Solution Problem Set I

Mauricio Sevilla

August 24, 2020

0.1 Pi Calculation

The Leibniz formula to calculate π goes as follows,

$$\pi = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{(2k+1)}$$

Create a function that calculates π as follows: - Initialize a variable named as you want where you are going to store your result. - You have to truncate the summation, so create another variable N with the largest value of k on the summation you want - Use a `for` loop to do the operation for each k saving the result on your variable. - print out the final result. - Plot how π vs k to see how it goes to the real value using as a reference the value saved on `numpy` as `numpy.pi`.

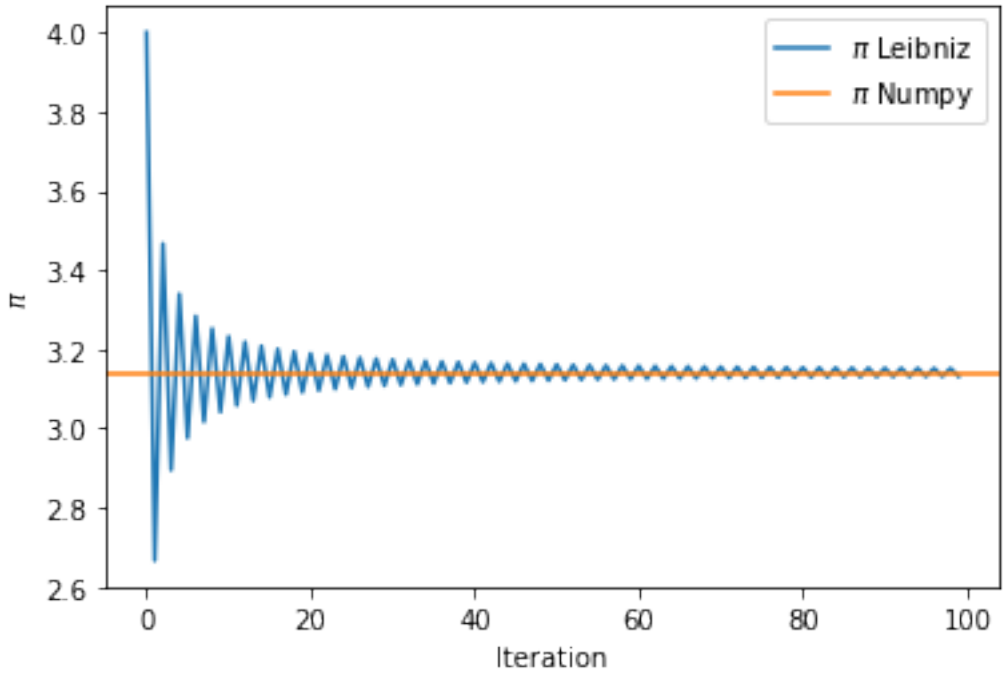
hint: To see the reference, you can use the function `axhline` of `matplotlib.pylab`.

[1]:
```
import numpy as np
import matplotlib.pylab as plt
```

[2]:
```
def pi(n):
    result=0
    for k in range(n+1):
        result+=4*((-1)**k)/((2*k+1))
    return result
values=[]
for i in range(100):
    values.append(pi(i))
```

[3]:
```
plt.plot(values,label='$\pi$ Leibniz')
plt.axhline(np.pi,color='C1',label='$\pi$ Numpy')
plt.legend()
plt.xlabel('Iteration')
plt.ylabel('$\pi$')
```

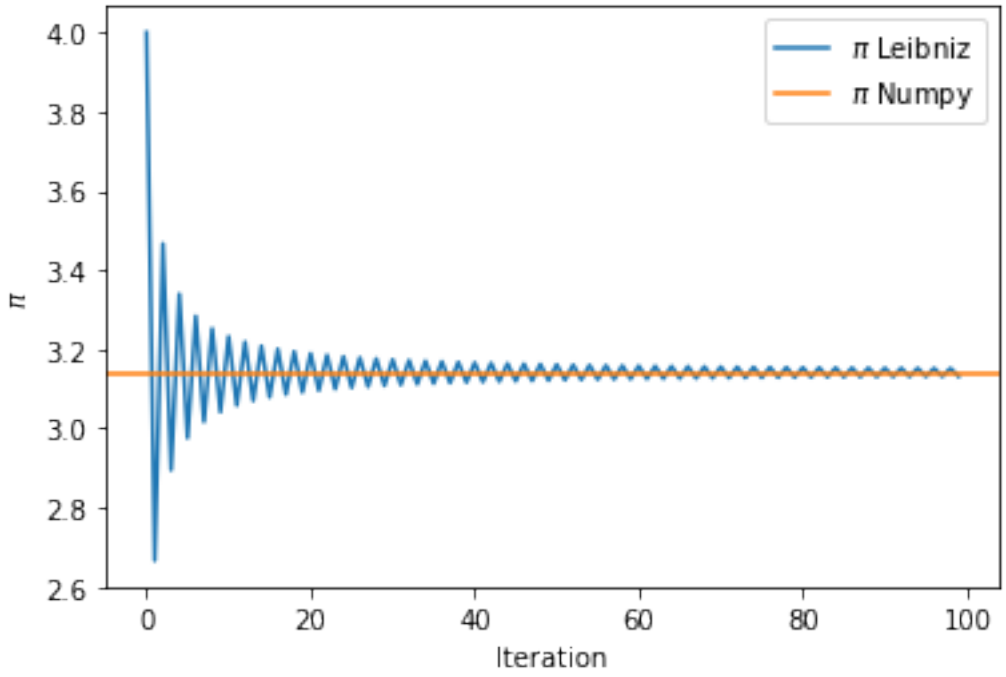
[3]:
```
Text(0, 0.5, '$\\pi$')
```



[4]:
```
def step_pi(n):
    k=np.linspace(0,n-1,n)
    return (4*((-1)**k)/((2*k+1))).cumsum()
step=step_pi(100)
```

[5]:
```
plt.plot(step,label='$\pi$ Leibniz')
plt.axhline(np.pi,color='C1',label='$\pi$ Numpy')
plt.legend()
plt.xlabel('Iteration')
plt.ylabel('$\pi$')
```

[5]:
```
Text(0, 0.5, '$\\pi$')
```



[6]:
```
%timeit pi(10000)
%timeit step_pi(10000)
```

```
4.97 ms ± 34 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
190 µs ± 1.4 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

0.2 Standard map

Let us consider the following map,

$$p_{n+1} = p_n + K \sin(\theta_n) \tag{1}$$
$$\theta_{n+1} = \theta_n + p_{n+1} \mod 2\pi \tag{2}$$

As the variable θ is an angle, one should expect to have it bounded. So - Import the library `math` to have the sin function and use $K = 1.5$ - Consider the initial conditions $\theta_0 = 0$ and $p_0 = 1$. - Construct a loop that runs N times. - If $\theta > 2\pi$ then use the operator `%` (Module) to make the map bounded. - Use the value of π you just calculated.

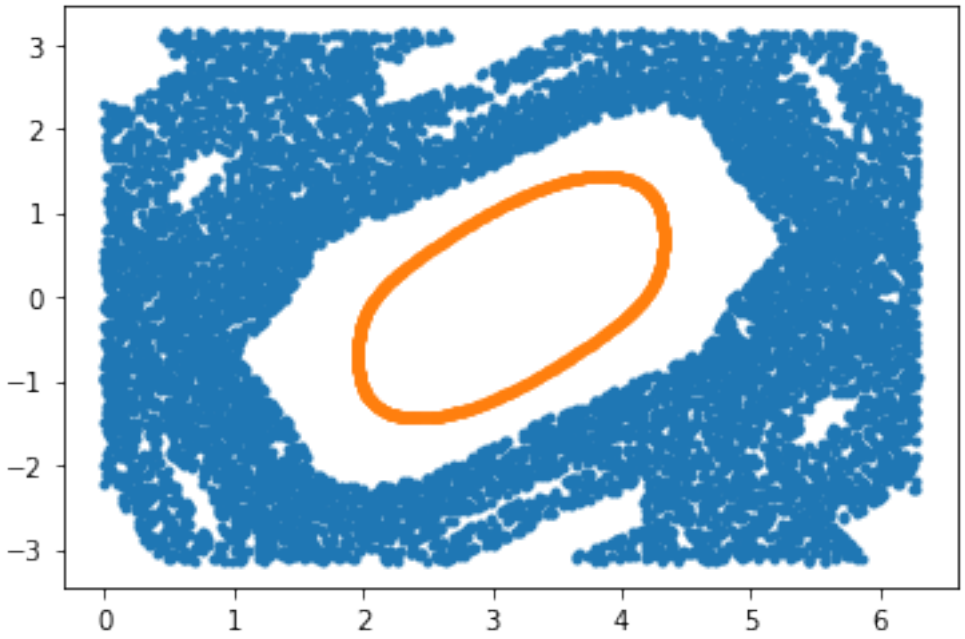
[7]:
```
MyPi=step_pi(10000)[-1]
def standard_map(p,q,K):
    p+=K*np.sin(q)
    q+=p
    return (p+MyPi)%(2*MyPi)-MyPi,(q%(2*MyPi))
```

[8]:
```
q=0
p=1

