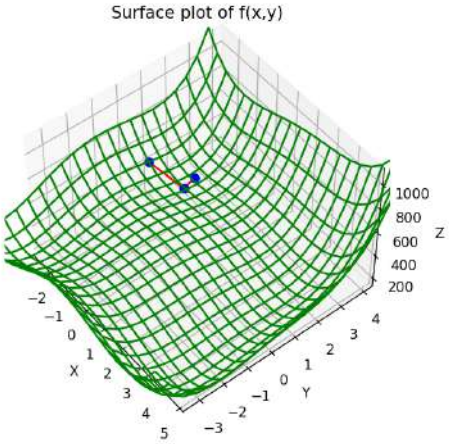
```
function: (x**2 + y - 11)**2 + (x + y**2 -7)**2
start points: x = 2, y = 5
learning_rate: golden(0.5), eps: 0.001

x: 2.9997990023386336, y: 2.001028925497618
f: 1.5364536108987987e-05, iterations: 10
work time: 0.0 ms
```

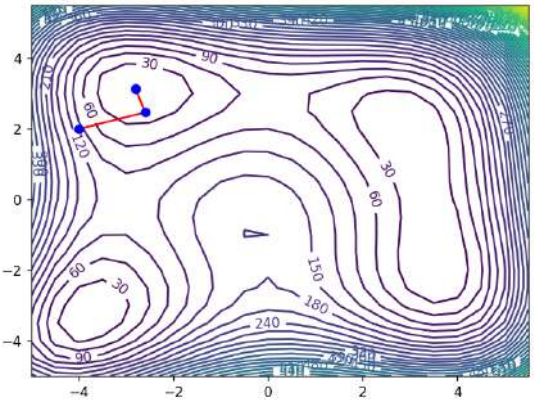




- (x, y) = (-3, 3)

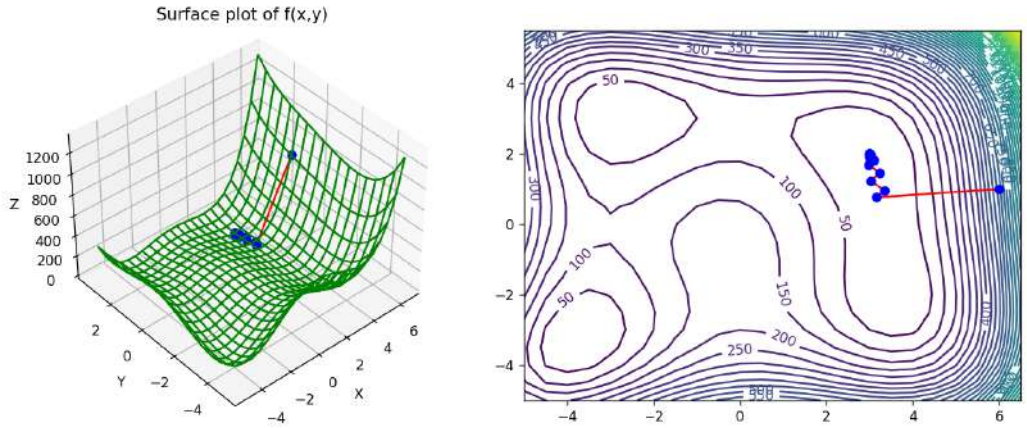
```
function: (x**2 + y - 11)**2 + (x + y**2 -7)**2
start points: x = -4, y = 2
learning_rate: golden(0.5), eps: 0.001

x: -2.804952608254765, y: 3.131538287386734
f: 2.988237525609604e-06, iterations: 3
work time: 0.0 ms
```





- (x, y) = (6, 1)

```
function: (x**2 + y - 11)**2 + (x + y**2 -7)**2
start points: x = 6, y = 1
learning_rate: golden(0.5), eps: 0.001

x: 3.000463588412583, y: 1.9993528503879792
f: 9.070387040977413e-06, iterations: 14
work time: 0.9999275207519531 ms
```



Итого мы видим, что метод градиентного спуска, основанный на золотом сечении, в зависимости от начальной точки идет к ближайшей точке минимума (в направлении наискорейшего спуска)

