Метод градиентного спуска

Решаемая задача - поиск локального минимума функции $f(x) : \mathbb{R}^n \to \mathbb{R}$

Описание метода

Основная идея - использование градиента для движения в направлении наискорейшего спуска
$\vec{x} = (x_1, \dots, x_n)$
$\nabla f(\vec{x}) = (\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n})$ - градиент
Шаг метода выглядит следующим образом:
$\vec{x}^{[i+1]} = \vec{x}^{[i]} - \lambda^{[i]} \bullet \nabla f(\vec{x}^{[i]})$
Здесь $\lambda^{[i]}$ - скорость градиентного спуска. Её можно выбрать различными способами:

- $\lambda^{[i]} = \mathrm{const}$. В этом случае алгоритм может расходится
- Скорость убывает в процессе спуска
- Скорость гарантирует наискорейший спуск

Критерий остановки

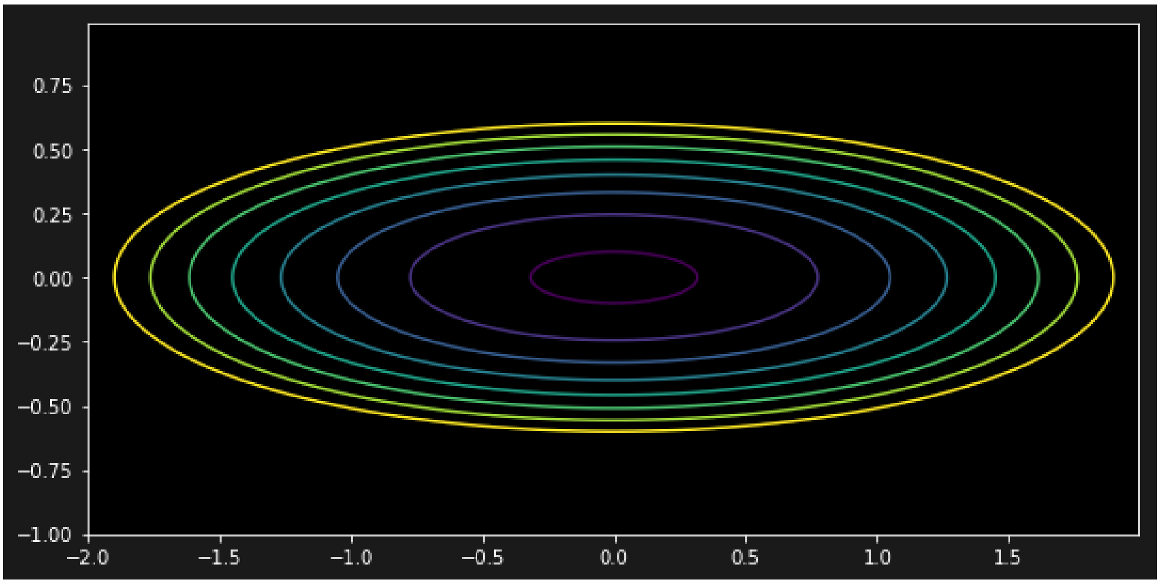
Есть несколько вариантов задания критерия остановки:
$|\vec{x}^{[i+1]} - \vec{x}^{[i]}| < \varepsilon$
$|f(\vec{x}^{[i+1]}) - f(\vec{x}^{[i]})| < \varepsilon$
$|\nabla f(x^{[i+1]})| < \varepsilon$
Очевидная проблема первых двух критериев в том, что при уменьшении шага придется уменьшать ε.
Третий критерий может просто никогда не сработать, если метод не сходится

Метод градиентного спуска с постоянным шагом

В данном случае скорость спуска задаётся константой. Рассмотрим следующую функцию:
$f(x, y) = x^2 + 10y^2$.
Частные производные такой функции:
$\frac{\partial f}{\partial x} = 2x$
$\frac{\partial f}{\partial y} = 20y$
В данном случае трехмерный рисунок будет нерепрезентативен, поэтому построим двумерный график с изолиниями (contour plot)

In [1]:

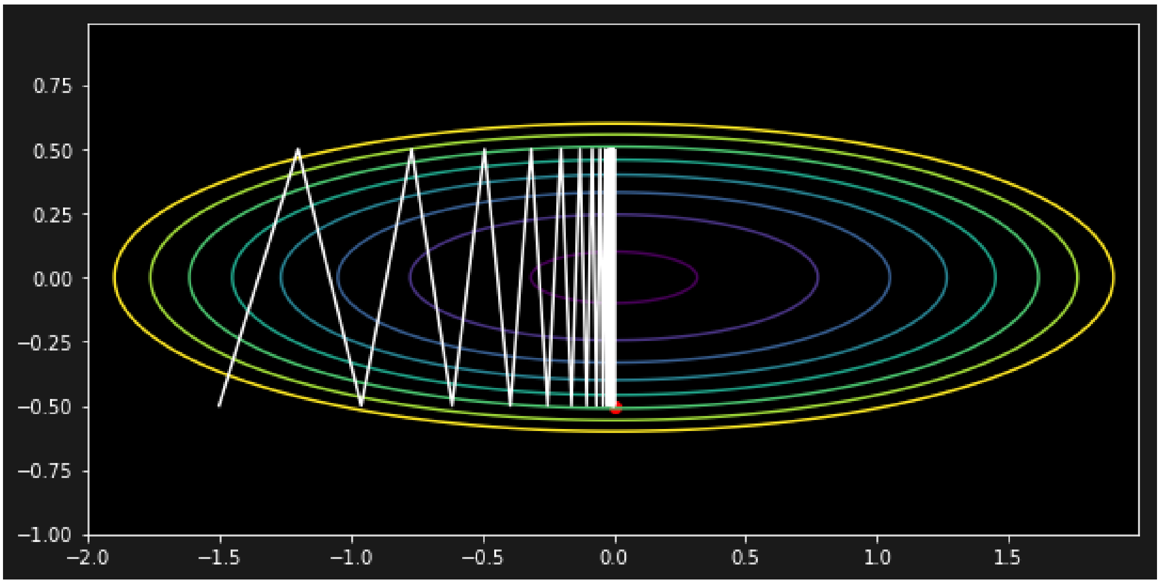
```
from matplotlib import pyplot as plt
import numpy as np
%matplotlib inline
plt.style.use('dark_background')

f = lambda x, y : np.power(x, 2) + 10 * np.power(y, 2)

plt.figure(facecolor='0.1', figsize=(10, 5))
X = np.arange(-2, 2, 0.01)
Y = np.arange(-1, 1, 0.01)
levels = np.arange(0.1, 4, 0.5)
X, Y = np.meshgrid(X, Y)
Z = f(X, Y)
plt.contour(X, Y, Z, levels=levels)
plt.show()
```



Теперь реализуем вычисление корня методом градиентного спуска

In [2]:

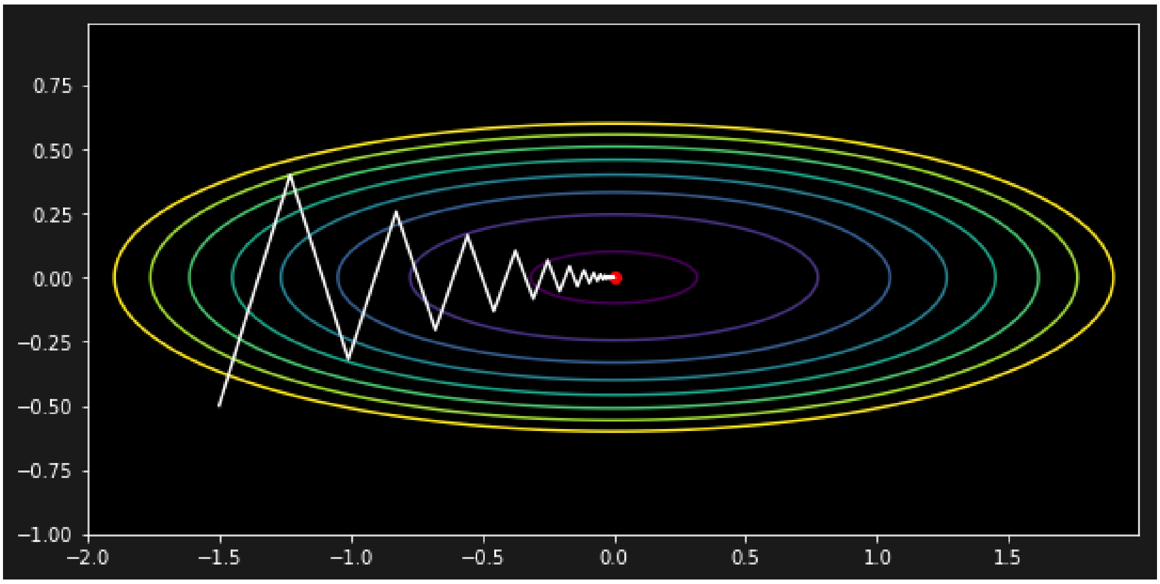
```
from numpy import linalg

def grad_descent_const(grad, x, y, eps, a):
    """
        Градиентный спуск с постоянным коэффициентом
        grad - функция градиента
        x, y - начальная точка
        eps - точность
        a - коэффициент
        Возвращаемые значения - путь, результат, особые точки (ни одной в данном
случае)
    """
    X = np.array([x, y])
    cnt = 0
    path = [X]
    while True:
        X1 = X - a * grad(*X)
        path.append(X1)
        if (linalg.norm(grad(*X)) < eps
            or cnt > 1000):
            break
        cnt += 1
        X = X1
    return path, X1, None

def plot_grad_descent(grad_descent_func, grad, x, y, **kwargs):
    path, res, pts = grad_descent_func(grad, x, y, **kwargs)
    p_X, p_Y = [p[0] for p in path], [p[1] for p in path]
    plt.figure(facecolor='0.1', figsize=(10, 5))
    plt.scatter(*res, color='r')
    if pts:
        x, y = [p[0] for p in pts], [p[1] for p in pts]
        plt.scatter(x, y, color='w', marker='o')
    plt.contour(X, Y, Z, levels=levels)
    plt.plot(p_X, p_Y, color='w')
    return path, res, pts

dfdx = lambda x, y: 2 * x
dfdy = lambda x, y: 20 * y
gradient = lambda x, y: np.array([dfdx(x, y), dfdy(x, y)])
plot_grad_descent(grad_descent_const, gradient, -1.5, -0.5, eps=0.1, a=0.1)
pass
```



Как видно, с такими параметрами польза метода сомнительна - нет сходимости. Поробуем немного уменьшить размер шага

In [3]:

```
plot_grad_descent(grad_descent_const, gradient, -1.5, -0.5, eps=0.0001, a=0.09)
pass
```



Сравним различные варианты задания отрезка

In [4]:

```
import pandas as pd
from IPython.display import display
a_r = np.arange(0.1, 0.01, -0.01)
data = [grad_descent_const(gradient, -1.5, -0.5, eps=0.0001, a=a) for a in a_r]
data = [[len(path), linalg.norm(X)] for path, X, pts in data]
df = pd.DataFrame(data, columns=['Iterations', 'Precicion'])
df['eps'] = pd.Series([0.0001] * len(a_r))
df['a'] = pd.Series(a_r)
display(df)
pass
```

	Iterations	Precicion	eps	a
0	1003	0.500000	0.0001	0.10
1	56	0.000027	0.0001	0.09
2	62	0.000036	0.0001	0.08
3	71	0.000039	0.0001	0.07
4	83	0.000042	0.0001	0.06
5	100	0.000044	0.0001	0.05
6	126	0.000045	0.0001	0.04
7	169	0.000046	0.0001	0.03
8	255	0.000047	0.0001	0.02

Как можно заметить, как только метод начинает сходится, точность самая высокая. Таким образом, лучше найти максимальное значение константы, при котором метод начинает сходится

