Assignment_3_Notebook

September 28, 2017

Here we create our molecular dynamics simulation. We will simulate a number of particles in a grid of size L.

We will observe phase transitions as we change the size of the box which basically increases the pressure on the system.

We can also extract correlation functions to observe the overall structure of the system.

In [3]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import math as math

#Intialize positions and velocities of system
def initialize_positions_and_velocities(rx,ry,vx,vy, Nx, Ny,L):
  dx=L/Nx;
  dy=L/Ny;
  np.random.seed(0)
  for i in range(Nx):
    for j in range(Ny):
      rx[i*Ny+j]=dx*(i+0.5)
      ry[i*Ny+j]=dy*(j+0.5)

      u=np.random.random() #This is box muller
      v=np.random.random()
      vx[i*Ny+j]=np.sqrt(-2*np.log(u))*np.cos(2.*np.pi*v)
      vy[i*Ny+j]=np.sqrt(-2*np.log(u))*np.sin(2.*np.pi*v)
  #subtract net velocity to avoid global drift
  vxav=sum(vx)/vx.size
  vyav=sum(vy)/vx.size
  vx-=vxav
  vy-=vyav

#Compute F_r = - dU/dr
def force(rsq):
  r = rsq**(1/2)
  rinv = 1./r
  return 4*(12*(rinv**13) - 6*(rinv**7))
```

```
def potential(rsq):
  rsqinv=1./rsq
  r6inv=rsqinv*rsqinv*rsqinv
  return -4*r6inv*(1-r6inv)

def compute_kinetic_energy(vx,vy):
  return 0.5*sum(vx*vx+vy*vy)

#computes potential energy
def compute_potential_energy(rx,ry,rcut,L):
  rcutsq=rcut*rcut
  rcutv=potential(rcutsq) #shift the potential to avoid jump at rc
  Epot=0.
  for i in range(rx.size):
    for j in range(i):
      dx=rx[i]-rx[j]
      dy=ry[i]-ry[j]
      #minimum image convention
      if(dx > L/2.): dx=dx-L
      if(dx <-L/2.): dx=dx+L
      if(dy > L/2.): dy=dy-L
      if(dy <-L/2.): dy=dy+L
      #print dx,dy
      #compute the distance
      rsq=dx*dx+dy*dy
      if(rsq < rcutsq and rsq != 0):
        Epot+=potential(rsq)-rcutv
  return Epot

