Homework 2. {Prokhorov}

- Домашку присылать в виде `.pdf` файла на адрес `homework@merkulov.top`.
- Дедлайн: **8 мая 23:59**.
- Есть несколько способов конвертировать `.ipynb` в `.pdf`. Самый простой - сохранить ноутбук как `.html`, а затем распечатать это в `.pdf` файл, нажав `ctrl + P` в браузере.
- Займитесь этим вопросом заранее, чтобы в последний момент не получить из за этого 0 баллов.
- Все ячейки должны быть запущены, а графики построены.

Problem 1. Обо всем по чуть чуть. Деревня Хоббитцов.

Источник (https://courses.cs.ut.ee/MTAT.03.227/2015_spring/uploads/Main/home-exercises-5.pdf)

In [2]:

```
%matplotlib inline
import numpy as np
from matplotlib import pyplot as plt
import matplotlib as mpl
import seaborn as sns
import scipy
sns.set()
```

Для Вашего удобства ниже написана функция, которая рисует небольшую одномерную деревню.

In [98]:

```
def plot_village(coordinates, l=1):
    # Checking, that all the coordinates are less than l
    assert (coordinates <= l).all(), 'All the houses should be in a village'

    # Draw horizontal line
    plt.hlines(0, 0, l)
    plt.xlim(0, l)
    plt.ylim(-0.5, 0.5)

    # Draw house points
    y = np.zeros(np.shape(coordinates))
    plt.title('The Hobbit Village')
    plt.plot(coordinates,y,'o',ms = 10)
    plt.axis('off')
    plt.xlabel('Coordinates')
    fig = plt.gcf()
    fig.set_size_inches(15, 1)
    plt.show()
```

In [99]:

```
N = 25
l = 1
x = np.random.rand(N)*l

plot_village(x, l)
```

Жители одномерной деревни хотят подключиться к интернету, для этого им необходимо поставить центральную обслуживающую станцию, от которой кабель будет тянуться ко всем домам деревни. Пусть цена кабеля, который надо тянуть от станции до каждого дома независимо, определяется некоторой функцией $p(d)$. Тогда ясно, что деревне придется заплатить следующую сумму за доступ в мировую паутину:

$$P(w, x) = \sum_{i=1}^{N} p(d_i) = \sum_{i=1}^{N} p(|w - x_i|)$$

Здесь w - координата станции, x_i - координата i-ого домика.

🤔

Найдите аналитически оптимальную позицию w^*, доставляющую минимум функции $P(w, x)$ при условии, что $p(d) = d^2$

$$P(w) = \sum_{i=1}^{n} (w - x_i)^2 \longrightarrow \min_w$$

$$P'(w) = 2\left(nw - \sum_{i=1}^{n} x_i\right) = 0 \qquad \Longrightarrow \qquad w^* = \frac{1}{n}\sum_{i=1}^{n} x_i$$

🤔

Напишите функцию `P`, которая берет на вход позицию станции w и вектор координат x и возвращает значение функции потерь P.

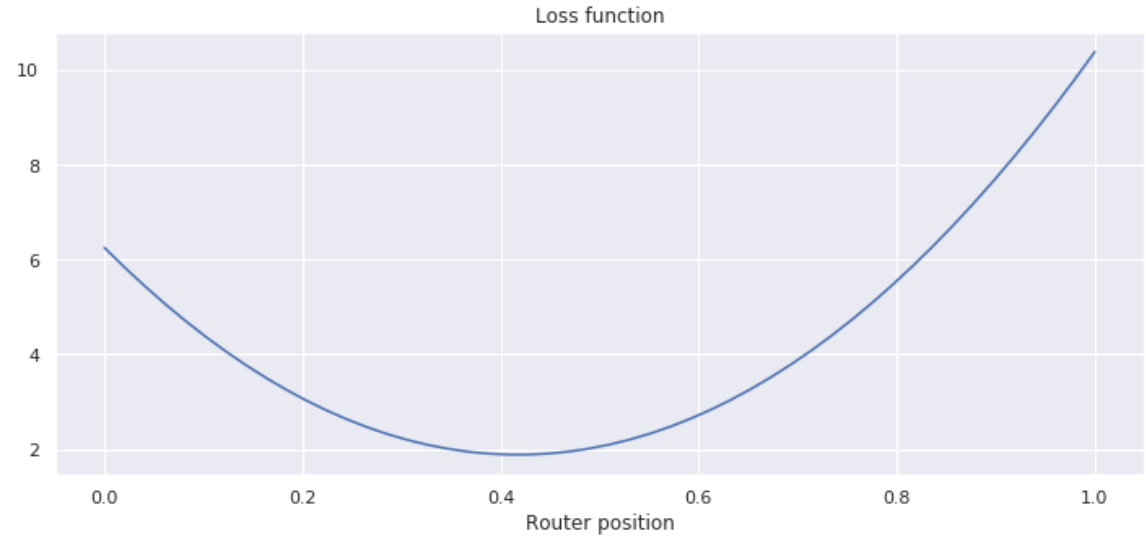
In [100]:

```
def P(w, x):
    return np.sum((x - w) * (x - w))
```

Постройте значение функции потерь для значения координат от 0 до l.

In [101]:

```
w = np.linspace(0,l)
p = [P(w_i, x) for w_i in w]
plt.title('Loss function')
plt.xlabel('Router position')
plt.plot(w,p)
fig = plt.gcf()
fig.set_size_inches(12, 5)
```



🤔

Напишите функцию `dP`, которая берет на вход позицию станции w и вектор координат x и возвращает значение градиента функции потерь ∇P как функции от w.

In [102]:

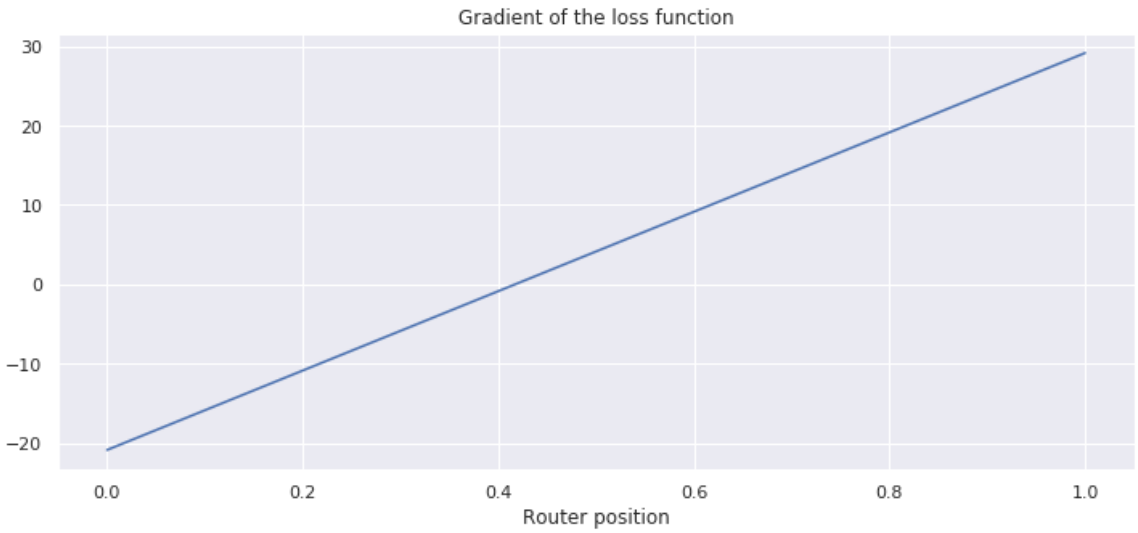
```
def dP(w, x):
    return 2*np.sum(w - x)
```

Постройте значение функции `dP` для значения координат от 0 до l. Какая точка на графике представляет особый интерес? Почему?

Нас интересует точка, в которой производная равна нулю, так как для дифференцируемой функции (а у нас такая) это является необходимым условием минимума. Так как у нас функция выпуклая, то это и достаточное условие.

In [104]:

```
w = np.linspace(0,1)
dp = [dP(w_i, x) for w_i in w]
plt.title('Gradient of the loss function')
plt.xlabel('Router position')
plt.plot(w,dp)
fig = plt.gcf()
fig.set_size_inches(12, 5)
```



🤔

Напишите функцию `gradient_descent`, которая возвращает значение w_k через фиксированное число шагов. Длина функции не должна превышать 5 строчек кода.

$$w_{k+1} = w_k - \mu \nabla P(w_k)$$

In [105]:

```
def gradient_descent(P, dP, w0, mu, Nsteps):
    w = w0
    for i in range(Nsteps):
        w = w - mu*dP(w, x)
    return w
```

🤔

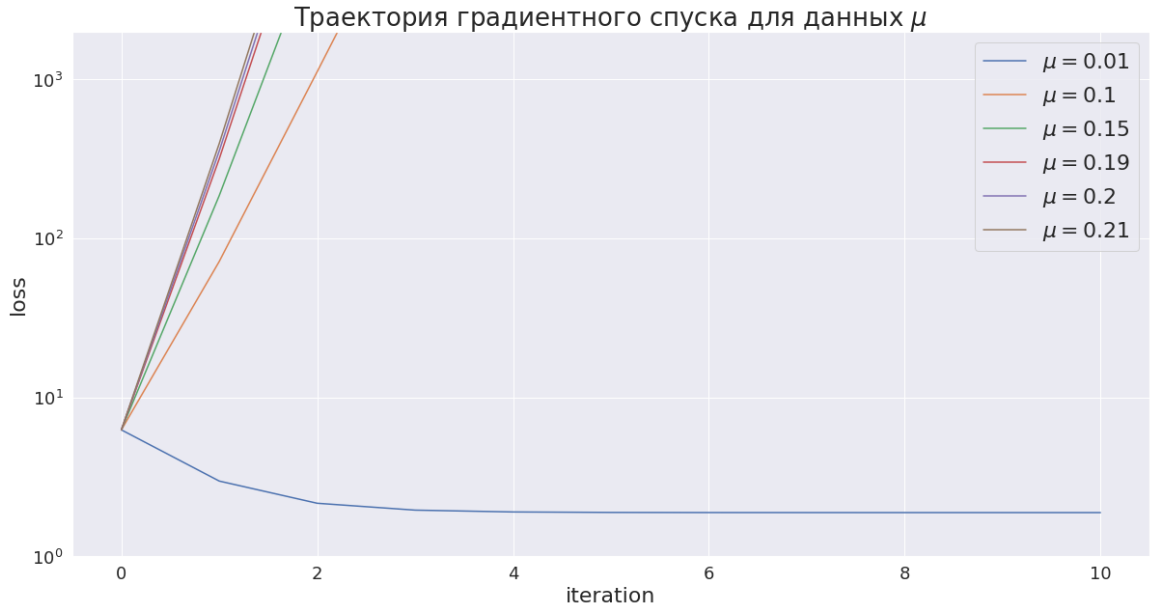
Модифицируйте функцию `gradient_descent` так, чтобы она возвращала всю траекторию оптимизации (все значения w_k). Постройте графики $P(w_k, x)$ для $\mu = 0.01, 0.1, 0.15, 0.19, 0.20, 0.21$. Сделайте выводы.

In [106]:

```
def gradient_descent(P, dP, w0, mu, Nsteps):
    tr = [w0]
    for i in range(Nsteps):
        tr.append(tr[-1] - mu*dP(tr[-1], x))
    return tr
```

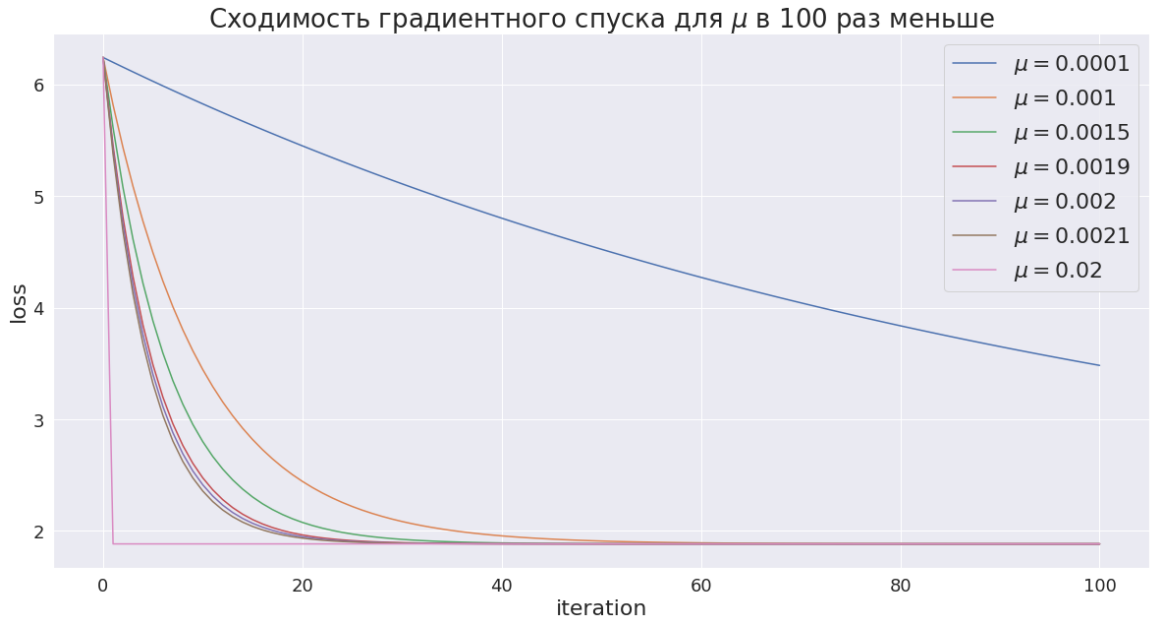
In [107]:

```
mus = [0.01, 0.1, 0.15, 0.19, 0.20, 0.21]
w0 = 0 # initial guess
Nsteps = 10

plt.figure(figsize=(20,10))
plt.title('Траектория градиентного спуска для данных $\mu$', fontsize=26)
plt.xlabel('iteration', fontsize=22)
plt.ylabel('loss', fontsize=22)
plt.ylim(1, 2000)
plt.yscale('log')
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
for mu in mus:
    ws = gradient_descent(P, dP, w0, mu, Nsteps)
    Ps = [P(w, x) for w in ws]
    plt.plot(range(len(ws)), Ps, label=('$\mu = $' + str(mu)))
plt.legend(fontsize=22)
plt.show()
```



In [109]:

```
mus = [0.01, 0.1, 0.15, 0.19, 0.20, 0.21, 2]
w0 = 0 # initial guess
Nsteps = 100

plt.figure(figsize=(20,10))
plt.title('Сходимость градиентного спуска для $\mu$ в 100 раз меньше', fontsize=26)
plt.xlabel('iteration', fontsize=22)
plt.ylabel('loss', fontsize=22)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
for mu in mus:
    ws = gradient_descent(P, dP, w0, mu*0.01, Nsteps)
    Ps = [P(w, x) for w in ws]
    #print(Ps)
    plt.plot(range(len(ws)), Ps, label=('$\mu = $' + str(mu*0.01)))
plt.legend(fontsize=22)
plt.show()
```



Вывод:

Градиентый спуск сошелся только при $\mu = 0.01$. Остальные μ были слишком большими. При них градиентный спуск шагал в нужную сторону, но слишком сильно перелетал минимум у оказывался дальше, чем был до этого. Чтобы получить более-менее показательные графики, можно уменьшить все μ в 100 раз.

Оптимальное значение μ:

$$\mu = \frac{1}{L} = \frac{1}{\|\nabla^2 P\|} = 0.02$$

В нашем случае гессиан постоянен: $\nabla^2 P \equiv 50$.

Отметим, что при $\mu = 0.02$ градиентный спуск совпадает с методом Ньютона (это следствие того, что задача квадратичная и одномерная).

🤔

Напишите функцию `ddP`, которая берет на вход позицию станции w и вектор координат x и возвращает значение гессиана функции потерь $\nabla^2 P$ как функции от w.

