PHYS 516: Methods of Computational Physics
ASSIGNMENT 4- Molecular Dynamics Simulation

Anup V Kanale
February 22, 2017

The purpose of this assignment is to get familiar with basic concepts in ordinary differential equations using a simple molecular dynamics (MD) program, *md.c*, as an example.

1 Liouville Theorem

Here, we show that for a particle in 1-D space, the velocity Verlet algorithm exactly preserves the phase space volume for arbitrary time discretization unit, Δ.

Let x be the position of the particle, $p = mv$ the momentum where, m and v are the mass and velocity of the particle. In the dimensionless form, $p = v$. Let the coordinate and momentum of the particle at time t be (x, p), and those at time $t + \Delta$ be (x', p')

$$x' = x + p\Delta + \frac{1}{2}a(x) + \Delta^2 \tag{1}$$

$$p' = p + \frac{a(x) + a(x')}{2}\Delta \tag{2}$$

To show this, it is enough to show that the Jacobian of the transformation $(x'(x, p), p'(x, p))$ is 1, i.e. $J = \left|\frac{\partial(x', p')}{\partial(x, p)}\right| = 1$. The derivatives are calculated from Eq (1) and (2) as follows:

$$\frac{\partial x'}{\partial x} = 1$$
$$\frac{\partial x'}{\partial p} = \Delta$$
$$\frac{\partial p'}{\partial x} = 0$$
$$\frac{\partial p'}{\partial p} = 1$$

Therefore, the Jacobian is given by

$$J = \begin{vmatrix} \frac{\partial x'}{\partial x} & \frac{\partial p'}{\partial x} \\ \frac{\partial x'}{\partial p} & \frac{\partial p'}{\partial p} \end{vmatrix} \tag{3}$$

$$= \begin{vmatrix} 1 & 0 \\ \Delta & 1 \end{vmatrix} \tag{4}$$

$$= 0 \tag{5}$$

2 Comparison of Euler and Velocity-Verlet Algorithms

Total energy of the system should be conserved. But due to numerical errors, energy conservation depends on the time-integration algorithm used. In this section, we illustrate this by comparing the Euler and Velocity-Verlet algorithms.

```
Euler Algorithm
Given a configuration $(\boldsymbol{r}_i(t), \boldsymbol{v}_i(t) | i = 1 \text{ to } N_{atom})$
1. Compute the acceleration: $\boldsymbol{a}_i(t)$
2. Update the positions: $\boldsymbol{r}_i(t+\Delta) \leftarrow \boldsymbol{r}_i(t) + \boldsymbol{v}_i(t)\Delta + \frac{1}{2}\boldsymbol{a}_i(t)\Delta^2$
3. Update the velocities: $\boldsymbol{v}_i(t+\Delta) \leftarrow \boldsymbol{v}_i(t) + \boldsymbol{a}_i(t)\Delta$
```

```
Velocity-Verlet Algorithm
Given a configuration $(\boldsymbol{r}_i(t), \boldsymbol{v}_i(t) | i = 1 \text{ to } N_{atom})$
1. Compute the acceleration: $\boldsymbol{a}_i(t)$
2. $\boldsymbol{v}_i(t+\frac{\Delta}{2}) \leftarrow \boldsymbol{v}_i(t) + \boldsymbol{a}_i(t)\frac{\Delta}{2}$
3. $\boldsymbol{r}_i(t+\Delta) \leftarrow \boldsymbol{r}_i(t) + \boldsymbol{v}_i(t+\frac{\Delta}{2})\Delta$
4. Compute the updated acceleration: $\boldsymbol{a}_i(t+\Delta)$
5. $\boldsymbol{v}_i(t+\Delta) \leftarrow \boldsymbol{v}_i(t+\frac{\Delta}{2}) + \boldsymbol{a}_i(t+\Delta)\frac{\Delta}{2}$
```

From the plot below, we can clearly see that Euler scheme blows up whereas Velocity Verlet scheme is stable and conserves total energy of the system.

