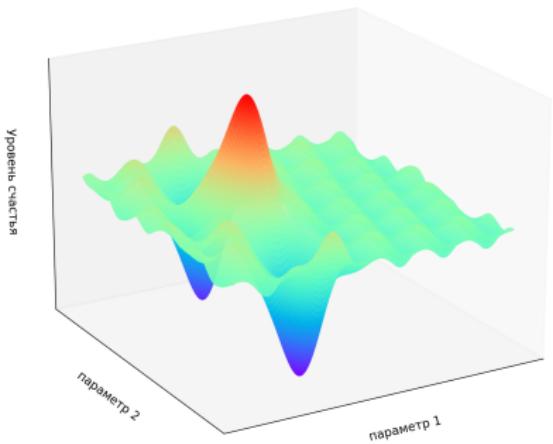


## Минимизация функций

Шмаков Владимир Евгеньевич - ФФКЭ гр. Б04-105

30 сентября 2024 г.

## Возможные постановки задачи

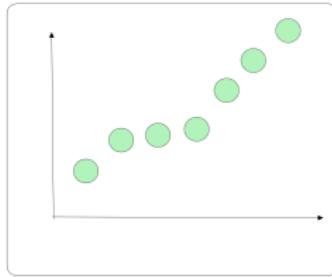


- ▶ Нахождение глобальных минимумов/максимумов функции
- ▶ Нахождение всех экстремумов

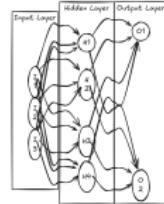
# Минимизируемые функции

$$y = F(x)$$

аналитически  
заданные функции



функции  
заданные на сетке



вычислительные графы

## Метод Фибоначи

1. Задаём начальные границы  $a$  и  $b$ .
2. Пока количество итераций не превысило  $N$ :

► Рассчитаем значения

$$x_1 = a + (b - a) \frac{F_{n-2}}{F_n}$$

$$x_2 = a + (b - a) \frac{F_{n-1}}{F_n}$$

- Если  $f(x_1) > f(x_2)$ , то сужаем левую границу  $a = x_1$ .  
► Иначе сужаем правую границу  $b = x_2$ .

3. Результат  $(x_1 + x_2)/2$

# Метод Фибоначи

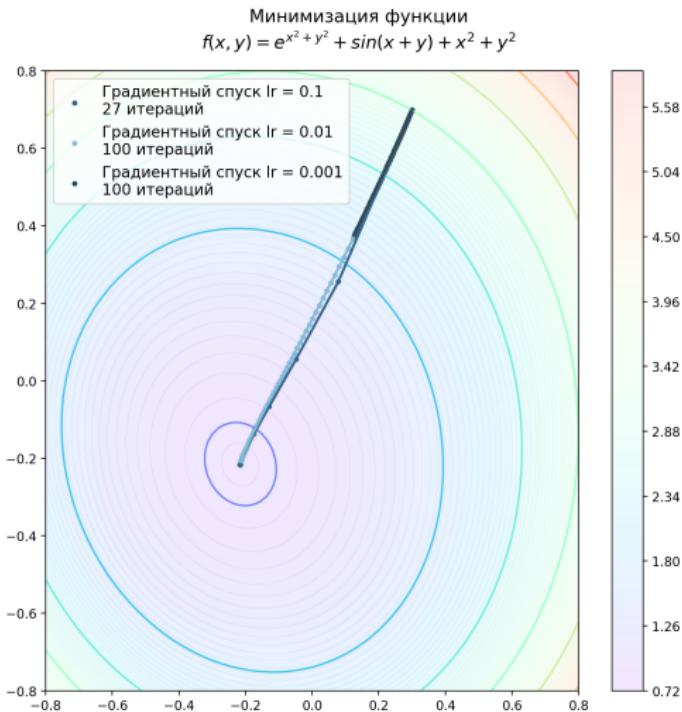
```
1 def fibonacci_minimization(func: tp.Callable, left_bound: float = -1.6, right_bound: float = 1.6, N: int = 70):
2     """Минимизация функции методом фибоначи
3
4     Args:
5         func (tp.Callable): Минимизируемая функция
6         left_bound (float, optional): Левая граница отрезка на котором минимизируется функция
7         right_bound (float, optional): Правая граница отрезка на котором минимизируется функция
8         N (int, optional): Число итераций. Defaults to 100.
9     """
10    history = {"a": [], "b": [], "x1": [], "x2": []}
11    a, b = left_bound, right_bound
12    for n in np.arange(N - 1, 1, -1, dtype = np.int64):
13        x1 = a + (b - a) * fibonacci_numbers[n - 2] / fibonacci_numbers[n]
14        x2 = a + (b - a) * fibonacci_numbers[n - 1] / fibonacci_numbers[n]
15        history["a"].append(a)
16        history["b"].append(b)
17        history["x2"].append(x2)
18        history["x1"].append(x1)
19        if func(x1) > func(x2):
20            a = x1
21        else:
22            b = x2
23
24    return (x1 + x2) / 2, history
25
```

