Домашнее задание 2.

Присылайте .pdf файл, сохраненный из этого ноутбука (можно просто нажать `Ctrl + P`) на `homework@merkulov.top`.

Дедлайн: 30.04.21 23:59:59

Adaptive metrics methods

Affine invariance

Метод Ньютона:

$$x_{k+1} = x_k - [f_{xx}(x_k)]^{-1}\nabla f(x_k) = x_k - [H(x_k)]^{-1}\nabla f(x_k) = x_k - B(x_k)\nabla f(x_k)$$

SR-1 (symmetric rank one) update

Для квазиньютоновского метода использует следующую формулу для уточнения **обратного** гессиана:

$$B_{k+1} = B_k + \frac{(\Delta x_k - B_k\Delta y_k)(\Delta x_k - B_k\Delta y_k)^\top}{\langle \Delta x_k - B_k\Delta y_k, \Delta y_k\rangle}, \quad \Delta y_k = \nabla f(x_{k+1}) - \nabla f(x_k), \quad \Delta x_k = x_{k+1} - x_k$$

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

Оценка гессиана при этом:

$$H_{k+1} = H_k + \frac{(\Delta y_k - H_k\Delta x_k)(\Delta y_k - H_k\Delta x_k)^\top}{(\Delta y_k - H_k\Delta x_k)^\top\Delta x_k}$$

BFGS

использует следующую формулу для уточнения **обратного** гессиана:

_ _ _ _ _

$$B_{k+1} = B_k + \frac{(\Delta x_k^\top \Delta y_k + \Delta y_k^\top B_k \Delta y_k)(\Delta x_k \Delta x_k^\top)}{(\Delta x_k^\top \Delta y_k)^2} - \frac{B_k \Delta y_k \Delta x_k^\top + \Delta x_k \Delta y_k^\top B_k}{\Delta x_k^\top \Delta y_k}.$$

Оценка гессиана при этом:

$$H_{k+1} = H_k + \frac{\Delta y_k \Delta y_k^\top}{\Delta y_k^\top \Delta x_k} - \frac{H_k \Delta x_k \Delta x_k^\top H_k^\top}{\Delta x_k^\top H_k \Delta x_k}$$

Докажите, что для метода Ньютона обладает афинной инвариантностью, т.е. если есть преобразование координат $\tilde{f}(z) = f(x)$, где $x = Sz + s, \quad s \in \mathbb{R}^n, S \in \mathbb{R}^{n \times n}$, то будет выполняться:

$$\nabla \tilde{f}(z) = S^\top \nabla f(x), \qquad \nabla^2 \tilde{f}(z) = S^\top \nabla^2 f(x) S$$

Покажите так же, что метод Ньютона и описанные выше оба квазиньютоновских метода запущенные независимо по координатам x и z будут работать так, что всегда будет выполняться связь $x_k = Sz_k + s$, если $x_0 = Sz_0 + s$ и

==YOUR ANSWER==

$\tilde{f}(z) = f(x), x_0 = Sz_0 + s$

Сначала покажем, что градиенты преобразуются так как написано в условии. Просто применим матричное дифференцирование, получим:

$$\nabla \tilde{f}(\bar{z}) = \frac{\partial \tilde{f}}{\partial \bar{z}} = \frac{\partial f}{\partial \bar{x}} \cdot \frac{\partial \bar{x}}{\partial \bar{z}} = \nabla f \cdot \frac{\partial (s\bar{z} + \bar{s})}{\partial \bar{z}} = S^\top \nabla f(x)$$
$$\nabla^2 \tilde{f}(\bar{z}) = \frac{\partial^2 \tilde{f}}{\partial z^2} = \frac{\partial (S^\top \nabla f)}{\partial \bar{x}} \frac{\partial \bar{x}}{\partial \bar{z}} = S^\top \nabla^2 f \cdot S$$

Получили то, что нужно. Теперь покажем, что при запуске методов независимо по координатам x и z будут работать так, что всегда будет выполняться связь $x_k = Sz_k + s$, если $x_0 = Sz_0 + s$ и инициализацией H_0 для метода по координате x и $S^\top H_0 S$ для координаты z

Выразим градиент $f(x)$ через градмент $\tilde{f}(z)$ из 1го выражения:

$$\nabla f(x) = \left(S^T\right)^{-1} \nabla \bar{f}(z)$$

Из 2го получаем, как преобразуется гессиан и матрица B:

$$\tilde{H} = S^\top B S$$

Значит:

$$\tilde{B} = S^{-1} B (S^\top)^{-1}, \quad B = S \tilde{B} S^T$$

Знаем, что координата x преобразуется как:

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

а координата z:

$$z_{k+1} = z_k - \tilde{B}_k \nabla \tilde{f}\,(x_k)$$

По индукции докажем, что для любого k выполнено

$$x_k = S z_k + s$$

База индукции уже выполнена (так как точки старта (k = 0) связаны именно так).

Предположение индукции: пусть для $k-1$ верно, докажем для k. Чтобы показать это, нужно доказать, что матрицы B и $\tilde{B}$ на произвольном шаге k связаны как:

$$B_k = S \tilde{B}_k S^T$$

Это сделаем тоже по индукции для каждого из методов по отдельности (база индукции выполнена: см. выше, выразили из преобразования гессиана)

Для SR-1 (symmetric rank one) update

$$B_{k+1} = B_k + \frac{(\Delta x_k - B_k \Delta y_k)(\Delta x_k - B_k \Delta y_k)^\top}{\langle \Delta x_k - B_k \Delta y_k, \Delta y_k \rangle}, \quad \Delta y_k = \nabla f(x_{k+1}) - \nabla f(x_k), \quad \Delta x_k = x_{k+1} - x_k$$

Хотим показать: $B_k = S \tilde{B}_k S^T$, пусть для $k-1$ выполнено, покажем для k:

Заметим, что:

$$B_k \nabla f = S \tilde{B}_k S^T \left(S^T\right)^{-1} \nabla \tilde{f} = S \left(\tilde{B}_k \nabla \tilde{f} \right)$$

$$B_k \Delta y = S \left(\tilde{B}_k \Delta \tilde{y} \right)$$

Тогда по формуле из метода:

$$B_{k+1} = S \tilde{B}_k S^T + \frac{\left(S \Delta z_k - S \tilde{B}_x S^T \left(S^\top\right)^{-1} \Delta \tilde{y}_k\right) \left(S \Delta z_k - S \tilde{B}_x S^T \left(S^\top\right)^{-1} \Delta \tilde{y}_k\right)^T}{\left\langle S \Delta z_k - S \bar{B}_k S^\top (S^T)^{-1} \Delta \tilde{y}_k, (S^T)^{-1} \Delta \bar{y}_k \right\rangle}$$

$$= S \left(\tilde{B}_k + \frac{\left(\Delta z_k - \tilde{B}_n \Delta \tilde{y}_k\right) \left(\Delta z_k - \widetilde{B}_k \Delta \bar{y}_k\right)}{\left\langle S^{-1} S \left(\Delta z_k - \widetilde{B}_k \Delta \tilde{y}_k\right), \Delta \tilde{y}_k \right\rangle} \right) S^\top = S \tilde{B_{k+1}} S^T$$

