

Введение в анализ данных

Домашнее задание 1. Numpy, matplotlib, scipy.stats

Правила:

- Дедлайн **25 марта 23:59**. После дедлайна работы не принимаются кроме случаев наличия уважительной причины.
- Выполненную работу нужно отправить на почту `mipt.stats@yandex.ru`, указав тему письма " [номер группы] Фамилия Имя - Задание 1". Квадратные скобки обязательны.
- Прислать нужно ноутбук и его pdf-версию (без архивов). Названия файлов должны быть такими: `1.N.ipynb` и `1.N.pdf`, где `N` -- ваш номер из таблицы с оценками. *pdf-версию можно сделать с помощью Ctrl+P. Пожалуйста, посмотрите ее полностью перед отправкой. Если что-то существенное не напечатается в pdf, то баллы могут быть снижены.*
- Решения, размещенные на каких-либо интернет-ресурсах, не принимаются. Кроме того, публикация решения в открытом доступе может быть приравнена к предоставлению возможности списать.
- Для выполнения задания используйте этот ноутбук в качестве основы, ничего не удаляя из него.
- Пропущенные описания принимаемых аргументов дописать на русском.
- Если код будет не понятен проверяющему, оценка может быть снижена.

Баллы за задание:

Легкая часть (достаточно на "хор"):

- Задача 1.1 -- 3 балла
- Задача 1.2 -- 3 балла
- Задача 2 -- 3 балла

Сложная часть (необходимо на "отл"):

- Задача 1.3 -- 3 балла
- Задача 3.1 -- 3 балла
- Задача 3.2 -- 3 балла
- Задача 3.3 -- 3 балла
- Задача 4 -- 4 балла

Баллы за разные части суммируются отдельно, нормируются впоследствии также отдельно. Иначе говоря, 1 балл за легкую часть может быть не равен 1 баллу за сложную часть.

In [1]:

```
import numpy as np
import scipy.stats as sps

import matplotlib.pyplot as plt
import matplotlib.cm as cm
from mpl_toolkits.mplot3d import Axes3D
import ipywidgets as widgets

import typing

%matplotlib inline
```

Легкая часть: генерация

В этой части другие библиотеки использовать запрещено. Шаблоны кода ниже менять нельзя.

Задача 1

Имеется симметричная монета. Напишите функцию генерации независимых случайных величин из нормального и экспоненциального распределений с заданными параметрами.

In [2]:

```
# Эта ячейка -- единственная в задаче 1, в которой нужно использовать
# библиотечную функцию для генерации случайных чисел.
# В других ячейках данной задачи используйте функцию coin.

# симметричная монета
coin = sps.bernoulli(p=0.5).rvs
```

Проверьте работоспособность функции, сгенерировав 10 бросков симметричной монеты.

In [3]:

```
coin(size=10)
```

Out[3]:

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

Часть 1. Напишите сначала функцию генерации случайных величин из равномерного распределения на отрезке $[0, 1]$ с заданной точностью. Это можно сделать, записав случайную величину $\xi \sim U[0, 1]$ в двоичной системе счисления $\xi = 0, \xi_1 \xi_2 \xi_3 \dots$. Тогда $\xi_i \sim \text{Bern}(1/2)$ и независимы в совокупности. Приближение заключается в том, что вместо генерации бесконечного количества ξ_i мы полагаем $\xi = 0, \xi_1 \xi_2 \xi_3 \dots \xi_n$.

Нужно реализовать функцию так, чтобы она могла принимать на вход в качестве параметра `size` как число, так и объект `tuple` любой размерности, и возвращать объект `numpy.array` соответствующей размерности. Например, если `size=(10, 1, 5)`, то функция должна вернуть объект размера $10 \times 1 \times 5$. Кроме того, функцию `coin` можно вызвать только один раз, и, конечно же, не использовать какие-либо циклы. Аргумент `precision` отвечает за число n .

In [4]:

```
def uniform(size=1, precision=30):  
    count = np.prod(size)  
    data = coin((precision, count))  
    multipliers = np.full((precision, count), 2).cumprod(axis=0)  
    return (data / multipliers).sum(axis=0).reshape(size)
```

Для $U[0, 1]$ сгенерируйте 200 независимых случайных величин, постройте график плотности на отрезке $[-0.25, 1.25]$, а также гистограмму по сгенерированным случайным величинам.

