# 3_randomnumbers

April 23, 2024

## 1 Random numbers

File as PDF

Random numbers are widely used in science and engineering computations. They can be used to simulate noisy data, or to model physical phenomena like the distribution of velocities of molecules in a gas, or to act like the roll of dice in a game. Monte Carlo simulation techniques, which will be part of a later theory lecture rely heavily and random number. Processes like single photon emission or Brownian motion are stochastic processes, with an intrisic randomness as well. But there are even methods for numerically evaluating multi-dimensional integrals using random numbers.

The basic idea of a random number generator is that it should be able to produce a sequence of numbers that are distributed according to some predetermined distribution function. NumPy provides a number of such random number generators in its library `numpy.random`.

The random number functions can be imported by

```
from numpy.random import *
```

### 1.1 Uniformly distributed random numbers

The rand(num) function creates an array of num floats uniformly distributed on the interval from 0 to 1.

[23]:
```
import matplotlib.pyplot as plt
import numpy as np
from numpy.random import *

%config InlineBackend.figure_format = 'retina'
plt.rcParams.update({'font.size': 8,
                     'lines.linewidth': 1,
                     'lines.markersize': 5,
                     'axes.labelsize': 10,
                     'xtick.labelsize' : 9,
                     'ytick.labelsize' : 9,
                     'legend.fontsize' : 6,
                     'contour.linewidth' : 1,
                     'xtick.top' : True,
                     'xtick.direction' : 'in',
                     'ytick.right' : True,
                     'ytick.direction' : 'in',
                     'figure.figsize': (4, 3),
                     'figure.dpi': 150 })

def get_size(w,h):
    return((w/2.54,h/2.54))
```

```
[2]: rand()
```

```
[2]: 0.4191827791095334
```

If you supply the argument *num* to the `rand(num)` function you obtain an array of equally distributed numbers with *num* elements.

```
[3]: rand(5)
```

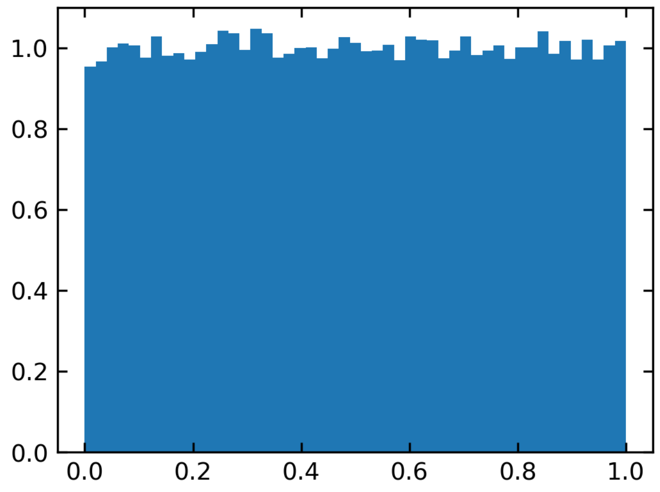
```
[3]: array([0.84434484, 0.27214362, 0.06893966, 0.02165642, 0.96088836])
```

You may also obtain a multi-dimensional array if you give two or more numbers to the the rand function.

```
[4]: rand(5,2)
```

```
[4]: array([[0.64052054, 0.14230412],
            [0.38727132, 0.39596272],
            [0.13436942, 0.63582847],
            [0.5333115 , 0.22051674],
            [0.68829207, 0.06064005]])
```

```
[5]: b=np.linspace(0,1,50)
     n,bins,_=plt.hist(rand(100000),bins=b,density=1)
     plt.show()
```



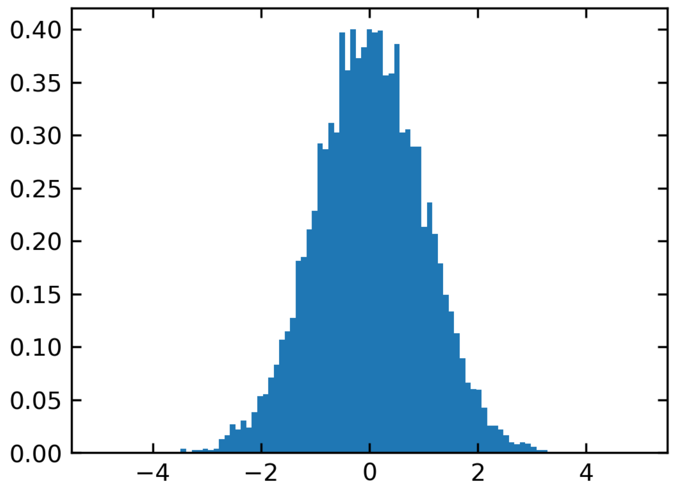
## 1.2 Normally distributed random numbers