# Градиентный спуск

1. Выбираем начальную точку  $x_0$
2. Пока не выполнено условие остановки
  - ▶ Вычисляем  $\nabla f(x_{k+1})$
  - ▶ Шаг градиентного спуска

$$x_{k+1} = x_k - \alpha \nabla f(x_{k+1})$$

Константу  $\alpha$  называют величиной шага(**step size**) или скоростью обучения (**learning rate**)

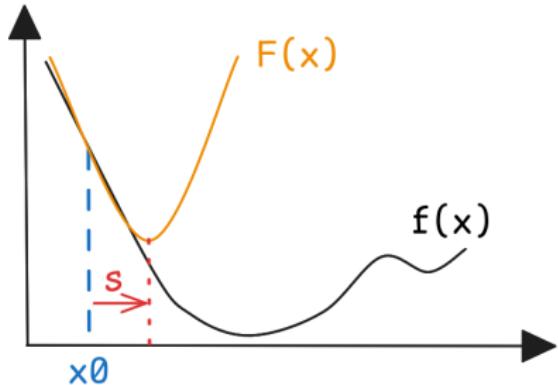


# Градиентный спуск

```
1 def gradient_descent(grad_f: callable,
2                      lr: float,
3                      x_0: np.ndarray,
4                      eps: float = 1e-6,
5                      max_iter: int = 100) -> tuple:
6     """Градиентный спуск
7
8     Args:
9         grad_f (callable): градиент минимизируемой функции
10        lr (float): learning rate
11        x_0 (np.ndarray): начальная точка
12        eps (float, optional): для критерия остановки. Defaults to 1e6.
13        max_iter (int, optional): максимально допустимое число итераций. Defaults to 100.
14
15    Returns:
16        tuple: результат, история
17        """
18    x = x_0.copy()
19    history = [x.copy()]
20    while len(history) < max_iter and np.linalg.norm(grad_f(x)) > eps:
21        x -= lr * grad_f(x)
22        history.append(x.copy())
23    return x, np.array(history)
```

## Метод Ньютона

Пусть  $f(x) \in C^2$ . Выберем начальную точку  $x_0$ . Вместо функции  $f$  рассмотрим её квадратичное приближение  $F(x)$  в окрестности  $x_0$ :

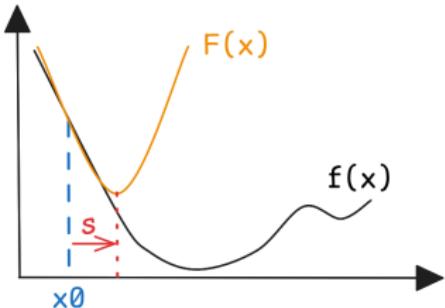


$$F(\vec{x}) = f(\vec{x}_0) + \nabla f(\vec{x}_0) \cdot (\vec{x} - \vec{x}_0) + \frac{1}{2} (\vec{x} - \vec{x}_0)^T G_f(\vec{x}_0) (\vec{x} - \vec{x}_0)$$

$$\vec{s} = \vec{x} - \vec{x}_0$$

$$F(\vec{s}) = f(\vec{x}_0) + \nabla f(\vec{x}_0) \cdot \vec{s} + \frac{1}{2} \vec{s}^T G_f(\vec{x}_0) \vec{s}$$

