In [16]:
```python
import numpy as np
import matplotlib.pyplot as plt
from math import sqrt

def func(x):
    return 1/2*(x[0]**2 + x[1]**2)

def dfunc1(x):
    return x[0]

def dfunc2(x):
    return x[1]

def gradf(x):
    x = [dfunc1(x), dfunc2(x)]
    x = np.array(x)
    return x
```

Метод наискорейшего спуска

Этот метод используется для поиска минимума дифференцируемой функции $f(x) = f(x_1, x_2, \ldots, x_n)$, смещая текущее решение в направлении отрицательного градиента $\nabla f = [\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}]^T$ на каждой итерации.

Метод состоит из следующих шагов:

Инициализация: Выбираются начальное приближение x_0, размер шага $\gamma > 0$, допустимая погрешность $\varepsilon > 0$, и максимальное количество итераций N.

Тело метода (итеративное применение): На k-ой итерации у нас есть $x_{k+1} = x_k - \gamma \nabla f(x_k)$

Критерий остановки: По окончании каждой итерации проверяем условие $\|\nabla f(x_k)\| \leq \varepsilon$. Когда это условие выполняется или когда мы достигаем максимального допустимого числа итераций, мы прекращаем выполнение алгоритма.

γ (шаг обучения): Значение gamma обычно выбирается в диапазоне от 0.1 до 0.9. Если ваш шаг обучения слишком большой, алгоритм может не сойтись, и, наоборот, если слишком мал, сходимость может быть медленной. Попробуйте начать с относительно небольшого значения, например, 0.1, и увеличивайте его, пока алгоритм не начнет сходиться.

In [17]:
```python
def steepest_descent(gradf, x0, gamma, epsilon, N):
    x = np.array(x0).reshape(len(x0), 1)
    for k in range(N):
        g = gradf(x)
        x = x - gamma*g
        if np.linalg.norm(g) < epsilon:
            break
    return x

gamma = 0.1
epsilon = 1e-4
max_steps = 100
x0 = [3,3]

print(steepest_descent(gradf, x0, gamma, epsilon, max_steps))
```

```
[[7.96841967e-05]
 [7.96841967e-05]]
```

Градиентный метод с моментом

В методе градиентного спуска с моментом, общая структура алгоритма остается неизменной, но текущая позиция в процессе поиска обновляется немного измененным способом:

$$\mathbf{v}_k = \omega \mathbf{v}_{k-1} + \gamma \nabla f(\mathbf{x}_k)$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{v}_k$$

γ (шаг обучения): Значение gamma обычно выбирается в диапазоне от 0.1 до 0.9. Если ваш шаг обучения слишком большой, алгоритм может не сойтись, и, наоборот, если слишком мал, сходимость может быть медленной. Попробуйте начать с относительно небольшого значения, например, 0.1, и увеличивайте его, пока алгоритм не начнет сходиться.

ω (коэффициент момента): Значение omega также обычно находится в диапазоне от 0.1 до 0.9. Можно начать с маленького значения, например, 0.1, и постепенно увеличивать, чтобы увидеть, как это влияет на производительность. Момент обычно помогает устранить осцилляции и ускорить сходимость.

In [18]:
```python
def steepest_descent_with_momentum(gradf, x0, gamma, epsilon, omega, max_
    x0 = np.array(x0)
    v = 0

    for i in range(max_iterations):
        g = gradf(x0)
        v = omega*v + gamma*g
        x0 = x0 - v
        if np.linalg.norm(g) < epsilon:
            break
    return x0, i+1

x0 = [-3,-0.1]
gamma = 0.7*0.8 ## разбиваем множитель для gamma и omega (который в сумме
epsilon = 1e-8
omega = 0.1*0.2
max_iterations = 1000
```

```
steepest_descent_with_momentum(gradf, x0, gamma, epsilon, omega, max_iter
```

Out[18]: (array([-1.78658039e-09, -5.95526798e-11]), 24)

ADAGRAD

Adagrad использует адаптивный градиент, специфичный для каждой оси (каждой переменной).

Пусть $g_{k,i}$ - градиент критерия оптимальности по i-й переменной в k-й итерации,

$$G_{k,i} = \sum_{j=1}^{k} (g_{j,i})^2$$

где $G_{k,i}$ - сумма квадратов градиентов по i-й переменной до k-й итерации.

Обновление i-й переменной:

$$x_{k+1,i} = x_{k,i} - \frac{\gamma}{\sqrt{G_{k,i}+\epsilon}} g_{k,i}$$

где γ - скорость обучения, ϵ - малая константа, предотвращающая деление на ноль.

Этот метод позволяет каждой переменной адаптивно регулировать скорость обучения, учитывая её историю градиентов.