The function randn(num) produces a normal or Gaussian distribution of num random numbers with a mean of $\mu = 0$ and a standard deviation of $\sigma = 1$. They are distributed according to

$$p(x)dx = \frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(x-\mu)^2}{2\sigma^2}}dx \tag{1}$$

Similarly as all the other random number function, you may supply one or multiple arguments to the *rand()* function. The result is again a multi-dimensional array of random numbers.

[6]:
```
b=np.linspace(-5,5,100)
plt.hist(randn(10000),bins=b,density=1)
plt.show()
```



If you want to create a normal distribution with different standard deviation and mean value, you have to multiply by the new standard deviation and add the mean.

[7]:
```
sigma=10
mu=10

d=sigma * randn(1000) + mu
```

**Note:** Physics Interlude

Brownian Motion, Random Walk

The following lines create random numbers, which are distributed by a normal distribution. You can use such normally distributed random numbers to generate a random walk. Such a random walk is a simple representation of Brownian motion of colloids suspended in a liquid.

Suppose a colloidal particle in solution is kicked by its surrounding solvent molecules such that it does a small step $(\Delta x_i, \Delta y_i)$ in a random direction. The length of the step $(\Delta x_i, \Delta y_i)$ will be normally distributed. If this motion is in 2 dimensions, then the position in x and y direction after N steps is

$$x(N) = \sum_{i=1}^{N} \Delta x_i \tag{2}$$

$$y(N) = \sum_{i=1}^{N} \Delta y_i \tag{3}$$

The position is therefore just the sum of a sequence of normally distributed random numbers, which is easy to realize in Python with the `np.sum()` function of `numpy`. One can even go a little bit farther by evaluating the sum at each index $i$ and not just only the result after $N$ steps. A function providing this results is `np.cumsum()`, the cummulative sum.

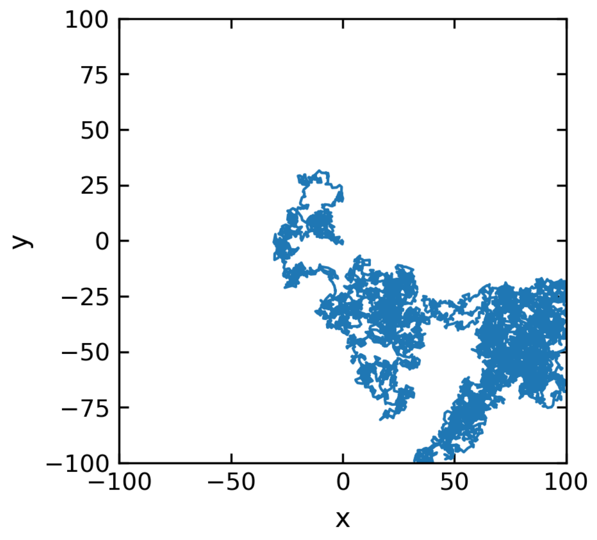
[15]:
```
x,y=[randn(10000).cumsum(),randn(10000).cumsum()]
```

Note that the above example uses the `cumsum()` function. The cumsum function of an array of number $[x_0, x_1, x_2, .., x_n]$ delivers an array with a progressive sum of elements $[x_0, x_0+x_1, x_0+x_1+x_2, ..., x_0+...+x_n]$. The line

`x,y=[randn(1000).cumsum(),randn(1000).cumsum()]`

is therefore all you need to generate a random walk in two dimensions.