q_plot=[]
p_plot=[]
for i in range(10000):
    p,q=standard_map(p,q,1.5)
    q_plot.append(q)
    p_plot.append(p)
```

[9]:
```
q=2
p=6

q_plot2=[]
p_plot2=[]
for i in range(1000):
    p,q=standard_map(p,q,1.5)
    q_plot2.append(q)
    p_plot2.append(p)
```

[10]:
```
plt.plot(q_plot,p_plot,'.')
plt.plot(q_plot2,p_plot2,'.')
```

[10]:
```
[<matplotlib.lines.Line2D at 0x11efac8b0>]
```



0.3 Standard map with extended angle coordinates

Consider the Standard map we have already studied a little,

$$p_{n+1} = p_n + K\sin(\theta_n) \tag{3}$$
$$\theta_{n+1} = \theta_n + p_{n+1} \tag{4}$$

With the difference now θ is not periodic anymore.

A quantity that one can study from here is the *diffusion*. The diffusion is always measured from the difference of an evolved coordinate minus the initial conditions. The dependence of the difference (On average) with the step (Or time for continuous evolution) goes generally as a power law if there is any diffusion, where the constant going with the power law is called the diffusion constant D, while the exponent α determines the rate of the diffusion.

$$\left\langle (\theta_0 - \theta_n)^2 \right\rangle = Dn^\alpha$$

If the exponent $\alpha = 1$, it is called diffusion, but if $\alpha \neq 1$, it is called anomalous diffusion. if $\alpha > 1$ it is called superdiffusion and $\alpha < 1$ is called subdiffusion.

Show that the standard map with extended angle coordinates, exhibits superdiffusion.

- Consider a large value of K (Chaotic regime) $K \approx 10$ is good.
- Take a large set of initial conditions (randomly distributed).
- Save the initial conditions.
- Evolve, saving after certain evolutions to do the plot.
- Average the square differences.
- Plot the results.

[11]:
```
def standard_map_ext(p,q,K):
    p+=K*np.sin(q)
    q+=p
    return p,q
```

[12]:
```
N=100000

q=np.random.random(N)
p=np.random.random(N)

q0=q.copy()
p0=p.copy()

D=[]

for i in range(1000):
    p,q=standard_map_ext(p,q,10)
    D.append(((q-q0)**2).mean())
```

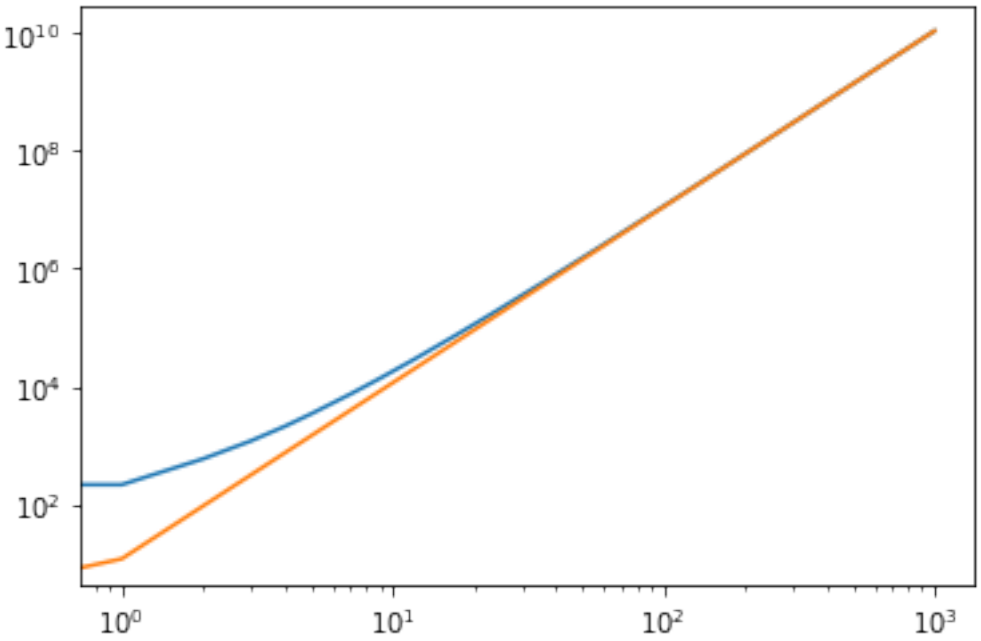
[13]:
```
def power_law(x,a,b):
    return a*x**b
```

[14]:
```
from scipy.optimize import curve_fit
```

[15]:
```
popt,pcov=curve_fit(power_law,range(len(D)),D)
```

[16]:
```
plt.loglog(D)
plt.loglog(range(len(D)),power_law(range(len(D)),*popt))
```

[16]:
```
[<matplotlib.lines.Line2D at 0x12c840430>]
```



[17]:
```
popt
```

[17]:
```
array([11.72921453,  2.99070216])
```

[18]:
```
pcov.diagonal()**0.5
```

[18]:
```
array([2.54865399e-03, 3.21173601e-05])
```

0.4 Wigner Semicircle Law

There is a very important distribution on random matrix theory, which is called Wigner Semicircle Distribution named after the physicist Eugene Wigner (Nobel prize awarded), also known for his phase space representation of quantum mechanics (I hope we can discuss this later).

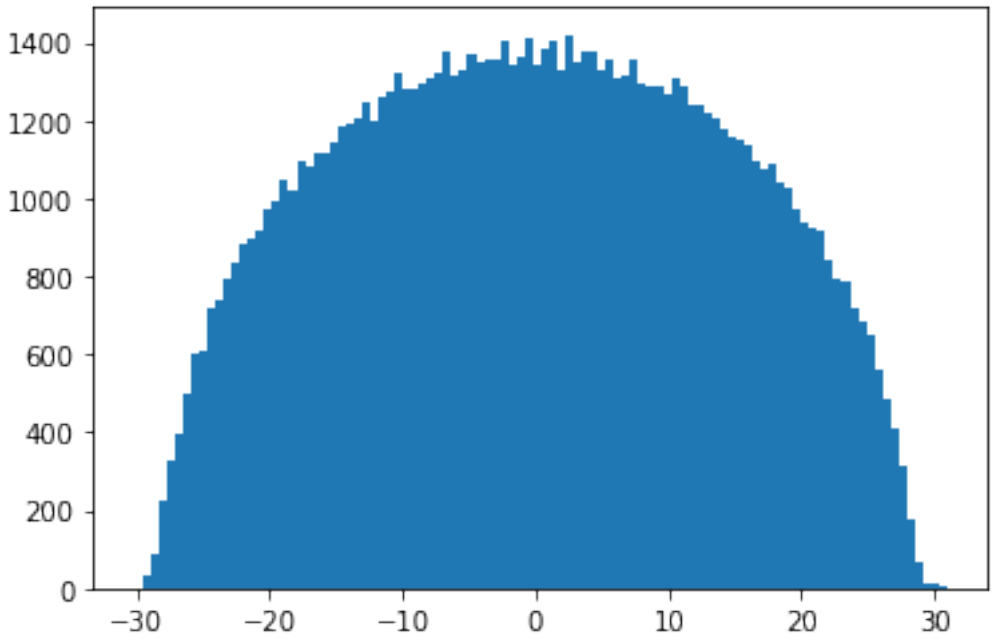
This distribution can be found by calculating the distribution of the eigenvalues of symmetric random matrices with Gaussian distributed entries, and that is exactly how we will calculate it.

We will generate 1000 random matrices of size 100×100, so it is a good idea for you to define those values before starting.

- Do a for loop running on the number of matrices (1000 in our case), and then inside that for, generate a Gaussian distributed matrix by using the function `np.random.normal` using $\mu = 0$ and $\sigma = 1$.
- Make the matrix symmetric. (Add it by the transpose.).
  - **Hint:** The transpose of a matrix `M` on python can be calculated as `M.T`
- For each matrix, calculate its eigenvalues, it is easily done with the function `eigen=np.linalg.eigvals(matrix)`
- Save on an array the values of all the eigenvalues of all the matrices, you can use the function append, such as `eigen_vals=np.append(eigen_vals,eigen)`
- Do a histogram of them. (Use 100 bins)

[19]:
```
eigen_vals=[]
for i in range(1000):
    m=np.random.normal(0,1,(100,100))
    m=m+m.T
    eigen=np.linalg.eigvals(m)
    eigen_vals=np.append(eigen_vals,eigen)
```

[20]:
```
plt.hist(eigen_vals,100);
```



0.5 Maxwell Boltzmann Distribution

This is one of the most important distributions on molecular dynamics, proposed by Maxwell and then by Boltzmann back in 1860 and 1872 (1877) respectively.

Based on the kinetic theory to describe ideal gases, the velocities on each direction distribute as a Gaussian but when combining them we get a different distribution,

$$v = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

One has to take a look to the derivation (Which with the statistical mechanics theory becomes very easy), but on the literature one can find that the velocity distribution is

$$f(v) = 4\pi v^2 \left(\frac{m}{2\pi k_B T}\right)^{3/2} e^{\frac{-mv^2}{2k_B T}}$$

but this result only applies to a 3 dimensional gas and here we will use results from a 2 Dimensional simulation!!!.

If you want to see a small demonstration, the following link has a small gif of the simulation we are studying link

The relationship we may use is

$$f(v) = 2\pi v \left(\frac{m}{2\pi k_B T}\right) e^{\frac{-mv^2}{2k_B T}}$$

On the following link, you may find the data result from the simulation link. There are 3 columns, v_x, v_y and $v = \sqrt{v_x^2 + v_y^2}$

- Collect the data.
- Convince yourself that the distributions of v_x and v_y are Gaussians by doing the histogram.
- To prove that the speeds distribute as we said before, we are going to do a fit of the histogram to the model, so first, define a function you want to use as your model.
  - **Hint:** You do not have to use all those values of $k_B T$ and m, use a parameter for the amplitude and another for the exponent.
- Do the histogram (with 20 bins) of the velocities (third column of the file) but!, save the values. (`histogram = plt.hist(velocities,bins=20)`). Then you will have save on histogram the amplitudes and limits of the boxes. To have one value for each box, we take the average