Проверили для 1го метода. Аналогично сделаем для 2го

Для BFGS

$$B_{k+1} = B_k + \frac{(\Delta x_k^\top \Delta y_k + \Delta y_k^\top B_k \Delta y_k)(\Delta x_k \Delta x_k^\top)}{(\Delta x_k^\top \Delta y_k)^2} - \frac{B_k \Delta y_k \Delta x_k^\top + \Delta x_k \Delta y_k^\top B_k}{\Delta x_k^\top \Delta y_k}.$$

$$B_{k+1} = S \tilde{B}_k S^T + \frac{((S \Delta z)^\top \left(S^\top\right)^{-1} \Delta \tilde{y}_k + \left(\left(S^\top\right)^{-1} \Delta \tilde{y}_k\right)^\top S \tilde{B}_k \Delta \tilde{y}_k)((S \Delta z_k)(S \Delta z_k)^T)}{\left((S \Delta z)^\top (S^\top)^{-1} \Delta \tilde{y}_k\right)^2}$$

$$- \frac{S \tilde{B}_k \Delta \tilde{y}_k (S_{\Delta z_k})^\top + S \Delta z_k \left(\left(S^\top\right)^{-1} \Delta \tilde{y}_k\right)^\top S \tilde{B} S^\top}{(S \Delta z_k)^\top (S)^{-1} \Delta \tilde{y}_k}$$

$$= S \tilde{B}_k S^T + \frac{\left(\Delta z^\top \Delta \tilde{y}_k + \Delta \tilde{y}_k^\top \tilde{B} \Delta \tilde{y}_k\right) (S \Delta z_k \Delta z_k^\top S^\top)}{\left(\Delta z_k^\top \Delta \tilde{y}_k\right)^2} - \frac{S \left(\tilde{B}_k \Delta \tilde{y}_k \Delta z_k^\top + \Delta z_k \Delta \tilde{y_k}^\top \tilde{B}_k\right) S^\top}{\Delta z_k^T \Delta \tilde{y}_k}$$

$$= S\left(\tilde{B}_k + \frac{\left(\Delta z_k^T \Delta\tilde{y}_k + \Delta\tilde{y}_k^\top \tilde{B}_k \Delta\tilde{y}_k\right)\left(\Delta z_k \Delta z_k{}^\top\right)}{\left(\Delta z_k \Delta\tilde{y_k}\right)^2} - \frac{\tilde{B}_k \Delta\tilde{y}_k \Delta z_k^T + \Delta z_k \Delta\tilde{y}^\top \tilde{B}_k}{\Delta z_k^\top \Delta\tilde{y}}\right) S^T = S\tilde{B_{k+1}} S^T$$

После долгих и не очень сложных выкладок снова получили то, что нужно

С помощью этого соотношения легко показать, верность выражения для связи координат на kом шаге:

из предположения индукции:

$$x_{k-1} = Sz_{k-1} + s$$

10/10

Тогда:

$$x_k = x_{k-1} - B_{k-1}\Delta f(x_{k-1}) = Sz_{k-1} + s - S\tilde{B_{k-1}} S^T (S^T)^{-1} \Delta\tilde{f}\left(z_{k-1}\right)$$
$$= S(z_{k-1} - \tilde{B_{k-1}}\Delta\tilde{f}\left(z_{k-1}\right) + s = Sz_k + s$$

Показали шаг индукции, следовательно доказал, что всегда будет выполняться связь: $x_k = Sz_k + s$

Newton convergence issue

Рассмотрите следующую функцию:

$$f(x, y) = \frac{x^4}{4} - x^2 + 2x + (y-1)^2$$

И точку старта $x_0 = (0, 2)^\top$. Как ведет себя метод Ньютона, запущенный с этой точки? Чем это можно объяснить?

Как ведет себя градиентный спуск с фикисрованным шагом $\alpha = 0.01$ и метод наискорейшего спуска в таких же условиях? (в этом задании не обязательно показывать численные симуляции)

==YOUR ANSWER==

идея была в том, чтоб формулу записать, нуок

Решил, что легче с численными симуляциями :)

```
import numpy as np
from matplotlib import pyplot as plt


def f(x, y):
    return x**4/4 - x**2 + 2*x + (y-1)**2


def df(x, y):
    return np.array([x**3-2*x+2, 2*(y-1)], dtype='float64')


def H(x,y):
    H = np.zeros((2,2), dtype='float64')
    H[0][0] = 3*x**2-2
    H[1][1] = 2
    return H


def Newton_method_coord(x0=np.array([0.0, 2.0]), N=10):
    x_n = np.zeros_like(x0)
    res = [x0]
    for i in range(N):
        x_n = x0 - np.linalg.inv(H(x0[0], x0[1])) @ df(x0[0], x0[1])
        res.append(x_n)
        x0 = x_n
    return res


res = Newton_method_coord()
```

Реализовал метод Ньютона. И запустил его на 10 итераий

```
res
```

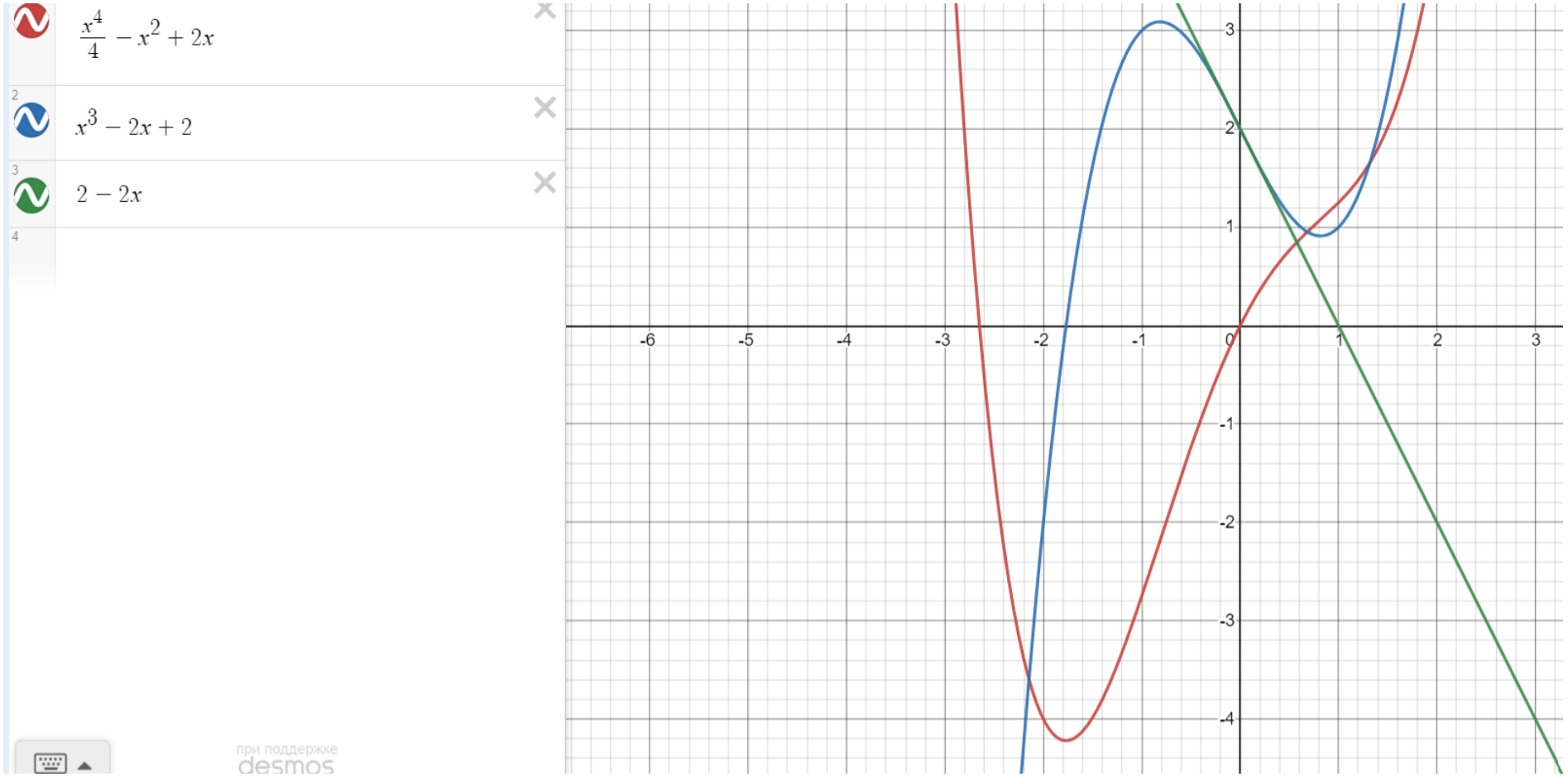
```
     [array([0., 2.]),
      array([1., 1.]),
      array([0., 1.]),
```

```
array([1., 1.]),
array([0., 1.]),
array([1., 1.]),
array([0., 1.]),
array([1., 1.]),
array([0., 1.]),
array([1., 1.]),
array([0., 1.])]
```

