1. Numpy

Задание 1. Какие способы позволяют создать матрицу размером два на два, где все элементы равны единице ?

Задание 2. Что будет выведено на экран ?

```
import numpy as np

a = [1, 2, 3]

b = np.array([1, 2, 3])

print(a + a, b + b)
```

Задание 3. Что будет в переменной `x` ?

```
import numpy as np


a = [1, 2, 3, 4, 5, 6]

b = [10, 11, 12, 13, 14, 15]

x = np.min((np.max(a), np.min(b)))
```

Задание 4. Что будет выведено на экран?

```
import numpy as np

A = np.array([[10, 1], [20, 2]])

x = np.sum(A, axis = 0)

print(x)
```

Задание 5. Что будет выведено на экран?

```
import numpy as np


I = np.eye(2)

A = np.array([[0, 1], [1, 0]])
```

```
x = np.sum( np.dot(A, I) )

y = np.sum( A * I )

print(x, y)
```

Задание 6. Дана матрица A , все элементы которой нулевые, кроме одного.

Напишите функцию nonzero(A) для поиска индекса строки и столбца ненулевого элемента.

```
import numpy as np

def nonzero(A):
    """
    A: <np.ndarray> - матрица
    ---------------
    Returns: x, y: <int>, <int> - найденный индекс строки и столбца, соответственно
    """
    # your code here


    return x, y

A = np.zeros((100,100))
A[56,70] = 1

print(nonzero(A))
```

Задание 7. Задан массив `x`, выведите индексы первых трех наименьших элементов.

Sample Input:

```
[3, 2, 5, 4, 1, 7]
```

Sample Output:

```
[4 1 0]
```

Задание 8. Предположим, что у нас имеется некоторая хаотичная структура из атомов, координаты (x,y,z) которых известны. Напишите функцию `closest` , которая определяет для `v-oro` атома количество атомов `k` , лежащих на расстоянии меньше `r` от него. Для оценки расстояния используйте Евклидову норму.

Пример работы программы:

```
r = 2.5
v = 1

vecs = np.array(
    [
        [1.0, 0.0, 2.0],
        [-1.0, 0.5, 2.0],
        [-2.0, 1.5, 0.0],
        [2.5, -1.2, -0.5],
        [1.5, 0.2, -0.2]
    ]
)

print(closest(vecs, v, r))   # Вернет 2


r = 0.1
v = 0

vecs = np.array([[0.71, 0.7, 0.22, 0.98, 0.01],
        [0.25, 0.43, 0.78, 0.2, 0.86]])

vecs = np.array([[0.71, 0.7, 0.22, 0.98, 0.01]])
```

2. Matplotlib

2.1 График функции

Создать массив x и массив значений функции y(x) по формуле $y(x) = \sin^2(2x)e^{\left(\frac{x+2}{10}\right)^2}$ на интервале от -10 до 10, разбитый на 1000 точек.

График функции строиться с использованием строк кода:

```
plt.plot(x, y)
plt.show()
```

Добавить размер экрана

```
plt.figure(figsize=(8,3))
```

Добавить сетку, с толщиной 0.5 и стилем пунктир.

```
plt.grid(lw=0.5, ls='--')
```

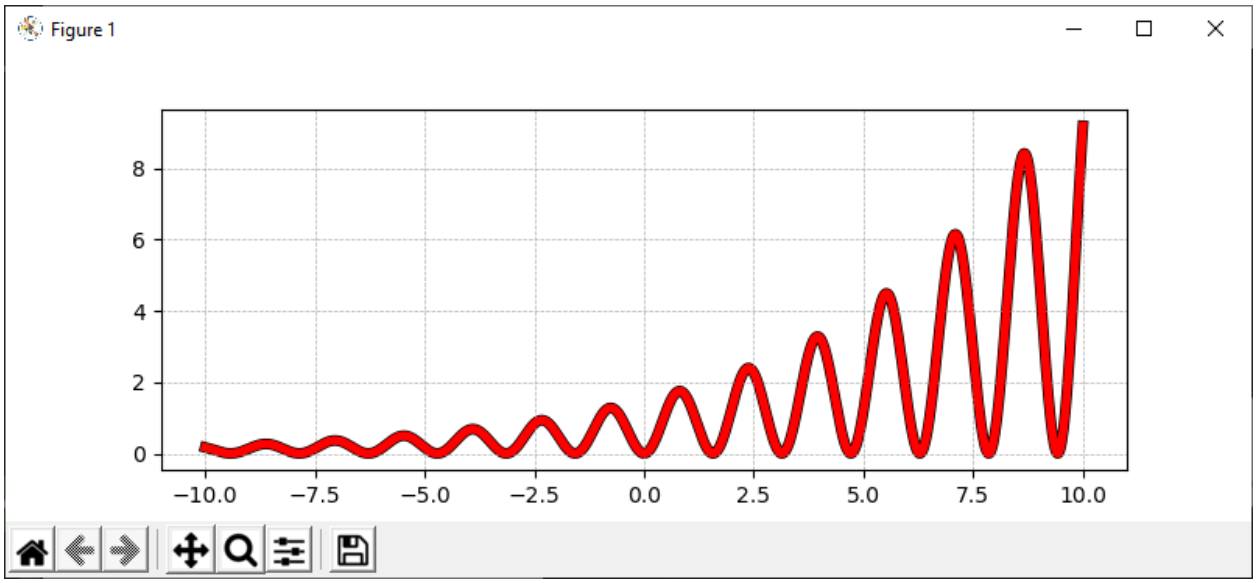
Изменить свойства графика.

```
plt.plot(x,y, lw = 4.0, #толщина линии
         color='red' #выбор одного из "вшитых" цветов, которые можно
подзывать по имени
         #color =(0.1, 0.9, 0.5, 0.95) #Задание цвета в формате rgba
         #color='#fc03d7' #задание цвета в формате hex
         )
```

Разместить графические объекты на различных геометрических слоях. Создадим «подложку» для кривой.

```
plt.plot(x,y, lw = 5.0, color='black', zorder=0)
```

Итог:



2.2 Диаграммы разброса (scatter plot)

Построить функцию синуса x на диапазоне [-3*pi, 3*pi], 250 значений с некоторым зашумлением (нормальный закон распределения, математическое ожидание 1, СКО 4, - умноженное затем в 20 раз).

Построить график разброса

```
plt.figure(figsize=(10, 5))
plt.scatter(x, y)
```

Изменить величину элементов, тип и цвет маркера, толщину и цвет границы, и задать орнамент маркеров

```
plt.scatter(x, y,
            s=300,
            marker='s',
            c='violet',
            lw=2,
            edgecolor='black',
            hatch='**'
            )
```

Задать заголовки и подписи к осям, шрифты

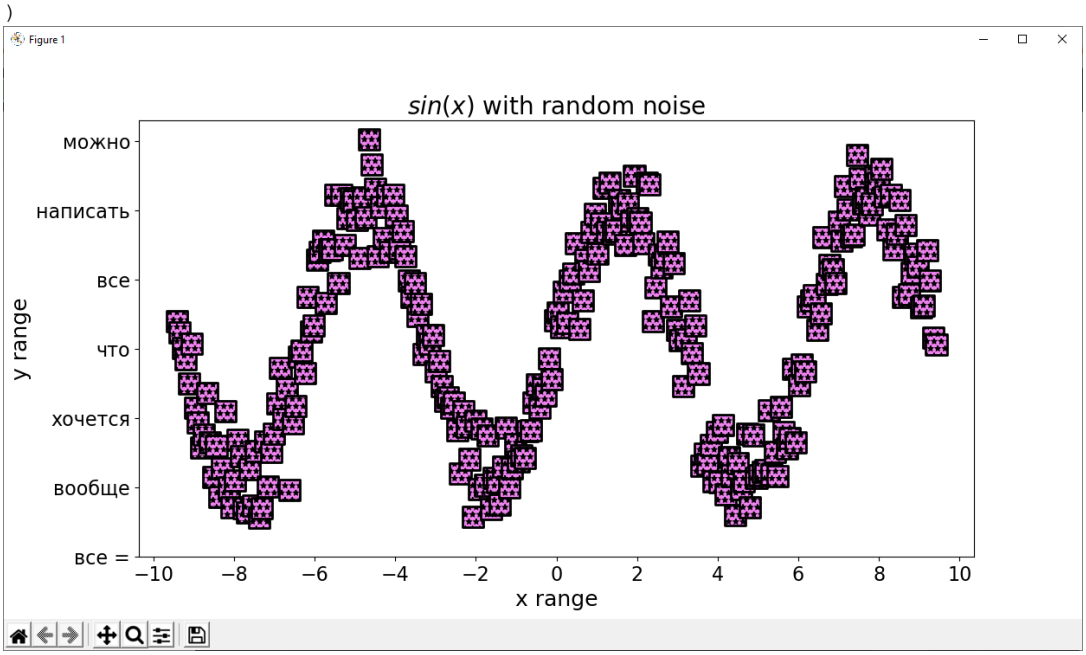
```
plt.title(
    label='$sin(x)$ with random noise', # Заголовок
    fontsize=20                         # Размер шрифта
)

plt.xlabel('x range', fontsize=18)
plt.ylabel('y range', fontsize=18)
plt.tick_params(labelsize=16)
```

Поставить свои шкалы для осей

```
plt.xticks(
    ticks=np.arange(-10, 11, 2) # Нужные значения по оси x
)
plt.yticks(
    ticks=np.arange(-1.5, 2,0.5),   # Значения по оси y будут заменены на подкписи,
                                    # Которые будут на этих позициях
```

```
    labels=['можно', 'написать', 'все', 'что', 'хочется', 'вообще', 'все ='][::-1]
)
```



2.2 Гистограммы (histogram)

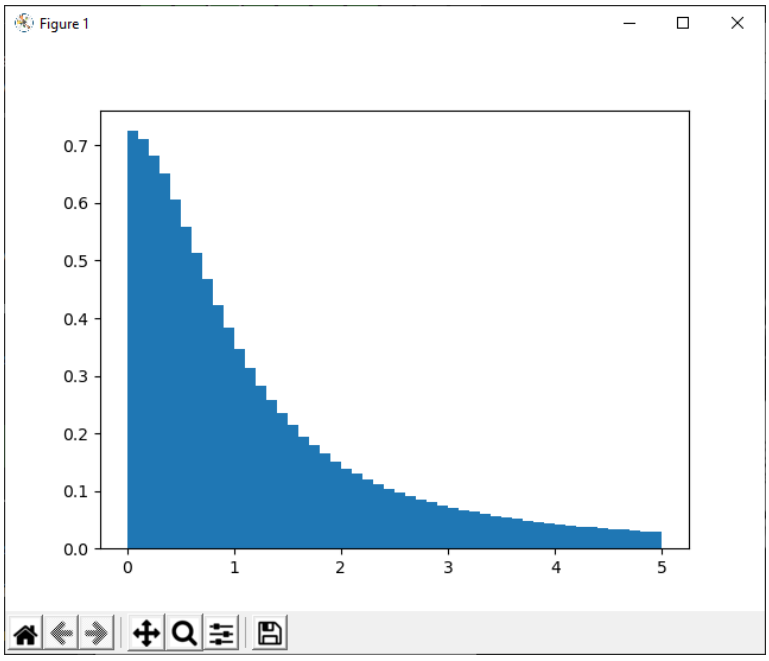
Сгенерировать данные y по распределению Коши (standatd_cauchy) – количество 10^7.

