Siddhanth Kumar Section 102

Workshop 5: PDF sampling and Statistics

Submit this notebook to bCourses to receive a grade for this Workshop.

Please complete workshop activities in code cells in this iPython notebook. The activities titled **Practice** are purely for you to explore Python, and no particular output is expected. Some of them have some code written, and you should try to modify it in different ways to understand how it works. Although no particular output is expected at submission time, it is *highly* recommended that you read and work through the practice activities before or alongside the exercises. However, the activities titled **Exercise** have specific tasks and specific outputs expected. Include comments in your code when necessary.

The workshop should be submitted on bCourses under the Assignments tab (both the .ipynb and .pdf files).

Preview: generating random numbers

We will discuss simulations in greater detail later in the semester. The first step in simulating nature -- which, despite Einstein's objections, is playing dice after all -- is to learn how to generate some numbers that appear random. Of course, computers cannot generate true random numbers -- they have to follow an algorithm. But the algorithm may be based on something that is difficult to predict (e.g. the time of day you are executing this code) and therefore *look* random to a human. Sequences of such numbers are called *pseudo-random*.

The random variables you generate will be distributed according to some *Probability Density Function* (PDF). The most common PDF is *flat*: $f(x) = \frac{1}{b-a}$ for $x \in [a..b)$. Here is how to get a random number uniformly distributed between $a = 0$ and $b = 1$ in Python:

In [2]:
```
# standard preamble
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [3]:
```
# generate one random number between [0,1)
x = np.random.rand()
print ('x=', x)

# generate an array of 10 random numbers between [0,1)
array = np.random.rand(10)
print (array)
```

```
x= 0.7102877723574138
[0.80156765 0.20555273 0.21508382 0.45922236 0.40847869 0.37404484
 0.83293712 0.13683539 0.73725692 0.68049659]
```

You can generate a set of randomly-distributed integer values instead:

In [4]:
```
a = np.random.randint(0,1000,10)
print(a)
```

```
[150 333 817 562 940 280 840 129 279 853]
```

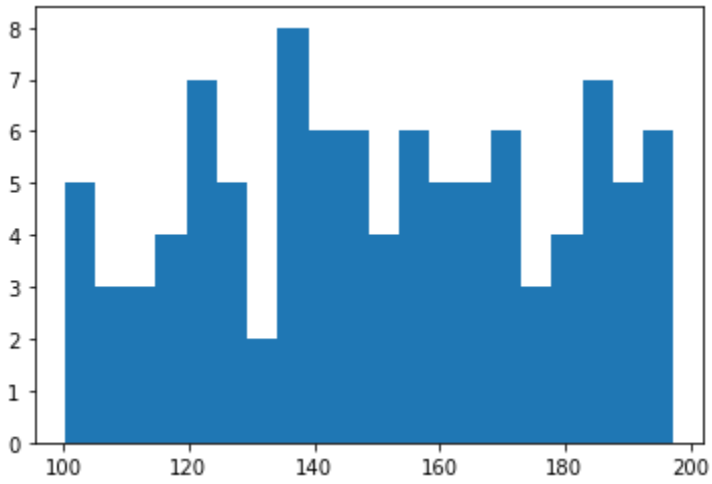
1d distributions

Moments of the distribution

Python's SciPy library contains a set of standard statistical functions. See a few examples below:

In [5]:
```
# create a set of data and compute mean and variance
# This creates an array of 100 elements, uniformly-distributed between 100 and 200

# Try changing the size parameter!
x = np.random.uniform(low=100,high=200,size=100)

print(x[0:10])
# make a histogram
n, bins, patches = plt.hist(x, 20)

# n is y value, bins is x value, patches is how many bars

# various measures of "average value":
print('Mean = {0:5.0f}'.format(np.mean(x)))
print( 'Median = {0:5.0f}'.format(np.median(x)))

# measure of the spread
print('Standard deviation = {0:5.1f}'.format(np.std(x)))
```

```
[127.13196052 164.58456645 161.44203494 175.21183457 123.85055681
 134.38496416 195.99570703 105.09223807 110.46273378 171.5359308 ]
Mean =   151
Median =   151
Standard deviation =  27.5
```



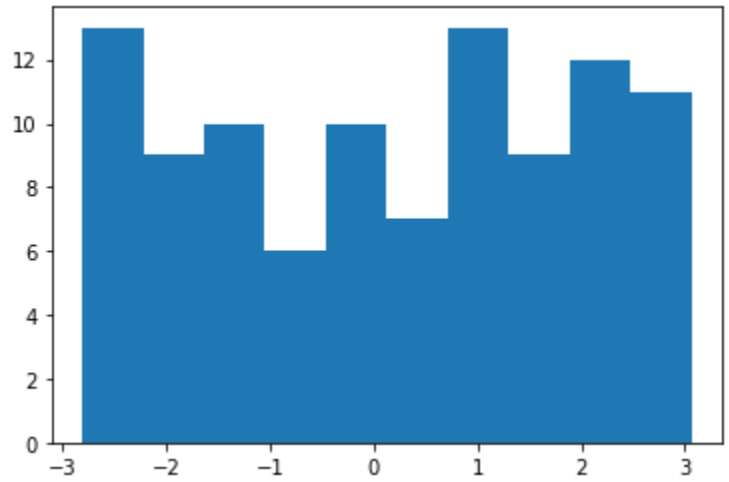
Exercise 1

We just introduced some new functions: `np.random.rand()`, `np.random.uniform()`, `plt.hist()`, `np.mean()`, and `np.median()`. So let's put them to work. You may also find `np.cos()`, `np.sin()`, and `np.std()` useful.

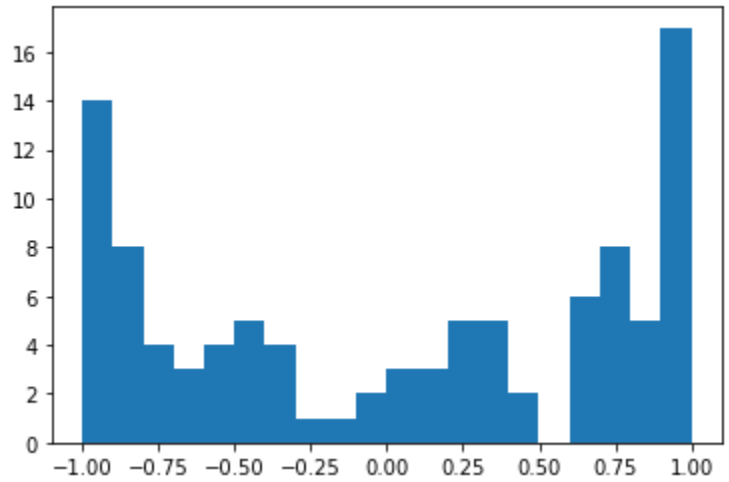
1. Generate 100 random numbers, uniformly distributed between [-π, π)
2. Plot them in a histogram.
3. Compute mean and standard deviation (RMS)
4. Plot a histogram of sin(x) and cos(x), where x is a uniformly distributed random number between [-π,π). Do you understand this distribution ?

In [6]:
```
# Your code for Exercise 1

negPI = -np.pi
PI = np.pi

x = np.random.uniform(low=negPI, high=PI, size=100)

n, bins, patches = plt.hist(x)