#Compute the forces from each particle on the others, Use newtons third law.
def compute_forces(rx,ry,dV_drx, dV_dry, N, L, rcut):
  rcutsq=rcut*rcut
  for i in range(N):
    for j in range(i):
      dx=rx[i]-rx[j] ;
      dy=ry[i]-ry[j] ;
      #minimum image convention
      if(dx > L/2.): dx=dx-L
      if(dx <-L/2.): dx=dx+L
      if(dy > L/2.): dy=dy-L
      if(dy <-L/2.): dy=dy+L
      #compute the distance
      rsq=dx*dx+dy*dy
      #check if we are < the cutoff radius
      if(rsq < rcutsq and rsq != 0):
        #here is the call of the force calculation
        dV_dr=force(rsq)
```

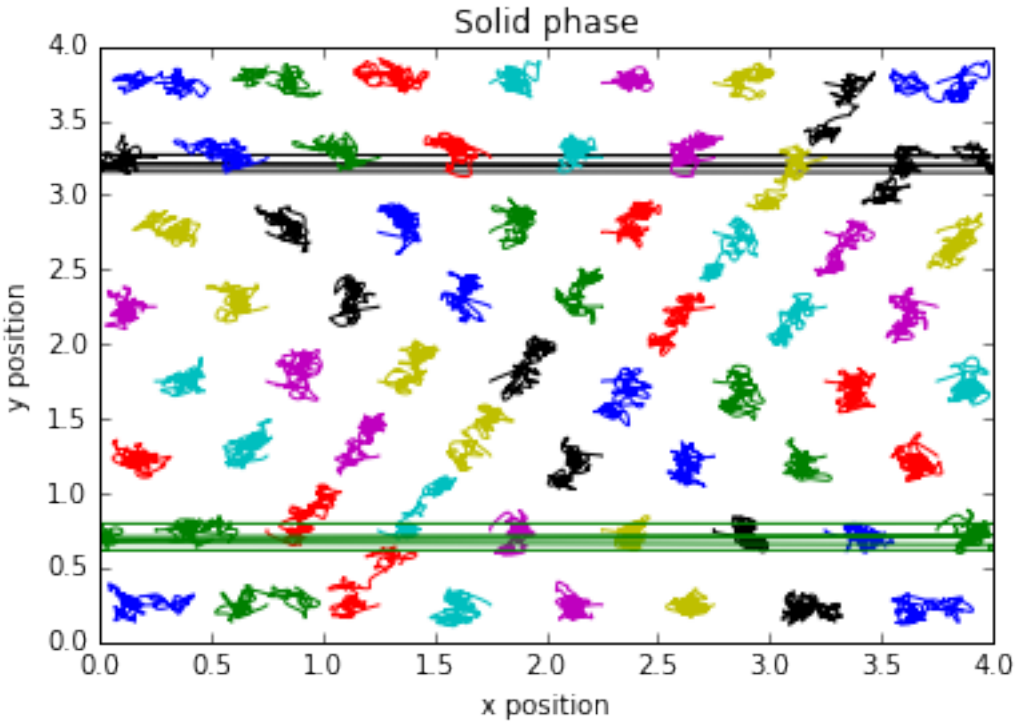
```
        #here the force is being added to the particle. Note the additional dx
        dV_drx[i]+=dx*dV_dr/np.sqrt(rsq)
        dV_drx[j]-=dx*dV_dr/np.sqrt(rsq)
        dV_dry[i]+=dy*dV_dr/np.sqrt(rsq)
        dV_dry[j]-=dy*dV_dr/np.sqrt(rsq)

#Euler algorithm to step particles forward in time
def euler(deltat,rx,ry,vx,vy,dV_drx,dV_dry):
  #update the positions
  rx+=deltat*vx
  ry+=deltat*vy

  #update the velocities
  vx+=deltat*dV_drx
  vy+=deltat*dV_dry

#Velocity verlet algorithm, improves on Euler method
def verlet(deltat, rx, ry, vx, vy, dV_drx, dV_dry, N, L, rcut):

  rx+= deltat*vx + .5*deltat*deltat*dV_drx
  ry+= deltat*vy + .5*deltat*deltat*dV_dry

  #Make sure to rebox!
  rebox(rx,ry,L);

  #Create new force array
  dV_drx_new = np.zeros(N)
  dV_dry_new = np.zeros(N)

  #Recompute forces from updates positions
  compute_forces(rx, ry, dV_drx_new, dV_dry_new, N, L, rcut);

  #compute the change in velocities, notice I changed this to a plus because I use F_r =
  vx+=.5*(dV_drx_new + dV_drx)*deltat
  vy+=.5*(dV_dry_new + dV_dry)*deltat


 #put back into box:
def rebox(rx,ry,L):
  for i in range(rx.size):
    if rx[i] > L:
      rx[i]=np.mod(rx[i],L)
    if rx[i] < 0:
      rx[i]=np.mod(rx[i],L)
    if ry[i] > L:
      ry[i]=np.mod(ry[i],L)
    if ry[i] < 0:
      ry[i]=np.mod(ry[i],L)
```

```
#Print out long list of positions and velocities of all particles
def print_result(rxlog,rylog,vxlog,vylog):
  fr=open("positions.dat",'w')
  fv=open("velocities.dat",'w')

  for j in range(rxlog.shape[1]):
    for i in range(rxlog.shape[0]):
      fr.write(str(rxlog[i,j])+" "+str(rylog[i,j])+'\n')
      fv.write(str(vxlog[i,j])+" "+str(vylog[i,j])+'\n')
    fr.write('\n')
    fv.write('\n')
```

Here we will use two different box sizes and observe the trajectory of the particles in the solid and gas phase.