In [5]:

```

size = 200
grid = np.linspace(-0.25, 1.25, 500)
sample = uniform(size=size, precision=50)

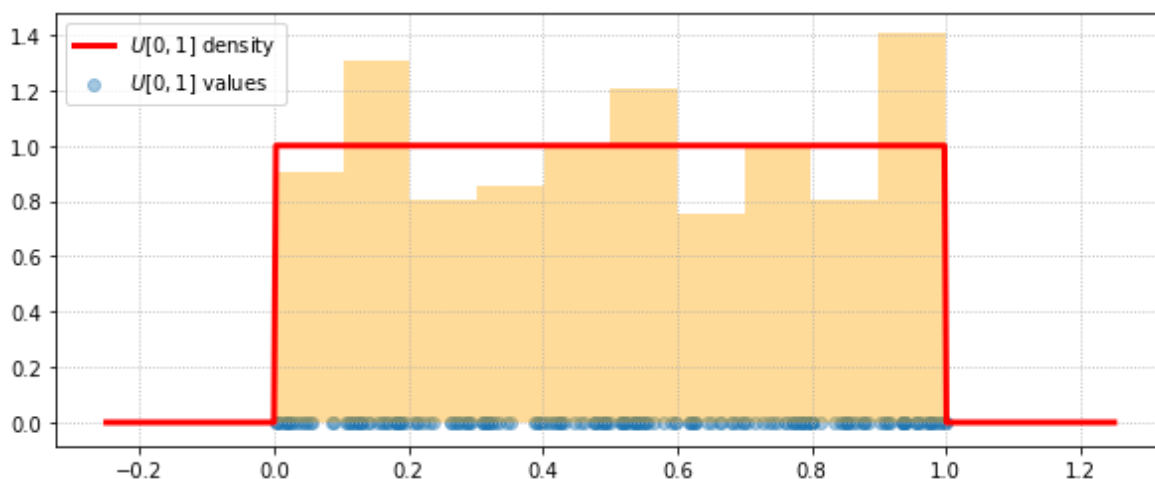
# Отрисовка графика
plt.figure(figsize=(10,4))

# отображаем значения случайных величин полупрозрачными точками
plt.scatter(
    sample,
    np.zeros(size),
    alpha=0.4,
    label=r'$U[0, 1]$ values'
)

# по точкам строим нормированную полупрозрачную гистограмму
plt.hist(
    sample,
    bins=10,
    density=True,
    alpha=0.4,
    color='orange'
)

# рисуем график плотности
plt.plot(
    grid,
    sps.uniform.pdf(grid),
    color='red',
    linewidth=3,
    label=r'$U[0, 1]$ density'
)
plt.legend()
plt.grid(ls=':')
plt.show()

```



Исследуйте, как меняются значения случайных величин в зависимости от `precision` .

In [6]:

```

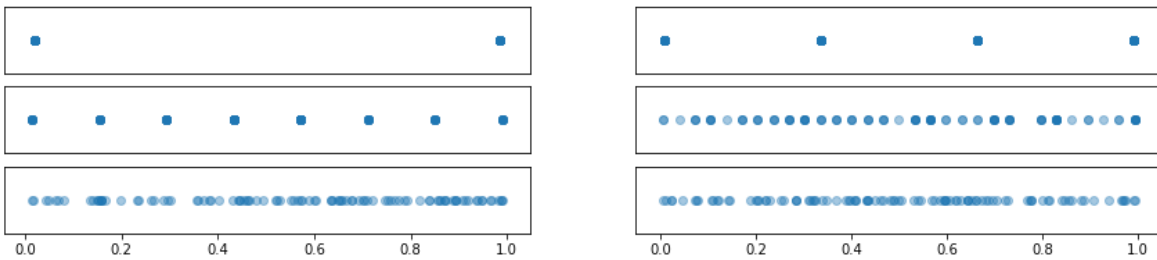
size = 100

plt.figure(figsize=(15, 3))

for i, precision in enumerate([1, 2, 3, 5, 10, 30]):
    plt.subplot(3, 2, i + 1)
    plt.scatter(
        uniform(size, precision),
        np.zeros(size),
        alpha=0.4
    )
    plt.yticks([])
    if i < 4: plt.xticks([])

plt.show()

```

**Вывод:**

Чем больше `precision`, тем равномернее распределены точки выборки. Это также следует из того, что при стремлении `precision` к бесконечности распределение стремится к абсолютно непрерывному равномерному распределению.

Часть 2. Напишите функцию генерации случайных величин в количестве `size` штук (как и раньше, тут может быть `tuple`) из распределения $\mathcal{N}(loc, scale^2)$ с помощью преобразования Бокса-Мюллера, которое заключается в следующем. Пусть ξ и η -- независимые случайные величины, равномерно распределенные на $(0, 1]$. Тогда случайные величины $X = \cos(2\pi\xi)\sqrt{-2\ln\eta}$, $Y = \sin(2\pi\xi)\sqrt{-2\ln\eta}$ являются независимыми нормальными $\mathcal{N}(0, 1)$.

Реализация должна быть без циклов. Желательно использовать как можно меньше бросков монеты.

In [7]:

```

def normal(size=1, loc=0, scale=1, precision=30):
    first_uniform = uniform(size, precision)
    second_uniform = uniform(size, precision)
    result = np.sin(2 * np.pi * first_uniform) * \
        np.sqrt(-2 * np.log(second_uniform))
    result = result * scale + loc
    return result

```

Для $\mathcal{N}(0, 1)$ сгенерируйте 200 независимых случайных величин, постройте график плотности на отрезке $[-3, 3]$, а также гистограмму по сгенерированным случайным величинам.

In [8]:

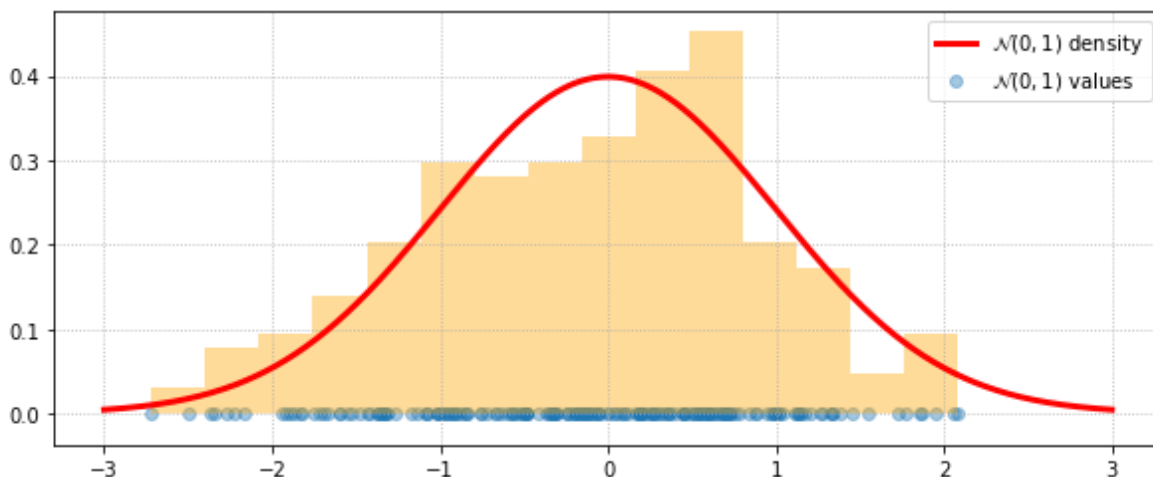
```
size = 200
grid = np.linspace(-3, 3, 500)
sample = normal(size=size, precision=50)

# Отрисовка графика
plt.figure(figsize=(10,4))

# отображаем значения случайных величин полупрозрачными точками
plt.scatter(
    sample,
    np.zeros(size),
    alpha=0.4,
    label=r'$\mathcal{N}(0, 1)$ values'
)

# по точкам строим нормированную полупрозрачную гистограмму
plt.hist(
    sample,
    bins=15,
    density=True,
    alpha=0.4,
    color='orange'
)

# рисуем график плотности
plt.plot(
    grid,
    sps.norm.pdf(grid),
    color='red',
    linewidth=3,
    label=r'$\mathcal{N}(0, 1)$ density'
)
plt.legend()
plt.grid(ls=':')
plt.show()
```



Сложная часть: генерация

Часть 3. Вы уже научились генерировать выборку из равномерного распределения. Напишите функцию генерации выборки из экспоненциального распределения, используя из теории вероятностей:

Если ξ --- случайная величина, имеющая абсолютно непрерывное распределение, и F --- ее функция распределения, то случайная величина $F(\xi)$ имеет равномерное распределение на $[0, 1]$.

Какое преобразование над равномерной случайной величиной необходимо совершить?

Пусть $\eta = F(\xi)$, $\eta \sim U[0, 1]$. Из курса теории вероятностей известна функция распределения экспоненциального распределения:

$$F(x) = (1 - e^{-\lambda x}) \cdot I_{[0; +\infty)}(x).$$

Поскольку случайная величина ξ принимает отрицательные значения с нулевой вероятностью, при подставлении её в её функцию распределения индикатор можно опустить:

$$F(\xi) = 1 - e^{-\lambda \xi}.$$

Преобразуем равенство:

$$\eta = 1 - e^{-\lambda \xi} \Leftrightarrow e^{-\lambda \xi} = 1 - \eta \Leftrightarrow -\lambda \xi = \ln(1 - \eta) \Leftrightarrow \xi = -\frac{1}{\lambda} \ln(1 - \eta).$$

Заметим, что случайные величины $1 - \eta$ и η распределены одинаково, поэтому вычитание не играет существенной роли. Итого: $\xi = -\frac{\ln \eta}{\lambda}$, $\eta \sim U[0, 1]$.

Для получения полного балла реализация должна быть без циклов, а параметр `size` может быть типа `tuple`.

In [9]:

```
def expon(size=1, lambd=1, precision=30):
    return (-1.0 / lambd) * np.log(uniform(size, precision))
```

Для $Exp(1)$ сгенерируйте выборку размера 100 и постройте график плотности этого распределения на отрезке $[-0.5, 5]$.

In [10]:

```

size = 100
grid = np.linspace(-0.5, 5, 500)
sample = expon(size=size, precision=50)

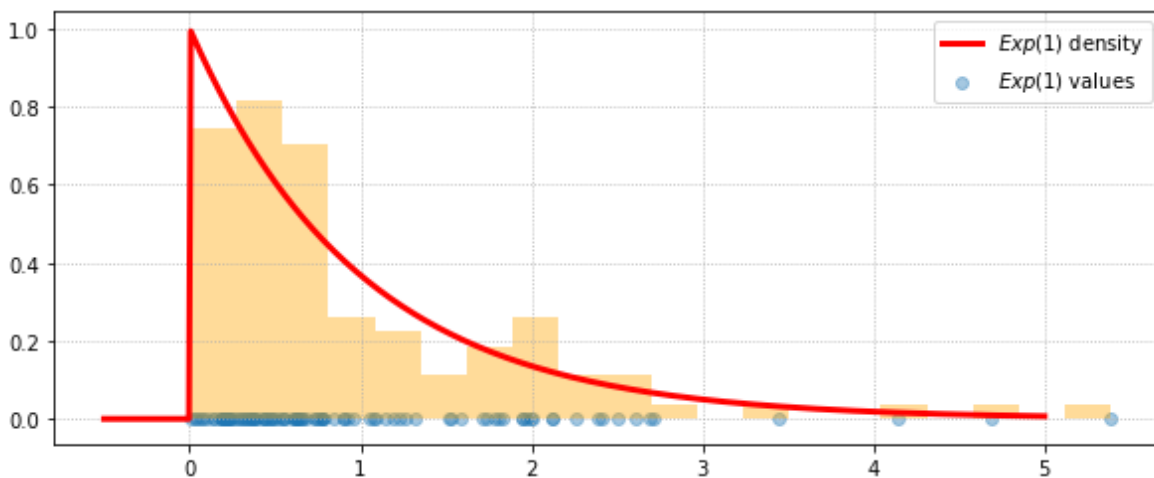
# Отрисовка графика
plt.figure(figsize=(10,4))

# отображаем значения случайных величин полупрозрачными точками
plt.scatter(
    sample,
    np.zeros(size),
    alpha=0.4,
    label=r'$Exp(1)$ values'
)

# по точкам строим нормированную полупрозрачную гистограмму
plt.hist(
    sample,
    bins=20,
    density=True,
    alpha=0.4,
    color='orange'
)

# рисуем график плотности
plt.plot(
    grid,
    sps.expon.pdf(grid),
    color='red',
    linewidth=3,
    label=r'$Exp(1)$ density'
)
plt.legend()
plt.grid(ls=':')
plt.show()

```

**Вывод по задаче:**

Применение функции F^{-1} к равномерному распределению на отрезке $[0, 1]$ позволяет получить случайную величину ξ , имеющую функцию распределения F . Такой метод может быть удобен для генерации случайных величин, имеющих удобные для исследования функции распределения.

