CH 420: Understanding Advanced Molecular Simulation
Block 2

Qianjun Xu

Question 1: Calculation of π

Part (a): How can π be calculated from the fraction of points that fall in the circle? Complete the small Monte Carlo program to calculate π using this method.

We know that the area of a square of length l is l^2 and the area of a circle of diameter d is $\frac{\pi d^2}{4}$. Thus the probability of a randomly dropped point fall in the circle is

$$P = \frac{\pi d^2}{4l^2} = \frac{\pi}{4ratio^2}$$

By calculate the ratio of the number of points fall in the circle and the number of the whole points, we can calculate the π and here is the program (Figure 1).

Part (b): How would you expect that the accuracy of the result depends on the ratio l/d and the number of generated coordinates?

Part (c): Derive a formula to calculate the relative standard deviation of the estimate of π.

This algorithm can be described by a binomial distribution $B(N, P)$, whose mean is NP and the variance is $NP(1-P)$. If we denote the experiment result as b_i, what we are interested in are

$$\bar{b} = \frac{1}{N}\sum_{i=1}^{N} b_i = P$$

$$\sigma^2(\bar{b}) = \frac{1}{N^2}\sum_{i=1}^{N} b_i^2 - \bar{b}^2 = \frac{P(1-P)}{N}$$

And thus the standard deviation of the estimate of π is $\sqrt{\sigma^2(\bar{b})} = \sqrt{\frac{P(1-P)}{N}}$. We here use the relative standard deviation to study the accuracy of the result.

$$\frac{\sigma}{\bar{b}} = \sqrt{\frac{1-P}{NP}} = \sqrt{\frac{4ratio^2}{N\pi} - \frac{1}{N}}$$

From this we can see that the accuracy of the result depends on the ratio l/d and the number of generated coordinates. The smaller the ratio is (but it must be at least 1), and the larger the number of generated coordinates is, the more accurate the result is.

Part (d): Is it a good idea to calculate many decimals of π using this method?

This method can provide us with a approximation of the π, but is not a good way to calculate the π accurately, because its value converge to the exact value of π in probability. That is to say, no matter how many points we drop, it is still possible that there is a large deviation from the exact value.

```
57      // Start Modifications
58        x = RandomNumber();
59        y = RandomNumber();
60        if (SQR(x * ratio) + SQR(y * ratio) <= 1.0)
61        {
62          NumberOfHits++;
63        }
64        NumberOfTrials++;
65      // End Modifications
```

Figure 1: The program of calculating π.

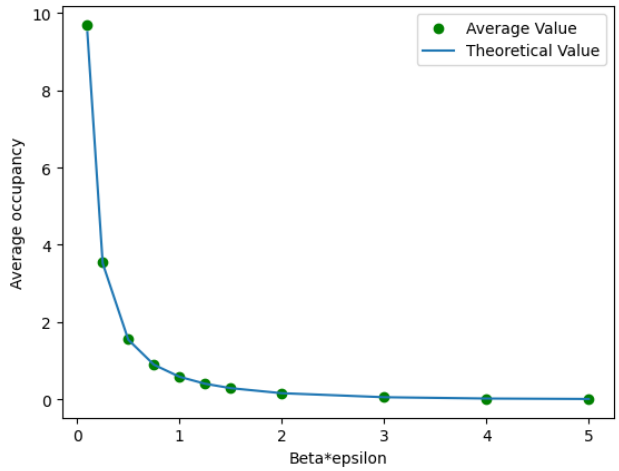

Figure 2: $\langle n_j \rangle$ as a function of $\beta\epsilon$.

Question 2: The Photon Gas

Part (a): How can this scheme obey detailed balance when $n_j = 0$?

To obey detaied balance when $n_j = 0$, if the trail move is to decrease n_j by 1, we cancel this move. Thus the probability for 0 to move to -1 and move from -1 are both 0.

Part (b): Is the algorithm still correct when trial moves are performed that change n_j with a random integer from the interval [-5, 5]? What happens when only trial moves are performed that change n_j with either -3 or +3?

