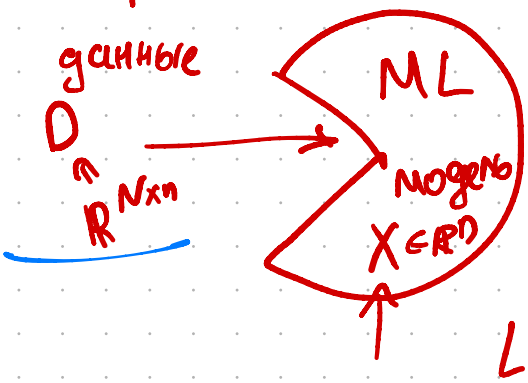
Градиентные методы в реальной жизни

$$f(x) = \frac{1}{N}\sum_{i=1}^{N} f_i(x) \to \min_{x \in \mathbb{R}^p}$$

N – # данных в датасете

p – # обучаемых параметров модели.



$$Loss = \frac{1}{N}\sum_{i=1}^{N}\left(y_{pred}(d_i, x) - y^i_{true}\right)^2 \to \min_{x \in \mathbb{R}^p}$$

$$y^i_{pred} = X^T \cdot d_i \to f(x) = \frac{1}{N}\sum_{i=1}^{N}\left(x^T d_i - y^i_{true}\right)^2.$$

GD $$\boxed{x_{k+1} = x_k - \alpha_k \cdot \nabla f(x_k)}$$

$$\nabla f(x_k) = \frac{1}{N}\sum_{i=1}^{N} \nabla f_i(x_k)$$

2. p параметров

Memory

$p \sim 10^{10} \sim 10^{11}$

$2 \cdot 10^{10} \cdot 16$ бит $\sim$

$\sim 10^9 \cdot 320$ бит

$\sim$ 320 Гбит

40 ГБ.

$N \sim 10^7$

- Классическое ускорение: **MOMENTUM**
- Как ускорить обучение модели с помощью SGD?

Beyond gradient descent, vol. 2

Концепция оптимальных методов, метод тяжёлого шарика ускоренный метод Нестерова

Схема получения оценок снизу на сложность методов и задач

- Фиксируем класс функций $\mathcal{F}$
- Фиксируем класс методов оптимизации $\mathcal{M}$
- Ищем настолько плохую функцию из класса $\mathcal{F}$, что любой метод из класса $\mathcal{M}$ сходится не лучше, чем некоторая оценка
- Такая оценка называется оценкой снизу

Пример

Фиксируем класс методов

Рассмотрим такие методы, что

$$x_{k+1} = x_0 + \mathrm{span}\{f'(x_0), \dots, f'(x_k)\}$$

- Далее в рамках этого семинара для краткости только такие методы будем называть методами первого порядка
- Весь последующий анализ **НЕ** применим, если

$$x_{k+1} = x_0 + G(f'(x_0), \dots, f'(x_k)),$$

где G - некоторая нелинейная функция

- С такими методами мы познакомимся на одном из ближайших занятий

Фиксируем класс функций

Выпуклые функции с липшицевым градиентом

Теорема. Существует выпуклая функция с Липшицевым градиентом, такая что

$$f(x_t) - f^* \geq \frac{3L\|x_0 - x^*\|_2^2}{32(t+1)^2},$$

нижняя оценка для методов 1-го порядка convex

где $x_k = x_0 + \mathrm{span}\{f'(x_0), \dots, f'(x_{k-1})\}, 1 \leq k \leq t$

Привести пример такой функции и доказать эту теорему Вам надо в домашнем задании.

Сильно выпуклые функции с липшицевым градиентом

Теорема. Существует сильно выпуклая функция с Липшицевым градиентом, такая что

$$f(x_t) - f^* \geq \frac{\mu}{2}\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^{2t}\|x_0 - x^*\|_2^2,$$

нижняя оц. μ сильно вып. $\kappa = \frac{L}{\mu}$ (L — λ_{max}, μ — λ_{min})

где $\kappa = \frac{L}{\mu}$ и $x_k = x_0 + \mathrm{span}\{f'(x_0), \dots, f'(x_{k-1})\}, 1 \leq k \leq t$

Привести пример такой функции и доказать эту теорему Вам надо в домашнем задании.

Оценки сходимости известных методов

Оценки сходимости для градиентного спуска: напоминание

- Пусть
  - $f(x)$ дифференцируема на $\mathbb{R}^n$
  - $f(x)$ выпукла
  - $f'(x)$ удовлетворяет условию Липшица с константой L
  - $\alpha = \dfrac{1}{L}$

    Тогда

    $$ f(x_k) - f^* \leq \dfrac{2L | x_0 - x^*|^2_2}{k+4} $$

$f(x_k) - f^* \leq \frac{2LR^2}{k+4}$ convex

$\alpha = \frac{1}{L}$

- Пусть
  - $f(x)$ дифференцируема на $\mathbb{R}^n$,
  - градиент $f(x)$ удовлетворяет условию Липшица с константой L
  - $f(x)$ является сильно выпуклой с константой μ
  - $\alpha = \dfrac{2}{\mu + L}$

    Тогда для градиентного спуска выполнено:

    $$ | x_k - x^* |_2 \leq \left(\dfrac{\kappa - 1}{\kappa + 1} \right)^k |x_0 - x^*|_2 $$

    $$f(x_k) - f^* \leq \dfrac{L}{2} \left(\dfrac{\kappa - 1}{\kappa + 1} \right)^{2k} | x_0 - x^*|^2_2, $$

    где $\kappa = \frac{L}{\mu}$

Оценки сходимости для метода сопряжённых градиентов: напоминание

Для сильно выпуклой квадратичной функции

$$f(x) = \frac{1}{2} x^\top A x - b^\top x$$

и метода сопряжённых градиентов справедлива следующая оценка сходимости

$$2(f_k - f^*) = \|x_k - x^*\|_A \leq 2\left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k \|x_0 - x^*\|_A,$$

где $\kappa(A) = \frac{\lambda_1(A)}{\lambda_n(A)} = \frac{L}{\mu}$ - число обусловленности матрицы A, $\lambda_1(A) \geq \ldots \geq \lambda_n(A) > 0$ - собственные значения матрицы A

Can we do better?

Существует ли метод, который сходится в соответствии с нижними оценками для

- произвольной сильно выпуклой функции с липшицевым градиентом (не только квадратичной)?
- произвольной выпуклой функции с липшицевым градиентом?