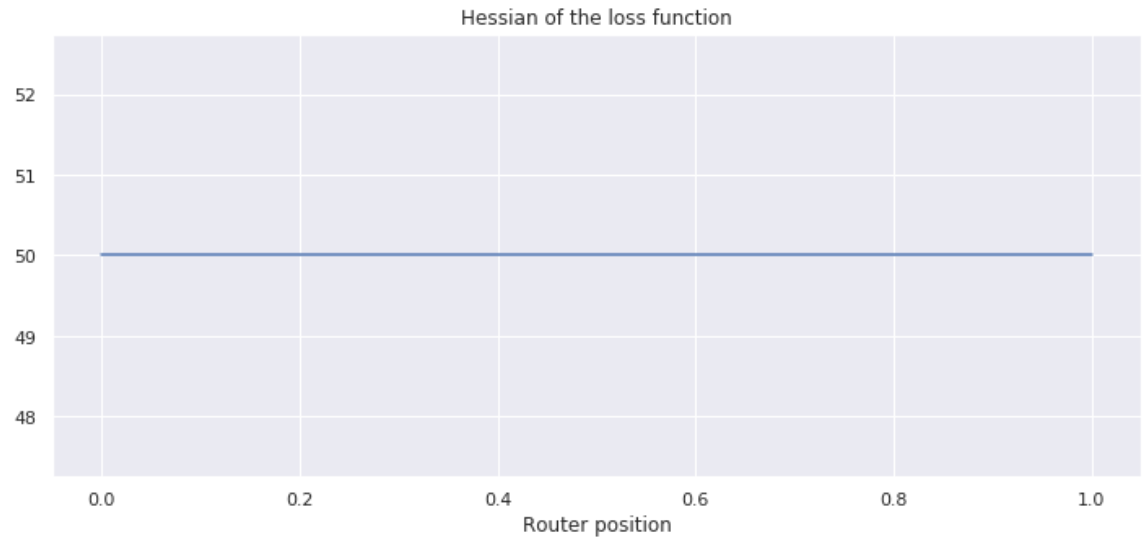
In [110]:

```
def ddP(w, x):
    return 2*x.shape[0]
```

Постройте значение функции `ddP` для значения координат от 0 до l.

In [111]:

```
w = np.linspace(0,1)
ddp = [ddP(w_i, x) for w_i in w]
plt.title('Hessian of the loss function')
plt.xlabel('Router position')
plt.plot(w,ddp)
fig = plt.gcf()
fig.set_size_inches(12, 5)
```



🤔

Напишите функцию `newton_descent`, которая возвращает всю оптимизационную траекторию (w_k) через фиксированное число шагов.

In [112]:

```
def newton_descent(P, dP, ddP, w0, Nsteps):
    tr = [w0]
    for i in range(Nsteps):
        tr.append(tr[-1] - dP(tr[-1], x) / ddP(tr[-1], x))
    return tr
```

Сравните траекторию метода с траекторией градиентного спуска. Исследуйте поведение метода Ньютона в зависимости от разной стартовой точки w_0.

In [113]:

```
mus = [0.013, 0.015, 0.02, 0.03]
w0 = 0 # initial guess
Nsteps = 5

plt.figure(figsize=(20,10))
plt.title('траектории градиетного спуска и метода Ньютона', fontsize=26)
plt.xlabel('iteration', fontsize=22)
plt.ylabel('loss', fontsize=22)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
plt.yscale('log')
for mu in mus:
    ws = gradient_descent(P, dP, w0, mu, Nsteps)
    Ps = [P(w, x) for w in ws]
    #print(ws)
    plt.plot(range(len(ws)), Ps, label=('$\mu = $' + str(mu)))

ws = newton_descent(P, dP, ddP, w0, Nsteps)
Ps = [P(w, x) for w in ws]
plt.plot(range(len(ws)), Ps, label='newton')

plt.legend(fontsize=22)
plt.show()
```

Вывод:

Метод Ньютона совпадает с градиентным спуском при оптимальном $\mu = \frac{1}{L} = 0.02$.

Метод Ньютона сходится всегда за 1 итерацию для любой начальной точки. Это связано с тем, что наша задача квадратичная. Метод Ньютона на каждом шаге строит аппроксимацию целевой функции параболой (по Тейлору) и идет в ее минимум, а так как наша функция и так является параболой, то мы сразу идем в ее минимум.

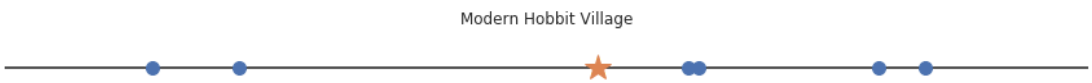
Давайте нарисуем оптимальное положение роутера в деревне хоббитцов:

In [114]:

```
def plot_village_with_internet(coordinates, router_coordinates, l=1):
    # Checking, that all the coordinates are less than l
    assert (np.array(coordinates) <= l).all(), 'All the houses should be in the
 village'
    assert (np.array(router_coordinates) <= l).all(), 'Router should be in the v
illage'

    # Draw horizontal line
    plt.hlines(0, 0, l)
    plt.xlim(0, l)
    plt.ylim(-0.5, 0.5)

    # Draw house points
    y = np.zeros(np.shape(coordinates))
    plt.title('Modern Hobbit Village')
    plt.plot(coordinates,y,'o',ms = 10)

    # Draw routers
    y = np.zeros(np.shape(router_coordinates))
    plt.plot(router_coordinates,y,'*',ms = 20)
    plt.axis('off')
    plt.xlabel('Coordinates')
    fig = plt.gcf()
    fig.set_size_inches(15, 1)
    plt.show()
```

In [119]:

```
N = 6
l = 1
x = np.random.rand(N)*l

w_ = newton_descent(P, dP, ddP, 0, 2)[-1]

plot_village_with_internet(x, w_, l)
```



🤔

Напишите функцию `stochastic_gradient_descent`, которая возвращает всю оптимизационную траекторию (w_k) через фиксированное число шагов по методу стохастического градиентного спуска (градиент считается не по всем домикам деревни хоббитцов, а по случайному подмножеству)

In [139]:

```
# Stochastic Gradient
def dP_sigma(w, x, p=0.5):
    random_mask = np.random.binomial(1, p, x.shape)
    return 2*np.sum((w - x)*random_mask)

def stochastic_gradient_descent(P, dP_sigma, w0, mu, Nsteps):
    tr = [w0]
    for i in range(Nsteps):
        tr.append(tr[-1] - mu*dP_sigma(tr[-1], x))
    return tr
```

Увеличьте число домиков до 50, сравните траектории алгоритмов градиентного и стохастического градиентного спуска. Сравните их.

Hint: придется изменить размер шага, чтобы работало:)

In [140]:

```
N = 50
l = 1
x = np.random.rand(N)*l

mus = [0.01, 0.019]
w0 = 0 # initial guess
Nsteps = 20

plt.figure(figsize=(20,10))
plt.title('траектории градиетного спуска и SGD', fontsize=26)
plt.xlabel('iteration', fontsize=22)
plt.ylabel('loss', fontsize=22)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
plt.yscale('log')
for mu in mus:
    ws = gradient_descent(P, dP, w0, mu, Nsteps)
    Ps = [P(w, x) for w in ws]
    #print(ws)
    plt.plot(range(len(ws)), Ps, label=('GD: $\mu = $' + str(mu)))

    ws = stochastic_gradient_descent(P, dP_sigma, w0, mu, Nsteps)
    Ps = [P(w, x) for w in ws]
    plt.plot(range(len(ws)), Ps, label=('SGD: $\mu = $' + str(mu)))

plt.legend(fontsize=22)
plt.show()
```

Вывод

Оптимальный шаг для стохастического градиетного спуска в $\frac{1}{p}$ (p - аргумент в `dP_sigma`) раз больше, чем оптимальный шаг в обычном градиентном спуске. Это связано с тем, что в среднем у нас число слагаемых в целевой функции равно pN, значит, стохастический гессиан равен $2pN$. Поэтому оптимальный шаг равен $\frac{1}{2pN} = \frac{1}{Lp} = \frac{\mu_{\text{opt}}}{p}$.

Также стоит отметить, что стохастический градиентный спуск, придя в минимум, может на последующих итерациях выпругнуть из него.

Только-только жизнь обитателей одномерной деревни наладилась, как роскомнадзор (казалось бы, какое ему дело) сказал, что цена на кабель должна считаться совсем по другой формуле:

$$p(d) = |d|$$

🤔

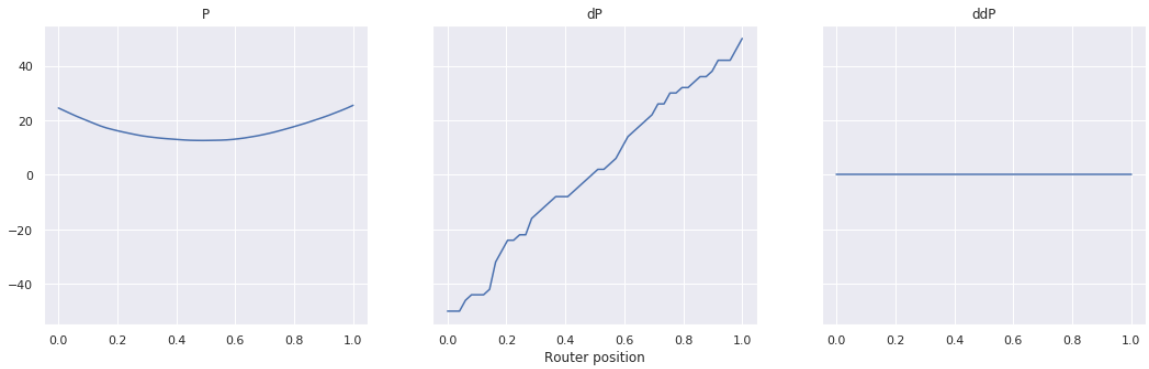
Напишите функции `P`, `dP`, `ddP` в новых реалиях. Постройте их для заданного x при различных w.

In [156]:

```
N = 50
l = 1
x = np.random.rand(N)*l

def P(w, x):
    return np.sum(np.abs(w - x))

def dP(w, x):
    return np.sum(np.sign(w - x))

# Гессиан недифференцируемой функции?

def ddP(w, x):
    return 0*w
```

In [149]:

```
w = np.linspace(0,1)

p = [P(w_i, x) for w_i in w]
dp = [dP(w_i, x) for w_i in w]
ddp = [ddP(w_i, x) for w_i in w]

f, (ax1, ax2, ax3) = plt.subplots(1, 3, sharey=True)
ax1.plot(w,p)
ax1.set_title('P')
ax2.plot(w,dp)
ax2.set_title('dP')
ax3.plot(w,ddp)
ax3.set_title('ddP')
ax2.set_xlabel('Router position')
f.set_size_inches(18, 5)
#f.suptitle('Fuck RKN')
plt.show()
```



Найдите оптимальное значение w^* в новых реалиях.

Целевая функция выпукла как сумма выпуклых функций:

$$P(w) = \sum_{i=1}^{N} |w - x_i|$$

По теореме Ферма, w - глобальный минимум тогда и только тогда, когда $0 \in \partial P(w)$ (субдифференциал).

По теореме Моро-Рокофеллара,

$$\partial P(w) = \sum_{i=1}^{N} \partial |w - x_i|$$

Легко найти, что

$$\partial |w - x_i| = \begin{cases} 1 & , \ w > x_i \\ [-1, 1] & , \ w = x_i \\ -1 & , \ w < x_i \end{cases}$$

Тогда получаем, что

$$\partial P(w) = R - L + \sum_{i=1}^{C} [-1, 1] = \Big[R - L - C, \ R - L + C\Big],$$

где $R = R(w)$ - число домов строго правее станции, $L = L(w)$ - число домов строго левее станции, $C = C(w)$ - число домов, совпавших со станцией.

Тогда $0 \in \partial P(w) \iff$

$$R - L - C \le 0 \le R - L + C$$

В частности, если все дома стоят в разных местах, то w оптимально тогда и только тогда, когда $R = L$.

🤔

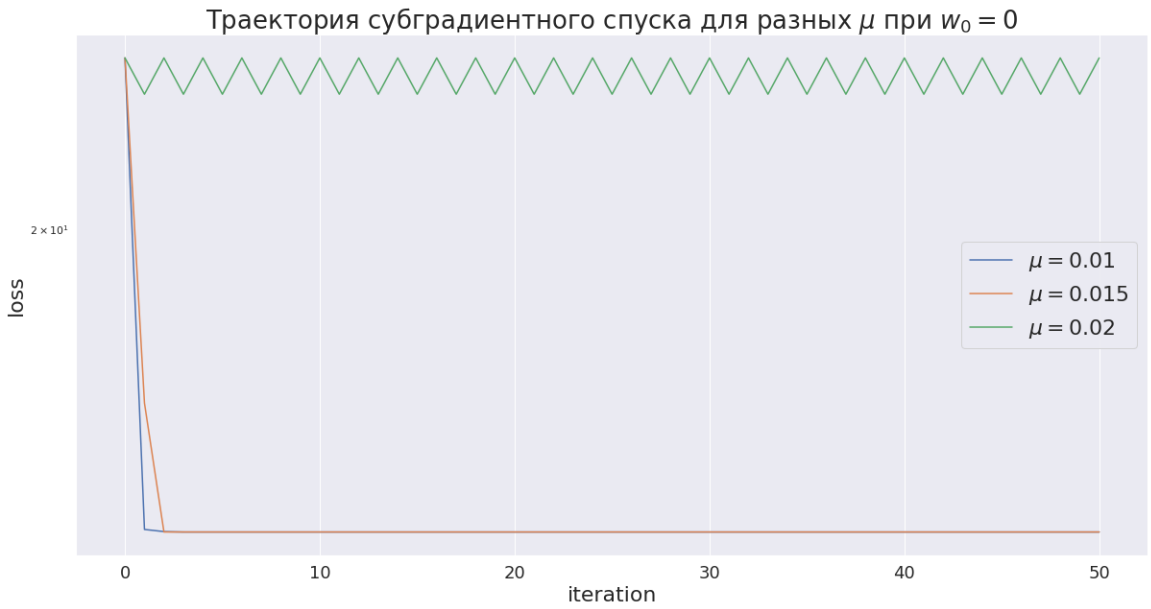
Постройте траекторию градиентного спуска в новых реалиях. Что Вы скажете роскомнадзору, когда он потребует решать эту задачу методом Ньютона (потому что они где то слышали, что надо использовать лучшие мировые практики)?

In [167]:

```
mus = [0.01, 0.015, 0.02]
w0 = -3.3 # initial guess
Nsteps = 50

plt.figure(figsize=(20,10))
plt.title('Траектория субградиентного спуска для разных $\mu$ при $w_0 = -3.3$', fontsize
=26)
plt.xlabel('iteration', fontsize=22)
plt.ylabel('loss', fontsize=22)
plt.yscale('log')
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
for mu in mus:
    ws = gradient_descent(P, dP, w0, mu, Nsteps)
    #print(ws)
    Ps = [P(w, x) for w in ws]
    plt.plot(range(len(ws)), Ps, label=('$\mu = $' + str(mu)))
plt.legend(fontsize=22)
plt.show()
```

In [169]:

```
mus = [0.01, 0.015, 0.02]
w0 = 0 # initial guess
Nsteps = 50

plt.figure(figsize=(20,10))
plt.title('Траектория субградиентного спуска для разных $\mu$ при $w_0 = 0$', fontsize=26
)
plt.xlabel('iteration', fontsize=22)
plt.ylabel('loss', fontsize=22)
plt.yscale('log')
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
for mu in mus:
    ws = gradient_descent(P, dP, w0, mu, Nsteps)
    #print(ws)
    Ps = [P(w, x) for w in ws]
    plt.plot(range(len(ws)), Ps, label=('$\mu = $' + str(mu)))
plt.legend(fontsize=22)
plt.show()
```



Вывод

Выше представлены графики субградиентного спуска для начальной точки $w_0 = -3.3$ и $w_0 = 0$. Так как далеко от деревни градиент везде постоянный, то оптимальнее идти с большим шагом, поэтому на ранних этапах чем больше шаг, тем лучше. Внутри деревни (в интервале от 0 до 1), градиент меняется, поэтому при $\mu = 0.02$ субградиентный спуск зациклился (он прыгает вокруг решения, и шаг слишком большой, чтобы спуситься к решению).

Метод Ньютона неприменим, потому что гессиан равен нулевой матрице почти всюду (т.е. в точках дифференцируемости функции).

🤔

Реализуйте функции `dP_sigma`, `stochastic_gradient_descent` в новых реалиях.