The algorithm is correct when trial moves are performed that change n_j with a random integer from the interval [-5, 5]. It however fails when only trial moves are performed that change n_j with either -3 or +3, beacuse in this case, the prior probability $\alpha[0 \to 1] \neq \alpha[1 \to 0] = 0$.

Part (c): Assume that $N = 1$ and $\epsilon_j = \epsilon$. Use the small Monte Carlo program and the "run_auto" script to calculate $\langle n_j \rangle$ as a function of $\beta\epsilon$. Compare your result with the analytical solution.

When we use the standard method, the result agrees with the analytical solution. (Figure 2).

```
69      // accept or reject
70      if(RandomNumber()<exp(-Beta*(New-Old)))
71      {
72        if((i>NumberOfInitializationSteps) && (Old != New))
73        {
74          Sum+=New;
75          Count+=1.0;
76        }
77        Old=New;
78      }
79      if (i>NumberOfInitializationSteps)
80      {
81
82        if (Old>maxold) maxold=Old;
83        Distribution[Old]++;
84      }
85      // calculate average occupancy result
```

Figure 3: The modified program of producing wrong result.

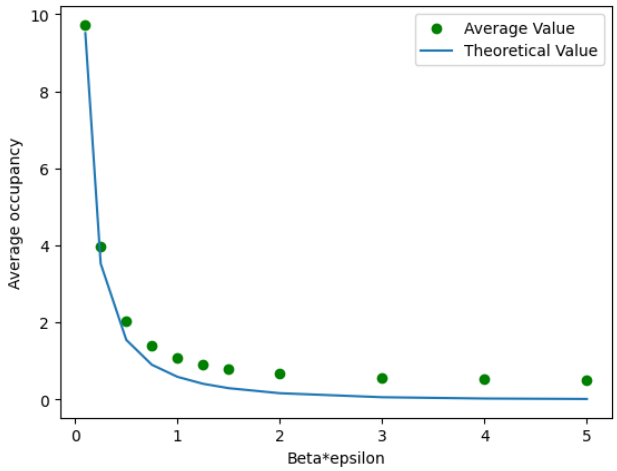

Figure 4: Wrong result of the modified program.

Part (d): Modify the program in such a way that the averages are updated only after an accepted trial move. Why does this lead to erroneous results? At which values of β does this error become more pronounced?

The modified program is shown in Figure 3. The erroneous results is more pronounced at larger values of β, that is to say, at lower temperature (Figure 4).

Part (e): Modify the program in such a way that the distribution of n_j is calculated as well. Compare this distribution with the analytical distribution

Figure 5 is the modified program. The following plot (Figure 6) is at $\beta = 0.10$. The simulation result agrees well with the theory.

```
69        // accept or reject
70        if(RandomNumber()<exp(-Beta*(New-Old)))
71        {
72          Old=New;
73        }
74        // calculate average occupancy result
75        if(i>NumberOfInitializationSteps)
76        {
77          Sum+=Old;
78          Count+=1.0;
79          if (Old>maxold) maxold=Old;
80          Distribution[Old]++;
81        }
82      }
83    }
```

Figure 5: Recording the distribution of n_j.

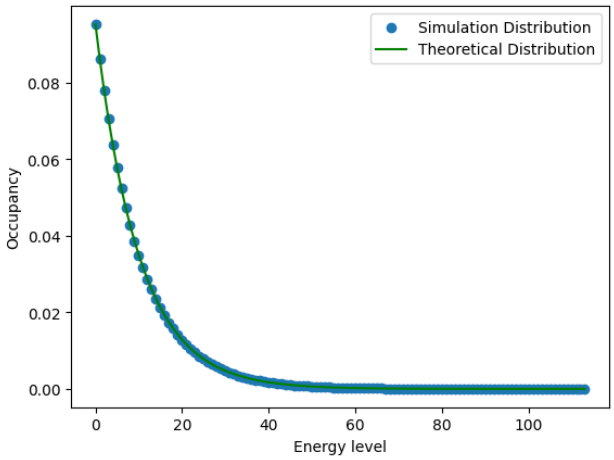

Figure 6: The distribution of n_j at $\beta = 0.10$.

```
36        // start modification virial
37        Virij+=-4.0*Epsilon*(-12.0*SQR(r6i)+6.0*r6i);
38        // end modification virial
```

Figure 7: The modification for calculating the pressure.