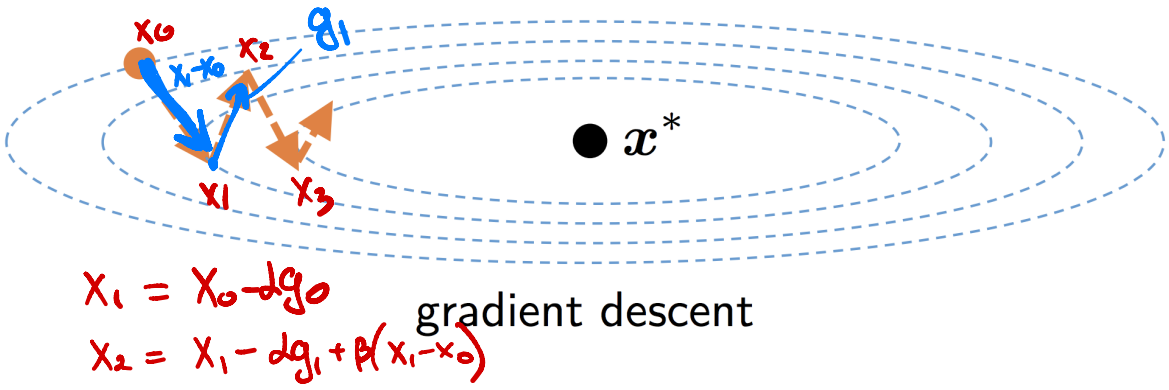
Метод тяжёлого шарика (heavy-ball method)

- Предложен в 1964 г. Б.Т. Поляком
- Для квадратичной целевой функции зигзагообразное поведение градиентного спуска обусловлено неоднородностью направлений
- Давайте учитывать предыдущие направления для поиска новой точки
- Метод тяжёлого шарика

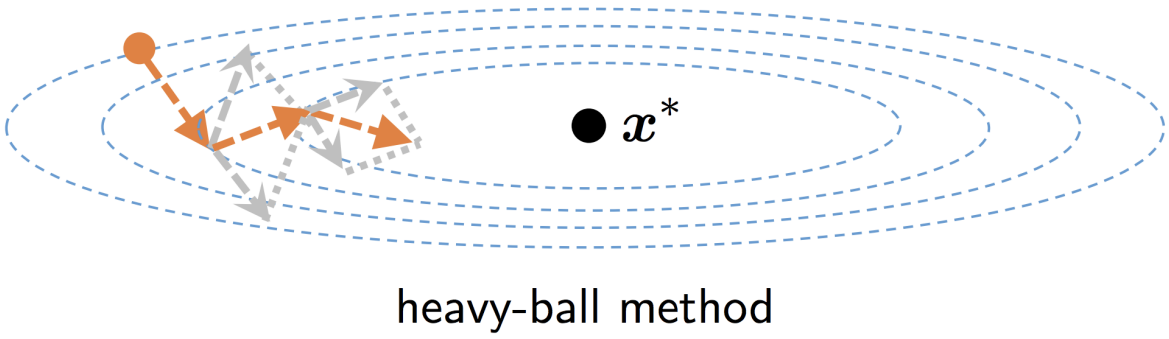
$$x_{k+1} = x_k - \alpha_k f'(x_k) + \beta_k (x_k - x_{k-1})$$

- Помимо параметра шага вдоль антиградиента α_k появился ещё один параметр β_k

Геометрическая интерпретация метода тяжёлого шарика



Картинка [отсюда]



Теорема сходимости

Пусть f сильно выпукла с Липшицевым градиентом. Тогда для

$$\alpha_k = \frac{4}{(\sqrt{L} + \sqrt{\mu})^2}$$

и

$$\beta_k = \max(|1 - \sqrt{\alpha_k L}|^2, |1 - \sqrt{\alpha_k \mu}|^2)$$

справедлива следующая оценка сходимости

$$\left\| \begin{bmatrix} x_{k+1} - x^* \\ x_k - x^* \end{bmatrix} \right\|_2 \leq \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^k \left\| \begin{bmatrix} x_1 - x^* \\ x_0 - x^* \end{bmatrix} \right\|_2$$

- Совпадает с оценкой снизу для методов первого порядка!
- Оптимальные параметры α_k и β_k определяются через **неизвестные** константы L и μ

Схема доказательства

- Перепишем метод как

$$\begin{bmatrix} x_{k+1} \\ x_k \end{bmatrix} = \begin{bmatrix} (1 + \beta_k)I & -\beta_k I \\ I & 0 \end{bmatrix} \begin{bmatrix} x_k \\ x_{k-1} \end{bmatrix} + \begin{bmatrix} -\alpha_k f'(x_k) \\ 0 \end{bmatrix}$$

- Используем теорему из анализа

$$\begin{bmatrix} x_{k+1} - x^* \\ x_k - x^* \end{bmatrix} = \underbrace{\begin{bmatrix} (1 + \beta_k)I - \alpha_k \int_0^1 f''(x(\tau))d\tau & -\beta_k I \\ I & 0 \end{bmatrix}}_{=A_k} \begin{bmatrix} x_k - x^* \\ x_{k-1} - x^* \end{bmatrix},$$

где $x(\tau) = x_k + \tau(x^* - x_k)$

- В силу интегральной теоремы о среднем $A_k(x) = \int_0^1 f''(x(\tau))d\tau = f''(z)$, поэтому L и μ ограничивают спектр $A_k(x)$
- Сходимость зависит от спектрального радиуса матрицы итераций A_k

- Получим оценку на спектр A_k

$$A_k = \begin{bmatrix} (1 + \beta_k)I - \alpha_k A(x_k) & -\beta_k I \\ I & 0 \end{bmatrix}$$

- Пусть $A(x_k) = U\Lambda(x_k)U^\top$, поскольку гессиан - симметричная матрица, тогда

$$\begin{bmatrix} U^\top & 0 \\ 0 & U^\top \end{bmatrix} \begin{bmatrix} (1 + \beta_k)I - \alpha_k A(x_k) & -\beta_k I \\ I & 0 \end{bmatrix} \begin{bmatrix} U & 0 \\ 0 & U \end{bmatrix} = \begin{bmatrix} (1 + \beta_k)I - \alpha_k \Lambda(x_k) & -\beta_k I \\ I & 0 \end{bmatrix} = \hat{A}_k$$

- Ортогональное преобразование не меняет спектральный радиус матрицы
- Далее сделаем перестановку строк и столбцов так, чтобы

$$\hat{A}_k \simeq \operatorname{diag}(T_1, \dots, T_n),$$

где $T_i = \begin{bmatrix} 1 + \beta_k - \alpha_k \lambda_i & -\beta_k \\ 1 & 0 \end{bmatrix}$ и $\simeq$ обозначает равенство спектральных радиусов поскольку матрица перестановки является ортогональной

