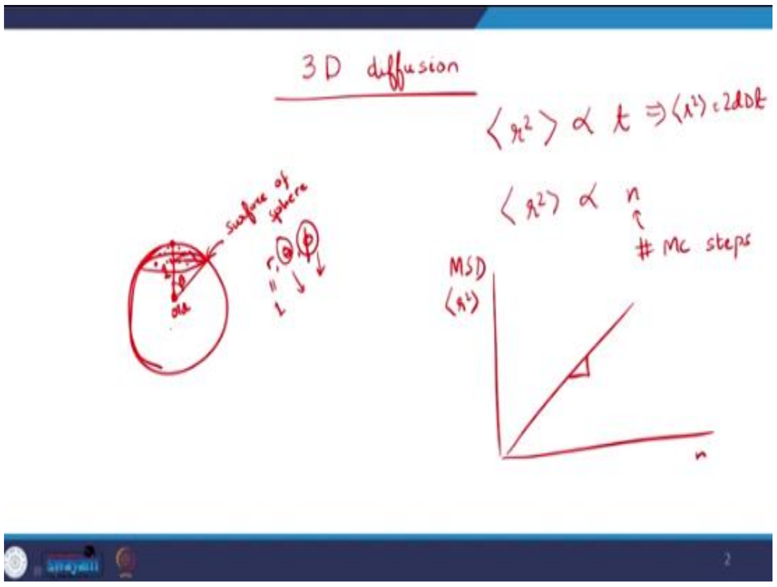
Advanced Thermodynamics and Molecular Simulations

Prof. Prateek Kumar Jha
Department of Chemical Engineering
Indian Institute of Technology, Roorkee

Lecture - 37
Numerical Implementation of Monte Carlo Simulation: Python Examples II

Hello all of you. So, in the last lecture we have been discussing how can we numerically implement a Monte Carlo scheme on a computer. We did some basic examples such as computation of π diffusion on 2 dimensional kind of phase space. I will do some more examples today and we will discuss how can, we incorporate boundary conditions in a Monte Carlo code.

(Refer Slide Time: 00:52)


3D diffusion
$\langle r^2 \rangle \propto t \Rightarrow \langle r^2 \rangle = 2dDt$
$\langle r^2 \rangle \propto n$
# MC steps
surface of sphere
MSD
$\langle r^2 \rangle$
n

So, the first thing we will discuss is extending the diffusion problem through a 3 dimensional diffusion just to quickly recap I was telling that according to Einstein relation the mean square displacement should go like the time or in the Monte Carlos simulation, it should go like the number of Monte Carlo steps.

$$< r^2 > \propto t \ and < r^2 > \propto n$$

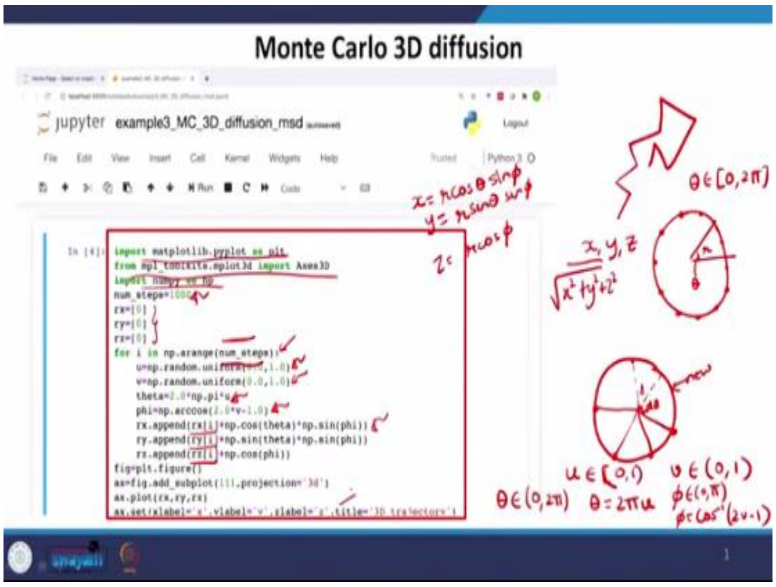
Here n= number of monte carlo (MC) steps.