Видим, что он зацикливается и не сходится к минимуму функции!

Почему такое получается? У нас переменные не смешиваются (то есть минимизировать нашу функцию от двух переменных, также как минимизировать по отдельности от y и от x, так как нигде нет xy). Для функции по y метод Ньютона находит оптимальное значение за 1 шаг. Конечно, так и должно быть ведь у нас функция от y - парабола, значит её вторая апроксимация - парабола и мы шагаем в минимум.

С функцией от x всё тяжелее и точку старта мы выбираем явно неудачную. Построим на график функции и производной и сделаем шаг методом Ньютона из точки 0



По графику видим, что точка старта выбрана между двумя точками экстремума для производной функции и метод Ньютона делает шаг (аппроксимирует производную прямой) к другой точке экстремума и происходит зацикливание

Делаем вывод, что нужно подбирать точку старта хорошо, иначе не сойдёмся (для этого в вычматах говорили, что первым шагом нужно локализовывать, а затем уже применять этот метод). Чтобы подвердить это, запустим из x=-3

```
Newton_method_coord(x0=np.array([-3,2]), N=4)
```

```
    [array([-3,  2]),
```

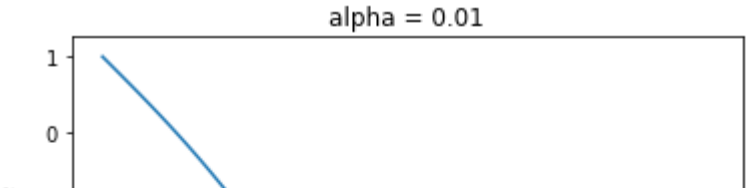
```
       array([-2.24,  1.  ]),
       array([-1.87537141,  1.        ]),
       array([-1.77655645,  1.        ]),
       array([-1.76932996,  1.        ])]
```

Сошлись к минимуму за 4 итерации с правильной выбранной точкой старта

Посмотрим как ведёт себя градиентный спуск

```
def grad_descent(x0=np.array([0.0, 2.0]), N=100):
    x_n = np.zeros_like(x0)
    res = [f(x0[0], x0[1])]
    alpha = 0.01
    for i in range(N):
        x_n = x0 - alpha * df(x0[0], x0[1])
        res.append(f(x_n[0], x_n[1]))
        x0 = x_n
    return res
```

```
N=100
plt.plot(np.arange(N+1), grad_descent(N=N));
plt.xlabel('номер итеррации')
plt.ylabel('function value')
plt.title(f'alpha = {0.01}');
```



Градиентный спуск достаточно медленно (так как шаг маленький) сходится к точке минимума

Метод наискорейшего спуска тоже сойдется к минимуму (так как точка минимуму 1 и функция выпуклая)

Comparison

Реализуйте на языке python:

- метод Ньютона
- метод SR-1

для минимизации следующих функций:

- Квадратичная форма $f(x) = \frac{1}{2}x^\top Ax + b^\top x, \quad x \in \mathbb{R}^n, A \in \mathbb{S}_+^{n\times n}$. Попробуйте n = 2, 50, 228
- Функция Розенброка $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$.

Сравните 2 реализованных Вами метода И метод `BFGS` из библиотеки `scipy`, а так же его модификацию L-BFGS в решении задачи минимизации описанных выше функций. точку старта необходимо инициализировать одинаковую для всех методов в рамках одного запуска. Необходимо провести не менее 10 запусков для каждого метода на каждой функции до достижения того критерия остановки, который вы выберете (например, расстояние до точки оптимума - во всех задачах мы её знаем)

В качестве результата нужно заполнить следующие таблички, заполнив в них усредненное по числу запусков количество итераций, необходимых для сходимости и времени работы:

Критерий остановки _____ норма между точным решением и полученным - менньше заданной точности tol

Число запусков _____ 20

P.S. если в силу каких то причин Вам не удалось сделать задание полностью, попробуйте сфокусироваться хотя бы на его части.