- Покажем как сделать такую перестановку на примере матрицы 4×4

$$\begin{bmatrix} a & 0 & c & 0 \\ 0 & b & 0 & c \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \to \begin{bmatrix} a & 0 & c & 0 \\ 1 & 0 & 0 & 0 \\ 0 & b & 0 & c \\ 0 & 1 & 0 & 0 \end{bmatrix} \to \begin{bmatrix} a & c & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & b & c \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

- Свели задачу к оценке спектрального радиуса блочно-диагональной матрицы $\hat{A}_k$

- $\rho(\hat{A}_k) = \max\limits_{i=1,\dots,n} \{|\lambda_1(T_i)|, |\lambda_2(T_i)|\}$
- Характеристическое уравнение для T_i

$$\beta_k - u(1 + \beta_k - \alpha_k \lambda_i - u) = 0 \quad u^2 - u(1 + \beta_k - \alpha_k \lambda_i) + \beta_k = 0$$

- Дальнейшее изучение распределения корней и их границ даёт оценку из условия теоремы

Эксперименты

Тестовая задача 1

$$f(x) = \frac{1}{2} x^\top A x - b^\top x \to \min_x,$$

где матрица A плохо обусловлена, но положительно определена!

```
import liboptpy.base_optimizer as base
import numpy as np
import liboptpy.unconstr_solvers.fo as fo
import liboptpy.step_size as ss
import matplotlib.pyplot as plt
%matplotlib inline

plt.rc("text", usetex=True)

class HeavyBall(base.LineSearchOptimizer):
    def __init__(self, f, grad, step_size, beta, **kwargs):
        super().__init__(f, grad, step_size, **kwargs)
        self._beta = beta

    def get_direction(self, x):
        self._current_grad = self._grad(x)
        return -self._current_grad

    def _f_update_x_next(self, x, alpha, h):
        if len(self.convergence) < 2:
            return x + alpha * h
        else:
            return x + alpha * h + self._beta * (x - self.convergence[-2])

    def get_stepsize(self):
        return self._step_size.get_stepsize(self._grad_mem[-1], self.convergence[-1], len(self.convergence))
```

```
np.random.seed(42)
n = 100
A = np.random.randn(n, n)
A = A.T.dot(A)
x_true = np.random.randn(n)
b = A.dot(x_true)
f = lambda x: 0.5 * x.dot(A.dot(x)) - b.dot(x)
grad = lambda x: A.dot(x) - b
A_eigvals = np.linalg.eigvalsh(A)
L = np.max(A_eigvals)
mu = np.min(A_eigvals)
print(L, mu)
print("Condition number = {}".format(L / mu))
alpha_opt = 4 / (np.sqrt(L) + np.sqrt(mu))**2
beta_opt = np.maximum((1 - np.sqrt(alpha_opt * L))**2,
                      (1 - np.sqrt(alpha_opt * mu))**2)
```

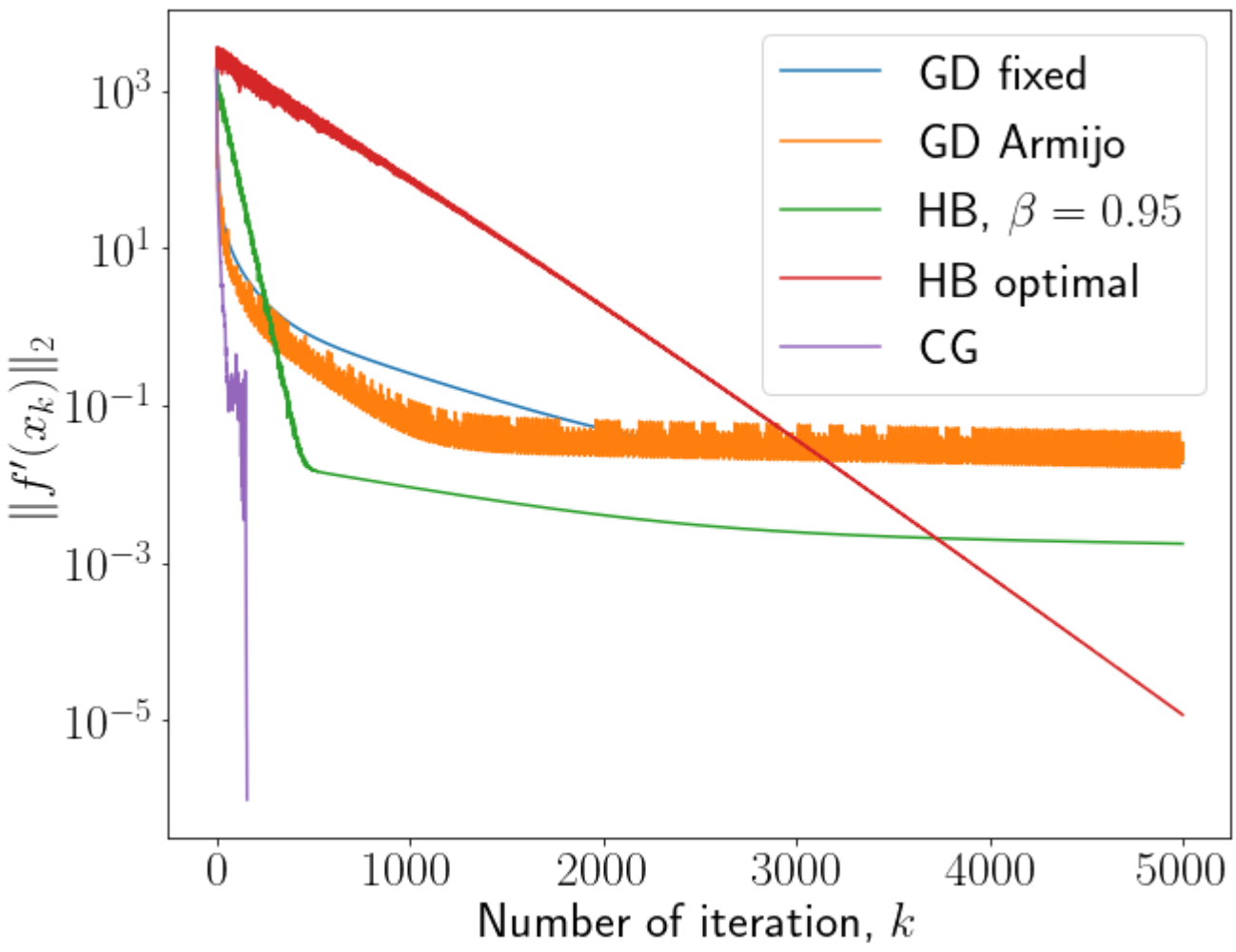
```
print(alpha_opt, beta_opt)
beta_test = 0.95
```

```
388.0536004941595 0.0017707060804411218
Condition number = 219151.89922287207
0.010263957266938104 0.9914918736025333
```

```
methods = {
    "GD fixed": fo.GradientDescent(f, grad, ss.ConstantStepSize(1 / L)),
    "GD Armijo": fo.GradientDescent(f, grad,
            ss.Backtracking("Armijo", rho=0.5, beta=0.1, init_alpha=1.)),
    r"HB, $\beta = {}$".format(beta_test): HeavyBall(f, grad, ss.ConstantStepSize(1 / L), beta=beta_test),
    "HB optimal": HeavyBall(f, grad, ss.ConstantStepSize(alpha_opt), beta = beta_opt),
    "CG": fo.ConjugateGradientQuad(A, b)
}
x0 = np.random.randn(n)
max_iter = 5000
tol = 1e-6
```

```
for m in methods:
    _ = methods[m].solve(x0=x0, max_iter=max_iter, tol=tol)
```

```
figsize = (10, 8)
fontsize = 26
plt.figure(figsize=figsize)
for m in methods:
    plt.semilogy([np.linalg.norm(grad(x)) for x in methods[m].get_convergence()], label=m)
plt.legend(fontsize=fontsize, loc="best")
plt.xlabel("Number of iteration, $k$", fontsize=fontsize)
plt.ylabel(r"$\| f'(x_k)\|_2$", fontsize=fontsize)
plt.xticks(fontsize=fontsize)
_ = plt.yticks(fontsize=fontsize)
```




```
for m in methods:
    print(m)
    %timeit methods[m].solve(x0=x0, max_iter=max_iter, tol=tol)
```

```
GD fixed
518 ms ± 87.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
GD Armijo
5.42 s ± 499 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
HB, $\beta = 0.95$
322 ms ± 44.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
HB optimal
299 ms ± 24 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
CG
22.6 ms ± 1.66 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Тестовая задача 2

$$f(w) = \frac{1}{2}\|w\|_2^2 + C\frac{1}{m}\sum_{i=1}^{m}\log(1 + \exp(-y_i\langle x_i, w\rangle)) \to \min_w$$