# n is frequency, bins is actual number, patches is number of rectangles

print('Mean: {:0.5f}'.format(np.mean(x)))
print('Standard deviation {:0.5f}'.format(np.std(x)))

plt.show()

n1, bins1, patches1 = plt.hist(np.sin(x), 20)
print('Mean: {:0.5f}'.format(np.mean(bins1)))
print('Standard deviation {:0.5f}'.format(np.std(bins1)))
plt.show()

n2, bins2, patches2 = plt.hist(np.cos(x), 20)
print('Mean: {:0.5f}'.format(np.mean(bins2)))
print('Standard deviation {:0.5f}'.format(np.std(bins2)))
```

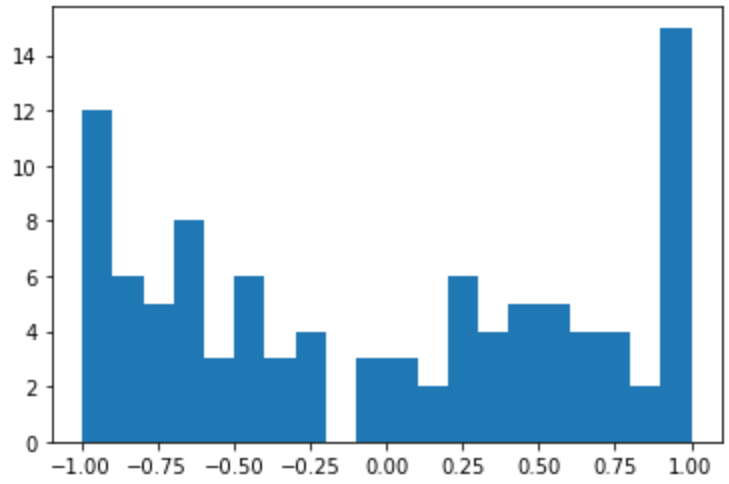
```
Mean: 0.17253
Standard deviation 1.79301
```




```
Mean: -0.00017
Standard deviation 0.60508
```




```
Mean: 0.00163
Standard deviation 0.60447
```



Gaussian/Normal distribution

You can also generate Gaussian-distributed numbers. Remember that a Gaussian (or Normal) distribution is a probability distribution given by

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

where μ is the average of the distribution and σ is the standard deviation. The **standard** normal distribution is a special case with $\mu = 0$ and $\sigma = 1$.

In [7]:
```
# generate a single random number, gaussian-distributed with mean=0 and sigma=1. This is
# a standard normal distribution
x = np.random.standard_normal()
print (x)

# generate an array of 10 such numbers
a = np.random.standard_normal(size=10)
print (a)
```

```
0.1645036847868933
[-0.15010606 -0.67331247 -0.24749219  0.26232335 -1.5982158  -0.3947012
  0.52973837 -0.4938953   1.02291358 -0.55656857]
```

Exercise 2

We now introduced `np.random.standard_normal()`.

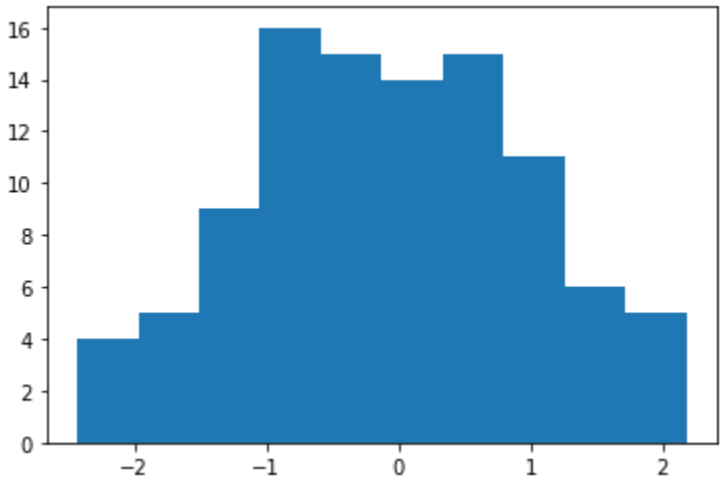
1. Generate $N = 100$ random numbers, Gaussian-distributed with $\mu = 0$ and $\sigma = 1$.
2. Plot them in a histogram.
3. Compute the mean, standard deviation (RMS), and standard error on the mean.

The standard error on the mean is defined as $\sigma_\mu = \frac{\sigma}{\sqrt{N}}$, where σ is the standard deviation.

In [8]:
```
# Your code for Exercise 2

def standard_error(arr, N):
    value = np.std(arr) / np.sqrt(N)
    return value

x = np.random.standard_normal(size=100)
```

```
plt.hist(x)

