This exercise was generated from a Jupyter notebook. You can download the notebook here (diffusion_via_coin_flips_continued.ipynb).

Objective

In this tutorial, we will computationally simulate the process of diffusion with "coin flips," where at each time step, the particle can either move to the left or the right, each with probability 0.5. From here, we can see how the distance a diffusing particle travels scale with time.

Modeling 1-D diffusion with coin flips

Diffusion can be understood as random motion in space caused by thermal fluctuations in the environment. In the cytoplasm of the cell different molecules undergo a 3-dimensional diffusive motion. On the other hand, diffusion on the cell membrane is chiefly 2-dimensional. Here we will consider a 1-dimensional diffusion motion to make the treatment simpler, but the ideas can be extended into higher dimensions.

```
# Import modules
import numpy as np
import matplotlib.pyplot as plt

# Show figures in the notebook
%matplotlib inline

# For pretty plots
import seaborn as sns
rc={'lines.linewidth': 2, 'axes.labelsize': 14, 'axes.titlesize': 14, \
    'xtick.labelsize' : 14, 'ytick.labelsize' : 14}
sns.set(rc=rc)

import scipy.stats as stats
```

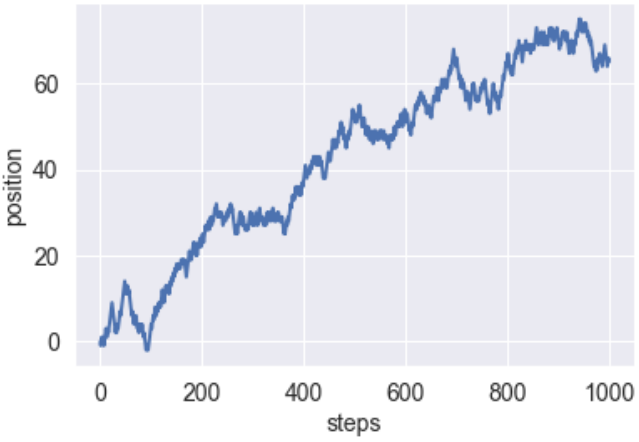
To simulate the flipping of a coin, we will make use of `numpy`'s `random.uniform()` function that produces a random number between 0 and 1. Let's see it in action by printing a few random numbers:

```
for i in range(10):
    print(np.random.uniform())
```

```
0.311810356524755
0.5089003563077501
0.29681371354447683
0.9442873860521601
0.0613220426031571
0.6923826236130496
0.05116772056527752
0.4189257062250654
0.5010688665552434
0.10483834668660841
```

We can now use these randomly generated numbers to simulate the process of a diffusing particle moving in one dimension, where any value below 0.5 corresponds to step to the left and any value above 0.5 corresponds to a step to the right. Below, we keep track of the position of a particle for 1000 steps, where each position is $+1$ or -1 from the previous position, as determined by the result of a coin flip.

```
# Number of steps
n_steps = 1000

# Array to store walker positions
positions = np.empty(n_steps)

# Initialize the walker's position
positions[0] = 0

# simulate the particle moving and store the new position
for i in range(1, n_steps):

    # generate random number
    rand = np.random.uniform()

    # step in the positive direction
    if rand > 0.5:
        positions[i] = positions[i-1] + 1

    # step in the negative direction
    else:
        positions[i] = positions[i-1] - 1

# Show the trajectory
plt.plot(positions)
plt.xlabel('steps')
plt.ylabel('position');
```



As we can see, the position of the particle moves about the origin in an undirected fashion as a result of the randomness of the steps taken. However, it's hard to conclude anything from this single trace. Only by simulating many of these trajectories can we begin to conclude some of the scaling properties of diffusing particles.