## Метод Ньютона



Найдём минимум функции  $F$

$$\nabla F(s) = 0 + \nabla f(\vec{x}_0) + \frac{1}{2}(G_f(\vec{x}_0)s + s^T G_f) = \nabla f(\vec{x}_0) + G_f s = 0$$

$$s = -G_f^{-1} \nabla f(x_0)$$

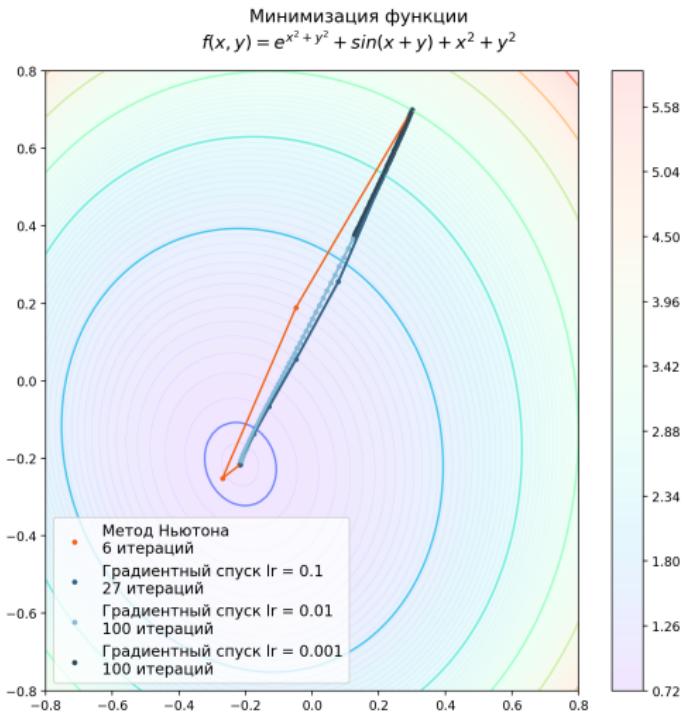
$$x = x_0 - G_f^{-1} \nabla f(x_0)$$

Для использования метода Ньютона необходима положительная определённость матрицы Гессса.

# Метод Ньютона

1. Выбираем начальную точку  $x_0$
2. Пока не выполнено условие остановки
  - ▶ Вычисляем  $\nabla f(x_{k+1})$  и  $G_f^{-1}(x_k)$
  - ▶ Шаг метода Ньютона

$$x_{k+1} = x_k - G_f^{-1}(x_k) \nabla f(x_k)$$



# Метод Ньютона

```
1 def newton_method(grad_f: callable,
2                     hessian_f: callable,
3                     x_0: np.ndarray,
4                     eps: float = 1e-6,
5                     max_iter: int = 100) -> tuple:
6     """Метод Ньютона
7
8     Args:
9         grad_f (callable): градиент минимизируемой функции
10        hessian_f (callable): Матрица Гессе
11        x_0 (np.ndarray): начальная точка
12        eps (float): для критерия остановки
13        max_iter (int): максимальное число итераций
14    Returns:
15        tuple: минимум, история
16    """
17    x = x_0.copy()
18    history = [x.copy()]
19    while np.linalg.norm(grad_f(x)) > eps and len(history) < max_iter:
20        grad, hessian = grad_f(x), hessian_f(x)
21        x -= np.linalg.inv(hessian) @ grad
22        history.append(x.copy())
23    return x, np.array(history)
```

