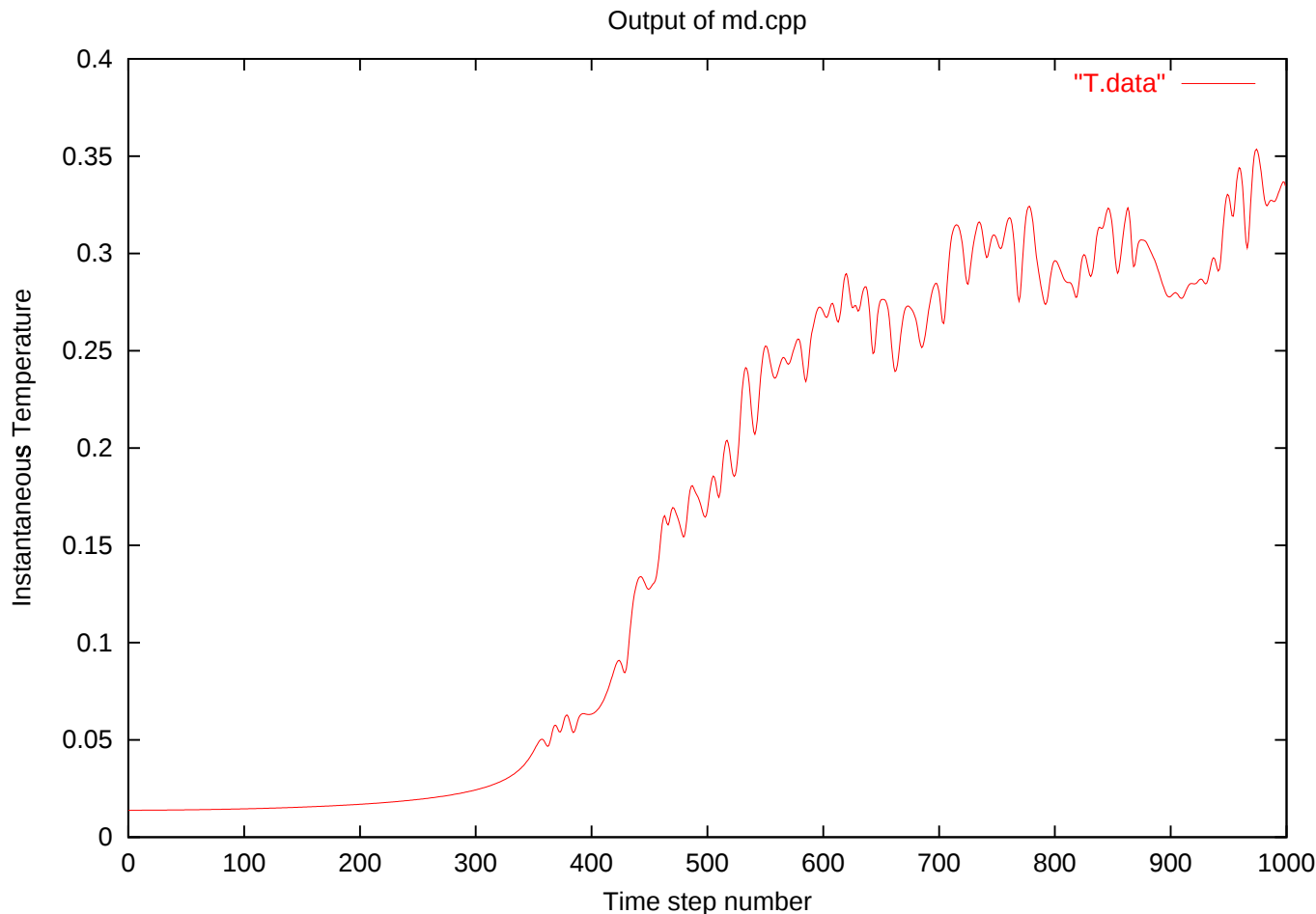


A simple MD program (continued)

The code `md.cpp` simulated the motion of 64 particles placed on a cubic lattice with random initial velocities for 1000 time steps of size $dt = 0.01$.

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. Can you understand this behavior?

Improving the MD program

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

// simulation parameters

int N = 64;           // number of particles
double rho = 1.2;     // 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 instantaneous temperature to T
double gasdev();      // Gaussian distributed random numbers
```

We will allocate particle arrays dynamically rather than statically

```
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 $N = 4M^2$, where $M = 1, 2, 3, \dots$, then the atoms can fill a cubical volume. So MD simulations are usually done with $32 = 4 \times 2^3$, $108 = 4 \times 3^3$, $256, 500, 864, \dots$ atoms.

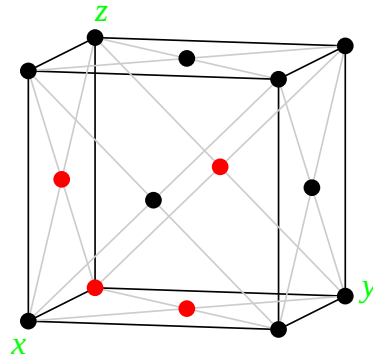
```
double L;                                // linear size of cubical volume  
  
void initPositions() {  
  
    // compute side of cube from number of particles and number density  
    L = pow(N / rho, 1.0/3);  
  
    // find M large enough to fit N atoms on an fcc lattice
```

```

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

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

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.

```

// 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.

```
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 :

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

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, also explained in Appendix B of Thijssen.

```
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

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

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

```
// 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

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

```
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

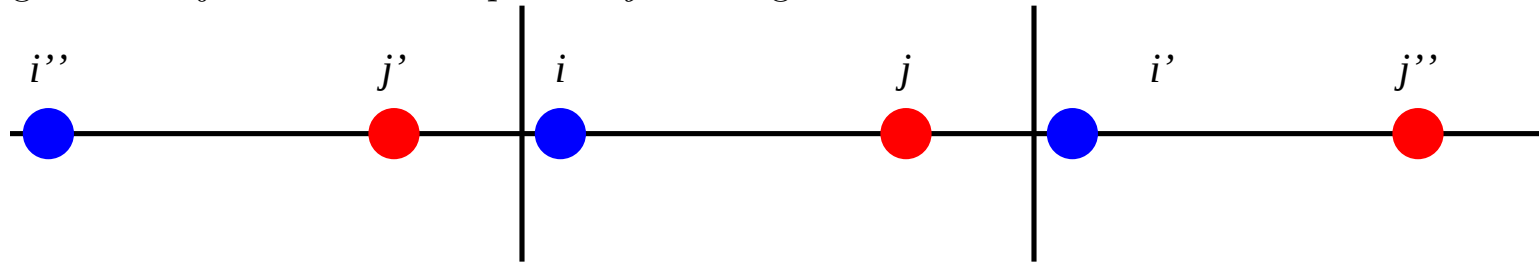
The same algorithms are used as in `md.cpp` with two improvements:

- 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.

```
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 .

```
// 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 = 12 * (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.

```

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:

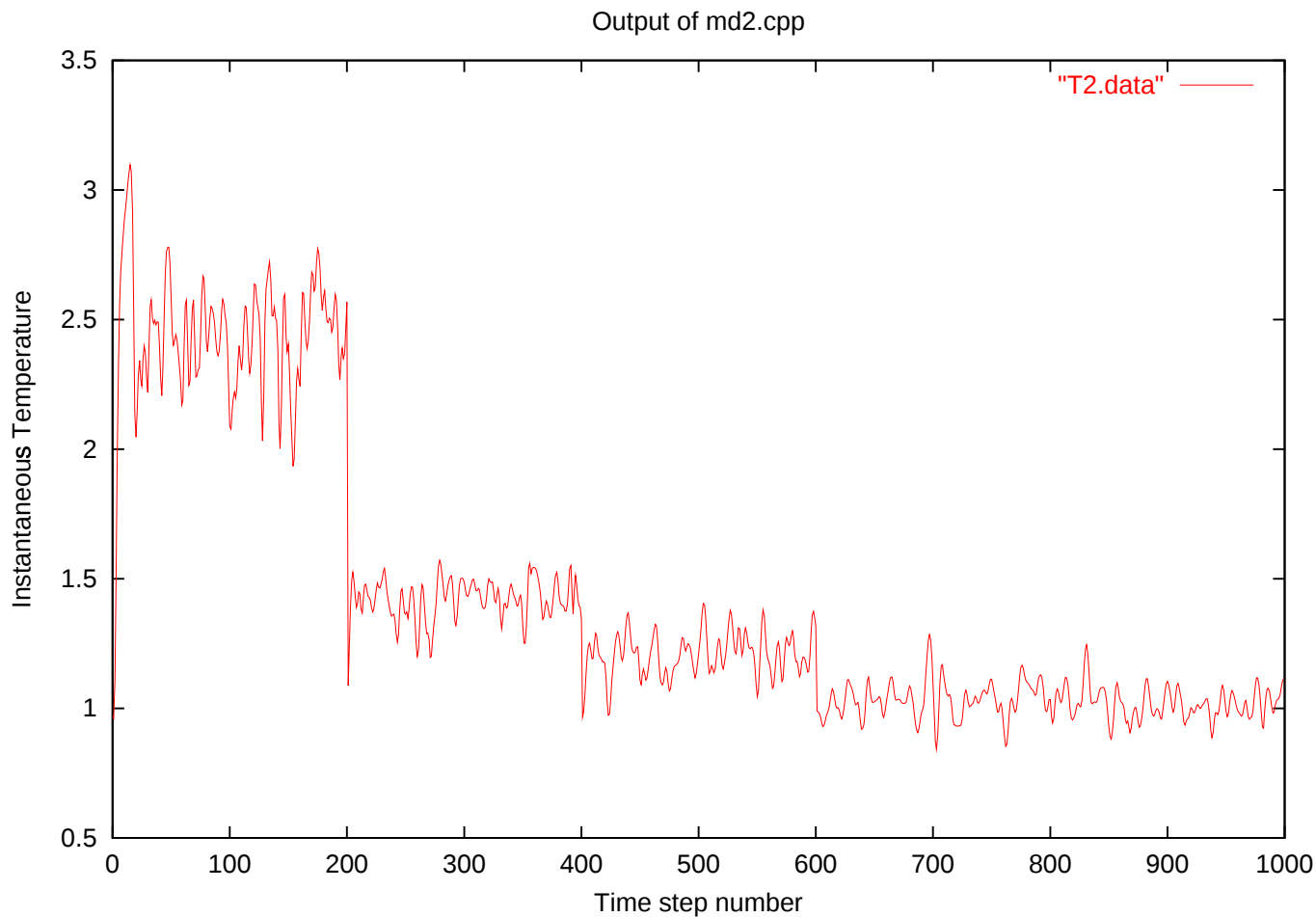
```
// use periodic boundary conditions
if (r[i][k] < 0)
    r[i][k] += L;
if (r[i][k] >= L)
    r[i][k] -= L;
v[i][k] += 0.5 * a[i][k] * dt;
}
computeAccelerations();
for (int i = 0; i < N; i++)
    for (int k = 0; k < 3; k++)
        v[i][k] += 0.5 * a[i][k] * dt;
}
```

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

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

by the function

```
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:

```
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.