目录

摘要

《分子动力学模拟入门》第二章：盒子与近邻列表

本章介绍三斜（triclinic）盒子与近邻列表。其中，近邻列表的高效创建对分子动力学模拟程序的计算速度有关键作用，而近邻列表创建的细节又与模拟盒子密切相关。

1 三斜盒子

1.1 三斜盒子的定义

我们首先讲上一章的正交盒子进行推广。在正交盒子中，我们只有三个关于盒子的自由度，分别是三个盒子边长 L_x、L_y 和 L_z。对于这样的盒子，我们其实假设了 L_x 朝着 x 的方向，L_y 朝着 y 的方向，L_z 朝着 z 的方向。其实，一个盒子中共点的三条有向线段可以表示成矢量，可记为 $\vec{a}$、$\vec{b}$、$\vec{c}$。它们可以用分量表示为

$$\vec{a} = a_x\vec{e}_x + a_y\vec{e}_y + a_z\vec{e}_z; \tag{1}$$

$$\vec{b} = b_x\vec{e}_x + b_y\vec{e}_y + b_z\vec{e}_z; \tag{2}$$

$$\vec{c} = c_x\vec{e}_x + c_y\vec{e}_y + c_z\vec{e}_z. \tag{3}$$

这样具有 9 个自由度的盒子，叫做三斜盒子（triclinic box）。对于正交盒子，我们有

$$\vec{a} = L_x\vec{e}_x + 0\vec{e}_y + 0\vec{e}_z; \tag{4}$$

$$\vec{b} = 0\vec{e}_x + L_y\vec{e}_y + 0\vec{e}_z; \tag{5}$$

$$\vec{c} = 0\vec{e}_x + 0\vec{e}_y + L_z\vec{e}_z. \tag{6}$$

我们可以将三个矢量的一共 9 个分量组合成一个矩阵，称为盒子矩阵，记为

$$H = \begin{pmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ a_z & b_z & c_z \end{pmatrix}. \tag{7}$$

对于正交盒子，该矩阵为对角矩阵：

$$\begin{pmatrix} L_x & 0 & 0 \\ 0 & L_y & 0 \\ 0 & 0 & L_z \end{pmatrix}. \tag{8}$$

读者也许要问，盒子矩阵为什么不定义为上述定义的转置？实际上，这只是一个约定，也许只是在后面的计算中比较方便而已。读者可以思考如果在定义中加一个转置，后面的讨论会如何改变。

根据高等数学知识，我们知道三斜盒子的体积 V 等于上述矩阵行列式 $\det(H)$ 的绝对值：

$$V = |\det(H)|. \tag{9}$$

对于 3×3 的矩阵的行列式，有如下计算公式：

$$\det(H) = a_x(b_yc_z - c_yb_z) + b_x(c_ya_z - a_yc_z) + c_x(a_yb_z - b_ya_z). \tag{10}$$

据此可写出如下 C++ 函数：

```
1 double getDet(const double* box)
2 {
3   return box[0] * (box[4] * box[8] - box[5] * box[7]) +
4          box[1] * (box[5] * box[6] - box[3] * box[8]) +
5          box[2] * (box[3] * box[7] - box[4] * box[6]);
6 }
```

这里我们假设矩阵 H 中的 9 个元素是按行主序（row-major order）的次序存储在数组 `double box[18]` 的前 9 个元素中的。该数组有 18 个元素，这里只用了 9 个，剩下的 9 个元素将用于存放矩阵 H 的逆:

$$G = H^{-1}. \tag{11}$$

矩阵求逆的 C++ 函数为：

```
void getInverseBox(double* box)
{
  box[9] = box[4] * box[8] - box[5] * box[7];
  box[10] = box[2] * box[7] - box[1] * box[8];
  box[11] = box[1] * box[5] - box[2] * box[4];
  box[12] = box[5] * box[6] - box[3] * box[8];
  box[13] = box[0] * box[8] - box[2] * box[6];
  box[14] = box[2] * box[3] - box[0] * box[5];
  box[15] = box[3] * box[7] - box[4] * box[6];
  box[16] = box[1] * box[6] - box[0] * box[7];
  box[17] = box[0] * box[4] - box[1] * box[3];
  double det = getDet(box);
  for (int n = 9; n < 18; ++n) {
    box[n] /= det;
  }
}
```