```
n = 300
m = 1000
import sklearn.datasets as skldata
import jax
import jax.numpy as jnp
import scipy.optimize as scopt
from jax.config import config
config.update("jax_enable_x64", True)

X, y = skldata.make_classification(n_classes=2, n_features=n, n_samples=m, n_informative=n//3)
C = 1

@jax.jit
def f(w):
    return jnp.linalg.norm(w)**2 / 2 +  C * jnp.mean(jnp.logaddexp(jnp.zeros(X.shape[0]), -y * (X @ w)))

# def grad(w):
#     denom = scspec.expit(-y * X.dot(w))
#     return w - C * X.T.dot(y * denom) / X.shape[0]

autograd_f = jax.jit(jax.grad(f))
x0 = jnp.ones(n)
print("Initial function value = {}".format(f(x0)))
print("Initial gradient norm = {}".format(jnp.linalg.norm(autograd_f(x0))))
```

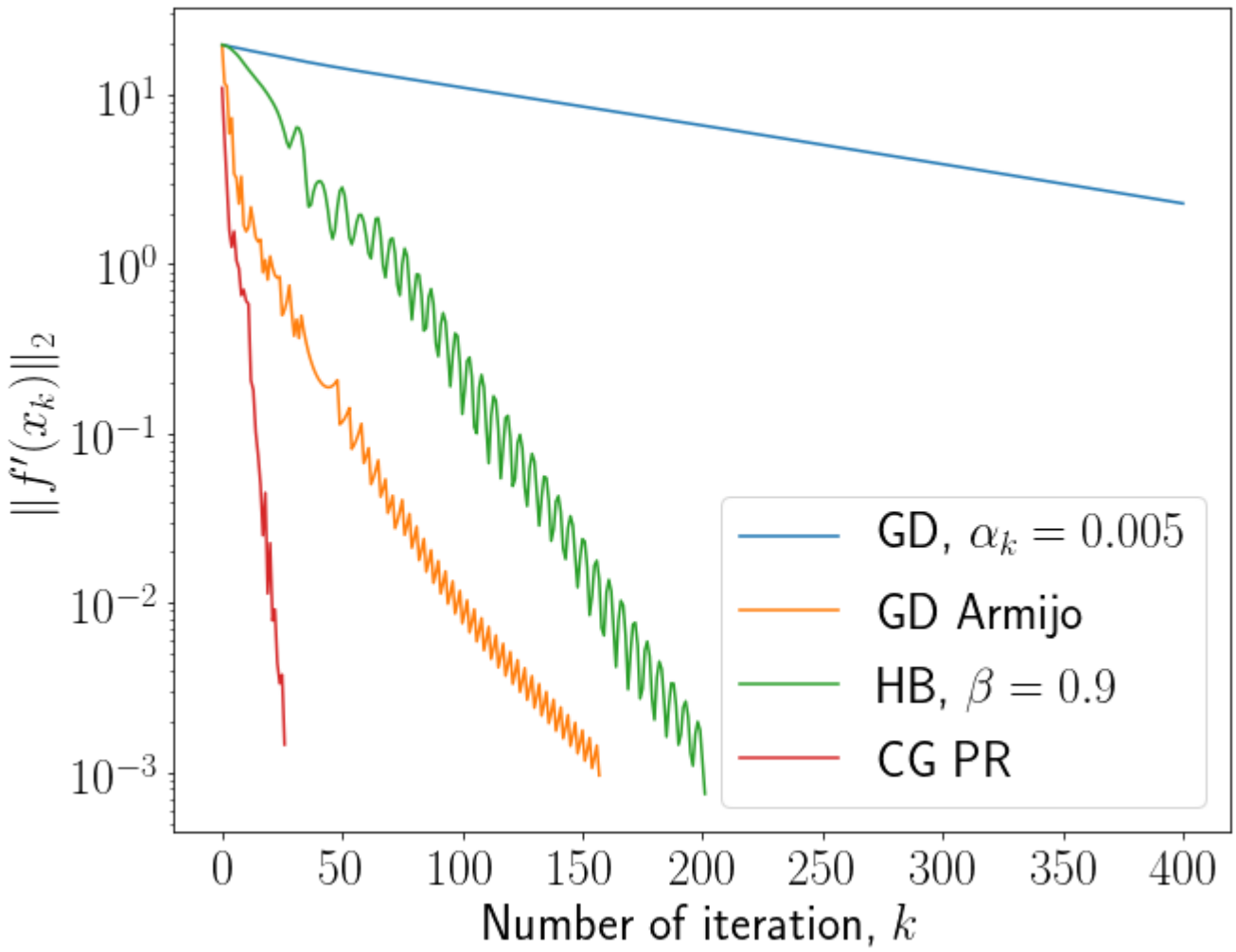
```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun for more info.)
Initial function value = 165.04873928789436
Initial gradient norm = 18.646187895005486
```