[16]:
```
plt.figure(figsize=(3,3))
plt.plot(x,y,'-')
plt.xlabel('x') # set the x-axis label
plt.ylabel('y') # set the y-axis label
plt.xlim(-100,100)
plt.ylim(-100,100)
plt.show()
```



## 1.3 Exponentially distributed numbers

A number of processes in physics reveal an exponential statistics. For example the probability to find a molecule under gravity at a certain height $h$ is distributed by a Boltzmann law for a non-zero temperature.

$$p(h)dh = p_0 \exp(-m \cdot g \cdot h/k_{\mathrm{B}}T)dh \tag{4}$$

On the other hand, the probability to emit a photon spontaneously after a certain time $t$ follwings the excited state preparation (two level system) is also exponentially distributed.

$$p(t)dt = p_0 \exp(-t/\tau)dt \tag{5}$$

Thus after each excitation i.e. by a laser pulse, a molecule emits a single photon with the probability $p(t)$ and the whole exponential character in the statistics is only appearing after repeating the experiment several times.

The exponential distribution of `numpy` can be supplied with two numbers.

`exponential(b, n)`

The parameter b is giving the decay parameter. The number n is optional and giving the number of samples to be provided. The numbers are distributed according to

$$\frac{1}{b}\exp(-x/b) \tag{6}$$

[17]:
```
exponential(1)
```

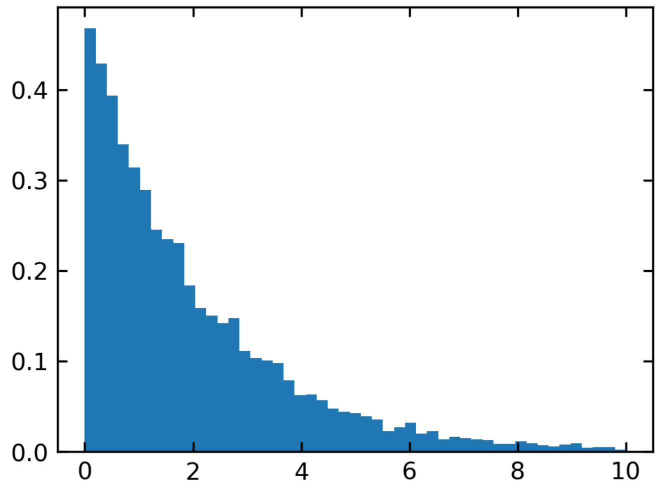
```
[17]: 0.01564909653180085
```

```
[18]: exponential(1,10000)
```

```
[18]: array([1.37491591, 2.08398478, 0.8786739 , …, 1.17561624, 0.30747314,
              0.87451049])
```

You may want to test the changes in the exponential distribution with the parameter **b**.

```
[19]: b=np.linspace(0,10,50)
      n,bins,_=plt.hist(exponential(2,10000),bins=b,density=1)
      plt.show()
```



## 1.4 Central Limit theorem

The central limit theorem (CLT) is a fundamental concept in statistics that describes the behavior of the sum or average of many independent random variables, even if the original variables themselves are not normally distributed.

- Consider a population with a mean $\mu$ and a finite variance $\sigma^2$.
- Take random samples of size $n$ from this population and calculate the sample mean for each sample.
- The central limit theorem states that as the sample size $n$ increases, the distribution of the sample means will approach a normal distribution, regardless of the original distribution of the population.
- The mean of the sampling distribution of the sample means will be equal to the population mean $\mu$.
- The standard deviation of the sampling distribution of the sample means will be $\sigma/\sqrt{n}$ , where $\sigma$ is the population standard deviation.