Легкая часть: матричное умножение

Задача 2

Напишите функцию, реализующую матричное умножение. При вычислении разрешается создавать объекты размерности три. Запрещается пользоваться функциями, реализующими матричное умножение (`numpy.dot` , операция `@` , операция умножения в классе `numpy.matrix`). Разрешено пользоваться только простыми векторно-арифметическими операциями над `numpy.array` , а также преобразованиями осей. Авторское решение занимает одну строчку.

In [11]:

```
def matrix_multiplication(A, B):  
    '''Возвращает результат матричного умножения матриц A и B  
  
    ...  
  
    A = np.array(A)  
    B = np.array(B)  
  
    assert A.ndim == 2 and B.ndim == 2, \  
        'Перемножаемые объекты не являются матрицами'  
    assert A.shape[1] == B.shape[0], \  
        ('Матрицы размерностей {} и {} неперемножаемы'.format(A.shape, B.shape))  
  
    return (np.expand_dims(A, axis=2) * np.expand_dims(B, axis=0)).sum(axis=1)
```

Проверьте правильность реализации на случайных матрицах. Должен получиться ноль.

In [12]:

```
A = sps.uniform.rvs(size=(10, 20))  
B = sps.uniform.rvs(size=(20, 30))  
np.abs(matrix_multiplication(A, B) - A @ B).sum()
```

Out[12]:

8.704148513061227e-14

На основе опыта: вот в таком стиле многие из вас присылали бы нам свои работы, если не стали бы делать это задание :)

In [13]:

```
def stupid_matrix_multiplication(A, B):  
    C = [[0 for j in range(len(B[0]))] for i in range(len(A))]  
    for i in range(len(A)):  
        for j in range(len(B[0])):  
            for k in range(len(B)):  
                C[i][j] += A[i][k] * B[k][j]  
    return C
```

Проверьте, насколько быстрее работает ваш код по сравнению с неэффективной реализацией

`stupid_matrix_multiplication`. Эффективный код должен работать почти в 200 раз быстрее. Для примера посмотрите также, насколько быстрее работают встроенные `numpy`-функции.

In [14]:

```
A = sps.uniform.rvs(size=(400, 200))
B = sps.uniform.rvs(size=(200, 300))

%time C1 = matrix_multiplication(A, B)
%time C2 = A @ B # python 3.5
%time C3 = np.matrix(A) * np.matrix(B)
%time C4 = stupid_matrix_multiplication(A, B)
%time C5 = np.einsum('ij,jk->ik', A, B)
```

```
CPU times: user 68 ms, sys: 44.8 ms, total: 113 ms
Wall time: 112 ms
CPU times: user 2.46 ms, sys: 318 µs, total: 2.77 ms
Wall time: 1.16 ms
CPU times: user 6.77 ms, sys: 0 ns, total: 6.77 ms
Wall time: 1.99 ms
CPU times: user 13 s, sys: 26.1 ms, total: 13 s
Wall time: 12.4 s
CPU times: user 8.17 ms, sys: 3 µs, total: 8.17 ms
Wall time: 7.94 ms
```

Сложная часть: броуновское движение

Задача 3

Познавательная часть задачи (не пригодится для решения задачи)

Абсолютное значение скорости движения частиц идеального газа, находящегося в состоянии ТД-равновесия, есть случайная величина, имеющая распределение Максвелла и зависящая только от одного термодинамического параметра — температуры T .

В общем случае плотность вероятности распределения Максвелла для n -мерного пространства имеет вид:

$$p(v) = C e^{-\frac{mv^2}{2kT}} v^{n-1},$$

где $v \in [0, +\infty)$, а константа C находится из условия нормировки $\int_0^{+\infty} p(v) dv = 1$.

Физический смысл этой функции таков: вероятность того, что скорость частицы входит в промежуток $[v_0, v_0 + dv]$, приближённо равна $p(v_0)dv$ при достаточно малом dv . Тут надо оговориться, что математически корректное утверждение таково:

$$\lim_{dv \rightarrow 0} \frac{P\{v \mid v \in [v_0, v_0 + dv]\}}{dv} = p(v_0).$$

Обратите внимание на использование аннотаций для типов аргументов и возвращаемого значения функции. В новых версиях Питона подобные возможности синтаксиса используются в качестве подсказок для программистов и статических анализаторов кода, и никакой дополнительной функциональности не добавляют.

Например, `typing.Union[int, float]` означает "или `int`, или `float`".

Что может оказаться полезным

- Генерация нормальной выборки: `scipy.stats.norm`. [Ссылка](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html) (<https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html>).
- Кумулятивная сумма: метод `cumsum` у `np.ndarray`. [Ссылка](https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.cumsum.html) (<https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.cumsum.html>).