```
In [7]: #simulation parameters
        Nx=8; Ny=8; N=Nx*Ny #set particles onto a grid initially
        L=4
        Nstep=500
        rcut=2.5 # a usual choice for the cutoff radius
        deltat = 0.0001

        vx=np.zeros(N)
        vy=np.zeros(N)
        rx=np.zeros(N)
        ry=np.zeros(N)

        rxlog=np.zeros([Nstep,N])
        rylog=np.zeros([Nstep,N])
        vxlog=np.zeros([Nstep,N])
        vylog=np.zeros([Nstep,N])

        initialize_positions_and_velocities(rx,ry,vx,vy,Nx,Ny,L)

        for i in range(Nstep):
            dV_drx=np.zeros(N)
            dV_dry=np.zeros(N)
            compute_forces(rx,ry,dV_drx,dV_dry, N, L, rcut)

            #propagate using forward Euler
            #euler(deltat,rx,ry,vx,vy,dV_drx,dV_dry)
            # Replace the FW Euler with a velocity verlet
            verlet(deltat,rx,ry,vx,vy,dV_drx,dV_dry, N, L, rcut)

            #make sure we're still in the box
            rebox(rx,ry,L)
```

```
        #keep track for printing
        rxlog[i]=rx
        rylog[i]=ry
        vxlog[i]=vx
        vylog[i]=vy

        #get some observables
        Epot=compute_potential_energy(rx,ry,rcut,L)
        Ekin=compute_kinetic_energy(vx,vy)
        #print(i,Epot,Ekin,Epot+Ekin)

print_result(rxlog,rylog,vxlog,vylog)

plt.plot(rxlog, rylog)
plt.xlim([0,L])
plt.ylim([0,L])
plt.xlabel('x position')
plt.ylabel('y position')
plt.title('Solid phase')
plt.show()
```



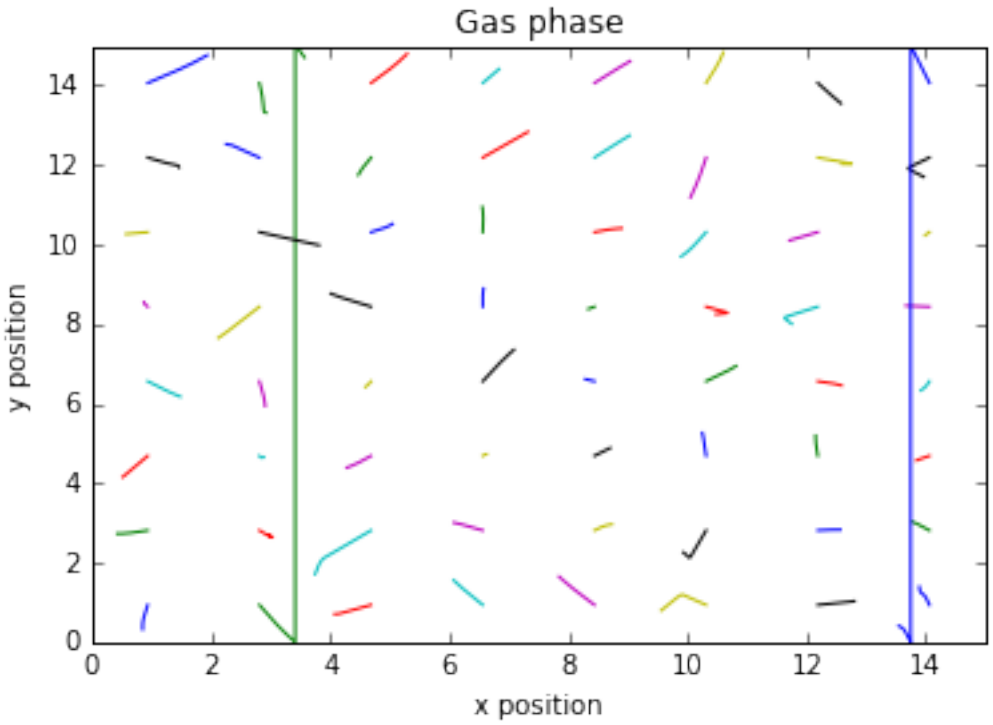
Here we have created a system of 64 particles interacting with the 12-6 potential and a box size of 4x4.

You can see the system is in the solid phase as the particles stay in the neighborhood of their initial position and do not move far before changing direction. The large lines going across the plot are just an artifact of matplotlibs plots as some particles move very close the edge of the walls and so spend some time on both sides.

Similarly if we make the box size larger (L = 15), we effectively reduce the pressure and we can see our system is now in the gas phase.