```
x_vals =(histogram [1][1:]+ histogram [1][:-1])/2
frequencies = histogram [0]
```

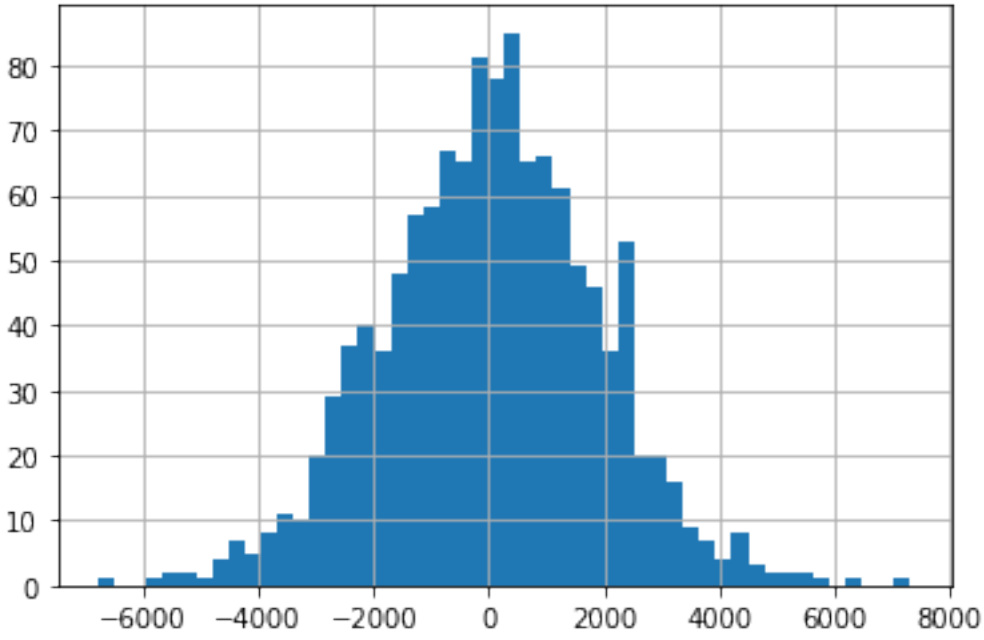
- Use the function `sciypy.optimize.curve_fit` with the variables `x_vals` and `frequencies`. **Hint:** you can use p0=[1000,1000**2.] as your initial parameters.
- Plot again the histogram of velocities, but also the result from the fit on the same plot.

[21]:
```
import pandas as pd
url='https://raw.githubusercontent.com/jmsevillam/LearningML/master/Problems/
 ↪Velocities.dat'
```

[22]:
```
data=pd.read_csv(url,delim_whitespace=True,header=None)
data.columns=['vx','vy','v']
```

[23]:
```
data.vx.hist(bins=50)
```

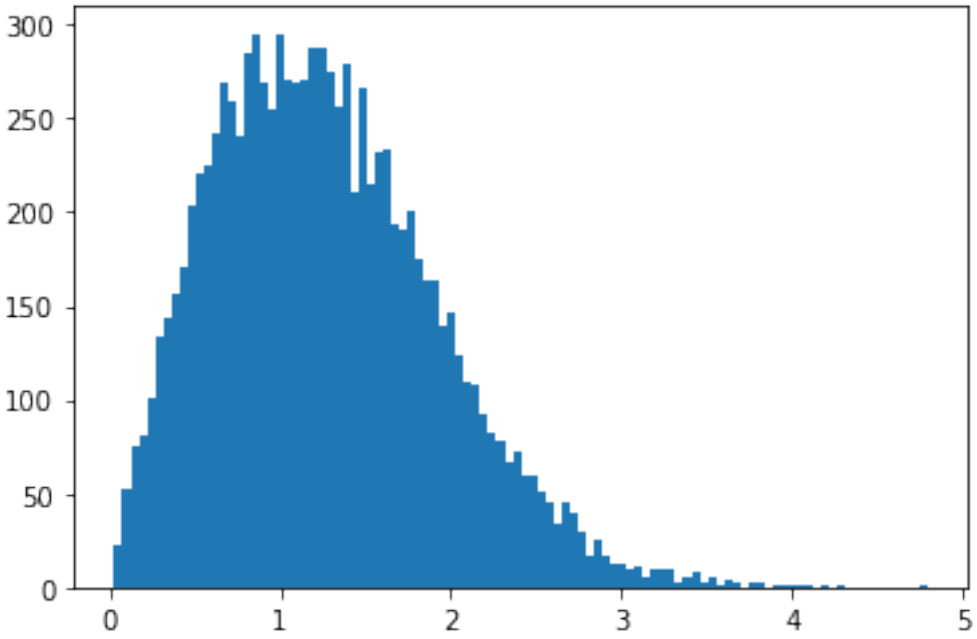
[23]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x12e79e730>
```



