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 $O(N^2)$.

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 of particles which interact with a given particle remains fixed, and hence the total number of interacting pairs is of $O(N)$ and not of $O(N^2)$.

- **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 $r_{ij} = |r_i - r_j| < r_{\text{cut-off}}$. At each time step, the positions $r_i$ of the particles change, so it appears that the calculation is still of $O(N^2)$. 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 $r_{\text{max}} > r_{\text{cut-off}}$ is chosen, and a list of all pairs $(ij)$ with $r_{ij} < r_{\text{max}}$ is maintained. In his paper, Verlet suggests $r_{\text{cut-off}} = 2.5\sigma$ and $r_{\text{max}} = 3.2\sigma$. 
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 $r_{\text{max}}$, or a separation $r_{ij} > r_{\text{max}}$ 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:

```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 $r_{ij} < r_{\text{max}}$ are indexed from 0 to $n\text{Pairs} - 1$, where $n\text{Pairs}$ is the number of such pairs at the time the pair list is updated.

• The indices $(ij)$ of the pair are stored in the $n\text{Pairs} \times 2$ array: $\text{pairList}[n][0] = i$, $\text{pairList}[n][1] = j$.

• $r_{ij} = r_i - r_j$ is stored in the $n\text{Pairs} \times 3$ array: $\text{drPair}[n][0] = x_{ij}$, $\text{drPair}[n][1] = y_{ij}$, $\text{drPair}[n][2] = z_{ij}$.

• $r_{\text{SqdPair}}[n] = r_{ij}^2$

// 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:

```cpp
// allocate memory for neighbor list variables
nPairs = 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.

```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++) {
```
Find all pairs with separation less than $r_{\text{max}}$

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

```c
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;
            }
        }
}
```
Find and store all pair separations less than $r_{\text{max}}$

The function `updatePairSeparations` computes the pair separations of all pairs in `pairList` and stores $r_i - r_j$ in `drPair` and $|r_i - r_j|^2$ in `rSqdPair`:

```c
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.

```c
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.

```c
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.

```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

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;
            if (rsq < 1.0) {  // Accept the pair with probability 1/\pi
                available = true;
                gset = sqrt(-2.0 * log(rsq) / rsq);
                break;
            }
        } while (true);
    }
    return fac * gset;
}
while (rsq >= 1.0 || rsq == 0.0);
fac = sqrt(-2.0 * log(rsq) / rsq);
gset = v1 * fac;
available = true;
return v2*fac;
}
}

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.


OpenGL Animation of the Molecular Dynamics Simulation

411-506 Computational Physics 2

Friday, January 18, 2013
It is interesting and useful to be able to visualize the state of the system of atoms as the simulation is running. Unlike Java, C++ has no standard graphics library functions. There are many specialized graphics libraries for particular applications. One of the best and most efficient general purpose libraries is OpenGL. The OpenGL functions are written in C, and bindings exist for many languages including C++.

The GLUT library enables one to write relatively simple OpenGL graphics programs which are portable across operating systems (Unix, Linux, Windows, Mac OS).

**OpenGL program md4.cpp**

In this fourth version of the basic MD program, we will add functions to display the state of the system after every few integration time steps. The OpenGL and GLUT library definitions are included via the header file `glut.h`, which automatically includes `gl.h` and `glu.h`.

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

#include <GL/glut.h>
```
We allow the user to enter simulation parameters

```cpp
// simulation parameters
int N; // number of particles
double rho; // density (number per unit volume)
double T; // temperature
double dt; // integration time step

void getInput() {
    cout << "Molecular Dynamics of 3D Lennard-Jones Gas" << endl
    << "------------------------------------------" << endl;
    cout << "Enter desired number of particles N = ";
    cin >> N;
    cout << "Enter density (particles/unit vol) rho = ";
    cin >> rho;
    cout << "Enter desired temperature T = ";
    cin >> T;
    cout << "Enter desired integration time step dt = ";
    cin >> dt;
}
```

The following variables and functions were used in md2.cpp:
double L; // will be computed from N and rho

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

// declare functions repeated from md2.cpp
void initPositions();
void initVelocities();
void rescaleVelocities();
double instantaneousTemperature();

The following variables and functions were used in 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 repeated from md3.cpp
void initialize();
void computeSeparation(int, int, double[], double&);  
void updatePairList();  
void updatePairSeparations();  
void computeAccelerations();  
void velocityVerlet(double dt);

New variables and functions for OpenGL animation

We now introduce some variables and a function which advances the simulation by one time step.

int step; // keeps track of integration step number
int displayInterval = 5; // display molecules every so many steps

void makeMolecules(); // this function re-draws the molecules

void takeStep() {
    velocityVerlet(dt);
    ++step;
    if (step % 200 == 0)
        rescaleVelocities();
    if (step % updateInterval == 0) {
        updatePairList();
        updatePairSeparations();
The following variables are used to draw a 3D picture of the system:

```cpp
const double pi = 4 * atan(1.0);
double radius = 0.5; // radius of molecule
double minExtent[3], maxExtent[3]; // extent of system volume
int xWindowSize = 640, yWindowSize = 640; // window size in screen pixels
GLdouble aspectRatio; // window aspect ratio
GLdouble fovy, nearClip, farClip; // variables for 3D projection
GLdouble eye[3], center[3], up[3]; // more projection variables
GLuint sphereID, configID; // display list ID numbers
int phi, theta; // to rotate system using arrow keys
int angle = 5; // rotation angle in degrees
```

The following functions generate *display lists* that store the OpenGL commands to draw a sphere, and to draw the collection of spheres, in an efficient *compiled* form:
void makeSphere(GLuint listID, double radius) {
    int nTheta = 9; // number of polar angle slices
    int nPhi = 18;  // number of azimuthal angle slices
    glNewList(listID, GL_COMPILE);
    glutSolidSphere(radius, nPhi, nTheta);
    glEndList();
}

void makeMolecules() {
    glNewList(configID, GL_COMPILE);
    glColor3f(0.0, 0.0, 1.0); // color the molecules blue
    glPushMatrix();
    glRotated(phi, 0, 1, 0);  // rotate about y axis
    glPushMatrix();
    glRotated(theta, 1, 0, 0); // rotate about x axis
    for (int i = 0; i < N; i++) {
        glPushMatrix();
        glTranslated(r[i][0] - L/2, r[i][1] - L/2, r[i][2] - L/2);
        glCallList(sphereID);
        glPopMatrix();
    }
    glColor3ub(255, 255, 255); // white
    glutWireCube(L); // cubical system volume
glPopMatrix();
glPopMatrix();
glEndList();

The following *callback* functions are invoked by the window system when the window needs to be re-drawn, and when the window has been re-shaped:

```c
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(eye[0], eye[1], eye[2],
              center[0], center[1], center[2],
              up[0], up[1], up[2]);
    glCallList(configID); // draw molecules
    glutSwapBuffers();
}

void reshape(int w, int h) {
    glViewport(0, 0, w, h);
    aspectRatio = w / double(h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
The following function sets up lighting and perspective viewing:

```c
void initView(double *minExtent, double *maxExtent) {
    // use a single light source to illuminate the scene
    GLfloat lightDiffuse[] = {1.0, 1.0, 1.0, 1.0};
    GLfloat lightPosition[] = {0.5, 0.5, 1.0, 0.0};
    glLightfv(GL_LIGHT0, GL_DIFFUSE, lightDiffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);               // use single light number 0
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_COLOR_MATERIAL);

    // compute the distance scale of the system
    double difExtent[3];
    for (int i = 0; i < 3; i++)
        difExtent[i] = maxExtent[i] - minExtent[i];
    double dist = 0;
}
```
for (int i = 0; i < 3; i++)
    dist += difExtent[i] * difExtent[i];
dist = sqrt(dist);

// locate the center of the system, camera position, and orientation
for (int i = 0; i < 3; i++)
    center[i] = minExtent[i] + difExtent[i] / 2;
eye[0] = center[0];
eye[1] = center[1];
up[0] = 0;
up[1] = 1;                     // y axis is up
up[2] = 0;

// set up clipping planes, field of view angle in degrees in y direction
nearClip = (dist - difExtent[2] / 2) / 2;
farClip = 2 * (dist + difExtent[2] / 2);
fovy = 2 * atan(fovy) / pi * 180;
fovy *= 1.2;
}
```c
void special(int key, int x, int y) {
    switch(key) {
        case GLUT_KEY_LEFT:   phi = (phi - angle) % 360; break;
        case GLUT_KEY_RIGHT:  phi = (phi + angle) % 360; break;
        case GLUT_KEY_UP:     theta = (theta - angle) % 360; break;
        case GLUT_KEY_DOWN:   theta = (theta + angle) % 360; break;
        default: break;
    }
}
```

Here is the main function which steers the simulation:

```c
int main(int argc, char *argv[]) {

    glutInit(&argc, argv);

    getInput();
    initialize();
    updatePairList();
    updatePairSeparations();
    computeAccelerations();
```
dt = 0.01;

glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
glutInitWindowSize(xWindowSize, yWindowSize);
glutCreateWindow("Molecular Dynamics of Lennard-Jones Gas");

for (int i = 0; i < 3; i++) {
    minExtent[i] = -L/2;
    maxExtent[i] = L/2;
}
initView(minExtent, maxExtent);

glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutIdleFunc(takeStep);
glutSpecialFunc(special);

sphereID = glGenLists(1);
makeSphere(sphereID, radius);
configID = glGenLists(1);
makeMolecules();

 glutMainLoop();
Functions repeated from md3.cpp

```cpp
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();

    // allocate memory for neighbor list variables
    nPairs = 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];
```
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];
    }
}

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) {

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;
    }
}

void computeAccelerations() {

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

\[
\text{for (int } p = 0; p < nPairs; p++) { \\
    \text{int } i = \text{pairList}[p][0]; \\
    \text{int } j = \text{pairList}[p][1]; \\
    \text{if (rSqdPair}[p] < rCutOff*rCutOff) { \\
        \text{double } r2Inv = 1 / \text{rSqdPair}[p]; \\
        \text{double } r6Inv = r2Inv*r2Inv*r2Inv; \\
        \text{double } f = 24*r2Inv*r6Inv*(2*r6Inv - 1); \\
        \text{for (int } d = 0; d < 3; d++) { \\
            a[i][d] += f * \text{drPair}[p][d]; \\
            a[j][d] -= f * \text{drPair}[p][d]; \\
        } \\
    } \\
}\]

\[
\text{void velocityVerlet(double } dt) { \\
    \text{// assume accelerations have been computed} \\
    \text{for (int } i = 0; i < N; i++) { \\
        \text{for (int } k = 0; k < 3; k++) { \\
            r[i][k] += v[i][k] * dt + 0.5 * a[i][k] * dt * dt; \\
        } \\
    } \\
    \text{// use periodic boundary conditions} \\
}\]
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;

Functions repeated from md2.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));
}