Institute for Theoretical Physics
University of Cologne

Prof. Dr. Simon Trebst
Peter Bröcker, Johannes Helmes

Computational Many-Body Physics

Assignment 4

Summer Term 2015

website: http://www.thp.uni-koeln.de/trebst/Lectures/2015-CompManyBody.shtml
due date: Monday, June 15th, 18:00 - send solutions to helmes [at] thp.uni-koeln.de

9. Need for speed

Programming technique

You have probably experienced during the last exercises that pure python code may yield a low (speed) performance. Often, identical operations need to be done on many elements of lists or arrays. Here we will present a few tricks that will speed up such code elements.

List slicing Python is very powerful for handling lists. The built-in routines are implemented by the python developers in a much more efficient way than doing the same operations using loops. The basic syntax is `numbers[start:stop:step]` which generates and returns a new list with the described part. The following example regroups a list into a list of pairs. The execution time is measured for both approaches. Download and run the code and compare the efficiency.

```
import numpy as np
from timeit import Timer

def pairing_with_loop(numbers):
    pairs=[]
    for i in range(0,len(numbers),2):
        pairs.append([numbers[i],numbers[i+1]])
    return pairs

def pairing_with_slicing(numbers):
    return zip(numbers[0::2],numbers[1::2])

rns=np.random.random(size=100000)
t1 = Timer(lambda: pairing_with_loop(rns))
t2 = Timer(lambda: pairing_with_slicing(rns))
print t1.timeit(number=100), "vs.",t2.timeit(number=100)
```

NumPy array operations The capabilities of numpy arrays not only include storing and accessing data but also performing arithmetic operations simultaneously on all their elements. Observe the superiority of their usage in this example where the sum of two arrays and the logarithm of all elements is determined.

```
import numpy as np
from timeit import Timer
from math import log

def add_with_loop(l1,l2):
```

```
    sum=np.empty(len(l1))
    for i in range(len(l1)):
        sum[i]=l1[i]+l2[i]
    return sum

def add_with_numpy(l1,l2):
    return l1+l2

def ln_with_loop(l):
    logs=np.empty(len(l))
    for i in range(len(l)):
        logs[i]=log(l[i])
    return logs

def ln_with_numpy(l):
    return np.log(l)

rns1=np.random.random(size=10000)
rns2=np.random.random(size=10000)
t1 = Timer(lambda: add_with_loop(rns1,rns2))
t2 = Timer(lambda: add_with_numpy(rns1,rns2))
t3 = Timer(lambda: ln_with_loop(rns1))
t4 = Timer(lambda: ln_with_numpy(rns1))
print t1.timeit(number=100), "vs.",t2.timeit(number=100)
print t3.timeit(number=100), "vs.",t4.timeit(number=100)
```

NumPy routines with a `size` keyword argument It is always a good idea to have a look in the numpy documentation and check if the library provides a routine for your specific problem. For example, many random numbers can be created at once when using the keyword argument `size=...` Also, diagonal matrices can be created in a single line. Have a look at this code to see their advantage.

```
import numpy as np
from timeit import Timer

def create_random_with_loop(N):
    rn=np.empty(N)
    for i in range(N):
        rn[i]=np.random.random()
    return rn

def create_random_with_numpy(N):
    return np.random.random(size=N)

def create_diag_with_loop(entries):
    diag_mat=np.zeros((N,N))
    for i in range(len(entries)):
        diag_mat[i,i]=entries[i]
    return diag_mat

def create_diag_with_numpy(entries):
    return np.diag(entries)


N=10000
t1 = Timer(lambda: create_random_with_loop(N))
```

```
t2 = Timer(lambda: create_random_with_numpy(N))
rns=np.random.random(size=N)
t3 = Timer(lambda: create_diag_with_loop(rns))
t4 = Timer(lambda: create_diag_with_numpy(rns))
print t1.timeit(number=100), "vs.",t2.timeit(number=100)
print t3.timeit(number=100), "vs.",t4.timeit(number=100)
```

10. Molecular Dynamics Simulation of Argon gas *10 points*

In the last two exercises you have deepened your understanding of Monte Carlo simulations — an approach which *replaces* the physical dynamics of a system by an ad-hoc artificial dynamics yielding the Markov chain. In contrast, the molecular dynamics simulations to be applied in this assignment *mimic* the true dynamics of the system.

Molecular dynamics simulations are suited to investigate the collective behavior of a large number of interacting particles in continuous space. A position and a velocity is assigned to every particle (or molecule). The basic principle is to determine the net force acting on each particle and — using Newton's second law — to update the velocity of the particle. Subsequently, all particles are moved according to their current velocity and the chosen time discretization. This procedure is repeated in a long sequence of consecutive time steps. The dynamical behavior of the system can then be observed both qualitatively by visualizing the positions the particles in an animation and quantitatively by measuring observables such as temperature, average velocity or pair-correlation functions.