γ (learning rate): Значение gamma (learning rate) обычно выбирается в диапазоне от 0.01 до 0.1. Если ваш шаг обучения слишком большой, это может привести к расходимости, а если слишком мал, оптимизация может быть слишком медленной. Начните с относительно небольшого значения и, при необходимости, постепенно увеличивайте.

ϵ (эпсилон для стабилизации): Значение eta обычно выбирается в диапазоне от 1e-8 до 1e-4. Эта константа добавляется в знаменатель в формуле обновления, чтобы избежать деления на очень маленькие значения. Начните с небольшого значения, например, 1e-8, и увеличивайте, если это необходимо.

In [19]:
```
def adagrad(gradf, x0, gamma, eta, epsilon, max_iterations):
    x0 = np.array(x0)
    v = np.zeros(len(x0))
    G = np.zeros(len(x0))

    for i in range(max_iterations):
        g = gradf(x0)
        G += np.multiply(g, g)
        v = gamma*np.ones_like(G) / np.sqrt(G+eta) * g
        x0 = x0 - v
        if (np.linalg.norm(g) < epsilon):
            break

    return x0, i

x0 = [3,-0.1]
gamma = 1
epsilon = 1e-8
eta = 1e-8 ## Порядка epsilon
max_iterations = 1000

adagrad(gradf, x0, gamma, eta, epsilon, max_iterations)
```

Out[19]: (array([6.10570401e-09, 2.92681603e-73]), 72)

ADAM

ADAM (ADAPTIVE MOMENT ESTIMATION) - одна из наиболее широко используемых современных модификаций алгоритма наискорейшего спуска.

Сначала определяются вспомогательные величины:

$$m_k = \omega_1 m_{k-1} + (1-\omega_1) g_k$$

$$v_k = \omega_2 v_{k-1} + (1-\omega_2) g_k^2$$

и их скорректированные версии:

$$\hat{m}_k = \frac{m_k}{1-\omega_1^k}$$

$$\hat{v}_k = \frac{v_k}{1-\omega_2^k}$$

Затем текущее решение обновляется по алгоритму:

$$x_{k+1} = x_k - \frac{\gamma}{\sqrt{\hat{v}_k + \epsilon}} \hat{m}_k$$

γ (learning rate): Значение gamma (learning rate) обычно выбирается в диапазоне от 1e-5 до 0.1. Можно начать с небольшого значения, например, 0.001, и затем экспериментировать с изменением этого значения в зависимости от производительности. Если процесс сходится слишком медленно или не сходится вовсе, увеличьте learning rate.

ω_1 (экспоненциальное затухание для момента первого порядка): Значение omega1 обычно лежит в диапазоне от 0.8 до 0.999. Чем ближе к 1, тем больше

веса отдаются предыдущим моментам первого порядка. Можно начать с 0.9 и далее настраивать в соответствии с результатами.

ω_2 (экспоненциальное затухание для момента второго порядка): Значение omega2 также обычно лежит в диапазоне от 0.8 до 0.999. Можно начать с 0.999, и по мере необходимости уменьшать это значение. Значение близкое к 1 дает больший вес более новым значениям второго момента.

ϵ (эпсилон для стабилизации): Значение eta обычно выбирается в диапазоне от 1e-8 до 1e-4. Можно начать с 1e-8 и увеличивать, если необходимо. Эта константа добавляется к знаменателю в формуле коррекции шага для избежания деления на очень маленькие значения.

In [20]:
```python
def adam(gradf, x0, gamma, omega1, omega2, eta, max_iterations, epsilon):
    x0 = np.array(x0)
    m = np.ones_like(x0)
    v = np.ones_like(x0)
    for i in range(max_iterations):
        g = gradf(x0)

        v = v * omega2 + (1 - omega2) * np.multiply(g, g)
        adj_v = v / (1 - omega2)
        gamma_adj = gamma * np.ones_like(x0) / (np.sqrt(adj_v + eta))

        m = m * omega1 + (1 - omega1) * g
        adj_m = m / (1 - omega1)

        x0 = x0 - np.multiply(gamma_adj, adj_m)

        if np.linalg.norm(g) < epsilon:
            break
    return x0, i

x0 = [3,0.1]
gamma = 0.45 ## скорость обучения
omega1, omega2 = 0.79, 0.99 ## omega1 наиболее близкая к 1, разница между
epsilon = 1e-4
eta = 1e-4 ## порядка epsilon
max_iterations = 10000

adam(gradf, x0, gamma, omega1, omega2, eta ,max_iterations, epsilon)
```

Out[20]:
```
(array([-1.06181367e-04,  2.05860890e-05]), 72)
```