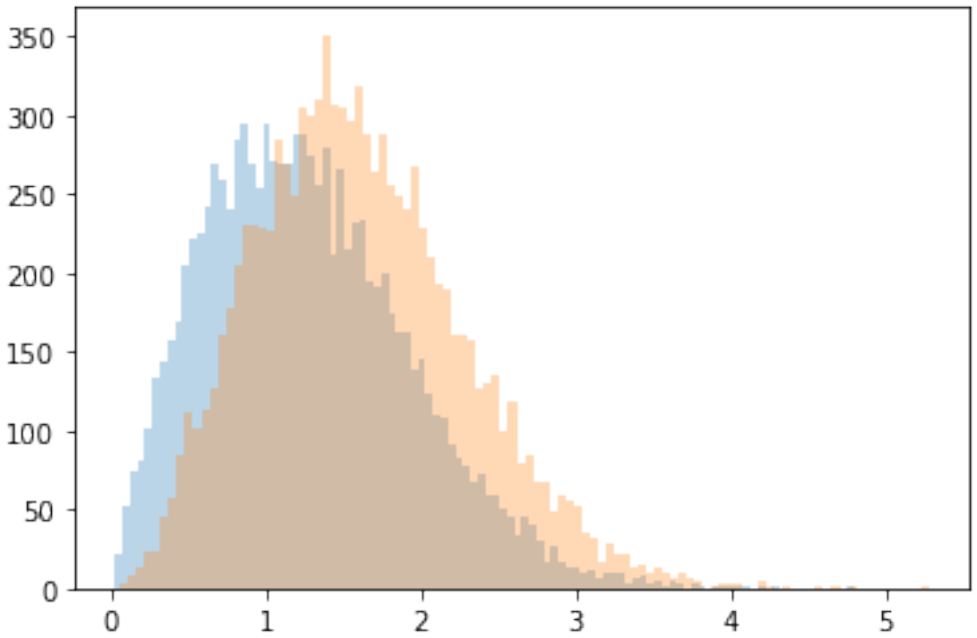
[24]:
```
vx=np.random.normal(0,1,10000)
vy=np.random.normal(0,1,10000)
vz=np.random.normal(0,1,10000)
```

[25]:
```
v=np.sqrt(vx**2+vy**2)
```

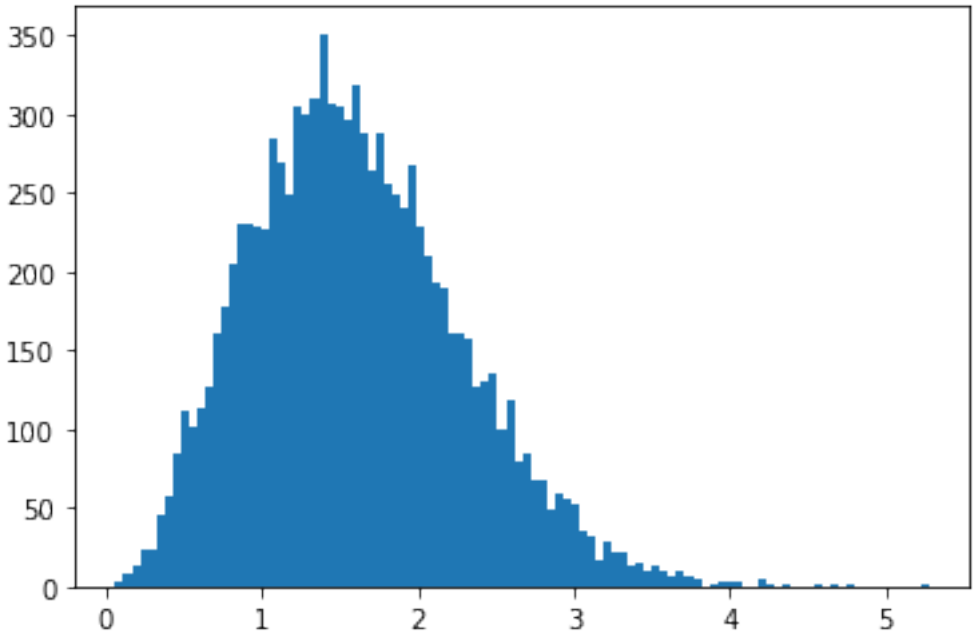
[26]:
```
plt.hist(v,bins=100);
```



[27]:
```
v2=np.sqrt(vx**2+vy**2+vz**2)
plt.hist(v,bins=100,alpha=0.3);
plt.hist(v2,bins=100,alpha=0.3);
```