```
alpha_test = 5e-3
beta_test = 0.9

methods = {
    r"GD, $\alpha_k = {}$".format(alpha_test): fo.GradientDescent(f, autograd_f, ss.ConstantStepSize(alpha_test)),
    "GD Armijo": fo.GradientDescent(f, autograd_f,
            ss.Backtracking("Armijo", rho=0.7, beta=0.1, init_alpha=1.)),
    r"HB, $\beta = {}$".format(beta_test): HeavyBall(f, autograd_f, ss.ConstantStepSize(alpha_test), beta=beta_test),

}
# x0 = np.random.rand(n)
# x0 = jnp.zeros(n)
x0 = jax.random.normal(jax.random.PRNGKey(0), (X.shape[1],))
max_iter = 400
tol = 1e-3
```

```
for m in methods:
    print(m)
    _ = methods[m].solve(x0=x0, max_iter=max_iter, tol=tol)

scopt_cg_array = []
def callback(x, arr):
    arr.append(x)

scopt_cg_callback = lambda x: callback(x, scopt_cg_array)
x = scopt.minimize(f, x0, tol=tol, method="CG", jac=autograd_f, callback=scopt_cg_callback, options={"maxiter": max_ite
x = x.x
```

```
GD, $\alpha_k = 0.005$
GD Armijo
HB, $\beta = 0.9$
```

```
figsize = (10, 8)
fontsize = 26
plt.figure(figsize=figsize)
for m in methods:
    plt.semilogy([np.linalg.norm(autograd_f(x)) for x in methods[m].get_convergence()], label=m)
plt.semilogy([np.linalg.norm(autograd_f(x)) for x in scopt_cg_array], label="CG PR")
plt.legend(fontsize=fontsize, loc="best")
plt.xlabel("Number of iteration, $k$", fontsize=fontsize)
plt.ylabel(r"$\| f'(x_k)\|_2$", fontsize=fontsize)
plt.xticks(fontsize=fontsize)
_ = plt.yticks(fontsize=fontsize)
```




```
1 for m in methods:
2     print(m)
3     %timeit methods[m].solve(x0=x0, max_iter=max_iter, tol=tol)
```

```
GD, $\alpha_k = 0.001$
53.1 ms ± 11.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
GD Armijo
613 ms ± 32.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
HB, $\beta = 0.9$
57.2 ms ± 11.9 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Главное про метод тяжёлого шарика

- Двухшаговый метод
- Не обязательно монотонный
- Параметры зависят от неизвестных констант
- Решает проблему осцилляций для плохо обусловленных задач
- Сходимость для сильно выпуклых функций совпадает с оптимальной оценкой

Ускоренный метод Нестерова

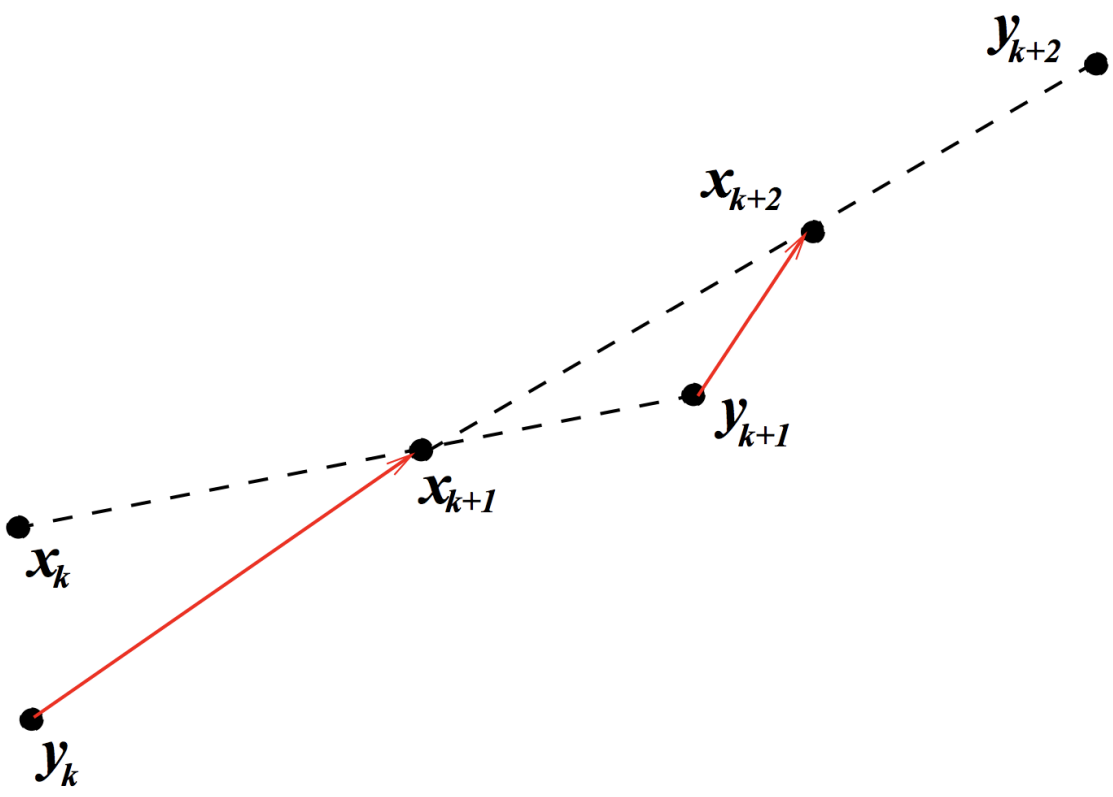
- Предложен в 1983 г. Ю.Е. Нестеровым 
- Одна из возможных форм записи

$$
\begin{aligned}
y_0 &= x_0 \\
x_{k+1} &= y_k - \alpha_k f'(y_k) \\
y_{k+1} &= x_{k+1} + \frac{k}{k+3}(x_{k+1} - x_k)
\end{aligned}
$$

- Сравните с методом тяжёлого шарика
- Также не обязательно монотонен

- Для любителей геометрии есть альтернативный метод под названием [geometric descent](https://arxiv.org/abs/1208.3461) с такой же скоростью сходимости

