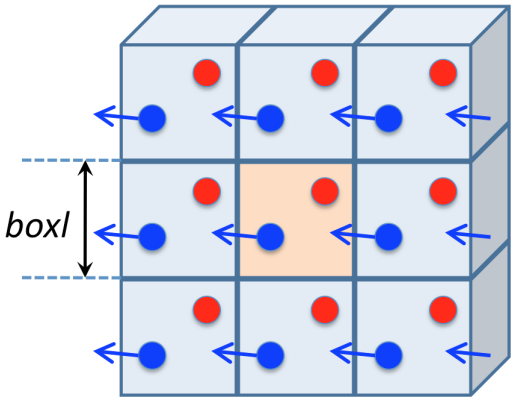
# Stochastic Processes:
# Data Analysis and Computer Simulation

## Brownian motion 3: data analysis

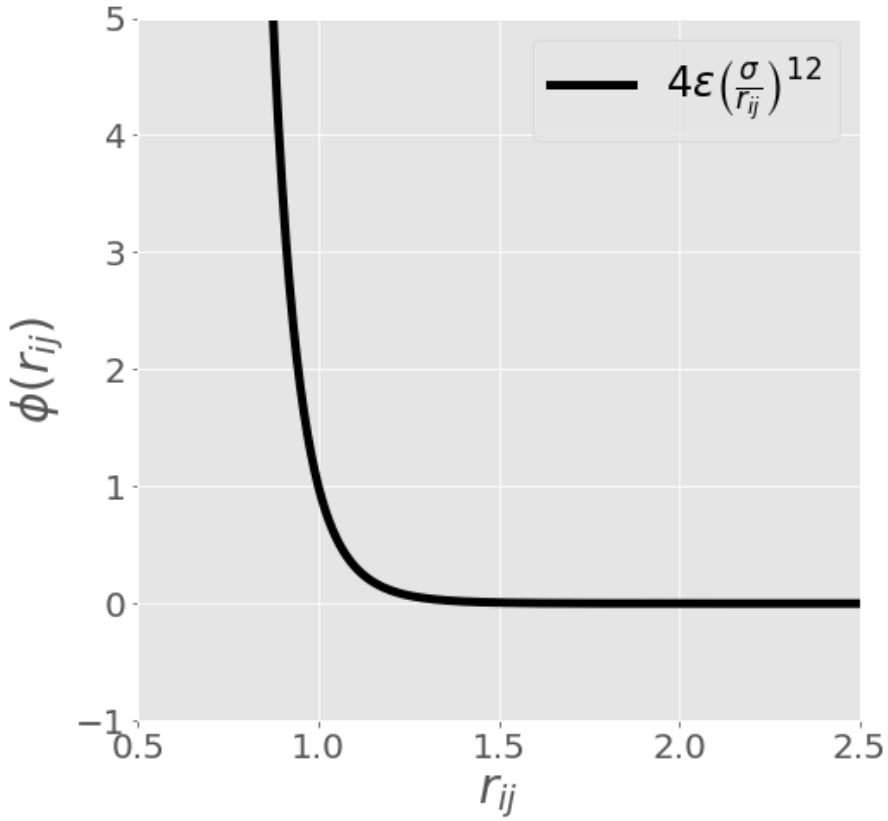
## 3. Interacting Brownian particles

## 3.1. Necessary changes for interacting Brownian particles

### Periodic boundary conditions



### Inter-particle interaction




$$U = \sum_{r_{ij}<2\sigma} \phi(r_{ij}) = \sum_{r_{ij}<2\sigma} 4\epsilon\left(\frac{\sigma}{r_{ij}}\right)^{12}, \quad \mathbf{F}_i = -\frac{\partial U}{\partial \mathbf{r}_{ij}} = \sum_{r_{ij}<2\sigma} 48\epsilon\left(\frac{\sigma}{r_{ij}}\right)^{12}\frac{\mathbf{r}_{ij}}{r_{ij}^2} \quad \text{(I1, I}$$

$$\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i, \quad r_{ij} = |\mathbf{r}_{ij}|$$

## 3.2. Simulation code for interacting Brownian particles with animation

### Import libraries

In [6]:
```
% matplotlib nbagg
import numpy as np # import numpy library as np
import matplotlib.pyplot as plt # import pyplot library as plt
import matplotlib.animation as animation # import animation modules from
from   mpl_toolkits.mplot3d import Axes3D # import Axes3D from mpl_toolk
from   numpy import newaxis # import newaxis used for inter-particle for
plt.style.use('ggplot') # use "ggplot" style for graphs
```