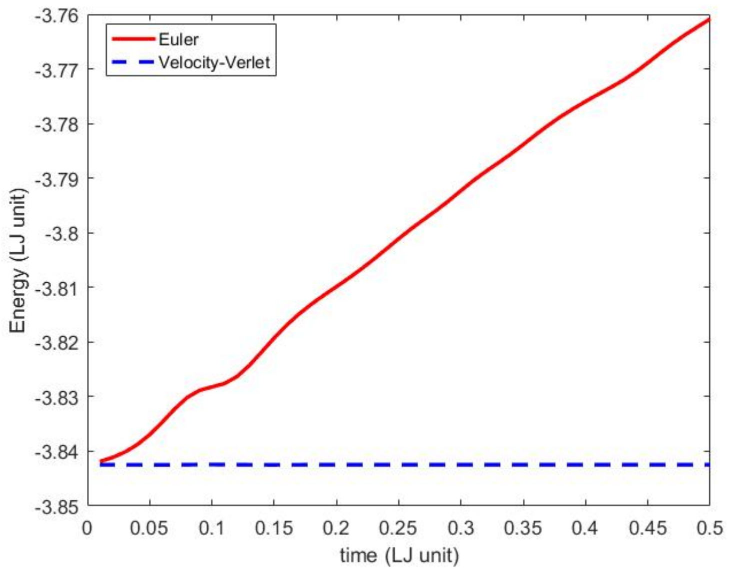
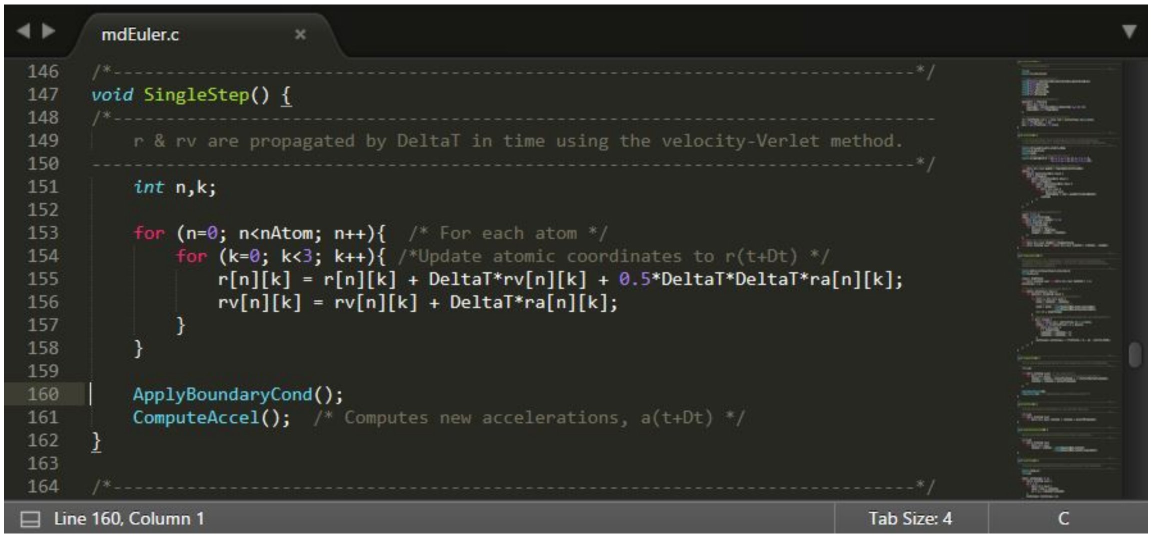


Figure 1: Comparison of total energy conservation for Euler and Velocity verlet algorithms

The code downloaded from the course website was used, which implements the Velocity Verlet algorithm by default. Euler scheme was implemented by modifying the `SingleStep()` section in the code. The time step was changed to 0.001, but for all the other parameters, the default values were used from the input file `md.in`.

Since the code is rather long and the modifications were minor, just the parts with the numerical schemes are shown below.