Table 1: Pressure versus density at $T = 2$.

Density	Pressure
0.1	0.178
0.2	0.331
0.3	0.485
0.4	0.694
0.5	1.072
0.6	1.760
0.7	2.982
0.8	5.398
0.9	9.148
1.0	15.12
1.2	39.84
1.4	95.92

Question 3: MC of a Lennard-Jones System

Part (a): In the present code, the pressure of the system is not calculated. Modify the code in such a way that the average pressure can be calculated. You will only have to make some modifications in the subroutine *energy.c* to calculate the virial. Using this quantity, the pressure will then be calculated automatically.

The modification is shown in Figure 7.

Part (b): Perform a simulation at $T = 2.0$ and various densities. Up to which density does the ideal gas law hold?

Below is the table (Table 1) and figure (Figure 8) for the pressure versus density at $T = 2$. It can be seen that the ideal gas law holds at $\rho \in [0.1,\ 0.4]$.

Part (c): The program produces a sequence of snapshots of the state of the system. Try to visualise these snapshots using the program *vmd*.

Figure 9 is the snapshot of the state of the system.

Part (d): Derive a formula for the dimensionless heat capacity. Modify the program (only in mc_nvt.c) in such a way that C_V is calculated.

Figure 10 is the calculation of the heat capacity.
The heat capacity $C_V = \frac{\langle U^2\rangle - \langle U\rangle^2}{Nk_BT^2}$ has a dimension of J/K. Thus, to make it a dimensionless, we can simply divide it by k_B:

$$C_V^* = \frac{\langle U^2\rangle - \langle U\rangle^2}{Nk_B^2T^2} = \frac{\langle U^2\rangle - \langle U\rangle^2}{N}\beta^2$$

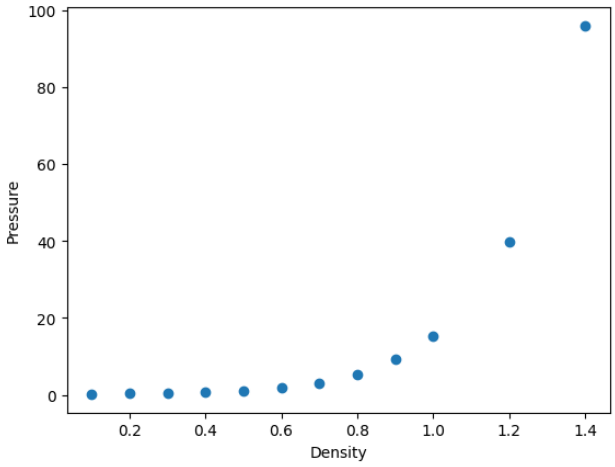

Figure 8: The distribution of n_j at $\beta = 0.10$.

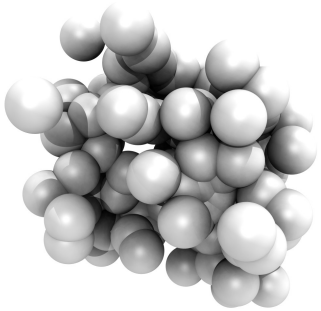

Figure 9: The snapshot of the state of the system.

```
88          // start modification question 4
89          // Heat capacity. You can use variables EnergySum, EnergyCount,
90          // and EnergySquaredSum.
91          EnergySquaredSum+=SQR(RunningEnergy);
92          EnergySum+=RunningEnergy;
93          EnergyCount+=1.0;
94
95          // end modification
```