print('Mean: {:0.5f}'.format(np.mean(x)))
print('Standard deviation {:0.5f}'.format(np.std(x)))
print('Standard error {:0.5f}'.format(standard_error(x, 100)))
```

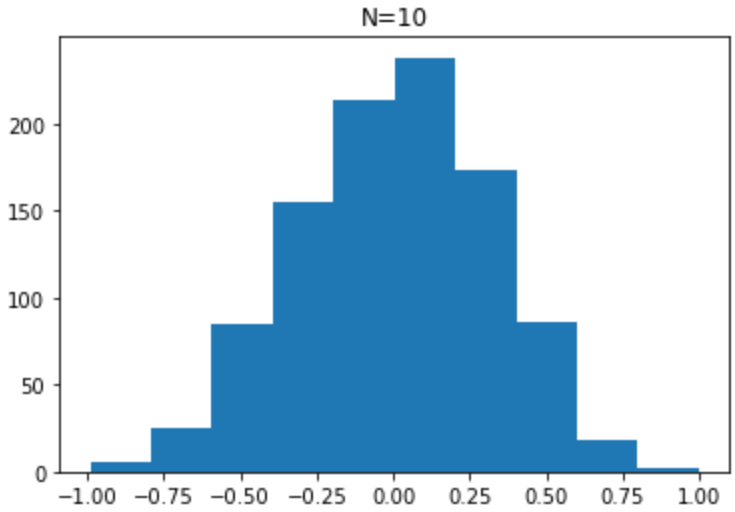
```
Mean: -0.07278
Standard deviation 1.04314
Standard error 0.10431
```

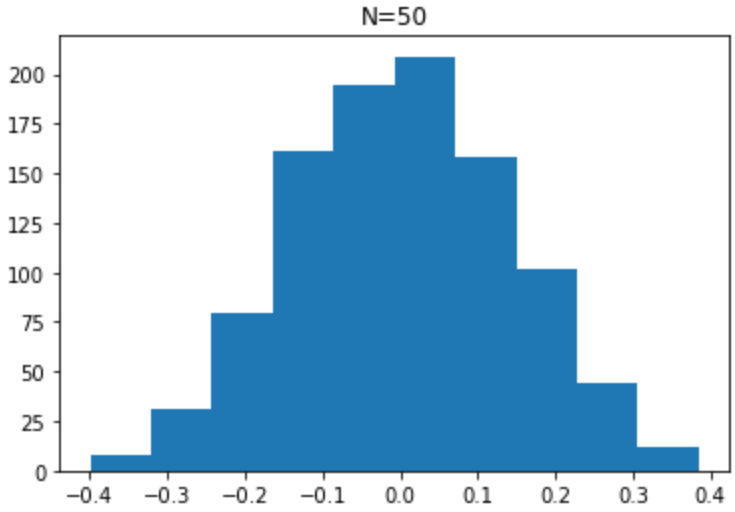


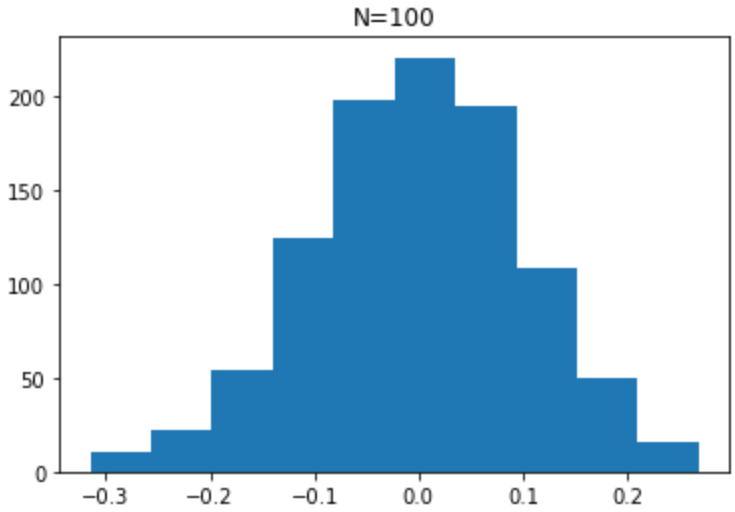
1. Now find the means of $M = 1000$ experiments of $N = 100$ measurements each (you'll end up generating 100,000 random numbers total). Plot a histogram of the means. Is it consistent with your calculation of the error on the mean for $N = 100$? About how many experiments yield a result within $1\sigma_\mu$ of the true mean of 0 ? About how many are within $2\sigma_\mu$?
2. Now repeat question 4 for $N = 10, 50, 1000, 10000$. Plot a graph of the RMS of the distribution of the means vs N. Is it consistent with your expectations ?

In [63]:
```
#Q4 Below

def standard_error(arr, N):
    value = np.std(arr) / np.sqrt(N)
    return value

stds = []
Ns = ['10','50','100','1000','10000',]

def experiment(M, N):
    means = []
    for i in range(M):
        x = np.random.standard_normal(size=N)
        means.append(np.mean(x))
    stds.append(np.std(means))
    return plt.hist(means)


plt.title("N=10")
n1, bins1, patches1 = experiment(1000, 10)
plt.show()

plt.title("N=50")
n2, bins2, patches2 = experiment(1000, 50)
plt.show()

plt.title('N=100')
n, bins, patches = experiment(1000, 100)
plt.show()
```

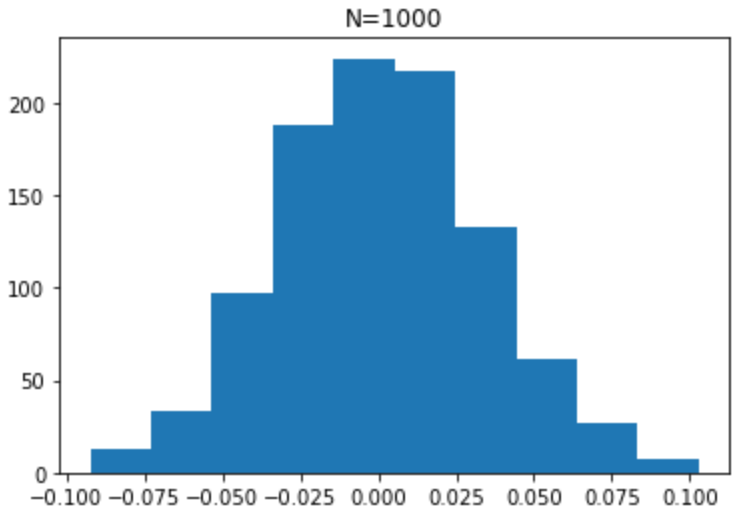
```
print("This graph and the previous one are different, more experiments means more accura
print("Around 600 are within one std and around 900 are within 2 std")

plt.title("N=1000")
n3, bins3, patches3 = experiment(1000, 1000)
plt.show()

plt.title("N=10000")
n4, bins4, patches4 = experiment(1000, 10000)
plt.show()

print(Ns)
print(stds)

plt.bar(Ns, stds)
plt.xlabel("N")
plt.ylabel("RMS")
plt.title('RMS (Actual)')
plt.show()