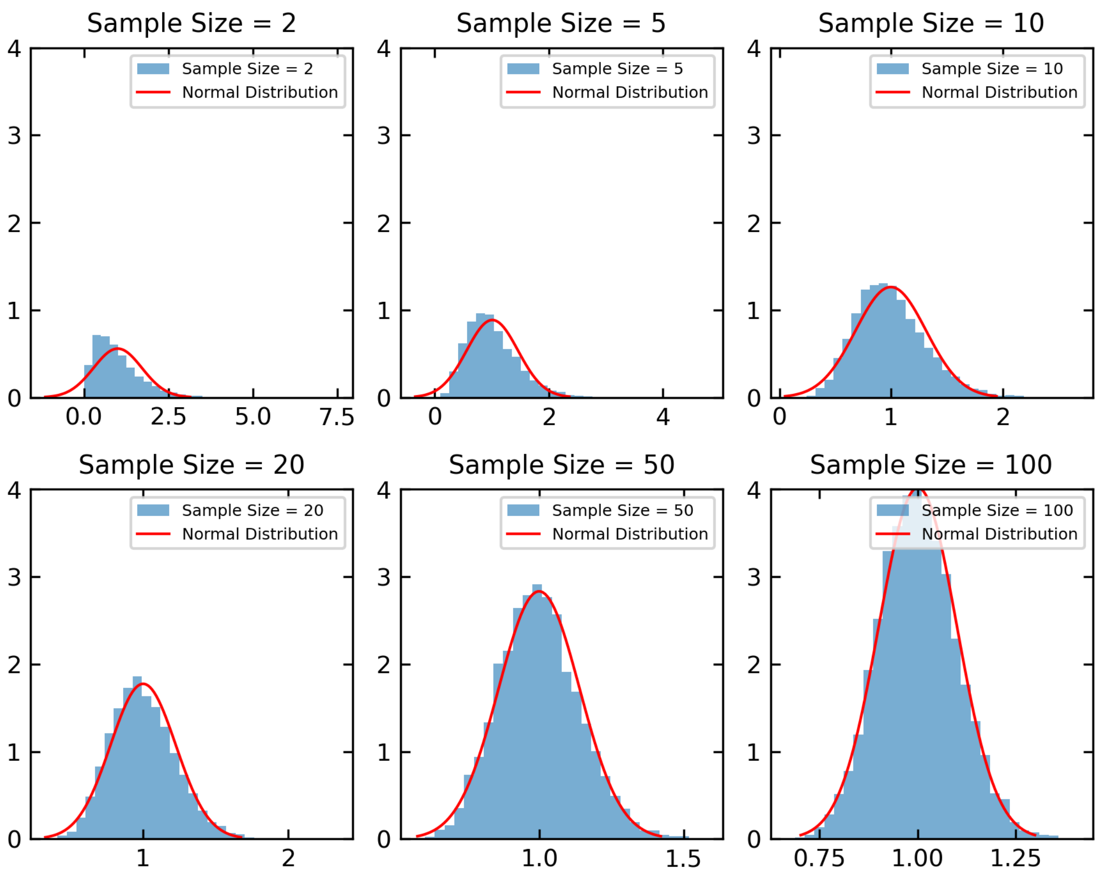
The intuition behind the central limit theorem is that when many independent random variables are combined, the cumulative effect of their different distributions tends to cancel out and the resulting sum or average converges to a normal distribution due to the central tendency of the data.

This theorem is extremely useful in statistics as it allows us to draw conclusions about the parameters of the population based on sample data, even if the original population distribution is unknown or non-normal. It justifies the use of many statistical methods that assume normality, such as hypothesis testing, confidence intervals and regression analysis, as long as the sample size is sufficiently large.

The central limit theorem is considered one of the most important and remarkable results in probability theory and statistics and has numerous applications in various fields, including science, engineering, finance and social sciences.

Below is a code, that demonstrates the central limit theorem using and exponential distribution and different sample sizes.

[32]:
```
population_dist = np.random.exponential

# Sample sizes to demonstrate
sample_sizes = [2, 5, 10, 20, 50, 100]

num_samples = 10000

sample_means = {size: [np.mean(population_dist(scale=1, size=size)) for _ in␣
 ↪range(num_samples)] for size in sample_sizes}

fig, axes = plt.subplots(2, 3, figsize=get_size(15, 12))
axes = axes.flatten()

for ax, size in zip(axes, sample_sizes ):

    ax.hist(sample_means[size], bins=30, density=True, alpha=0.6,␣
 ↪label=f'Sample Size = {size}')

    mean = np.mean(sample_means[size])
    std = np.std(sample_means[size])
    x = np.linspace(mean - 3*std, mean + 3*std, 100)
    y = np.exp(-(x - mean)**2 / (2 * std**2)) / (np.sqrt(2 * np.pi) * std)
    ax.plot(x, y, 'r', lw=1, label='Normal Distribution')
    ax.set_ylim(0,4)
    ax.set_title(f'Sample Size = {size}')
    ax.legend()

plt.tight_layout()
plt.show()
```



## 1.5 Random distribution of integers

The function randint(low, high, num) produces a uniform random distribution of num integers between low (inclusive) and high (exclusive).

[18]:
```
randint(1,20,10)
```

[18]:
```
array([ 8, 14, 11,  9, 14,  8,  7, 18, 18,  1])
```

[19]:
```
items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
shuffle(items)

print(items)
```

```
[1, 3, 5, 8, 2, 4, 6, 7, 10, 9]
```

There are a number of other methods available in the random module of numpy. Please refere to the documentation.