```
In [8]: #simulation parameters
        Nx=8; Ny=8; N=Nx*Ny #set particles onto a grid initially
        L=15
        Nstep=5000
        rcut=2.5 # a usual choice for the cutoff radius
        deltat = 0.0001

        vx=np.zeros(N)
        vy=np.zeros(N)
        rx=np.zeros(N)
        ry=np.zeros(N)

        rxlog=np.zeros([Nstep,N])
        rylog=np.zeros([Nstep,N])
        vxlog=np.zeros([Nstep,N])
        vylog=np.zeros([Nstep,N])

        initialize_positions_and_velocities(rx,ry,vx,vy,Nx,Ny,L)

        for i in range(Nstep):
            dV_drx=np.zeros(N)
            dV_dry=np.zeros(N)
            compute_forces(rx,ry,dV_drx,dV_dry, N, L, rcut)

            #propagate using forward Euler
            #euler(deltat,rx,ry,vx,vy,dV_drx,dV_dry)
            # Replace the FW Euler with a velocity verlet
            verlet(deltat,rx,ry,vx,vy,dV_drx,dV_dry, N, L, rcut)

            #make sure we're still in the box
            rebox(rx,ry,L)

            #keep track for printing
            rxlog[i]=rx
            rylog[i]=ry
            vxlog[i]=vx
            vylog[i]=vy

            #get some observables
            Epot=compute_potential_energy(rx,ry,rcut,L)
            Ekin=compute_kinetic_energy(vx,vy)
```

```
        #print(i,Epot,Ekin,Epot+Ekin)

print_result(rxlog,rylog,vxlog,vylog)

plt.plot(rxlog, rylog)
plt.xlim([0,L])
plt.ylim([0,L])
plt.xlabel('x position')
plt.ylabel('y position')
plt.title('Gas phase')
plt.show()
```



We see the particles move in mostly straight lines, rarely interacting with eachother, so the mean free path is much larger.

Correlation functions provide a measure of order and structure in a system. We can measure the correlation function by looping over all particles and choosing a radius r and thickness dr. We count all the particles within a disk of radius r to r + dr of the given particle. We then do this for all of our particles, and repeat for many values of r, and take a time average over the lifetime of our system. This is called the angle integrate radial correlation function. We will compute it for both the solid and gas phase. We will compute it for both the solid and gas phase.

```
In [ ]: def gcorr(dr, upper_limit, Nstep, L):

            radii = np.linspace(0.1,upper_limit,59)
```

```
gr = np.zeros(len(radii), float)

for r in range(0,len(radii)):
  for i in range(0,Nstep):
    for j in range(0,N):
      for k in range(0,N):
        if j != k:
          dx = rxlog[i,j] - rxlog[i,k]
          dy = rylog[i,j] - rylog[i,k]

          if(dx > L/2.): dx=dx-L
          if(dx <-L/2.): dx=dx+L
          if(dy > L/2.): dy=dy-L
          if(dy <-L/2.): dy=dy+L


          dist = np.sqrt((dx)**2 + (dy)**2)
          if dist >= radii[r] and dist <= radii[r] + dr:
            gr[r]+= 1

  gr[r] = gr[r]/(N*2*(math.pi)*radii[r]*dr*(25/L*L)*Nstep)

