Molecular Dynamics Simulation of Argon

The fundamental work on this problem was done by A. Rahman, Phys. Rev. **136**, A405 (1964). It was extended in many important ways by L. Verlet, Phys. Rev. **159**, 98 (1967), who introduced the Verlet algorithm and the use of a neighbor list to speed up the calculation.

Simple model of interacting Argon atoms

Consider N atoms of argon each with mass $m = 6.69 \times 10^{-26}\,\mathrm{kg}$. Argon is an *inert gas*: argons atoms behave approximately like hard spheres which attract one another with weak van der Waals forces. The forces between two argon atoms can be approximated quite well by a Lennard-Jones potential energy function:

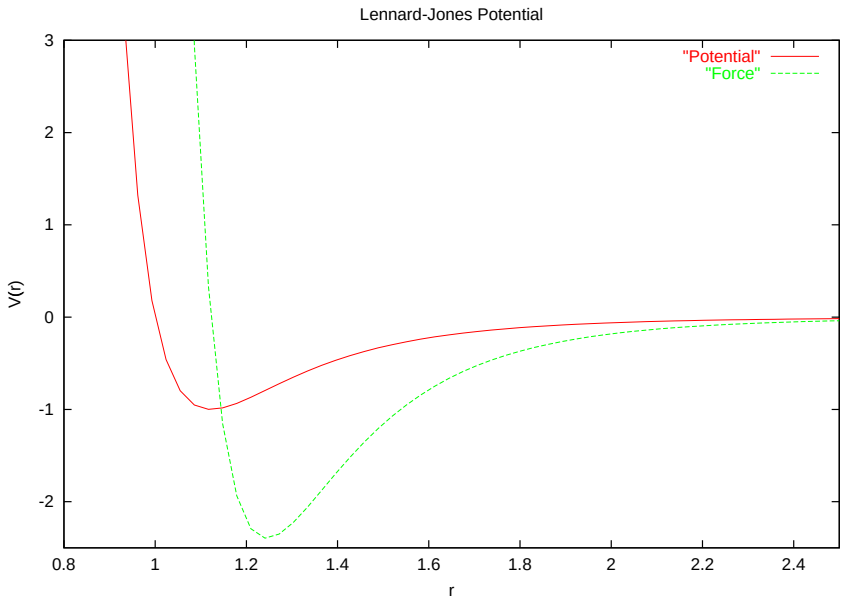
$$V(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] ,$$

where r is the distance between the centers of the two atoms, $\varepsilon = 1.65{\times}10^{-21}\,\mathrm{J}$ is the strength of the potential energy, and $\sigma = 3.4 \times 10^{-10}\,\mathrm{m}$ is the value of r at which the energy is zero. The $1/r^{12}$ term represents a repulsive *hard core* interaction between the argon atoms. The $1/r^{6}$ term represents an attractive dipole-dipole (van der Waals) interaction between the non-polar atoms. The potential has its minimum $V(2^{\frac{1}{6}}\sigma) = -\varepsilon$ at $r = 2^{\frac{1}{6}}\sigma$.

The shape of the potential and the strength of the Lennard-Jones force

$$F(r) = -\frac{dV(r)}{dr} = \frac{24\varepsilon}{\sigma} \left[2\left(\frac{\sigma}{r}\right)^{13} - \left(\frac{\sigma}{r}\right)^{7} \right] ,$$

are shown in the following figure:



We will choose units of mass, length and energy so that $m = 1$, $\sigma = 1$, and $\varepsilon = 1$. The unit of time in this system is given by

$$\tau = \sqrt{\frac{m\sigma^2}{\varepsilon}} = 2.17 \times 10^{-12}\,\mathrm{s}\,,$$

which shows that the natural time scale for the dynamics of this system is a few picoseconds!

A simple MD program

First include some standard headers

md.cpp

```
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
```

Define data structures to describe the kinematics of the system

md.cpp


```
const int N = 64;          // number of particles
double r[N][3];            // positions
double v[N][3];            // velocities
double a[N][3];            // accelerations
```

We next need to set the initial positions and velocities of the particles. This is actually a complicated problem! Because the system can be simulated only for a few nanoseconds, the starting configuration must be very close to equilibrium to get good results. For a dense system, the atoms are usually placed at the vertices of a face-centered cubic lattice, which is tends to minimize the potential energy. The atoms are also given random velocities to approximate the desired temerature.

In this preliminary program we will put the system in a cubical volume of side L and place the particles at the vertices of a simple cubic lattice:

md.cpp

```
double L = 10;             // linear size of cubical volume
double vMax = 0.1;         // maximum initial velocity component

void initialize() {

    // initialize positions
    int n = int(ceil(pow(N, 1.0/3)));  // number of atoms in each direction
```

```
    double a = L / n;                   // lattice spacing
    int p = 0;                          // particles placed so far
    for (int x = 0; x < n; x++)
        for (int y = 0; y < n; y++)
            for (int z = 0; z < n; z++) {
                    if (p < N) {
                        r[p][0] = (x + 0.5) * a;
                        r[p][1] = (y + 0.5) * a;
                        r[p][2] = (z + 0.5) * a;
                    }
                    ++p;
            }

    // initialize velocities
    for (int p = 0; p < N; p++)
        for (int i = 0; i < 3; i++)
            v[p][i] = vMax * (2 * rand() / double(RAND_MAX) - 1);
}
```


Newton's Equations of Motion

The vector forces between atoms with positions $\mathbf{r}_i$ and $\mathbf{r}_j$ are given by

$$\mathbf{F}_{\text{on } i \text{ by } j} = -\mathbf{F}_{\text{on } j \text{ by } i} = 24(\mathbf{r}_i - \mathbf{r}_j)\left[2\left(\frac{1}{r}\right)^{-14} - \left(\frac{1}{r}\right)^{-8}\right] ,$$

where $r = |\mathbf{r}_i - \mathbf{r}_j|$.

The net force on atom i due to all of the other $N-1$ atoms is given by

$$\mathbf{F}_i = \sum_{\substack{j=1 \\ j\neq i}}^{N} \mathbf{F}_{\text{on } i \text{ by } j} .$$

The equation of motion for atom i is

$$\mathbf{a}_i(t) \equiv \frac{d\mathbf{v}_i(t)}{dt} = \frac{d^2\mathbf{r}_i(t)}{dt^2} = \frac{\mathbf{F}_i}{m} ,$$

where $\mathbf{v}_i$ and $\mathbf{a}_i$ are the velocity and acceleration of atom i. These $3N$ second order ordinary differential equations (ODE's) have a unique solution as function of time t if *initial conditions*, that is, the values of positions $\mathbf{r}_i(t_0)$ and velocities $\mathbf{v}_i(t_0)$ of the particles are specified at some initial time t_0. The equations can be integrated numerically by choosing a small time step h and a discrete approximation to the equations to advance the solution by one step at a time.