print('N is the size of the dataset, as the size grows the deviation in the set decrease
```

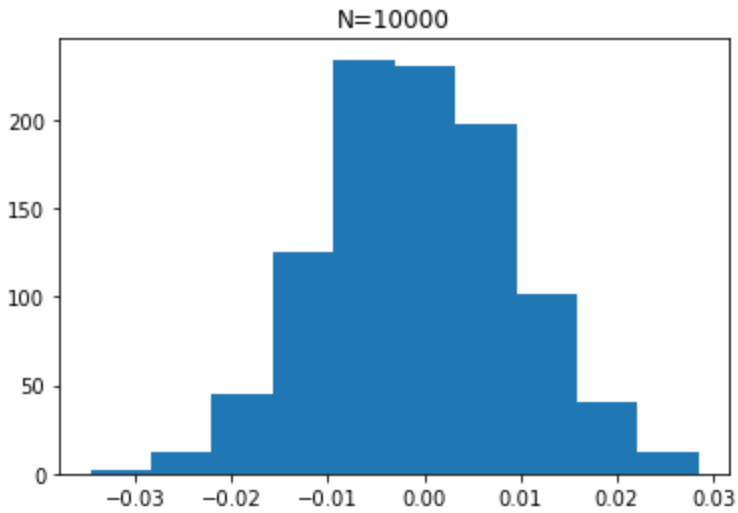




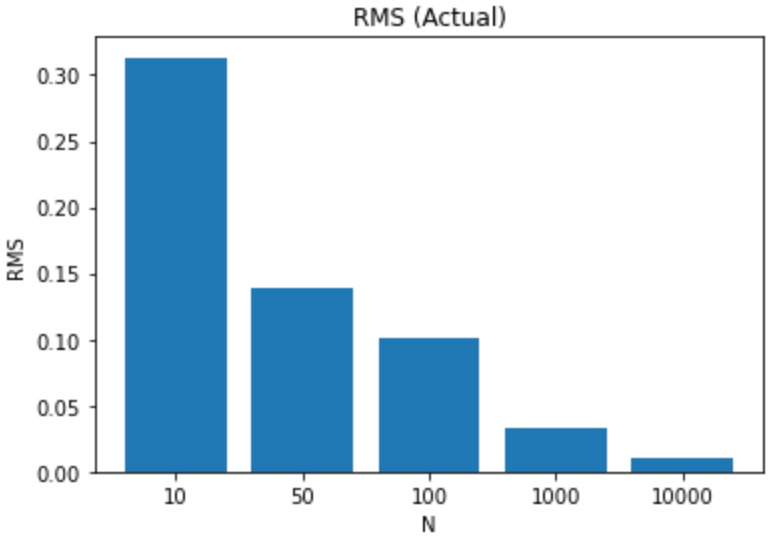


This graph and the previous one are different, more experiments means more accurate dat
a.
Around 600 are within one std and around 900 are within 2 std





['10', '50', '100', '1000', '10000']
[0.31327949017263423, 0.1395719505080962, 0.10116968908891387, 0.03289727297632446, 0.00
9941002832611697]



N is the size of the dataset, as the size grows the deviation in the set decreases

Exponential distribution

In this part we will repeat the above process, but now using lists of exponentially distributed random numbers. The probability of selecting a random number between x and $x + dx$ is $\propto e^{-x} dx$. Exponential distributions often appear in lossy systems, e.g. if you plot an amplitude of a damped oscillator as a function of time. Or you may see it when you plot the number of decays of a radioactive isotope as a function of time.

In [10]:
```
# generate a single random number, exponentially-distributed with scale=1.
x = np.random.exponential()
print (x)

# generate an array of 10 such numbers
a = np.random.exponential(size=10)
print (a)
```

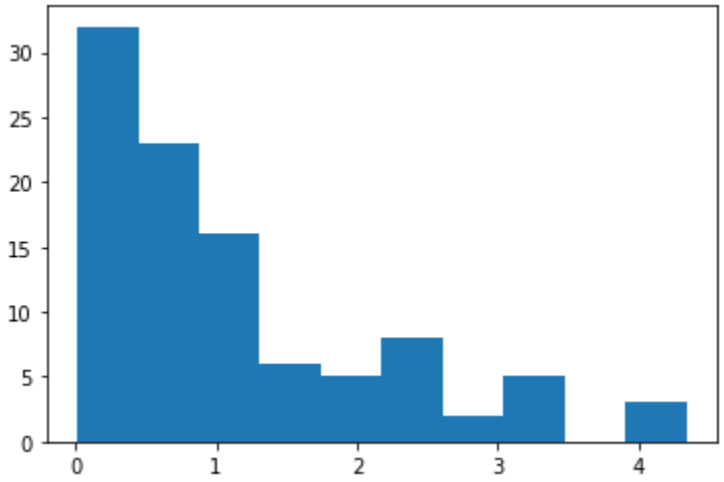
```
1.7471191117017497
[0.55153832 0.30813636 1.38517682 0.07049012 3.8741695  0.11931386
 1.06990108 0.91809289 0.04802045 1.4480876 ]
```

Exercise 3

We now introduced `np.random.exponential()`. This function can take up to two keywords, one of which is `size` as shown above. The other is `scale`. Use the documentation and experiment with this exercise to see what it does.

1. What do you expect to be the mean of the distribution? What do you expect to be the standard deviation?
2. Generate $N = 100$ random numbers, exponentially-distributed with the keyword `scale` set to 1.
3. Plot them in a histogram.
4. Compute mean, standard deviation (RMS), and the error on the mean. Is this what you expected?
5. Now find the means, standard deviations, and errors on the means for each of the $M = 1000$ experiments of $N = 100$ measurements each. Plot a histogram of each quantity. Is the RMS of the distribution of the means consistent with your calculation of the error on the mean for $N = 100$?
6. Now repeat question 5 for $N = 10, 100, 1000, 10000$. Plot a graph of the RMS of the distribution of the means vs N. Is it consistent with your expectations ? This is a demonstration of the *Central Limit Theorem*

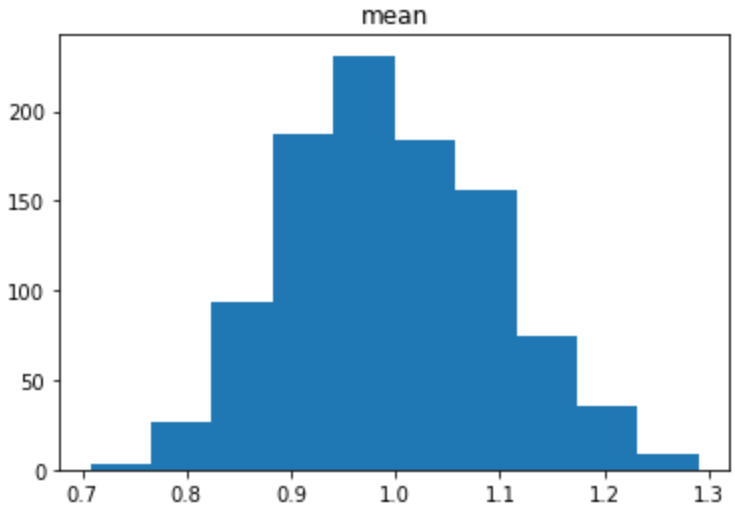
In [11]:
```
# Your code for Exercise 3