And therefore if I compute the slope of that is some measure of the diffusion coefficient-

$$< r^2 > = 2dDt$$

Where small d is dimensionality and capital D is the diffusion coefficient. So, this is what we want to get until so far if you recall there is no boundary condition coming. So, a particle is simply moving in this space and it can just go anywhere in reality, it is going to be bounded and that is what we will discuss next. But first let us discuss what will happen simply by extending 2 dimension to 3 dimension. What does the mean square displacement the MSD versus time looks like so here is the code for doing that**,** so I have imported again the plotting library.

(Refer Slide Time: 02:15)



Monte Carlo 3D diffusion

jupyter example3_MC_3D_diffusion_msd

```
In [4]: import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D
        import numpy as np
        num_steps=1000
        rx=[0]
        ry=[0]
        rz=[0]
        for i in np.arange(num_steps):
            u=np.random.uniform(0.0,1.0)
            v=np.random.uniform(0.0,1.0)
            theta=2.0*np.pi*u
            phi=np.arccos(2.0*v-1.0)
            rx.append(rx[i]+np.cos(theta)*np.sin(phi))
            ry.append(ry[i]+np.sin(theta)*np.sin(phi))
            rz.append(rz[i]+np.cos(phi))
        fig=plt.figure()
        ax=fig.add_subplot(111,projection='3d')
        ax.plot(rx,ry,rz)
        ax.set(xlabel='x',ylabel='y',zlabel='z',title='3D trajectory')
```

Now, since we will be doing a 3-dimensional plot it is slightly different library than earlier we want to have some kind of a 3D access, so I am using this particular library here. There are plenty of options on web you can find. So, I am using one of that and then I of course need the numpy library and I am calling them as. So, then we define the number of steps. Now, there are three coordinates x y and z and I am starting from the origin but other than that the idea remains the same. That has to generate the steps of unit length, but now in 3D space.

So, now it may seem natural that I should instead of like generating like x y in 2D I should generate x y z randomly in 3D and then simply divide them by their num value that is under root of $x^2 + y^2 + z^2$. It turns out that this is kind of there is some problem with that or it is not so easier to work with. There is a more efficient way of doing it and that is using the idea that whenever I say that I am going to be making a step of length unity, essentially what I am saying is that in 2D the new point is going to lie on the periphery of the circle centered at the old point.

So, the old point is a center and new point can lie anywhere on the circle of unit radius drawn with the old point at the center.

So, instead of saying that I will generate x and y and divide them by the under root $x^2 + y^2$ we could have as well said that my r is fixed. I will simply pick a θ value and the θ can be anything from 0 to 2π and that will generate points on there. The advantage of doing that if you think about it is that now I will be generating only one random number for theta between 0 to 2π in 2 dimensions and that is of course computationally more efficient than generating two numbers x and y and performing the square root operation.

Keep in mind that every square root operation comes at good amount of expense to a computer because normal arithmetic operations like addition multiplications and all that are pretty quick on a computer. But Square root takes a bit of time you may not feel that on modern computers, but keep in mind that that is going to be much larger in comparison to simple arithmetic manipulations. So, it is always better that we avoid square root operations as much we can in the code that will really make a big difference to your performance because in one step it may not make any difference, but if I make say a billion steps then all that effort of every SQRT operation gets multiplied over the number of steps and then the code efficiency really sufferers.

So, we have to really make the part within a loop highly efficient because it is those loops which are going to be a bottom neck for the code. I was telling you in the motivation part that one of the reasons that we cannot do a very large systems is because there is so much computational cost of doing that. So, when I make our code more efficient clearly we can do larger systems for lesser computation times and this is why we have to really see which operations I am doing within the loop and see how can we optimize that and one of the ways is for example this I choose a θ value into dimensions between 0 to 2π that could have done in 2D but I have not done that but you can try that on.

Now let us see I want to do the same idea in 3D, so what it is going to be? So, in 3D essentially we will be having again points lying, but now not on the circle, but on the on a sphere with a unit radius centered at the old point. So, my points can be anywhere on the surface of the sphere. So, using what I have been telling you in the same idea here. Now you may imagine that there are three coordinates for a sphere rθ and Φ. So, instead of working with x, y, z I can work with r, Φ, Φ. So, r is always 1 when I look at points on this sphere and Φ and Φ they vary in a range. So, I can pick Φ from range and Φ from the range. So we will have only two generations of under numbers as opposed to generating x, y and z.

It turns out that if I pick θ uniformly and Φ uniformly that does not generate the points on the sphere uniformly. And the real reason really gets into the geometry I do not want to spend too much time on that. But keep in mind that one of the angle is something like this angle and the other angle is something like that is applying on one of the cuts on the on the circle. So, this angle is something like this.