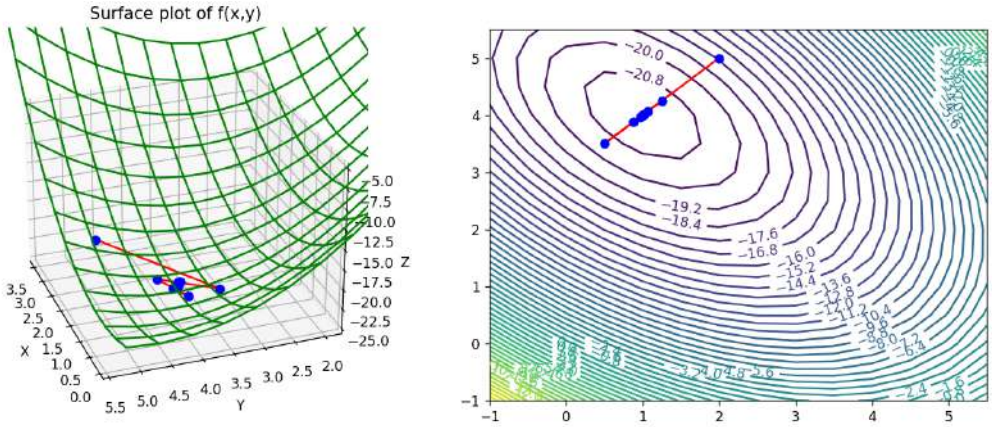
Функции с зашумленными значениями

Давайте зашумим нашу функцию $f(x, y) = x^2 + x * y + y^2 - 6 * x - 9 * y$ рандомными значениями. То есть, получится функция $f(x, y) = x^2 + x * y + y^2 - 6 * x - 9 * y + random$, где $random$ - рандомное число от -1 до 1

Рассмотрим метод градиентного спуска с постоянным шагом

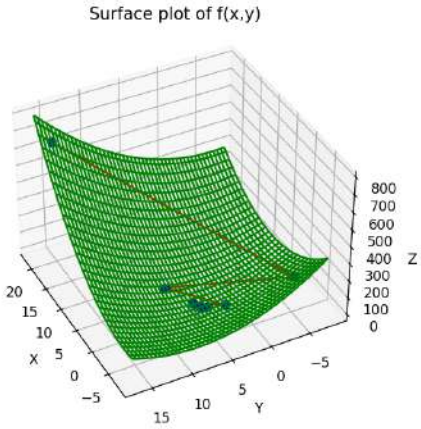
- (x, y) = (2, 5)

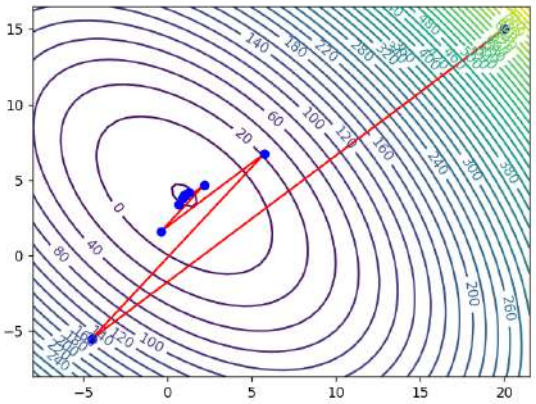
```
function: (x**2 + y - 11)**2 + (x + y**2 -7)**2 + random
start points: x = 2, y = 5
learning_rate: 0.5, eps: 0.001

x: 1.000244140625, y: 4.000244140625
f: -21.439339606683575, iterations: 12
work time: 0.0 ms
```



- (x, y) = (20, 15)

```
function: (x**2 + y - 11)**2 + (x + y**2 -7)**2 + random
start points: x = 20, y = 15
learning_rate: 0.5, eps: 0.001

x: 0.999664306640625, y: 3.999420166015625
f: -20.752718676513346, iterations: 15
work time: 0.0 ms
```