A naive approach of determining the effective force onto a particular particle is to sum up the forces exerted by the totality of all other particles. However, doing so would strongly restrict the number of particles N, because the number of mutual interacting forces to be considered would scale as $(N-1)+(N-2)+(N-3)+\ldots \sim O(N^2)$. Therefore, we apply an advanced strategy to cope with the large number of mutual forces which is appropriate for short ranged interactions, namely the **cell method**.

Your task will be to implement a molecular dynamics simulation at $T = 150K$ for the argon gas in two dimensions whose interaction is described by the Lennard-Jones potential

$$V(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{1}$$

with parameters $\varepsilon = 1.65 \times 10^{-21} J$ and $\sigma = 0.335$nm. The atomic mass of argon is $m = 40$u, and $k_B = 1.38 \times 10^{-23}$ J/K. Use the following units for the simulation: [Length] = 1nm, [Time] = 1ps, [Mass] = 1u and [Temperature] = 1K. You can use our cython skeleton for your code.

1. Set up a system with 100 atoms initially arranged on a square lattice with periodic boundary conditions. Draw the initial velocities from a Maxwell-Boltzmann distribution, which is a Gaussian distribution with $\sigma^2 = \frac{k_B T}{m}$. To generate the random numbers you can use the `np.random.normal(loc=mu,scale=sigma)` routine.
2. Implement a function that determines the interacting force between two atoms using the Lennard-Jones potential and a function for the update of positions and velocites using the Verlet method. Complete your code so that you obtain a working MD simulation with a

temporal discretization $\Delta t = 0.01 ps$. (Do not try to perform measurements at this stage!)

3. Apply and implement the cell method with 6×6 cells in order to reduce the number of interactions.

   The idea of the cell method is as follows: The system is divided into equally sized cells, so that each particle is contained in exactly one particular cell. For every particle, only interactions with those of the other particles are considered which reside in the same cell or one of the neighboring cells. Hence, for a two-dimensional system only 9 cells need to be considered. Technically, the method can be implemented using two lists **A** and **B**. The indices of **A** enumerate the cells and the indices of **B** enumerate all particles. **A** has one entry for every cell, namely the particle index of the first particle in this cell. (Which of the particles in the cell is *first* is arbitrary.) If a cell is empty, the corresponding list entry for that cell should by set to -1. **B** has one entry for every particle, namely the particle index of the *next* particle in the same cell. (Again, the ordering of the particles within the cell is arbitrary.) For the last particle, the list entry should be set to -1 in order to indicate that there is no next particle. Update the two lists only if a particle has moved to another cell. Only three list entries need to be changed in that case.

   To illustrate the method in a little example, consider 3 cells and 6 particles. Particles 2 and 4 are in cell 1, cell 2 is empty and the rest of the particles are in cell 3. The two lists could look like this: $A[1] = 2,\ A[2] = -1,\ A[3] = 5$ and $B[1] = -1,\ B[2] = 4,\ B[3] = 1,\ B[4] = -1,\ B[5] = 6,\ B[6] = 3$.

4. Perform a velocity rescaling after every 10 MD steps in order to keep the temperature constant at T=150K.

5. Run your MD simulation for densities $\rho = 0.0007/\mathrm{nm}^2$ and $\rho = 0.001/\mathrm{nm}^2$ and display the evolving system in an animation.

6. Measure the velocity distribution and the radial pair-correlation function $g(r)$ averaged over time. Interpret your result for $g(r)$ in comparison with the animation. Reminder: The radial pair-correlation function for a 2D system is defined as

$$g(r) = \frac{1}{2\pi} \int g(\vec{r}) d\theta, \tag{2}$$

   where

$$g(\vec{r}) = \frac{1}{\rho(N-1)} \left\langle \sum_{i \neq j} \delta(\vec{r} - \vec{r}_{ij}) \right\rangle. \tag{3}$$

   Use the discretized formulation of $g(r_k)$ for $r_k = k \cdot \Delta r$ which needs discrete histograms of the particle separations: $h(r_k)$ is the number of particle *pairs* with $r_{ij} \in [(k - \frac{1}{2})\Delta r, (k + \frac{1}{2})\Delta r]$. We then have

$$g(r_k) = \frac{2}{\rho(N-1)} \frac{\langle h(r_k) \rangle}{2\pi r_k \Delta r}. \tag{4}$$