Квадратичная форма. n = 2	Iterations	Time
Newton		
SR-1		
BFGS		
L-BFGS		

Квадратичная форма. n = 50	Iterations	Time
Newton		
SR-1		
BFGS		
L-BFGS		

Квадратичная форма. n = 228	Iterations	Time
Newton		
SR-1		
BFGS		
L-BFGS		

Функция Розенброка	Iterations	Time
Newton		
SR-1		
BFGS		
L-BFGS		

Квадратичная форма

```
import time
from scipy.optimize import minimize


def gener_data(n):
```

```
    A = np.random.uniform(size=(n, n))
    A = A @ A.T  # чтобы была симметричной
    b = np.random.uniform(size=n)
    return A, b


N = 2


A_m, b_m = gener_data(n=N)


A_m, b_m
```

```
     (array([[0.69855219, 1.05345407],
             [1.05345407, 1.77716227]]), array([0.39813017, 0.82622251]))
```

```
def exact_solution(A, b):
    return -b @ np.linalg.inv(A + A.T)


x_exac = exact_solution(A_m, b_m)


def quadratic(x, A, b):
    return 1/2 * x.T @ A @ x + b.T @ x


def grad_quadratic(A, b, x):
    return b + (A + A.T) @ x  # A - симметричная, но все равно напишем в общем виде


def hes_quadratic(A, b, x):
    return A.T + A


def method_Newton(A, b, x0, tol=10**(-2)):
    res = []
    while np.linalg.norm(x0 - x_exac) > tol:
        x_new = x0 - np.linalg.inv(hes_quadratic(A, b, x0)) @ grad_quadratic(A, b, x0)
        res.append(quadratic(A, b, x_new))
```

```
        x0 = x_new
    return res


def SR1(A, b, x0, tol=10**(-4)):
    B0 = np.eye(A.shape[0])
    res = []
    while np.linalg.norm(x0 - x_exac) > tol:
        x_new = x0 - B0 @ grad_quadratic(A, b, x0)
        dx = x_new - x0
        dy = grad_quadratic(A, b, x_new) - grad_quadratic(A, b, x0)
        dop = (dx - B0 @ dy).reshape((-1, 1))
        B0 += dop @ dop.T / np.inner((dx - B0 @ dy), dy)
        x0 = x_new
        res.append(quadratic(A, b, x0))
    return res


final_res = {}
for N in [2, 50, 228]:

    A_m, b_m = gener_data(n=N)
    x_exac = exact_solution(A_m, b_m)

    result_experiment = {'Newton': {'time': [], 'iter': []}, 'SR1': {'time': [], 'iter': []},
                         'BFGS': {'time': [], 'iter': []}, 'LBFGS': {'time': [], 'iter': []}}
    for i in range(20):
        x0 = np.random.uniform(low=-1, high=2, size=N)

        t_start = time.time()
        res_Newton = method_Newton(A_m, b_m, x0)
        t_end = time.time()
        result_experiment['Newton']['time'].append(t_end - t_start)
        result_experiment['Newton']['iter'].append(len(res_Newton))

        t_start = time.time()
        res_SR1 = SR1(A_m, b_m, x0)
        t_end = time.time()
        result_experiment['SR1']['time'].append(t_end - t_start)
        result_experiment['SR1']['iter'].append(len(res_SR1))
```

```
        result_experiment['SR1']['iter'].append(len(res_SR1))

        t_start = time.time()
        res_BFGS = minimize(quadratic, x0, args=(A_m, b_m), method='BFGS', tol=10**(-2))
        t_end = time.time()

        result_experiment['BFGS']['time'].append(t_end - t_start)
        result_experiment['BFGS']['iter'].append(res_BFGS.nit)

        t_start = time.time()
        res_LBFGS = minimize(quadratic, x0, args=(A_m, b_m), method='L-BFGS-B', tol=10**(-2))
        t_end = time.time()

        result_experiment['LBFGS']['time'].append(t_end - t_start)
        result_experiment['LBFGS']['iter'].append(res_LBFGS.nit)

    result_experiment['Newton']['iter'] = np.mean(result_experiment['Newton']['iter'])
    result_experiment['SR1']['iter'] = np.mean(result_experiment['SR1']['iter'])
    result_experiment['BFGS']['iter'] = np.mean(result_experiment['BFGS']['iter'])
    result_experiment['LBFGS']['iter'] = np.mean(result_experiment['LBFGS']['iter'])

    result_experiment['Newton']['time'] = np.mean(result_experiment['Newton']['time'])
    result_experiment['SR1']['time'] = np.mean(result_experiment['SR1']['time'])
    result_experiment['BFGS']['time'] = np.mean(result_experiment['BFGS']['time'])
    result_experiment['LBFGS']['time'] = np.mean(result_experiment['LBFGS']['time'])

    final_res[N] = result_experiment


final_res
```

```
 {2: {'Newton': {'time': 0.00014977455139160157, 'iter': 1.0},
   'SR1': {'time': 0.0001496553421020508, 'iter': 3.0},
   'BFGS': {'time': 0.0006734251976013184, 'iter': 6.25},
   'LBFGS': {'time': 0.00039905309677124023, 'iter': 4.55}},
  50: {'Newton': {'time': 0.00020028352737426757, 'iter': 1.0},
   'SR1': {'time': 0.003965687751770019, 'iter': 52.0},
   'BFGS': {'time': 0.05697555541992187, 'iter': 62.7},
   'LBFGS': {'time': 0.014597010612487794, 'iter': 22.1}},
  228: {'Newton': {'time': 0.0022580862045288087, 'iter': 1.0},
   'SR1': {'time': 0.172900927066803, 'iter': 231.1},
```

```
        'BFGS': {'time': 4.798973619937897, 'iter': 532.05},
        'LBFGS': {'time': 0.18865784406661987, 'iter': 27.8}}}
```

Функция Розенброка

```
def Rosenbrok(x, y):
    return (1 - x)**2 + 100*(y - x**2)**2


def grad_Rosenbrok(x, y):
    return np.array([-2*(1 - x) - 400*(y - x**2)*x, 200 * (y - x**2)])


def hes_Rosenbrok(x, y):
    H = np.zeros((2,2), dtype='float64')
    H[0][0] = 800*x**2 + 2 - 400*(y - x**2)
    H[1][1] = 200
    H[0][1] = H[1][0] = -400*x
    return H


def method_Newton_Ros(x0, tol=10**(-2)):
    res = []
    while np.linalg.norm(x0 - x_exac) > tol:
        x_new = x0 - np.linalg.inv(hes_Rosenbrok(x0[0], x0[1])) @ grad_Rosenbrok(x0[0], x0[1])
        res.append(Rosenbrok(x_new[0], x_new[1]))
        x0 = x_new
    return res


def SR1_Ros(x0, tol=10**(-4)):
    B0 = np.eye(2)
    res = []
    while np.linalg.norm(x0 - x_exac) > tol:
        x_new = x0 - B0 @ grad_Rosenbrok(x0[0], x0[1])
        dx = x_new - x0
        dy = grad_Rosenbrok(x_new[0], x_new[1]) - grad_Rosenbrok(x0[0], x0[1])
```

```
        dy = grad_Rosenbrok(x_new[0], x_new[1]) - grad_Rosenbrok(x0[0], x0[1])
        dop = (dx - B0 @ dy).reshape((-1, 1))
        B0 += dop @ dop.T / np.inner((dx - B0 @ dy), dy)
        x0 = x_new
        res.append(Rosenbrok(x_new[0], x_new[1]))
    return res


result_experiment = {'Newton': {'time': [], 'iter': []}, 'SR1': {'time': [], 'iter': []},
                     'BFGS': {'time': [], 'iter': []}, 'LBFGS': {'time': [], 'iter': []}}
x_exac = np.array([1, 1])
for i in range(20):
    x0 = np.random.uniform(low=-1, high=2, size=2)

    t_start = time.time()
    res_Newton = method_Newton_Ros(x0)
    t_end = time.time()
    result_experiment['Newton']['time'].append(t_end - t_start)
    result_experiment['Newton']['iter'].append(len(res_Newton))

    t_start = time.time()
    res_SR1 = SR1_Ros(x0)
    t_end = time.time()
    result_experiment['SR1']['time'].append(t_end - t_start)
    result_experiment['SR1']['iter'].append(len(res_SR1))

    t_start = time.time()
    res_BFGS = minimize(Rosenbrok, x0[0], x0[1], method='BFGS', tol=10**(-2))
    t_end = time.time()

    result_experiment['BFGS']['time'].append(t_end - t_start)
    result_experiment['BFGS']['iter'].append(res_BFGS.nit)

    t_start = time.time()
    res_LBFGS = minimize(Rosenbrok, x0[0], x0[1], method='L-BFGS-B', tol=10**(-2))
    t_end = time.time()

    result_experiment['LBFGS']['time'].append(t_end - t_start)
    result_experiment['LBFGS']['iter'].append(res_LBFGS.nit)
```

```
result_experiment['Newton']['iter'] = np.mean(result_experiment['Newton']['iter'])
result_experiment['SR1']['iter'] = np.mean(result_experiment['SR1']['iter'])
result_experiment['BFGS']['iter'] = np.mean(result_experiment['BFGS']['iter'])
result_experiment['LBFGS']['iter'] = np.mean(result_experiment['LBFGS']['iter'])

result_experiment['Newton']['time'] = np.mean(result_experiment['Newton']['time'])
result_experiment['SR1']['time'] = np.mean(result_experiment['SR1']['time'])
result_experiment['BFGS']['time'] = np.mean(result_experiment['BFGS']['time'])
result_experiment['LBFGS']['time'] = np.mean(result_experiment['LBFGS']['time'])


result_experiment
```

```
{'Newton': {'time': 0.0001485586166381836, 'iter': 3.35},
 'SR1': {'time': 0.0047882080078125, 'iter': 146.55},
 'BFGS': {'time': 0.0008981347084045411, 'iter': 4.65},
 'LBFGS': {'time': 0.0003984928131103516, 'iter': 3.65}}
```