def standard_error(arr, N):
    value = np.std(arr) / np.sqrt(N)
    return value

x = np.random.exponential(size=100, scale=1)
n, bins, patches = plt.hist(x)
plt.show()

print('Mean: {:0.5f}'.format(np.mean(bins)))
print('Standard deviation {:0.5f}'.format(np.std(bins)))
print('Standard error {:0.5f}'.format(standard_error(bins, 100)))
```

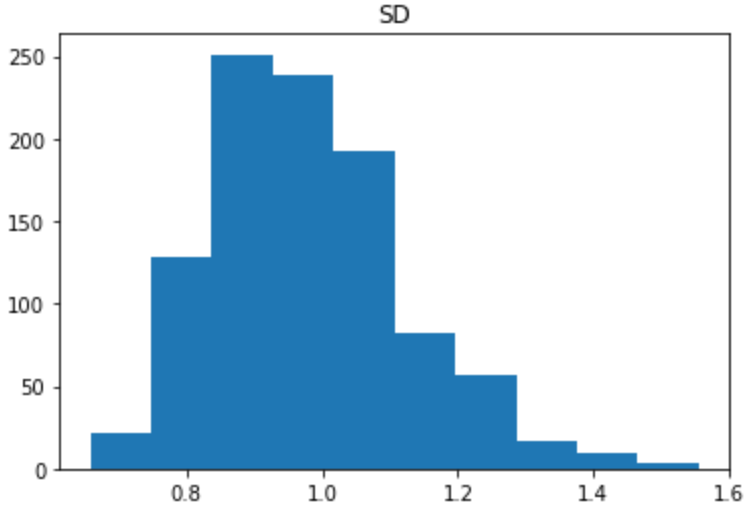


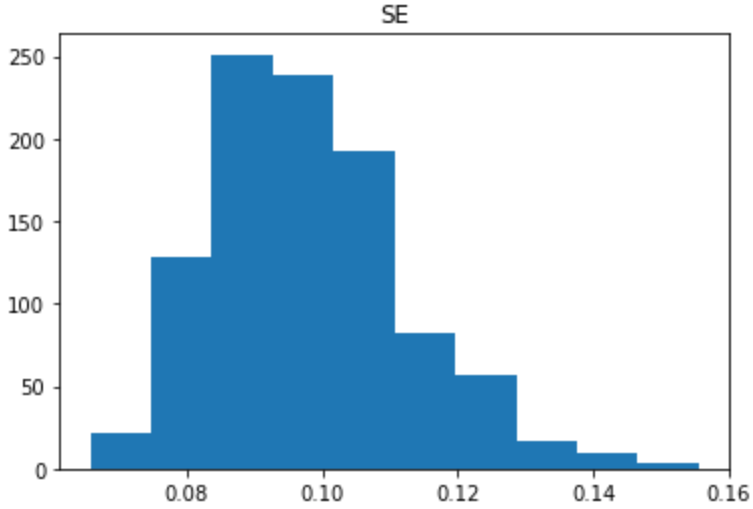

```
Mean: 2.17509
Standard deviation 1.36578
Standard error 0.13658
```

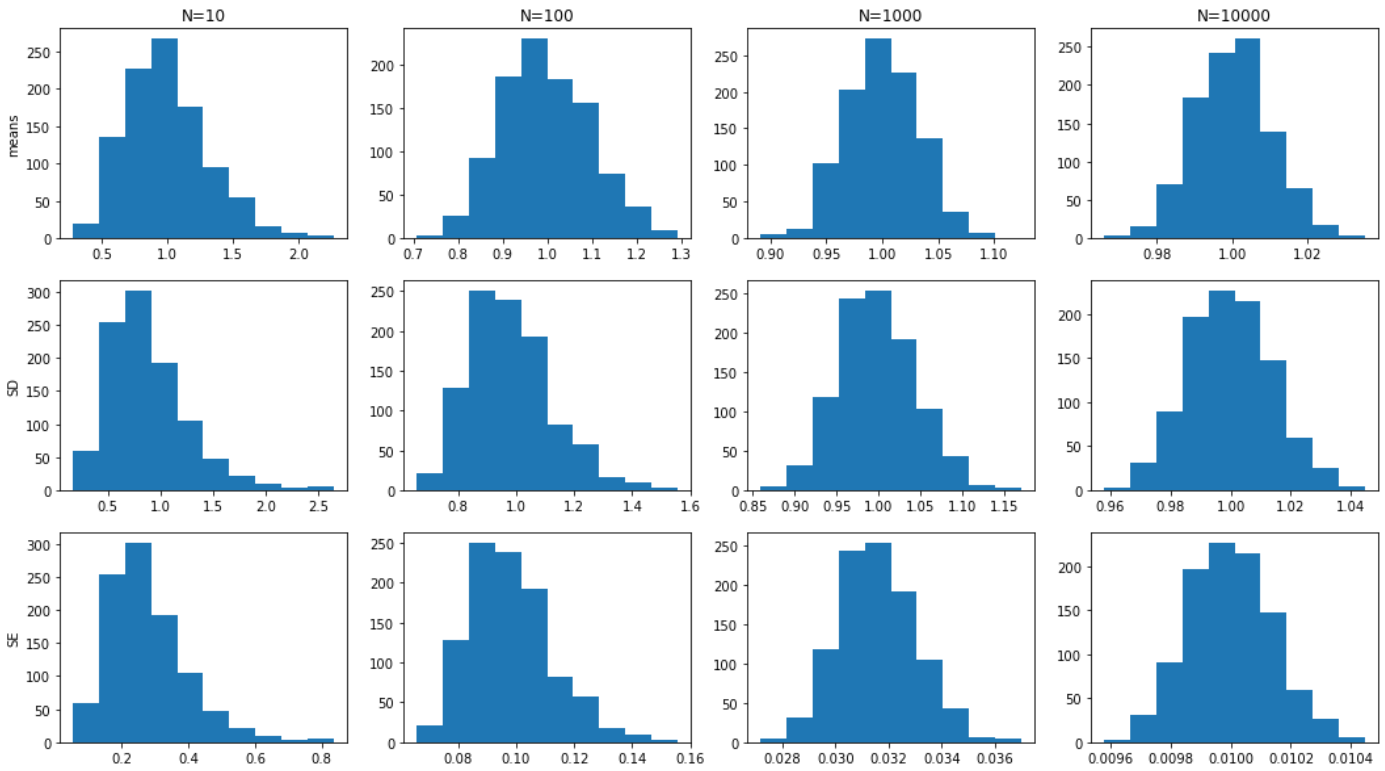
In [56]:
```
#Q5