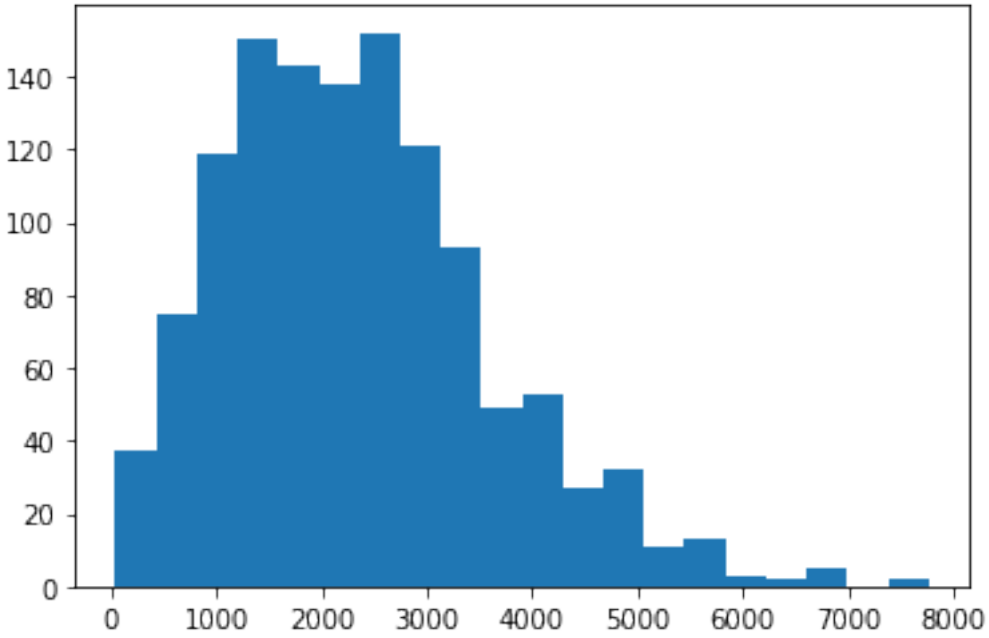


[28]:
```
plt.hist(v2,bins=100);
```



[29]:
```
v=data.v
```

[30]:
```
histogram=plt.hist(v,bins=20)
```

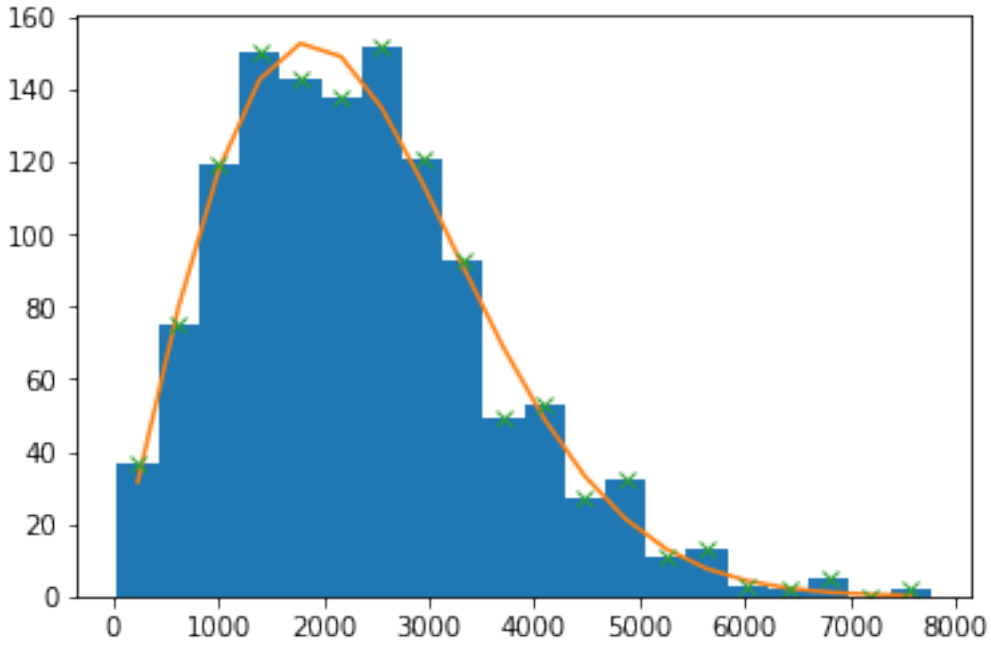


[31]:
```
x_vals =(histogram [1][1:]+ histogram [1][:-1])/2
frequencies = histogram [0]
```

[32]:
```
def Maxwell_Boltzmann(x,a,b):
    return a*x*np.exp(-x**2/b)
```

[33]:
```
popt,pcov=curve_fit(Maxwell_Boltzmann,x_vals,frequencies,p0=[1,1000**2.])
```

[34]:
```
plt.hist(v,bins=20);
plt.plot(x_vals,Maxwell_Boltzmann(x_vals,*popt))
plt.plot(x_vals,frequencies,'x');
```



[35]:
```
popt
```

[35]:
```
array([1.35689143e-01, 6.90520420e+06])
```

0.6 Central Limit Theorem

The central Limit Theorem is one of the reasons, Gaussians are so important.

It stands that under some general conditions, the distribution of the sum S_n of n independent random variables, can be approximated by a Gaussian even if the original variables themselves are not normally distributed.

That means that, some results of Gaussian distribution, can be used (In some sense), to other

distributions. We are going to construct a Gaussian distribution from the averages of uniform distributions.

We have to make a loop to generate the data. Use a `for` from `0` to `10000` (The number of Experiments we are having), and inside, create a set of random uniform variables (1000) and calculate the mean.

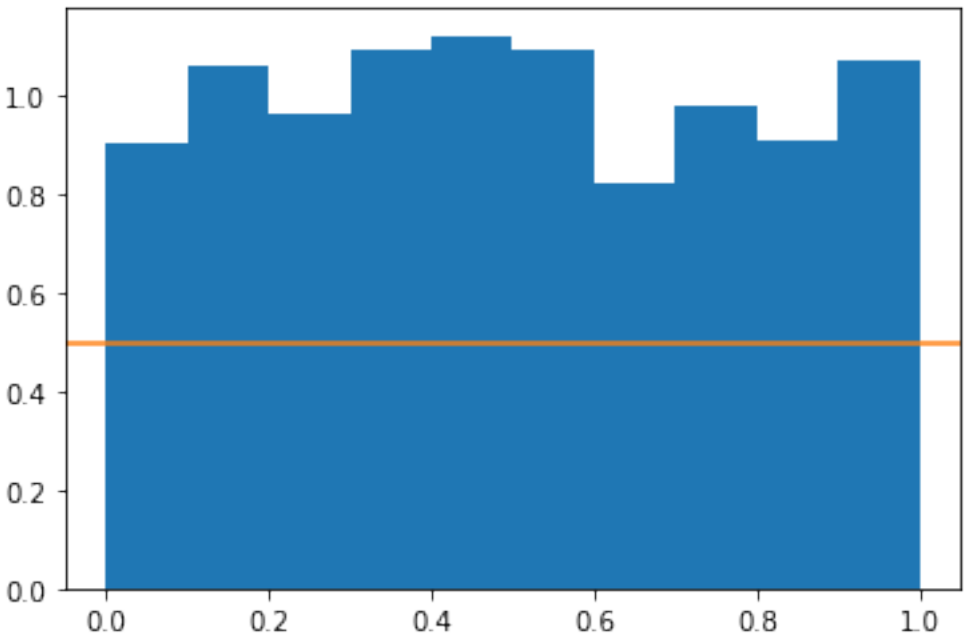
Hint: you can create a list or array outside the for, and use the append method, for example, if you choose a list you can use `data.append(np.random.random(1000).mean())`

- Plot the histogram of data.
- To be sure it is a Gaussian, define a function to make a fit (just like we did on the previous assignment.).
- Use the same strategy we saw before to do the fit generate the data to do the fit as,

```
histogram = plt. hist (data , bins =20)
x_vals =( histogram [1][1:]+ histogram [1][:-1])/2
frequencies = histogram [0]
```

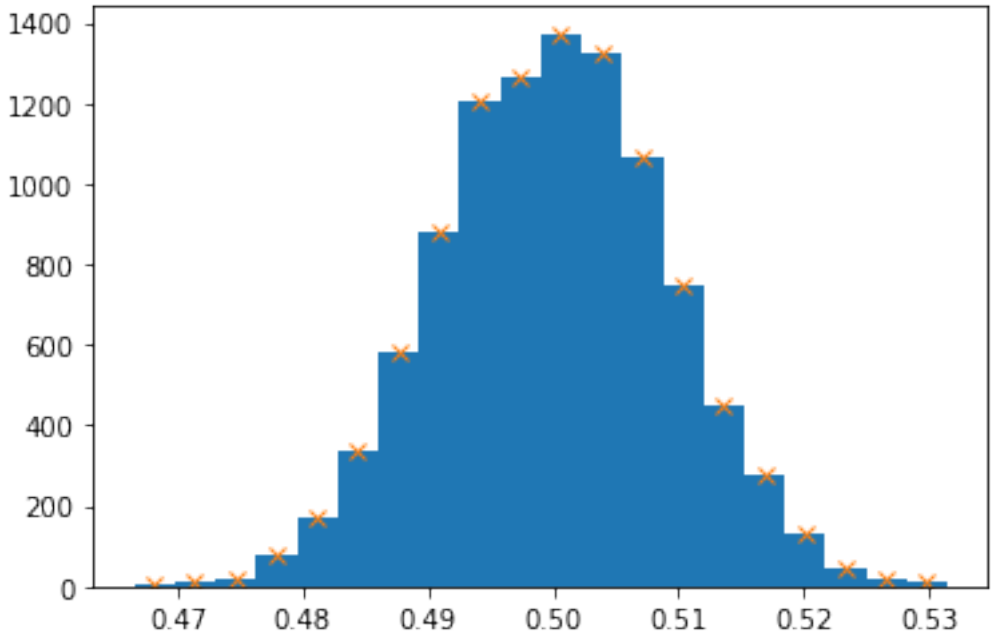
and use the function `scipy.optimize.curve_fit` to find the optimal parameters. - Plot again the histogram, but also the function fitted.