The following function computes the accelerations of the particles from their current positions:

md.cpp

```
void computeAccelerations() {

    for (int i = 0; i < N; i++)          // set all accelerations to zero
        for (int k = 0; k < 3; k++)
            a[i][k] = 0;

    for (int i = 0; i < N-1; i++)        // loop over all distinct pairs i,j
        for (int j = i+1; j < N; j++) {
            double rij[3];               // position of i relative to j
            double rSqd = 0;
            for (int k = 0; k < 3; k++) {
                rij[k] = r[i][k] - r[j][k];
                rSqd += rij[k] * rij[k];
            }
            double f = 24 * (2 * pow(rSqd, -7) - pow(rSqd, -4));
            for (int k = 0; k < 3; k++) {
                 a[i][k] += rij[k] * f;
                 a[j][k] -= rij[k] * f;
            }
        }
}
```


Velocity Verlet Integration Algorithm

There are many algorithms which can be used to solve ODE's. Verlet has developed several algorithms which are very widely used in MD simulations. One of them is the *velocity Verlet algorithm*

$$\mathbf{r}_i(t+dt) = \mathbf{r}_i(t) + \mathbf{v}_i(t)dt + \frac{1}{2}\mathbf{a}_i(t)dt^2$$
$$\mathbf{v}_i(t+dt) = \mathbf{v}_i(t) + \frac{1}{2}[\mathbf{a}_i(t+dt) + \mathbf{a}_i(t)]dt$$

It can be shown that the errors in this algorithm are of $\mathcal{O}(dt^4)$, and that it is very stable in MD applications and in particular conserves energy very well.

The following function advances the positions and velocities of the particles by one time step:

md.cpp

```
void velocityVerlet(double dt) {
    computeAccelerations();
    for (int i = 0; i < N; i++)
        for (int k = 0; k < 3; k++) {
            r[i][k] += v[i][k] * dt + 0.5 * a[i][k] * dt * dt;
            v[i][k] += 0.5 * a[i][k] * dt;
        }
    computeAccelerations();
```

```
    for (int i = 0; i < N; i++)
        for (int k = 0; k < 3; k++)
            v[i][k] += 0.5 * a[i][k] * dt;
}
```

The instantaneous temperature

This is a simulation in which the number of particles N and the volume L^3 of the system are fixed. Because the Lennard-Jones force is *conservative*, the total energy of the system is also constant.

If the system is in thermal equilibrium, then Boltzmann's *Equipartition Theorem* relates the absolute temperature T to the kinetic energy:

$$3(N-1) \times \frac{1}{2} k_{\mathrm{B}} T = \left\langle \frac{m}{2} \sum_{i=1}^{N} v_i^2 \right\rangle .$$

Here the angle brackets $\langle ... \rangle$ represent a thermal ensemble average. The factor $3(N-1)$ is the number of *internal* translational degrees of freedom which contribute to thermal motion: the motion of the center of mass of the system does not represent thermal energy!

md.cpp

```
double instantaneousTemperature() {
    double sum = 0;
```



```
    for (int i = 0; i < N; i++)
        for (int k = 0; k < 3; k++)
            sum += v[i][k] * v[i][k];
    return sum / (3 * (N - 1));
}
```

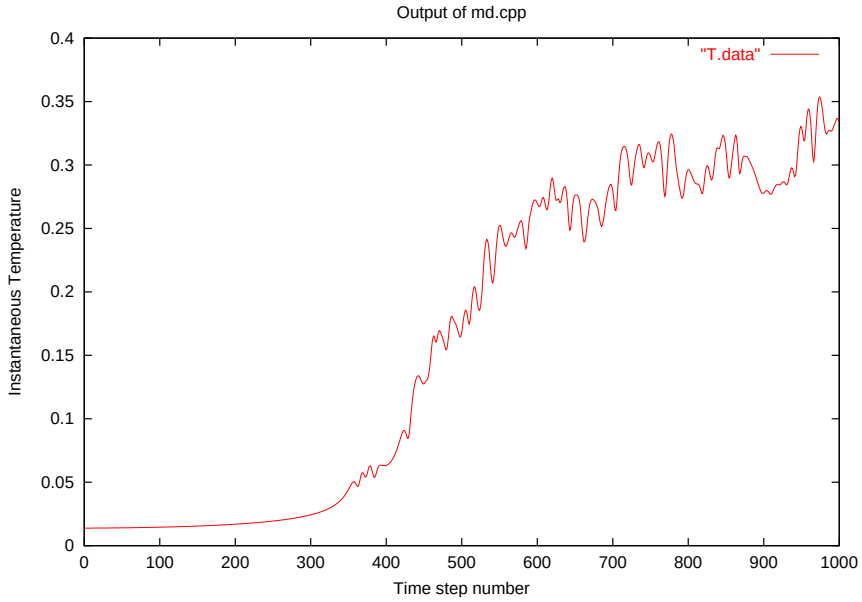
Finally, here is the main function which steers the simulation:

md.cpp

```
int main() {
    initialize();
    double dt = 0.01;
    ofstream file("T.data");
    for (int i = 0; i < 1000; i++) {
        velocityVerlet(dt);
        file << instantaneousTemperature() << '\n';
    }
    file.close();
}
```

Output of simple program md.cpp

The program output file was plotted using Gnuplot. The instantaneous temperature is approximately constant for around one or two time units, and then it starts increasing with fluctuations.