def standard_error(arr, N):
    value = np.std(arr) / np.sqrt(N)
    return value

meanSTDs = []
Ns = []

def experiment(M, N):
    stds = []
    stdError = []
    means = []
    for i in range(M):
        x = np.random.exponential(size=N, scale=1)
        means.append(np.mean(x))
        stds.append(np.std(x))
        stdError.append(standard_error(x, N))
    meanSTDs.append(np.std(means))
    Ns.append(N)
    return stds, means, stdError

stds100, means100, stdError100 = experiment(1000, 100)

plt.title('mean')
plt.hist(means100)
plt.show()

plt.title('SD')
plt.hist(stds100)
plt.show()
```

```
plt.title('SE')
plt.hist(stdError100)
plt.show()


stds10, means10, stdError10 = experiment(1000, 10)



stds1000, means1000, stdError1000 = experiment(1000, 1000)


stds10000, means10000, stdError10000 = experiment(1000, 10000)


fig, ax = plt.subplots(3, 4, figsize=(18, 10))

ax[0,0].set_ylabel('means')
ax[1,0].set_ylabel('SD')
ax[2,0].set_ylabel('SE')

ax[0,0].set_title('N=10')
ax[0,1].set_title("N=100")
ax[0,2].set_title("N=1000")
ax[0,3].set_title("N=10000")

ax[0,0].hist(means10)
ax[1,0].hist(stds10)
ax[2,0].hist(stdError10)

ax[0,1].hist(means100)
ax[1,1].hist(stds100)
ax[2,1].hist(stdError100)

ax[0,2].hist(means1000)
ax[1,2].hist(stds1000)
ax[2,2].hist(stdError1000)

ax[0,3].hist(means10000)
ax[1,3].hist(stds10000)
ax[2,3].hist(stdError10000)



fig.show()

#plt.plot(Ns, meanSTDs)
#plt.show()
```


mean
200
150
100
50
0
0.7
0.8
0.9
1.0
1.1
1.2
1.3


SD
250
200
150
100
50
0
0.8
1.0
1.2
1.4
1.6


SE
250
200
150
100
50
0
0.08
0.10
0.12
0.14
0.16



Binomial distribution

The binomial distribution with parameters n and p is the *discrete* probability distribution of the number of successes in a sequence of n independent yes/no experiments, each of which yields success with probability p. A typical example is a distribution of the number of *heads* for n coin flips ($p = 0.5$)

In [78]:
```
# Simulates flipping 1 fair coin  one time. Returns 0 for heads and 1 for tails
p = 0.5
print (np.random.binomial(1,p))

# Simulates flipping 5 biased coins three times
p = 0.7
print (np.random.binomial(5,p, size=50))
```

```
0
[5 4 3 4 3 4 3 4 3 4 2 3 4 4 2 4 2 3 3 2 3 1 4 2 3 2 3 5 3 2 4 4 2 5 4 3 3
 4 5 4 4 5 4 4 3 4 2 4 2 4]
```

Exercise 4

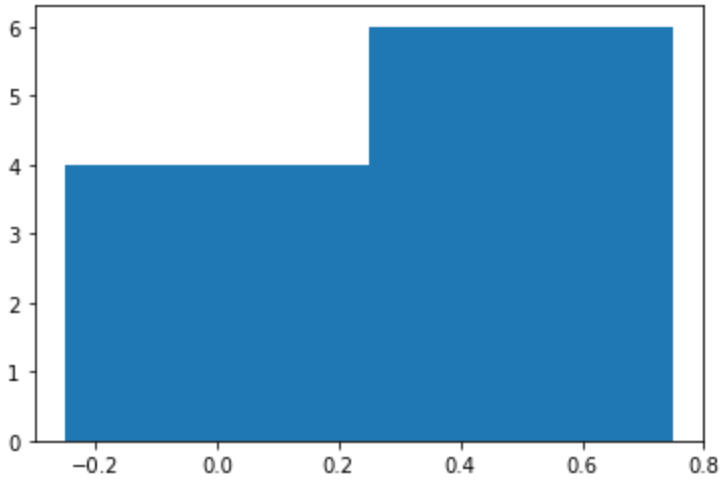
We now introduced the function `np.random.binomial(n,p)` which requires two arguments, `n` the number of coins being flipped in a single trial and `p` the probability that a particular coin lands tails. As usual, `size` is another optional keyword argument.

1. Generate an array of outcomes for flipping 1 unbiased coin 10 times.
2. Plot the outcomes in a histogram (0=heads, 1=tails).
3. Compute mean, standard deviation (RMS), and the error on the mean. Is this what you expected?

In [58]:
```
# Your code for Exercise 4
import numpy as np
import matplotlib.pyplot as plt

def standard_error(arr, N):
    value = np.std(arr) / np.sqrt(N)
```

```
    return value

x = np.random.binomial(1, 0.5, size=10)
n, bins, patches = plt.hist(x,bins=2, align='left')

print(bins)

print('Mean: {:0.4f}'.format(np.mean(x)))
print('Standard deviation {:0.5f}'.format(np.std(x)))
print('Standard error {:0.5f}'.format(standard_error(x, 10)))
```

```
[0.  0.5 1. ]
Mean: 0.6000
Standard deviation 0.48990
Standard error 0.15492
```



Poisson distribution

The Poisson distribution is a *discrete* probability distribution that expresses the probability of a given number of events n occurring in a fixed interval of time T if these events occur with a known average rate ν/T and independently of the time since the last event. The *expectation value* of n is ν. The variance of n is also ν, so the standard deviation of n is $\sigma(n) = \sqrt{\nu}$

In [12]:
```
nu = 10 # expected number of events
n = np.random.poisson(nu)  # generate a Poisson-distributed number.
print (n)
```