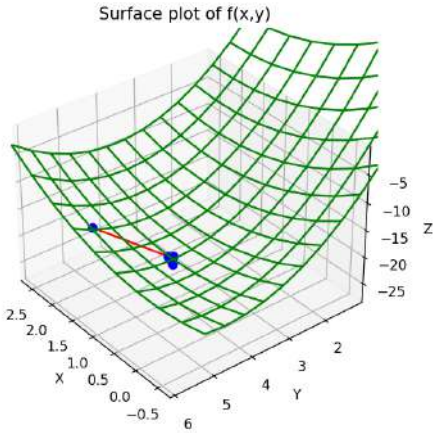


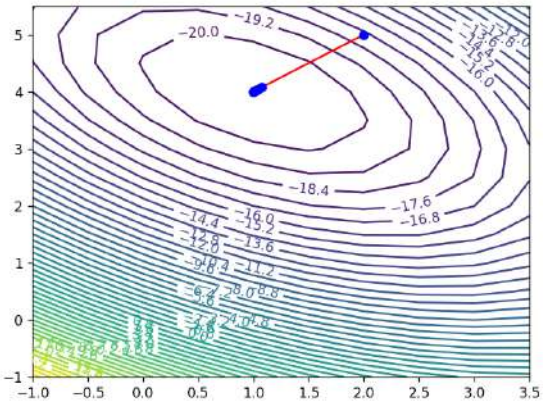
Можно заметить, что метод градиентного спуска с постоянным шагом работает на защумленной функции практически так же, как и на незашумленной. Вероятно, дело в том, что алгоритм постоянно двигается на фиксированное расстояние (шаг) и не заостряет внимание на шумах.

Теперь рассмотрим метод градиентного спуска, основанный на методе золотого сечения

- (x, y) = (2, 5)

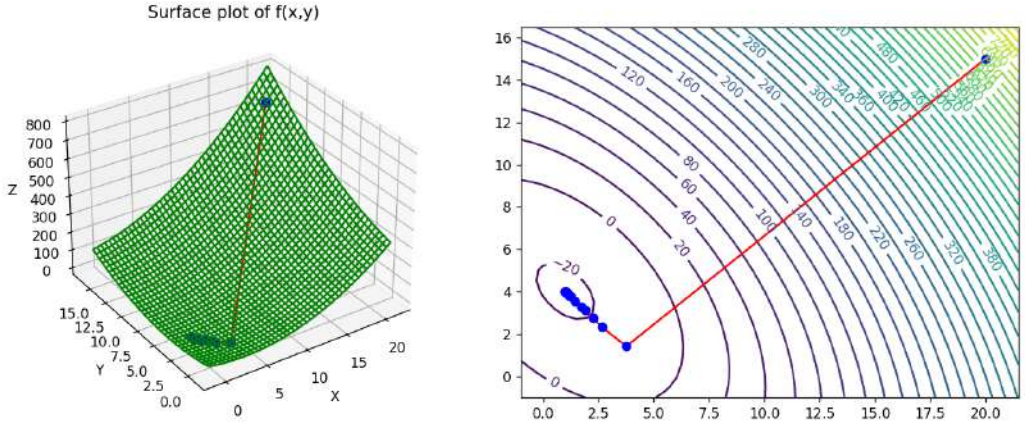
```
function: x ** 2 + x*y + y ** 2 - 6*x - 9*y + random
start points: x = 2, y = 5
learning_rate: golden(0.5), eps: 0.001

x: 0.9998908221356587, y: 3.9998908221356584
f: -21.783824594713366, iterations: 6
work time: 0.0 ms
```





- (x, y) = (20, 15)

```
function: x ** 2 + x*y + y ** 2 - 6*x - 9*y + random
start points: x = 20, y = 15
learning_rate: golden(0.5), eps: 0.001

x: 1.0017041553556485, y: 3.9982958446443555
f: -20.00401883740896, iterations: 23
work time: 1.0035037994384766 ms
```



В данном случае уже можно заметить некоторые отличия от незашумленной функции. Во-первых, в некоторых ситуациях меняется количество итераций, что означает, что метод дольше ищет путь к точке минимума. Во-вторых, точность полученных координат немного другая. Но это понятно, почему. Если мы прибавляли к функции *random*, то точка минимума могла немного измениться и поменять свои кординаты.

Получается, что метод может вместо оптимального шага выбрать неоптимальный из-за шумов, на что, соответственно, тратятся дополнительные итерации и время.

P.S.: ГИТХАБ РЕПОЗИТОРИЙ