Постоить гистограмму

```
plt.hist(y)
```

Задать диапазон (0, 5), количество прямоугольников и пронормировать гистограмму

```
plt.hist(y,
         range=(0, 5),
         bins=50,
         density=True
         )
```



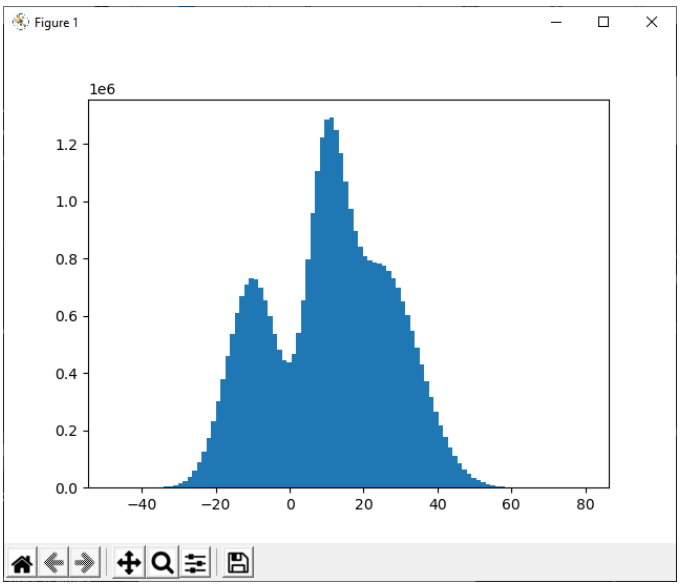
Пример 2.

Сгенерировать данные, смешав между собой три нормальных распредления с разными данными. Задать x равномерно в диапазоне (-5pi, 5pi), выборка 10^7. С использованием normal фукнции сгенерировать y1, y2, y3 с параметрами мат. ожидания -10, 10 и 25 и СКО 7, 5, 10. Для y3 задать размер в 1,5 раза больше x.

Смешение законов

```
y = np.concatenate([y1, y2, y3])
```

Построить гистограмму, число столбцов 100.



2.3 Круговые графики

1. График в полярных координатах

150 значений, полярный радиус *r* изменяется в диапазоне от 0 до 2, равномерный закон распределения. Полярный угол *theta* по равномерному закону от 0 до 360 градусов.

```
plt.figure(figsize=(6, 6))
plt.axes(projection='polar')
plt.scatter(theta, r)
```

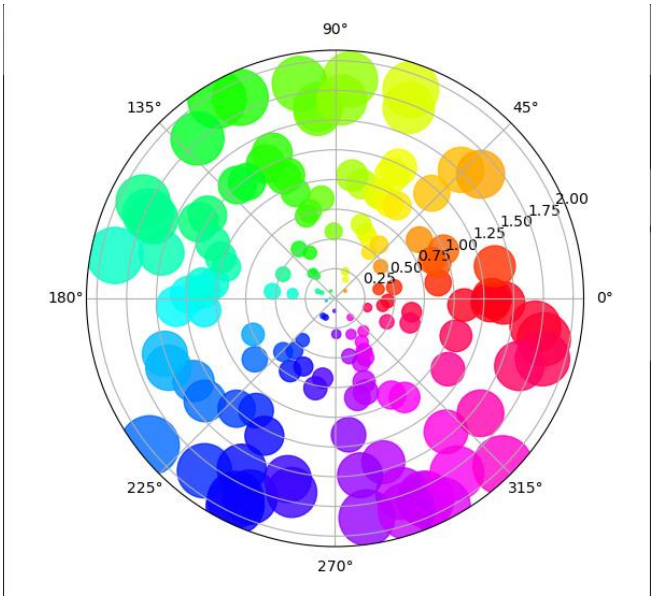
Размер точек сделать переменным, в зависимости от радиуса.

Цвет точек переменный от полярного угла.

Поменять цветовую палитру (cmap).

Сделать точки прозрачными на 20%.

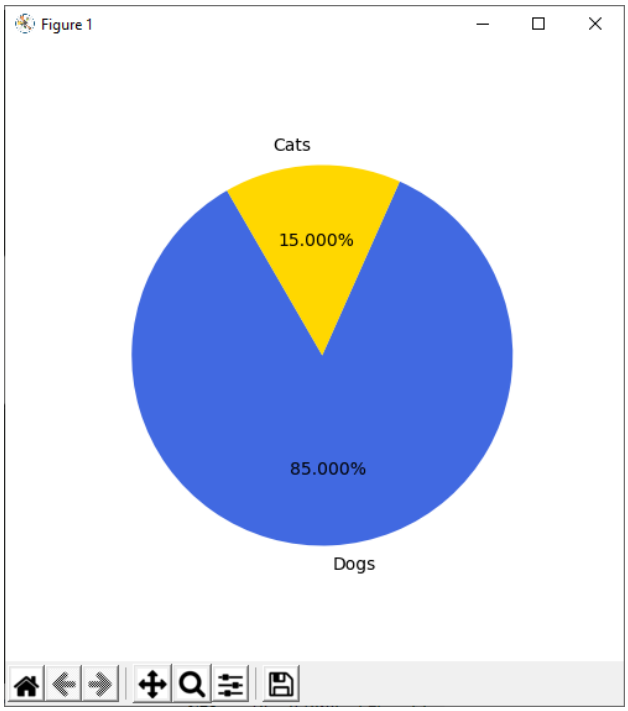
```
plt.scatter(theta, r,
            s=400*r**2,
            c=theta,
            cmap='hsv',
            alpha=0.8
            )
```



2. Круговая диаграмма

Задать круговую диаграмму для двух категорий в пропорции 17:3.

```
counts = [17, 3]
plt.figure(figsize=(5, 5))
plt.pie(counts,
        colors=['royalblue', 'gold'], #цвета долей
        labels=['Dogs', 'Cats'],       # подписи
        startangle=120,                # начальный угол
        autopct='%.3f%%'              # формат вывода значений долей
            )
```

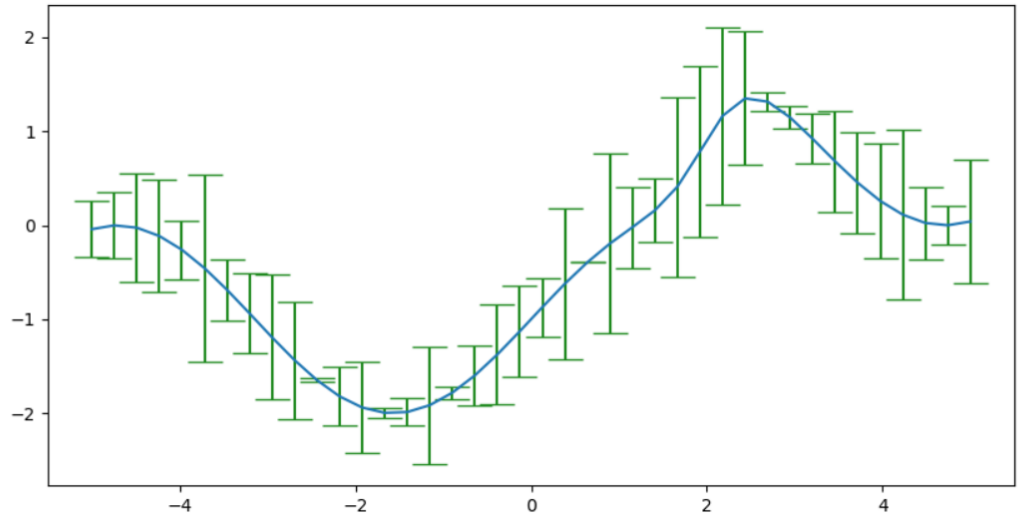


2.4 Графики с погрешностями

Создать массив x и массив значений функции y(x) по формуле $y(x) = \sin x + \tan(2 \cdot (x - 2))$ на интервале от -5 до 5, разбитый на 40 точек. Сгенерировать погрешности массив погрешностей yerr с использованием функции random.sample, от -1 до 1 (для каждой из 40 точек).

Построить график с погрешностями с помощью функции errorbar.

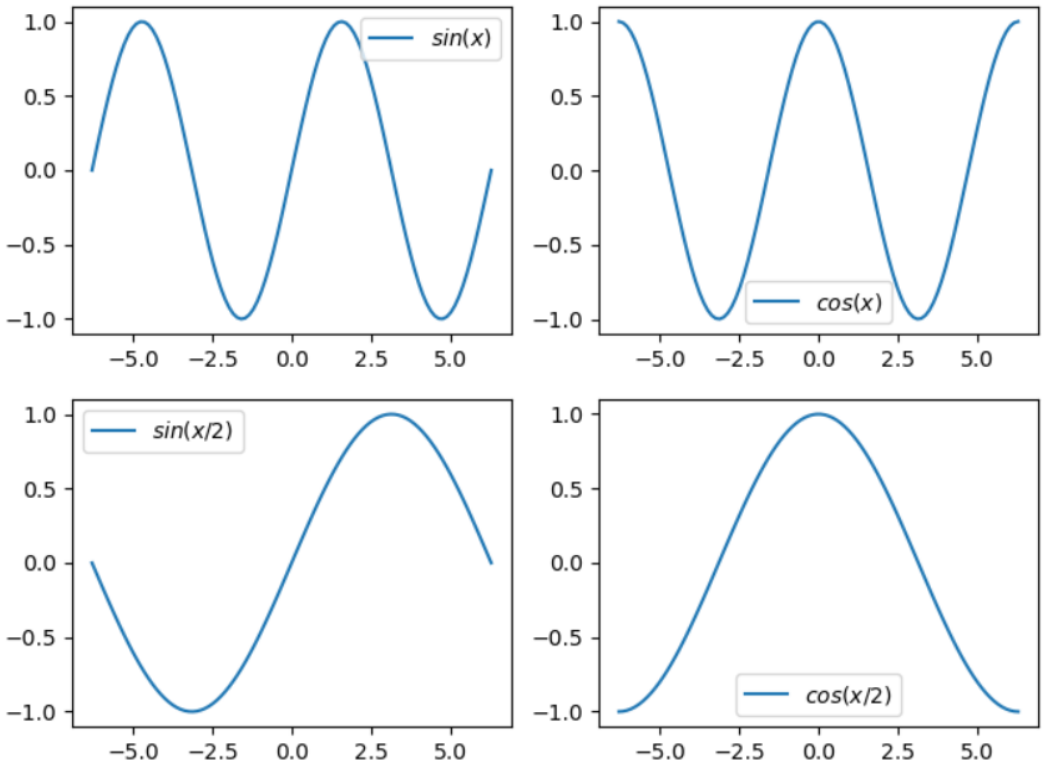
```
plt.figure(figsize=(10, 5))
plt.errorbar(x, y,
        yerr=yerr,
        ecolor='forestgreen', # цвет
        capsize=10,           # ширина
        elinewidth=1.5        # толщина
        )
```



2.5 Подграфики

Разбить полотно на 4 подграфика, 2x2. Аргумент во всех случаях изменяется в диапазоне -2*pi, 2*pi. Функции sin(x), sin(x/2), cos(x), cos(x/2). Сделать подписи к графикам.

```
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(8, 6))
```



2.6 Плотностные и трехмерные графики

1. Плотностные графики

Необходимо нарисовать график фукнции $f(x,y) = (\mathrm{x}-1)^2 + (\mathrm{y}+2)^2$.