Занесём все результаты в таблицы:

Квадратичная форма. n = 2	Iterations	Time
Newton	1.0	0.00014977
SR-1	3.0	0.00014965
BFGS	6.25	0.00067342
L-BFGS	4.55	0.00039905

Квадратичная форма. n = 50	Iterations	Time
Newton	1.0	0.000200284
SR-1	52.0	0.003965687
BFGS	62.7	0.056975555
L-BFGS	22.1	0.014597011

Квадратичная форма. n = 228	Iterations	Time
Newton	1.0	0.002258086
SR-1	231.1	0.1729009
BFGS	532.05	4.79897361

Квадратичная форма. n = 228	Iterations	Time
L-BFGS	27.8	0.18865784

Функция Розенброка	Iterations	Time
Newton	3.35	0.000148558
SR-1	146.55	0.004788208
BFGS	4.65	0.000898134

Метод Ньютона, как и должен из теории, для квадратичной функции сходится за 1 итерацию и быстрее всех итерационных методов

↗↘ Conjugate gradients

Метод

$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
if $\mathbf{r}_0$ is sufficiently small, then return $\mathbf{x}_0$ as the result
$\mathbf{p}_0 := \mathbf{r}_0$
$k := 0$
repeat
$\alpha_k := \frac{\mathbf{r}_k^\mathsf{T} \mathbf{r}_k}{\mathbf{p}_k^\mathsf{T} \mathbf{A} \mathbf{p}_k}$
$\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$
$\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A} \mathbf{p}_k$
if $\mathbf{r}_{k+1}$ is sufficiently small, then exit loop
$\beta_k := \frac{\mathbf{r}_{k+1}^\mathsf{T} \mathbf{r}_{k+1}}{\mathbf{r}_k^\mathsf{T} \mathbf{r}_k}$
$\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$
$k := k + 1$
end repeat
return $\mathbf{x}_{k+1}$ as the result

В этом задании Вам предлагается рассмотреть как влияют предобуславливатели на время работы метода сопряженных градиентов.

Рассмотрим задачу наименьших квадратов:

$$f(x) = \frac{1}{2}||Ax - b||_2^2 = \frac{1}{2}\sum_{i=1}^{n}(a_i^T x - b_i)^2$$

где $A \in \mathbb{S}^n_{++}, b \in \mathbb{R}^n$.

Как мы знаем, эта задача выпукла и минимум находится из условия $\nabla f(x^*) = Ax^* - b = 0$. То есть для решения задачи необходимо разрешить систему уравнений $Ax = b$. Можно просто применить метод сопряженных градиентов, но если матрица плохо обусловлена ($\frac{\lambda_{max}}{\lambda_{min}} >> 1$), метод работает медленно (буквально, скорость сходимости CG прямо пропорциональна $\sqrt{\kappa(A)}$).

Preconditioning

Один из способов борьбы с этим - использование матриц-предобуславливателей разных видов и последующее решение другой задачи:

$$M^{-1}Ax = M^{-1}b$$

Здесь матрица **предобуславливателя** M подбирается таким образом, чтобы итоговая матрица $\tilde{A} = M^{-1}A$ имела меньшее число обусловленности. Существует несколько довольно простых, но зачастую сильно улучшающих работу метода предоубславливателей:

- M = diag($A_{11}, A_{22} \dots A_{nn}$) (Jacobi)
- M $\approx \hat{A}$, где например $\hat{A}$ - неполная факторизация Холецкого

Preconditioned Conjugate Gradients

Нет никаких проблем в том, чтобы решать новую систему $\tilde{A}x = \tilde{b}$ методов сопряженных градиентов. Однако, нативное встраивание предобуславливателя в алгоритм, делает использование этой идеи еще более эффективной. Для этого надо детально модифицировать классический CG. Кроме того, мы потребуем положительности новой матрицы $\tilde{A}$. Для этого будем использовать следующий вариант построения матрицы M:

$$M^{-1} = LL^{\top}$$

$$\begin{aligned} Ax = b &\leftrightarrow M^{-1}Ax = M^{-1}b \\ &\leftrightarrow L^{\top}Ax = L^{\top}b \\ &\leftrightarrow \underbrace{L^{\top}AL}_{\tilde{A}} \cdot \underbrace{L^{-1}x}_{\tilde{x}} = \underbrace{L^{\top}b}_{\tilde{b}} \end{aligned}$$

В новых переменных ($\tilde{A}, \tilde{x}, \tilde{b}$) невязка запишется, как:

$$\tilde{r}_k = \tilde{b} - \tilde{A}\tilde{x}_k = L^{\top}b - (L^{\top}AL)(L^{-1}x_k) = L^{\top}b - L^{\top}Ax_k = L^{\top}r_k$$

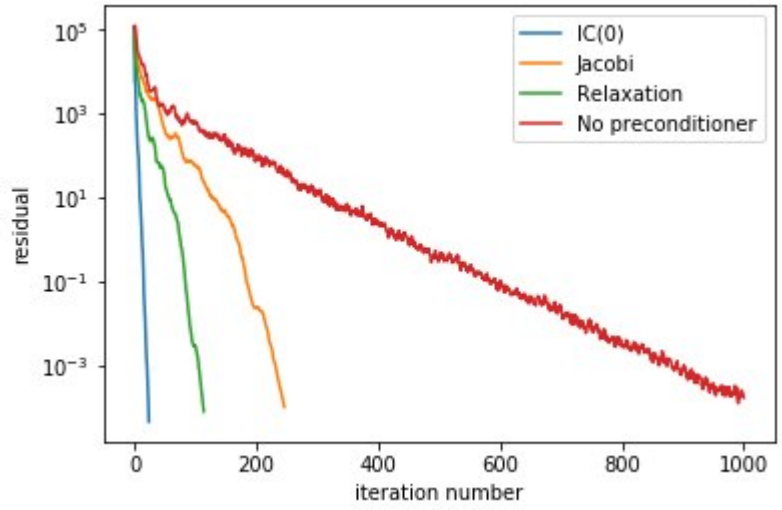
Факторизация Холецкого s.p.d. матрицы A - ее разложение на произведение нижнетреугольной и верхнетреугольной матрицы: $A = L^T L$ wiki. Есть несколько упрощений этого алгоритма, позволяющих получить матрицу, "похожую" на A. Мы будем использовать следующую: $if \quad (a_{i,j} = 0) \rightarrow l_{i,j} = 0$, а далее по алгоритму.

Задание Выбрать 1 задачу отсюда, исследовать как влияет на скорость сходимости тот или иной предоубславливатель:

1) Сравнить число итераций, за которое метод сходится с точностью 10^{-7} для двух предобуславливателей и для обычного метода сопряженных градиентов.

2) Построить графики зависимости нормы невязки $||r_k|| = ||Ax_k - b||$ от номера итерации для двух предобуславливателей и для обычного метода сопряженных градиентов.