### Define `init` function for `FuncAnimation`

In [7]:
```
def init():
    global R,V,W,F,Rs,Vs,Ws,time # define global variables
    initconf()          # create random particle configuration without c
    V[:,:] = 0.0        # initialize all the variables to zero
    W[:,:] = 0.0        # initialize all the variables to zero
    F[:,:] = 0.0        # initialize all the variables to zero
    Rs[:,:,:] = 0.0     # initialize all the variables to zero
    Rs[0,:,:] = R[:,:]  # store initial particle positions in Rs
    Vs[:,:,:] = 0.0     # initialize all the variables to zero
    Ws[:,:,:] = 0.0     # initialize all the variables to zero
    time[:]   = 0.0     # initialize all the variables to zero
    title.set_text(r'') # empty title
    line.set_data([],[]) # set line data to show the trajectory of parti
    line.set_3d_properties([]) # add z-data separately for 3d plot
    particles.set_data([],[]) # set position current (x,y) position data
    particles.set_3d_properties([]) # add current z data of particles to
    return particles,title,line # return listed objects that will be dra
```

## Define `animate` function for `FuncAnimation`

The Euler method used in previous `animate` function is replaced with the Leap-Frog method in the following new `animate` function because of the numerical instability taking place with the former method for the present case of interacting Brownian particles.

In [8]:
```
def animate(i):
    global R,V,W,F,Rs,Vs,Ws,time # define global variables
    time[i]=i*dt # store time in each step in an array time
    particleforces() # compute inter-particle force F by examining all n
    W = std*np.random.randn(nump,dim) # generate an array of random forc
    V = V*(1-zeta/m*dt)+F/m*dt+W/m # update velocity via Eq.(I11) using
    R = R+V*dt # update position via Eq.(I12) using the L-F method
    Rs[i,:,:]=R # accumulate particle positions at each step in an array
    Vs[i,:,:]=V # accumulate particle velocitys at each step in an array
    Ws[i,:,:]=W # accumulate random forces at each step in an array Ws
    title.set_text(r"t = "+str(time[i])+"/"+str((nums-1)*dt))  # set the
    line.set_data(Rs[:i+1,n,0],Rs[:i+1,n,1]) # set the line in 2D (x,y)
    line.set_3d_properties(Rs[:i+1,n,2]) # add z axis to set the line in
    particles.set_data(pbc(R[:,0],box[0]), pbc(R[:,1],box[1])) # set the
    particles.set_3d_properties(pbc(R[:,2],box[2])) # add z axis to set
    return particles,title,line # return listed objects that will be dra
```

## Newly defined functions for interacting Brownian particles

In [9]:
```
def pbc(r, lbox): # enforce Periodic Boundary Conditions for all positic
    return np.fmod(r+lbox,lbox)
def distance(r1,r2,lbox): # Compute distance vector R2 - R1 with PBC
    return r2-r1-np.around((r2-r1)/lbox)*lbox
def fij(r2,rij): # calculate Fij=dU/drij
#    f=-24*eps*(2*(r2/sig**2)**(-6)-(r2/sig**2)**(-3))/r2*rij # Lennard-
    f=-48*eps*((r2/sig**2)**(-6))/r2*rij # soft-core potential
    return f
def particleforces(): # compute inter-particle force F by examining all
    global F
    F[:,:] = 0.0
    for n in range(nump): # repeat below for all particles
        rij = distance(R[n,:], R, box) # distance vectors rij=R_i-R_j fc
        r2  = np.linalg.norm(rij, axis=1)**2 # square distance rij**2
        nei = (r2 < (2.0*sig)**2)  # list neighbor particles of j
        nei[n] = False # ignore self pair (i=j)
        F[n,:] = np.sum(fij(r2[nei, newaxis], rij[nei,:]), axis=0) # tot
def initconf():  # create random particle configuration without overlapp
    global R,V,W,F,Rs,Vs,Ws,time
    for n in range(nump): # repeat below from n=0 to nump-1
        nn=0 # set overlap true to perform while loop below for the n-th
        while nn == 0: # repeat the loop below while overlap is true (nn
            R[n,:]=np.random.rand(dim)*box # generate a position candida
            nn = 1 # initialize overlap as false
            for l in range(n): # examine overlap generated positions (fr
                rij = distance(R[n,:],R[l,:],box) # calculate distance v
                r2  = np.linalg.norm(rij)**2 # calculate the squared dis
                if r2 < (0.90*sig)**2: # check if the distance is smalle
                # Yes -> perform below (nn=0) -> repeat while loop, No (
                    nn = 0 # set overlap true
```