So, what it eventually gives me is that you tend to generate more points at the poles of the circle and lesser points in the near the equator or the density of points happens to be more near the poles and lesser near the equator and you do not want that because in a Monte Carlo is scheme you really want to have a uniform distribution. So, therefore it is not wise to generate θ and Φin the range, whatever it is defined just like what I did in 2D.In 2D it turns out to be not a problem, but in 3D this happens to be somewhat problematic. Now there is a mathematical way around it and that is the following that is what we have implemented here.

So, instead of generating θ and Φ you may generate two numbers u and v in the range 0 to 1. And theta then that has to be in the range 0 to 2π will be generated as 2π multiplied by u. So, there is no problem with that and Φ that is in the range 0 to π will be generated as cos inverse of 2v - 1.

So, cos inverse of 2v - 1 if v is in the range of minus 0 to 1 it is going to vary from cos inverse of - 1 to cos inverse of 1 that is the entire range of cos and that will give me a Φ value accordingly. I turns out that this really solves this problem of uniform generation of points you can find very nice demonstrations on web. I will not spend too much time on that, but keep in mind that this is done to generate points uniformly on the sphere surface. So, this is what we are doing here, we are generating, so now I am inside the loop where I am running n number of steps and the num steps, declare the number of steps here u between 0 to 1, v between 0 to 1, θ as 2πu and Φ as cos inverse of to 2v - 1. And then I know that my, x is in Cartesian coordinates can be written in terms of the spherical coordinate variables, as-

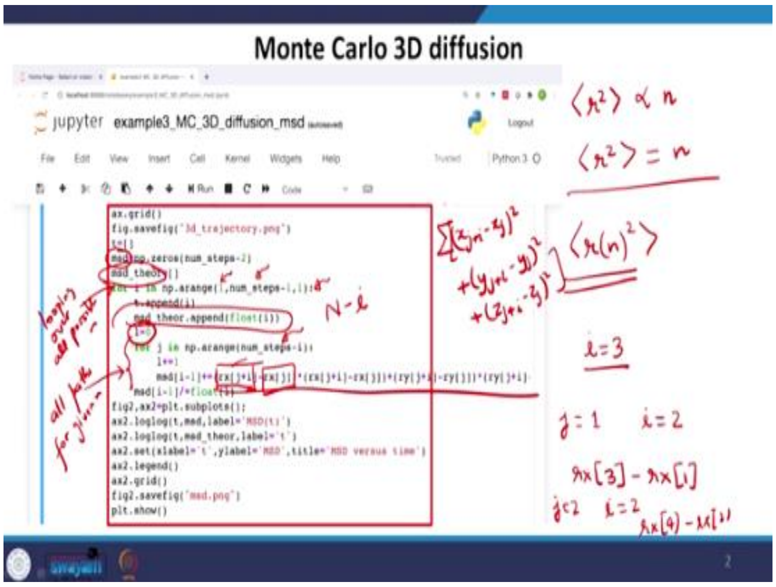
$$x = r\cos\theta\sin\phi$$

$$y = r\sin\theta\sin\phi$$

$$z = r\cos\phi$$

So, once I have found θ and Φ I can find x, y and z coordinates. And I add to the previous step just like what I have done in the 2D example except that we have 3 coordinates now. And we are generating the new position is slightly differently than we have done clearly we are in 3D and we have figured that there should be an efficient scheme to generate points that is generating points on the spheres of the unit circle around the; with keeping that old point at the center of it. So, with that, I can then plot it.

(Refer Slide Time: 12:28)



Monte Carlo 3D diffusion

```
ax.grid()
fig.savefig('3d_trajectory.png')
t=[]
msd=np.zeros(num_steps-2)
msd_theor=[]
for i in np.arange(1,num_steps-1,1):
    t.append(i)
    msd_theor.append(float(i))
    l=0
    for j in np.arange(num_steps-i):
        l+=1
        msd[i-1]+=(rx[j+i]-rx[j])*(rx[j+i]-rx[j])+(ry[j+i]-ry[j])*(ry[j+i]-
    msd[i-1]/=float(l)
fig2,ax2=plt.subplots();
ax2.loglog(t,msd,label='MSD(t)')
ax2.loglog(t,msd_theor,label='t')
ax2.set(xlabel='t',ylabel='MSD',title='MSD versus time')
ax2.legend()
ax2.grid()
fig2.savefig('msd.png')
plt.show()
```

jupyter example3_MC_3D_diffusion_msd
File Edit View Insert Cell Kernel Widgets Help
Python 3
$\langle r^2 \rangle \propto n$
$\langle r^2 \rangle = n$
$\sum (x_{j+i}-x_j)^2 + (y_{j+i}-y_j)^2 + (z_{j+i}-z_j)^2$
$\langle r(n)^2 \rangle$
N-i
looping over all points
all paths for given i
$i=3$
$j=1$ $i=2$
$rx[3]-rx[1]$
$j=2$ $i=2$
$rx[4]-rx[2]$