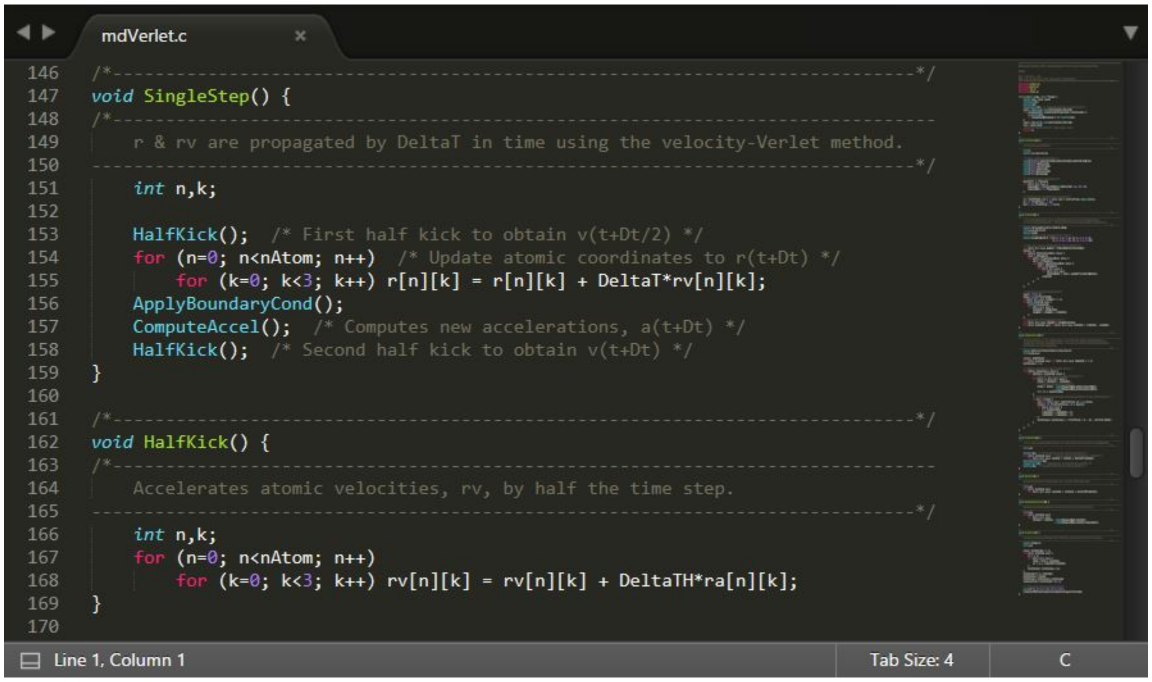


mdEuler.c

```c
/*--------------------------------------------------------------------------*/
void SingleStep() {
/*--------------------------------------------------------------------------
    r & rv are propagated by DeltaT in time using the velocity-Verlet method.
--------------------------------------------------------------------------*/
    int n,k;

    for (n=0; n<nAtom; n++){  /* For each atom */
        for (k=0; k<3; k++){ /*Update atomic coordinates to r(t+Dt) */
            r[n][k] = r[n][k] + DeltaT*rv[n][k] + 0.5*DeltaT*DeltaT*ra[n][k];
            rv[n][k] = rv[n][k] + DeltaT*ra[n][k];
        }
    }

    ApplyBoundaryCond();
    ComputeAccel();  /* Computes new accelerations, a(t+Dt) */
}

/*--------------------------------------------------------------------------*/
```

Line 160, Column 1   Tab Size: 4   C

Figure 2: Euler scheme implementation



mdVerlet.c

```c
/*--------------------------------------------------------------------------*/
void SingleStep() {
/*--------------------------------------------------------------------------
    r & rv are propagated by DeltaT in time using the velocity-Verlet method.
--------------------------------------------------------------------------*/
    int n,k;

    HalfKick();  /* First half kick to obtain v(t+Dt/2) */
    for (n=0; n<nAtom; n++)  /* Update atomic coordinates to r(t+Dt) */
        for (k=0; k<3; k++) r[n][k] = r[n][k] + DeltaT*rv[n][k];
    ApplyBoundaryCond();
    ComputeAccel();  /* Computes new accelerations, a(t+Dt) */
    HalfKick();  /* Second half kick to obtain v(t+Dt) */
}

/*--------------------------------------------------------------------------*/
void HalfKick() {
/*--------------------------------------------------------------------------
    Accelerates atomic velocities, rv, by half the time step.
--------------------------------------------------------------------------*/
    int n,k;
    for (n=0; n<nAtom; n++)
        for (k=0; k<3; k++) rv[n][k] = rv[n][k] + DeltaTH*ra[n][k];
}