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

x = np.linspace(-20, 20, 100)
y = np.linspace(-20, 20, 100)
```

Необходимо использовать фукнцию *meshgrid*, создающую прямоугольную сетку на основе значений векторов x и y. С помощью фунции *func* создается матрица координат Z.

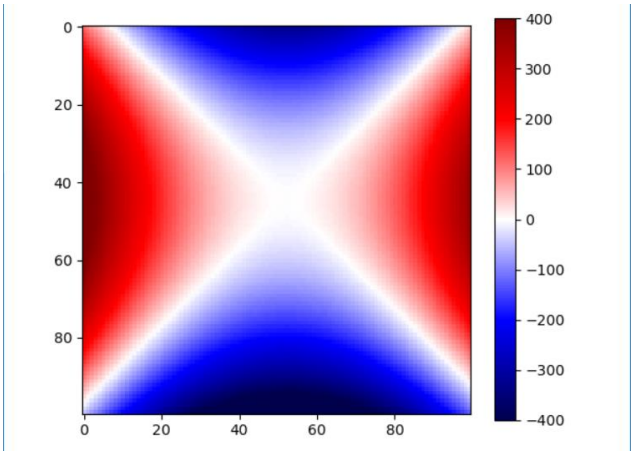
```
X, Y = np.meshgrid(x, y)
Z = func(X, Y) #Матрица z
```

Для отображения проекции трехмерной фигуры используется функция *imshow*.

```
fig, ax = plt.subplots(1, 1, figsize=(6, 5))
obj = ax.imshow(Z, cmap=plt.cm.seismic, vmin=-400, vmax=400)
```

Для просмотра координат z используется цветовая шкала

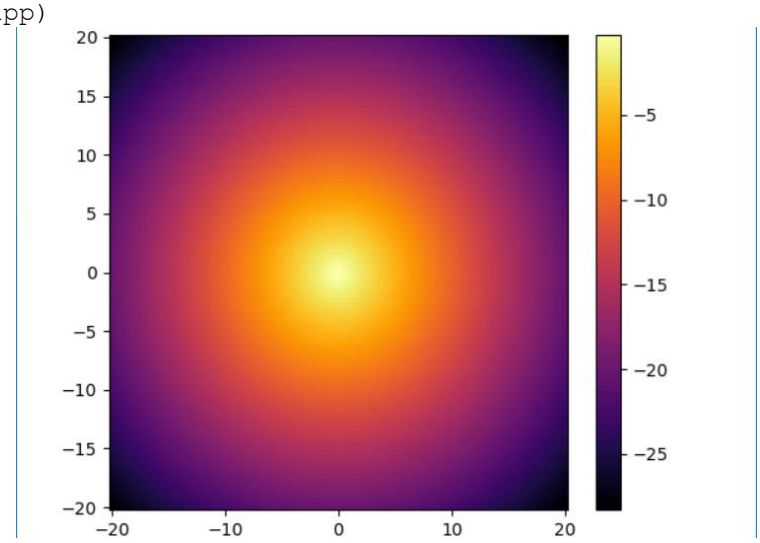
```
fig.colorbar(obj, vmin=-400, vmax=400)
```



Для рисования изображения произвольной сетки существует фукнция *pcolormesh*.

Нарисуем график для фукнции $r = -\sqrt{x^2 + y^2}$. Задайте для вычисления значений r функцию func2. X, Y берем из предыдущего примера, вычисляем матрицу I.

```
fig, ax = plt.subplots(1, 1, figsize=(6, 5))
mapp = ax.pcolormesh(X, Y, I, cmap='inferno', shading='auto',
edgecolor='face')
fig.colorbar(mapp)
```



2. 3D графики

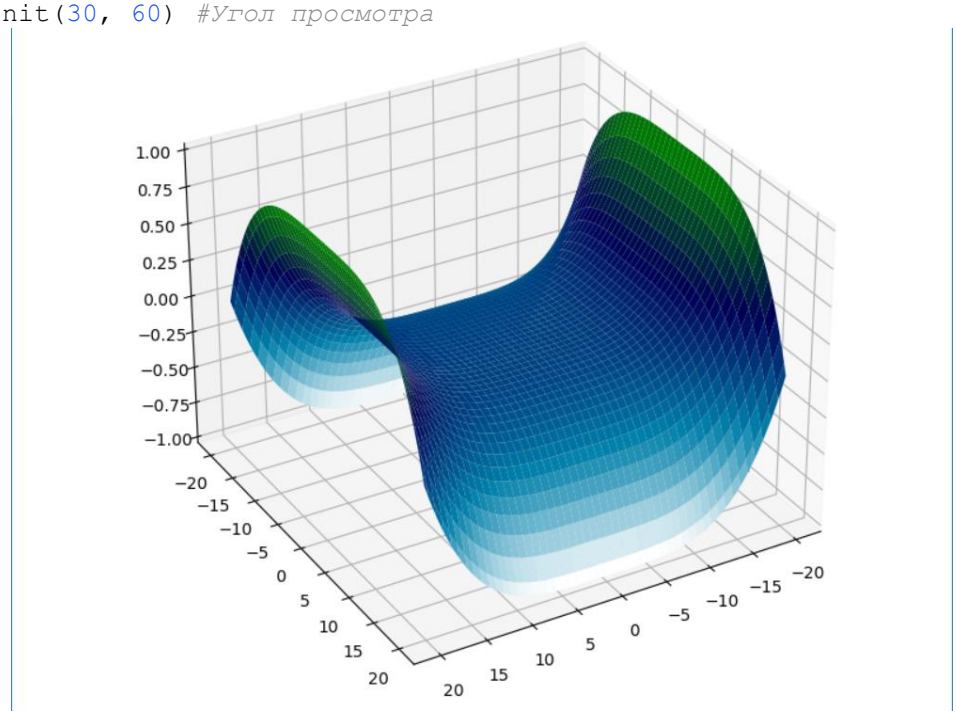
Задается атрибут *subplot_kw*.

```
fig, ax = plt.subplots(1, 1, figsize=(8, 8), subplot_kw={'projection': "3d"})
```

Задайте функцию $z = x^4 - y^4$ для сетки, рассмотренной ранее.

Построение графика осуществляется с использованием функции *plot_surface*.

```
ax.plot_surface(X, Y, Z / Z.max(), cmap=plt.cm.ocean_r)
ax.view_init(30, 60) #Угол просмотра
```



Задание 9.

Постройте двумерный график функции и визуально определите положение ее минимумов (с точностью до десятых).

$$f(x,y) = \frac{1}{4} sin\left(\frac{1}{2}x^2 - y\right) \quad - e^{-((x-5)^2+(y-2)^2)}$$

$$x \in [2,8]$$

$$y \in [0,5]$$

Примечание I. Для создания сетки координат используйте функцию `np.meshgrid`, которая создает прямоугольную сетку из двух одномерных массивов.

Задание 10.

Построить поверхность

$$f(x,y) = \frac{\sin(x^2 + y^2)}{x^2 + y^2}$$

Задание 11.

Построить поверхность

$$\begin{cases} \mathrm{x} = (R + r \cdot \cos\varphi) \cdot \cos\theta\,, \\ \mathrm{y} = (R + r \cdot \sin\varphi) \cdot \sin\theta, R > r > 0 \\ z = r \cdot \sin\varphi\,. \end{cases}$$

3. SciPy

3.1 Решение простейших дифференциальных уравнений

Задание 12.

Используется функция *odeint*:

```
from scipy.integrate import odeint
```

Функция решает дифференциальное уравнение вида:

$$\begin{cases} \frac{dy}{dx} = f(y, t, \dots) \\ y(t = t_0) = y_0 \end{cases}$$

Соответственно, для функции *odeint* нужно задать три обязательных аргумента: производную функции y по аргументу t (функцию f), начальное условие y_0 для функции y $(t = t_0)$ и определить точки t для которых будет найдено решение (причем первый элемент массива t равен t_0).

Решим простое дифференциальное уравнение первого порядка:

$\frac{dy}{dt} = t^2$.

Функцию нужно написать в формате, подходящем для *odeint*, поэтому здесь указаны две переменных: y и t.

```
func = lambda y, t: t**2
```

Далее зададим массив t

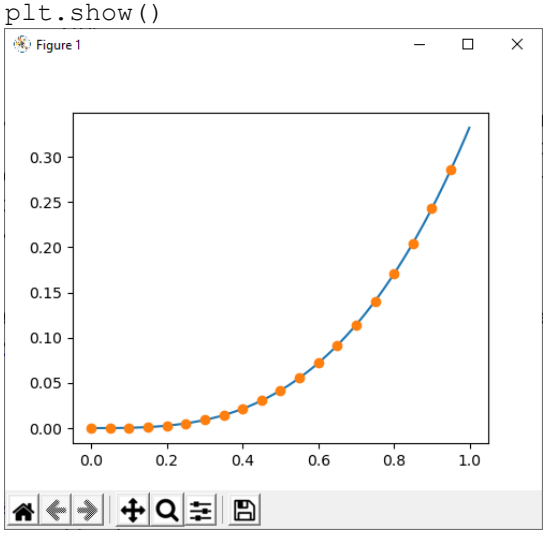
```
dt=1e-3
t = np.arange(0,1,dt)
```

И передадим все аргументы в функцию *odeint*, решение будет записано в *res*:

```
res = odeint(func, y0 = 0, t = t)
```

Решение уравнения $\frac{dy}{dt} = t^2$ есть $y(t) = \frac{t^3}{3} + C$, где C – некоторая константа, которая в данном случае равна нулю из-за начального условия.

```
plt.figure(figsize=(5,4))
plt.plot(t, res)
plt.plot(t[::50], t[::50]**3/3, 'o')
plt.show()
```



Задание 13.

Решим теперь дифференциальное уравнение второго порядка:

$$\frac{d^2f}{dt^2} - 2\frac{df}{dt} + f = 0.$$

Функция *odeint* не способна решать уравнения второго порядка. Однако, при помощи замены переменных это уравнение можно свести к системе из двух ОДУ:

$$\begin{cases} \dfrac{df}{dt} = u \\ \dfrac{du}{dt} = 2\dfrac{df}{dt} - f = 2u - f \end{cases}$$

Так как теперь у нас система из двух уравнений, на вход *odeint* должен подаваться вектор из двух компонент: $\frac{dy}{dt} = \left[\frac{du}{dt}, \frac{df}{dt}\right]$, где $y = [u, f]$. Следующая функция возвращает $\frac{dy}{dt}$:

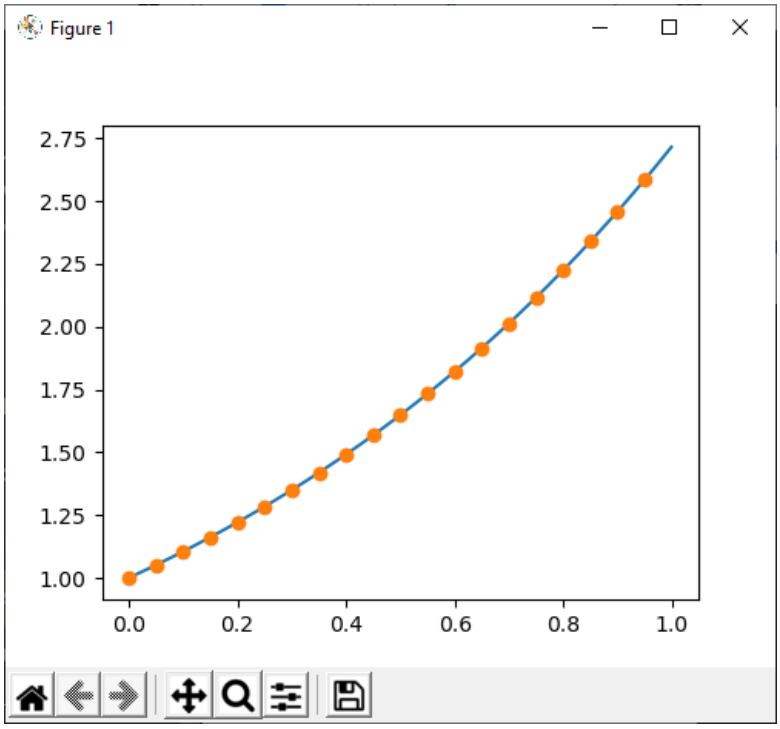
```
def func(y,t):
    u, f = y
    dydt = [2 * u - f, u]
    return dydt
```

Для системы из двух уравнений нужны два начальных условия: на фукнцию и ее производную. Выберем их в виде $f(0) = 1$ и $f'(0) = 1$.