Figure 10: The calculation of the C_v.

```
  VECTOR NewPositions[NumberOfParticles];
  VECTOR TEMP[NumberOfParticles];
  for(int i=0;i<NumberOfParticles;i++)
  {
     TEMP[i].x=Positions[i].x;
     TEMP[i].y=Positions[i].y;
     TEMP[i].z=Positions[i].z;
  }

  EnergySystem();
  EnergyOld=TotalEnergy;
  VirialOld=TotalVirial;
  for (int j = 0; j < NumberOfParticles; j++)
  {
    NewPositions[j].x = Positions[j].x+(RandomNumber()-0.5)*MaximumDisplacement;
    NewPositions[j].y = Positions[j].y+(RandomNumber()-0.5)*MaximumDisplacement;
    NewPositions[j].z = Positions[j].z+(RandomNumber()-0.5)*MaximumDisplacement;
  }
  for(int i=0;i<NumberOfParticles;i++)
  {
     Positions[i].x=NewPositions[i].x;
     Positions[i].y=NewPositions[i].y;
     Positions[i].z=NewPositions[i].z;
  }
  EnergySystem();
  EnergyNew=TotalEnergy;
  VirialNew=TotalVirial;
  if(RandomNumber()<exp(-Beta*(EnergyNew-EnergyOld)))
  {
    // accept
    NumberOfAcceptedMoves++;
    RunningEnergy+=(EnergyNew-EnergyOld);
    RunningVirial+=(VirialNew-VirialOld);
  }
    // put particle in simulation box
/*
    if(NewPosition.x<0.0)
      NewPosition.x+=Box;
    else if(NewPosition.x>=Box)
      NewPosition.x-=Box;

    if(NewPosition.y<0.0)
      NewPosition.y+=Box;
    else if(NewPosition.y>=Box)
      NewPosition.y-=Box;

    if(NewPosition.z<0.0)
      NewPosition.z+=Box;
    else if(NewPosition.z>=Box)
      NewPosition.z-=Box;
*/
  else
  {
   // update new position
   for(int i=0;i<NumberOfParticles;i++)
   {
      Positions[i].x=TEMP[i].x;
      Positions[i].y=TEMP[i].y;
      Positions[i].z=TEMP[i].z;
   }

  }
}
```

Figure 11: The algorithm of moving the whole system.

Part (e): Instead of performing a trial move in which only one particle is displaced, one can do a trial move in which all particles are displaced. Compare the maximum displacements of these moves when 50% of all displacements are accepted.

Figure 11 is the algorithm of moving the whole system. The maximal displacement of moving a single particle is 0.20. For moving all particles, the maximal displacement is 0.05. ($T = 1.0,\ \rho = 0.1$) Because the maximal displacement of moving a single particle is larger, the simulation is more efficient.

Part (f): Instead of using a uniformly distributed displacement, one can also use a Gaussian displacement. Does this increase the efficiency of the simulation?

A larger displacement is the core of sampling the system. The Gaussian distribution gives more smaller displacements while reduces the possibility of lager displacements. Thus the simulation might be less efficient.

```
35  if(Step<NumberOfInitializationSteps)
36
37    // Scale=sqrt(Temperature*(3.0*NumberOfParticles-3.0)/(2.0*UKinetic));
38    Scale=sqrt(Temperature*(3.0*NumberOfParticles-4.0)/(2.0*UKinetic));
39  else
40    Scale=1.0;
41
```

(a) One error in *integrate.c* in the calculation of the degree of freedom.

```
186  if(Step<NumberOfInitializationSteps)
187
188    // Scale=sqrt(Temperature*(3.0*NumberOfParticles-3.0)/(2.
       0*UKinetic));
189    Scale=sqrt(Temperature*(3.0*NumberOfParticles-4.0)/(2.
       0*UKinetic));
190  else
191    Scale=1.0;
192
```

(b) The other error in *integrate.c* also in the calculation of the degree of freedom.

```
46        UPotential+=4.0*r6i*(r6i-1.0)-Ecut;
47        // Ff=48.0*r2i*(r6i-0.5);
48        Ff=48.0*r6i*(r6i-0.5);
49        Pressure+=Ff;
50        Ff=Ff*r2i;
```

(c) One error in *force.c* in the calculation of the force.

Figure 12: The three errors.

Question 4: MD of a Lennard-Jones System

Part (a): Find the three errors in the code.

Two errors are found in *integrate.c* and one is found in *force.c* (Figure 12).

Part (b): How is the temperature controlled in this program? What does it mean to "control the temperature": after all, in a simulation at constant NVE it is the total energy of the system that should be constant, whilst the temperature fluctuates.