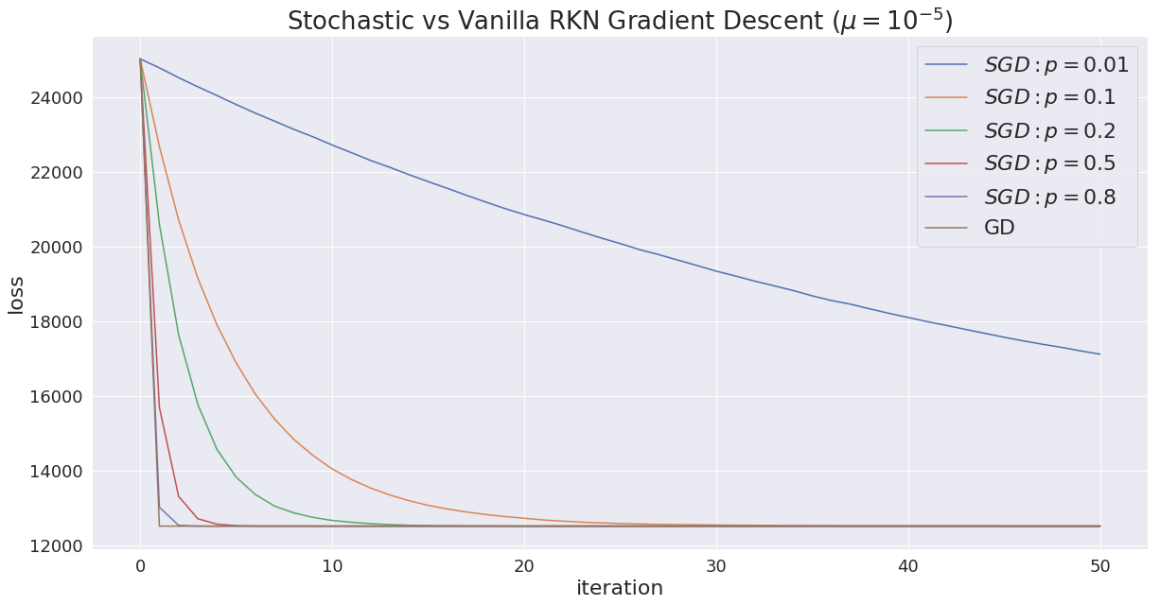
In [191]:

```
def dP_sigma(w, x, p=0.1):
    random_mask = np.random.binomial(1, p, x.shape)
    return np.sum(np.sign(w - x)*random_mask)

def stochastic_gradient_descent(P, dP_sigma, w0, mu, Nsteps, p=0.1):
    tr = [w0]
    for i in range(Nsteps):
        tr.append(tr[-1] - mu*dP_sigma(tr[-1], x, p))
    return tr
```

Постройте траектории градиентного и стохастического градиентного спуска. Поэксперементируйте с уровнем шума в стох. градиенте путем изменения доли домиков, по которым считается градиент (p). Проведите эксперименты для большого числа домиков (от 10000) и сравните результаты.

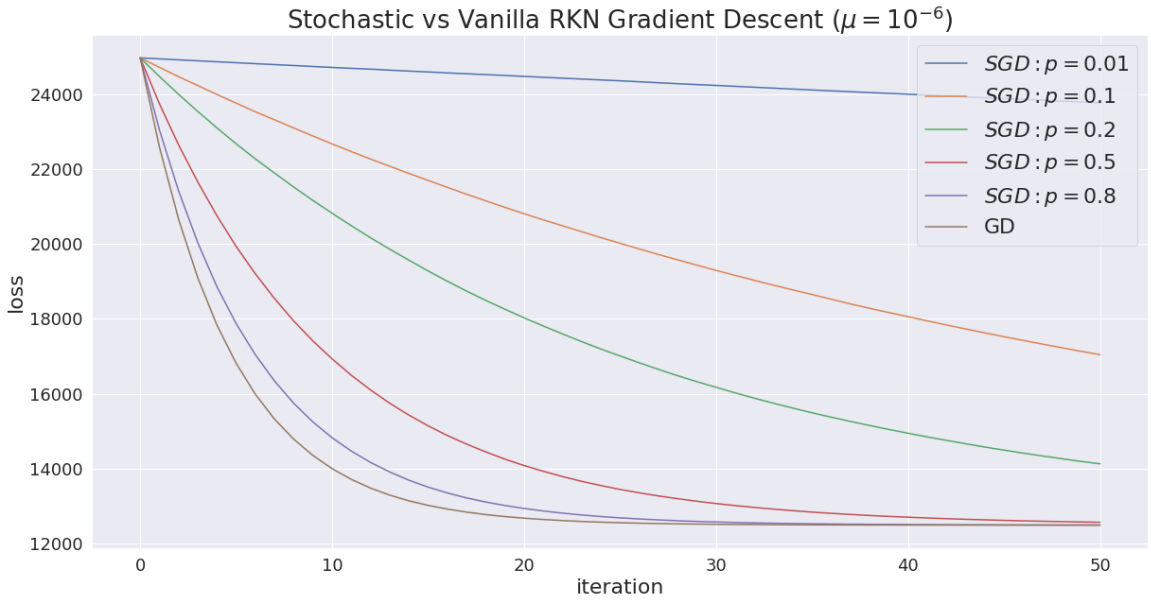
In [195]:

```
N = 50000
l = 1
x = np.random.rand(N)*l
mu = 1e-5
ps = [0.01, 0.1, 0.2, 0.5, 0.8]

plt.figure(figsize=(20,10))
plt.title('Stochastic vs Vanilla RKN Gradient Descent ($\mu = 10^{-5}$)', fontsi
ze=26)
plt.xlabel('iteration', fontsize=22)
plt.ylabel('loss', fontsize=22)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
for p in ps:
    ws = stochastic_gradient_descent(P, dP_sigma, 0, mu, Nsteps, p)
    #print(ws)
    Ps = [P(w, x) for w in ws]
    plt.plot(range(len(ws)), Ps, label=('$SGD: p = $' + str(p)))

ws = gradient_descent(P, dP, 0, mu, Nsteps)
Ps = [P(w, x) for w in ws]
plt.plot(range(len(ws)), Ps, label='GD')

plt.legend(fontsize=22)
plt.show()
```



In [198]:

```
N = 50000
l = 1
x = np.random.rand(N)*l
mu = 1e-6
ps = [0.01, 0.1, 0.2, 0.5, 0.8]

plt.figure(figsize=(20,10))
plt.title('Stochastic vs Vanilla RKN Gradient Descent ($\mu = 10^{-6}$)', fontsi
ze=26)
plt.xlabel('iteration', fontsize=22)
plt.ylabel('loss', fontsize=22)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
for p in ps:
    ws = stochastic_gradient_descent(P, dP_sigma, 0, mu, Nsteps, p)
    #print(ws)
    Ps = [P(w, x) for w in ws]
    plt.plot(range(len(ws)), Ps, label=('$SGD: p = $' + str(p)))

ws = gradient_descent(P, dP, 0, mu, Nsteps)
Ps = [P(w, x) for w in ws]
plt.plot(range(len(ws)), Ps, label='GD')

plt.legend(fontsize=22)
plt.show()
```



Вывод:

Так как у нас очень много домиков, то оптимальное решение скорее всего близко к 0.5. Начальная точка у нас $w_0 = 0$, и так как все домики находятся справа от нее, то в этой точке градиент $\nabla P(0) = -N$. Чтобы как можно ближе оказаться к решению после первого шага:

$$w_1 = w_0 - \mu \cdot (-N) \approx \frac{1}{2}$$

нужно взять шаг $\mu_{\text{opt}} = \dfrac{1}{2N} = 10^{-5}$.

Также стоит отметить, что при фиксированном μ стохастический градиентный спуск сходится тем быстрее, чем больше параметр p. Кроме того, оптимальный шаг для стохастического градиентного спуска равен

$$\mu = \frac{\mu_{\text{opt}}}{p}$$

по тем же причинам, что и ранее.

Чем больше p, тем ближе траектория стохастического градиентного спуска к обычному градиентному спуску.

Дела шли своим чередом в деревне хоббитцов. Однако, как и всякое процветающее общество (коим без всяких сомнений себя считали хоббитцы), они решили, что их количество увеличилось достаточно для того, чтобы поставить второй датацетр (роутер) и проводить интернет к каждому дому от ближайшего к нему роутера.

$$P(w_1, w_2, x) = \sum_{i=1}^{N} p(d_i) = \sum_{i=1}^{N} p\left(\min\left(|w_1 - x_i|, |w_2 - x_i|\right)\right)$$

🤔

Напишите функции `P`, `dP` в новых реалиях. Постройте их для заданного x при различных w_1, w_2. Помните, что градиент в этом случае представляет собой двумерный вектор.

Считаем, что $p(d) = |d|$.

Градиент функции $\min(x, y)$ определим следующим логичным образом:

$$\nabla \min(x, y) = \begin{cases} (1, 0)^T & , \ x < y \\ (1/2, 1/2)^T & , \ x = y \\ (0, 1)^T & , \ x > y \end{cases}$$

In [302]:

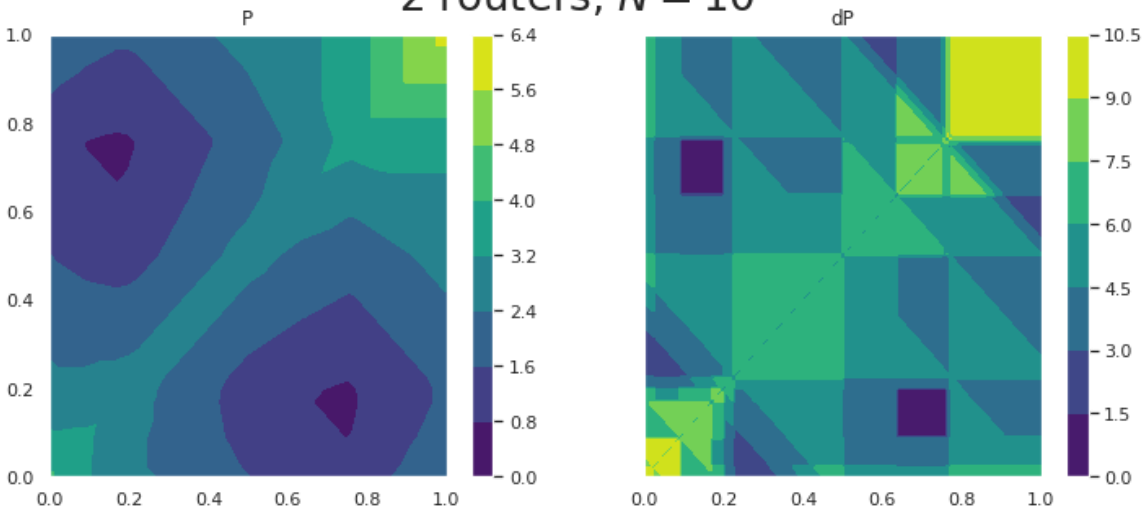
```
def P(w1, w2, x):
    return np.sum(np.amin(np.vstack((np.abs(w1 - x), np.abs(w2 - x))), 0))

def dP(w1, w2, x):
    a1 = np.abs(w1 - x)
    a2 = np.abs(w2 - x)
    g1 = (np.sign(a2 - a1 + 1e-5) + 1) / 2 # partial derivatives w.r.t. a1
    g2 = 1 - g1 # partial derivatives w.r.t. a2
    g1 = g1 * np.sign(w1 - x)
    g2 = g2 * np.sign(w2 - x)
    return np.array([np.sum(g1), np.sum(g2)])

def dP_sigma(w1, w2, x, p=0.5):
    random_mask = np.random.binomial(1, p, x.shape)
    a1 = np.abs(w1 - x)
    a2 = np.abs(w2 - x)
    g1 = (np.sign(a2 - a1 + 1e-5) + 1) / 2 # partial derivatives w.r.t. a1
    g2 = 1 - g1 # partial derivatives w.r.t. a2
    g1 = g1 * np.sign(w1 - x) * random_mask
    g2 = g2 * np.sign(w2 - x) * random_mask
    return np.array([np.sum(g1), np.sum(g2)])
```

Постройте графики $P(w_1, w_2), \nabla P(w_1, w_2)$ для различных значений N. Прокомментируйте, что происходит по мере увеличения N.

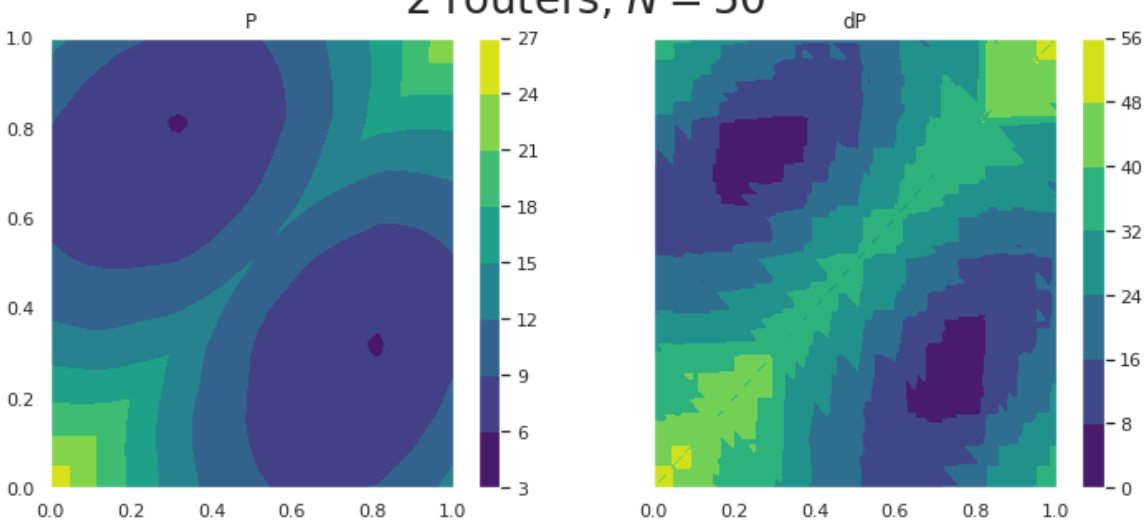
In [244]:

```
np.random.seed(10)

Ns = [10, 50, 500, 5000]
l = 1

for N in Ns:
    x = np.random.rand(N)*l

    w1 = np.linspace(0,l,200)
    w2 = np.linspace(0,l,200)



    p = np.zeros([w1.shape[0], w1.shape[0]])
    dp = np.zeros([w1.shape[0], w1.shape[0]])

    i = 0
    for w1_ in w1:
        j = 0
        for w2_ in w2:
            p[i][j]  =  P(w1_, w2_, x)
            dp[i][j] = np.linalg.norm(dP(w1_, w2_, x))
            j += 1
        i += 1

    f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
    c1 = ax1.contourf(w1, w2, p, cmap="viridis")
    plt.colorbar(c1, ax = ax1)
    ax1.set_title('P')
    c2 = ax2.contourf(w1, w2, dp, cmap="viridis")
    plt.colorbar(c2, ax = ax2)
    ax2.set_title('dP')

    f.set_size_inches(12, 5)
    f.suptitle('2 routers, $N = $'+str(N), fontsize=26)
    plt.show()
```

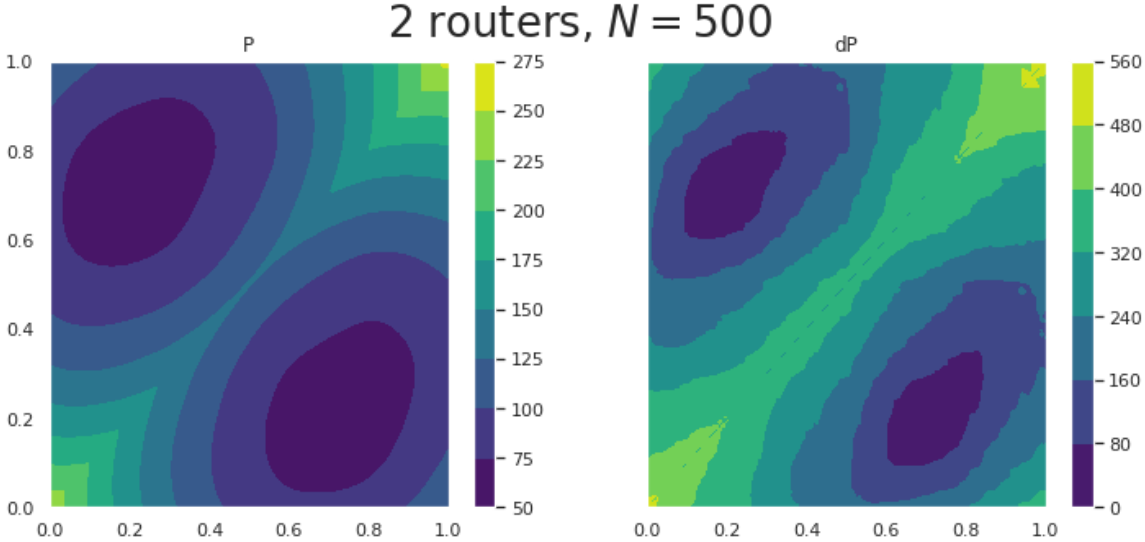
2 routers, $N = 10$

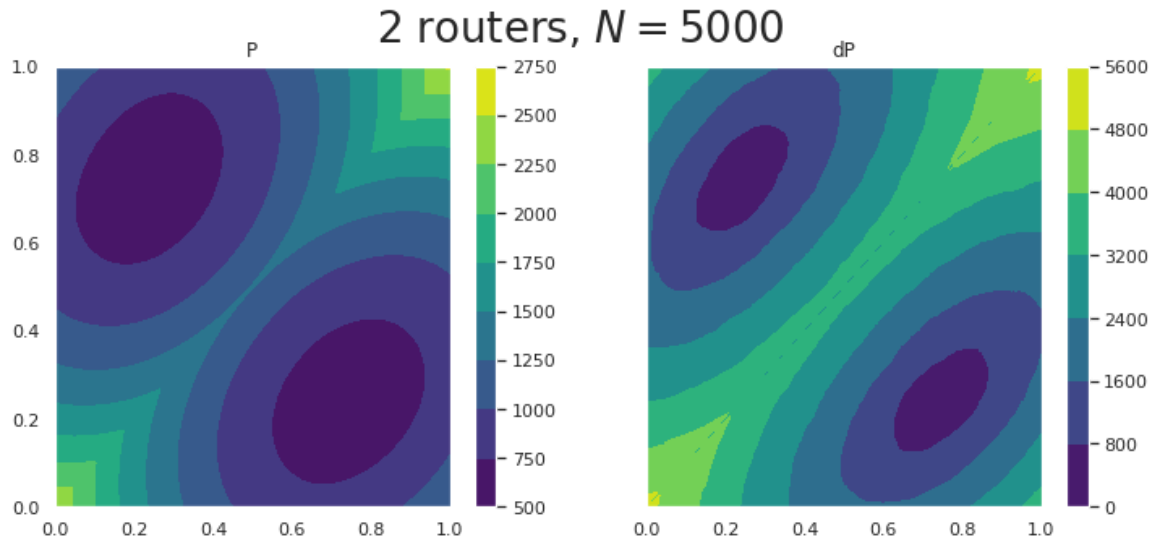


2 routers, $N = 50$



2 routers, $N = 500$





Видно, что с увеличением N линии уровня становятся более гладкими, и все больше напоминают форму эллипса, т.е. задача принимает некоторую непрерывную форму.