# Метод Ньютона

```
1 def gd_newton(grad_f: callable,
2                 hessian_f: callable,
3                 x_0: np.ndarray,
4                 alpha: float,
5                 eps: float = 1e-6,
6                 max_iter: int = 100) -> tuple:
7     """Метод Ньютона. Если гессиан определен отрицательно, производим шаг при помощи градиентного спуска
8     Args:
9         grad_f (callable): градиент минимизируемой функции
10        hessian_f (callable): Матрица Гессе
11        x_0 (np.ndarray): начальная точка
12        alpha (float): величина шага при обновлении точки градиентным спуском
13        eps (float): для критерия остановки
14        max_iter (int): максимальное число итераций
15    Returns:
16        tuple: минимум, история
17    """
18    x = x_0.copy()
19    history = [x.copy()]
20    while np.linalg.norm(grad_f(x)) > eps and len(history) < max_iter:
21        grad, hessian = grad_f(x), hessian_f(x)
22        if np.all(np.linalg.eigvals(hessian) > 0):
23            x -= np.linalg.inv(hessian) @ grad
24        else:
25            x -= alpha * grad
26        history.append(x.copy())
27    return x, np.array(history)
```

## Улучшение градиентного спуска



Изменять величину шага в зависимости от обстоятельств

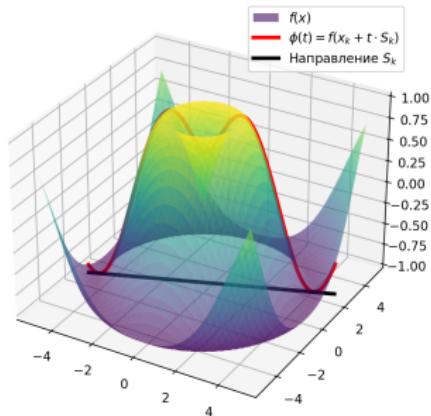


При выборе нового направления учитывать предыдущее направление

# Метод сопряженных градиентов

1. Выбрать начальную точку  $x_0$ . В качестве начального направления выберем направление антиградинета  $S_0 = -\nabla f(x_0)$ .
2. Пока не выполнено условие остановки
  - ▶ Сконструируем функцию одной переменной  $\phi(t) = f(x_k + t \cdot S_k)$ .
  - ▶ Найдём минимум функции  $\phi$ .  
 $x_{k+1} = x_k + t_{opt} \cdot S_k$
  - ▶ Вычислим новое направление

$$S_{k+1} = \nabla f(x_{k+1}) + \beta(\nabla f(x_{k+1}), \nabla f(x_k), S_k)S_k$$



## Метод сопряженных градиентов

Метод сопряженных градиентов - класс методов, отличающихся функциями  $\beta(\nabla f(x_{k+1}), \nabla f(x_k), S_k)$



$$\beta_k^{HS} = \frac{\langle \nabla f(x_{k+1}), \nabla f(x_{k+1}) - \nabla f(x_k) \rangle}{\langle S_k, \nabla f(x_{k+1}) - \nabla f(x_k) \rangle} \quad \text{Хестенс, Штифель}$$



$$\beta_k^{FR} = \frac{|\nabla f(x_{k+1})|^2}{|\nabla f(x_k)|^2} \quad \text{Флетчер, Ривз}$$