```
7
```

Exercise 5

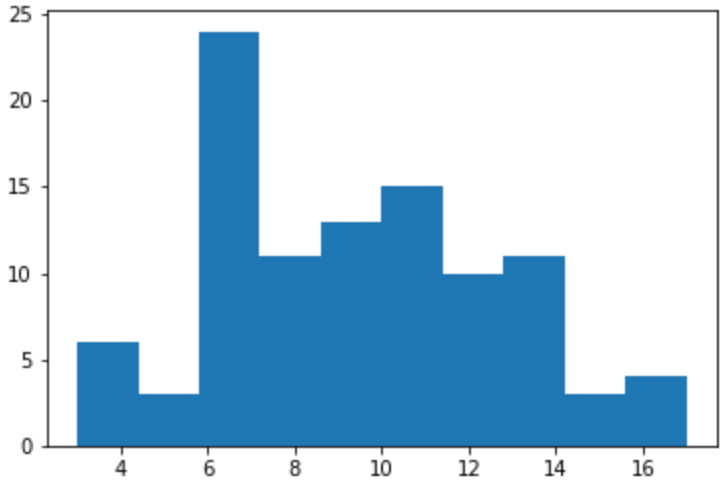
We introduced `np.random.poisson()`. As usual, you can use the keyword argument `size` to draw multiple samples.

1. Generate $N = 100$ random numbers, Poisson-distributed with $\nu = 10$.
2. Plot them in a histogram.
3. Compute mean, standard deviation (RMS), and the error on the mean. Is this what you expected?
4. Now repeat question 3 for $\nu = 1, 5, 100, 10000$. Plot a graph of the RMS vs ν. Is it consistent with your expectations ?

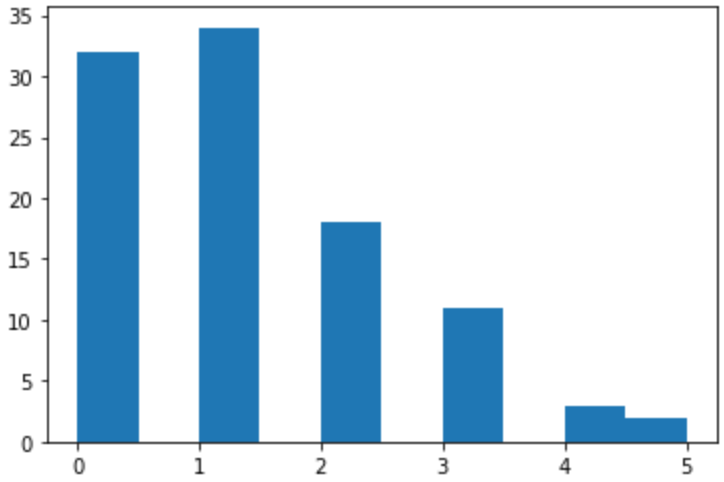
In [19]:
```
# Your code for Exercise 5

Vs = []
stds = []
```

```
def poisson(v):
    x = np.random.poisson(v, size=100)

    n, bins, patches = plt.hist(x)

    print('Mean: {0:0.4f} /// for nu = {1:0f}'.format(np.mean(x), v))
    print('Standard deviation: {0:0.5f} /// for nu = {1:.0f}'.format(np.std(x), v))
    print('Standard error: {0:0.5f} /// for nu = {1:.0f}'.format(standard_error(x, 10),
    Vs.append(v)
    stds.append(np.std(x))
    plt.show()
    return n, bins, patches

poisson(10)
poisson(1)
poisson(5)
poisson(100)
poisson(10000)

plt.scatter(Vs, stds)
```

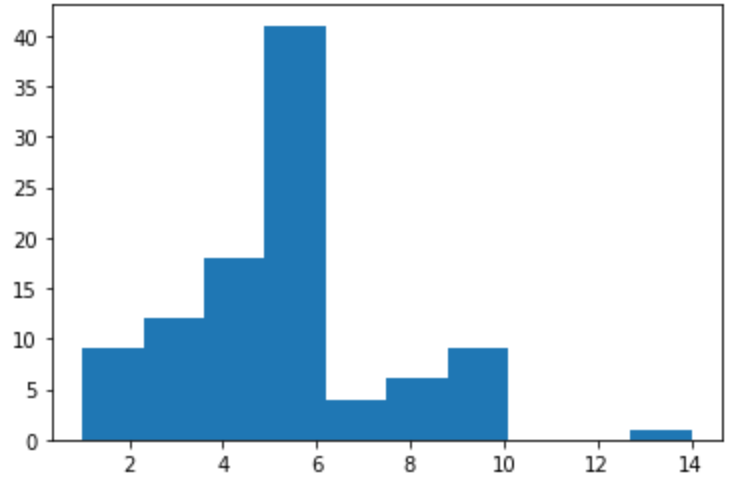
```
Mean: 9.3900 /// for nu = 10.000000
Standard deviation: 3.26771 /// for nu = 10
Standard error: 1.03334 /// for nu = 10
```



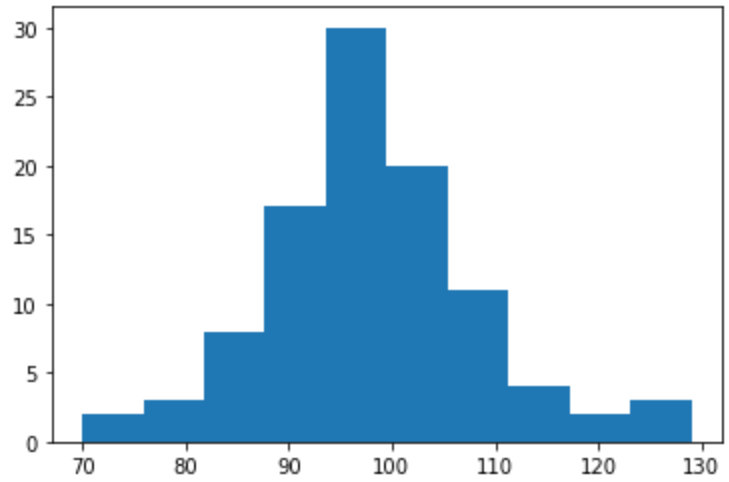

```
Mean: 1.2500 /// for nu = 1.000000
Standard deviation: 1.21140 /// for nu = 1
Standard error: 0.38308 /// for nu = 1
```



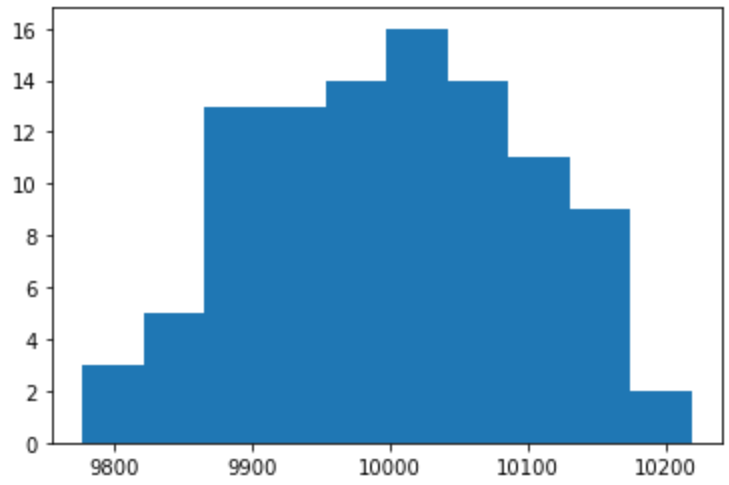