During the initialization phase, the temperature is controlled by multiply the calculated velocity with a factor

$$c = \sqrt{\frac{T_{expectation}}{\frac{\sum_{i=1}^{N} m v_{\alpha,\,i}^2(t)}{N_f}}} = \sqrt{\frac{T_{expectation}(3N-4)}{\sum_{i=1}^{N} m v_{\alpha,\,i}^2(t)}}$$

This scaling factor is defined based on the relationship

$$\left\langle v_\alpha^2 \right\rangle = k_B T/m$$

which describes the relationship between the kinetic energy and the instantaneous temperature. When performing the simulation, the instantanuous temperature sometimes deviates from the temperature we expect.

$$T/T_{expectation} = \frac{m\left\langle v_\alpha^2 \right\rangle}{k_B T_{expectation}}$$

To tune the ratio to be 1, a scaling can be performed on the velocity

$$\frac{m\left\langle (cv_\alpha)^2 \right\rangle}{k_B T_{expectation}} = 1$$

which leads to

$$c = \sqrt{\frac{k_B T_{expectation}}{m\left\langle v_\alpha^2 \right\rangle}}$$

and the expectation of the squared velocity is defined by

$$\left\langle v_\alpha^2 \right\rangle = \frac{\sum_{i=1}^{N} v_{\alpha,\,i}^2(t)}{N_f} = \frac{v_{\alpha,\,i}^2(t)}{3N-4}$$

This is how the first equation comes. Through this we can see the control over the temperature is actually a control of the kinetic energy. This helps us relax our initial structure, which might be unreasonable at first and leads to a blow up to the simulation system, and a sufficient equibilirium can help avoid this. After passing the first initialization steps, we no longer need this scaling anymore.

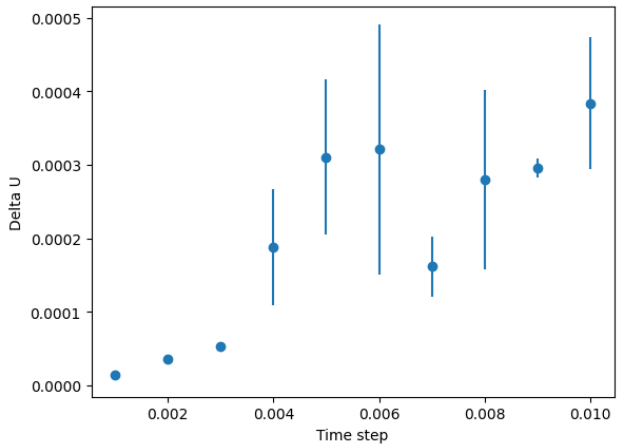

Figure 13: The drift of the total energy versus the length of time step.

Part (c): How does the time step for a given energy drift change with the temperature and density?

From the Figure 13, it can be seen that the drift of total energy increases with the length of the time step.
Additionally, for a given energy drift, the higher the temperature and density are, the smaller the time step is.

Part (d): As you might have noticed in the code, the implementation of the periodic boundary conditions is often done as ibox is used instead of 1/box. Why would one do this?

Because it is more time efficient to calculate the result of $1/box$ and store it, instead of calculate it every time. However, this performance improvement might be very small. For me, I would prefer to use $1/box$ for the readablity of the code.

Part (e): Modify the program in such a way that the self-diffusion coefficient can be calculated using both methods. Only modifications in subroutine sample_diff.c are needed. It is not a good idea (although not wrong) to take every time step as a new time origin for the calculation of the VACF and of the mean-squared displacement. Explain. What is the unit of D in SI units? How can one transform D into dimensionless units?