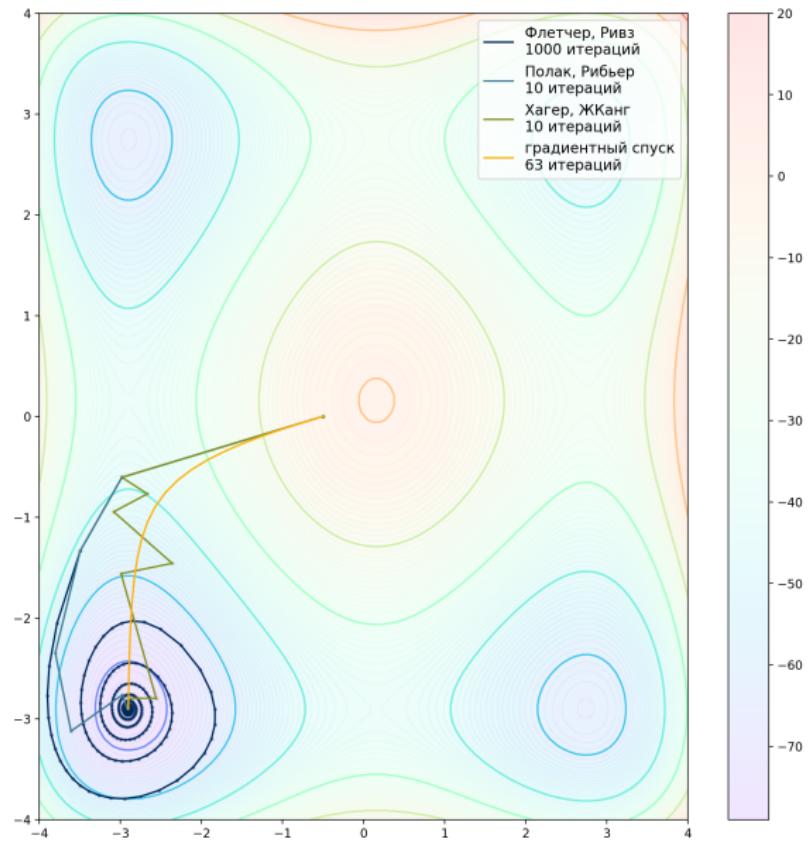
$$\beta_k^{PR} = \frac{\langle \nabla f(x_{k+1}), \nabla f(x_{k+1}) - \nabla f(x_k) \rangle}{|\nabla f(x_{k+1})|^2} \quad \text{Полак, Рибъер}$$



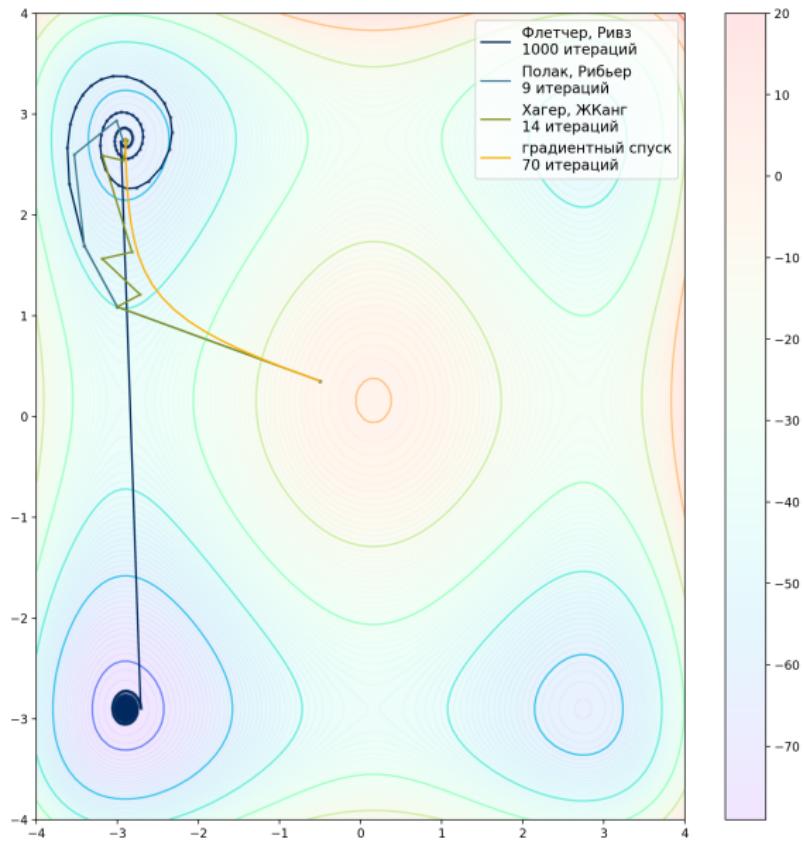
# Метод сопряженных градиентов

```
1 def minimize(f: callable,
2             grad_f: callable,
3             x0: np.ndarray,
4             Bk: callable,
5             phi_minimization: callable,
6             max_iter: int = int(1e3),
7             eps = 1e-6):
8     """Минимизация методом сопряженных градиентов
9
10    Args:
11        f (tp.Callable): Минимизируемая функция: Rn -> R
12        grad_f (tp.Callable): Градиент минимизируемой функции Rn -> Rn
13        x0 (np.ndarray): Начальное приближение точки минимума
14        Bk (tp.Callable): Метод для выбора константы Bk на k-ой итерации
15        phi_minimization (tp.Callable, optional): Метод минимизации функции вдоль выбранного направления.
16        max_iter (int, optional): Максимальное число итераций. Defaults to int(1e4).
17        eps (_type_, optional): условие остановки на градиент. Defaults to 1e-6.
18    """
19    x = x0
20    S = -grad_f(x0)
21    history = [x0]
22    while len(history) < max_iter and np.linalg.norm(grad_f(x)) > eps and np.isfinite(np.linalg.norm(x)):
23        alpha_min = phi_minimization(lambda alpha: f(x + alpha * S))
24        x_new = x + alpha_min * S
25        Beta = Bk(rk = -grad_f(x_new), rk_prev = -grad_f(x), Sk = S)
26        S = -grad_f(x_new) + Beta * S
27        x = x_new
28        history.append(x.copy())
29    return x, np.array(history)
30
```