Improving the MD program

There are several ways in which this simple program needs to be improved:

- The volume is not really constant because the particles can move out of it! We need to impose suitable boundary conditions, for example periodic boundary conditions.
- The initial positions and velocities need to be chosen more carefully. We will place the particles on a face-centered cubic lattice, and use a Maxwell-Boltzmann distribution for the velocities.
- The system needs to be allowed to come to thermal equilibrium at the desired temperature.
- Thermal averages of various quantities need to be measured.

md2.cpp

```
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

// simulation parameters
```

```
int N = 64;                // number of particles
double rho = 1.0;          // density (number per unit volume)
double T = 1.0;            // temperature

// function declarations

void initialize();         // allocates memory, calls following 2 functions
void initPositions();      // places particles on an fcc lattice
void initVelocities();     // initial Maxwell-Boltzmann velocity distribution
void rescaleVelocities(); // adjust the instanteous temperature to T
double gasdev();           // Gaussian distributed random numbers
```

We will allocate particle arrays dynamically rather than statically

md2.cpp

```
double **r;                // positions
double **v;                // velocities
double **a;                // accelerations

void initialize() {
    r = new double* [N];
```



```
    v = new double* [N];
    a = new double* [N];
    for (int i = 0; i < N; i++) {
        r[i] = new double [3];
        v[i] = new double [3];
        a[i] = new double [3];
    }
    initPositions();
    initVelocities();
}
```

Position particles on a face-centered cubic lattice

The minimum energy configuration of this Lennard-Jones system is an fcc lattice. This has 4 lattice sites in each conventional cubic unit cell. If the number of atoms $N = 4M^3$, where $M = 1, 2, 3, ...$, then the atoms can fill a cubical volume. So MD simulations are usually done with $32 = 4{\times}2^3, 108 = 4{\times}3^3, 256, 500, 864, ...$ atoms.

md2.cpp

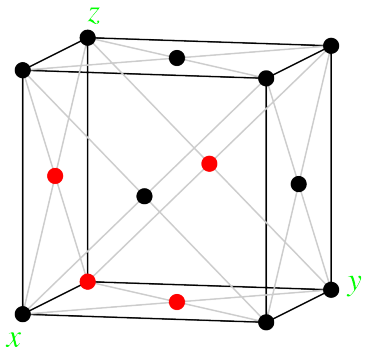
```
double L;                      // linear size of cubical volume

void initPositions() {
```

```
// compute side of cube from number of particles and number density
L = pow(N / rho, 1.0/3);

// find M large enough to fit N atoms on an fcc lattice
int M = 1;
while (4 * M * M * M < N)
    ++M;
double a = L / M;           // lattice constant of conventional cell
```

The figure shows a conventional cubical unit cell:



The 4 atoms shown in red provide a basis for the conventional cell. Their positions in units of a are given by

$$(0,\ 0,\ 0) \qquad (0.5,\ 0.5,\ 0) \qquad (0.5,\ 0,\ 0.5) \qquad (0,\ 0.5,\ 0.5)\ .$$

In the following code we shift the basis by $(0.5,\ 0.5,\ 0.5)$ so the all atoms are inside the volume and none are on its boundaries.

md2.cpp

```
// 4 atomic positions in fcc unit cell
double xCell[4] = {0.25, 0.75, 0.75, 0.25};
double yCell[4] = {0.25, 0.75, 0.25, 0.75};
double zCell[4] = {0.25, 0.25, 0.75, 0.75};
```

Next, the atoms are placed on the fcc lattice. If $N \neq 4M^3$ then some of the lattice sites are left unoccupied.

md2.cpp

```
    int n = 0;                  // atoms placed so far
    for (int x = 0; x < M; x++)
        for (int y = 0; y < M; y++)
            for (int z = 0; z < M; z++)
                for (int k = 0; k < 4; k++)
                    if (n < N) {
                        r[n][0] = (x + xCell[k]) * a;
                        r[n][1] = (y + yCell[k]) * a;
                        r[n][2] = (z + zCell[k]) * a;
                        ++n;
                    }
}
```


Draw initial velocities from a Maxwell-Boltzmann distribution

A more realistic initial velocity distribution is that of an ideal gas at temperature T:

$$P(\mathbf{v}) = \left(\frac{m}{2\pi k_{\mathrm{B}} T}\right)^{3/2} e^{-m\left(v_x^2+v_y^2+v_z^2\right)/(2k_{\mathrm{B}}T)} .$$

Note that each velocity component is Gaussian distributed with mean zero and width $\sim \sqrt{T}$.

The function gasdev from *Numerical Recipes* returns random numbers with a Gaussian probability distribution

$$P(x) = \frac{e^{-(x-x_0)^2/(2\sigma^2)}}{\sqrt{2\pi\sigma^2}} ,$$

with center $x_0 = 0$ and unit variance $\sigma^2 = 1$. This function uses the Box-Müller algorithm.

md2.cpp

```
double gasdev () {
     static bool available = false;
     static double gset;
     double fac, rsq, v1, v2;
     if (!available) {
          do {
               v1 = 2.0 * rand() / double(RAND_MAX) - 1.0;
               v2 = 2.0 * rand() / double(RAND_MAX) - 1.0;
```

```
            rsq = v1 * v1 + v2 * v2;
        } while (rsq >= 1.0 || rsq == 0.0);
        fac = sqrt(-2.0 * log(rsq) / rsq);
        gset = v1 * fac;
        available = true;
        return v2*fac;
    } else {
        available = false;
        return gset;
    }
}

void initVelocities() {

    // Gaussian with unit variance
    for (int n = 0; n < N; n++)
        for (int i = 0; i < 3; i++)
            v[n][i] = gasdev();
```

Since these velocities are randomly distributed around zero, the total momentum of the system will be close

to zero but not exactly zero. To prevent the system from drifting in space, the center-of-mass velocity

$$\mathbf{v}_{\rm CM} = \frac{\sum_{i=1}^{N} m\mathbf{v}_i}{\sum_{i=1}^{N} m}$$

is computed and used to transform the atom velocities to the center-of-mass frame of reference.

md2.cpp

```
    // Adjust velocities so center-of-mass velocity is zero
    double vCM[3] = {0, 0, 0};
    for (int n = 0; n < N; n++)
        for (int i = 0; i < 3; i++)
            vCM[i] += v[n][i];
    for (int i = 0; i < 3; i++)
        vCM[i] /= N;
    for (int n = 0; n < N; n++)
        for (int i = 0; i < 3; i++)
            v[n][i] -= vCM[i];

    // Rescale velocities to get the desired instantaneous temperature
    rescaleVelocities();
}
```

After setting the CM velocity to zero, the velocities are *scaled*

$$\mathbf{v}_i \longrightarrow \lambda \mathbf{v}_i$$

so that the instantaneous temperature has the desired value T

$$\lambda = \sqrt{\frac{3(N-1)k_{\rm B}T}{\sum_{i=1}^{N} m v_i^2}} \, .$$

md2.cpp

```
void rescaleVelocities() {
    double vSqdSum = 0;
    for (int n = 0; n < N; n++)
        for (int i = 0; i < 3; i++)
            vSqdSum += v[n][i] * v[n][i];
    double lambda = sqrt( 3 * (N-1) * T / vSqdSum );
    for (int n = 0; n < N; n++)
        for (int i = 0; i < 3; i++)
            v[n][i] *= lambda;
}
```