Геометрическая интерпретация ускоренного метода Нестерова



Теорема сходимости

- Пусть f выпукла с Липшицевым градиентом, а шаг $\alpha_k = \frac{1}{L}$. Тогда ускоренный метод Нестерова сходится как

$$f(x_k) - f^* \leq \frac{2L\|x_0 - x^*\|_2^2}{(k+1)^2}$$

- Пусть f сильно выпукла с липшицевым градиентом. Тогда ускоренный метод Нестерова при шаге $\alpha_k = \frac{1}{L}$ сходится как

$$f(x_k) - f^* \leq L\|x_k - x_0\|_2^2 \left(1 - \frac{1}{\sqrt{\kappa}}\right)^k$$

Тестовая задача

$$f(x) = \frac{1}{2}x^\top Ax - b^\top x \to \min_x,$$

где матрица A положительно полуопределённая, то есть функция **НЕ** является сильно выпуклой

```
import numpy as np
import liboptpy.unconstr_solvers.fo as fo
import liboptpy.step_size as ss
import liboptpy.base_optimizer as base
import matplotlib.pyplot as plt
%matplotlib inline

np.random.seed(42)
n = 100
A = np.random.randn(n, n)
A = A.T.dot(A)
A_eigvals = np.linalg.eigvalsh(A)
mu = np.min(A_eigvals)
A = A - (mu - 1e-6) * np.eye(n)
x_true = np.random.randn(n)
b = A.dot(x_true)
f = lambda x: 0.5 * x.dot(A.dot(x)) - b.dot(x)
grad = lambda x: A.dot(x) - b
A_eigvals = np.linalg.eigvalsh(A)
L = np.max(A_eigvals)
mu = np.min(A_eigvals)
print(L, mu)
```

```
388.0518307880791 9.999999702771782e-07
```

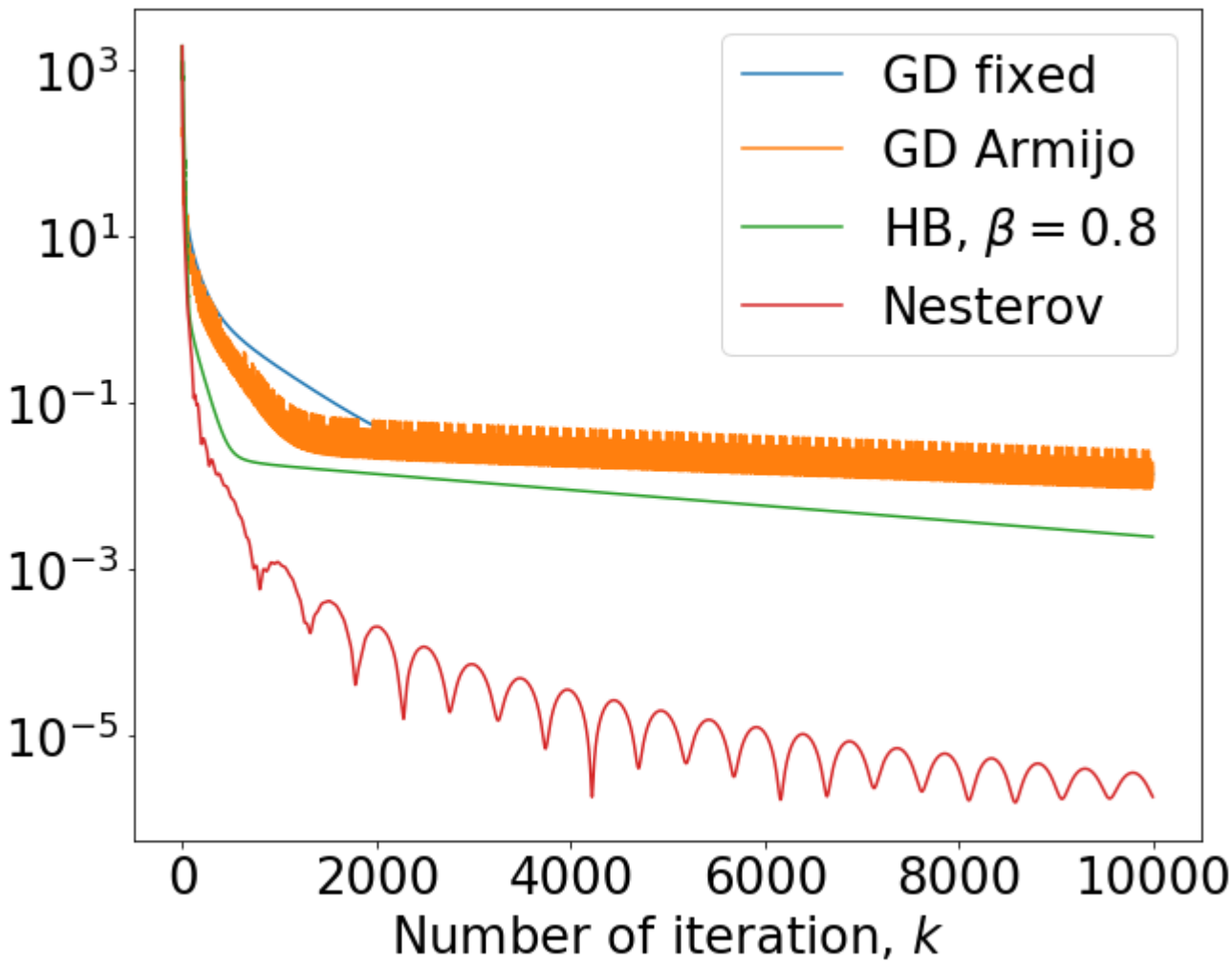
```
class HeavyBall(base.LineSearchOptimizer):
    def __init__(self, f, grad, step_size, beta, **kwargs):
```

```
        super().__init__(f, grad, step_size, **kwargs)
        self._beta = beta

    def get_direction(self, x):
        self._current_grad = self._grad(x)
        return -self._current_grad

    def _f_update_x_next(self, x, alpha, h):
        if len(self.convergence) < 2:
            return x + alpha * h
        else:
            return x + alpha * h + self._beta * (x - self.convergence[-2])

    def get_stepsize(self):
        return self._step_size.get_stepsize(self._grad_mem[-1], self.convergence[-1], len(self.convergence))
```

```
beta_test = 0.8
methods = {
    "GD fixed": fo.GradientDescent(f, grad, ss.ConstantStepSize(1 / L)),
    "GD Armijo": fo.GradientDescent(f, grad,
            ss.Backtracking("Armijo", rho=0.5, beta=0.1, init_alpha=1.)),
    r"HB, $\beta = {}$".format(beta_test): HeavyBall(f, grad, ss.ConstantStepSize(1 / L), beta=beta_test),
    "Nesterov": fo.AcceleratedGD(f, grad, ss.ConstantStepSize(1 / L)),
}
x0 = np.random.randn(n)
max_iter = 10000
tol = 1e-6
```

```
for m in methods:
    _ = methods[m].solve(x0=x0, max_iter=max_iter, tol=tol)
```

```
figsize = (10, 8)
fontsize = 26
plt.figure(figsize=figsize)
for m in methods:
    plt.semilogy([np.linalg.norm(grad(x)) for x in methods[m].get_convergence()], label=m)
#     plt.semilogy([np.linalg.norm(x - x_true) for x in methods[m].get_convergence()], label=m)
#     plt.semilogy([f(x) - f(x_true) for x in methods[m].get_convergence()], label=m)
plt.legend(fontsize=fontsize, loc="best")
plt.xlabel("Number of iteration, $k$", fontsize=fontsize)
# plt.ylabel(r"$\| f'(x_k)\|_2$", fontsize=fontsize)
plt.xticks(fontsize=fontsize)
_ = plt.yticks(fontsize=fontsize)
```