In [15]:

```
def generate_brownian(sigma: typing.Union[int, float] = 1,
                      *,
                      n_proc: int = 10,
                      n_dims: int = 2,
                      n_steps: int = 100) -> np.ndarray:
    """
    :param sigma: стандартное отклонение нормального распределения,
                  генерирующего пошаговые смещения координат
    :param n_proc: количество исследуемых броуновских частиц
    :param n_dims: размерность пространства,
                  в котором двигаются броуновские частицы
    :param n_steps: количество шагов симуляции

    :return: np.ndarray размера (n_proc, n_dims, n_steps), содержащий
             на позиции [i,j,k] значение j-й координаты i-й частицы
             на k-м шаге.
    """
    if not np.issubdtype(type(sigma), np.number):
        raise TypeError("Параметр 'sigma' должен быть числом")
    if not np.issubdtype(type(n_proc), np.integer):
        raise TypeError("Параметр 'n_proc' должен быть целым числом")
    if not np.issubdtype(type(n_dims), np.integer):
        raise TypeError("Параметр 'n_dims' должен быть целым числом")
    if not np.issubdtype(type(n_steps), np.integer):
        raise TypeError("Параметр 'n_steps' должен быть целым числом")

    brownian = sps.norm.rvs(size=(n_proc, n_dims, n_steps), scale=sigma)
    return brownian.cumsum(axis=2)
```

Символ `*` в заголовке означает, что все аргументы, объявленные после него, необходимо определять только по имени.

Например,

```
generate_brownian(323, 3)           # Ошибка
generate_brownian(323, n_steps=3)   # OK
```

При проверке типов остальных аргументов, по аналогии с `np.number`, можно использовать `np.integer`. Конструкция `np.issubdtype(type(param), np.integer)` используется по причине того, что стандартная питоновская проверка `isinstance(sigma, (int, float))` не будет работать

для numpy-чисел int64, int32, float64 и т.д.

In [16]:

```
brownian_2d = generate_brownian(2, n_steps=12000, n_proc=500, n_dims=2)
assert brownian_2d.shape == (500, 2, 12000)
```

2. Визуализируйте траектории для 9-ти первых броуновских частиц

Что нужно сделать

- Нарисовать 2D-графики для brownian_2d.
- Нарисовать 3D-графики для brownian_3d = generate_brownian(2, n_steps=12000, n_proc=500, n_dims=3).

Общие требования

- Установить соотношение масштабов осей, равное 1, для каждого из подграфиков.

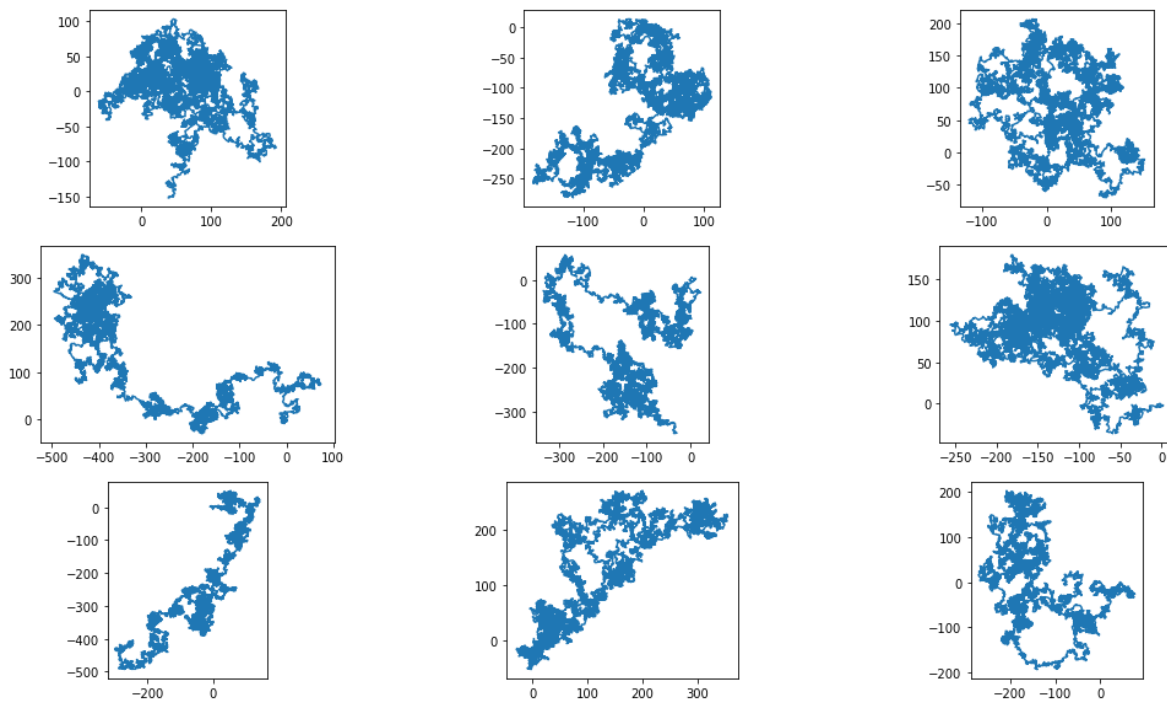
Что может оказаться полезным

- [Тьюториал \(https://matplotlib.org/devdocs/gallery/subplots_axes_and_figures/subplots_demo.html\)](https://matplotlib.org/devdocs/gallery/subplots_axes_and_figures/subplots_demo.html) по построению нескольких графиков на одной странице.
- Метод plot у AxesSubplot (переменная ax в цикле ниже).
- Метод set_aspect у AxesSubplot.