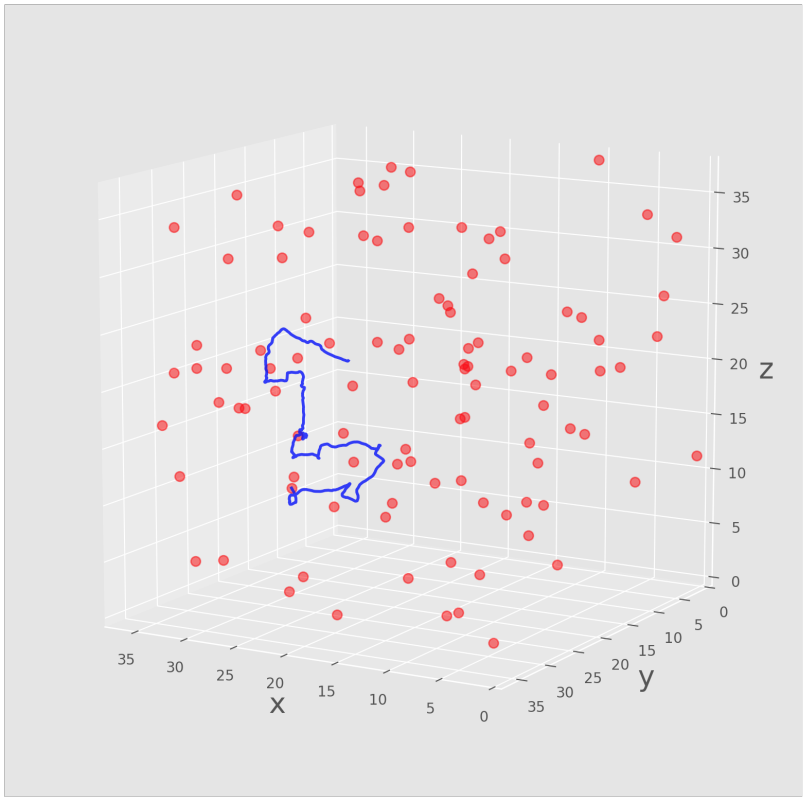
## Set parameters and initialize variables

In [10]:
```
dim  = 3    # system dimension (x,y,z)
nump = 100  # number of interacting Brownian particles to simulate
nums = 4096 # number of simulation steps
dt   = 0.01 # set time increment, \Delta t
zeta = 1.0  # set friction constant, \zeta
m    = 1.0  # set particle mass, m
kBT  = 1.0  # set thermal energy, k_B T
std  = np.sqrt(2*kBT*zeta*dt) # calculate std for \Delta W via Eq.(F11)
sig  = 1.0  # unit of length of inter-particle potential
eps  = 1.0  # unit of energy inter-particle potential
vf   = 0.001 ##volume fraction of particles < 0.45
boxl = np.power(nump*np.pi/6/vf,1/3) # calculate the side length of unit
print('Volume fraction =',vf,'  boxl =',boxl) # print vf and boxl
box  = np.array([boxl,boxl,boxl])*sig # set array box[dim]
np.random.seed(0) # initialize random number generator with a seed=0
R = np.zeros([nump,dim]) # array to store current positions and set init
V = np.zeros([nump,dim]) # array to store current velocities and set ini
W = np.zeros([nump,dim]) # array to store current random forcces
F = np.zeros([nump,dim]) # rray to store current particle orcces
Rs = np.zeros([nums,nump,dim]) # array to store positions at all steps
Vs = np.zeros([nums,nump,dim]) # array to store velocities at all steps
Ws = np.zeros([nums,nump,dim]) # array to store random forces at all ste
time = np.zeros([nums]) # an array to store time at all steps
```

```
Volume fraction = 0.001   boxl = 37.4110192682
```