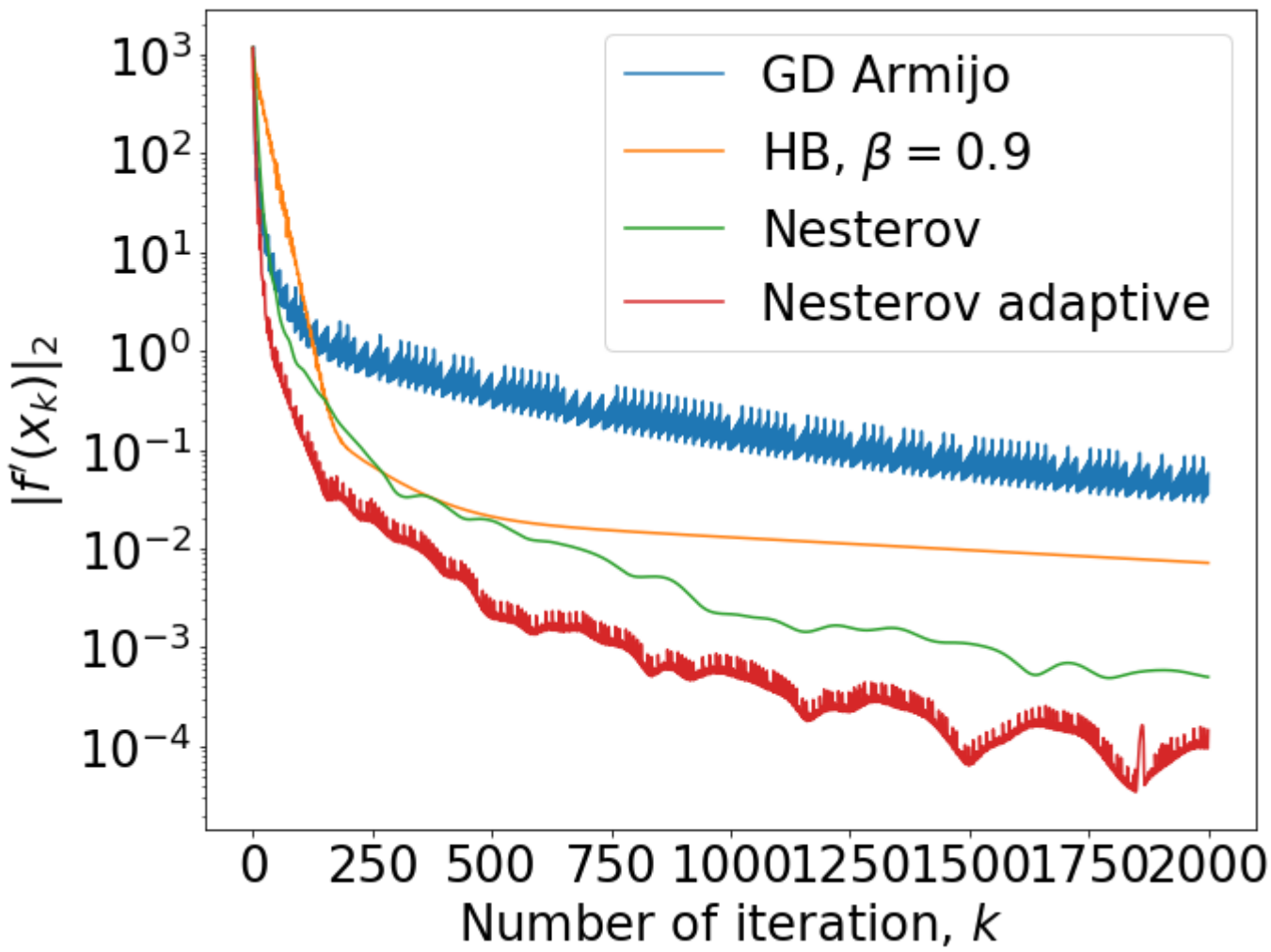



```
for m in methods:
    print(m)
    %timeit methods[m].solve(x0=x0, max_iter=max_iter, tol=tol)
```

```
GD fixed
```

```
for m in methods:
    _ = methods[m].solve(x0=x0, max_iter=max_iter, tol=tol)
```

```
figsize = (10, 8)
fontsize = 26
plt.figure(figsize=figsize)
for m in methods:
    plt.semilogy([np.linalg.norm(grad(x)) for x in methods[m].get_convergence()], label=m)
plt.legend(fontsize=fontsize, loc="best")
plt.xlabel("Number of iteration, $k$", fontsize=fontsize)
plt.ylabel(r"$\| f'(x_k)\|_2$", fontsize=fontsize)
plt.xticks(fontsize=fontsize)
_ = plt.yticks(fontsize=fontsize)
```




```
for m in methods:
    print(m)
    %timeit methods[m].solve(x0=x0, max_iter=max_iter, tol=tol)
```

```
GD Armijo
402 ms ± 2.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
HB, $\beta = 0.9$
43.4 ms ± 949 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
Nesterov
52.5 ms ± 1.15 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
Nesterov adaptive
209 ms ± 7.13 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Эксперимент на неквадратичной задаче

$$f(w) = \frac{1}{2}\|w\|_2^2 + C\frac{1}{m}\sum_{i=1}^{m}\log(1 + \exp(-y_i\langle x_i, w\rangle)) \to \min_w$$

```
import jax
import jax.numpy as jnp
from jax.config import config
config.update("jax_enable_x64", True)
import sklearn.datasets as skldata
n = 300
m = 1000