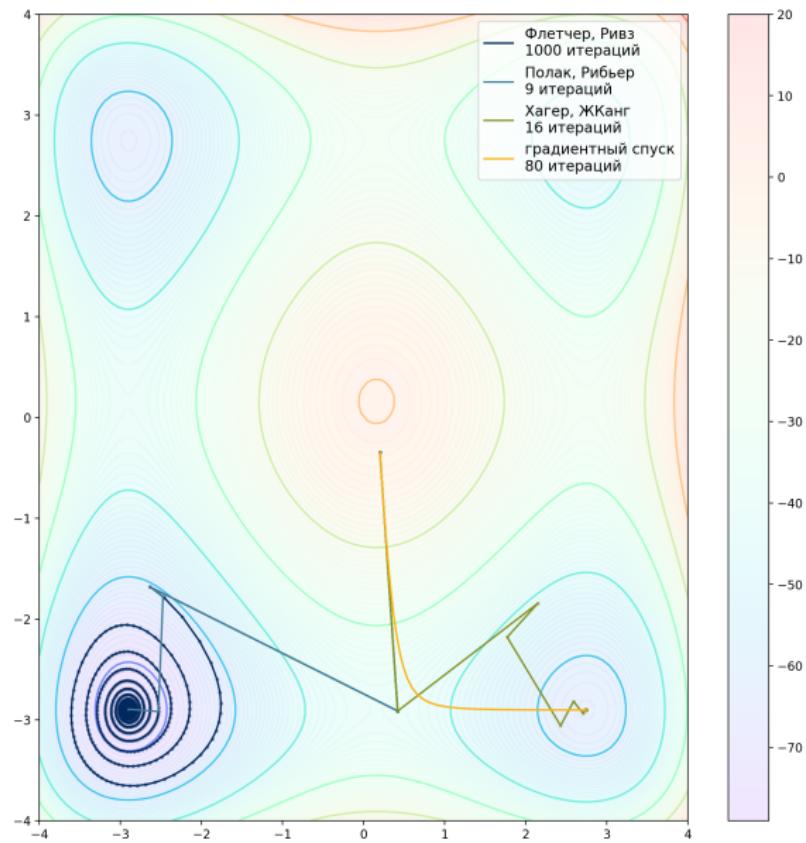
# Метод сопряженных градиентов



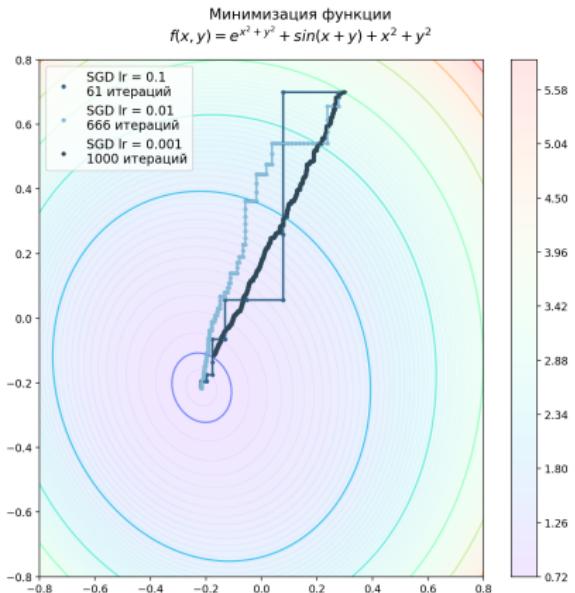
# Метод сопряженных градиентов



# Метод сопряженных градиентов



# Стохастический градиентный спуск



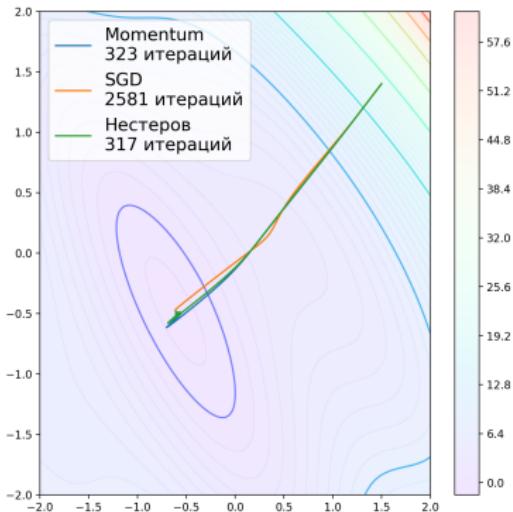
- ▶ Выбрать начальную точку  $x_0 \in R^N$ , величину шага  $\alpha$ , число  $b \leq N$ .
- ▶ Пока не выполнено условие остановки
  1. Случайно выбираем  $N$  попарно различных индексов:  $\{n_i\}_{i=1}^b$ ,  $n_i < N$
  2. Составим вектор  $d = (0, \dots, \frac{\partial f}{\partial x_{n_1}}, \dots, \frac{\partial f}{\partial x_{n_b}}, 0, 0, \dots) \in R^N$
  3. Шаг стохастического градиентного спуска

$$x_{k+1} = x_k - \alpha d$$

# Стохастический градиентный спуск

```
1 def sgd(grad_f: callable,
2         lr: float,
3         x_0: np.ndarray,
4         batch_size: int,
5         max_iter: int = 1000,
6         eps: float = 1e-6) -> tuple:
7     """Стохастический градиентный спуск
8
9     Args:
10        grad_f (callable): градиент минимизируемой функции
11        lr (float): величина шага
12        x_0 (np.ndarray): начальная точка
13        batch_size (int): размер батча
14        max_iter (int): максимальное число итераций
15        eps (float): для условия остановки
16
17    Returns:
18        tuple: результат, история
19    """
20    x = x_0.copy()
21    history = [x.copy()]
22    grad = grad_f(x)
23    while np.linalg.norm(grad) > eps and len(history) < max_iter:
24        #выбираем индексы по которым будет зануляться градиент
25        indecies = np.random.choice(len(x), len(x) - batch_size)
26        d = grad.copy()
27        d[indecies] = 0
28        x -= lr * d
29        grad = grad_f(x)
30        history.append(x.copy())
31    return x, np.array(history)
```

# Momentum



Алгоритм momentum - градиентный спуск, учитывающий предыдущее направление.