```
res = odeint(func, y0 = [1,1], t = t)
```

Получается двумерный массив (из предыдущего примера для t 1000x2). В первом столбце для переменной u, во втором – для f. Так как нам необходимо решение для f, берем только значения второго столбца. Решение - экспонента

```
plt.plot(t, res[:, 1])
plt.plot(t[::50], np.exp(t[::50]), 'o')
```



Задание 14.

Напишите код, который возвращает значение $f(t = 1)$ с точностью до второго знака, где f - решение дифференциального уравнения $\frac{df}{dt} = f + t$ с заданным начальным условием $f(0) = f_0$

Sample Input:

```
1
```

Sample Output:

```
3.44
```

```
from scipy.integrate import odeint
import numpy as np
f0 = float(input())
dt = 0.001
t = np.arange(0,2,dt)
#ваш код здесь
```

3.2 Алгоритмы минимизации

Для описания физических явлений ученые-теоретики строят математические модели, которые должны с хорошей точностью описывать изучаемую систему. Математическая модель представляет собой функцию, зависящую от нескольких параметров, описывающих поведение изучаемой системы (количество параметров может варьироваться от одного до нескольких тысяч в зависимости от задачи, но в любом случае оно намного меньше, чем число факторов, влияющих на поведение реальной физической системы). Примером такой модели может служить закон Гука (wiki), в которой коэффициент упругости является таким параметром и имеет свое значение для каждого вещества. Главной проблемой при построении математических моделей является подбор таких значений параметров, при которых модель лучше всего описывает изучаемую систему. Обычно о поведении физической системы известно из некоторого набора экспериментальных данных. Значения параметров в математической модели нужно подобрать так, чтобы модель максимально хорошо воспроизводила эти данные. Такая задача называется задачей минимизации.

В этом разделе мы научимся численно решать задачи минимизации инструментами библиотеки `SciPy`. Мы рассмотрим основные алгоритмы, используемые для решения таких задач, рассмотрим их преимущества и недостатки. В заключение раздела мы решим задачу о распространении света в плотной среде и увидим, как при определенных условиях свет может огибать препятствия.

Поиск минимума одномерной функции

Задание 15.

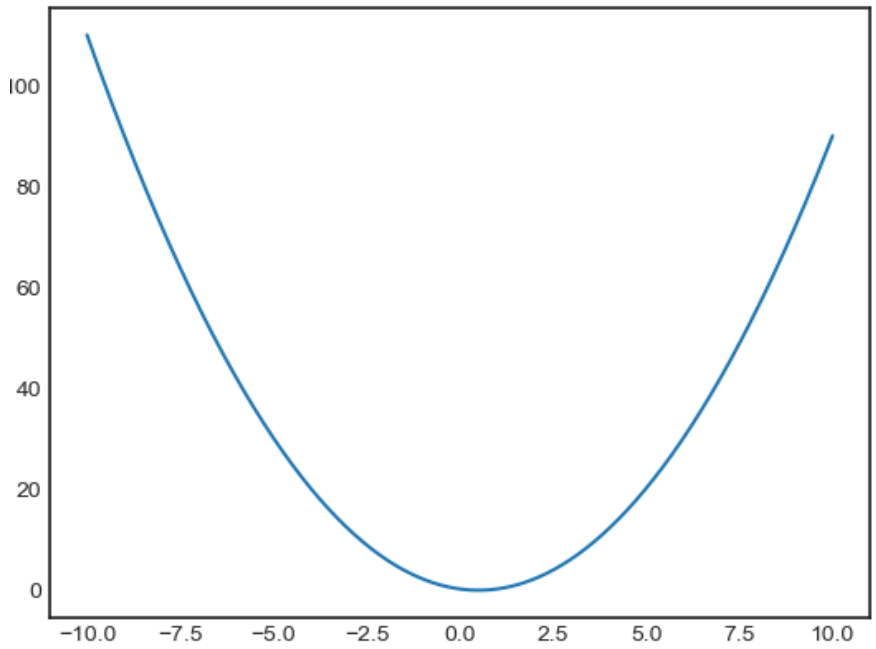
Для начала рассмотрим пример задачи минимизации для одномерной функции вида $f(x) = (x - 0.5)^2, x \in [-10,10]$.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import warnings

mpl.style.use('seaborn-white')
warnings.filterwarnings('ignore')

dx = 0.001
x = np.arange(-10, 10, dx)
f0 = lambda x: (x - 0.5)**2

plt.plot(x, f0(x))
plt.show()
```



По виду функции сразу можно сказать, что минимум будет в точке x=0.5, что и видно на графике.

Теперь найдем положение минимума при помощи функции *minimize* из библиотеки *scipy*. Первым аргументом для функции *minimize* является функция, которую нужно минимизировать, вторым - начальная точка, с которой будет стартовать алгоритм минимизации.

```
from scipy.optimize import minimize
result = minimize(f0, x0=1)
```

Посмотрим на выдачу результата минимизации. Важными параметрами для нас являются *fun*, который показывает значение минимизируемой функции в точке минимума; *success* - булевый параметр, который показывает, сошелся алгоритм или нет; x - самый важный параметр, который показывает ответ - точку минимума.

```
print(result)
```

```
     fun: 1.3877787807814457e-17
 hess_inv: array([[1]])
     jac: array([2.23517418e-08])
 message: 'Optimization terminated successfully.'
    nfev: 6
     nit: 1
```

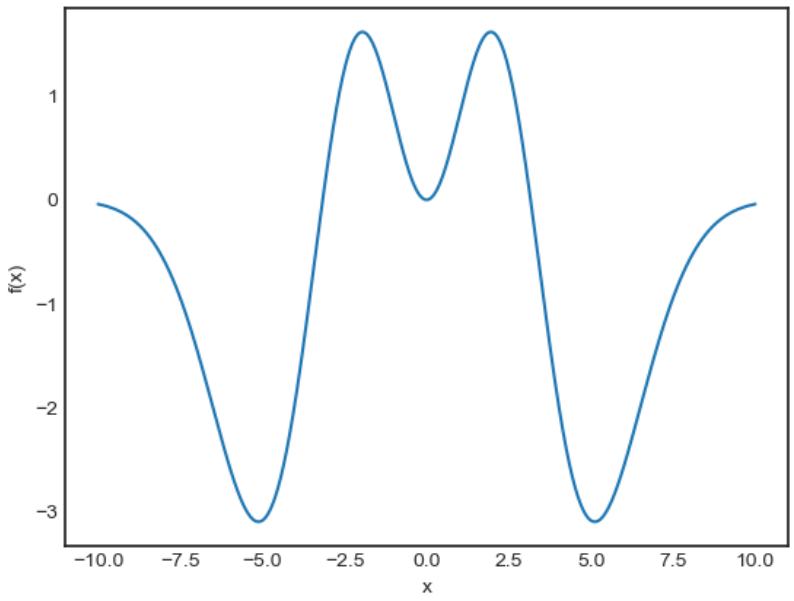
njev: 3
status: 0
success: True
x: array([0.5])

Задание 16.

Рассмотрим теперь более сложный пример одномерной функции с несколькими экстремумами: $f(x) = x^2 \cdot (1 - 0.1x^2) \cdot e^{-0.1x^2}$.

```
f = lambda x: x**2 * ( 1 - 0.1 * (x) **2 )* np.exp(- 0.1 * (x)**2)

fig, ax = plt.subplots()
ax.set(xlabel='x', ylabel='f(x)')
ax.plot(x, f(x))
plt.show()
```



Поиск минимума в этом случае будет более сложным, поэтому будет интересно отследить, как алгоритм будет перемещаться по оси *x*. Для этого определим функцию *get_path*. В этой лекции подробно останавливаться на описании работы этой функции мы не будем, нам лишь важно знать, что она будет записывать в список *path* координаты всех пройденных точек.

```
def get_path(xc):
    global path
    path.append(xc)
```

Выберем начальную точку и точность расчета *tol* и запустим процесс минимизации:

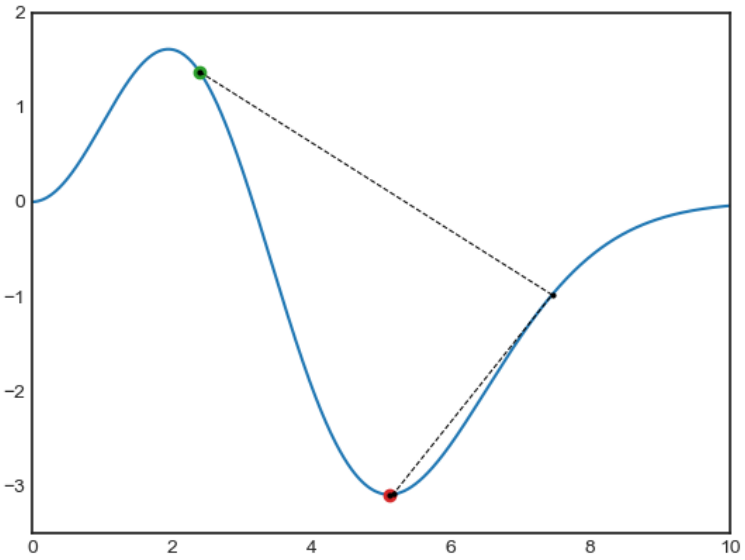
```
x0 = 2.4
path = [x0]
result = minimize(f, x0=x0, tol=1e-2, callback=get_path)
x1 = result.x
print(result)
```

fun: -3.090047003364168
hess_inv: array([[2.91066268]])
jac: array([0.00171024])
message: 'Optimization terminated successfully.'
nfev: 12

nit: 3
njev: 6
status: 0
success: True
x: array([5.11767426])

Отобразим на графике все точки, содержащиеся в *path*. Начальная точка показана зеленым, конечная – красным, а все промежуточные – черным:

```
plt.scatter([x0], [f(x0)], color = 'tab:green')
plt.plot(path, [f(i) for i in path], '--o', color="black", lw=0.75,
markersize=2)
plt.scatter([x1], [f(x1)], color = 'tab:red')
plt.plot(x, f(x), zorder = 0)
plt.xlim(0, 10)
plt.ylim(-3.5, 2)
plt.show()
```



Главный недостаток алгоритмов минимизации заключается в том, что вместо глобального минимума алгоритм может найти локальный. В нашем примере если начальная точка x_0 выбрана неудачно, алгоритм найдет лишь локальный минимум.