Solving Newton's equations of motion

The same algorithms are used as in `md.cpp` with two improvements:

- periodic boundary conditions will be used to ensure that the number of particles in the simulation volume remains constant,
- the *minimum image convention* is used to compute the accelerations.

md2.cpp

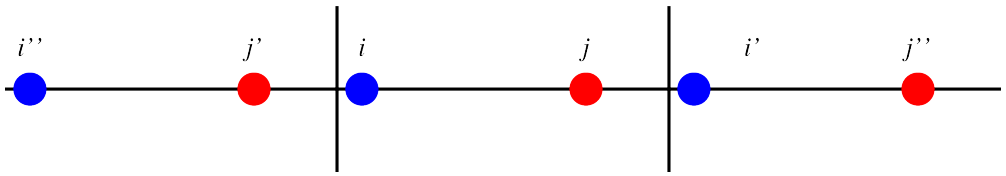
```
void computeAccelerations() {

    for (int i = 0; i < N; i++)          // set all accelerations to zero
        for (int k = 0; k < 3; k++)
            a[i][k] = 0;

    for (int i = 0; i < N-1; i++)        // loop over all distinct pairs i,j
        for (int j = i+1; j < N; j++) {
            double rij[3];               // position of i relative to j
            double rSqd = 0;
            for (int k = 0; k < 3; k++) {
                rij[k] = r[i][k] - r[j][k];
```

Since we are using periodic boundary conditions, the system actually has an infinite number of copies of the N particles contained in the volume L^3. Thus there are an infinite number of pairs of particles, all of which interact with one another! The forces between a particular particle and its periodic copies actually cancel,

but this is not true of pairs which are not images of one another. Since the Lennard Jones interaction is short ranged, we can safely neglect forces between particles in volumes that are not adjacent to one another. For adjacent volumes, we have to be more careful. It can happen that the separation $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ is *larger* than the separation $r_{ij'} = |\mathbf{r}_i - \mathbf{r}'_j|$ where j' is an image in an adjacent volume of particle j. The figure illustrates this in one dimension:



When this occurs we take into account the stronger force between i and the image j' and neglect the weaker force between i and j.

md2.cpp

```
            // closest image convention
            if (abs(rij[k]) > 0.5 * L) {
                if (rij[k] > 0)
                    rij[k] -= L;
                else
                    rij[k] += L;
            }
```



```
                rSqd += rij[k] * rij[k];
            }
            double f = 24 * (2 * pow(rSqd, -7) - pow(rSqd, -4));
            for (int k = 0; k < 3; k++) {
                 a[i][k] += rij[k] * f;
                 a[j][k] -= rij[k] * f;
            }
        }
}
```

Velocity Verlet Integration Algorithm

The same algorithm

$$\mathbf{r}_i(t+dt) = \mathbf{r}_i(t) + \mathbf{v}_i(t)dt + \frac{1}{2}\mathbf{a}_i(t)dt^2$$
$$\mathbf{v}_i(t+dt) = \mathbf{v}_i(t) + \frac{1}{2}[\mathbf{a}_i(t+dt) + \mathbf{a}_i(t)]dt$$

is used as in the simple program `md.cpp`. Periodic boundary conditions will be imposed as the time step is implemented.

md2.cpp

```
void velocityVerlet(double dt) {
```

```
    computeAccelerations();
    for (int i = 0; i < N; i++)
        for (int k = 0; k < 3; k++) {
            r[i][k] += v[i][k] * dt + 0.5 * a[i][k] * dt * dt;
```

Once the atom is moved, periodic boundary conditions are imposed to move it back into the system volume if it has exited. This done for each component of the position as soon as it has been updated:

md2.cpp

```
            // use periodic boundary conditions
            if (r[i][k] < 0)
                r[i][k] += L;
            if (r[i][k] >= L)
                r[i][k] -= L;
            v[i][k] += 0.5 * a[i][k] * dt;
        }
    computeAccelerations();
    for (int i = 0; i < N; i++)
        for (int k = 0; k < 3; k++)
            v[i][k] += 0.5 * a[i][k] * dt;
}
```


The instantaneous temperature is computed as in `md.cpp` from the equipartition formula

$$3(N-1) \times \frac{1}{2} k_{\mathrm{B}} T = \left\langle \frac{m}{2} \sum_{i=1}^{N} v_i^2 \right\rangle .$$

by the function

md2.cpp

```
double instantaneousTemperature() {
    double sum = 0;
    for (int i = 0; i < N; i++)
        for (int k = 0; k < 3; k++)
            sum += v[i][k] * v[i][k];
    return sum / (3 * (N - 1));
}
```

Finally, here is the main function which steers the simulation:

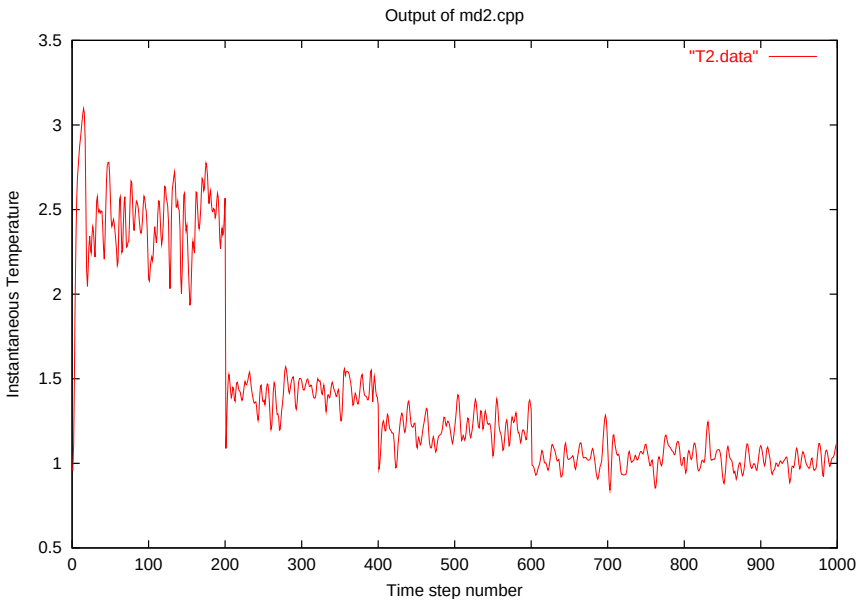
md2.cpp

```
int main() {
    initialize();
    double dt = 0.01;
    ofstream file("T2.data");
```

```
    for (int i = 0; i < 1000; i++) {
        velocityVerlet(dt);
        file << instantaneousTemperature() << '\n';
        if (i % 200 == 0)
            rescaleVelocities();
    }
    file.close();
}
```

The simulation is run for 1000 time steps. After every 200 steps, the velocities of the atoms are rescaled to drive the average temperature towards the desired value. The output shows that

- The temperature rises rapidly from the desired value $T = 1.0$ when the simulation is started. Why?
- It takes a few rescaling to push the temperature back to the desired value, and then the system appears to be in equilibrium.