## Perform and animate the simulation using `FuncAnimation`

In [11]:
```
fig = plt.figure(figsize=(10,10)) # set fig with its size 10 x 10 inch
ax = fig.add_subplot(111,projection='3d') # creates an additional axis t
ax.set_xlim(0.0,box[0]) # set x-range
ax.set_ylim(0.0,box[1]) # set y-range
ax.set_zlim(0.0,box[2]) # set z-range
ax.set_xlabel(r"x",fontsize=20) # set x-lavel
ax.set_ylabel(r"y",fontsize=20) # set y-lavel
ax.set_zlabel(r"z",fontsize=20) # set z-lavel
ax.view_init(elev=12,azim=120)  # set view point
particles, = ax.plot([],[],[],linestyle='None',color='r',marker='o',ms=2
title = ax.text(0.,0.,0.,r'',transform=ax.transAxes,va='center') # defin
line, = ax.plot([],[],[],'b',lw=2,alpha=0.8) # define object line
n = 0  # trajectry line is plotted for the n-th particle
anim = animation.FuncAnimation(fig,func=animate,init_func=init,
            frames=nums,interval=5,blit=True,repeat=False)
## If you have ffmpeg installed on your machine
## you can save the animation by uncomment the last line
## You may install ffmpeg by typing the following command in command prc
## conda install -c menpo ffmpeg
##
#anim.save('movie.mp4',fps=20,dpi=400)
```

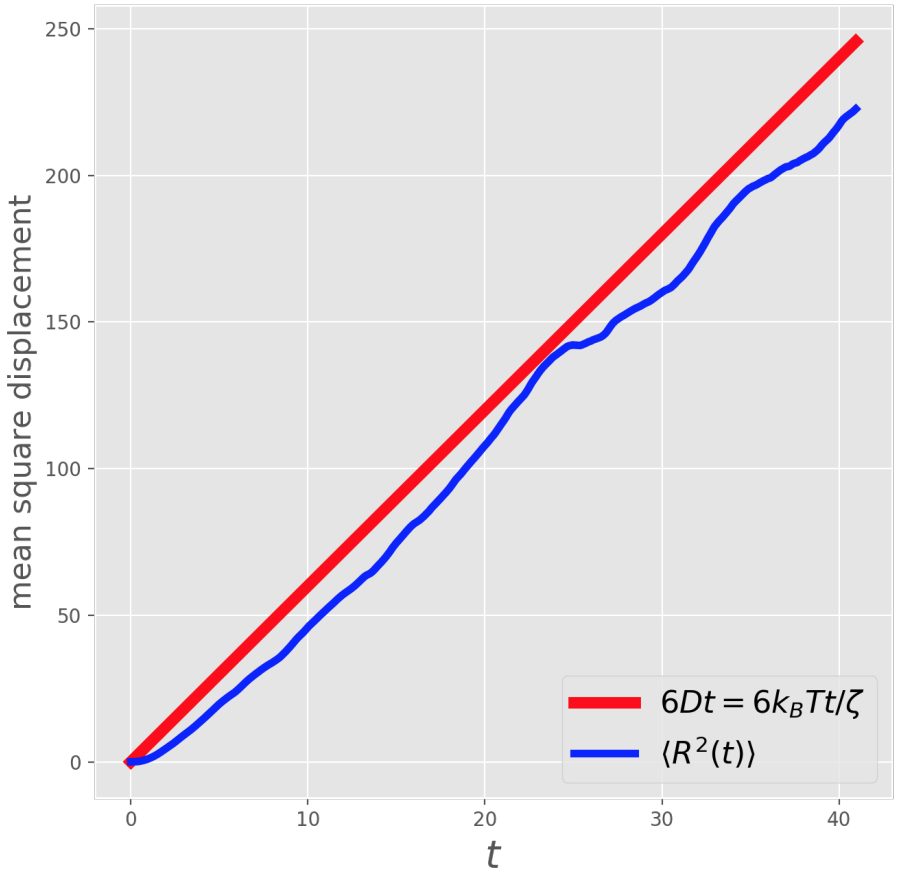



t = 40.95/40.95

## 3.3. The mean square displacement and the diffusion constant

In [12]:
```
# mean square displacement vs time
msd = np.zeros([nums])
for i in range(nums):
    for n in range(nump):
        msd[i]=msd[i]+np.linalg.norm(Rs[i,n,:]-Rs[0,n,:])**2 # (R(t) - R
    msd[i] = msd[i]/nump
dmsd = np.trapz(msd, dx=dt)/(3*(nums*dt)**2)
print('D_0 =',kBT/zeta,'(Theoretical)')
print('D =',dmsd,'(Simulation via MSD)')
print('Volume fraction =',vf) ### print vf
fig, ax = plt.subplots(figsize=(7.5,7.5))
ax.set_xlabel(r"$t$", fontsize=20)
ax.set_ylabel(r"mean square displacement", fontsize=16)
ax.plot(time,6*kBT/zeta*time,'r',lw=6,label=r'$6Dt={6k_BT t}/{\zeta}$')
ax.plot(time,msd,'b',lw=4, label=r'$\langle R^2(t)\rangle$')
ax.legend(fontsize=16, loc=4)
plt.show()
```

```
D_0 = 1.0 (Theoretical)
D = 0.885552952684 (Simulation via MSD)
Volume fraction = 0.001
```