Пример:



Реализуем метод сопряжённых градиентов

```
MAX_ITER = 150


def con_grad(x, A, b):
    rks = []
    r = b - np.dot(A, x)
    p = r
    r_k_norm = np.dot(r.T, r)

    for _ in range(MAX_ITER):
```

```
        rks.append(r_k_norm)#/np.linalg.norm(b))
        Ap = np.dot(A, p)
        alpha = r_k_norm / np.dot(p.T, Ap)
        x += alpha * p
        r -= alpha * Ap
        r_kplus1_norm = np.dot(r.T, r)
        beta = r_kplus1_norm / r_k_norm
        r_k_norm = r_kplus1_norm
        if r_kplus1_norm < 1e-7:
            print('Itr:', _)
            break
        p = r + beta * p
    return x, rks
```

Сгенерим матрицу 2 на 2 и вектор b и с помощью реализованного метода решим её, чтобы убедиться, что метод работат верно (подсчитаем точное решение и сравним с ним)

```
N = 2
A = np.random.random([N, N])*5
A = A @ A.T # сделаем A симметричной
b = np.random.random(N)*5
x0 = np.random.random(N)
```

```
x = con_grad(x0, A, b)[0]
x
```

```
Itr: 2
array([-0.65600541,  0.64818336])
```

```
x_exac = np.linalg.inv(A) @ b
x_exac
```

```
array([-0.65603336,  0.64820825])
```

Решение совпадают, значит, метод сопряжённых градиентов работает верно и можно делать предобуславливатели

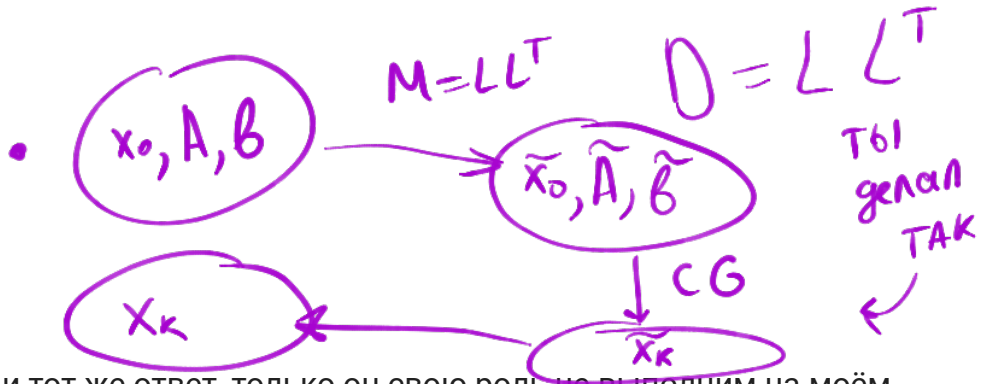
строго говоря, нет. Но отличная идея

- взять N побольше
- x_0 тоже надо бы изменить

$\tilde{x_0} = L^{-1} x_0$

```
# Диогональный предобуславливатель
x0 = np.random.random(N)

d = np.diag(1/np.diag(A))
A_diag = d @ A @ d
b_diag = d @ b
ans = con_grad(x0, A_diag, b_diag)
rs = ans[1]
x = d @ ans[0]
x
```

```
     Itr: 4
     array([-0.65439013,  0.64694923])
```



Видим, что с диагональным предобуславливателем получили тот же ответ, только он свою роль не выполним на моём игрушечном примере (количество итераций увеличилось, а не уменьшилось, хотя для обычной матрицы их было 2, быстрее как известно нельзя)

Но можно лучше (в ноутбуке) есть ссылка на .pdf

```
# Разложение Холецкого
L = np.linalg.cholesky(A)
A_hol = L.T @ A @ L
b_hol = L.T @ b
x0 = np.random.random(N)*10
x_hol = con_grad(x0, A_hol, b_hol)[0]
x_res = L @ x_hol
x_res
```

```
     Itr: 2
     array([-0.65603338,  0.64820828])
```

Также получили верный ответ! (тут количество итеараций осталось минимальным - 2)

Не понял, можно ли использовать готовую реализацию разложения Холецкого, поэтому решил реализовать алгоритм

```
# ещё раз генерим матрицу
A = np.random.random([N, N])
A = A @ A.T # сделаем A симметричной
```

можно, конечно!

```
def ichol(A):
    n = A.shape[0]
    L = np.zeros((n, n))
    for i in range(n):
        for j in range(i+1):
            res = 0
            for l in range(j):
                res += L[i][l] * L[j][l]

            if i == j:
                if A[i][j] == 0:
                    L[i][j] = 0
                L[i][j] = np.sqrt(A[i][i] - res)

            else:
                if A[i][j] == 0:
                    L[i][j] = 0
                L[i][j] = 1.0 / L[j][j] * (A[i][j] - res)
    return L
```

супер!

```
np.linalg.cholesky(A)
```

```
array([[0.2271735 , 0.        ],
       [0.3066954 , 0.11232951]])
```

```
ichol(A)
```

```
array([[0.2271735 , 0.        ],
       [0.3066954 , 0.11232951]])
```

Написанное разложение работает! Теперь запустим на большой задаче

```
import scipy
from scipy import io


A = scipy.io.mmread("1138_bus.mtx").toarray()


A.shape
```

```
     (1138, 1138)
```

```
N = A.shape[0]


b = np.random.random(N)
x0 = np.random.random(N)


MAX_ITER = 100000


res_con_grad = con_grad(x0, A, b)
res_con_grad[0]
```

```
     Itr: 3651
     array([  0.39657035,  79.54408351, 125.50385824, ..., 145.4213698 ,
            144.04103   , 145.56991054])
```

```
# Диогональный предобуславливатель
N = A.shape[0]
x0 = np.random.random(N)

d = np.diag(1/np.diag(A))
A_diag = d @ A @ d
b_diag = d @ b
```

```
ans = con_grad(x0, A_diag, b_diag)
rs = ans[1]
x = d @ ans[0]
x
```

```
Itr: 95207
array([  0.3965699 ,  79.54343235, 125.50250154, ..., 145.42071239,
       144.038127  , 145.5663504 ])
```

```
# Разложение Холецкого
L = np.linalg.cholesky(A)
A_hol = L.T @ A @ L
b_hol = L.T @ b
x0 = np.random.random(N)*10
x_hol = con_grad(x0, A_hol, b_hol)
x_res = L @ x_hol[0]
x_res
```

```
array([ 0.00146197,  0.27061231,  0.23906075, ..., -0.03952301,
       -0.20583641, -0.24296715])
```

```
x_exac = np.linalg.inv(A) @ b
x_exac
```

```
array([  0.39657039,  79.54409551, 125.50387541, ..., 145.42138999,
       144.04105237, 145.56993329])
```

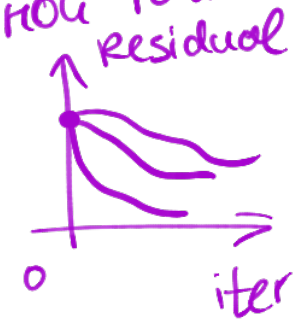
Получаем очень странный результат: предобуславливатель мешает методу сходиться, а не помогает ему...

Почему так получается? Непонятно(

Разные методы необходимо сравнивать в полностью одинаковых условиях

x_0, и т.д.

т.е. графики должны выходить из одной точки.



6 8/10

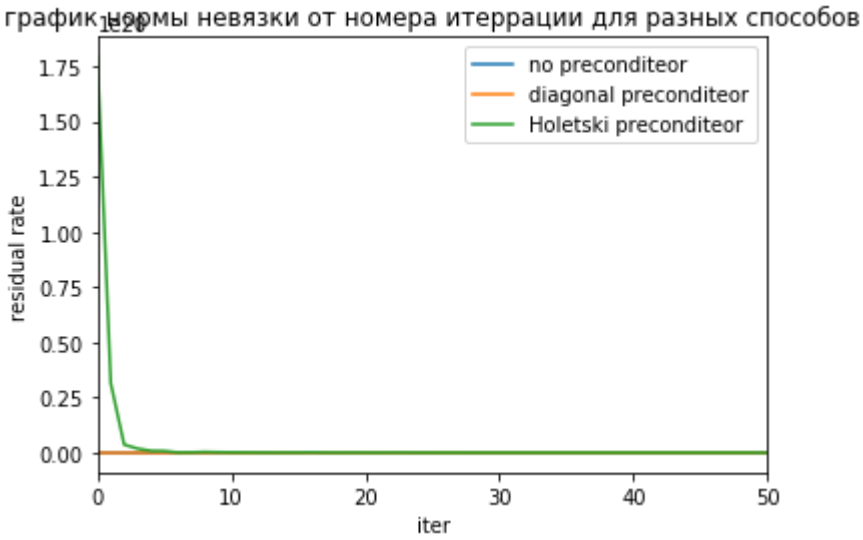
97.1/100

8/10

Возможно, абзац про то, что нужно менять метод, а не просто переходить к решению новой системы не просто так (хотя не понял, в чём разница). Но и новую систему решить должно быть проще (по итерациям). Возможно, ошибка в реализации, но ответы сходятся

```
plt.plot(range(len(res_con_grad[1])), res_con_grad[1], label="no preconditeor");
plt.plot(range(len(rs)), rs, label="diagonal preconditeor");
plt.plot(range(len(x_hol[1])), x_hol[1], label="Holetski preconditeor");
plt.xlabel('iter')
plt.ylabel('residual rate')
plt.title("график нормы невязки от номера итеррации для разных способов");
plt.xlim(0, 50)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x263acfa9160>
```



📱🎧💻 LP. Covers manufacturing

Random Corp is producing covers for following products:

- 📱 phones
- 🎧 headphones
- 💻 laptops

The company's production facilities are such that if we devote the entire production to headphones covers, we can produce 5000 of them in one day. If we devote the entire production to phone covers or laptop covers, we can produce 4000 or 2000 of them in one day.

The production schedule is one week (6 working days), and the week's production must be stored before distribution. Storing 1000 headphones covers (packaging included) takes up 30 cubic feet of space. Storing 1000 phone covers (packaging included) takes up 50 cubic feet of space, and storing 1000 laptop covers (packaging included) takes up 220 cubic feet of space. The total storage space available is 6000 cubic feet.

Due to commercial agreements with Random Corp has to deliver at least 4500 headphones covers and 3000 laptop covers per week in order to strengthen the product's diffusion.

The marketing department estimates that the weekly demand for headphones covers, phone, and laptop covers does not exceed 9000 and 14000, and 7000 units, therefore the company does not want to produce more than these amounts for headphones, phone, and laptop covers.

Finally, the net profit per each headphones cover, phone cover, and laptop cover is $5, $7, and $12, respectively.

The aim is to determine a weekly production schedule that maximizes the total net profit.

==YOUR ANSWER==

Find the solution to the problem using PuLP

```
!pip install pulp --quiet
! sudo apt-get install glpk-utils --quiet  # GLPK
! sudo apt-get install coinor-cbc --quiet  # CoinOR
```

```
     "sudo" Ґ пў«пҐвбп ўгваҐҐ© Ё«Ё ўҐиҐ©
     Є®¬ ¤®©, Ёбї®«пҐ¬® Їа®Ја ¬¬®© Ё«Ё Ї ЄҐвл¬ д ©«®¬.
     "sudo" Ґ пў«пҐвбп ўгваҐҐ© Ё«Ё ўҐиҐ©
     Є®¬ ¤®©, Ёбї®«пҐ¬® Їа®Ја ¬¬®© Ё«Ё Ї ЄҐвл¬ д ©«®¬.
```

```
import pulp
```

```
# name ofproducts
buf_products = ["LAPTOPS", "TELEPHONES", "HEADPHONES"]


# costs of products
dict_costs = {"LAPTOPS": 12, "TELEPHONES": 7, "HEADPHONES": 5}


# performance
performance = {"LAPTOPS": 1.0/2000, "TELEPHONES": 1.0/4000, "HEADPHONES": 1.0/5000}


# min amount of product in week
min_num_prod = {"LAPTOPS": 3000, "TELEPHONES": 0, "HEADPHONES": 4500}

# max amount of product in week
max_num_prod = {"LAPTOPS": 7000, "TELEPHONES": 14000, "HEADPHONES": 9000}


# value of products
value = {"LAPTOPS": 220.0/1000, "TELEPHONES": 50.0/1000, "HEADPHONES": 30.0/1000}

TOTAL_VALUE = 6000.0


# create LP object,
# set up as a maximization problem --> since we want to maximize sum cost of product
prob = pulp.LpProblem('goods_problem', pulp.LpMaximize)


num_products = pulp.LpVariable.dicts("num_products", buf_products)

# ограничения на максимальное и минимальное произведёное кол-во товаров
for pr in buf_products:
    num_products[pr].lowBound = min_num_prod[pr]
    num_products[pr].upBound = max_num_prod[pr]


# ограничения на обьём всех занимаемых товаров
```

```
prob += pulp.lpSum([value[pr] * num_products[pr] for pr in buf_products]) <= TOTAL_VALUE

# ограничение на то, что в день можем производить не больше определённого кол-ва товаров
prob += pulp.lpSum([performance[pr] * num_products[pr] for pr in buf_products]) <= 6.0

# функция для максимизации
prob += pulp.lpSum([dict_costs[pr] * num_products[pr] for pr in buf_products])

prob.solve()
```

```
1
```

```
pulp.LpStatus[prob.status]
```

```
'Optimal'
```

```
for el in prob.variables():
    print(el.name, '=', el.varValue)
```

```
num_products_HEADPHONES = 5000.0
num_products_LAPTOPS = 3000.0
num_products_TELEPHONES = 14000.0
```

Получили решение нашей задачи линейного программирования, на котором получается максимальная прибыль за неделю

🚧 Idea of barrier methods

Идея барьерных методов довольно проста: давайте заменим задачу условной оптимизации на последовательность модифицированных задач безусловной оптимизации, которая сходится к исходной. Рассмотрим простой пример такой задачи.

$$\min_{x \in \mathbb{R}^n} c^\top x$$
$$Ax \leq b$$

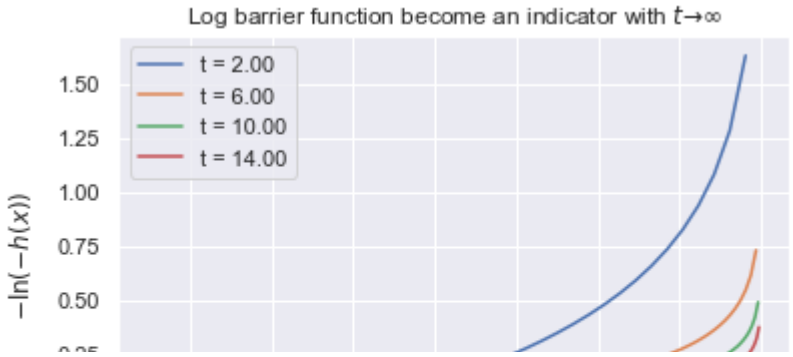
Вместо этого предлагается рассмотреть задачу безусловной оптимизации:

$$\min_{x \in \mathbb{R}^n} c^\top x + \frac{1}{t} \sum_{i=1}^{m} \left[-\ln\left(-(a_i^\top x - b_i)\right) \right]$$

По сути, все ограничения типа неравенств инкорпорируются в целевую функцию как **барьерные функции** с помощью (например) логарифмического барьера $-\ln -h(x)$ при $t \to \infty$ (см. картинку ниже)

```
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
sns.set()

t = np.linspace(2,14, 4)

for t_ in t:
    x = np.linspace(-1.5,0, 20*t_)
    y = -np.log(-x)
    plt.plot(x,y/t_, label = f't = {t_:.2f}')

plt.title(f'Log barrier function become an indicator with $t \\to \infty$')
plt.ylabel(f'$-\ln(-h(x))$')
plt.xlabel(f'$x$')
plt.legend()
plt.show()
```

```
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:9: DeprecationWarning: object of type <class 'numpy.f
  if __name__ == '__main__':
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:10: RuntimeWarning: divide by zero encountered in log
  # Remove the CWD from sys.path while we load stuff.
```



Таким образом, формируется последовательности вспомогательных задач, в которых барьерная функция практически ничего не добавляет к исходной функции, но начинает заметно штрафовать по мере приближения к границе бюджетного множества.

Алгоритм можно сформулировать следующим образом:

- $t = t_0 > 0, x_0 \in S$ - корректная инициализация

Повторять для $\mu > 1$ и $k = 1, 2, 3, \dots$

- Шаг оптимизационного алгоритма для функции $t_k f(x_k) + \varphi(x_k) \to x_{k+1}$
- если $\frac{m}{t} \leq \varepsilon$ - конец алгоритма
- Увеличить t: $t_{k+1} = \mu t_k$

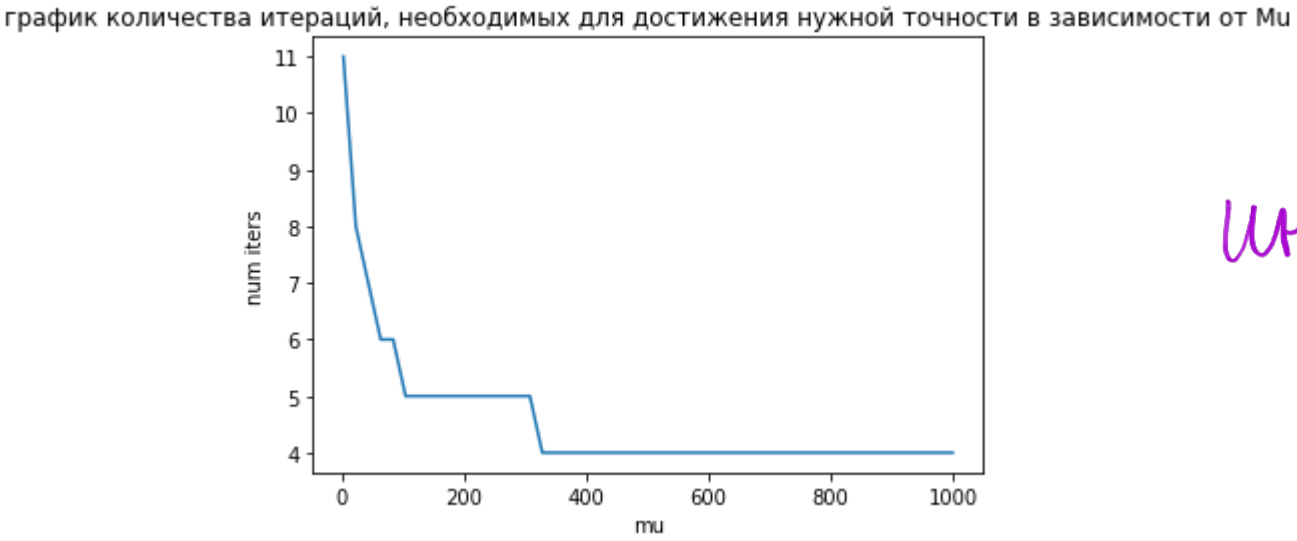
Постройте график количества итераций, необходимых для достижения $\varepsilon = 10^{-8}$ точности для $n = 50, m = 100$ в зависимости от значения параметра μ.

Рассмотрите при этом метод Ньютона для целевой функции с фиксированным шагом или демпфированный метод Ньютона. (По ссылке доступен код на матлабе для решения задачи, можно в него смотреть и понять, как делать эту задачу.)

Рассмотрите значения μ в интервале $[2, 1000]$ с помощью функции `mus = np.linspace(2,1000)`

```
def calc_grad_hes(A, b, c, t, x):
    m, n = A.shape
    r = A @ x - b
    R = np.diag(1.0/r)

    func = t*c.T @ x - np.log(-r.T) @ np.ones(m)

    grad = t*c - A.T @ R @ np.ones(m)

    hes = A.T @ R @ R @ A

    return func, grad, hes


m, n = 100, 50
A = np.random.uniform(size=(m, n))
b = 1+ np.random.uniform(size=m)
c = np.random.uniform(size=n)


buf_iters = []

for mu in np.linspace(2, 1000):
    t = 1
    x = np.random.uniform(size=n)
    num_iter = 0

    while m / t >= 1e-8:
        func, grad, hes = calc_grad_hes(A, b, c, t, x)

        x_new = x - np.linalg.inv(hes) @ grad

        if np.linalg.norm(x_new - x) <= 1e-8:
            break

        t *= mu
        num_iter += 1
        x = x_new
    buf_iters.append(num_iter)
```

```
plt.plot(np.linspace(2, 1000), buf_iters);
plt.xlabel('mu')
plt.ylabel('num iters')
plt.title("график количества итераций, необходимых для достижения нужной точности в зависимости от Mu");
```



интересно

После того, как посмотрел код на матлабе полностью и внимательно решил повторить их эксперимент с демпфированный метод Ньютона и понять, как будет отличаться от обычного метода Ньютона

```
BETA = 0.7
ALPHA = 0.1


buf_iters = []

for mu in np.linspace(2, 1000):
    t = 1
    x = np.zeros(n)
    num_iter = 0

    while m / t >= 1e-8:
```

```
        func, grad, hes = calc_grad_hes(A, b, c, t, x)

        v =  np.linalg.inv(hes) @ grad

        newton_decrement = grad.T @ v
        s = 1
        while min(b - A @ (x + s*v)) < 0:
            s = BETA*s

        while calc_grad_hes(A, b, c, t, x+s*v)[0] > func + ALPHA*s*newton_decrement:
            s = BETA*s

        x += s * v

        if abs(newton_decrement) <= 1e-8:
            break

        t *= mu
        num_iter += 1
    buf_iters.append(num_iter)


plt.plot(np.linspace(2, 1000), buf_iters);
plt.xlabel('mu')
plt.ylabel('num iters')
plt.title("график количества итераций, необходимых для достижения нужной точности в зависимости от Mu");
```

график количества итераций, необходимых для достижения нужной точности в зависимости от Mu