plt.plot(radii, gr)
plt.show()
```

In [ ]:

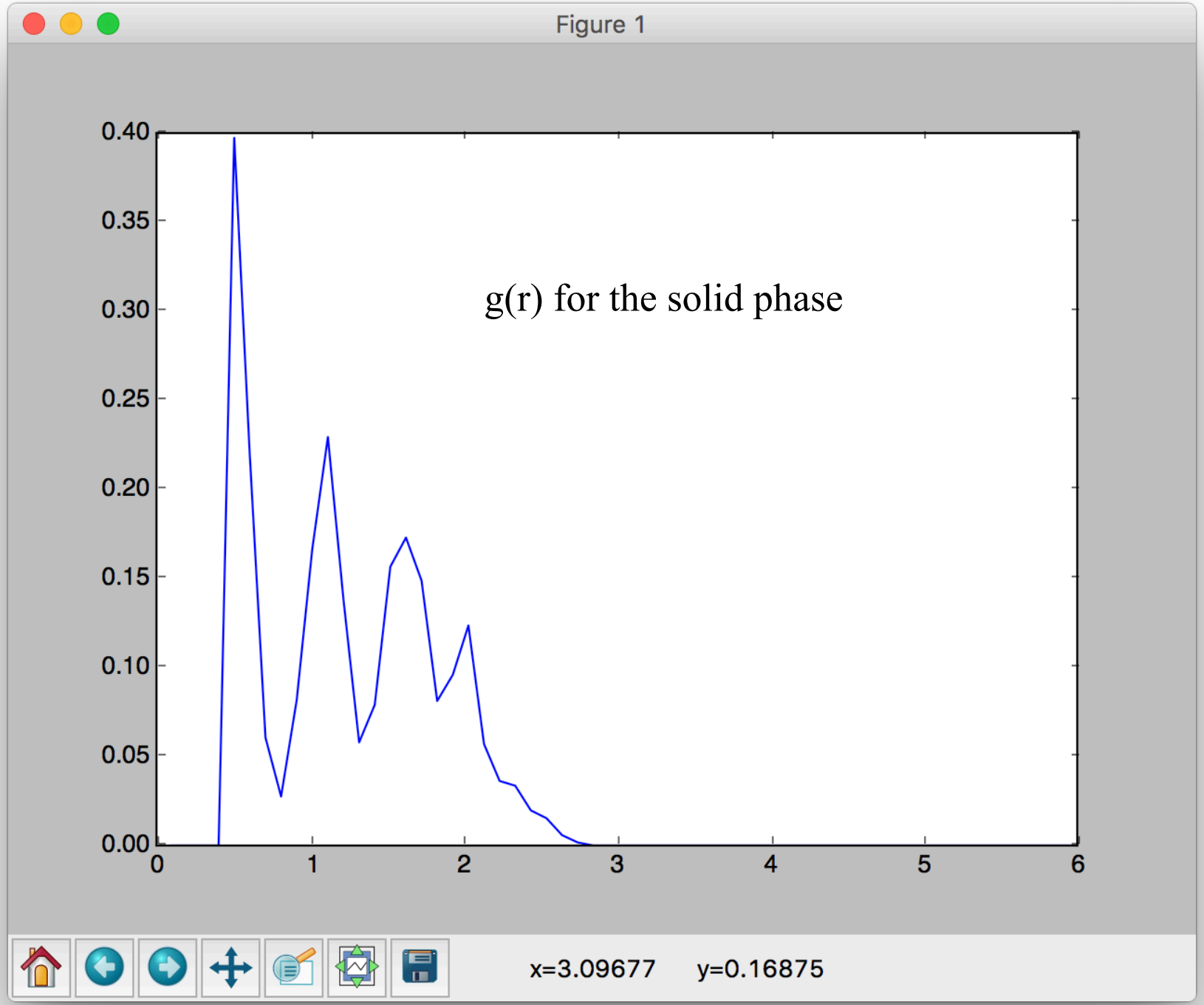


Figure 1
g(r) for the solid phase
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