```

Line 1, Column 1   Tab Size: 4   C

Figure 3: Euler scheme implementation

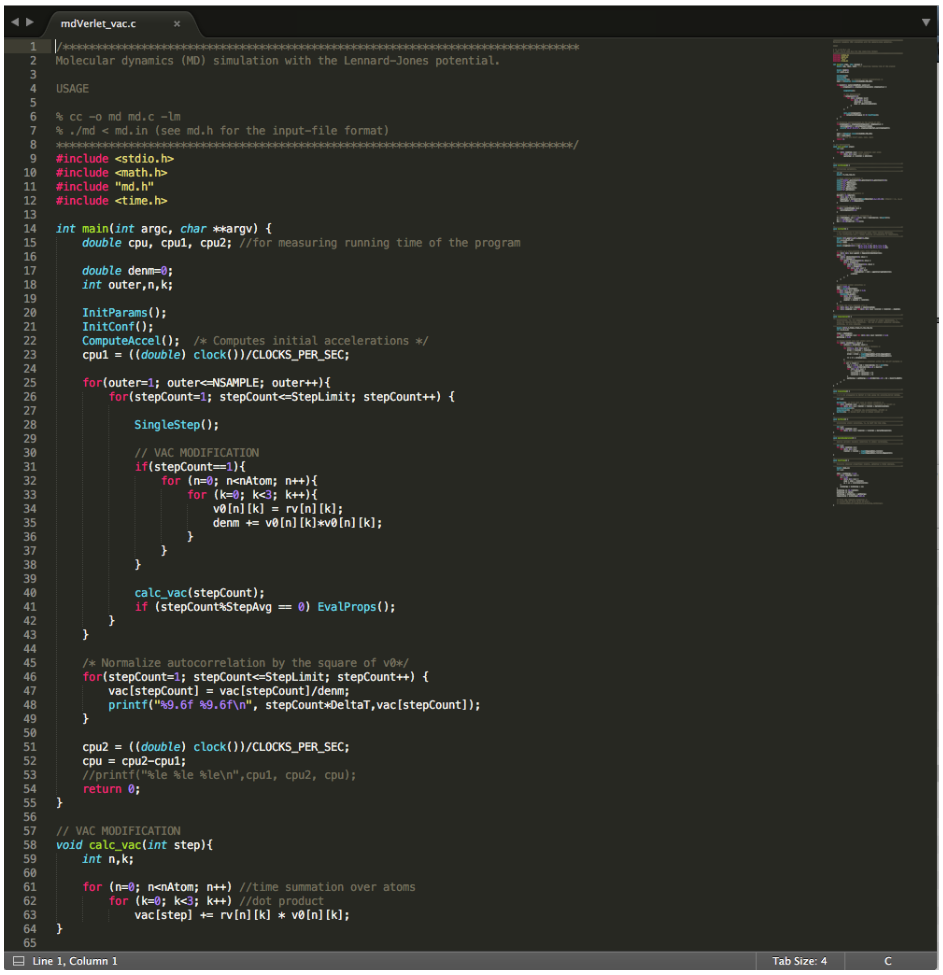
3 Velocity Autocorrelation Function

Here, we calculate the velocity autocorrelation function

$$Z(t) = \frac{\langle \vec{v}_i(t+t_0) \cdot \vec{v}_i(t_0) \rangle}{\langle \vec{v}_i(t_0) \cdot \vec{v}_i(t_0) \rangle} = \frac{\sum_{t_0} \sum_{i=1}^{N} \vec{v}_i(t+t_0) \cdot \vec{v}_i(t_0)}{\sum_{t_0} \sum_{i=1}^{N} \vec{v}_i(t_0) \cdot \vec{v}_i(t_0)} \tag{6}$$

where $\vec{v}_i(t)$ is the velocity of the i-th atom at time t. The bracket denotes averages over atoms, i, and the time origin, t_0.

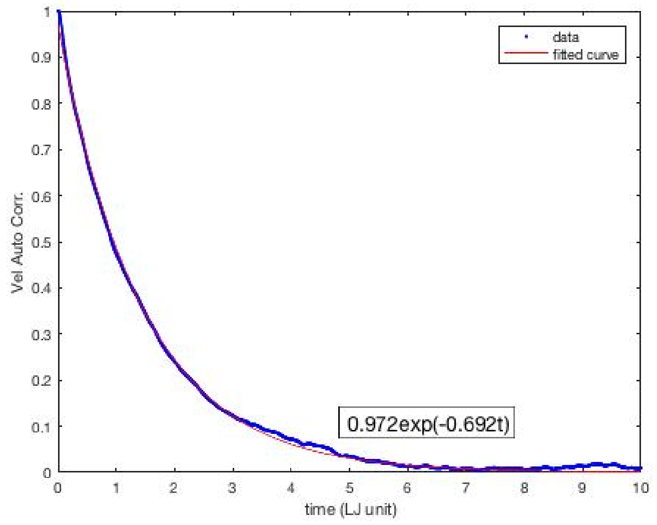
The `md.c` program from the course website was modified to calculate autocorrelation function. In summary, the `calc_vac` function was added to calculate the numerator term of Eqn (6), and some lines were added in the `main` function– variable `denm` to accumulate the sums in the denominator term and an outer loop to run the simulation `NSAMPLE` times and take averages. The required variable were added in the `md.h` header file, but are not shown here as these additions were not significant. The modified part of the `md.c` is shown below.