In [362]:

```
N = 500
l = 1
x = np.random.rand(N)*l
```

🤔

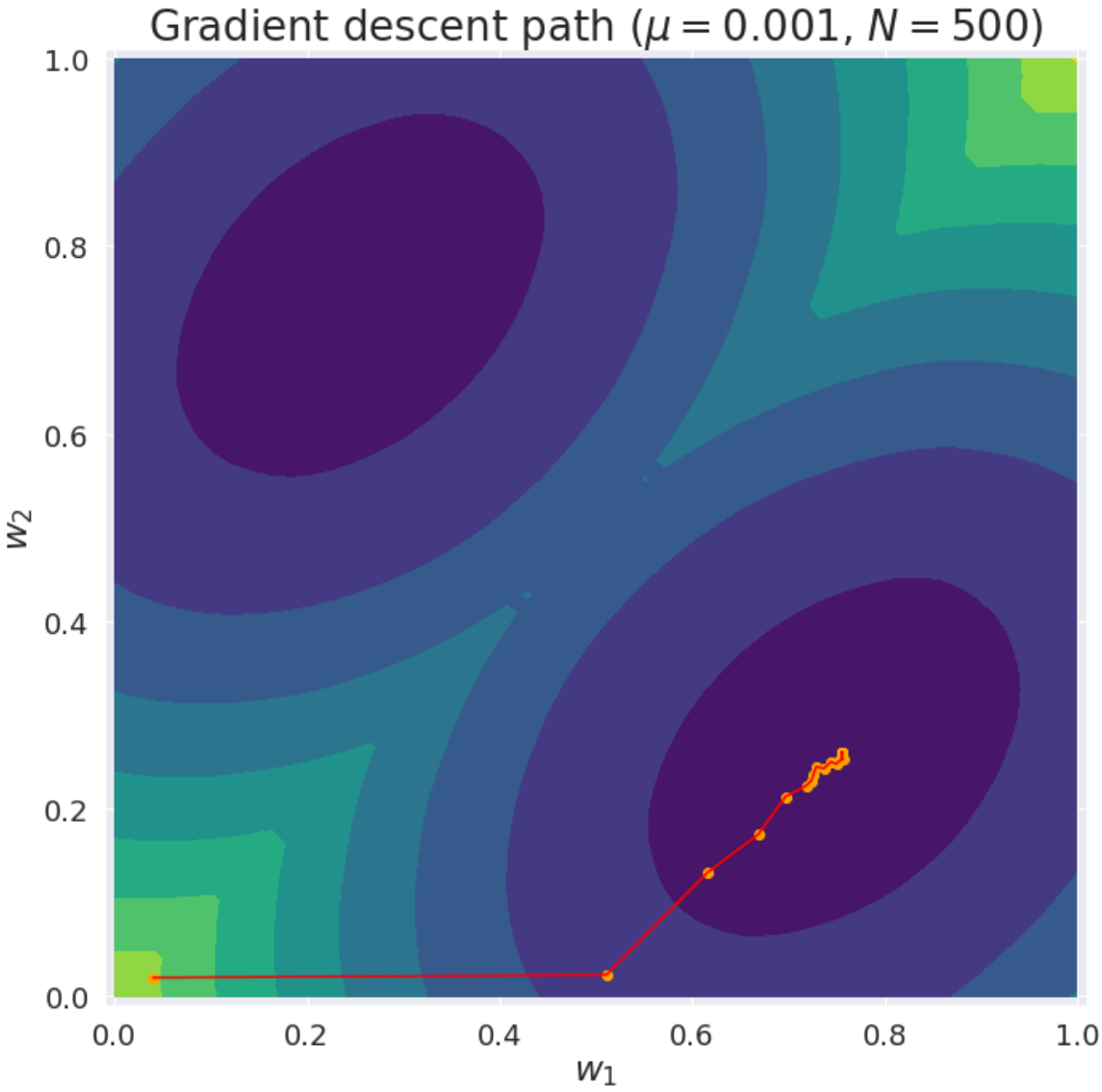
Напишите функцию `gradient_descent`, которая возвращает всю оптимизационную траекторию (w_k) через фиксированное число шагов и рисует процесс на графиках P и ∇P, что были выше (анимацию).

In [369]:

```
def gradient_descent(P, dP, w0, mu, Nsteps, plot=False):
    tr = [w0]
    for i in range(Nsteps):
        tr.append(tr[-1] - mu*dP(tr[-1][0], tr[-1][1], x))
    if plot:
        w1 = np.linspace(0,1,50)
        w2 = np.linspace(0,1,50)
        p = np.zeros([w1.shape[0], w1.shape[0]])

        i = 0
        for w1_ in w1:
            j = 0
            for w2_ in w2:
                p[i][j]  =  P(w1_, w2_, x)
                j += 1
            i += 1

        plt.figure(figsize=(11,11))
        plt.title('Gradient descent path ($\mu=$'+str(mu)+', $N=$'+str(N)+')', f
ontsize=26)
        plt.xlabel('$w_1$', fontsize=22)
        plt.ylabel('$w_2$', fontsize=22)
        plt.xticks(fontsize=18)
        plt.yticks(fontsize=18)
        plt.xlim(-0.01,1.01)
        plt.ylim(-0.01,1.01)
        plt.contourf(w1, w2, p, cmap='viridis')

        w1s = [w[0] for w in tr]
        w2s = [w[1] for w in tr]
        plt.scatter(w1s, w2s, color='orange')
        plt.plot(w1s, w2s, color='red')

        plt.show()

    return tr
```

In [370]:

```
ws = gradient_descent(P, dP, (0.04,0.02), 0.001, 100, True)
```



🤔

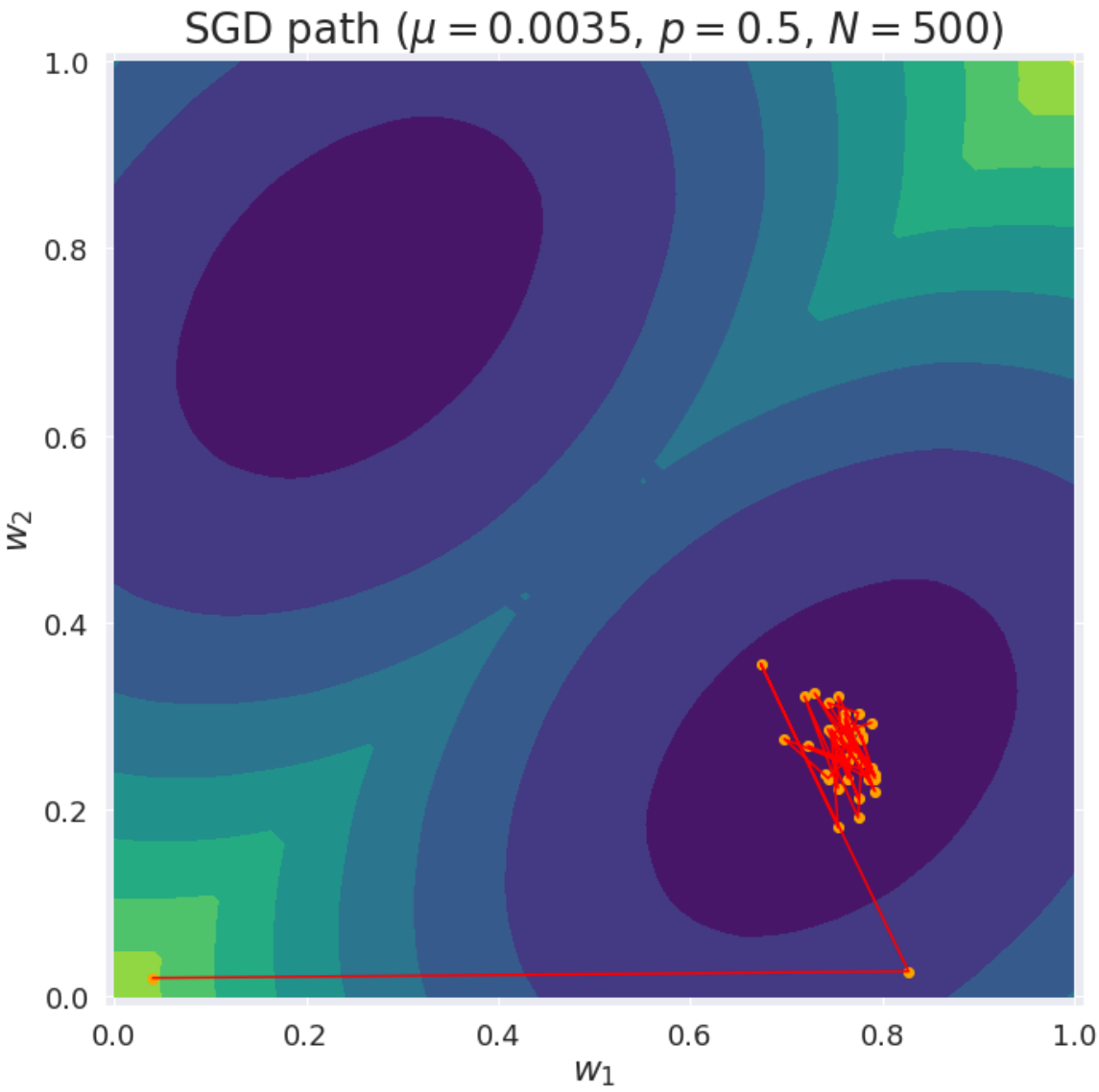
Напишите функцию `stochastic_gradient_descent`, которая возвращает всю оптимизационную траекторию (w_k) метода стохастического градиентного спуска через фиксированное число шагов и рисует процесс на графиках P и ∇P, что были выше (анимацию).

In [367]:

```
def stochastic_gradient_descent(P, dP_sigma, w0, mu, Nsteps, p=0.5, plot=False):
    tr = [w0]
    for i in range(Nsteps):
        tr.append(tr[-1] - mu*dP_sigma(tr[-1][0], tr[-1][1], x, p))
    if plot:
        w1 = np.linspace(0,1,50)
        w2 = np.linspace(0,1,50)
        p_ = np.zeros([w1.shape[0], w1.shape[0]])

        i = 0
        for w1_ in w1:
            j = 0
            for w2_ in w2:
                p_[i][j]  =  P(w1_, w2_, x)
                j += 1
            i += 1

        plt.figure(figsize=(11,11))
        plt.title('SGD path ($\mu=$'+str(mu)+', $p=$'+str(p)+', $N=$'+str(N)+')'
, fontsize=26)
        plt.xlabel('$w_1$', fontsize=22)
        plt.ylabel('$w_2$', fontsize=22)
        plt.xticks(fontsize=18)
        plt.yticks(fontsize=18)
        plt.xlim(-0.01,1.01)
        plt.ylim(-0.01,1.01)
        plt.contourf(w1, w2, p_, cmap='viridis')

        w1s = [w[0] for w in tr]
        w2s = [w[1] for w in tr]
        plt.scatter(w1s, w2s, color='orange')
        plt.plot(w1s, w2s, color='red')

        plt.show()

    return tr
```

In [368]:

```
ws = stochastic_gradient_descent(P, dP_sigma, (0.04,0.02), 0.0035, 40, 0.5, True
)
```



🤔

На практике Вам не так часто нужно будет писать свой собстенный алгоритм оптимизации. Решите поставленную выше задачу (2 станции, роскомнадзоровская норма) любым алгоритмом оптимизации из любой библиотеки Python. (cvxpy, scipy и т.д.) Нарисуйте такие же анимашки.

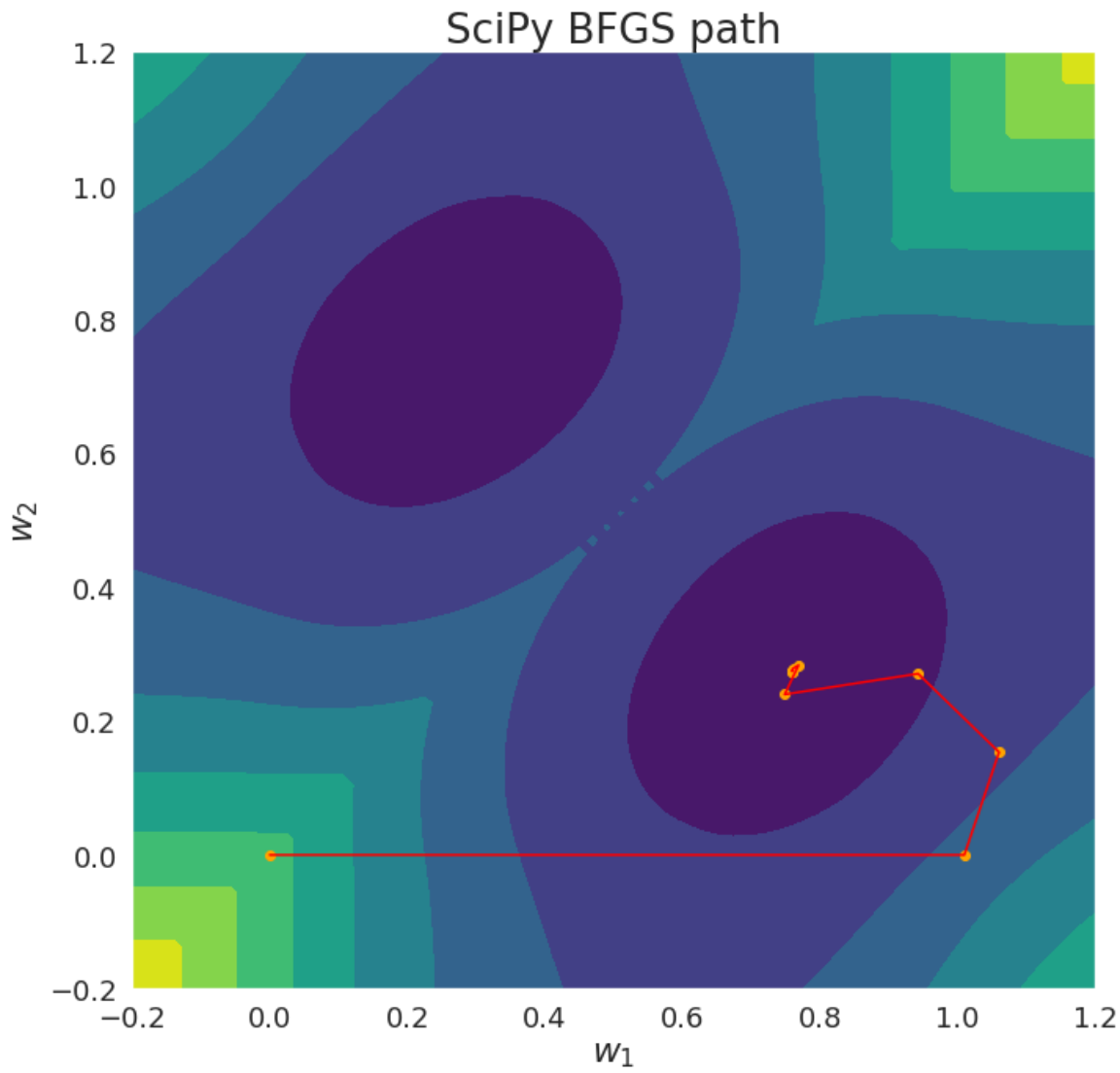
In [371]:

```
N = 1000
l = 1
x = np.random.rand(N)*l
```