So, I can save that trajectory whatever we have. But the next thing we want to do in the same code is we also want to find the mean square displacement. So, the way we will go around it is first we know that my theoretical value is something like this-

$$< r^2 > \propto n$$

So, I will basically keep a theoretical plot as something like this-

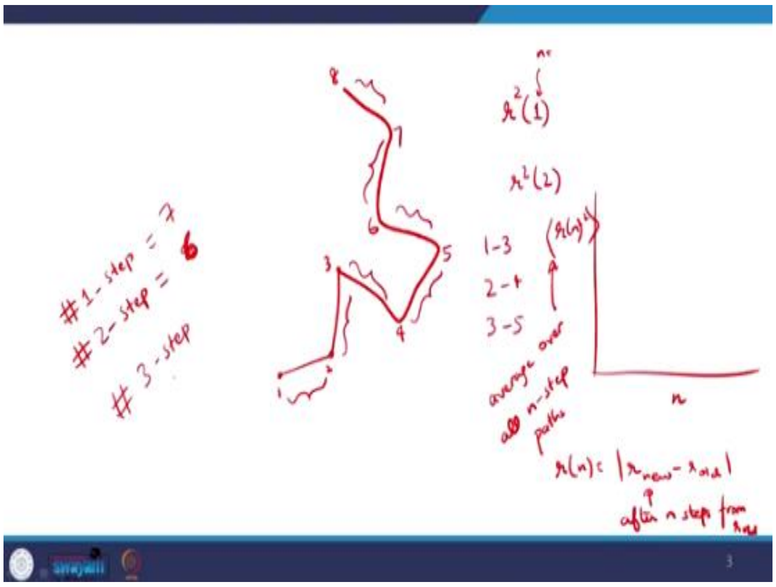
$$<r^2>= n$$

We only care about the slope here, so the pre-factors I am not really putting in there.

So, therefore, I create a theoretical mean square displacement array MSD theoretical, I start with empty array. And I also create an MSD array that will contain the actual mean square displacement we are getting from that and we compare both of that. So, Clearly I cannot do it for all the steps, because now I am looking at the mean square displacement with the between the previous step and the new step. So, we have to be a bit careful here when we compute the mean square displacement. So, what is the mean square displacement as a function of time?

So, essentially when we say that we will find this kind of quantity where n is the number of Monte Carlo steps. So, essentially I am looking at difference between two time points. So, as to speak but in the Monte Carlo sense this difference between the 2-steps in the Monte Carlo simulation. But here the key idea is that the origin is not important for the computation of mean square displacement that is to say that, let us say for example this is the trajectory we are interested in and let us say I want to find the value of the mean square at for n is equal to 1. Now in this case, you may imagine that I can do only for the first step that I have made and that should be the value I should take.

(Refer Slide Time: 14:27)



The answer is no because even going from 2 to 3. I am making the step of 1 unit length, so this is also a unit length step this is also a unit length step. So, is this, so is this, so is this, so is this. So, even though we have moved only one particles we have something like n - 1, 1-step paths in the trajectory. Similarly if I look at r square 2 now I am looking at 2-step paths. So, 1 to 3 is a 2-step path but 2 to 4 is also a 2-step path 3 to 5 is also a 2-step path. So, basically whenever I compute the mean square displacement, ultimately, we are interested in the, like r n square versus n and the average of that. This average is computed over all n-step paths that you have in the system and for each of the individual paths we are going to have. The r defined as something like r of the new minus r of the old but the new is after n-steps from r of old.

