

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 $r_{\text{max}} > r_{\text{cut-off}}$ 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_{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`:

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

// simulation parameters
int N = 864;           // number of particles
double rho = 1.2;      // 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 $n\text{Pairs} \times 2$ array: $\text{pairList}[n][0] = i, \text{pairList}[n][1] = j$.
- $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ is stored in the $n\text{Pairs} \times 3$ array: $\text{drPair}[n][0] = x_{ij}, \text{drPair}[n][1] = y_{ij}, \text{drPair}[n][2] = z_{ij}$.
- $\text{rSqPair}[n] = r_{ij}^2$

```
// 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 *rSqPair;          // 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:

```
// 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.

```
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:

```
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 `rSqPair`:

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

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`.

```
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 = 12*r2Inv*r6Inv*(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.

```
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.

```
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;
    } 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];
}
```

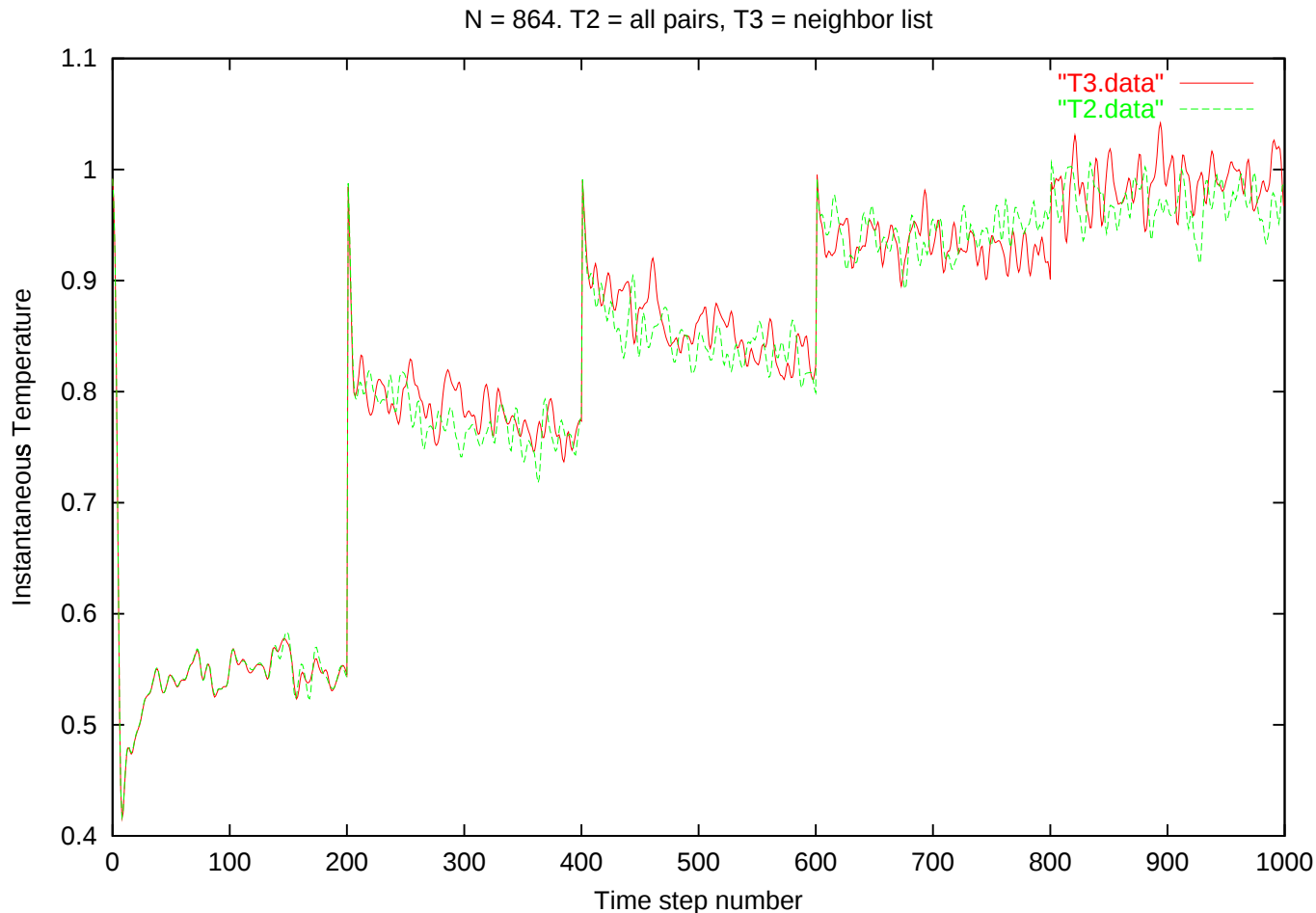
```
// Rescale velocities to get the desired instantaneous temperature
rescaleVelocities();
}

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. This effect is discussed in §8.2 of Thijssen's book. 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 the modified potential given in Eq. (8.12):

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

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. Eqs. (8.17) and (8.18) show how these discrepancies can be corrected for in a simulation. It is straightforward to include these corrections in the MD simulation program.