Градиентный метод с дроблением шага

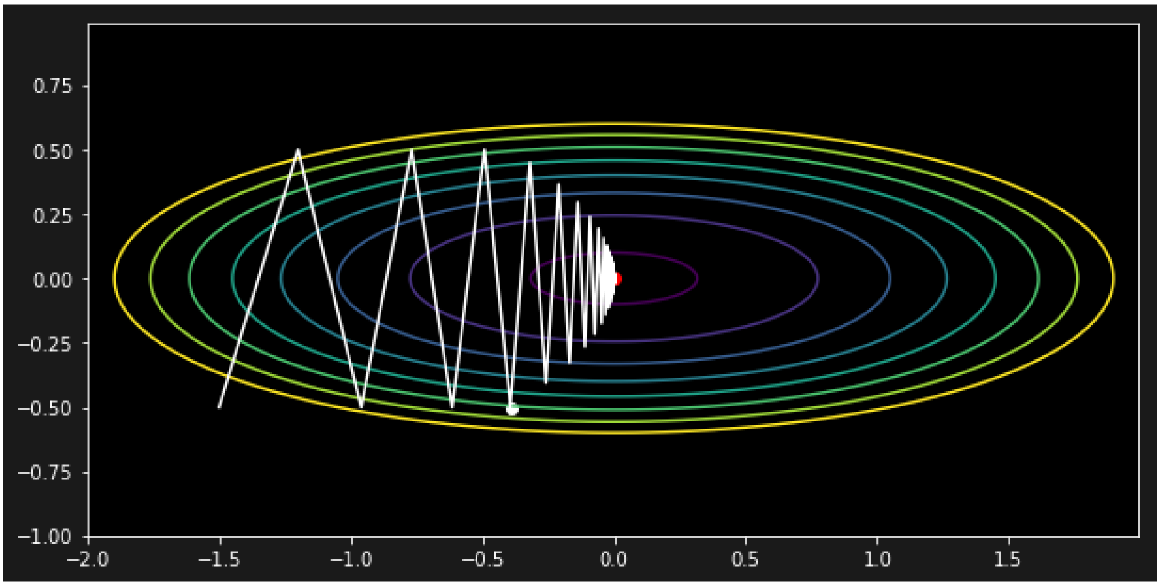
Для каждого шага можно записать условие сходимости
$f(\vec{x}^{[i+1]}) = f(\vec{x}^{[i]} - \lambda^{[i]} \bullet \nabla f(\vec{x}^{[i]})) \leq f(\vec{x}^{[i]}) - \varepsilon\lambda^{[i]}|\nabla f(\vec{x}^{[i]})|^2$
Где $\varepsilon \in (0, 1)$
На каждом шаге выполняется проверка условия. Если условие выполняется, то алгоритм продолжается. Если нет - то шаг делится на число $\delta \in (0, 1)$, пока условие не выполнится. После этого проверка продолжается

In [5]:

```
def grad_descent_splt(grad, x, y, f, eps, e, d, a):
    """
        Градиентный метод с дроблением шага
        grad - функция градиента
        x, y - начальная точка
        eps - точность для условия остановки
        e - коэффициент для проверки условия сходимости
        d - коэффициент дробления шага
        a - начальное значение коэффициента
        f - анализируемая функция
        Возвращаемые значения - путь, результат, особые точки
    """
    X = np.array([x, y])
    cnt = 0
    path = [X]
    pts = []
    while True:
        while True:
            check = f(*(X - a * grad(*X))) <= f(*X) - e * a * f(*X)**2
            if check:
                break
            pts.append(X)
            a *= d
        X1 = X - a * grad(*X)
        path.append(X1)
        if (linalg.norm(grad(*X)) < eps
            or cnt > 1000):
            break
        cnt += 1
        X = X1
    return path, X1, pts
```

In [6]:

```
plot_grad_descent(grad_descent_splt, gradient, -1.5, -0.5, eps=0.0001, e=0.1, d=
0.95, a=0.1, f=f)
pass
```



С коэффициентом 0.1 метод с постоянным шагом бы не сошёлся, но в данном случае в одном месте произведено уменьшение коэффициента, и метод начал сходится.
Выполним вычисление с разными параметрами

In [7]:

```
params = pd.DataFrame({
    "eps": [0.0001] * 5,
    "e": [0.95, 0.1, 0.1, 0.1, 0.9],
    "d": [0.95, 0.95, 0.1, 0.95, 0.95],
    "a": [1, 1, 1, 0.1, 0.1]
})
iteration, precision, split = [], [], []
for index, row in params.iterrows():
    path, res, pts = grad_descent_splt(gradient, -1.5, -0.5, eps=row['eps'],
                                       e=row['e'], d=row['d'], a=row['a'], f=f)
    split.append(len(pts))
    iteration.append(len(path))
    precision.append(linalg.norm(res))

params['Iterations'] = pd.Series(iteration)
params['Precision'] = pd.Series(precision)
params['Split times'] = pd.Series(split)
display(params)
```

	eps	e	d	a	Iterations	Precision	Split times
0	0.0001	0.95	0.95	1.0	58	0.000035	48
1	0.0001	0.10	0.95	1.0	1003	0.000007	45
2	0.0001	0.10	0.10	1.0	453	0.000048	2
3	0.0001	0.10	0.95	0.1	118	0.000004	1
4	0.0001	0.90	0.95	0.1	57	0.000040	3