So, we count every n-step path and we find the distance for that particular n-step path or the displacement and we compute the square of the displacement that is the mean square displacement in there. So, this is what we are essentially doing in this particular code. So, what we do here is the theoretical part is clearly easy we just take as this but to compute this from the experiment. I run a loop within that loop. So, I go from the step number 1 to the last step but then within that I look at all the previous step before that particular step. So, let us say if i is equal to 3, so then I am looking at the j values which are before that. So, this is what exactly we are we are doing in here, So, we are counting all the n-step paths by doing this particular second loop, so in this case rx (j + i) and rx (j) means that j + i is the coordinate after j + i steps. That is and j can be like any value in the range that I have specified. So, let us say for example, if j is equal to 1 and i is equal to 2 then I am looking at 2-step paths between the something like rx [3] - rx [1].

So, it is looking at 2-step path starting from 1 to 3 now if j is equal to 2i equal to 2 I am doing something like rx [4] - rx [2] so within the second loop I am looking at all the paths of the given number of steps. So, all paths for given n and the first loop we are looping over all possible n. So, if you agree with me, then you see in the first loop I am running from 1 to n - 1. So, if n is the total number of steps that I have taken, then the maximum path length will be going to be something like 1 - n-1.

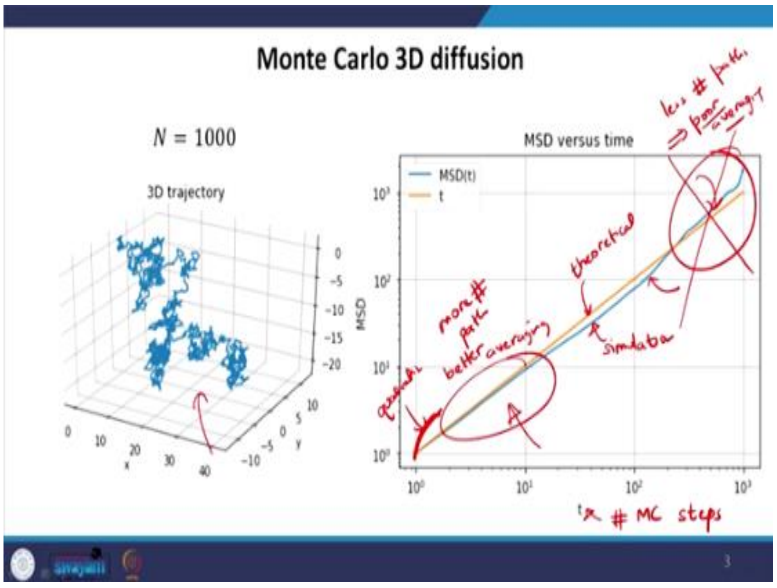
So, therefore, that is where I run the i loop from but the j loop will run to from 0 to something like n - π. So, let us say if i value is equal to 1. So, I will run that loop from 0 to n - 1. If i value is equal to 2 I will run from 0 to n - 2. If i value is equal to n - 1, I will run from 0 to 1. So, clearly what you see is that as the i value increases the range of j value decreases that is natural because the number of 1-step paths is clearly higher than the number of 2- step paths and that is clearly higher than the number of 3-step paths and so on because that our number of points are limited.

Now within that what I am computing I am computing the sum of-

$$\sum\left[\left(x_{j+i}-x_j\right)^2+\left(y_{j+i}-y_j\right)^2+\left(z_{j+i}-z_i\right)^2\right]$$

that is the square of the distance for that particular i step path and we add all the all the possible paths and we divide it by the number of such parts to count the number of paths available. . I have added a counter variable l, so every time there is a path available I increment l by 1. So, then whatever MSD I have, can computed I have simply summed over all the possible paths of a given number of steps but then I divide by the number of possible paths and therefore I get the average mean square displacement for a given number of steps and then we plot the both the theoretical value and the simulation value for the mean square displacement versus time. This is what the code is doing here.

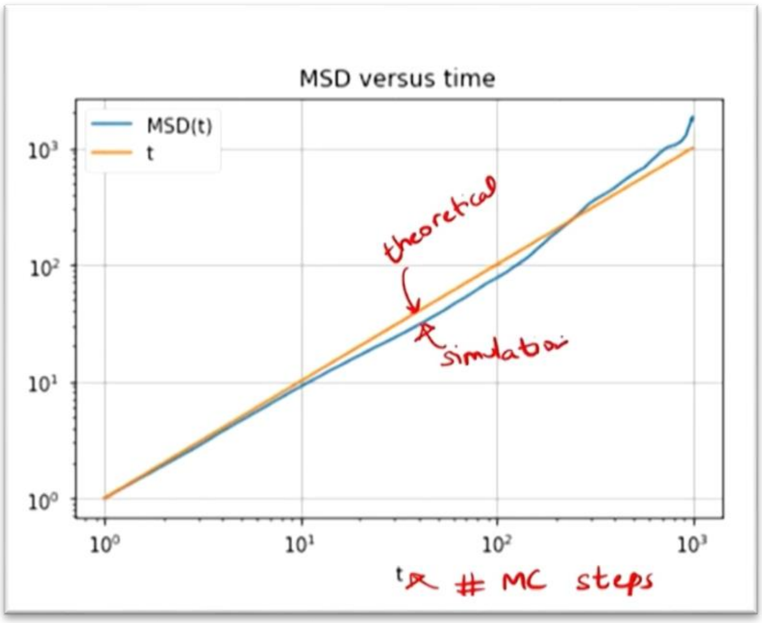
(Refer Slide Time: 22:52)