[36]:
```
x=np.random.random(1000)
hist=plt.hist(x,density=True);
plt.axhline(x.mean(),color='C1');
```



[37]:
```
means=[]
for i in range(10000):
    means.append(np.random.random(1000).mean())
```

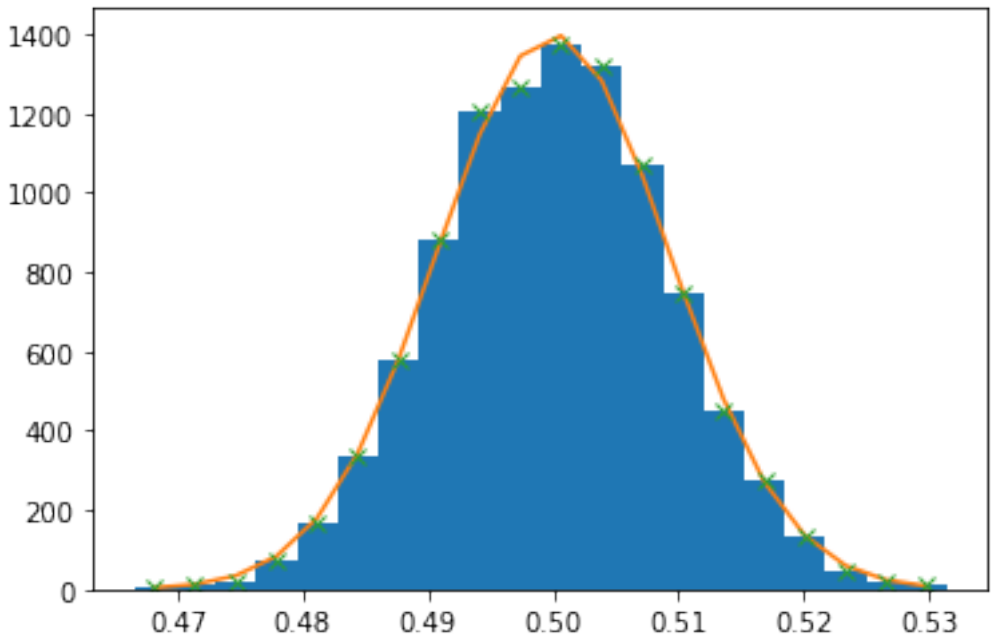
[38]:
```
histogram = plt. hist (means , bins =20)
x_vals =( histogram [1][1:]+ histogram [1][:-1])/2
frequencies = histogram [0]
plt.plot(x_vals,frequencies,'x');
```



[39]:
```
def Gauss(x,a,b,c):
    return a*np.exp(-c*(x-b)**2)
```

[40]:
```
popt,pcov=curve_fit(Gauss,x_vals,frequencies,p0=[1400,0.5,100])
```

[41]:
```
plt. hist (means , bins =20);
plt.plot(x_vals,Gauss(x_vals,*popt))
plt.plot(x_vals,frequencies,'x');
```



[42]:
```
popt
```

[42]:
```
array([1.39893199e+03, 4.99998143e-01, 5.76651594e+03])
```

[43]:
```
pcov.diagonal()**0.5
```

[43]:
```
array([1.54043369e+01, 1.18394774e-04, 1.46656046e+02])
```

0.7 Random Walks: Diffusion

Random walks can reproduce a diffusion process, to see that one has to take into account more than just one walker as it is measured on averages, we are going to work on two dimensions.

- As we are going to have more than one random walker, generate two arrays filled with zeros (Use 10000 components so that is the number of walkers we are going to use).
- Do a loop (1000 steps), and add a random normal number to each of the walkers
- Save, for each iteration, the standard deviation, you can use `np.var` and taking the square root.
- Plot the standard deviations (σ_x and σ_y) as a function of the step.
- A diffusion process is characterized because the standard deviation increase as the square root of the time, test this with a curve fit such as

$$f(t) = D\sqrt{t}$$

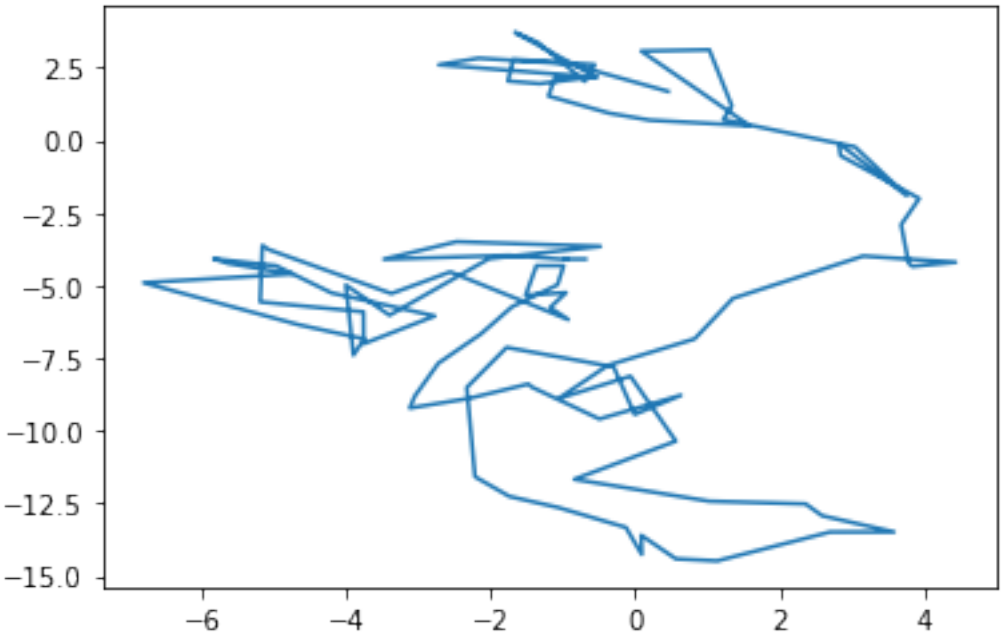
- The value of a is related with the so called, Diffusion constant which describes the speed of the diffusion process.

- You can plot the 2D histogram of `x` and `y` and see that the result is kind of a Gaussian, and we are measuring the rate this Gaussian spreads with the diffusion constant.

[44]:
```
y=np.random.normal(0,1,100).cumsum()
x=np.random.normal(0,1,100).cumsum()
```