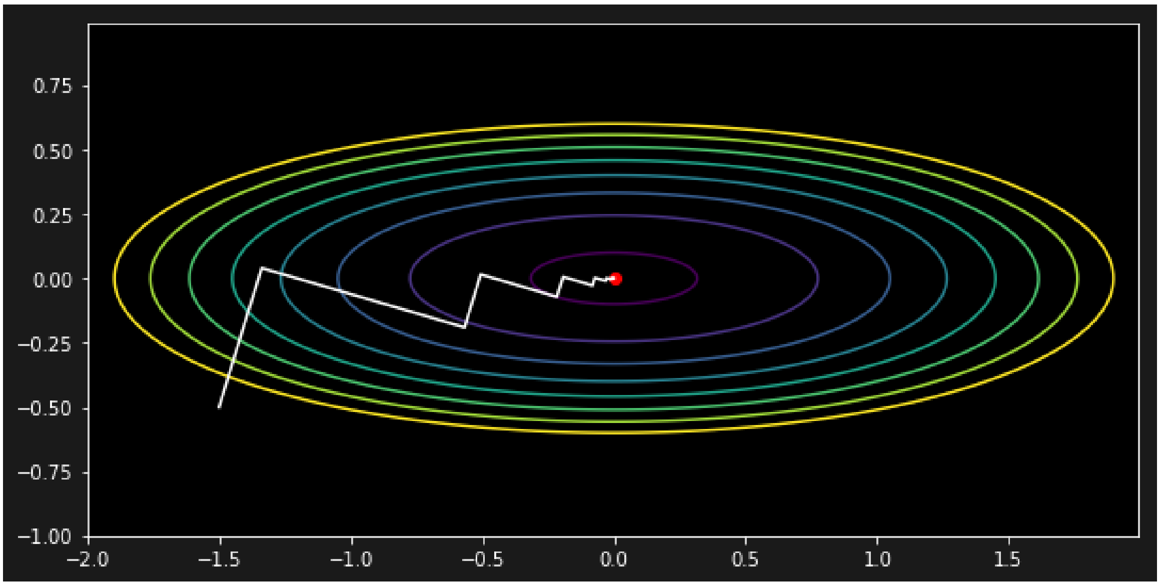
Метод наискорейшего спуска

В этом способе длина шага выбирается таким образом, чтобы движение по лучу заканчивалось в точке минимума функции.
$\operatorname{argmin}_x f(x) \in \{x | \forall y : f(y) \geq f(y)\}$ - аргумент минимизации
С использованием этой функции можно записать следующее условие:
$\alpha^{[i]} = \operatorname{argmin}_{x \in [0,\infty)} (f(\vec{x}^{[i]} - \lambda^{[i]} \bullet \nabla f(\vec{x}^{[i]})))$
Чтобы выбрать значение миниммального аргумента, нужно найти локальный экстремум функции на луче. Это можно сделать более эффективно, чем в данном примере - например, методом золотого сечения.

In [8]:

```
import pdb

def grad_descent_spd(grad, x, y, eps, f):
    """
        Градиентный спуск с выбором коэффициента для наискорейшего спуска
        grad - функция градиента
        x, y - начальная точка
        eps - точность
        f - функция
        Возвращаемые значения - путь, результат, особые точки (ни одной в данном
случае)
    """
    X = np.array([x, y])
    cnt = 0
    path = [X]
    while True:
        a_r = np.arange(0, 1, eps)
        f_r = [f(*(X - a * grad(*X))) for a in a_r]
        a = a_r[np.argmin(f_r)]
        X1 = X - a * grad(*X)
        path.append(X1)
        if (linalg.norm(grad(*X)) < eps
            or cnt > 1000):
            break
        cnt += 1
        X = X1
    return path, X1, None

path, res, pts = plot_grad_descent(grad_descent_spd, gradient, -1.5, -0.5, eps=
0.001, f=f)
print(f"Iterations: {len(path)}, precision: {linalg.norm(res)}")
```

```
Iterations: 21, precision: 9.789814634458175e-05
```



Как видно, в данном случае шаг выбирается действительно наиболее эффективным методом. Также направления шагов ортогональны друг другу. Однако решение задачи оптимизации отнимает существенную часть ресурсов. Также, для некоторых функций, метод наискорейшего спуска может быть не намного эффективнее, чем более простые методы