In [17]:

```
fig, axes = plt.subplots(3, 3, figsize=(18, 10))  
fig.suptitle('Траектории броуновского движения, 2D', fontsize=20)  
  
for ax, (xs, ys) in zip(axes.flat, brownian_2d):  
    ax.plot(xs, ys)  
    ax.set_aspect(1)  
plt.show()
```

Траектории броуновского движения, 2D



In [18]:

```
brownian_3d = generate_brownian(2, n_steps=12000, n_proc=500, n_dims=3)
```

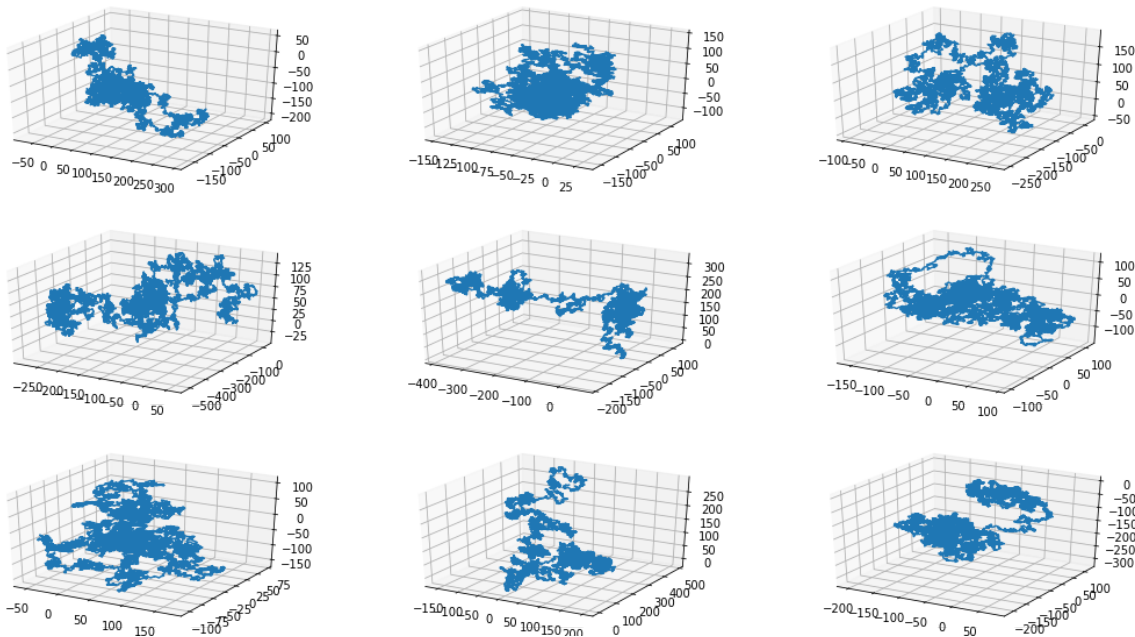
In [19]:

```
fig = plt.figure(figsize=(18, 10))
fig.suptitle('Траектории броуновского движения, 3D', fontsize=20)

for i, (xs, ys, zs) in enumerate(brownian_3d[:9]):
    ax = fig.add_subplot(3, 3, i + 1, projection='3d')
    ax.plot(xs, ys, zs)

plt.show()
```

Траектории броуновского движения, 3D



3. Постройте график среднего расстояния частицы от начала координат в зависимости от времени (шага)

- Постройте для n_dims от 1 до 5 включительно.
- Кривые должны быть отрисованы на одном графике. Каждая кривая должна иметь легенду.
- Для графиков подписи к осям обязательны.

Вопросы

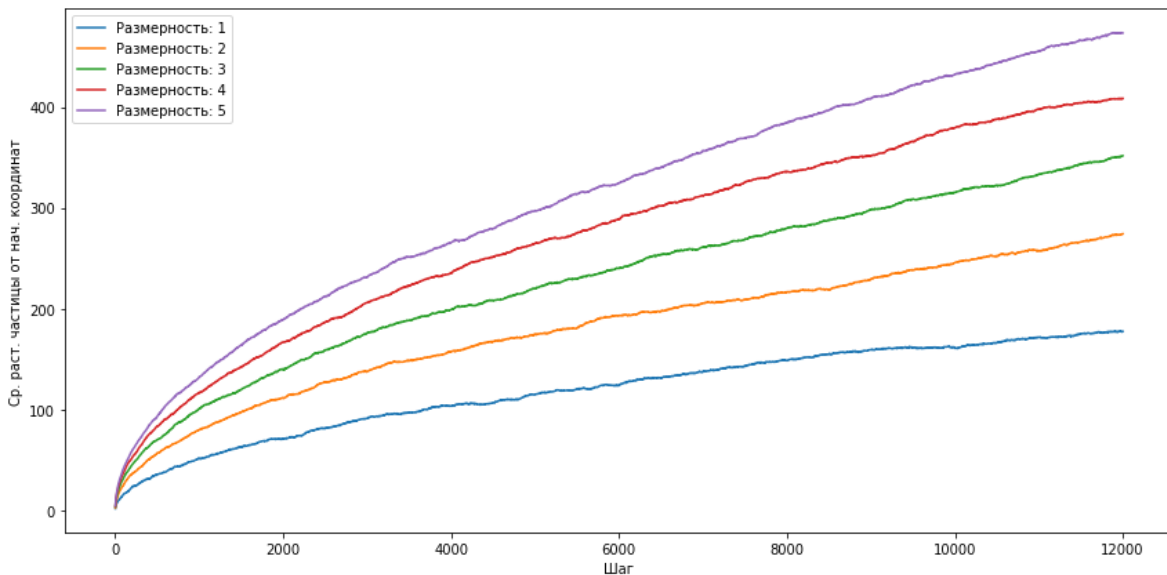
- Как вы думаете, какой функцией может описываться данная зависимость?
- Сильно ли её вид зависит от размерности пространства?
- Можно ли её линеаризовать? Если да, нарисуйте график с такими же требованиями.