[45]:
```
plt.plot(x,y)
```

[45]:
```
[<matplotlib.lines.Line2D at 0x12ec6cc70>]
```

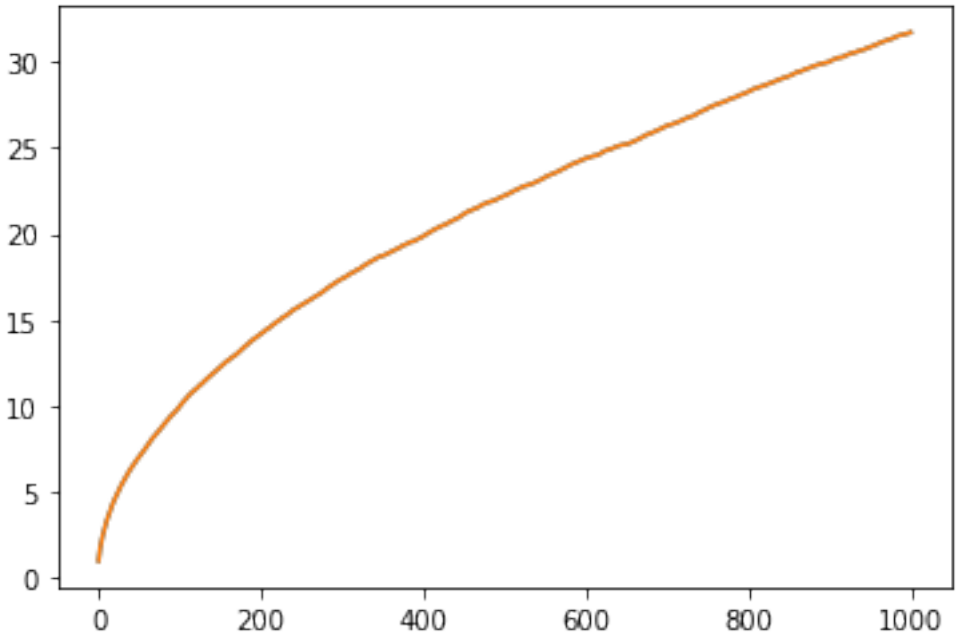


[46]:
```
x=np.zeros(10000)
y=np.zeros(10000)
```

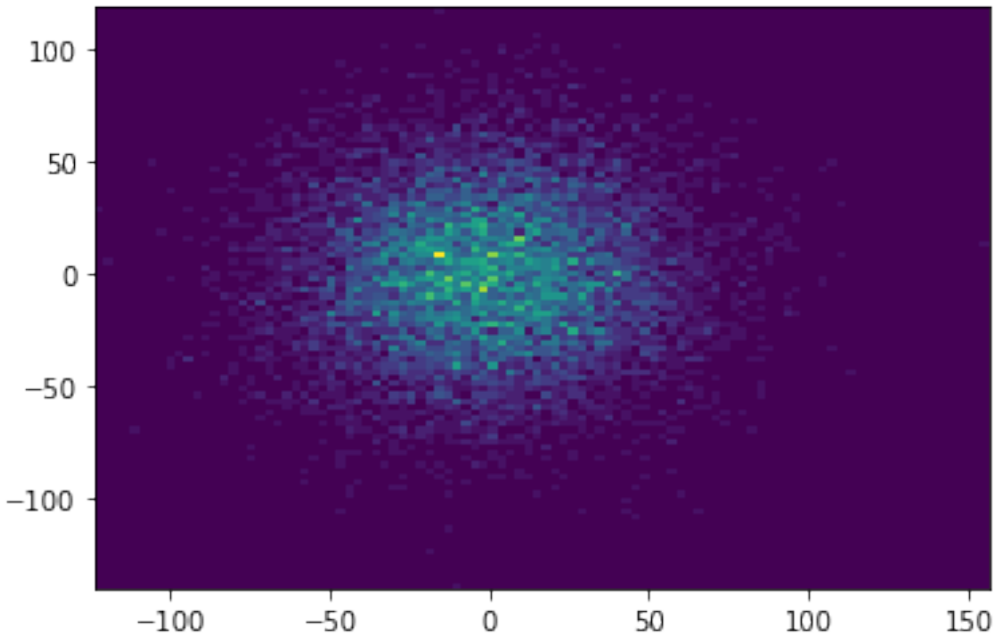
[47]:
```
x_std=[]
y_std=[]
for i in range(1000):
    x+=np.random.normal(0,1,10000)
    y+=np.random.normal(0,1,10000)

    x_std.append(x.var()**0.5)
    y_std.append(x.var()**0.5)
```

[48]:
```
plt.plot(x_std)
plt.plot(y_std)
```

[48]:
```
[<matplotlib.lines.Line2D at 0x12eb6b0d0>]
```



[49]:
```
plt.hist2d(x,y,bins=100);
```



0.8 Entropy - Information Theory

On information theory, Shannon defines the entropy as

$$S = -\sum_i p_i \log(p_i)$$

in other words as the expected value of $\log(p)$.

Using that definition, one can calculate the entropy of different probability distributions, so we are going to compare some.

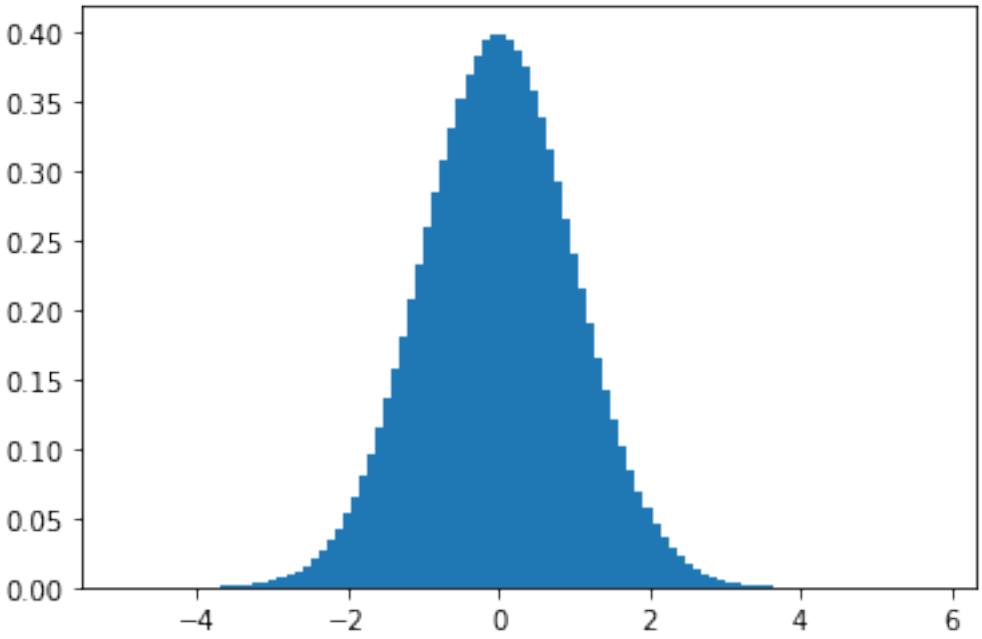
For each distribution, you have to

- Generate 1000000 points following the desire distribution.
- Calculate the histogram with 100 bins, you can do it with matplotlib such that `hist=plt.histogram(x,density=True,bins=100)` or with `numpy`
- Calculate the entropy of the frequencies, on my previous line would be hist. (You MUST normalize the histogram, so that take each frequency and divide it over the sum of all the frequencies. (Hence the sum is 1).) As it is possible that you have some bins with 0 data there, put a condition that only calculate if $p \neq 0$ (The logarithm diverges on 0).
- Save the result on a array-like structure.
- Repeat for at least 5 distributions, including
  - Uniform
  - Gaussian
  - Poisson
  - Choose the other two you want
- Plot the different entropies.
- Name the distribution with bigger entropy and discuss why is that particular one.

Hint: Entropy means the lack of information.

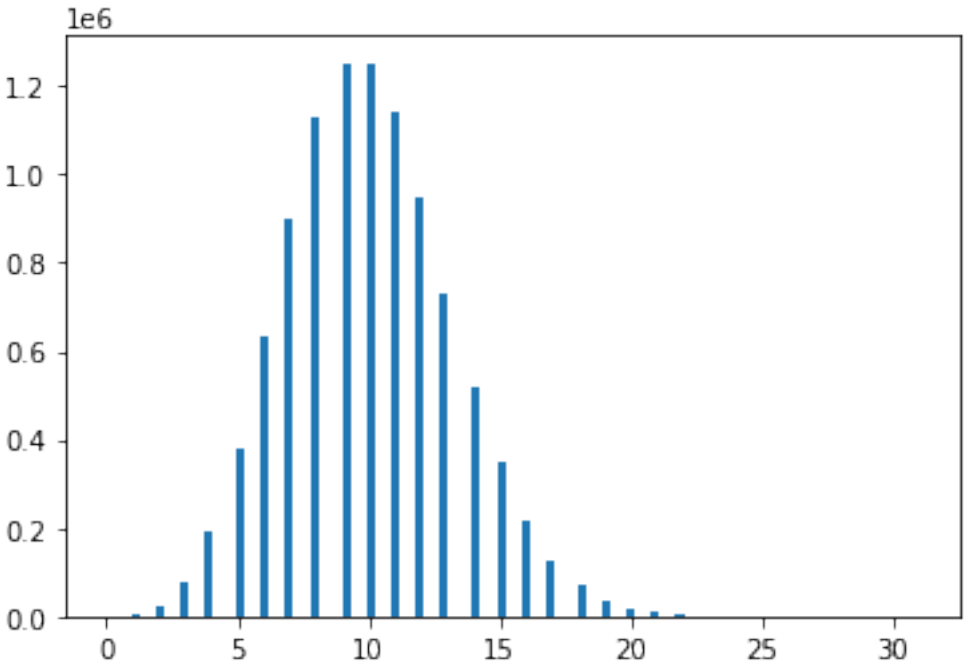
[50]:
```
def entropy(x):
    hist=np.histogram(x,density=True,bins=100)
    freq=hist[0]/hist[0].sum()
    freq[freq==0]=1
    return -(freq*np.log(freq)).sum()
```

