Lab12/CW7: Gas, Pressure and Mean-free-path

Objectives

In Chapter S1 of *Matter & Interactions 4e* you have learned about the ideal gas and Maxwell distribution of velocities of the gas molecules. In this activity we will simulate a gas system consisted 50 He atoms in a box of side length L at temperature T. Your mission is to find the pressure, and the mean free path between collisions of the atoms.

1 Numpy arrays

In this lab, we will use data structure and commands from the Numerical Python (numpy) package for efficiency. Vpython is built on top of numpy so no further explicit import is necessary. You can find numpy documentations at http://devdocs.io/numpy~1.12/.

The main object in numpy is the numpy array, which is a (usually fixed-size) multidimensional container of items of the same type and size. The number of dimensions and items in an array is defined by its `shape`, which is a tuple of N positive integers that specify the sizes of each dimension.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

```
from __future__ import division, print_function
import numpy as np
a = np.array([1, 2, 3])  # Create a rank 1 array
print(type(a))
print(a.shape)
print(a[0], a[1], a[2])
a[0] = 5
print(a)

b = np.array([[1,2,3],[4,5,6]])   # Create a rank 2 array
print(b.shape)
print(b[0, 0], b[0, 1], b[1, 0])
```

Numpy also provides many functions to create arrays:

```
from __future__ import division, print_function
import numpy as np

a = np.zeros((2,2))  # Create an array of all zeros
print(a)             # Prints "[[ 0.  0.]
                     #          [ 0.  0.]]"

b = np.ones((1,2))   # Create an array of all ones
print(b)             # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7) # Create a constant array
print(c)              # Prints "[[ 7.  7.]
                      #          [ 7.  7.]]"

d = np.eye(2)        # Create a 2x2 identity matrix
print(d)             # Prints "[[ 1.  0.]
                     #          [ 0.  1.]]"

e = np.random.random((2,2)) # Create an array filled with random values
print(e)                    # Might print "[[ 0.91940167  0.08143941]
                            #               [ 0.68744134  0.87236687]]"
```

```
np.greater_equal([4, 2, 1], [2, 2, 2])
array([ True, True, False], dtype=bool)
```

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

```
from __future__ import division, print_function
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

To compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices, use `dot`. `dot` is available both as a function in the numpy module and as an instance method of array objects:

```
from __future__ import division, print_function
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
```

```
# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

2 Handle collisions between atoms

In this simulation, one important aspect is to handle collisions between atoms.
Here is an efficient code snippet that can be used to detect the collisions, where `pos` is a numpy array of size of $N \times 3$ which contains the positions of the atoms after the update, and `radius` is an $N \times 1$ numpy array which stores the radii of the atoms,

```
r = pos-pos[:,newaxis] # all pairs of atom-to-atom vectors
rmag = sqrt(sum(square(r),-1)) # atom-to-atom scalar distances
hit = less_equal(rmag,radius+radius[:,newaxis])-identity(N)
hitlist = sort(nonzero(hit.flat)[0]).tolist() # i,j encoded as i*N+j
```

The first line generates an $N \times N$ matrix of inter-atomic vectors between atom i and j (`r` is an $N \times N \times 3$ numpy array).
The second line generates an $N \times N$ matrix of the inter-atomic distances.
The third line uses a numpy function `less_equal` to determine if a collision occurs. `hit` is an $N \times N$ matrix where `hit[i,j]` indicates if a collision occurred between atom i and j. The fourth line find the indices of nonzero elements in `hit` by flattening it to a 1d array first. The flattening stores the element `hit[i,j]` to the element `i*N+j`

You will loop through `hitlist` to handle the collisions. Notice that `hitlist` stores collision between atoms i and j as `[i,j]` and `[j,i]`. When handling collision, you want to remove one of them first so that you don't handle the collision twice.

If a collision is detected, you want to trace back in time to when the two atoms make contacts, handles the bound, and move forward in time using the new velocities.

Here is a code snippet that handles the collision between two atoms. The velocities can be obtained by transforming into the center-of-mass frame, handling the bounce and transforming back to the lab frame.

```
def vcollision(a1,a2):
    '''
    Function to find the velocities of atoms after each collision
    '''
    v1 = a1.v - 2 * a2.m/(a1.m+a2.m) *(a1.pos-a2.pos)\
     * dot (a1.v-a2.v, a1.pos-a2.pos) / abs(a1.pos-a2.pos)**2
    v2 = a2.v - 2 * a1.m/(a1.m+a2.m) *(a2.pos-a1.pos) \
    * dot (a2.v-a1.v, a2.pos-a1.pos) / abs(a2.pos-a1.pos)**2
    return v1, v2
```

3 Handle collisions between atoms and walls

The collisions between atoms and walls can be easily handled by checking if the particle hits the wall.
Here is a code snippet that handles the collisions with the walls,

```
outside = less_equal(pos,Ratom) # walls closest to origin
p1 = p*outside
p = p-p1+abs(p1) # force p component inward
outside = greater_equal(pos,L-Ratom) # walls farther from origin
p1 = p*outside
p = p-p1-abs(p1) # force p component inward
```

⇒ Find the pressure of the system. You should find the pressure is much larger than the pressure of an ideal gas at this temperature and density. The reason is that we use larger size of atoms. You can modify the atom size to the real size of He atom of 31 pm (31E-12) and get a better result, but then the atoms rarely collide and it takes very long time to reach the equilibrium state.

⇒ Run the simulation long enough that the gas reaches equilibrium. The speed distribution histogram approaches the theoretical Maxwell distribution.

⇒ Add some codes to find the mean free path of the atoms. Free path is the distance one atom travels between two consecutive collisions with other atoms (this means that the collisions of the atoms with the walls are not considered for calculating the free path). Obtain every free path of all atoms, average them, and print the mean free path every 1000*dt. Compare your mean free path to the theoretical result $l = \frac{V}{\sqrt{2}\pi d^2 N}$, where V is the volume of the container, N the number of particles contained, and d is the diameter of the atoms.