In [20]:

```
plt.figure(figsize=(12, 6))

n_steps=12000
xs = np.linspace(1, n_steps, n_steps)
for n_dims in range(1, 6):
    brownian = generate_brownian(2, n_steps=n_steps, n_proc=500, n_dims=n_dims)
    avg_distance = np.sqrt(np.square(brownian).sum(axis=1)).mean(axis=0)
    plt.plot(
        xs,
        avg_distance,
        label=f'Размерность: {n_dims}'
    )

plt.ylabel('Ср. раст. частицы от нач. координат')
plt.xlabel('Шаг')
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```



По графикам можно сделать следующие предположения:

1. Среднее расстояние от начала координат пропорционально корню из числа шагов (это согласуется и с курсом физики).
2. Вид зависимости не меняется с ростом размерности пространства, зависимости отличаются только умножением на константу.

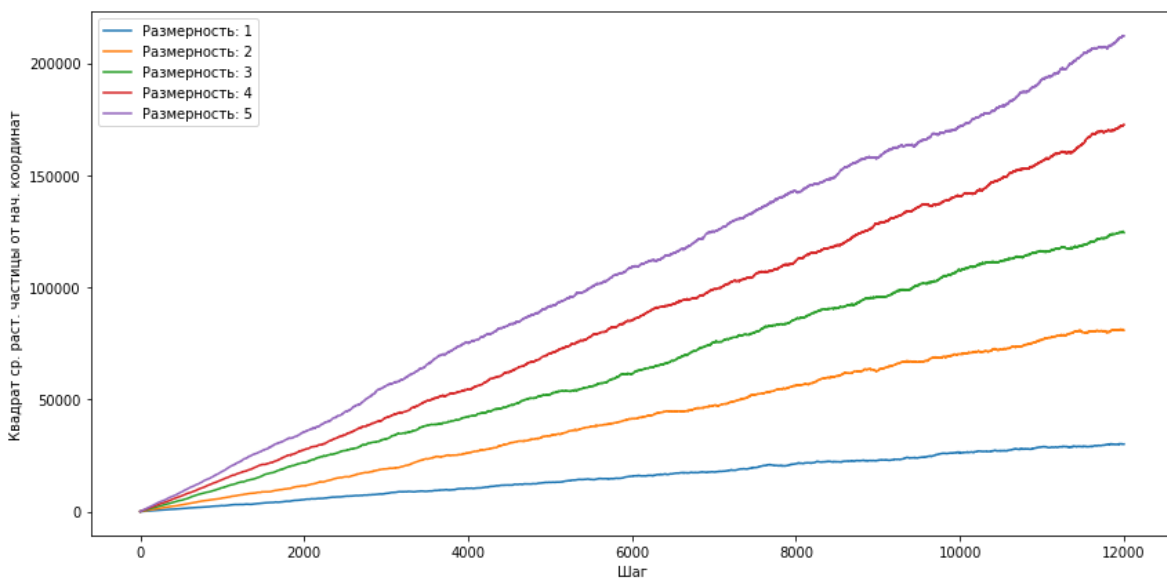
Для проверки первой гипотезы построим график зависимости квадрата среднего расстояния от числа шагов симуляции:

In [21]:

```
plt.figure(figsize=(12, 6))

n_steps=12000
xs = np.linspace(1, n_steps, n_steps)
for n_dims in range(1, 6):
    brownian = generate_brownian(2, n_steps=n_steps, n_proc=500, n_dims=n_dims)
    avg_distance = np.square(np.sqrt(np.square(brownian).sum(axis=1))).mean(axis=0)
    plt.plot(
        xs,
        avg_distance,
        label=f'Размерность: {n_dims}'
    )

plt.ylabel('Квадрат ср. раст. частицы от нач. координат')
plt.xlabel('Шаг')
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```



Полученные зависимости близки к линейным, значит зависимость среднего расстояния от начала координат действительно пропорциональна корню из числа шагов.

Сложная часть: визуализация распределений

Задача 4

В этой задаче вам нужно исследовать свойства дискретных распределений и абсолютно непрерывных распределений.

Для перечисленных ниже распределений нужно

1) На основе графиков дискретной плотности (функции массы) для различных параметров пояснить, за что отвечает каждый параметр.

- 2) Сгенерировать набор независимых случайных величин из этого распределения и построить по ним гистограмму.
- 3) Сделать выводы о свойствах каждого из распределений.

Распределения:

- Бернулли
- Биномиальное
- Равномерное
- Геометрическое

Для выполнения данного задания можно использовать код с лекции.

In [22]:


```
def plot_distribution(distr, name, rng, sample_size, n_bins, title):
    grid = np.linspace(rng[0], rng[1], rng[1] - rng[0] + 1)
    sample = distr.rvs(size=sample_size)
    plt.figure(figsize=(10, 4))
    plt.scatter(
        grid,
        distr.pmf(grid),
        color='r',
        label=name + ' weight function'
    )
    plt.hist(sample,
             density=True,
             bins=n_bins,
             color='orange',
             alpha=0.4,
             label=name + ' histogram')
    plt.title(title, fontsize='20')
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.show()
```


In [23]:

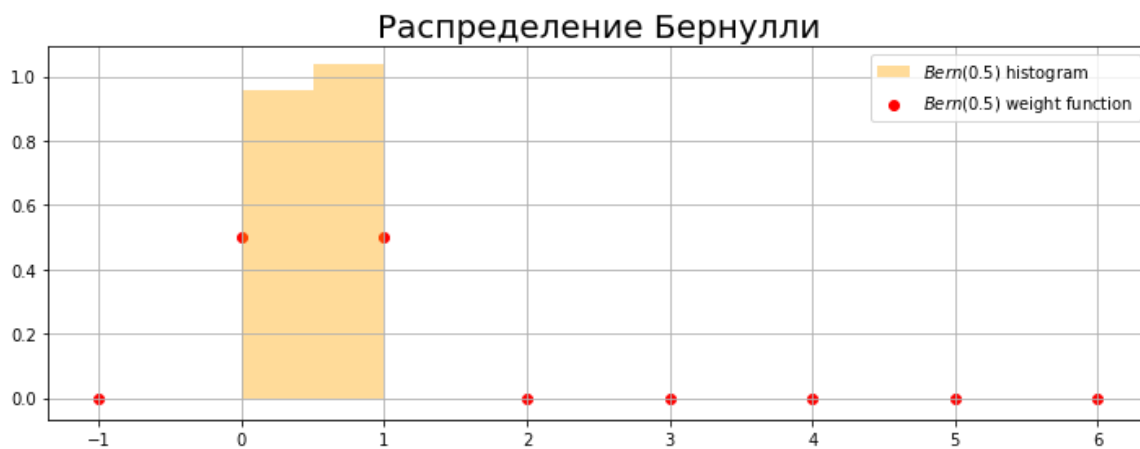
```
def plot_bernoulli(p=0.5, loc=0, sample_size=100):
    plot_distribution(
        sps.bernoulli(p, loc),
        fr'$Bern({p})$',
        (-1, 6),
        sample_size,
        n_bins=2,
        title='Распределение Бернулли'
    )
```

In [24]:

```
loc_slider = widgets.IntSlider(
    value=0,
    min=0,
    max=4,
    step=1,
    continuous_update=False
)
p_slider = widgets.FloatSlider(
    min=0,
    max=1,
    step=0.001,
    value=0.5,
    continuous_update=False
)
widgets.interact(
    plot_bernoulli,
    p=p_slider,
    loc=loc_slider,
    sample_size=widgets.fixed(100)
)
```

p  0.50

loc  0



Out[24]:

```
<function __main__.plot_bernoulli(p=0.5, loc=0, sample_size=100)>
```

Вывод: Параметр *loc* отвечает за положение пиков распределения, а параметр *p* --- за высоту одного из этих. Всего же распределение обладает двумя пиками с ненулевой весовой функцией.

In [25]:

```
def plot_binom(n, p, loc=0, sample_size=500):  
    plot_distribution(  
        sps.binom(n=n, p=p, loc=loc),  
        fr'$Bern({n}, {p})$',  
        (-1, 50),  
        sample_size,  
        n_bins=n+1,  
        title='Биноминальное распределение'  
    )
```

In [26]:

```

loc_slider = widgets.IntSlider(
    value=0,
    min=0,
    max=4,
    step=1,
    continuous_update=False
)
p_slider = widgets.FloatSlider(
    min=0,
    max=1,
    step=0.001,
    value=0.5,
    continuous_update=False
)
n_slider = widgets.IntSlider(
    value=10,
    min=1,
    max=45,
    step=1,
    continuous_update=False
)
widgets.interact(
    plot_binom,
    n=n_slider,
    p=p_slider,
    loc=loc_slider,
    sample_size=widgets.fixed(1000)
)

```

n 10
 p 0.50
 loc 0



Out[26]:

```
<function __main__.plot_binom(n, p, loc=0, sample_size=500)>
```

Вывод: Параметр *loc* влияет только на параллельный перенос графиков дискретной плотности. Параметр *n* отвечает за диапазон значений, которые может принимать случайная величина, а параметр *p* --- за смещение наиболее вероятного значения внутри диапазона $[0, n] \cap \mathbb{N}$. В целом же весовая

функция распределения плавно возрастает до наиболее вероятного значения и убывает после него.

In [27]:

```
def plot_uniform(low, high, sample_size=500):
    plot_distribution(
        sps.randint(low, high),
        fr'$U(\{low\}, \{high\})$',
        (-1, 50),
        sample_size,
        n_bins=high - low,
        title='Равномерное распределение'
    )
```

In [28]:

```
low_slider = widgets.IntSlider(
    value=0,
    min=0,
    max=20,
    step=1,
    continuous_update=False
)
high_slider = widgets.IntSlider(
    value=25,
    min=21,
    max=49,
    step=1,
    continuous_update=False
)
widgets.interact(
    plot_uniform,
    low=low_slider,
    high=high_slider,
    sample_size=widgets.fixed(500)
)
```

low 0
 high 25



Out[28]:

```
<function __main__.plot_uniform(low, high, sample_size=500)>
```


Вывод: Случайная величина, соответствующая данному распределению, равновероятно принимает все значения из $[low, high) \cap \mathbb{N}$.


In [29]:

```
def plot_geom(p, loc, sample_size=500):  
    plot_distribution(  
        sps.geom(p, loc),  
        fr'$Geom({p})$',  
        (-1, 20),  
        sample_size,  
        n_bins=20,  
        title='Геометрическое распределение'  
    )
```

In [30]:

```
loc_slider = widgets.IntSlider(
    value=0,
    min=0,
    max=4,
    step=1,
    continuous_update=False
)
p_slider = widgets.FloatSlider(
    min=0.2,
    max=0.4,
    step=0.001,
    value=0.3,
    continuous_update=False
)
widgets.interact(
    plot_geom,
    p=p_slider,
    loc=loc_slider,
    sample_size=widgets.fixed(100)
)
```

p  0.30

loc  0



Out[30]:

```
<function __main__.plot_geom(p, loc, sample_size=500)>
```

Вывод: Весовая функция распределения убывает, причем чем больше параметр p , тем быстрее убывание и тем выше стартовая точка.