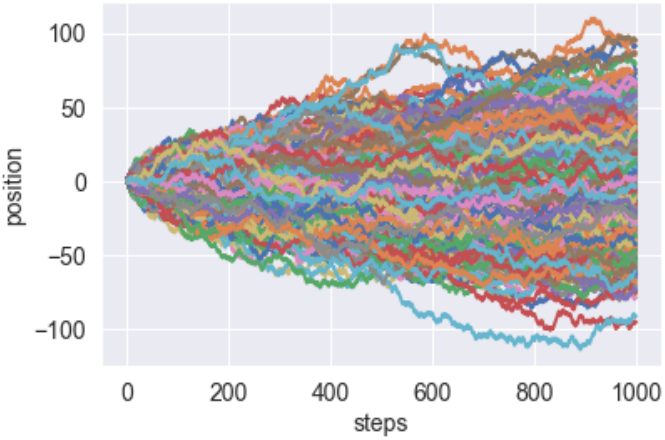
Average behavior of diffusing particles

Now let's generate multiple random trajectories and see their collective behavior. To do that, we will create a 2-dimensional `numpy` array where each row will be a different trajectory. 2D arrays can be sliced such that `[i,:]` refers to all the values in the `i`th row, and `[:,j]` refers to all the values in `j`th column.

```
# Number of trajectories
n_traj = 1000

# 2d array for storing the trajectories
positions_2D = np.empty([n_traj, n_steps])

# Initialize the position of all the walkers to 0
positions_2D[:,0] = 0

# first iterate through the trajectories
for i in range(n_traj):

    # then iterate through the steps
    for j in range(1, n_steps):

        # generate random number
        rand = np.random.uniform()

        # step in the positive direction
        if rand > 0.5:
            positions_2D[i, j] = positions_2D[i, j-1] + 1

        # step in the negative direction
        else:
            positions_2D[i, j] = positions_2D[i, j-1] - 1
```

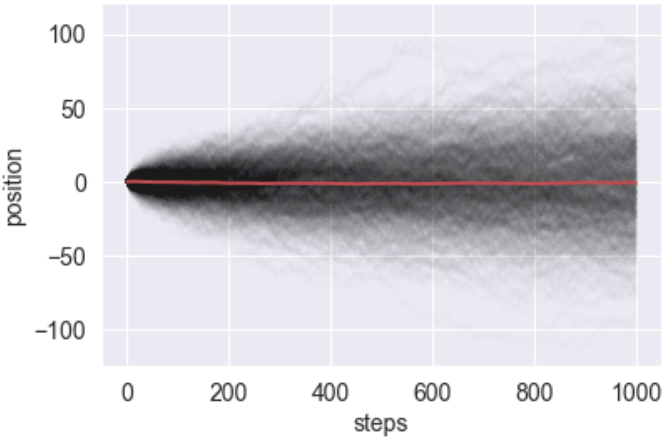
Now let's plot the results, once again by looping.

```
# iterate through each trajectory and plot
for i in range(n_traj):
    plt.plot(positions_2D[i,:])

# label
plt.xlabel('steps')
plt.ylabel('position');
```



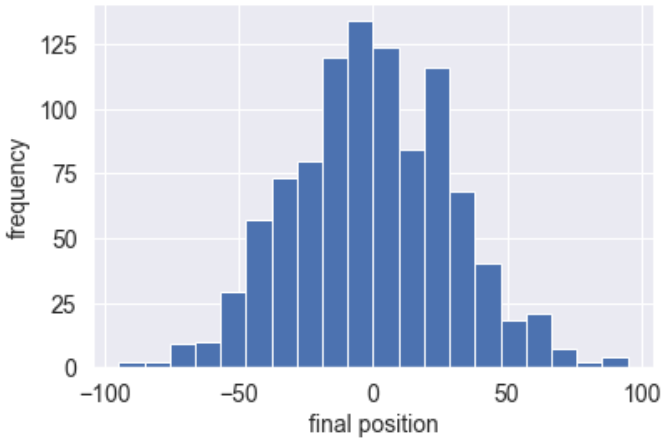
The overall tendency is that the average displacement from the origin increases with the number of time steps. Because each trajectory is assigned a solid color and all trajectories are overlaid on top of each other, it's hard to see the distribution of the walker position at a given number of times steps. To get a better intuition about the distribution of the walker's position at different steps, we will assign the same color to each trajectory and add transparency to each of them so that the more densely populated regions have a darker color.

```
# iterate through each trajectory and plot
for i in range(n_traj):
    # lower alpha corresponds to lighter line
    plt.plot(positions_2D[i,:], alpha=0.01, color='k')

# array for storing average position
avgs = np.empty(n_steps)

# loop through time to get average position
for j in range(n_steps):
    avgs[j] = np.mean(positions_2D[:,j])

# plot average position on top and label
plt.plot(avgs, color='r')
plt.xlabel('steps')
plt.ylabel('position');
```