Making the MD simulation more efficient

The most time consuming part of a molecular dynamics program is the computation of the forces between pairs of particles and hence the accelerations of the particles. There are $N(N-1)/2$ pairs of particles, and hence computing the forces takes time of $\mathcal{O}(N^2)$.

In a paper by L. Verlet, *Phys. Rev.* **159**, 98 (1967), two ways of speeding up the molecular dynamics simulation of Rahman were introduced:

Cut-off on the potential: Since the Lennard-Jones force is short ranged and the potential decreases rapidly with distance $r > \sigma$, it makes sense to introduce a cut-off distance $r_{\text{cut-off}}$ beyond which the potential and force are approximated by zero. If $r_{\text{cut-off}}$ is smaller than $L/2$, which is the maximum distance between interacting pairs according to the closest image convention, then the number of pairs for which the force must be computed is reduced from $N(N-1)/2$. If N is increased while holding the density of particles fixed, then the number particles which interact with a given particle remains fixed, and hence the total number of interacting pairs is of $\mathcal{O}(N)$ and not of $\mathcal{O}(N^2)$.

Neighbor list: The problem with using a cut-off is that all $N(N-1)/2$ pairs must be examined to find those for which $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j| < r_{\text{cut-off}}$. At each time step, the positions $\mathbf{r}_i$ of the particles change, so it appears that the calculation is still of $\mathcal{O}(N^2)$. However, Verlet noted that the change in positions at each time step is small because dt is chosen small to reduce numerical errors in the integration of Newton's equations.

- A maximum distance $r_{\max} > r_{\text{cut-off}}$ is chosen, and a list of all pairs (ij) with $r_{ij} < r_{\max}$ is maintained. In his paper, Verlet suggests $r_{\text{cut-off}} = 2.5\sigma$ and $r_{\max} = 3.2\sigma$.
- The list of interacting pairs is *not* updated at every time step, but rather after some fixed number of steps, say 10 or 20. This fixed update interval is chosen so that it is unlikely that a separation $r_{ij} < r_{\text{cut-off}}$ increases beyond $r_{\max}$, or a separation $r_{ij} > r_{\max}$ decreases below $r_{\text{cut-off}}$, during this interval.

Verlet found that these simple approximations made his MD simulations run ten times faster with little loss in accuracy!

Improved program `md3.cpp`

The following program implements the cut-off and neighbor lists introduced by Verlet.

First include standard libraries, and declare some variables and functions as in `md2.cpp`:

md3.cpp

```
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <string>
```

```
using namespace std;

// simulation parameters
int N = 864;              // number of particles
double rho = 1.0;         // density (number per unit volume)
double T = 1.0;           // temperature
double L;                 // will be computed from N and rho

double **r, **v, **a;     // positions, velocities, accelerations

// declare some functions
void initPositions();
void initVelocities();
void rescaleVelocities();
double instantaneousTemperature();
```

Variables and functions for cut-off and neighbor list

- Pairs with $r_{ij} < r_{\max}$ are indexed from 0 to nPairs-1, where nPairs is the number of such pairs at the time the pair list is updated.
- The indices (ij) of the pair are stored in the nPairs$\times 2$ array: pairList[n][0] = i, pairList[n][1] = j.

- $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ is stored in the nPairs$\times 3$ array: drPair[n][0] = x_{ij}, drPair[n][1] = y_{ij}, drPair[n][2] = z_{ij}.
- rSqdPair[n] = r_{ij}^2

md3.cpp

```
// variables to implement Verlet's neighbor list
double rCutOff = 2.5;     // cut-off on Lennard-Jones potential and force
double rMax = 3.3;        // maximum separation to include in pair list
int nPairs;               // number of pairs currently in pair list
int **pairList;           // the list of pair indices (i,j)
double **drPair;          // vector separations of each pair (i,j)
double *rSqdPair;         // squared separation of each pair (i,j)
int updateInterval = 10;  // number of time steps between updates of pair list

// declare functions to implement neighbor list
void computeSeparation(int, int, double[], double&);
void updatePairList();
void updatePairSeparations();

void initialize() {
    r = new double* [N];
```

```
    v = new double* [N];
    a = new double* [N];
    for (int i = 0; i < N; i++) {
        r[i] = new double [3];
        v[i] = new double [3];
        a[i] = new double [3];
    }
    initPositions();
    initVelocities();
```

The `initialize` function from `md2.cpp` is modified to allocate memory sufficient to store the maximum number $N(N-1)/2$ of pairs:

md3.cpp

```
    // allocate memory for neighbor list variables
    nPairs = ceil(N * (N - 1) / 2);
    pairList = new int* [nPairs];
    drPair = new double* [nPairs];
    for (int p = 0; p < nPairs; p++) {
        pairList[p] = new int [2];      // to store indices i and j
        drPair[p] = new double [3];     // to store components x,y,z
    }
```



```
    rSqdPair = new double [nPairs];
}
```

Compute separation between two particles

The following function computes the separation between particles i and j using periodic boundary conditions and the closest image convention.

md3.cpp

```
void computeSeparation (int i, int j, double dr[], double& rSqd) {

    // find separation using closest image convention
    rSqd = 0;
    for (int d = 0; d < 3; d++) {
        dr[d] = r[i][d] - r[j][d];
        if (dr[d] >= 0.5*L)
            dr[d] -= L;
        if (dr[d] < -0.5*L)
            dr[d] += L;
        rSqd += dr[d]*dr[d];
    }
```

```
}
```

Find all pairs with separation less than $r_{\max}$

The function `updatePairList` loops over all distinct pairs and adds pairs with separation less than $r_{\max}$ to the `pairList` array:

md3.cpp

```
void updatePairList() {
    nPairs = 0;
    double dr[3];
    for (int i = 0; i < N-1; i++)               // all distinct pairs
        for (int j = i+1; j < N; j++) {         // of particles i,j
            double rSqd;
            computeSeparation(i, j, dr, rSqd);
            if (rSqd < rMax*rMax) {
                pairList[nPairs][0] = i;
                pairList[nPairs][1] = j;
                ++nPairs;
            }
        }
```



```
}
```