The implementation of the periodic boundry conditions (PBC) is to enable one to simulate a infinitly large system with countable atoms. The update of the coordinates of atoms out of the PBC box is necessary because the interactions between the atoms need to be calculated correctly.
To take every time step as a new time origin does increase the available data for calculate the VACF and the mean-squared displacement, while it is not efficient in terms of sampling. Whatsmore, when we have a very long trajectory, the calculation of the VACF and the mean-squared displacement is very expensive.
The modification of the program is shown in Figure 14.
The unit of D in SI units is m^2/s. According to [Naghizadeh, 1962](https://doi.org/10.1063/1.1732357), if atoms interact pairwise with a two parameter potention, the reduced diffusion coefficient can be defined as

$$\tilde{D} = (m/\epsilon\sigma^2)^{1/2} D$$

where m is the mass of the particle, ϵ is the depth of the potential well and σ is a length related to the size of the particle.

```
78      // start modification
79
80      for(i=0;i<MIN(t0Counter, MAXT0);i++)
81      {
82        CorrelTime=time-t0time[i];
83        if (CorrelTime < MAXT)
84        {
85
86          SampleCounter[CorrelTime]+=1;
87          for (j=0;j<NumberOfParticles;j++)
88          {
89
90            R2[CorrelTime]+=SQR(PositionsNONPDB[j].x-Rx0[j][i])+SQR(PositionsNONPDB[j].y-Ry0[j][i])+SQR
              (PositionsNONPDB[j].z-Rz0[j][i]);
91            Vacf[CorrelTime]+=Vxt0[j][i]*Velocities[j].x+Vyt0[j][i]*Velocities[j].y+Vzt0[j][i]*Velocities[j].z;
92          }
93        }
94      }
95      // end modification
```

Figure 14: The calculation of the self-diffusion coefficient.

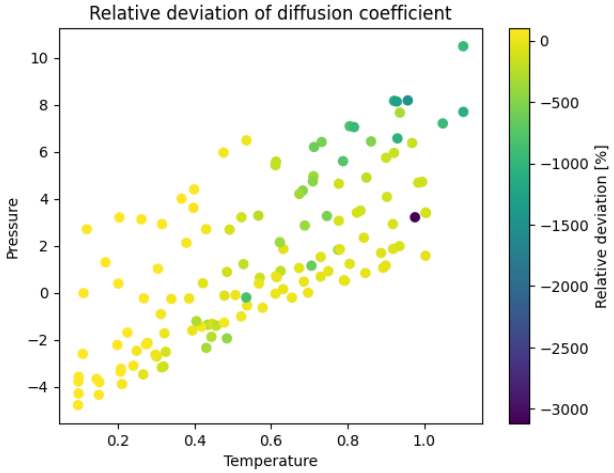

Figure 15: The relative deviation of diffusion coefficient.

Part (f): For Lennard-Jones liquids, Naghizadeh and Rice report the following equation for the self-diffusion coefficient. Try to confirm this equation with simulations.

Simulations are done to see if the equation can be got from simulation. Here I calculate the relative deviation of the self-diffusion coefficient.

$$\frac{\tilde{D}_{simulation} - \tilde{D}_{equation}}{\tilde{D}_{simulation}}$$

and plot the result in Figure 15. It can be seen that the equation can only be confirmed in low temperature and low presure area.

Part (g): Make a plot of the energy drift ΔU for the following integration algorithms.

In the three algorithms, the Euler method is the worst, which is not stable even at a very short time step of 0.0001. Verlet and Velocity-Verlet both perform well, and Velocity-Verlet is a little bit better than Verlet (Figure 16).

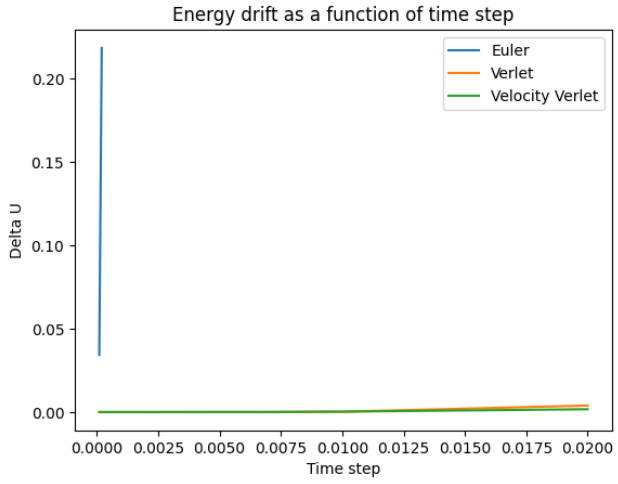

Figure 16: The energy drift as a function of time step of Euler, Verlet and Velocity-Verlet.