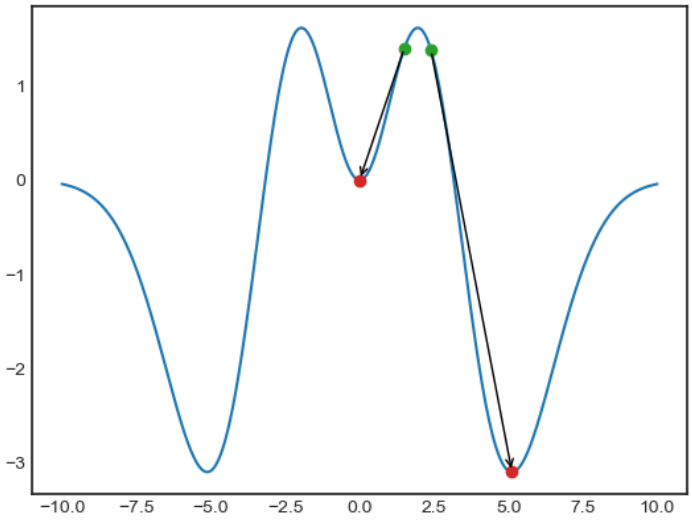
```
result = minimize(f, x0=1.5)
x2 = result.x
print(result)
```

fun: 1.4145952559875395e-12
hess_inv: array([[0.50122726]])
jac: array([-2.36383401e-06])
message: 'Optimization terminated successfully.'
nfev: 14
nit: 3
njev: 7
status: 0
success: True

x: array([-1.18936759e-06])

На следующем графике видно, как ответ зависит от задания начального значения x_0:

```
plt.annotate("", xytext=(2.4, f(2.4)), xy=(x1, f(x1)),
arrowprops=dict(arrowstyle="->"))
plt.annotate("", xytext=(1.5, f(1.5)), xy=(x2, f(x2)),
arrowprops=dict(arrowstyle="->"))
plt.scatter([x1, x2], [f(x1), f(x2)], color = 'tab:red')
plt.scatter([2.4, 1.5], [f(2.4), f(1.5)], color = 'tab:green')
plt.plot(x, f(x), zorder=0)
plt.show()
```



Все дело в том, что алгоритм, который мы использовали - градиентный. Существуют другие алгоритмы нахождения минимума, например, алгоритм дифференциальной эволюции, который относится к классу стохастических методов. Для нашего примера он сможет различить локальный и глобальный минимумы. В качестве аргументов для него подаются функция и границы, в которых ищется точка минимума.

```
from scipy.optimize import differential_evolution
result = differential_evolution(f, [(x.min(),x.max())])
print(result)
```

fun: array([-3.09004786])
jac: array([-4.44089213e-08])
message: 'Optimization terminated successfully.'
nfev: 143
nit: 8
success: True
x: array([-5.11667271])

Поиск минимума функции двух переменных

Задание 17.

В качестве примера найдем максимум Гауссиана:

$$f(x,y) = \frac{1}{2\pi\sigma_x\sigma_y}\exp\left(-\left(\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(x-x_0)^2}{2\sigma_y^2}\right)\right)$$

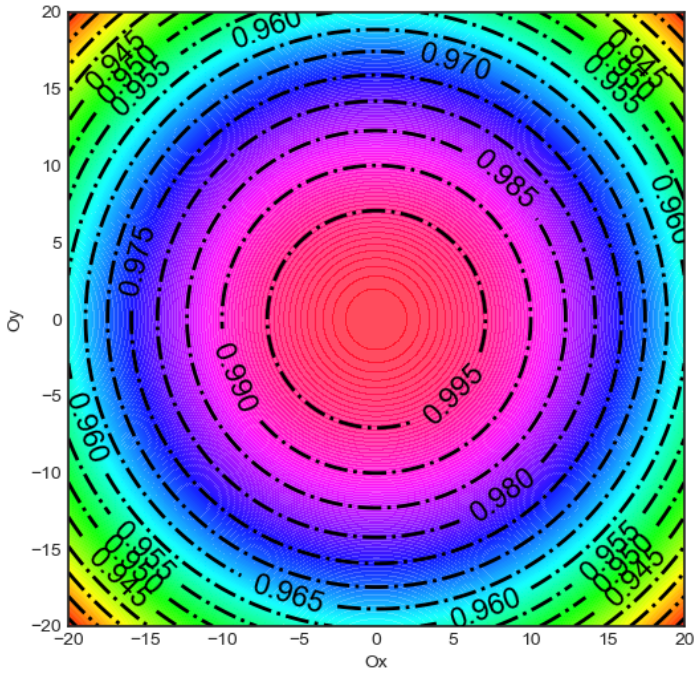
Для поиска максимума функции достаточно просто поставить перед функцией знак минус, и задача сведется к поиску минимума.

```
def gauss(z, sigma, x0, y0):
    x, y = z
    return np.exp(-((x-x0)**2 + (y-y0)**2) / sigma**2)

neg_gauss = lambda z, sigma, x0, y0: -gauss(z, sigma, x0, y0)

x = np.linspace(-20, 20, 100)
y = np.linspace(-20, 20, 100)
X, Y = np.meshgrid(x, y)
Z = gauss((X, Y), 100, 0, 0)

fig, ax = plt.subplots(1, 1, figsize=(6, 6))
contours = plt.contour(X, Y, Z, 15, colors="black", linewidths=2,
linestyles='-.')
ax.clabel(contours, inline = True, fontsize=16)
contours = plt.contourf(X, Y, Z, 200, cmap=plt.cm.hsv, alpha=0.7)
ax.set(xlabel="Ox", ylabel="Oy")
plt.show()
```




```
from scipy.optimize import minimize

a, b = 0, 0

def func(t, a, b):
    x, y = t[0], t[1]
    return (x + y)**2 - 2 * x * (y + a) - 2 * y * b + a + b

res = minimize(func, ((0, 0), ), args=(a, b))
print(" ".join(str(i) for i in res.x))
```

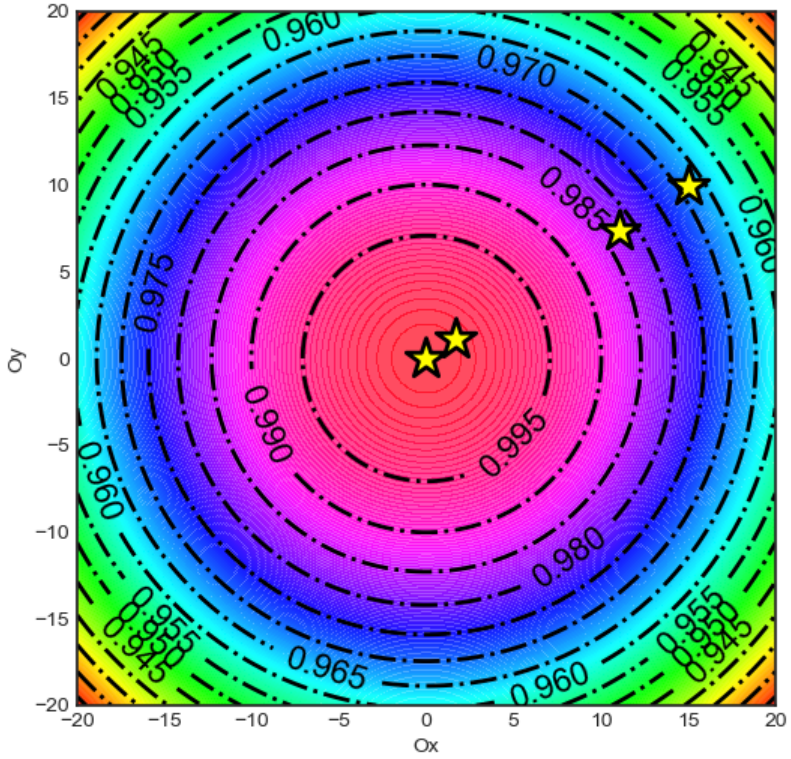
[] 0.0 0.0

```
path = [(15, 10)]
```

```
result = minimize(neg_gauss, (15, 10), args=(100, 0, 0), callback=get_path,)
path = np.array(path)
```

Как мы помним, обязательными аргументами функции *minimize* являются минимизируемая функция и начальная точка: в данном случае это *neg_gauss* и (15,10). Наша функция *neg_gauss* зависит от нескольких переменных, однако минимизировать ее мы хотим только по переменной *z*, поэтому все остальные параметры (это *sigmsa*, *x0*, *y0*) нужно передать в именованный аргумент *args* в виде кортежа. Путь алгоритма к точке минимума показан звездочками.

```
fig, ax = plt.subplots(1, 1, figsize=(6, 6))
contours = plt.contour(X, Y, Z, 15, colors="black", linewidths=2,
linestyles='-.')
ax.clabel(contours, inline = True, fontsize=16)
contours = plt.contourf(X, Y, Z, 200, cmap=plt.cm.hsv, alpha=0.7)
ax.scatter(path[:, 0], path[:, 1], s=400, c='yellow', marker='*', alpha=1,
edgecolor='black', linewidth=2)
ax.set(xlabel="Ox", ylabel="Oy")
plt.show()
```



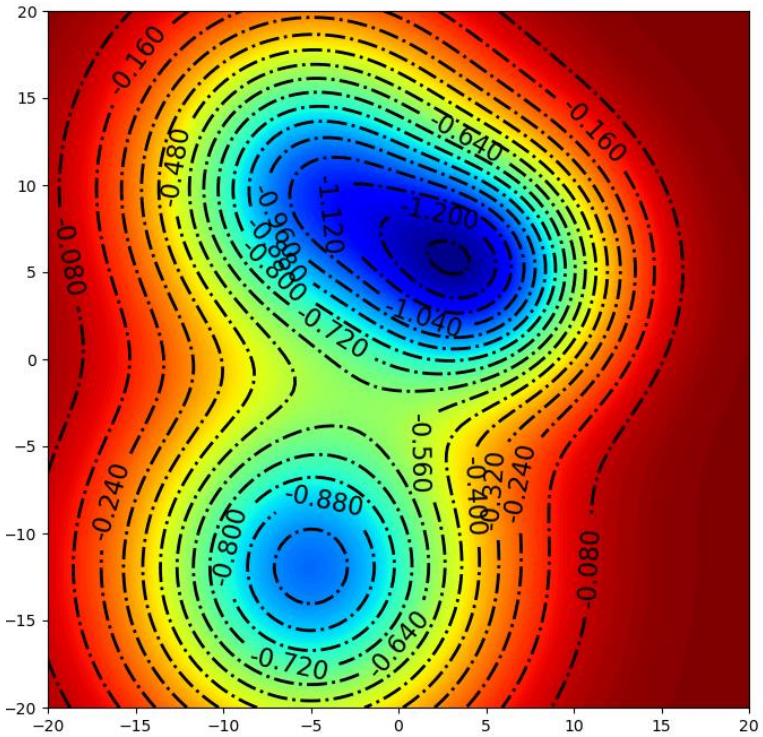
Задание 18.