In [372]:

```
from scipy.optimize import fmin_bfgs

def P_target(w):
    return P(w[0], w[1], x)

def dP_target(w):
    return dP(w[0], w[1], x)

w0 = np.array([0,0])

w_opt, tr = fmin_bfgs(P_target, w0, dP_target, retall=True)
print(w_opt)

w1 = np.linspace(-0.2,1+0.2,70)
w2 = np.linspace(-0.2,1+0.2,70)
p = np.zeros([w1.shape[0], w1.shape[0]])
dp = np.zeros([w1.shape[0], w1.shape[0]])

i = 0
for w1_ in w1:
    j = 0
    for w2_ in w2:
        p[i][j]  =  P(w1_, w2_, x)
        dp[i][j] = np.linalg.norm(dP(w1_, w2_, x))
        j += 1
    i += 1

plt.figure(figsize=(11,11))
plt.title('SciPy BFGS path', fontsize=26)
plt.xlabel('$w_1$', fontsize=22)
plt.ylabel('$w_2$', fontsize=22)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
plt.xlim(-0.2,1.2)
plt.ylim(-0.2,1.2)
plt.contourf(w1, w2, p, cmap='viridis')

w1s = [w[0] for w in tr]
w2s = [w[1] for w in tr]
plt.scatter(w1s, w2s, color='orange')
plt.plot(w1s, w2s, color='red')

plt.show()
```

```
Warning: Desired error not necessarily achieved due to precision los
s.
         Current function value: 119.444002
         Iterations: 10
         Function evaluations: 71
         Gradient evaluations: 67
[0.75958768 0.27469131]
```



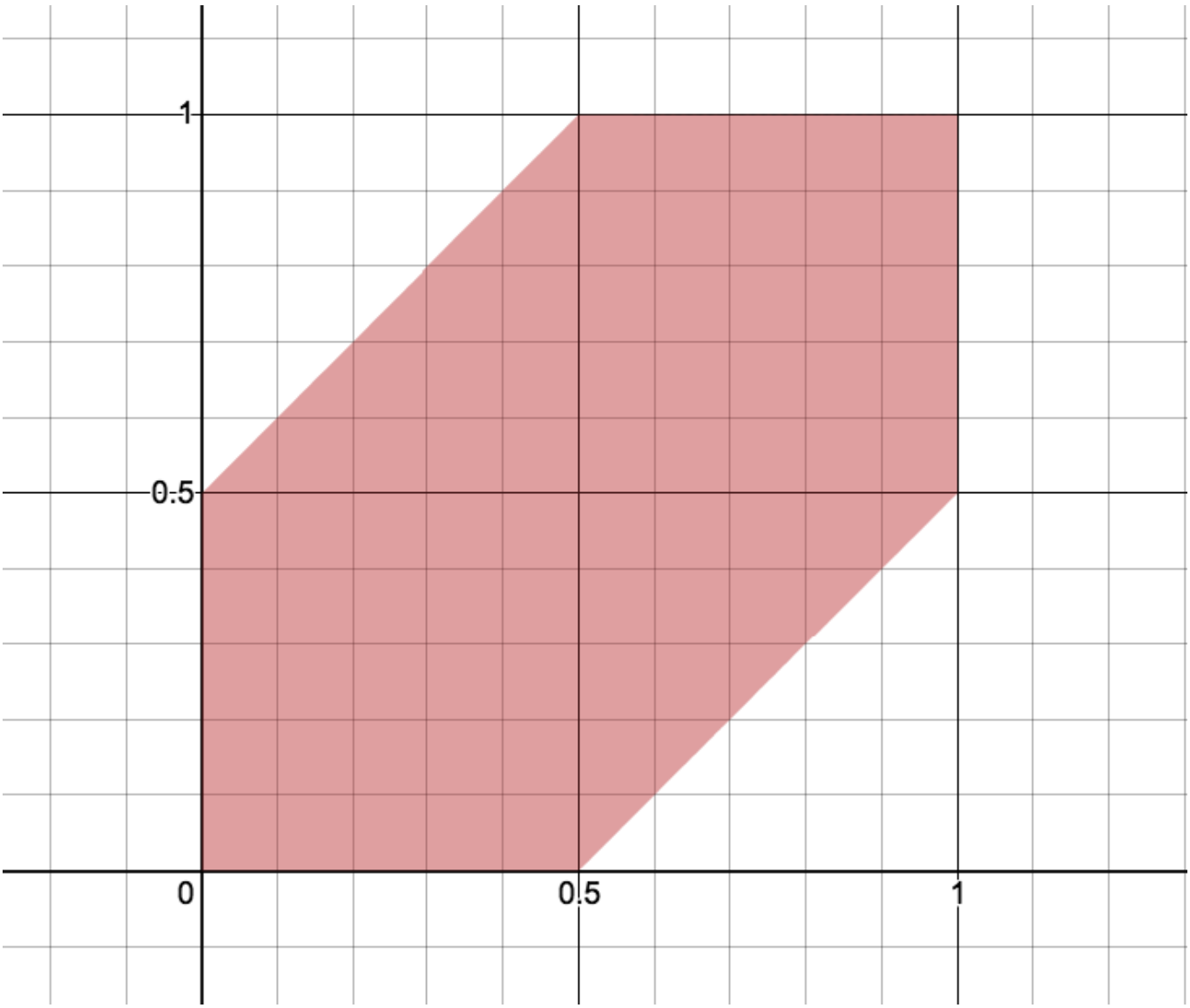
Одномерная деревня научилась решать свои проблемы при любом числе и расположении жителей в мгновение ока. Улучшив качество интернета, хоббитцы смотрели онлайн-курсы и мемесы и жили припеваючи. Как Вы могли догадаться, роскомнадзору это не очень по душе. Вот тут то они выкатили новое требование о том, что дата-центры (роутеры) должны стоять не более, чем на расстоянии половины деревни друг от друга, чтобы силовым структурам было легче собирать информацию о том, как используют интернет жители деревни Одномерново.

$$|w_1 - w_2| \leq \frac{l}{2}$$

🤔

Нарисуйте на плоскости допустимое множество решений. Выпукло ли оно?

Да, оно выпукло (также считаем, что есть ограничения, что станции внутри деревни).



🤔

Напишите функцию `conditional_SGD`, которая возвращает всю оптимизационную траекторию (w_k) метода условного стохастического градиентного спуска через фиксированное число шагов и рисует процесс на графиках P и ∇P, что были выше (анимацию).

Метод условного градиентного спуска заключается в том, чтобы делать градиентный шаг, а после этого проверять принадлежность полученной точки целевому множеству. Если она ему принадлежит, то алгоритм продолжается, иначе делается ортогональная проекция на целевое множество.

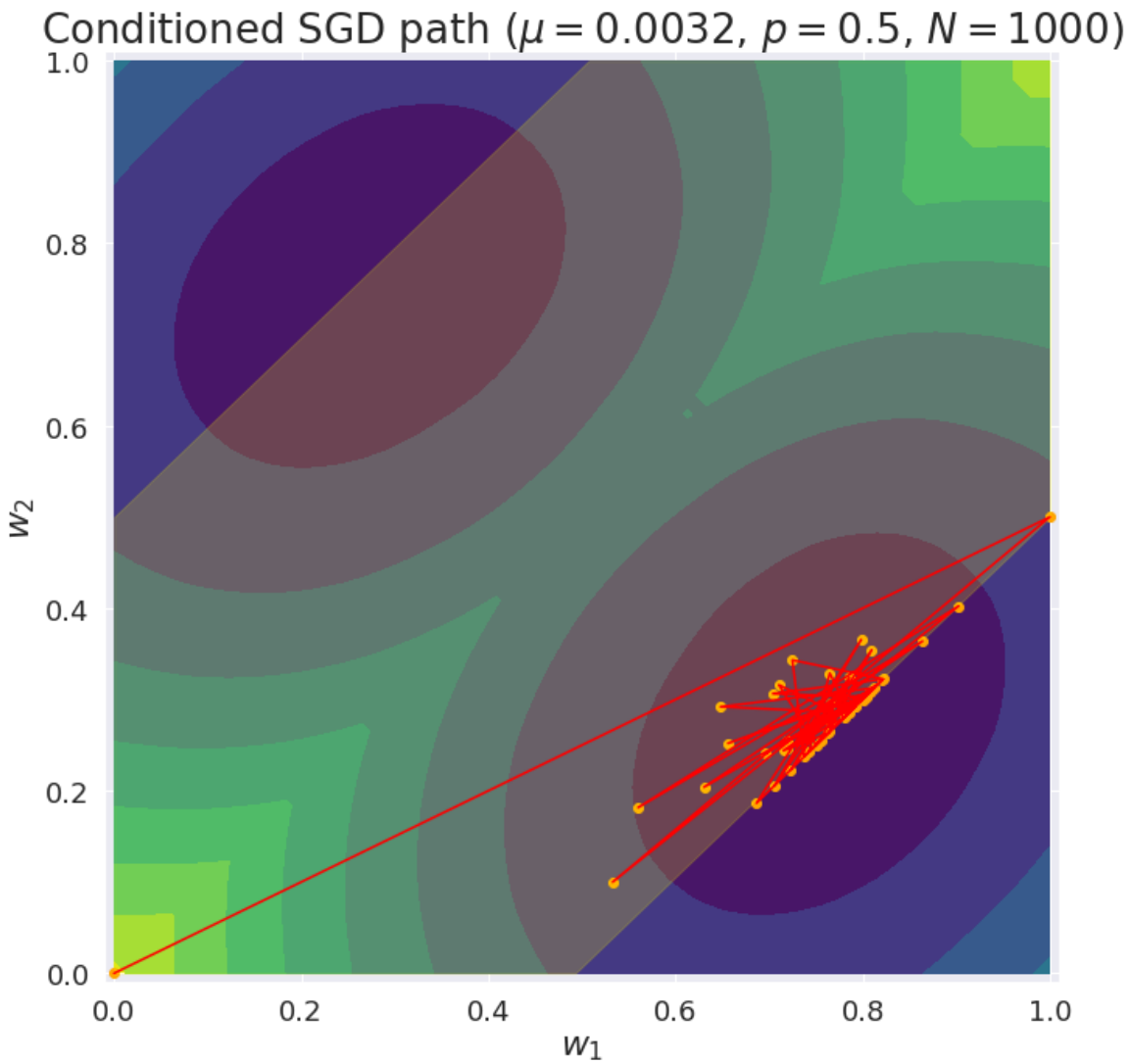
In [394]:

```
def conditional_SGD(P, dP_sigma, w0, mu, Nsteps, p, plot=False):
    tr = [w0]
    for i in range(Nsteps):
        w = tr[-1] - mu*dP_sigma(tr[-1][0], tr[-1][1], x, p)

        # projection
        if np.abs(w[0] - w[1]) > 0.5:
            a = (2*np.abs(w[0] - w[1]) - 1) / 4
            if w[0] > w[1]:
                if w[0] + w[1] < 1.5 and w[0] + w[1] > 0.5:
                    w[0] -= a
                    w[1] += a
                elif w[0] + w[1] >= 1.5:
                    w[0] = min(w[0], 1)
                    w[1] = min(max(w[1], 0.5), 1)
                else:
                    w[0] = min(max(w[0], 0), 0.5)
                    w[1] = max(w[1], 0)
            else:
                if w[0] + w[1] < 1.5 and w[0] + w[1] > 0.5:
                    w[0] += a
                    w[1] -= a
                elif w[0] + w[1] >= 1.5:
                    w[0] = min(max(w[0], 0.5), 1)
                    w[1] = min(w[1], 1)
                else:
                    w[0] = max(w[0], 0)
                    w[1] = min(max(w[1], 0), 0.5)
        else:
            w[0] = min(max(w[0], 0), 1)
            w[1] = min(max(w[1], 0), 1)

        tr.append(w)

    if plot:
        w1 = np.linspace(0,1,50)
        w2 = np.linspace(0,1,50)
        p_ = np.zeros([w1.shape[0], w1.shape[0]])

        i = 0
        for w1_ in w1:
            j = 0
            for w2_ in w2:
                p_[i][j]  =  P(w1_, w2_, x)
                j += 1
            i += 1

        plt.figure(figsize=(11,11))
        plt.title('Conditioned SGD path ($\mu=$'+str(mu)+', $p=$'+str(p)+', $N=
$'+str(N)+')', fontsize=26)
        plt.xlabel('$w_1$', fontsize=22)
        plt.ylabel('$w_2$', fontsize=22)
        plt.xticks(fontsize=18)
        plt.yticks(fontsize=18)
        plt.xlim(-0.01,1.01)
        plt.ylim(-0.01,1.01)
        plt.contourf(w1, w2, p_, cmap='viridis')

        w1s = [w[0] for w in tr]
```

```
        w2s = [w[1] for w in tr]
        plt.scatter(w1s, w2s, color='orange')

        x1 = np.linspace(0, 1, 70)
        y1 = np.concatenate((np.zeros(34), np.linspace(0,0.5,36)))
        y2 = np.concatenate((np.linspace(0.5,1,36), np.ones(34)))
        plt.fill_between(x1, y1, y2, color='yellow', alpha=0.2)

        plt.plot(w1s, w2s, color='red')

        plt.show()
    return tr
```

In [395]:

```
tr = conditional_SGD(P, dP_sigma, (0,0), 0.0032, 40, 0.5, True)
```



🤔

Напишите функцию `multi_newton`, которая реализует решение двумерной задачи с $p(d) = d^3$ методом Ньютона и возвращает траекторию работы алгоритма. Сравните результаты для данной задачи с методом стохастического градиентного спуска.

In [46]:

```
def P(w, x):
    return np.sum(np.amin(np.vstack((np.abs(w[0] - x), np.abs(w[1] - x))), 0) **
3)

def dP(w, x):
    w1 = w[0]
    w2 = w[1]
    d = np.amin(np.vstack((np.abs(w1 - x), np.abs(w2 - x))), 0) # d's
    a1 = np.abs(w1 - x)
    a2 = np.abs(w2 - x)
    g1 = (np.sign(a2 - a1 + 1e-5) + 1) / 2 # partial derivatives w.r.t. a1
    g2 = 1 - g1 # partial derivatives w.r.t. a2
    g1 = g1 * np.sign(w1 - x) * 3 * d * d
    g2 = g2 * np.sign(w2 - x) * 3 * d * d
    return np.array([np.sum(g1), np.sum(g2)])

def dP_sigma(w, x, p=0.5):
    w1 = w[0]
    w2 = w[1]
    random_mask = np.random.binomial(1, p, x.shape)
    d = np.amin(np.vstack((np.abs(w1 - x), np.abs(w2 - x))), 0) # d's
    a1 = np.abs(w1 - x)
    a2 = np.abs(w2 - x)
    g1 = (np.sign(a2 - a1 + 1e-5) + 1) / 2 # partial derivatives w.r.t. a1
    g2 = 1 - g1 # partial derivatives w.r.t. a2
    g1 = g1 * np.sign(w1 - x) * 3 * d * d * random_mask
    g2 = g2 * np.sign(w2 - x) * 3 * d * d * random_mask
    return np.array([np.sum(g1), np.sum(g2)])

def ddP(w, x):
    w1 = w[0]
    w2 = w[1]
    d = np.amin(np.vstack((np.abs(w1 - x), np.abs(w2 - x))), 0)
    a1 = np.abs(w1 - x)
    a2 = np.abs(w2 - x)
    h11 = (np.sign(a2 - a1 + 1e-5) + 1) * 3 * d
    h22 = (-np.sign(a2 - a1 + 1e-5) + 1) * 3 * d - 1e-8
    return np.diag([h11.sum(), h22.sum()])
```

In [51]:

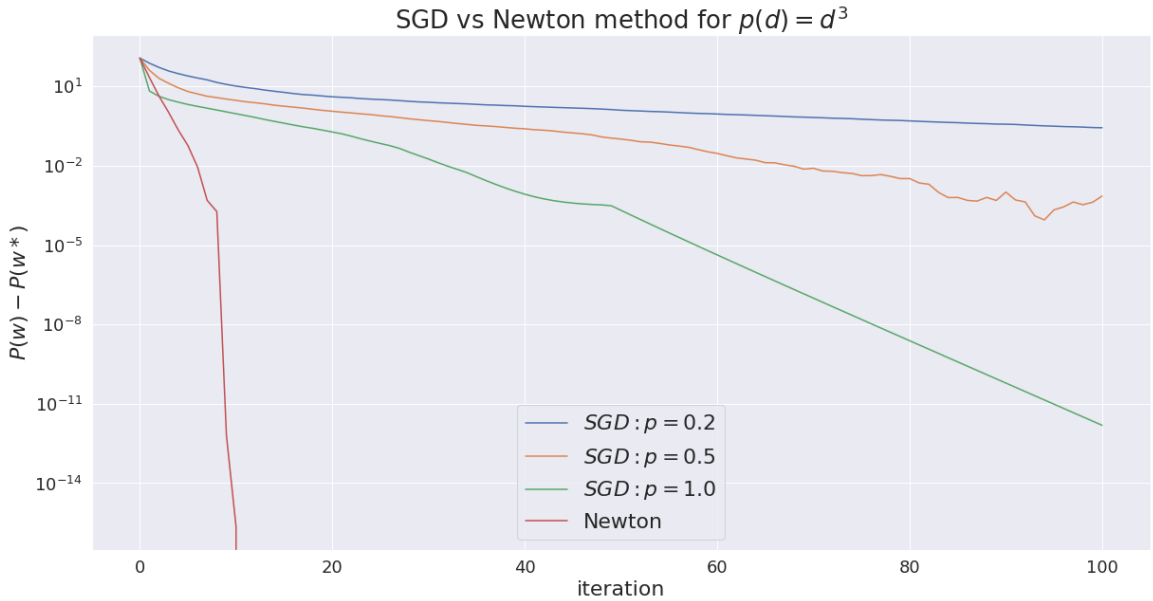
```
def newton_descent(P, dP, ddP, w0, Nsteps):
    tr = [w0]
    for i in range(Nsteps):
        tr.append(tr[-1] - np.linalg.inv(ddP(tr[-1], x)) @ dP(tr[-1], x))
    return tr
```

In [5]:

```
np.random.seed(10)
N = 500
l = 1
x = np.random.rand(N)*l
```

In [67]:

```
ps = [0.2, 0.5, 1.0]
w0 = np.array([0,0])
mu = 1e-3
Nsteps = 100

plt.figure(figsize=(20,10))
plt.title('SGD vs Newton method for $p(d) = d^3$', fontsize=26)
plt.xlabel('iteration', fontsize=22)
plt.ylabel('$P(w) - P(w*)$', fontsize=22)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
for p in ps:
    ws = stochastic_gradient_descent(P, dP_sigma, w0, mu, Nsteps, p)
    #print(ws)
    Ps = [P(w, x) - 1.8823255916935078 for w in ws]
    plt.plot(range(len(ws)), Ps, label=('$SGD: p = $' + str(p)))

ws = newton_descent(P, dP, ddP, w0, Nsteps)
Ps = [P(w, x) - 1.8823255916935078 for w in ws]
plt.plot(range(len(ws)), Ps, label='Newton')
plt.yscale('log')

plt.legend(fontsize=22)
plt.show()
```



Problem 2. Adaptive metrics methods

В этом задании Вам предлагается погрузиться в специфику тестирования алгоритмов оптимизации на одинаковых классах задач на примере методов адаптивной метрики (Newton, Quasi Newton)

Приведем стандартные обозначения.

Афинная инвариантность

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 инициализацией H_0 для метода по координате x и $S^\top H_0 S$ для координаты z

Решение

1. Дифференцируем равенство $\widetilde{f}(z) = f(Sz + s)$ по z_i:

$$\frac{\partial \widetilde{f}}{\partial z_j} = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}\frac{\partial x_i}{\partial z_j} = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} S_{ij} = \left\langle \left(S^T\right)_j, \nabla f(x) \right\rangle \qquad \Longrightarrow \qquad \nabla \tilde{f}(z) = S^T \nabla f(x), \qquad \nabla f(x) = S^-$$

Для доказательства второго равенства рассмотрим $F(x) = \left(S^T \nabla f(x)\right)_j$ и $\widetilde{F}(z) = \left(\nabla \widetilde{f}(z)\right)_j$.

Применяя для них доказанное выше равенство, (во втором равенстве градиент аналогично расписан по формуле сложной функции)

$$\nabla \frac{\partial \widetilde{f}}{\partial z_j} = S^T \nabla \left(S^T \nabla f(x)\right)_j = S^T \left(\nabla^2 f(x) S\right)_j \qquad \Longrightarrow$$

$$\nabla^2 \widetilde{f}(z) = S^T \nabla^2 f(x) S, \qquad \nabla^2 f(x) = S^{-T} \nabla^2 \widetilde{f}(z) S^{-1}$$

2. Теперь рассмотрим метод Ньютона. Итерации для x_k и z_k:

$$x_{k+1} = x_k - \nabla^2 f(x_k)^{-1} \nabla f(x_k), \qquad z_{k+1} = z_k - \nabla^2 \widetilde{f}(z_k)^{-1} \nabla \widetilde{f}(z_k)$$

При $k = 0$ связь выполнена по условию. Допустим, что она выполнена при некотором k. Тогда из доказанного выше

$$\begin{aligned} x_{k+1} &= Sz_k + s - \left(S^{-T} \nabla^2 \widetilde{f}(z_k) S^{-1}\right)^{-1} S^{-T} \nabla \widetilde{f}(z_k) = Sz_k + s - S \nabla^2 \widetilde{f}(z_k)^{-1} S^T S^{-T} \widetilde{f}(z_k) = \\ &= S\left[z_k - \nabla^2 \widetilde{f}(z_k)^{-1} \nabla \widetilde{f}(z_k)\right] + s = Sz_{k+1} + s \end{aligned}$$

3. Теперь рассмотрим метод SR-1. Достаточно доказать, что для оценок обратного гессиана выполнено соотношение

$$\widetilde{B}_k = S^{-1} B_k S^{-T}$$

и тогда из предыдущего доказательство будет следовать связь $x_k = Sz_k + s$. При $k = 0$ это выполнено по условию. Пусть это выполнено при некотором k. Сначала получим вспомогательные связи:

$$\Delta \widetilde{y}_k = \nabla \widetilde{f}(z_{k+1}) - \nabla \widetilde{f}(z_k) = S^T \nabla f(x_{k+1}) - S^T \nabla f(x_k) = S^T \Delta y_k$$

$$\Delta z_k = z_{k+1} - z_k = S^{-1}(x_{k+1} - s) - S^{-1}(x_k - s) = S^{-1} \Delta x_k$$

Они верны при данном k. Тогда, используя индукционное предположение, получаем

$$\widetilde{B}_{k+1} = \widetilde{B}_k + \frac{(\Delta z_k - \widetilde{B}_k \Delta\widetilde{y}_k)(\Delta z_k - \widetilde{B}_k \Delta\widetilde{y}_k)^T}{\langle \Delta z_k - \widetilde{B}_k \Delta\widetilde{y}_k, \Delta\widetilde{y}_k \rangle} =$$

$$= \widetilde{B}_k + \frac{(S^{-1}\Delta x_k - S^{-1}B_k S^{-T}S^T\Delta y_k)(S^{-1}\Delta x_k - S^{-1}B_k S^{-T}S^T\Delta y_k)^T}{\langle S^{-1}\Delta x_k - S^{-1}B_k S^{-T}S^T\Delta y_k, S^T\Delta y_k \rangle} =$$

$$= \widetilde{B}_k + S^{-1}\frac{(\Delta x_k - B_k\Delta y_k)(\Delta x_k - B_k\Delta y_k)^T}{\langle SS^{-1}\Delta x_k - SS^{-1}B_k\Delta y_k, \Delta y_k \rangle} S^{-T} =$$

$$= S^{-1}\left[B_k + \frac{(\Delta x_k - B_k\Delta y_k)(\Delta x_k - B_k\Delta y_k)^T}{\langle \Delta x_k - B_k\Delta y_k, \Delta y_k \rangle}\right] S^{-T} =$$

$$= S^{-1}B_{k+1}S^{-T}$$

4. Теперь рассмотрим метод BFGS. Аналогично,

$$\widetilde{B}_{k+1} = \widetilde{B}_k + \frac{(\Delta z_k^T\Delta\widetilde{y}_k + \Delta\widetilde{y}_k^T\widetilde{B}_k\Delta\widetilde{y}_k)(\Delta z_k\Delta z_k^T)}{(\Delta z_k^T\Delta\widetilde{y}_k)^2} - \frac{\widetilde{B}_k\Delta\widetilde{y}_k\Delta z_k^T + \Delta z_k\Delta\widetilde{y}_k^T\widetilde{B}_k}{\Delta z_k^T\Delta\widetilde{y}_k} =$$

$$= \widetilde{B}_k + S^{-1}\frac{(\Delta x_k^T\Delta y_k + \Delta y_k^T B_k\Delta y_k)(\Delta x_k\Delta x_k^T)}{(\Delta x_k^T\Delta y_k)^2} S^{-T} - S^{-1}\frac{B_k\Delta y_k\Delta x_k^T + \Delta x_k\Delta y_k^T B_k}{\Delta x_k^T\Delta y_k} S^{-T} =$$

$$= S^{-1}\left[B_k + \frac{(\Delta x_k^T\Delta y_k + \Delta y_k^T B_k\Delta y_k)(\Delta x_k\Delta x_k^T)}{(\Delta x_k^T\Delta y_k)^2} - \frac{B_k\Delta y_k\Delta x_k^T + \Delta x_k\Delta y_k^T B_k}{\Delta x_k^T\Delta y_k}\right] S^{-T} =$$

$$= S^{-1}B_{k+1}S^{-T}$$

Newton convergence issue

Рассмотрите следующую функцию:

$$f(x) = \frac{x^4}{4} - x^2 + 2x + (y-1)^2 = f_1(x) + f_2(y)$$

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)

Решение

Градиент и гессиан:

$$\nabla f(x, y) = \begin{bmatrix} f_1'(x) \\ f_2'(y) \end{bmatrix} = \begin{bmatrix} x^3 - 2x + 2 \\ 2(y-1) \end{bmatrix}, \qquad \nabla^2 f(x, y) = \begin{bmatrix} f_1''(x) & 0 \\ 0 & f_2''(y) \end{bmatrix} = \begin{bmatrix} 3x^2 - 2 & 0 \\ 0 & 2 \end{bmatrix}$$

1. **Метод Ньютона** для функции двух переменных расщипляется на два независимых метода Ньютона для одномерных функций:

   $$x_{k+1} = x_k - \frac{f_1'(x)}{f_1''(x)}, \qquad y_{k+1} = y_k - \frac{f_2'(y)}{f_2''(y)}$$

   Так как функция $f_2(y) = (y-1)^2$ квадратичная, то метод Ньютона по y сойдется за одну итерацию.

   Функция $f_1(x)$ не выпукла, поэтому гарантий сходимости метода Ньютона вообще нет. В данной начальной точке функция вогнута, и оказывается, что по x-ам метод Ньютона будет прыгать из 0 в 1 и обратно.

1. **Градиентный спуск** также расщипляется на две независимых процедуры:

   $$x_{k+1} = x_k - \alpha f_1'(x_k), \qquad y_{k+1} = y_k - \alpha f_2'(y_k)$$

   Функция $f_2(y)$ выпукла, поэтому градиентный спуск гарантированно сойдется. Функция $f_1(x)$ не выпукла, однако градиентный спуск тоже сойдется. Докажем это.

   Рассмотрим любой отрезок выпуклости, содержащий решение, $f_1(x)$: $[-2, -\sqrt{2/3}]$. На нем вторая производная $|f_1''(x)| \leq 10 = L$, поэтому если мы начнем спуск на этом отрезке с шагом $\alpha < \frac{2}{L} = 0.2$, то мы гарантированно сойдемся. Осталось показать, что мы можем за конечное число шагов дойти до этого отрезка из точки $x = 0$. Действительно, у $f_1(x)$ нет других точек экстремума, поэтому из гладкости f_1 следует, что производная ограничена снизу. Поэтому на каждой итерации мы будем делать шаг в сторону данного отрезка, и длина этих шагов ограничена снизу. Значит, за конечное число шагов мы придем на $[-2, -\sqrt{2/3}]$. А оттуда мы уже сойдемся, так как $\alpha = 0.01 < 0.2$.

1. **Наискорейший спуск** не расщипляется, потому что шаг α зависит от обоих координат x, y.

   Наискорейший спуск сойдется довольно быстро, потому что первый градиент направлен примерно в сторону оптимума.

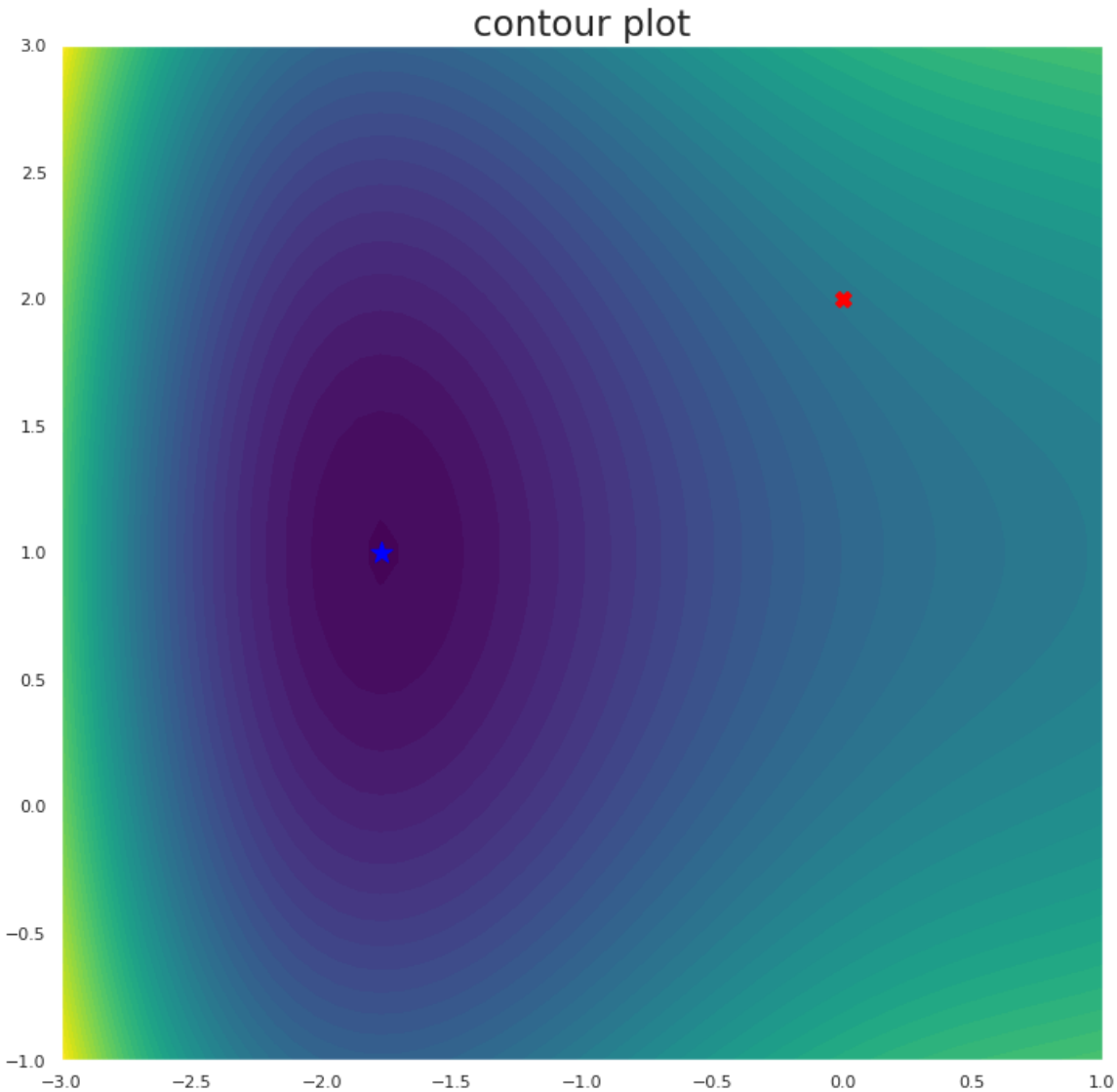
In [85]:

```
x = 0
for i in range(1000):
    x = x - 0.01*(x*x*x - 2*x + 2)
print(x)
```

```
-1.76929235423863
```

In [103]:

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

N = 50
xs = np.linspace(-3,1,N)
ys = np.linspace(-1,3,N)
xs, ys = np.meshgrid(xs, ys)

fs = f(xs, ys)

plt.figure(figsize=(12,12))
plt.contourf(xs, ys, fs, 50, cmap="viridis")
plt.scatter([0], [2], marker='X', s=100, color='red')
plt.scatter([-1.7693], [1], marker='*', s=200, color='blue')
plt.title('contour plot', fontsize=23)
plt.show()
```



Сравнение методов

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 (https://docs.scipy.org/doc/scipy/reference/optimize.minimize-bfgs.html) `BFGS` из библиотеки `scipy`, а так же его модификацию L-BFGS (https://docs.scipy.org/doc/scipy/reference/optimize.minimize-lbfgsb.html) 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: точность по координате $\varepsilon = 10^{-6}$.

Число запусков: через %%timeit в среднем от 700 до 7000 (не менее 70 для самых долгих).

Начальная точка в квадратичных задачах бралась из всех единиц, а для функции Розенброка $(4, -2)$. B_0 в методе SR-1 инициализируется единичной матрицей.

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, ms**
Newton	1	0.079
SR-1	3	0.115
BFGS	5	0.646
L-BFGS	8	0.433

Квадратичная форма. n = 50	**Iterations**	**Time, ms**
Newton	1	0.301
SR-1	52	2.85
BFGS	64	11.1
L-BFGS	237	10.5

Квадратичная форма. n = 228	**Iterations**	**Time, ms**
Newton	1	3.78
SR-1	231	106.0
BFGS	238	545

Квадратичная форма. n = 228	Iterations	Time, ms
L-BFGS	755	80.5

Функция Розенброка	Iterations	Time, ms
Newton	5	0.348
SR-1	222	7.4
BFGS	63	6.53
L-BFGS	29	1.06

In [510]:

```
np.random.seed(11)

N = 228
A = np.random.normal(0, 2, size=(N,N))
A = A.T @ A
b = np.random.normal(0, 4, size=(N,))

def f1(x):
    return 0.5 * x @ (A @ x) + b @ x

def f1_opt():
    return np.linalg.solve(A, -b)

def f1_grad(x):
    return A @ x + b

def f1_hes(x):
    return A

def f2(x):
    return (1-x[0])**2 + 100 * (x[1] - x[0]*x[0])**2

def f2_grad(x):
    return np.array([2*(x[0] - 1) + 400*x[0]*(x[0]*x[0] - x[1]), 200 * (x[1] - x
[0]*x[0])])

def f2_hes(x):
    return np.array([[2+1200*x[0]*x[0]-400*x[1], -400*x[0]], [-400*x[0], 200]])

def f2_opt():
    return np.array([1,1])
```

In [399]:

```
def newton_descent(f, f_grad, f_hes, x0, opt, eps, max_iter=50):
    x = x0
    for i in range(max_iter):
        x = x - np.linalg.inv(f_hes(x)) @ f_grad(x)
        if np.linalg.norm(x - opt) < eps:
            #print("eps =", eps, ",", i+1, "iterations")
            break
    return x, i+1
```

In [400]:

```
def sr1(f, f_grad, x0, B0, opt, eps, max_iter=10000):
    x_ = x0
    B = B0
    for i in range(max_iter):
        if i == 0:
            g_ = f_grad(x_)
            x = x_ - B @ g_
            continue
        dx = x - x_
        g = f_grad(x)
        dy = g - g_
        c = (dx - B @ dy).reshape((-1,1))
        B = B + (c @ c.T) / (np.dot(c.flatten(), dy))
        x_ = x
        g_ = g
        x = x - B @ g

        if np.linalg.norm(x - opt) < eps:
            #print("eps =", eps, ",", i+1, "iterations")
            break
    return x, i+1
```

1. Newton descent

In [426]:

```
%%timeit
x0 = np.ones(N)
newton_descent(f1, f1_grad, f1_hes, x0, f1_opt(), 1e-6)
```

```
4.12 ms ± 187 µs per loop (mean ± std. dev. of 7 runs, 100 loops eac
h)
```

In [433]:

```
%%timeit
x0 = np.array([4,-2])
newton_descent(f2, f2_grad, f2_hes, x0, f2_opt(), 1e-6)
```

```
336 µs ± 18.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops eac
h)
```

2. SR-1

In [476]:

```
#%%timeit
x0 = np.ones(N)
B0 = np.eye(N)
sr1(f1, f1_grad, x0, B0, f1_opt(), 1e-6)
```

Out[476]:

```
(array([-0.84613964,  3.00366668]), 3)
```

In [485]:

```
#%%timeit
x0 = np.array([4,-2])
B0 = np.eye(2)
sr1(f2, f2_grad, x0, B0, f2_opt(), 1e-6, 10000)
```

Out[485]:

```
(array([0.99999999, 0.99999999]), 222)
```

3. BFGS

In [456]:

```
import scipy
```

In [489]:

```
%%timeit
x0 = np.ones(N)
scipy.optimize.minimize(f1, x0, method='BFGS', jac=f1_grad, tol=1e-6)
```

```
545 ms ± 10.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [486]:

```
%%timeit
x0 = np.array([4,-2])
scipy.optimize.minimize(f2, x0, method='BFGS', jac=f2_grad, tol=1e-6)
```

```
6.53 ms ± 173 µs per loop (mean ± std. dev. of 7 runs, 100 loops eac
h)
```

4. L-BFGS

In [514]:

```
%%timeit
x0 = np.ones(N)
scipy.optimize.minimize(f1, x0, method='L-BFGS-B', jac=f1_grad, tol=1e-6)
```

```
76.1 ms ± 16.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops eac
h)
```

In [501]:

```
%%timeit
x0 = np.array([4,-2])
scipy.optimize.minimize(f2, x0, method='L-BFGS-B', jac=f2_grad, tol=1e-6)
```

```
1.06 ms ± 106 µs per loop (mean ± std. dev. of 7 runs, 1000 loops eac
h)
```

Problem 3. 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
$\alpha_k := \frac{\mathbf{r}_k^\top \mathbf{r}_k}{\mathbf{p}_k^\top \mathbf{A}\mathbf{p}_k}$
$\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k$
$\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$
if $\mathbf{r}_{k+1}$ is sufficiently small, then exit loop
$\beta_k := \frac{\mathbf{r}_{k+1}^\top \mathbf{r}_{k+1}}{\mathbf{r}_k^\top \mathbf{r}_k}$
$\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$
$k := k + 1$
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 (https://stanford.edu/class/ee364b/lectures/conj_grad_slides.pdf) 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 (http://www.math.iit.edu/~fass/477577_Chapter_16.pdf) 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:

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

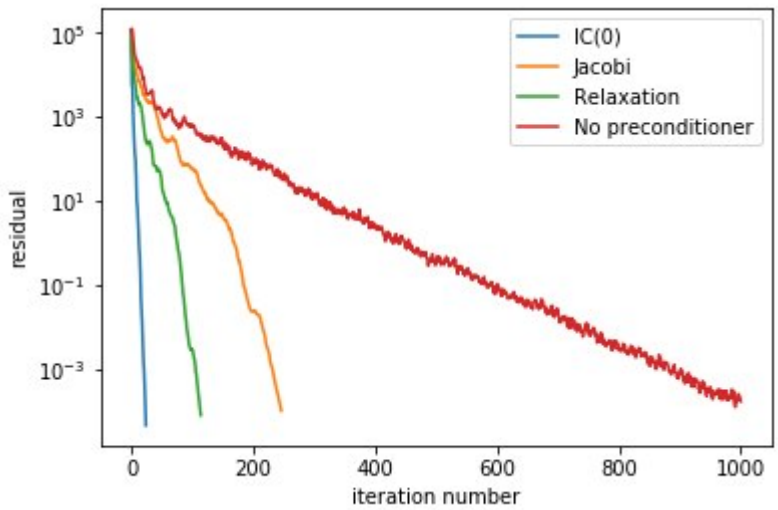
Факторизация Холецкого s.p.d. матрицы A - ее разложение на произведение нижнетреугольной и верхнетреугольной матрицы: $A = L^T L$ wiki (https://en.wikipedia.org/wiki/Cholesky_decomposition). Есть несколько упрощений этого алгоритма, позволяющих получить матрицу, "похожую" на A. Мы будем использовать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

Задание Выбрать 1 задачу отсюда (https://sparse.tamu.edu/), 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:



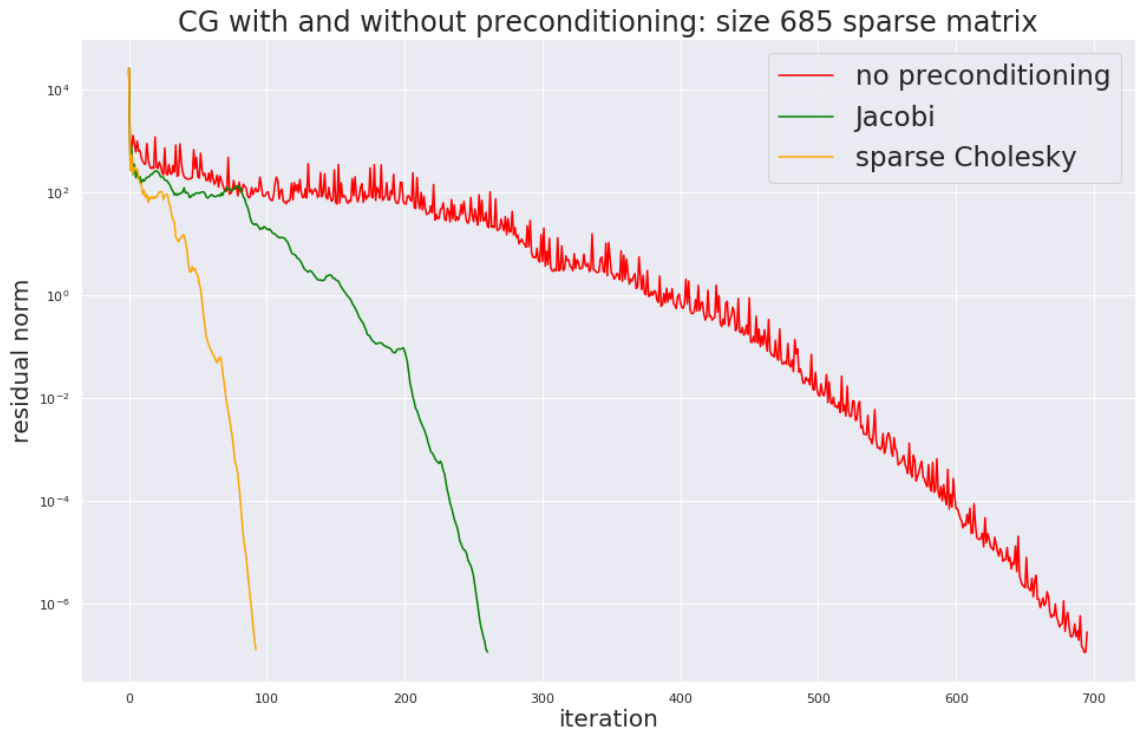
Решение

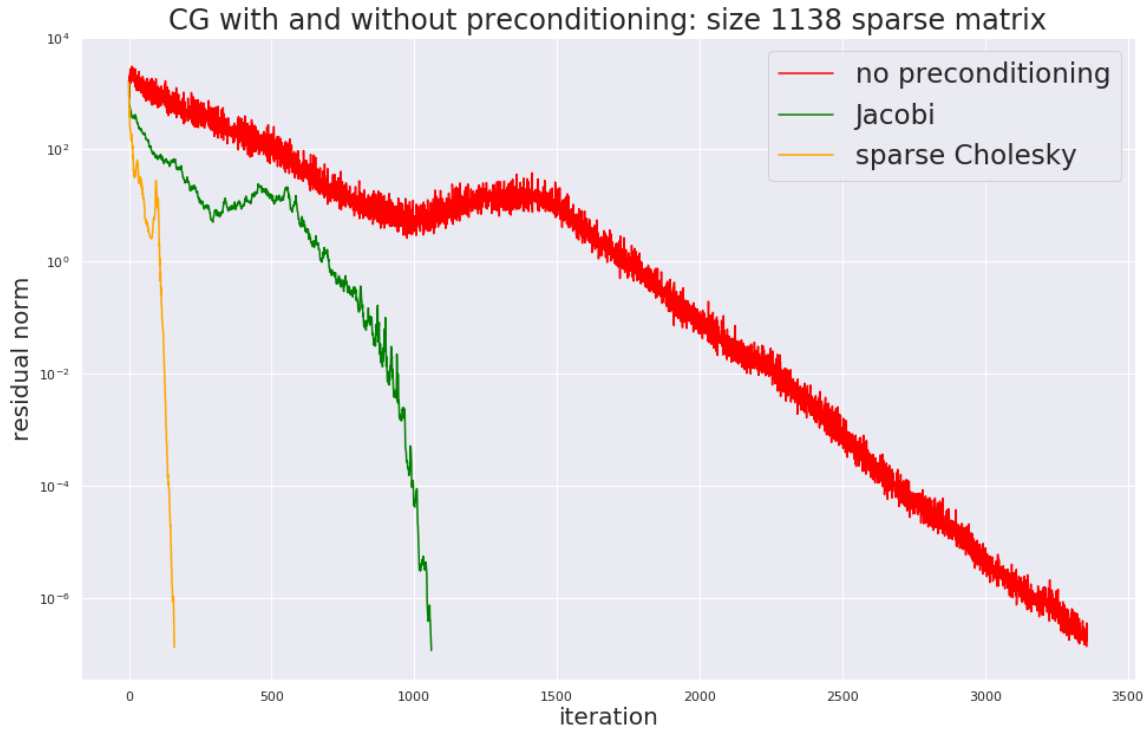
Используются две симметричные положительно определенные матрицы:

- матрица №4 `HB/685_bus` размера 685×685 с 0.69% ненулевыми элементами (число обусловленности $\kappa(A) = 4.2 \cdot 10^5$)
- матрица №1 `HB/1138_bus` размера 1138×1138 с 0.38% ненулевыми элементами (число обусловленности $\kappa(A) = 8.6 \cdot 10^6$)

Сравнение числа итераций до достижения точности 10^{-7} по норме невязки:

Число итераций	**685_bus**	**1138_bus**
no preconditioning	695	3358
Jacobi	260	1060
incomplete (sparse) Cholesky	92	160





In [175]:

```
from scipy.io import mmread
np.random.seed(10)

A = mmread('685_bus.mtx').toarray()
b = np.random.normal(0, 20, (A.shape[0],))
```

In [176]:

```
np.random.seed(11)

A = mmread('1138_bus.mtx').toarray()
b = np.random.normal(0, 20, (A.shape[0],))
```

In [136]:

```
def chol(A):
    n = A.shape[0]
    L = np.zeros_like(A)
    for j in range(n):
        L[j,j] = np.sqrt(A[j,j] - (L*L)[j,:j].sum())
        for i in range(j, n):
            L[i,j] = 1/L[j,j] * (A[i,j] - (L[i,:j] * L[j,:j]).sum())
    return L

def chol_sparse(A):
    n = A.shape[0]
    L = np.zeros_like(A)
    for j in range(n):
        L[j,j] = np.sqrt(A[j,j] - (L*L)[j,:j].sum())
        for i in range(j, n):
            if A[i,j] == 0:
                L[i,j] = 0
                continue
            L[i,j] = 1/L[j,j] * (A[i,j] - (L[i,:j] * L[j,:j]).sum())
    return L
```

In [120]:

```
import numpy as np

def conjugate_grad(A, b, x=None, max_iter_mult=1.0, eps=1e-7):
    """
    Description
    -----------
    Solve a linear equation Ax = b with conjugate gradient method.
    Parameters
    ----------
    A: 2d numpy.array of positive semi-definite (symmetric) matrix
    b: 1d numpy.array
    x: 1d numpy.array of initial point
    Returns
    -------
    list of residuals
    """
    n = len(b)
    rks = []
    if x is None:
        x = np.ones(n)
    r = b - A @ x
    #print(r[:3])
    p = np.copy(r)
    r_k_norm = r.T @ r
    for i in range(int(max_iter_mult*n)):
        #print(i+1)
        rks.append(np.sqrt(r_k_norm))
        Ap = A @ p
        alpha = r_k_norm / (p @ Ap)
        x += alpha * p
        r -= alpha * Ap
        #print(i, r[:3])
        r_kplus1_norm = r @ r
        beta = r_kplus1_norm / r_k_norm
        #print(i, beta)
        r_k_norm = r_kplus1_norm
        if np.sqrt(r_kplus1_norm) < eps:
            print('Iterations:', i)
            break
        #print(i, r[0:2], beta, p[0:2])
        p = r + beta * p
        #print(i, p[:3])

    return x, rks

def preconditioned_conjugate_grad(A, b, M, x=None, max_iter_mult=1.0, eps=1e-7):
    """
    Description
    -----------
    Solve a linear equation Ax = b with conjugate gradient method.
    Parameters
    ----------
    A: 2d numpy.array of positive semi-definite (symmetric) matrix
    M: 2d numpy.array of positive semi-definite (symmetric) matrix - preconditio
ner (already inverted)
    b: 1d numpy.array
    x: 1d numpy.array of initial point
    Returns
    -------
```

```
    list of residuals
    """
    n = len(b)
    rks = []
    if x is None:
        x = np.ones(n)
    r = b - A @ x
    #print(r[:3])
    z = M @ r
    p = np.copy(z)
    r_k_norm = np.linalg.norm(r)
    r_k_z_k = r @ z
    for i in range(int(max_iter_mult*n)):
        #print(i+1)
        rks.append(r_k_norm)
        Ap = A @ p
        alpha = (r_k_z_k) / (p @ Ap)
        x += alpha * p
        r -= alpha * Ap
        #print(i, r[:3])
        r_kplus1_norm = np.linalg.norm(r)
        z = M @ r
        r_k1_z_k1 = r @ z
        beta = r_k1_z_k1 / r_k_z_k
        #print(i, beta)
        r_k_norm = r_kplus1_norm
        r_k_z_k = r_k1_z_k1
        if r_kplus1_norm < eps:
            print('Iterations:', i)
            break
        #print(i, z[0:2], beta, p[0:2])
        p = z + beta * p
        #print(i, p[:3])

    return x, rks
```

In [177]:

```
x1, rks1 = conjugate_grad(A, b, x=np.ones_like(b), eps=1e-7, max_iter_mult=50)

M_jacobi = np.diag(1 / np.diag(A))
x2, rks2 = preconditioned_conjugate_grad(A, b, M_jacobi, x=np.ones_like(b), eps=
1e-7, max_iter_mult=50)

L = chol_sparse(A)
L_inv = scipy.linalg.solve_triangular(L, np.eye(A.shape[0]), lower=True)
M_chol = L_inv.T @ L_inv
x3, rks3 = preconditioned_conjugate_grad(A, b, M_chol, x=np.ones_like(b), eps=1e
-7, max_iter_mult=50)
```

```
Iterations: 3358
Iterations: 1060
Iterations: 160
```

In [ ]:

```
plt.figure(figsize=(16,10))
plt.plot(rks1, color='red', label='no preconditioning')
plt.plot(rks2, color='green', label='Jacobi')
plt.plot(rks3, color='orange', label='sparse Cholesky')
plt.yscale('log')
plt.xlabel('iteration', fontsize=20)
plt.ylabel('residual norm', fontsize=20)
plt.title('CG with and without preconditioning: size 685 sparse matrix', fontsiz
e=24)
plt.legend(fontsize=23)
#plt.savefig('task3_1138.png')
plt.show()
```

Problem 4. 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.

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

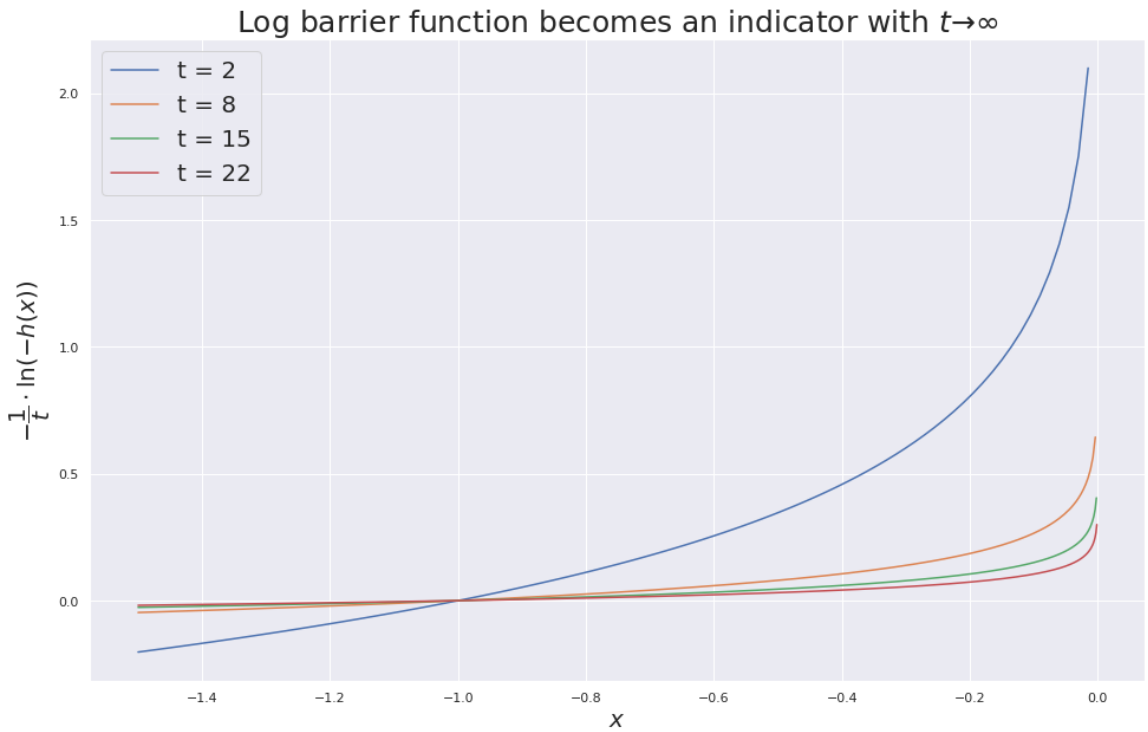
Вместо этого предлагается рассмотреть задачу безусловной оптимизации:

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

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)

In [196]:

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

t = np.linspace(2,22, 4)

plt.figure(figsize=(16,10))
for t_ in t:
    x = np.linspace(-1.5,0, 50*int(t_), endpoint=False)
    y = -np.log(-x)
    plt.plot(x,y/t_, label = 't = '+str(int(t_)))

plt.title('Log barrier function becomes an indicator with $t \\to \infty$', font
size=25)
plt.ylabel('$-\dfrac{1}{t} \cdot \ln(-h(x))$', fontsize=20)
plt.xlabel('$x$', fontsize=20)
plt.legend(fontsize=20)
plt.show()
```



Таким образом, формируется последовательность 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 $k = 1, 2, 3, \ldots$

- 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 (https://statweb.stanford.edu/~candes/teaching/acm113/Handouts/sumt.pdf) 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)`

Решение

Сначала сгенерируем задачу линейного программирования и найдем ее решение. Вектор b берем неотрицательным, чтобы легко было взять начальную точку из допустимого множества (из всех нулей).

Функция потерь в зависимости от параметра t (a_i^T - i-ая строка матрицы A):

$$f(x;t) = t \cdot c^T x + \sum_{i=1}^{m} \left[-\ln\left(-(a_i^T x - b_i)\right) \right]$$

Ее градиент:

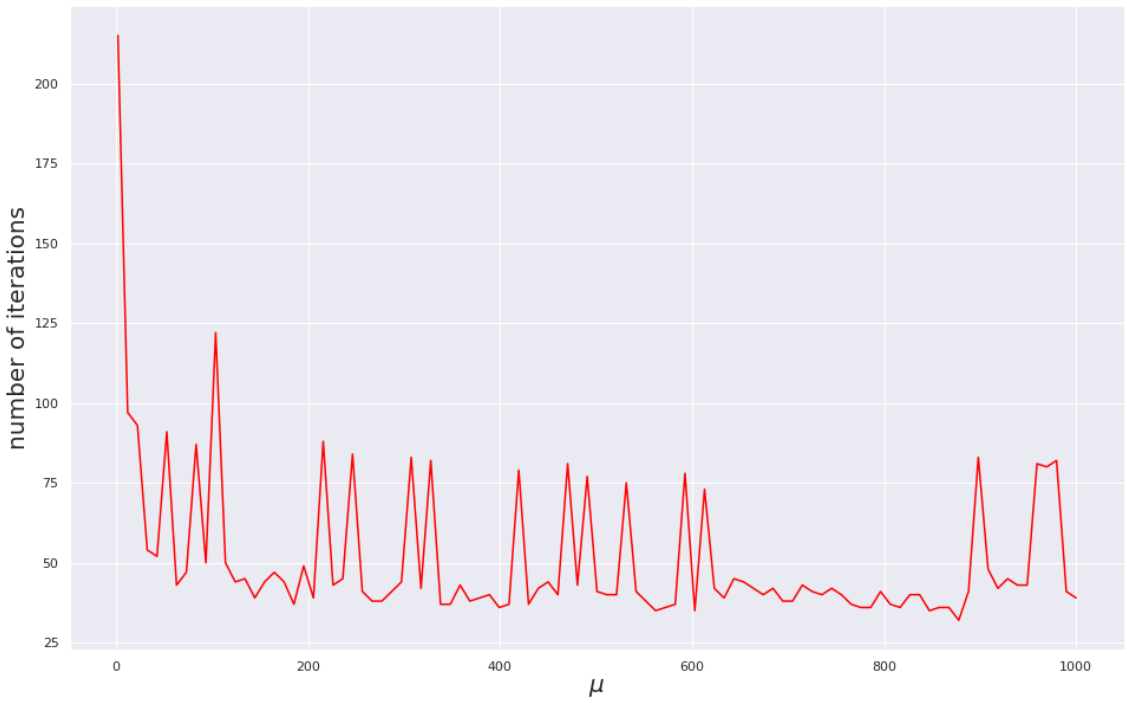
$$\nabla f(x,t) = t \cdot c + \sum_{i=1}^{m} \frac{a_i}{b_i - a_i^T x}$$

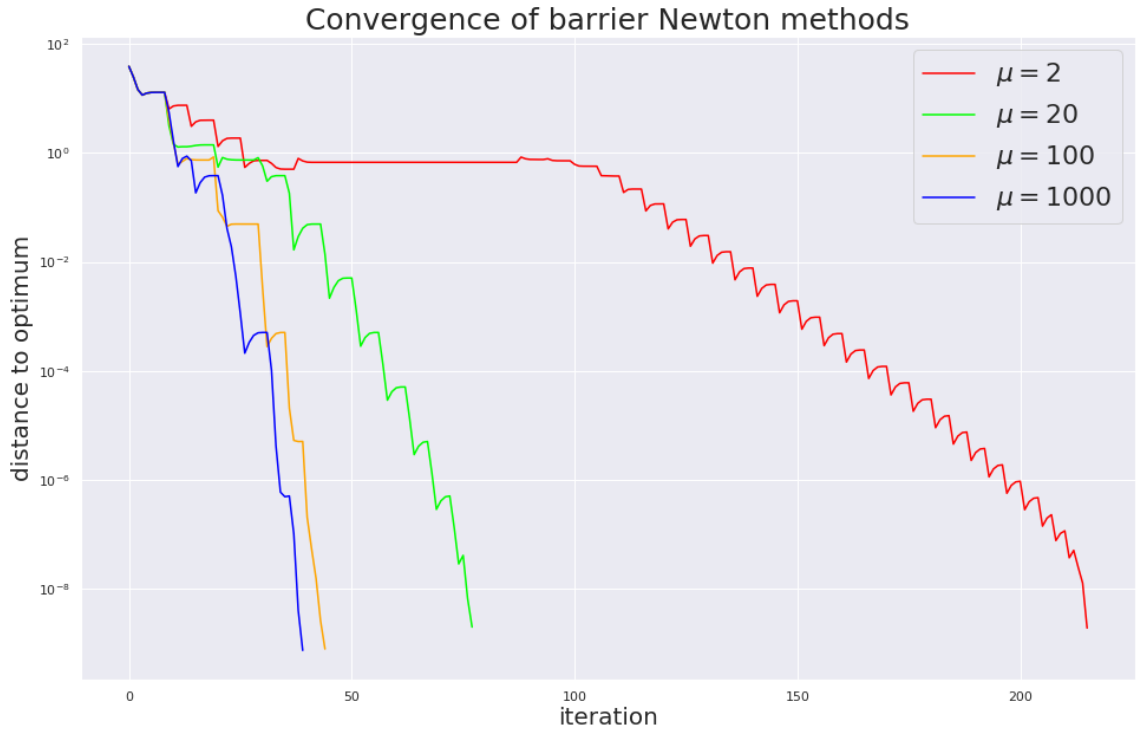
Ее гессиан:

$$\nabla^2 f(x,t) = \sum_{i=1}^{m} \frac{a_i a_i^T}{(b_i - a_i^T x)^2} \succeq 0,$$

т.е. задача выпукла. Однако шаг метода Ньютона может выпрыгнуть за пределы допустимого множества, поэтому мы будем делать поиск следующей допустимой точки вдоль направления спуска.

Критерий остановки в решении "барьерной" задачи: норма градиента меньше ε (непонятно, какой критерий имеется в виду в описанном в задании алгоритме, поэтому я выбрал этот).





In [459]:

```
np.random.seed(50)
m = 100
n = 50
c = np.random.normal(1, 5, size=(n,))
A = np.random.normal(0, 5, size=(m // 2,n))
A = np.vstack((A,-A))
b = np.random.uniform(1, 4, size=(m,))
```

In [460]:

```
from scipy.optimize import linprog

bounds = [(None, None)] * 50
res = scipy.optimize.linprog(c, A_ub=A, b_ub=b, bounds=bounds, options={"disp":
True})
#print(res)
opt = res['x']
```

```
Optimization terminated successfully.
         Current function value: -92.269259
         Iterations: 327
```

In [496]:

```
def f(x, t):
    return t * c @ x - np.log(b - A @ x).sum()

def f_grad(x, t):
    return t * c + np.sum(A / ((b - A @ x)[:,None]), axis=0)

def f_hes(x, t):
    den = (b - A @ x) ** 2
    res = np.zeros((x.shape[0], x.shape[0]))
    for i in range(A.shape[0]):
        res += A[i].reshape((-1,1)) @ A[i].reshape((1,-1)) / den[i]
    return res
```

In [535]:

```
def barrier_newton_descent(f, f_grad, f_hes, x0, t0, mu, eps, opt, alpha=0.0, be
ta=0.7, max_iter=50):
    # beta - parameter of line search

    t = t0
    x = x0
    norm = np.linalg.norm(x - opt)
    res = [norm]
    i = 0

    while t <= int(1e12):

        j = 0
        while j < max_iter:
            g = f_grad(x, t)
            v = - np.linalg.inv(f_hes(x, t)) @ g # Newton descent direction
            if np.linalg.norm(v) < eps:
                break
            s = 1 # descent rate
            while np.min(b - A @ (x + s*v)) <= 0:
                #print(s)
                s = beta * s
            df = g @ v
            f_ = f(x,t)
            while f(x+s*v, t) - f_ > alpha * s * df:
                s = beta * s
            x = x + s*v
            norm = np.linalg.norm(x - opt)
            #print(i+j, t, norm)
            res.append(norm)
            j += 1

        t = mu * t
        i += j

    return x, res
```

In [552]:

```
x0 = np.zeros_like(c)
x1, res1 = barrier_newton_descent(f, f_grad, f_hes, x0, 1, 2, 1e-8, opt, max_ite
r=50)
x2, res2 = barrier_newton_descent(f, f_grad, f_hes, x0, 1, 10, 1e-8, opt, max_it
er=50)
x3, res3 = barrier_newton_descent(f, f_grad, f_hes, x0, 1, 100, 1e-8, opt, max_i
ter=50)
x4, res4 = barrier_newton_descent(f, f_grad, f_hes, x0, 1, 1000, 1e-8, opt, max_
iter=50)
```

In [541]:

```
mus = np.linspace(2,1000,99)
iters = []
x0 = np.zeros_like(c)

for mu in mus:
    x, res = barrier_newton_descent(f, f_grad, f_hes, x0, 1, mu, 1e-8, opt, max_
iter=50)
    iters.append(len(res)-1)
```

In [ ]:

```
plt.figure(figsize=(16,10))
plt.plot(mus, iters, color='red')
plt.xlabel('$\mu$', fontsize=20)
plt.ylabel('number of iterations', fontsize=20)
plt.savefig('task4_mu.png')
plt.show()
```

In [ ]:

```
plt.figure(figsize=(16,10))
plt.plot(res1, color='red', label='$\mu = 2$')
plt.plot(res2, color='lime', label='$\mu = 20$')
plt.plot(res3, color='orange', label='$\mu = 100$')
plt.plot(res4, color='blue', label='$\mu = 1000$')
plt.yscale('log')
plt.xlabel('iteration', fontsize=20)
plt.ylabel('distance to optimum', fontsize=20)
plt.title('Convergence of barrier Newton methods', fontsize=25)
plt.legend(fontsize=22)
plt.savefig('task4_conv.png')
plt.show()
```