# References


- The SciPy.org website, https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html (https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html)
- Listed in the major unsolved problems in physics, https://en.wikipedia.org/wiki/List_of_unsolved_problems_in_physics#Condensed_matter_physics (https://en.wikipedia.org/wiki/List_of_unsolved_problems_in_physics#Condensed_matter_physics)


## Summary of simulation method

### Original differential equation

$i$: particle index

$$\frac{d\mathbf{R}_i(t)}{dt} = \mathbf{V}_i(t) \tag{I3}$$

$$m\frac{d\mathbf{V}_i(t)}{dt} = -\zeta\mathbf{V}_i(t) + \mathbf{F}_i(t) + \mathbf{F}_R(t) \tag{I4}$$

with

$$\langle\mathbf{F}_R(t)\rangle = \mathbf{0} \tag{I5}$$

$$\langle\mathbf{F}_R(t)\mathbf{F}_R(0)\rangle = 2k_BT\zeta\mathbf{I}\delta(t) \tag{I6}$$

### Euler method

$i$: index of particles, $n$: time step

$$\mathbf{V}_{i,n+1} = \left(1 - \frac{\zeta}{m}\Delta t\right)\mathbf{V}_{i,n} + \frac{1}{m}\mathbf{F}_{i,n}\Delta t + \frac{1}{m}\Delta\mathbf{W}_n \tag{I7}$$

$$\mathbf{R}_{i,n+1} = \mathbf{R}_{i,n} + \mathbf{V}_{i,n}\Delta t \tag{I8}$$

with

$$\langle\Delta\mathbf{W}_n\rangle = \mathbf{0} \tag{I9}$$

$$\langle\Delta\mathbf{W}_n\Delta\mathbf{W}_l\rangle = 2k_BT\zeta\Delta t\mathbf{I}\delta_{nl} \tag{I10}$$

### Leap-Frog method

$i$: index of particles, $n$: time step

$$\mathbf{V}_{i,n+\frac{1}{2}} = \left(1 - \frac{\zeta}{m}\Delta t\right)\mathbf{V}_{i,n-\frac{1}{2}} + \frac{1}{m}\mathbf{F}_{i,n}\Delta t + \frac{1}{m}\Delta\mathbf{W}_n \tag{I11}$$

$$\mathbf{R}_{i,n+1} = \mathbf{R}_{i,n} + \mathbf{V}_{i,n+\frac{1}{2}}\Delta t \tag{I12}$$

with

$$\langle\Delta\mathbf{W}_n\rangle = \mathbf{0} \tag{I13}$$

$$\langle\Delta\mathbf{W}_n\Delta\mathbf{W}_l\rangle = 2k_BT\zeta\Delta t\mathbf{I}\delta_{nl} \tag{I14}$$

### 2nd order Runge-Kutta method

$i$: index of particles, $n$: time step

$$\mathbf{V}'_{i,n+\frac{1}{2}} = \mathbf{V}_{i,n} - \frac{\zeta}{m}\frac{\Delta t}{2}\mathbf{V}_{i,n} + \frac{1}{m}\mathbf{F}_{i,n}\frac{\Delta t}{2} = \left(1 - \frac{\zeta}{m}\frac{\Delta t}{2}\right)\mathbf{V}_{i,n} + \frac{1}{m}\mathbf{F}_{i,n}\frac{\Delta t}{2} \tag{I15}$$

$$\mathbf{R}'_{i,n+\frac{1}{2}} = \mathbf{R}_{i,n} + \mathbf{V}_{i,n}\frac{\Delta t}{2} \tag{I16}$$

$$\mathbf{V}_{n,i+1} = \mathbf{V}_{i,n} - \frac{\zeta}{m}\Delta t\mathbf{V}'_{n,i+\frac{1}{2}} + \frac{1}{m}\Delta t\mathbf{F}_{i,n+\frac{1}{2}} + \frac{1}{m}\Delta\mathbf{W}_i \tag{I17}$$

$$\mathbf{R}_{n,i+1} = \mathbf{R}_{n,i} + \mathbf{V}'_{n,i+\frac{1}{2}}\Delta t \tag{I18}$$

with

$$\langle\Delta\mathbf{W}_n\rangle = \mathbf{0} \tag{I19}$$

$$\langle\Delta\mathbf{W}_n\Delta\mathbf{W}_l\rangle = 2k_BT\zeta\Delta t\mathbf{I}\delta_{nl} \tag{I20}$$