Для более сложного примера рассмотрим функцию состоящую из суммы нескольких Гауссианов, с несколькими локальными минимумами.

```
def mixed(z, *args):
    return np.sum(neg_gauss(z, *params) for params in args)

Z = mixed((X, Y), (10, -5, -12), (7, 5, 5), (9, -5, 10))

fig, ax = plt.subplots(1, 1, figsize=(8, 8))
contours = ax.contour(X, Y, Z, 16, colors="black", linewidths=2,
linestyles='-.')
ax.clabel(contours, inline=True, fontsize=16)
contours = ax.contourf(X, Y, Z, 200, cmap=plt.cm.jet)
plt.show()
```



Из вида функции *mixed* видно, что *args* должен содержать набор кортежей (или списков), каждый из которых должен содержать три переменных - значения sigmsa, x0, y0 для каждого гауссиана. Количество таких наборов и определит число гауссианов, входящих в сумму для нашей функции. В нашем примере функция будет суммой трех гауссианов. Кроме того, для алгоритма дифференциальной эволюции мы здесь задали несколько дополнительных параметров. init - это набор точек, с которых стартует алгоритм (для данного алгоритма нужно задать по меньшей мере пять начальных точек), *recombination* - параметр, который находится в диапазоне от 0 до 1 и характеризует способность алгоритма выходить из локальных минимумов, *seed* - задает генерацию случайных чисел для возможности воспроизведения результатов на разных компьютерах.

```
# Gradient method
x0 = (-10, -2)
path = [x0]
result = minimize(mixed, x0,
                  args=((10, -5, -12), (7, 5, 5), (9, -5, 10)),
                  callback=get_path,
                 )

Z = mixed((X, Y), (10, -5, -12), (7, 5, 5), (9, -5, 10))
path = np.array(path)

fig, ax = plt.subplots(1, 2, figsize=(20, 10))
contours = ax[0].contour(X, Y, Z, 16, colors="black", linewidths=2,
linestyles='-.')
ax[0].clabel(contours, inline=True, fontsize=16)
contours = ax[0].contourf(X, Y, Z, 200, cmap=plt.cm.jet)

ax[0].scatter(path[:, 0], path[:, 1], s=1600, c='yellow',
              marker='*',
              alpha=1,
              edgecolor='black',
              linewidth=2,
              zorder=1
```

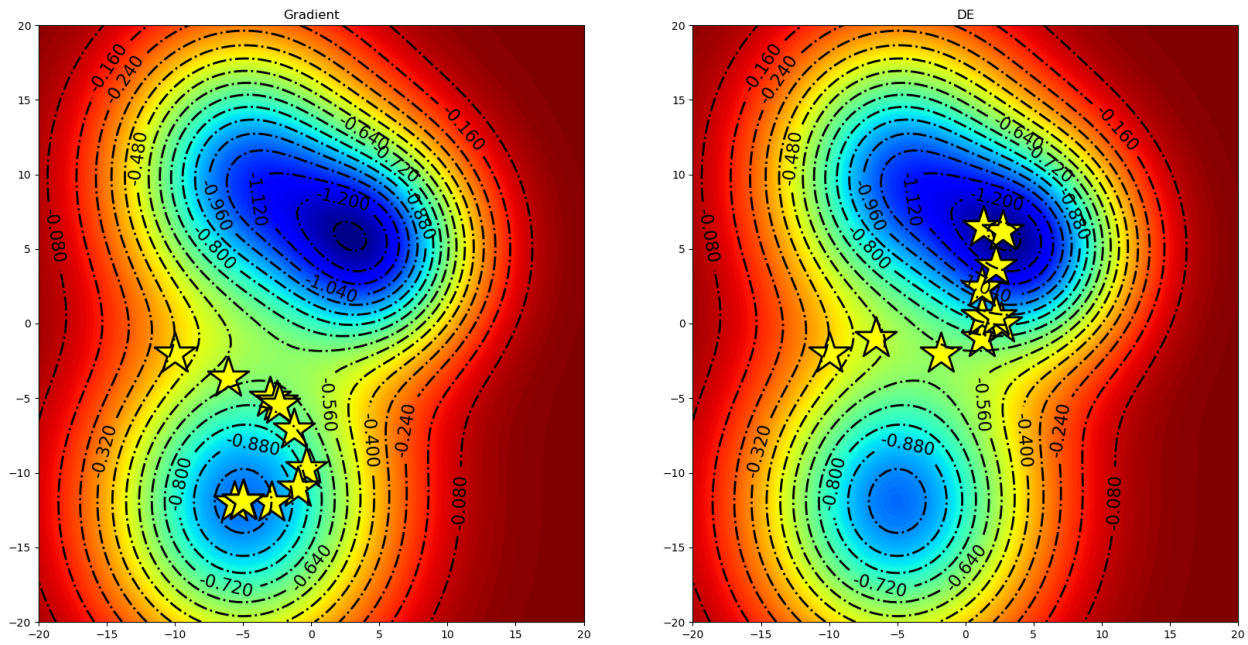
```
                )
ax[0].set_title('Gradient')


# Differential Evoluttion
path = [x0]

# Здесь нужно немного изменить функцию get_path для соответствования
требованиям алгоритма диффренциальной эволюции
def get_path(xc, convergence=0):
    global path
    path.append(xc)

result = differential_evolution(mixed, ((-20, 20), (-20, 20)),
                                init=np.array([x0, (-9, -1), (-11, -3), (-8,
0), (-12, -4)]),
                                args=((10, -5, -12), (7, 5, 5), (9, -5, 10)),
                                recombination=0.15,
                                seed=2,
                                callback=get_path
                               )
path = np.array(path)

contours = ax[1].contour(X, Y, Z, 16, colors="black", linewidths=2,
linestyles='-.')
ax[1].clabel(contours, inline=True, fontsize=16)
contours = ax[1].contourf(X, Y, Z, 200, cmap=plt.cm.jet)
ax[1].scatter(path[:, 0], path[:, 1], s=1600, c='yellow',
              marker='*',
              alpha=1,
              edgecolor='black',
              linewidth=2,
              zorder=1
             )
ax[1].set_title('DE')
plt.show()
```



На рисунке слева видно, что алгоритм minimize свалился в локальный минимум, а алгоритм дифференциальной эволюции (справа на рисунке) сумел правильно найти глобальный минимум.

Задание 19.

Напишите код, который при помощи ***minimize*** находит координату x локального максимума функции

$f(x) = x^2(x-4)^2e^{-x^2}$ вблизи заданного x_0.

Ответ округлите до второго знака после запятой

Sample Input:

```
1
```

Sample Output:

```
0.85
```

```
from scipy.optimize import minimize
import numpy as np
x0 = float(input()) #начальное предположение
dx = 0.001
x = np.arange(-4,4,dx)
#ваш код здесь
```

Задание 20.

Напишите код, который при помощи функции ***minimize*** находит координату (x,y) локального минимума вблизи точки $(0,0)$ функции $f(x,y) = (x+y)^2 - 2x(y+a) - 2yb + a + b$ для заднных a,b .

Sample Input:

```
0 0
```

Sample Output:

```
0.0 0.0
```

3.3 Преобразование Фурье

Одномерное Фурье-преобразование. Фильтрация сигнала

Преобразование Фурье - один из примеров *функционалов*, то есть это правило, по которому одной функции (например, $f(x)$) ставится в соответствие другая функция ($f(\omega)$). Функция $f(\omega)$ называется в этом случае *фурье-образом* функции $f(x)$. Общая формула для фурье-преобразования имеет вид

$$f(\omega) = \frac{1}{2\pi}\int_{-\infty}^{\infty} f(x)e^{i\omega x}dx$$

Преобразование Фурье используется во многих областях науки — в физике, оптике, акустике, статистике, криптографии, океанологии, обработке сигналов и изображений, геометрии и многих других.

В этом уроке мы будем использовать Фурье-преобразование для восстановления сигнала из зашумленных данных и для фильтрации двумерных изображений.

Задание 21.

Подключаем необходимые библиотеки

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

Смоделируем сигнал, состоящий из двух синусоид, искаженных случайным шумом:

$$f(t) = (\sin 2\pi\omega t + \sin 4\pi\omega t) + 2\Delta\ ,$$

где Δ – случайное число, равномерно распределенное на отрезке [−1,1].

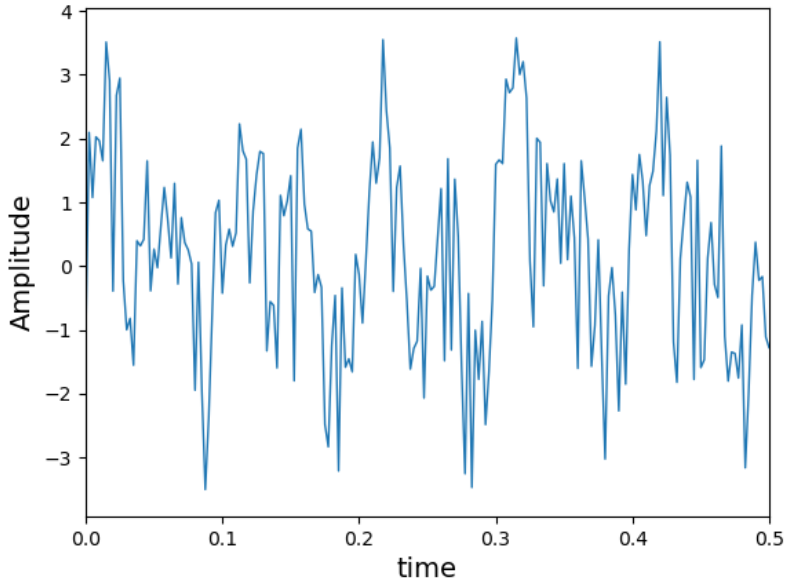
Задаем длительность сигнала (1 с) и интервал дискретизации (0.0025 с).

```
tmax = 1
dt = 0.0025
t = np.arange(0, tmax, dt)

freq = 10
wave = np.sin(2 * np.pi * freq * t) + np.sin(2 * np.pi * 2 * freq * t)
noise = 2 * (2 * np.random.sample(t.size) - 1)
```

В таком случае наш сигнал будет выглядеть следующим образом

```
plt.plot(t, wave + noise, lw = 1, color = 'tab:blue', label = 'noisy')
plt.xlim(0, 0.5)
plt.xlabel('time', fontsize = 14)
plt.ylabel('Amplitude', fontsize = 14)
plt.show()
```



При помощи преобразования Фурье можно избавиться от шумов и восстановить исходный сигнал, воспользовавшись функцией *fft* из модуль *numpy.fft*. Здесь *fft* – аббревиатура от fast Fourier transform – быстрое преобразование Фурье.

Функция *numpy.fft.fft* осуществляет одномерное преобразование Фурье

В начале давайте получим фурье представление нашего зашумленного сигнала, то есть

$$f(\omega) = \sum_{n=0}^{N} e^{-i\omega t_n} f(t_n)$$

$$e^{-i\omega t_n} = \cos(\omega t_n) - i\sin(\omega t_n)$$

Таким образом преобразованная функция представляет собой ряд из косинусов и синусов

$$f(\omega) = \sum_{n=0}^{N} f(t_n)(\cos(\omega t_n) - i\sin(\omega t_n))$$

```
signal = np.fft.fft(wave + noise, t.size)
```

Мы перевели наш временной сигнал в частотное пространство. Теперь получим сами частоты, для которых наш новых сигнал определен

```
freqs = np.fft.fftfreq(t.size, dt)
```

Мы хотим оценить вклад каждой из частот в разложении обратного Фурье-преобразования:

$$f(\omega) = \sum_{m=0}^{N} f(\omega_m)e^{-i\omega t_n} = \sum_{m=0}^{N} f(\omega_m)(\cos(\omega_n t) - i\sin(\omega_m t))$$

Из разложения видно, что вклад частоты ω_m в результирующий сигнал определяется Фурье-образом $f(\omega_m)$. Для анализа и визуализации частотного содержания сигнала используется амплитудный спектр:

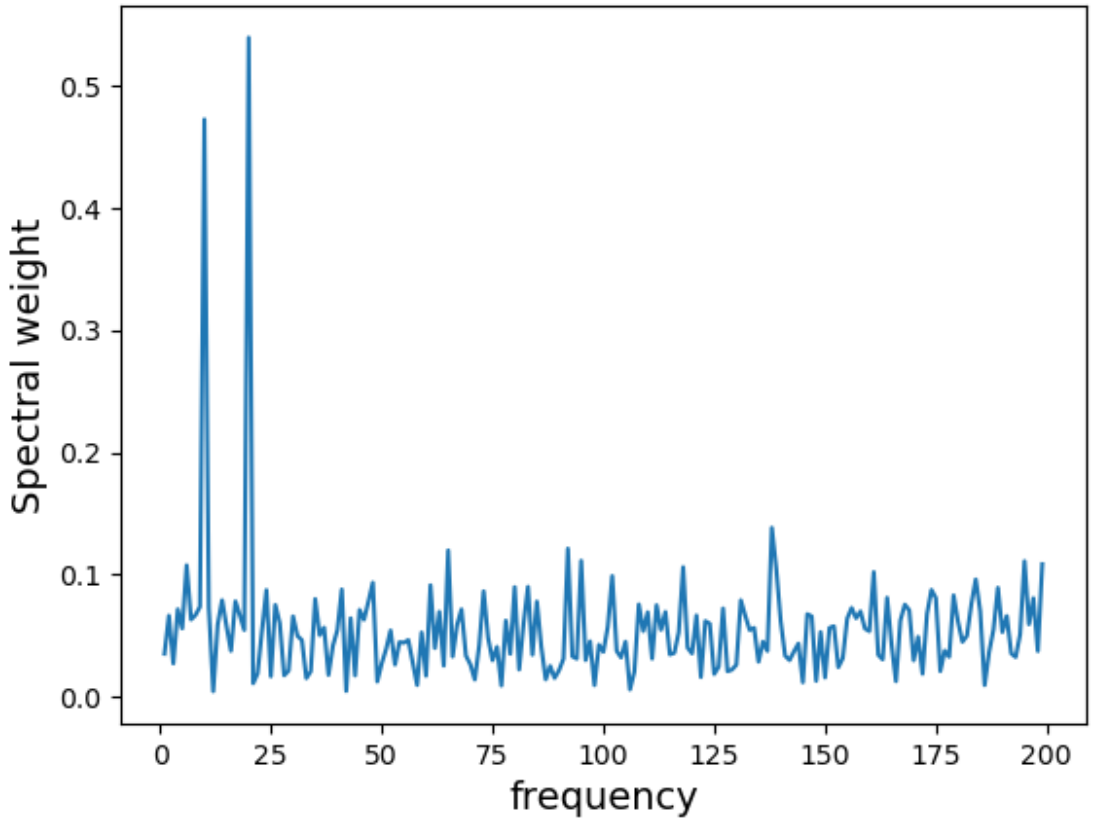
$$A_m = \sqrt{(Re[f(\omega_m)])^2 + (Im[f(\omega_m)])^2} = |f(\omega_m)|$$

С его помощью можно определить, какие частоты содержатся в сигнале и какие из них являются доминирующими. Это используется в различных задачах, таких как анализ музыки для приложений, связанных с обработкой звука, или анализ временных рядов в финансовой сфере.

```
Spectrum = np.abs(signal) / t.size
L = np.arange(1, t.size // 2, dtype = int)
```

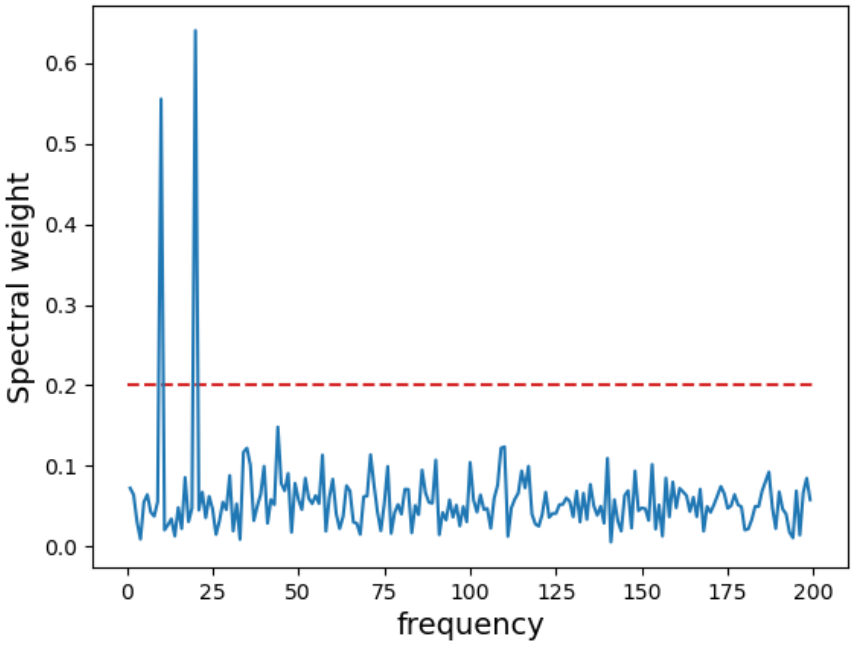
На спектральном графике видны два высоких пика. Они соответствуют частотам, присутствующим в чистом сигнале (ω=10 и ω=20). Все остальные пики – частотный спектр, возникший из-за зашумления сигнала:

```
plt.plot(freqs[L], Spectrum[L])
plt.xlabel('frequency', fontsize = 14)
plt.ylabel('Spectral weight', fontsize = 14)
plt.show()
```



Наша задача – отделить истинный спектр сигнала от шума. Для этого выберем границу спектрального веса, ниже которой весь сигнал мы будем считать шумом (то есть применим спектральный фильтр):

```
plt.plot(freqs[L], Spectrum[L])
plt.hlines(0.2, 0, 200, ls = '--', color = 'tab:red')
plt.xlabel('frequency', fontsize = 14)
plt.ylabel('Spectral weight', fontsize = 14)
plt.show()
```

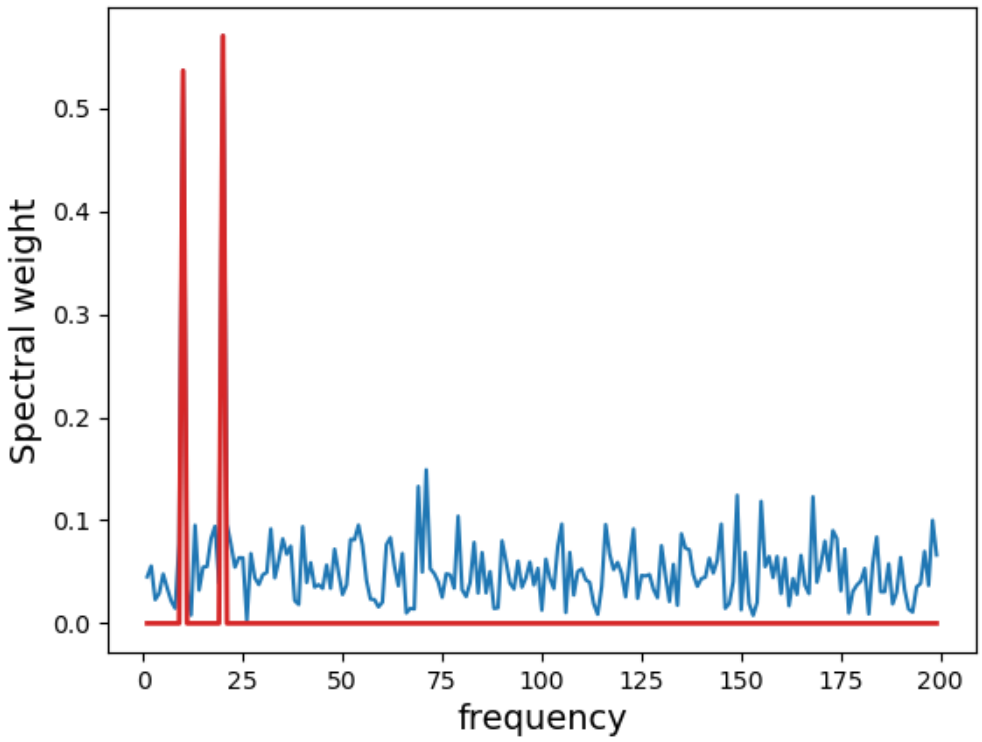


Удалим из спектра все частоты, спектральный вес которых меньше границы:

```
ind = Spectrum > 0.2
Spectrum_clean = Spectrum * ind
```

Отфильтрованный спектр показан красным цветом:

```
plt.plot(freqs[L], Spectrum[L])
plt.plot(freqs[L], Spectrum_clean[L], lw=2, color='tab:red')
plt.xlabel('frequency', fontsize=14)
plt.ylabel('Spectral weight', fontsize=14)
plt.show()
```



Давайте посмотрим на зашумленный и отфильтрованный амплитудные спектры. Мы видим, что фильтрация произведена правильно и в отфильтрованном амплитудном спектре содержатся лишь два доминирующих пика.

Теперь при помощи обратного преобразования Фурье

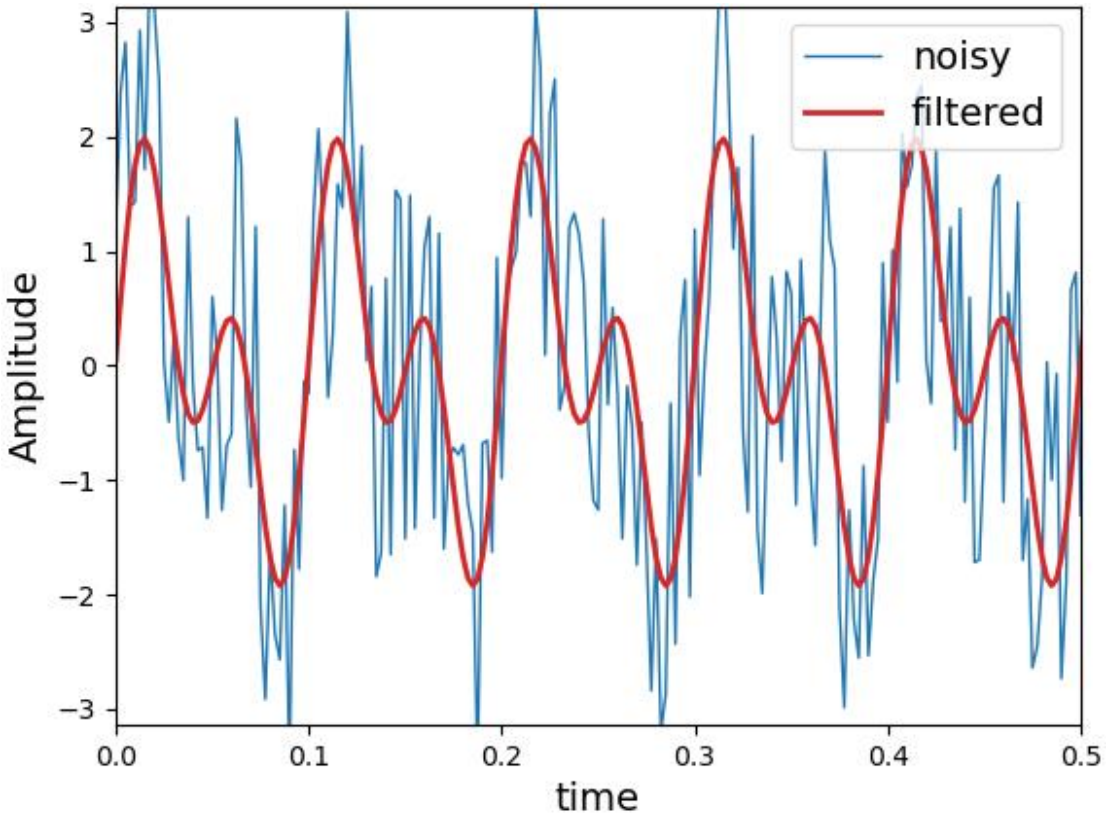
$$f(t) = \sum_{n=0}^{N} e^{i\omega t_n} f(\omega_m)$$

из отфильтрованного спектра восстанавливаем сигнал и убеждаемся, что он совпадает с исходным:

Такую же фильтрацию произведем над фурье-образом, для того чтобы получить отфильтрованный сигнал после обратного фурье преобразования, которое выполним при помощи функции ifft.

```
signal_clean = ind * signal
isignal = np.fft.ifft(signal_clean)
```

Посмотрим на наш изначальный зашумленный и отфильтрованный сигналы. Мы видим, что получившийся отфильтрованный сигнал не содержит в себе шумов и действительно представляет собой лишь сумму двух синусоид.