[51]:
```
x=np.random.normal(0,1,10000000)
plt.hist(x,density=True,bins=100);
```

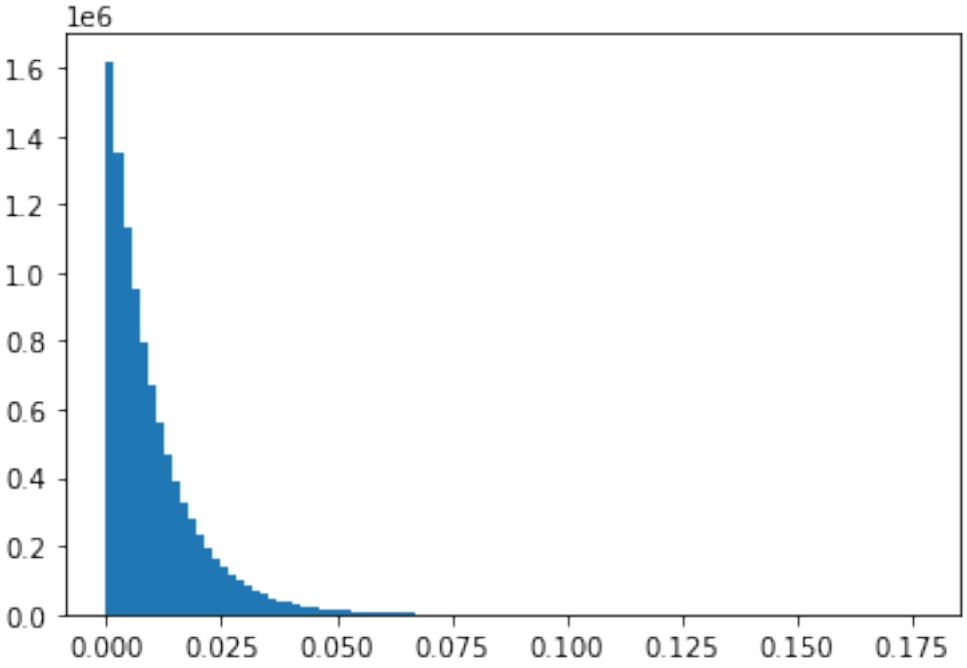


[52]:
```
x_normal=np.random.normal(0,1,10000000)
x_uni=np.random.random(10000000)
x_poisson=np.random.poisson(10,10000000)
x_pareto=np.random.pareto(100,10000000)
x_chi=np.random.chisquare(10,10000000)
```

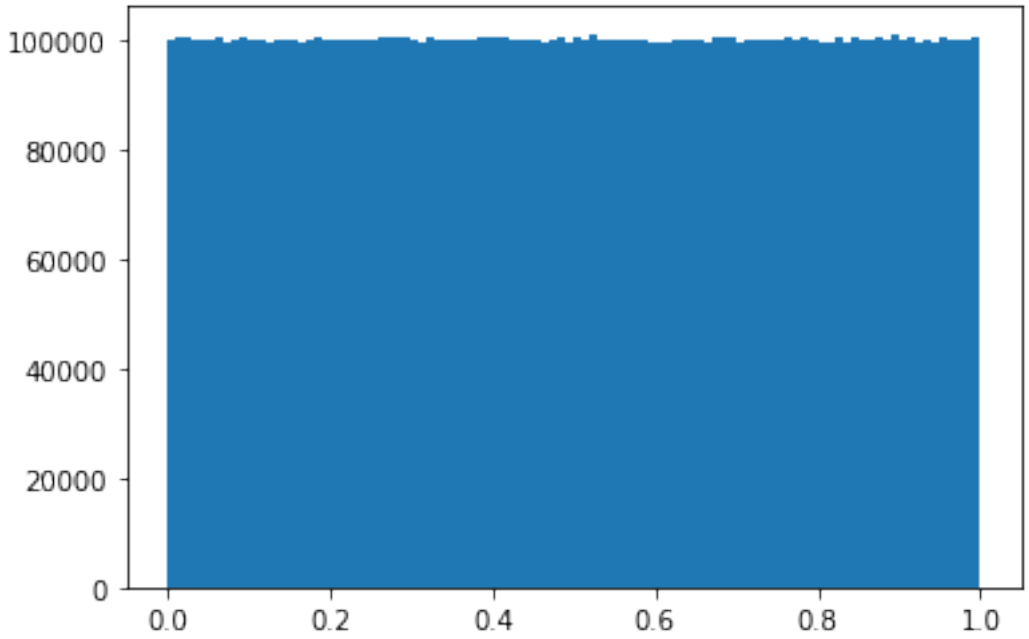
[53]:
```
plt.hist(x_poisson,bins=100);
```



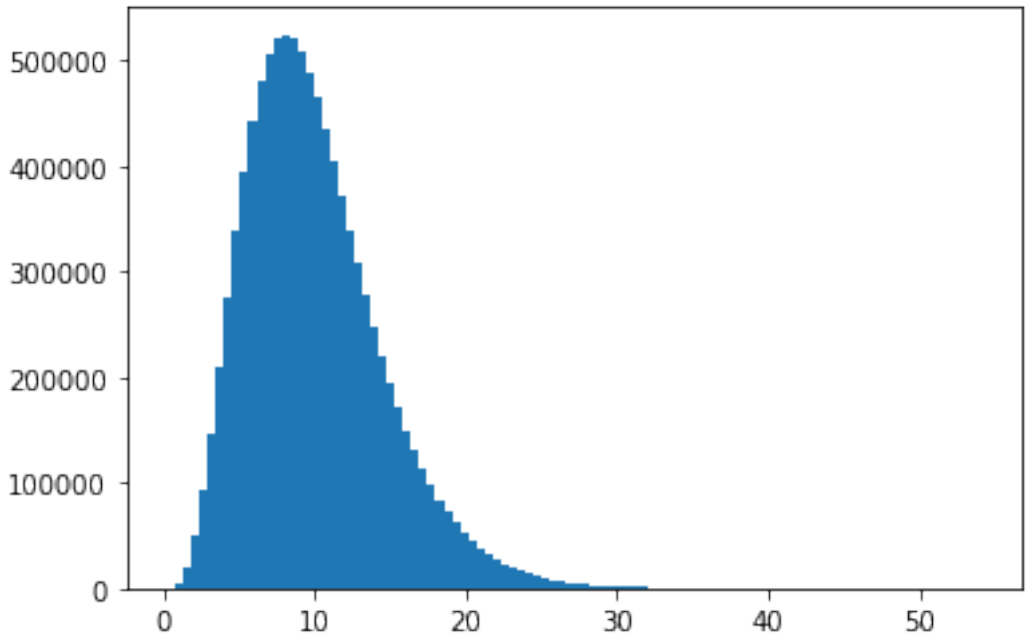
[54]:
```
plt.hist(x_pareto,bins=100);
```



[55]:
```
plt.hist(x_uni,bins=100);
```



[56]:
```
plt.hist(x_chi,bins=100);
```



[57]:
```
distros={'Normal':x_normal,'Uniform':x_uni,"Poisson":x_poisson,"Pareto":
↪x_pareto,"Chi":x_chi}
```

[58]:
```
for name in distros.keys():
    print(name,'\t',entropy(distros[name]))
```

```
Normal   3.6633018681239853
Uniform          4.605165331453245
Poisson          2.5615612082323307
Pareto   2.7446666531134287
Chi      3.4670887811024396
```