Find and store all pair separations less than $r_{\max}$

The function `updatePairSeparations` computes the pair separations of all pairs in `pairList` and stores $\mathbf{r}_i - \mathbf{r}_j$ in `drPair` and $|\mathbf{r}_i - \mathbf{r}_j|^2$ in `rSqdPair`:

md3.cpp

```
void updatePairSeparations() {
    double dr[3];
    for (int p = 0; p < nPairs; p++) {
        int i = pairList[p][0];
        int j = pairList[p][1];
        double rSqd;
        computeSeparation(i, j, dr, rSqd);
        for (int d = 0; d < 3; d++)
            drPair[p][d] = dr[d];
        rSqdPair[p] = rSqd;
    }
}
```

Compute accelerations

The function `computeAccelerations` contains the crucial Verlet modifications. Instead of looping over all pairs, only those pairs in `pairList` are examined, and from these pairs, only those with $r_{ij} < r_{\text{cut-off}}$ are actually used in the force calculation. This makes the function execute much faster than the corresponding function in `md2.cpp`.

md3.cpp

```
void computeAccelerations() {

    for (int i = 0; i < N; i++)          // set all accelerations to zero
        for (int k = 0; k < 3; k++)
            a[i][k] = 0;

    for (int p = 0; p < nPairs; p++) {
        int i = pairList[p][0];
        int j = pairList[p][1];
        if (rSqdPair[p] < rCutOff*rCutOff) {
            double r2Inv = 1 / rSqdPair[p];
            double r6Inv = r2Inv*r2Inv*r2Inv;
            double f = 24*r2Inv*r6Inv*(2*r6Inv - 1);
```



```
            for (int d = 0; d < 3; d++) {
                a[i][d] += f * drPair[p][d];
                a[j][d] -= f * drPair[p][d];
            }
        }
    }
}
```

Velocity-Verlet integration algorithm

The function `velocityVerlet` is modified from `md2.cpp` in two ways:

- The accelerations are computed only once each time step. This simple change should speed up the program considerably.
- At each time step, `updatePairSeparations` is called after all of the particle positions have been updated. The new forces and accelerations can then be computed.

md3.cpp

```
void velocityVerlet(double dt) {
    // assume accelerations have been computed
    for (int i = 0; i < N; i++)
```

```
        for (int k = 0; k < 3; k++) {
            r[i][k] += v[i][k] * dt + 0.5 * a[i][k] * dt * dt;

            // use periodic boundary conditions
            if (r[i][k] < 0)
                r[i][k] += L;
            if (r[i][k] >= L)
                r[i][k] -= L;
            v[i][k] += 0.5 * a[i][k] * dt;
        }
    updatePairSeparations();
    computeAccelerations();
    for (int i = 0; i < N; i++)
        for (int k = 0; k < 3; k++)
            v[i][k] += 0.5 * a[i][k] * dt;
}
```

Steering the simulation

The `main` function is modified to call `updatePairList` every `updateInterval` time steps.

md3.cpp


```
int main() {
    initialize();
    updatePairList();
    updatePairSeparations();
    computeAccelerations();
    double dt = 0.01;
    ofstream file("T3.data");
    for (int i = 0; i < 1000; i++) {
        velocityVerlet(dt);
        file << instantaneousTemperature() << '\n';
        if (i % 200 == 0)
            rescaleVelocities();
        if (i % updateInterval == 0) {
            updatePairList();
            updatePairSeparations();
        }
    }
    file.close();
}
```

Functions repeated from `md2.cpp`

md3.cpp

```
void initPositions() {

    // compute side of cube from number of particles and number density
    L = pow(N / rho, 1.0/3);

    // find M large enough to fit N atoms on an fcc lattice
    int M = 1;
    while (4 * M * M * M < N)
        ++M;
    double a = L / M;           // lattice constant of conventional cell

    // 4 atomic positions in fcc unit cell
    double xCell[4] = {0.25, 0.75, 0.75, 0.25};
    double yCell[4] = {0.25, 0.75, 0.25, 0.75};
    double zCell[4] = {0.25, 0.25, 0.75, 0.75};

    int n = 0;                  // atoms placed so far
    for (int x = 0; x < M; x++)
```



```
        for (int y = 0; y < M; y++)
            for (int z = 0; z < M; z++)
                for (int k = 0; k < 4; k++)
                    if (n < N) {
                        r[n][0] = (x + xCell[k]) * a;
                        r[n][1] = (y + yCell[k]) * a;
                        r[n][2] = (z + zCell[k]) * a;
                        ++n;
                    }
}

double gasdev () {
     static bool available = false;
     static double gset;
     double fac, rsq, v1, v2;
     if (!available) {
          do {
               v1 = 2.0 * rand() / double(RAND_MAX) - 1.0;
               v2 = 2.0 * rand() / double(RAND_MAX) - 1.0;
               rsq = v1 * v1 + v2 * v2;
          } while (rsq >= 1.0 || rsq == 0.0);
          fac = sqrt(-2.0 * log(rsq) / rsq);
```

```
          gset = v1 * fac;
          available = true;
          return v2*fac;
     } else {
          available = false;
          return gset;
     }
}

void initVelocities() {

    // Gaussian with unit variance
    for (int n = 0; n < N; n++)
        for (int i = 0; i < 3; i++)
            v[n][i] = gasdev();
    // Adjust velocities so center-of-mass velocity is zero
    double vCM[3] = {0, 0, 0};
    for (int n = 0; n < N; n++)
        for (int i = 0; i < 3; i++)
            vCM[i] += v[n][i];
    for (int i = 0; i < 3; i++)
        vCM[i] /= N;
```



```
    for (int n = 0; n < N; n++)
        for (int i = 0; i < 3; i++)
            v[n][i] -= vCM[i];

    // Rescale velocities to get the desired instantaneous temperature
    rescaleVelocities();
}

void rescaleVelocities() {
    double vSqdSum = 0;
    for (int n = 0; n < N; n++)
        for (int i = 0; i < 3; i++)
            vSqdSum += v[n][i] * v[n][i];
    double lambda = sqrt( 3 * (N-1) * T / vSqdSum );
    for (int n = 0; n < N; n++)
        for (int i = 0; i < 3; i++)
            v[n][i] *= lambda;
}

double instantaneousTemperature() {
    double sum = 0;
    for (int i = 0; i < N; i++)
```

```
        for (int k = 0; k < 3; k++)
            sum += v[i][k] * v[i][k];
    return sum / (3 * (N - 1));
}
```

Output of the neighbor list program

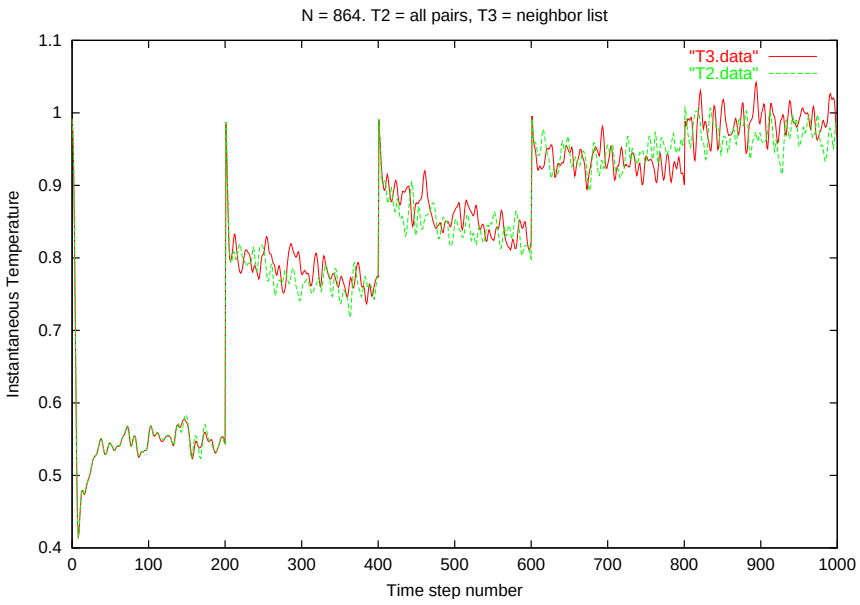
The figure compares the output of `md3.cpp` with that of `md2.cpp` with $N = 864$ particles. Cutting off the Lennard-Jones force at $r_{\text{cut-off}}$ does not appreciably affect the results of the simulation. Running the two programs shows that `md3.cpp` is roughly 10 times faster.

Correcting for the cut-off

The differences between the outputs of `md3.cpp` and `md2.cpp` are due to the use of the cut-off potential. Cutting off the force violates energy conservation and also causes errors in integrating Newton's equations of motion. These effects can be corrected by using a modified potential:

$$U_{\text{force shift}}(r) = U(r) - \frac{d}{dr}U(r_{\text{cut-off}})\,(r - r_{\text{cut-off}}) \ .$$

Since the potential has been changed, observables such as the pressure and average potential energy will not have the same values as for the original Lennard-Jones potential. It is possible to correct for these deviations in the MD simulation program.



Molecular Dynamics of 2-D Lennard-Jones System

The following OpenGL code simulates a 2-dimensional Lennard-Jones gas.

md-2d.cpp

```
// Molecular Dynamics of 2D Lennard-Jones System: OpenGL Visualization

#include <cmath>
#include <cstdlib>
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

#ifdef __APPLE__
#   include <GLUT/glut.h>
#else
#   include <GL/freeglut.h>
#endif
```



```
int N = 16;                     // number of molecules
double L = 6;                   // linear size of square region
double kT = 1;                  // initial kinetic energy/molecule
int skip = 0;                   // steps to skip to speed graphics

void getInput() {
    cout << "Molecular Dynamics simulation of 2D Lennard-Jones system\n";
    cout << "Enter number of molecules N = ";
    cin >> N;
    cout << "Enter size of square region L = ";
    cin >> L;
    cout << "Enter kinetic energy/molecule in units of kT: ";
    cin >> kT;
    cout << "Enter steps to skip to speed graphics: ";
    cin >> skip;
}

double *x, *y, *vx, *vy;        // position and velocity components
double *ax, *ay;                // acceleration components

void computeAccelerations() {
```

```
    for (int i = 1; i <= N; i++)
        ax[i] = ay[i] = 0;

    for (int i = 1; i < N; i++)
    for (int j = i + 1; j <= N; j++) {
        double dx = x[i] - x[j];
        double dy = y[i] - y[j];
        // use closest periodic image
        if (abs(dx) > 0.5 * L)
            dx *= 1 - L / abs(dx);
        if (abs(dy) > 0.5 * L)
            dy *= 1 - L / abs(dy);
        double dr = sqrt(dx * dx + dy * dy);
        double f = 48 * pow(dr, -13.0) - 24 * pow(dr, -7.0);
        ax[i] += f * dx / dr;
        ay[i] += f * dy / dr;
        ax[j] -= f * dx / dr;
        ay[j] -= f * dy / dr;
    }
}

double t = 0;                   // time
```



```
double dt = 0.01;               // integration time step
int step = 0;                   // step number
double T;                       // temperature
double Tsum;                    // to compute average T
int step0;                      // starting step for computing average

int nBins = 50;                 // number of velocity bins
double *vBins;                  // for Maxwell-Boltzmann distribution
double vMax = 4;                // maximum velocity to bin
double dv = vMax / nBins;       // bin size

void resetHistogram() {
    for (int bin = 0; bin <= nBins; bin++)
        vBins[bin] = 0;
    Tsum = 0;
    step0 = step;
}

void initialize() {

    // allocate storage
    x = new double [N+1];
```

```
    y = new double [N+1];
    vx = new double [N+1];
    vy = new double [N+1];
    ax = new double [N+1];
    ay = new double [N+1];
    vBins = new double[nBins+1];

    // initialize positions on a square lattice
    int m = int(ceil(sqrt(double(N))));  // molecules in each direction
    double d = L / m;                    // lattice spacing
    int n = 0;                           // molecules placed so far
    for (int i = 0; i < m; i++)
    for (int j = 0; j < m; j++) {
        if (n < N) {
            ++n;
            x[n] = (i + 0.5) * d;
            y[n] = (j + 0.5) * d;
        }
    }

    // initialize velocities with random directions
    double pi = 4 * atan(1.0);
```



```
    double v = sqrt(2 * kT);
    for (int n = 1; n <= N; n++) {
        double theta = 2 * pi * rand() / double(RAND_MAX);
        vx[n] = v * cos(theta);
        vy[n] = v * sin(theta);
    }

    computeAccelerations();
    resetHistogram();
    T = kT;
}

void timeStep () {

    t += dt;
    ++step;
    double K = 0;
    for (int i = 1; i <= N; i++) {

        // integrate using velocity Verlet algorithm
        x[i] += vx[i] * dt + 0.5 * ax[i] * dt * dt;
        y[i] += vy[i] * dt + 0.5 * ay[i] * dt * dt;
```

```
        // periodic boundary conditions
        if (x[i] < 0) x[i] += L;
        if (x[i] > L) x[i] -= L;
        if (y[i] < 0) y[i] += L;
        if (y[i] > L) y[i] -= L;

        vx[i] += 0.5 * ax[i] * dt;
        vy[i] += 0.5 * ay[i] * dt;

        computeAccelerations();

        vx[i] += 0.5 * ax[i] * dt;
        vy[i] += 0.5 * ay[i] * dt;

        double v = sqrt(vx[i] * vx[i] + vy[i] * vy[i]);
        K += 0.5 * v * v;
        int bin = (int) (nBins * v / vMax);
        if (bin > 0 && bin <= nBins)
            ++vBins[bin];
    }
    Tsum += K / N;
```



```
    T = Tsum / (step - step0);
}

void scaleVelocities(int heat) {
    double scale = heat ? 1.2 : 1/1.2;
    for (int n = 1; n <= N; n++) {
        vx[n] *= scale;
        vy[n] *= scale;
    }
    resetHistogram();
}

int mainWindow, molsWindow, histWindow;
int margin = 10;

void takeStep() {
    for (int i = 0; i < skip; i++)
        timeStep();
    timeStep();
    glutSetWindow(molsWindow);
    glutPostRedisplay();
    if (step % 10 == 0) {
```

```
        glutSetWindow(histWindow);
        glutPostRedisplay();
    }
}

void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glutSwapBuffers();
}

void drawText(const string &str, double x, double y) {
    glRasterPos2d(x, y);
    int len = str.find('\0');
    for (int i = 0; i < len; i++)
       glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, str[i]);
}

void displayMols() {
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3ub(0, 0, 255);
    double pi = 4 * atan(1.0);
    for (int n = 1; n <= N; n++) {
```



```
        glBegin(GL_TRIANGLE_FAN);
            glVertex2d(x[n], y[n]);
            double phi = 2 * pi / 24;
            for (int j = 0; j < 25; j++)
                glVertex2d(x[n] + 0.5 * cos(phi*j), y[n] + 0.5 * sin(phi*j));
        glEnd();
    }
    glColor3ub(255, 255, 255);
    ostringstream os;
    os << "Step No: " << step << "   Time t = " << t << ends;
    drawText(os.str(), L / 50, L / 50);
    os.seekp(0);
    glutSwapBuffers();
}

double MaxwellBoltzmann(double v, double T) {
    double Pmax = sqrt(T) * exp(-0.5);
    return v * exp( -0.5 * v * v / T) / Pmax;
}

void displayHist() {
    glClear(GL_COLOR_BUFFER_BIT);
```

```
    glColor3ub(0, 0, 0);
    ostringstream os;
    os << "Temperature T = " << T << ends;
    drawText(os.str(), vMax / 2, 0.95);
    os.seekp(0);
    os << "Press left button to heat/cool" << ends;
    drawText(os.str(), vMax / 2, 0.9);
    os.seekp(0);

    double Pmax = 0;
    for (int b = 1; b <= nBins; b++)
        if (Pmax < vBins[b])
           Pmax = vBins[b];
    if (Pmax == 0) {
        glutSwapBuffers();
        return;
    }

    // draw velocity histogram
    glColor3ub(0, 255, 255);
    double dv = vMax / nBins;
```



```
    for (int b = 1; b <= nBins; b++) {
        double v = b * dv;
        double P = vBins[b] / Pmax;
        glBegin(GL_POLYGON);
            glVertex2d(v - dv/3, 0);
            glVertex2d(v - dv/3, P);
            glVertex2d(v + dv/3, P);
            glVertex2d(v + dv/3, 0);
        glEnd();
    }

    // compare with Maxwell Boltzmann
    glColor3ub(255, 0, 255);
    glBegin(GL_LINE_STRIP);
        glVertex2d(0, 0);
        for (int b = 1; b < nBins; b++) {
            double v = b * dv;
            double P = MaxwellBoltzmann(v, T);
            glVertex2d(v, P);
        }
    glEnd();
```

```
    glColor3ub(0, 0, 0);
    os << "v_max = " << vMax << ends;
    drawText(os.str(), 0.8 * vMax, 0.02);
    os.seekp(0);

    glutSwapBuffers();
}

void reshape(int w, int h) {

    glutSetWindow(molsWindow);
    int width = h - 2*margin;
    glutReshapeWindow(width, width);
    glViewport(0, 0, width, width);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, L, 0, L);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    glutSetWindow(histWindow);
    glutPositionWindow(h + margin, margin);
```



```
    width = w - h - 2*margin;
    if (width < h - 2*margin)
        width = h - 2*margin;
    glutReshapeWindow(width, h - 2*margin);
    glViewport(0, 0, width, width);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, vMax, 0, 1);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

bool running = false;

void molsMouse(int button, int state, int x, int y) {
    switch (button) {
    case GLUT_LEFT_BUTTON:
        if (state == GLUT_DOWN) {
            if (running) {
                glutIdleFunc(NULL);
                running = false;
            } else {
```

```
                glutIdleFunc(takeStep);
                running = true;
            }
        }
    }
}

void makeWindows() {

    // main window
    glutInitWindowSize(800, 400);
    glutInitWindowPosition(100, 100);
    mainWindow = glutCreateWindow("Molecular Dynamics of 2D "
                                  " Lennard-Jones System");
    glShadeModel(GL_FLAT);
    glClearColor(1.0, 1.0, 1.0, 0.0);
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);

    // molecules window
    molsWindow = glutCreateSubWindow(mainWindow, margin, margin,
                                     400 - 2*margin, 400 - 2*margin);
```



```
    glClearColor(1.0, 0.0, 0.0, 0.0);
    glutDisplayFunc(displayMols);
    glutMouseFunc(molsMouse);

    // histogram window
    histWindow = glutCreateSubWindow(mainWindow, 400 + margin, margin,
                                     400 - 2*margin, 400 - 2*margin);
    glClearColor(0.0, 1.0, 0.0, 0.0);
    glutDisplayFunc(displayHist);
    glutCreateMenu(scaleVelocities);
    glutAddMenuEntry("Heat", 1);
    glutAddMenuEntry("Cool", 0);
    glutAttachMenu(GLUT_LEFT_BUTTON);
}

int main(int argc, char *argv[]) {
    glutInit(&argc, argv);
    if (argc == 1)
        getInput();
    initialize();
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    makeWindows();
```

```
    glutMainLoop();
}
```

Molecular Dynamics Simulation of Argon: OpenGL Visualization