X, y = skldata.make_classification(n_classes=2, n_features=n, n_samples=m, n_informative=n//3, random_state=42)
C = 1

@jax.jit
def f(w):
    return jnp.linalg.norm(w)**2 / 2 +  C * jnp.mean(jnp.logaddexp(jnp.zeros(X.shape[0]), -y * (X @ w)))

```

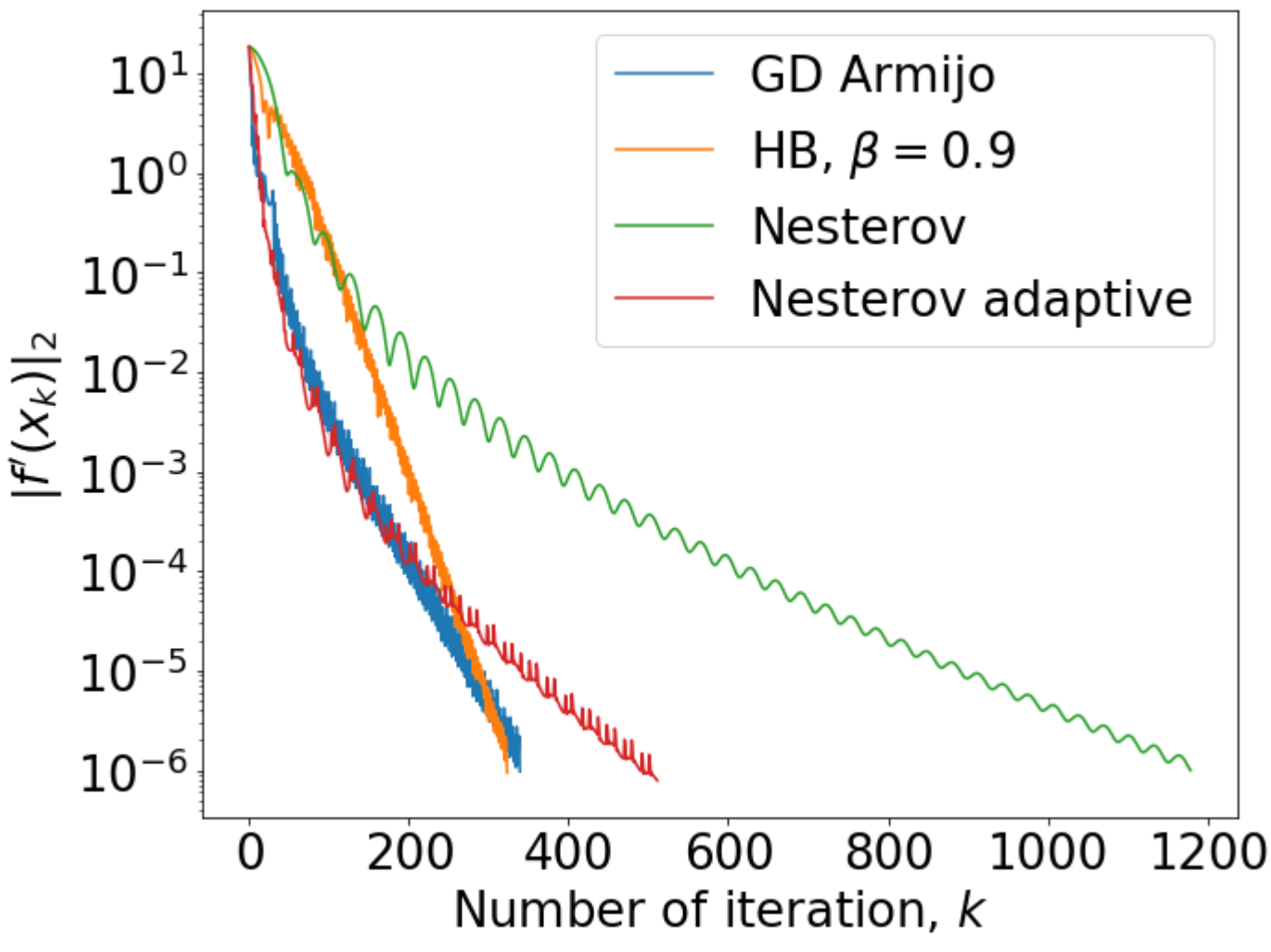
```
# def grad(w):
#     denom = scspec.expit(-y * X.dot(w))
#     return w - C * X.T.dot(y * denom) / X.shape[0]

autograd_f = jax.jit(jax.grad(f))
x0 = jnp.ones(n)
print("Initial function value = {}".format(f(x0)))
print("Initial gradient norm = {}".format(jnp.linalg.norm(autograd_f(x0))))
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun for more info.)
Initial function value = 162.63079545367793
Initial gradient norm = 18.723119978580637
```

```
beta_test = 0.9
L_trial = (1 + C*100)
methods = {
    "GD Armijo": fo.GradientDescent(f, autograd_f,
            ss.Backtracking("Armijo", rho=0.5, beta=0.01, init_alpha=1.)),
    r"HB, $\beta = {}$".format(beta_test): HeavyBall(f, autograd_f, ss.ConstantStepSize(1 / L_trial), beta=beta_test),
    "Nesterov": fo.AcceleratedGD(f, autograd_f, ss.ConstantStepSize(1 / L_trial)),
    "Nesterov adaptive": fo.AcceleratedGD(f, autograd_f, ss.Backtracking(rule_type="Lipschitz", rho=0.5, init_alpha=1))
}
max_iter = 2000
tol = 1e-6
```

```
for m in methods:
    _ = methods[m].solve(x0=x0, max_iter=max_iter, tol=tol)
```

```
figsize = (10, 8)
fontsize = 26
plt.figure(figsize=figsize)
for m in methods:
    plt.semilogy([np.linalg.norm(autograd_f(x)) for x in methods[m].get_convergence()], label=m)
plt.legend(fontsize=fontsize, loc="best")
plt.xlabel("Number of iteration, $k$", fontsize=fontsize)
plt.ylabel(r"$\| f'(x_k)\|_2$", fontsize=fontsize)
plt.xticks(fontsize=fontsize)
_ = plt.yticks(fontsize=fontsize)
```




```
for m in methods:
    print(m)
    %timeit methods[m].solve(x0=x0, max_iter=max_iter, tol=tol)
```

```
GD Armijo
3.9 s ± 259 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
HB, $\beta = 0.9$
379 ms ± 8.48 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
Nesterov
2.45 s ± 108 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
Nesterov adaptive
9.77 s ± 1.21 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Выводы

- Нижние оценки для методов первого порядка
- Метод тяжёлого шарика
- Ускоренный метод Нестерова
- Поиск константы Липшица градиента L