Двумерное Фурье-преобразование. Фильтрация изображения

Двумерное преобразование Фурье используется в самых различных областях науки и техники, а также для обработки изображений. Мы решим задачу аналогичную разобранной выше о восстановлении сигнала (изображения) путем фильтрации шума, но теперь в двумерном случае.

Задание 22.

Задаем пространственную сетку, на которой будет определен сигнал:

```
xmax, ymax = 1, 1
dx,dy = 0.01, 0.01
x,y = np.arange(0, xmax, dx), np.arange(0, ymax, dy)
```

В каждой точке {*x*,*y*} амплитуда сигнала определяется следующим выражением:

$$I(x,y) = (\sin 2\pi\omega_x x + \sin 4\pi\omega_x x)\big(\sin 2\pi\omega_y y + \sin 4\pi\omega_y y\big),$$

где

$$\omega_x = 10, \omega_y = 20.$$

Функция tensordot позволяет тензорно (т.е. каждый с каждым) перемножить элементы двух массивов и получить матрицу. На языке формул это означает следующее:

I = np.tensordot(x, y, axes = 0) эквивалентно $I_{ij} = x_i y_j$.

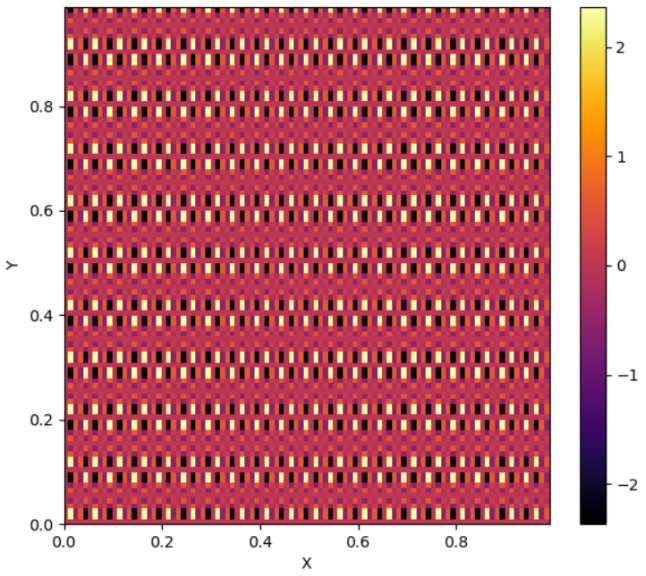
```
f0x = 10
f0y = 20


fx = np.sin(2 * np.pi * f0x * x) + np.sin(2 * np.pi * 2 * f0x * x)
fy = np.sin(2 * np.pi * f0y * y) + np.sin(2 * np.pi * 2 * f0y * y)

I = np.tensordot(fx, fy, axes = 0)
```

Построим изображение сигнала:

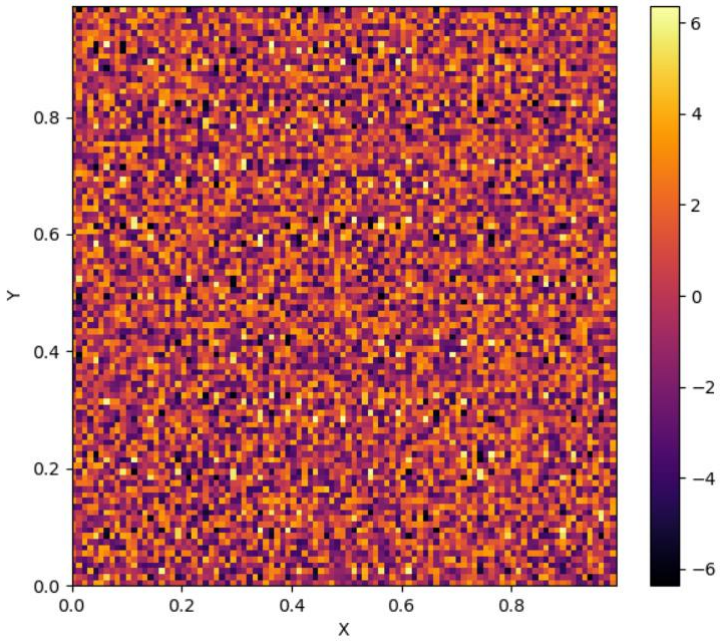
```
X,Y = np.meshgrid(x,y)
plt.figure(figsize = (7,6))
plt.pcolor(X, Y, I, cmap = 'inferno')
plt.xlabel('X')
plt.ylabel('Y')
plt.colorbar()
plt.show()
```



Теперь добавим к сигналу случайный шум $I_n = 4\Delta$, где Δ – число, равномерно распределённое на отрезке [−1,1].

```
noise = 4 * (2 * np.random.random_sample((x.size, y.size)) - 1)

plt.figure(figsize = (7,6))
plt.pcolor(X, Y, I + noise, cmap = 'inferno')
plt.xlabel('X')
plt.ylabel('Y')
plt.colorbar()
plt.show()
```



Тем не менее сигнал может быть восстановлен при помощи преобразования Фурье.

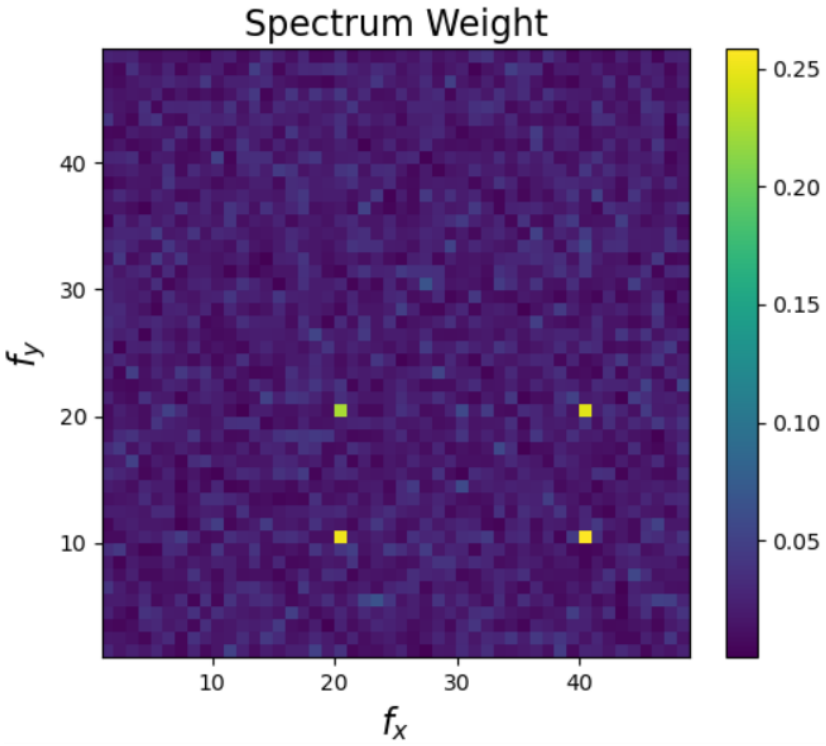
```
fI = np.fft.fft2(I + noise)
freqx = np.fft.fftfreq(x.size, dx)
freqy = np.fft.fftfreq(y.size, dy)

Lx = np.arange(1, np.floor(x.size/2), dtype = int)
Ly = np.arange(1, np.floor(x.size/2), dtype = int)

Spectrum = np.abs(fI) / x.size / y.size
Fx, Fy = np.meshgrid(freqx[Lx], freqy[Ly])
```

На рисунке отчетливо видны четыре пика, соответствующих частотам ω_x=10,20 и ω_y=20,40:

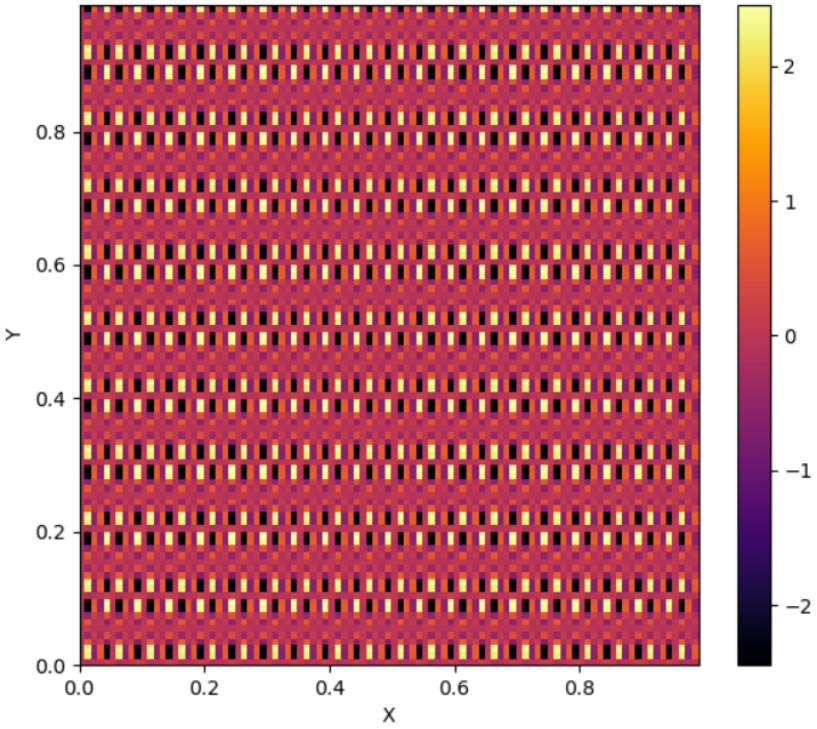
```
plt.figure(figsize = (6,5))
plt.title('Spectrum Weight', fontsize = 16)
plt.pcolor(Fx, Fy, Spectrum[Lx][:, Ly])
plt.xlabel('$f_x$', fontsize = 16)
plt.ylabel('$f_y$', fontsize = 16)
plt.colorbar()
plt.show()
```



Оставляем частоты, амплитудный спектр которых выше определенного значения. Фильтруем сигнал и восстанавливаем исходное изображение, используя обратное Фурье преобразование, которое в этот раз осуществляется при помощи функции ifft2:

```
ind = Spectrum > 0.1
I_filtered = np.fft.ifft2(fI * ind)

plt.figure(figsize = (7,6))
plt.pcolor(X, Y, I_filtered.real, cmap = 'inferno')
plt.xlabel('X')
plt.ylabel('Y')
plt.colorbar()
plt.show()
```



Задание 23.

Напишите код, который находит неотрицательную частоту с максимальным спектральным весом для функции $f(t)=sin(vcos(2\pi t))$ для заданного $v \in [1,10]$.

Sample Input:

6

Sample Output:

5.0