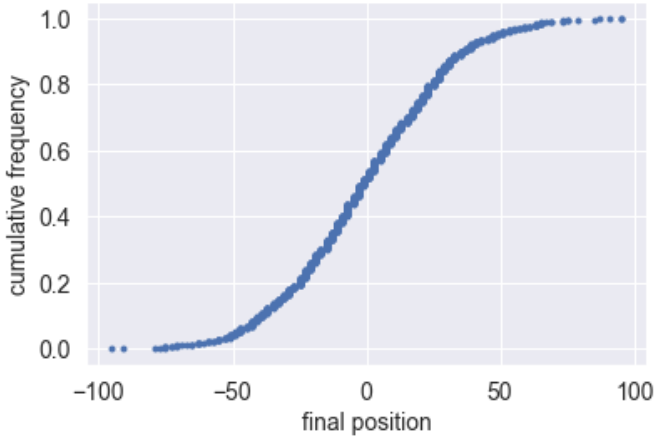
As we can see, over the course of diffusion the distribution of the walker's position becomes wider but remains centered around the origin, indicative of the unbiased nature of the random walk. To see how the walkers are distributed at this last time point, let's make a histogram of the walker's final positions.

```
# make a histogram of final positions
_ = plt.hist(positions_2D[:,-1], bins=20)
plt.xlabel('final position')
plt.ylabel('frequency');
```



As expected, the distribution is centered around the origin and has a Gaussian-like shape. The more trajectories we sample, the "more Gaussian" the distribution will become. However, we may notice that the distribution appears to change depending on the number of bins we choose. This is known as *bin bias* and doesn't reflect anything about our data itself, just how we choose to represent it. An alternative (and arguably better) way to present the data is as a *empirical cumulative distribution function* (or ECDF), where we don't specify a number of bins, but instead plot each data point. For our cumulative frequency distribution, the x-axis corresponds to the final position of a particle and the y-axis corresponds to the proportion of particles that ended at this position or a more negative position.

```
# sort the final positions
sorted_positons = np.sort(positions_2D[:,n_steps-1])
# make the corresponding y_values (i.e. percentiles)
y_values = np.linspace(start=0, stop=1, num=len(sorted_positons))

# plot the cumulative histogram
plt.plot(sorted_positons, y_values, '.')
plt.xlabel("final position")
plt.ylabel("cumulative frequency");
```

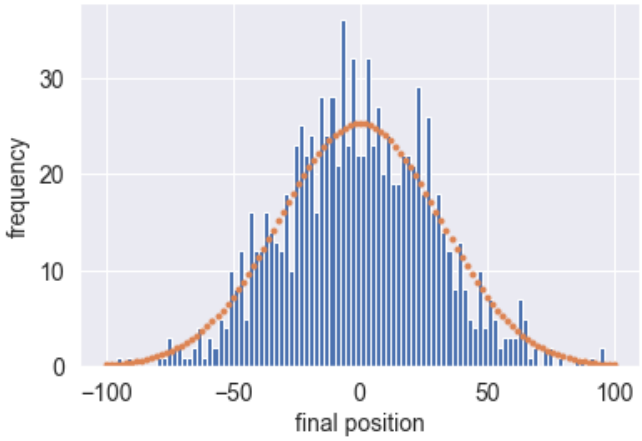


This way of visualizing the data makes it easier to tell that distribution of walkers is in fact symmetric around 0. That is, 50% of the walkers ended on a negative position, while 50% of the walkers ended on a positive position.

Comparing to the binomial distribution

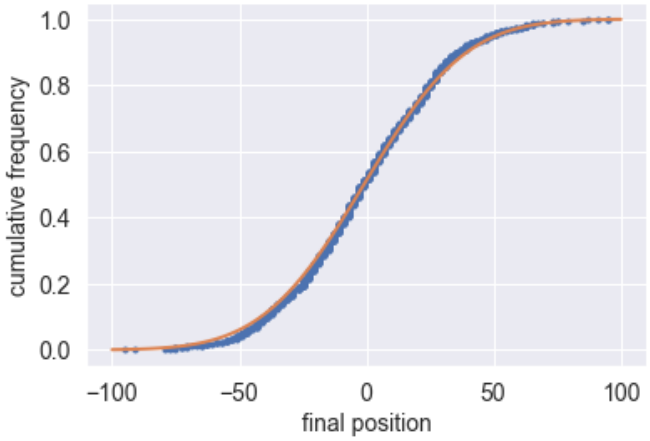
Returning to our histogram of the final particle positions, we've discussed that it should be binomially distributed, but it would be nice to verify this in some way. To do this, we can plot the *known* binomial distribution on top of our histograms, by calling the `stats.binom.pmf(k, n, p)` function. This function returns the probability of getting k heads, from n coin flips, given probability p of getting heads. We've seen that our diffusing particles ultimately take on some final position on the interval $[-100, 100]$. To understand the probability of this occurring, we need to convert these positions to the number of "heads" (or the number of right steps taken). In this case, the number of heads corresponds to the interval $[450, 550]$. That is, if you got 450 heads (and thus 550 tails), you've traveled 100 mores steps to the left than to the right and ultimately ended up at position -100. By the converse argument, if you got 550 heads, you end up at position 100. Let's use `stats.binom.pmf()` to determine the chance of getting these number of heads for the total number of time steps (1000) we computed and plot the results over this range $[450, 550]$.

```
# make a histogram of final positions, hardcoding bin positions
_ = plt.hist(positions_2D[:,-1], bins=range(-100,100,2))

# final positions occur on the interval [-100, 100]
final_positions = np.linspace(-100,100,101)

# number of heads corresponding to the final_position
# should be interval [450, 500]
n_heads = (n_steps + final_positions)/2

# use binomial distribution to determine prob of n_heads from taking n_steps
# scaled by the number of trajectories for overlaying on our histogram
binomial_pmf = stats.binom.pmf(n_heads, n_steps, p=0.5)*n_traj

# plot and label
plt.plot(final_positions,binomial_pmf,'.')
plt.xlabel('final position')
plt.ylabel('frequency');
```



These seem to overlap pretty well, and it seems as though our final particle positions are in fact binomially distributed. To see this even more explicitly, we can plot the CDF for the binomial distribution over our ECDF. This is done by simply using `stats.binom.cdf()` instead of `stats.binom.pmf()`.

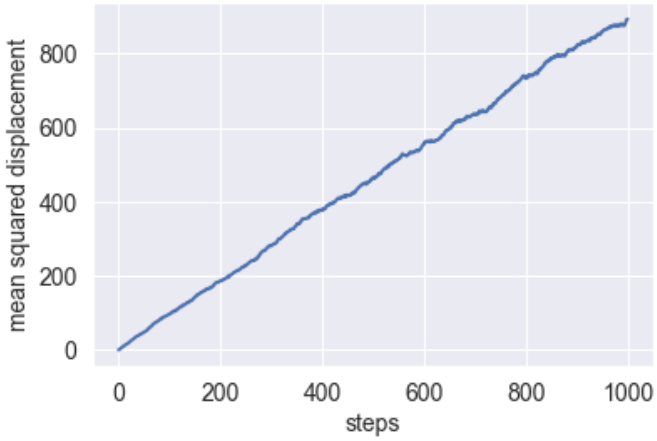
```
# sort the final positions
sorted_positons = np.sort(positions_2D[:,n_steps-1])
# make the corresponding y_values (i.e. percentiles)
y_values = np.linspace(start=0, stop=1, num=len(sorted_positons))

# binomial cdf
binomial_cdf = stats.binom.cdf(n_heads, n_steps, p=0.5)

# plot the cumulative histogram
plt.plot(sorted_positons, y_values, '.')
plt.plot(final_positions,binomial_cdf)
plt.xlabel("final position")
plt.ylabel("cumulative frequency");
```



Mean squared displacement (MSD)

From visual inspection we could tell that the particles tend to go further away from the origin as the number of steps increases. To get a more quantitative understanding of the particles' positions, we can compute something known as the *mean squared displacement* or MSD. This serves as a metric of how far the particles have traveled from the origin that is invariant to the direction the particle moved. Let's plot the how the mean squared displacement scales with the number of steps.

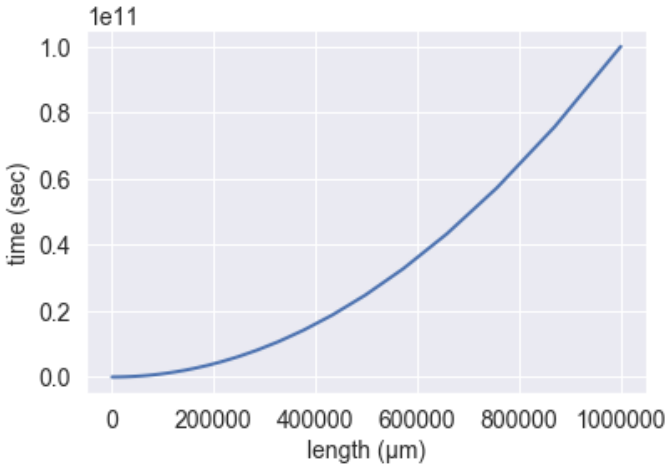
```
# array for storing the MSD
MSDs = np.empty(n_steps)

# calculate the MSDs for each step
for j in range(n_steps):
    MSDs[j] = np.mean(positions_2D[:,j]**2)

# Plot the MSD
plt.plot(MSDs)
plt.xlabel('steps')
plt.ylabel('mean squared displacement');
```



We see that the mean squared displacement clearly increases linearly with the number of steps. This leads to the important result that for diffusion, the time it takes to reach a certain distance scales with the square of that distance. This means that by diffusion, it will take 4 times as long to travel twice the distance. This is in contrast to ballistic motion (like a car driving on a freeway), where the time to travel scales linearly with the distance.

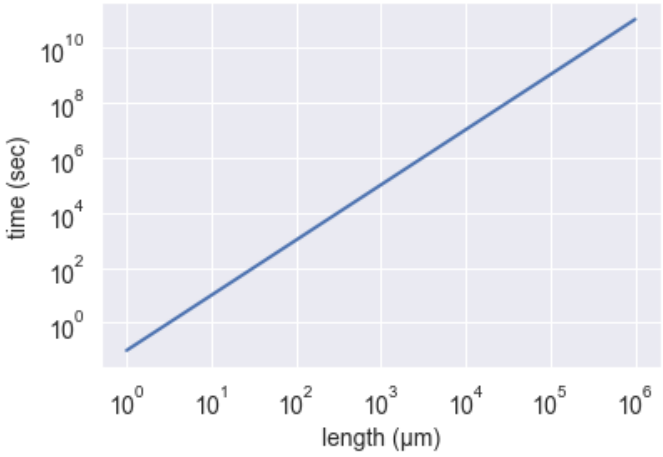
This result leads us to the relationship that the time t it takes to diffuses scales with $\frac{L^2}{D}$, where L is the length and D is the diffusion coefficient. Let's explore this relationship by plotting t vs. L for length scales relevant to biology, like 1 µm (i.e. an *E. coli*) to 1 m (i.e. the longest neuron in the human body). We will do this for $D = 10\,\mu m^2/\mathrm{sec}$, which is typical for a protein in cytoplasm.

```
# diffusion constant
D = 10 # μm^2/sec

# range of lengths in microns from 10^0 to 10^6
L = np.logspace(0, 6, 100)

# calculate time to diffuse
t = L**2 / D # sec

# and plot
plt.plot(L, t)
plt.xlabel('length (μm)')
plt.ylabel('time (sec)');
```



From this, we see we recapitulate what we calculated yesterday, where it takes 10^{11} seconds, or ≈ 3000 years for a protein to diffuse the length of a meter! However, the way we've plotted this makes it hard to tell what's going on at small length scales, so let's instead plot the data on a log-log plot.

```
# Make a log-log plot of the diffusion timescale
plt.loglog(L, t)
plt.xlabel('length (μm)')
plt.ylabel('time (sec)');
```



From here, we can better see that for short length scales of 1 μm, the time to diffuse is quite fast, less than a second. As we go to length scales of hundreds of microns, the diffusion times become on the order of minutes to an hour. This provides an argument for why bacteria use diffusion as a means of transport within the cell, while eukaryotic cells rely on directed transport via motor proteins to shuttle materials around the cell.