$$x_{k+1} = x_k - \alpha \nabla f(x_k) + \beta(x_k - x_{k-1})$$

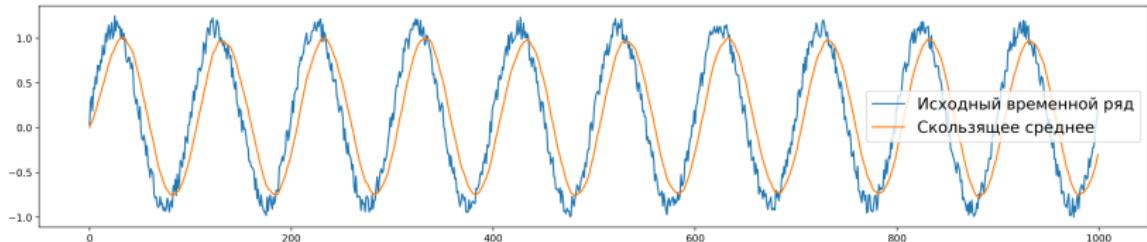
Формулировка Нестерова

$$x_{k+1} = x_k - \alpha(\nabla f(x_k) + \beta(x_k - x_{k-1}))$$

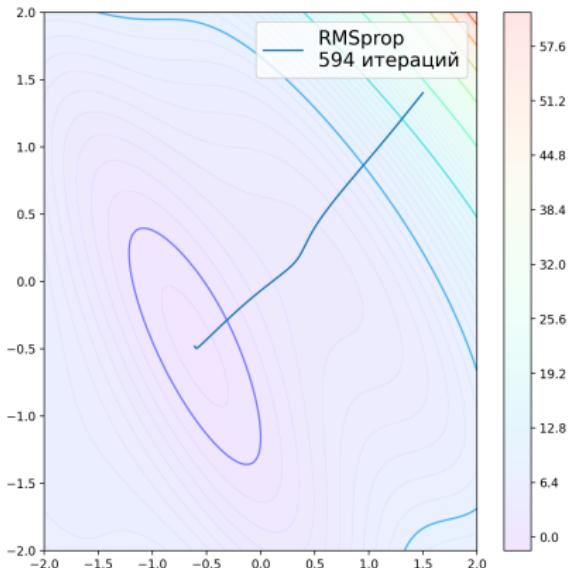
## Почему momentum работает?

Алгоритм momentum аналогичен алгоритму «скользящее среднее», используемого для фильтрации сигналов. Временной ряд градиентов «сглаживается»

```
1 t = np.linspace(0, 20, 1000)
2 sig = np.sin(2 * np.pi * 0.5 * t) + np.random.random(t.shape) / 4
3 filtered = []
4 mean = 0
5 beta = 0.1
6 for s in sig:
7     mean = (1 - beta) * mean + beta * s
8     filtered.append(mean)
```



## Адаптивные методы - RMSprop



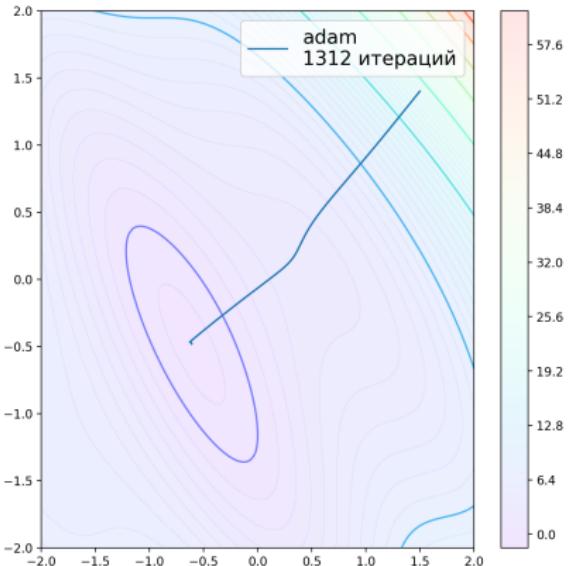
$$V_{t+1} = \alpha V_t + (1 - \alpha) \nabla f(x_t)^2$$

$$x_{t+1} = x_t - \gamma \frac{\nabla f(x_t)}{\sqrt{V_{t+1}} + \epsilon}$$

При вычислении  $V$  используем поэлементное возведение в квадрат. При вычислении  $x$  - поэлементное деление и взятие корня.

## Адаптивные методы - Adam

Скользящее среднее используется не только для оценки величины шага, но и для оценки «момента».



$$m_{t+1} = \beta V_t + (1 - \beta) \nabla f(x_t)$$

$$V_{t+1} = \alpha V_t + (1 - \alpha) \nabla f(x_t)^2$$

$$x_{t+1} = x_t - \gamma \frac{m_{t+1}}{\sqrt{V_{t+1}} + \epsilon}$$

Спасибо за внимание!