**Note 1**

* In the previous lesson, we have developed two Python codes to simulate the motion of Brownian particles with and without on-the-fly animation capabilities.
* In the present lesson, we will use the theoretical concepts we have introduced to analyze the trajectory data produced by our simulations.

**Note 2**

* For the current purposes, we do not require any visualization.
* Therefore, let us use our simple simulation code, which generates the trajectories of 'nump' non-interacting Brownian particles.
* We are not interested now in animating the results, what we want is to perform a quantitative analysis of the trajectories.

**Note 3**

* To begin, let us examine the positions of the 1000 independent Brownian particles as a function of time.
* To easily compare the trajectories, we will plot the x, y, and z positions of each particle independently.
* Here, we plot t as a function of R. Thus, time increases in the vertical direction, and the horizontal position gives the x, y, or z position of the particles.
* In this code example, we use red, blue, and green to represent x, y, and z, respectively.
* At t=0 all the particles were located at the origin. As you can easily see, for t>0 they start to move in a highly stochastic manner
* The individual paths look random, and you might question how much useful information we can actually extract from these trajectories.
* However, as you might expect by now, the distribution of the paths obeys a well defined rule.
* While the average position is zero for all times, the variance increases as time progresses. You can see this in the way the dense central region becomes wider as time increases, but stays centered at the origin.
* Now, let us examine these properties in more detail.

**Note 4**

* Here we calculate the distribution function for the probability of finding a particle at a position Rα at the end of the simulation (t=nums\*dt).
* The distributions can be calculated separately for the x, y, and z components. Given the symmetry of the problem, and the fact that the diffusion coefficient is the same in all directions, we expect the results to be statistically the same.
* Thus, the three distribution functions computed from the simulation should be equal, and they should match the theoretical form given by Eq.(G1-G3).
* The agreement between our simulation results and the theoretical prediction is seen to be very good, showing that the particle motion is isotropic and the variance increases linearly with time as predicted by Eq.(G3).

**Note 5**

* Next we plot the velocity of the 1000 independent Brownian particles as a function of time, in the same way that we plotted the time evolution of the positions.
* As before, we plot the x, y, and z velocity of each particle independently.
* Just as with the distribution of the positions, we see that the velocity components fluctuate around zero.
* However, in contrast with the positions, we see that the variance of the distribution does not increase in time.
* The distribution of velocities is the same for all times. Although it is again given by a Gaussian distribution, in this case, the variance is not a function of time.

**Note 6**

* We calculate the distribution function for the particle velocities separately for the x, y, and z components.
* We have plotted the three velocity histograms using the last half of the trajectory data.
* The agreement between our simulation results and the theoretical prediction is perfect, showing that the Maxwell-Boltzmann distribution is exactly satisfied.
* In this case, the distribution is clearly time-independent, therefore, we could get better results by taking averages over particles and time.
* In contrast, we could not do the same for the distribution of particle positions, since the distribution is changing in time with the standard deviation scaling with t.

**Note 7**

* We now calculate the velocity auto-correlation function, and compare it with the theoretical results.
* Here, we can greatly reduce the work we must do by using numpy's built-in correlate function.
* Given two-arrays or signals, a and b, it calculates the correlation between them.
* This function has several options for how to handle the overlap of the signals, which determines the size of the output and the boundary effects that are observed.
* We do not go into the details here, you can go to the numpy documentation for more information.
* For our current purposes, we have defined an "auto-correlate" function which handles all these options for you and returns the appropriate correlation function.
* Here, we use this function to compute the single-particle velocity autocorrelation function.
* Note that we compute the correlations over the x, y, and z components of the velocity separately for each particle, add them, and then divide by the number of particles to obtain the proper average.
* As you can see here, the agreement with the theoretical prediction Eq.(G7) is perfect.

**Note 8**

* Finally we calculate the power spectrum of the particle velocity, and compare it with the theoretical results.
* To compute the spectrum SV(ω) using Eq.(G8), we need the Fourier transform of the particle velocity V(ω) via Eq.(G9).
* This can be efficiently calculated using the Fast Fourier Transform.
* As you would expect, the results are in perfect agreement with the theoretical curve.
* Note however, that the point for ω=0 deviates from the theory.
* The reason for this is simple: we have used a finite time window, therefore we do not have accurate information about the long time / small frequency processes.