```
Mean: 5.2600 /// for nu = 5.000000
Standard deviation: 2.21639 /// for nu = 5
Standard error: 0.70089 /// for nu = 5
```




```
Mean: 97.7500 /// for nu = 100.000000
Standard deviation: 10.49035 /// for nu = 100
Standard error: 3.31734 /// for nu = 100
```

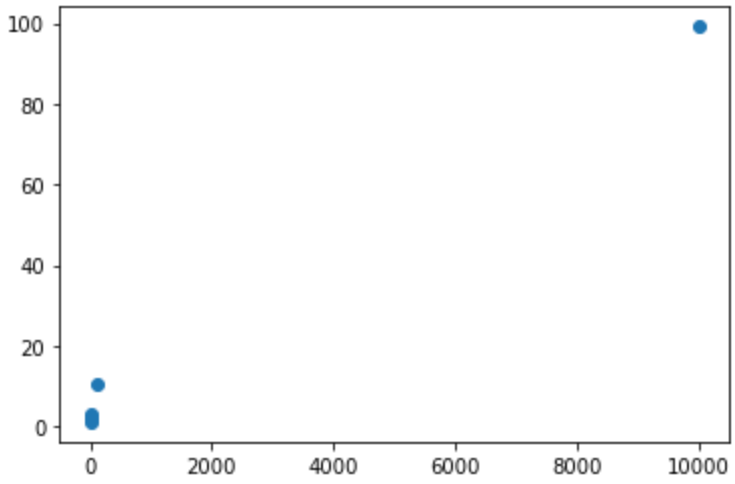



```
Mean: 10000.7000 /// for nu = 10000.000000
Standard deviation: 99.15992 /// for nu = 10000
Standard error: 31.35712 /// for nu = 10000
```



Out[19]:

```
<matplotlib.collections.PathCollection at 0x7fb174b95b20>
```



Doing something "useful" with a distribution

Random walks show up when studying statistical mechanics (and many other fields). The simplest random walk is this:

Imagine a person stuck walking along a straight line. Each second, they randomly step either 1 meter forward or 1 meter backward.

With this in mind, you can start to ask many different questions. After one minute, how far do they end up from their starting point? How many times do they cross the starting point? (The exact answers require repeating this "experiment" many times and taking an average across all the trials.) How much do you have to pay someone to walk along this line for several hours?

There are lots of interesting ways to generalize this problem. You can extend the random walk to 2+ dimensions, make stepping in some directions more likely than others, draw the step sizes from some probability distribution, etc. If you're curious, it's fun to plot the paths of 2D random walks to visualize Brownian motion.

Exercise 6

Use `np.random.binomial(1, 0.5)` (or some other random number generator) to simulate a random walk along one dimension (the numbers from the binomial distribution signify either stepping forward or backward). It would be helpful to write a function that takes N steps in the random walk, and then returns the distance from the starting point.

In [27]:
```
def random_walk(N):

    '''This function will return the distance from the starting point
       after a 1-dimensional random walk of N steps'''
    dist = 0

    x = np.random.binomial(1, 0.5, size=N)

    for val in x:
        if val == 1:
            dist += 1
        else:
            dist -= 1
    return np.abs(dist)
```

```
random_walk(5000670)
    # Use np.random.binomial(1,0.5) or another np.random function to "simulate" the rand
```

Out[27]:
```
2578
```

Now that you have a function that simulates a single random walk for a given N, write a function (or just some lines of code) that simulates $M = 1000$ of these random walks and returns the mean (average) distance traveled for a given N.

In [28]:
```
def average_distance(N):

    '''This function simulates 1000 random walks of N steps
       and then returns the average distance from the start.'''

    walks = []

    for i in range(1000):
        x = random_walk(N)
        walks.append(x)


    return np.mean(walks)

average_distance(50)

    # Use the random_walk(N) function 1000 times and return the average of the results
```

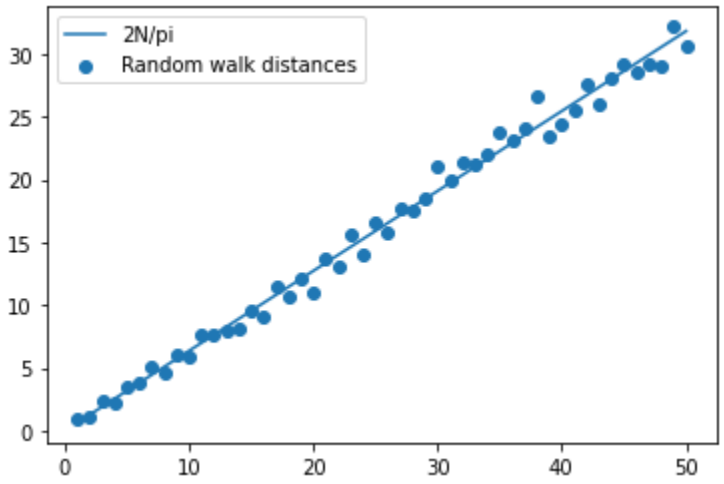
Out[28]:
```
5.648
```

It turns out that you can now use these random walk simulations to estimate the value of π (although in an extremely inefficient way). For values of N from 1 to 50, use your functions/code to find the mean distance D after N steps. Then make a plot of D^2 vs N. If you've done it correctly, the plot should be a straight line with a slope of $\frac{2}{\pi}$.

Once we get to fitting in Python, you could find the slope and solve for π. For now, just draw the line $\frac{2N}{\pi}$ over your simulated data.

In [59]:
```
Dsquared = []
simulation = []
vals = []
for val in range(1, 51):
    vals.append(val)
    x = average_distance(val)
    Dsquared.append(x**2)
    line = (2*val)/np.pi
    simulation.append(line)



plt.scatter(vals, Dsquared)

plt.plot(vals, simulation)
plt.legend(['2N/pi', 'Random walk distances'])
```

Out[59]:
```
<matplotlib.legend.Legend at 0x7f7434af0070>
```



In [ ]: