

Topic 2: Molecular Dynamics of Lennard-Jones System (continued)

The program files `md.h` and `md.c` implement some basic functions required for the simulation.

- The function `computeForces` contains the basic `for` loop which evaluates the accelerations of the N particles and stores them in the array `double a[MAX_N][MAX_DIM]`:

```
for (n = 0; n < N; n++)
    for (d = 0; d < dim; d++)
        a[n][d] = 0;

for (i = 0; i < N - 1; i++) {
    for (j = i + 1; j < N; j++) {
        rSqd = 0;
        for (d = 0; d < dim; d++) {
            rij[d] = r[i][d] - r[j][d];
            sign = rij[d] > 0 ? +1 : -1;
            mag = rij[d] * sign;
            if (mag > 0.5 * L)
                rij[d] -= sign * L;
            rSqd += mag * mag;
        }
        rInv2 = 1 / rSqd;
        rInv6 = rInv2 * rInv2 * rInv2;
    }
}
```

```

    rInv12 = rInv6 * rInv6;
    mag = 24 * rInv2 * (2 * rInv12 - rInv6);
    for (d = 0; d < dim; d++) {
        aij[d] = rij[d] * mag;
        a[i][d] += aij[d];
        a[j][d] -= aij[d];
    }
}
}

```

Note the use of the *nearest image approximation* in computing the distance between any two particles: if any component of $\mathbf{r}_i - \mathbf{r}_j$ is larger than $L/2$, then because we are using periodic boundary conditions, there is an image particle located closer which exerts a larger force.

- The function `takeTimeStep` implements the velocity-Verlet algorithm

$$\mathbf{r}_i(t+h) = \mathbf{r}_i(t) + h\mathbf{v}_i(t) + \frac{h^2}{2}\mathbf{a}_i(t)$$

$$\mathbf{v}_i(t+h) = \mathbf{v}_i(t) + \frac{h}{2}[\mathbf{a}_i(t+h) + \mathbf{a}_i(t)]$$

```

for (n = 0; n < N; n++) {
    for (d = 0; d < dim; d++) {
        r[n][d] += v[n][d] * dt + 0.5 * a[n][d] * dt * dt;
        if (r[n][d] < 0.0)

```

```
                r[n][d] += L;
            if (r[n][d] >= L)
                r[n][d] -= L;
            v[n][d] += 0.5 * a[n][d] * dt;
        }
    }
    computeForces();
    for (n = 0; n < N; n++)
        for (d = 0; d < dim; d++)
            v[n][d] += 0.5 * a[n][d] * dt;
```

Note that the application of periodic boundary conditions after the particle positions are updated.

- The function `rescaleVelocities` is used to adjust the temperature T of the system to the desired value:

```
vSqdSum = 0;
for (n = 0; n < N; n++)
    for (d = 0; d < dim; d++)
        vSqdSum += v[n][d] * v[n][d];
scale = dim * (N - 1) * T / vSqdSum;
scale = sqrt(scale);
for (n = 0; n < N; n++)
    for (d = 0; d < dim; d++)
```

```
v[n][d] *= scale;
```

This simulation is performed at constant N, V, E (microcanonical ensemble). The temperature T is determined by the average kinetic energy

$$\frac{3}{2}Nk_{\text{B}}T = \left\langle \frac{m}{2} \sum_{i=1}^N \mathbf{v}_i^2 \right\rangle .$$

(Strictly speaking, $3N$ should be replaced by $3(N - 1)$ which is the number of internal degrees of freedom.) Since the kinetic energy fluctuates, so does T . The relation between E and $\langle T \rangle$ is not possible to predict exactly in advance of the simulation because it involves the potential energy, i.e., the separations of all pairs of particles. A simple way to solve this problem is to choose some reasonable initial configuration. During the equilibration phase of the simulation, all of the particle velocities are scaled

$$\mathbf{v}_i(t) \longrightarrow \lambda \mathbf{v}_i(t) ,$$

where

$$\lambda = \sqrt{\frac{(N - 1)3Nk_{\text{B}}T_{\text{desired}}}{\sum_{i=1}^N \mathbf{v}_i^2}} .$$

- The function `initSystem` initializes the positions of the particles at the sites of a *face centered cubic* (fcc) lattice in a volume determined by the requested particle density ρ , and the initial velocity components to be Gaussian distributed with magnitude appropriate to the requested temperature T .

The functions defined in `md.c` and `md.h` can be called by simulation programs defined in other program files. For example, the file `md_anim.c` animates the particle system using OpenGL graphics.

Simple OpenGL graphics

The OpenGL library is useful for programming 3 dimensional graphics and animations. There are OpenGL functions to draw, color, and texturize points, lines, and surface, as well as bitmapped images. The OpenGL Utility Toolkit (GLUT) has functions manipulate windows on computer displays: these functions are very convenient because they are portable across operating systems.

The program `square.cpp` is a simple OpenGL taken from the OpenGL programming guide.

The main function calls several glut functions

```
int main (int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(250, 250);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Simple Glut Program");
    init(); // initialize 3-D scene
    glutDisplayFunc(display); // register callback function
    glutMainLoop();
}
```

}

- Initialize the window system after extracting any command line arguments which relate to the GUI (like `-geometry` on Unix).
- Set the display mode to a single frame buffer, using the RGB color model.
- Set the window size to 250×250 pixels.
- Set the window position.
- Create a window with this size and position with title “Simple Glut Program”.
- Register a *callback function* to be run whenever the window needs to be repainted.
- Enter an infinite loop which waits for user input.

The main function calls a function to initialize the *scene*:

```
void init () {  
    // clearing background color  
    glClearColor(0.0, 0.0, 0.0, 0.0);  
    // initialize viewing values  
    glMatrixMode(GL_PROJECTION);
```

```
    glLoadIdentity();  
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);  
}
```

- Set the background color to black $R = 0, G = 0, B = 0$ with *alpha value* = 0.
- Select the *projection transformation* matrix.
- Initialize it to the identity (unit matrix).
- Multiply it by an *orthographic parallel viewing volume* matrix. The arguments represent (*left, right, bottom, top, near, far*).

Finally, there is the display callback function:

```
void display () { // called whenever window needs refreshing  
    // clear all pixels  
    glClear(GL_COLOR_BUFFER_BIT);  
    // draw a white square  
    glColor3f(1.0, 1.0, 1.0);  
    glBegin(GL_POLYGON);  
        glVertex3f(0.25, 0.25, 0.0);  
        glVertex3f(0.75, 0.25, 0.0);  
        glVertex3f(0.75, 0.75, 0.0);  
        glVertex3f(0.25, 0.75, 0.0);  
    glEnd();  
    glFlush();  
}
```

```
    glEnd();  
    // start drawing it immediately  
    glFlush();  
}
```

- Clear the background, i.e., paint it black.
- Set the current color to white $R = 1, G = 1, B = 1$.
- Create a *polygon* by specifying 4 vertices.
- Flush the graphics commands in the connection between program memory and graphics hardware.

Simple animation with OpenGL

The program `cube.cpp` which is taken from a game developer tutorial shows how to animate a three-dimensional object, in this case a cube.