1
2
3
4
5
6
x=3.09677
y=0.16875

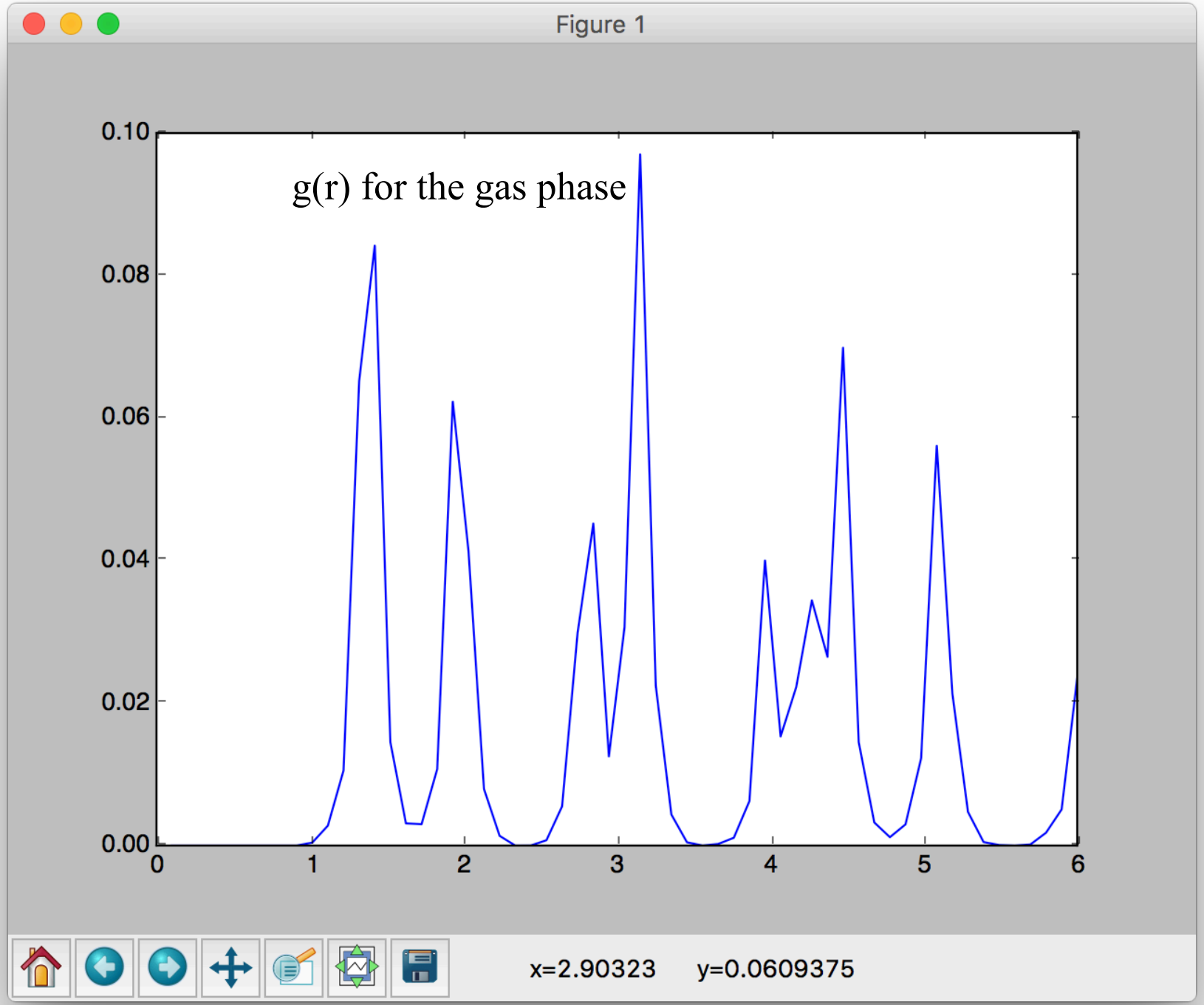


Figure 1
g(r) for the gas phase
0.10
0.08
0.06
0.04
0.02
0.00
0
1
2
3
4
5
6
x=2.90323
y=0.0609375