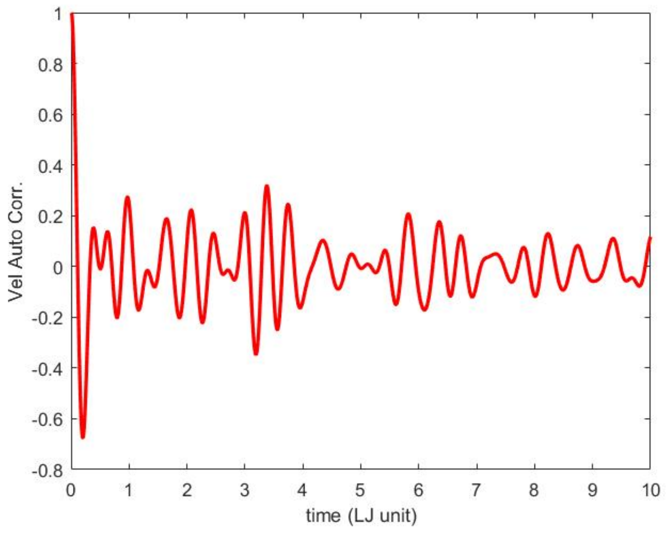
```c
/********************************************************************************
Molecular dynamics (MD) simulation with the Lennard-Jones potential.

USAGE

% cc -o md md.c -lm
% ./md < md.in (see md.h for the input-file format)
********************************************************************************/
#include <stdio.h>
#include <math.h>
#include "md.h"
#include <time.h>

int main(int argc, char **argv) {
    double cpu, cpu1, cpu2; //for measuring running time of the program

    double denm=0;
    int outer,n,k;

    InitParams();
    InitConf();
    ComputeAccel();  /* Computes initial accelerations */
    cpu1 = ((double) clock())/CLOCKS_PER_SEC;

    for(outer=1; outer<=NSAMPLE; outer++){
        for(stepCount=1; stepCount<=StepLimit; stepCount++) {

            SingleStep();

            // VAC MODIFICATION
            if(stepCount==1){
                for (n=0; n<nAtom; n++){
                    for (k=0; k<3; k++){
                        v0[n][k] = rv[n][k];
                        denm += v0[n][k]*v0[n][k];
                    }
                }
            }

            calc_vac(stepCount);
            if (stepCount%StepAvg == 0) EvalProps();
        }
    }

    /* Normalize autocorrelation by the square of v0*/
    for(stepCount=1; stepCount<=StepLimit; stepCount++) {
        vac[stepCount] = vac[stepCount]/denm;
        printf("%9.6f %9.6f\n", stepCount*DeltaT,vac[stepCount]);
    }

    cpu2 = ((double) clock())/CLOCKS_PER_SEC;
    cpu = cpu2-cpu1;
    //printf("%le %le %le\n",cpu1, cpu2, cpu);
    return 0;
}

// VAC MODIFICATION
void calc_vac(int step){
    int n,k;

    for (n=0; n<nAtom; n++) //time summation over atoms
        for (k=0; k<3; k++) //dot product
            vac[step] += rv[n][k] * v0[n][k];
}
```

Figure 4: Modified Code block

The program was run using `InitUcell = 3,3,3, DeltaT = 0.005` and `StepAvg = 10,` for `StepLimit = 2000` steps to calculate VAC in the time range of `[0, tmax = DeltaT*StepLimit = 0.005*2000 = 10.0]` for both gas phase (`Density = 0.1, InitTemp = 1.0`) and solid phase (`Density = 1.0, InitTemp = 0.1`). As The results are shown below:



(a) Velocity autocorrelation vs time for gas phase



(b) Velocity autocorrelation vs. time for solid phase

As expected, vac for gas phase dies down from 1 to 0 exponentially while for the the solid phase it just fluctuates about zero.

4 Split Operator Formalism- Deriving the Velocity Verlet algorithm