Monte Carlo 3D diffusion

$N = 1000$

So, let us see what it looks like, this is what the trajectory looks like now in the 3-dimensional space now, we are doing it for 1000 points. Now, I am plotting the MSD versus time here keep in mind although I have said time it actually refers to the number of Monte Carlo steps and you see this is my theoretical line that is given by the Einstein relation and I am doing it in the log. So, it is easier to compare and this is what we get from the simulation and you can see that it is pretty close any deviation we can possibly attribute to the fact that the number, of steps are small and therefore there will be some deviation from the theoretical value but if I increase n that deviation may probably go away.



There is another way of doing that we could have run many simulations and averaged over many simulations for the same value of n and that also will improve the averaging but nonetheless, the key point in the mean square displacement plot is that even though we are running a single simulation we are actually able to do an average because there are so many 1-step paths, so many 2-step paths, so many 3-step paths and so on but as soon as we go to larger values of n somewhere here then I was telling you that as the number of steps increases then you have fewer paths available. So, there are more 1-step path than in comparison to 2-step, more 2-step than comparison to 3-step. So, there will be lesser and lesser number of paths as I increase the one number of Monte Carlo steps and therefore somewhere in this regime I am able to do better averaging even by running a single particle simulation just because we have more number of paths available from the same simulation.

On the other hand in here since the number of paths I get is less we have actually a poor averaging and this is something that will, the problem will remain in a molecular simulation. So, we may get a linear kind of a behavior or following the Einstein relation for short times but at longer times we will always have a deviation from the theoretical line and that is coming from the fact that our averaging is poor we are not averaging over large number of possible paths and this has no physical meaning except the fact that this coming from a numerical thing.

So, whenever we want to interpret these kind of data and want to extract the behaviour we have to discard the later part where the averaging is poor and focus on the initial part where the averaging is better, now keep in mind I was telling you yesterday that in the very, very beginning before the collisions take place you actually also have a quadratic regime and that is clearly not captured here because we are doing a single particle system when we actually are doing multiple system then only we can have collisions in this case we are doing a random walk of a single particle, so therefore we do not see that initial quadratic region. But I was telling you there will happen for very, very short time, so even if you are doing a multi particle system, you may see it or you may not see it but as a good practice you should discard always the later part because you have averaging issues there and you should also discard some initial part because you may expect that the behavior may be quadratic in this regime not for the random walk problem, but when we are extracting the diffusion coefficient from a multi particle simulation.

So with that I want to discuss now one of the problems that I will take up in the next lecture and that is in here, we have said that the particle can just go anywhere in the infinite space. There was telling you that in reality the particle is going to be bounded in some particular volume you need to specify some sort of a boundary condition that is even more important when you are doing a multi-particle system because you do not want all the particles to go far apart that is not what is realistic enough what I was telling you in the previous classes that you can do something like a periodic boundary condition, for example, or you can even do a fixed boundary condition where you confined them to a box.

Now the question is what effect those boundary conditions have on the diffusion process? If the boundary condition really changes the diffusion coefficient significantly now the question rises that is the simulation then correct. On the other hand if the diffusion coefficients are being captured well then we can say our simulation is probably running well, we are not completely sure we have to look at other properties but diffusion coefficient is clearly the one that can be a starting point to see whether our molecular simulation is behaving as expected because as per the Einstein relationship the mean square displacement should be linear in time. So, our system should be such and our simulation should be such that at least this should be followed. And if this is happening we can be somewhat sure that other things may also fall in order. It turns out the diffusion coefficient is actually much more difficult to compute than some other properties that we will discuss later in the class.

So, with that I want to stop this lecture, thank you.