1.2 三斜盒子情况下的周期边界条件

为了理解盒子矩阵的作用，我们假设将一个坐标 $\vec{r}$ 表示为：

$$\vec{r} = s_a\vec{a} + s_b\vec{b} + s_c\vec{c}. \tag{12}$$

可以看出，$(s_a, s_b, s_c) = (0,0,0), (1,0,0), (0,1,0), (1,1,0), (0,0,1), (1,0,1), (0,1,1), (1,1,1)$ 分别代表盒子的 8 个顶点。那么，如果要求 $0 \le s_a \le 1$, $0 \le s_b \le 1$, $0 \le s_c \le 1$，上述坐标就完全在盒子内（或者表面）。这里的坐标分量 (s_a, s_b, s_c) 称为“分数坐标”。上式可用矩阵表示如下

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ a_z & b_z & c_z \end{pmatrix} \begin{pmatrix} s_a \\ s_b \\ s_c \end{pmatrix}. \tag{13}$$

简写为：

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = H \begin{pmatrix} s_a \\ s_b \\ s_c \end{pmatrix}. \tag{14}$$

反过来，我们有

$$\begin{pmatrix} s_a \\ s_b \\ s_z \end{pmatrix} = G \begin{pmatrix} x \\ y \\ z \end{pmatrix}. \tag{15}$$

也就是说，盒子矩阵的逆矩阵 G 可以将一个盒子内的坐标变换为相对于盒子的分数坐标。所有的分数坐标形成一个边长为 1 的立方体。结合上一章关于正交盒子的最小镜像约定，我们可以得到如下的关于三斜盒子的最小镜像约定的算法：

1. 对于相对坐标 $\vec{r}_{ij} = (x_{ij}, y_{ij}, z_{ij})$，首先用盒子逆矩阵 G 将其变换为分数相对坐标 $\vec{s}_{ij} = (\xi_{ij}, \eta_{ij}, \zeta_{ij})$:

$$\begin{pmatrix} \xi_{ij} \\ \eta_{ij} \\ \zeta_{ij} \end{pmatrix} = G \begin{pmatrix} x_{ij} \\ y_{ij} \\ z_{ij} \end{pmatrix}. \tag{16}$$

2. 对分数相对坐标实施如下最小镜像约定操作。例如，当 $\xi_{ij} < -1/2$ 时，将其换位 $\xi_{ij} + 1$；当 $\xi_{ij} > 1/2$ 时，将其换位 $\xi_{ij} - 1$。

3. 将实施了最小镜像约定操作的分数相对坐标变换到普通相对坐标：

$$\begin{pmatrix} x_{ij} \\ y_{ij} \\ z_{ij} \end{pmatrix} = H \begin{pmatrix} \xi_{ij} \\ \eta_{ij} \\ \zeta_{ij} \end{pmatrix}. \tag{17}$$

该算法由如下两个函数实现：

```
1  void applyMicOne(double& x12)
2  {
3    if (x12 < -0.5)
4      x12 += 1.0;
5    else if (x12 > +0.5)
6      x12 -= 1.0;
7  }
8
9  void applyMic(const double* box, double& x12, double& y12, double& z12)
10 {
11   double sx12 = box[9] * x12 + box[10] * y12 + box[11] * z12;
12   double sy12 = box[12] * x12 + box[13] * y12 + box[14] * z12;
13   double sz12 = box[15] * x12 + box[16] * y12 + box[17] * z12;
```

```
14    applyMicOne(sx12);
15    applyMicOne(sy12);
16    applyMicOne(sz12);
17    x12 = box[0] * sx12 + box[1] * sy12 + box[2] * sz12;
18    y12 = box[3] * sx12 + box[4] * sy12 + box[5] * sz12;
19    z12 = box[6] * sx12 + box[7] * sy12 + box[8] * sz12;
20  }
```

2 近邻列表的创建

2.1 为什么要用近邻列表？

根据上一章的程序，我们知道，如果用一个自变量为原子数 N 的函数表示程序的计算量 y，那么该函数一定是：

$$y = c_0 + c_1 N + c_2 N^2. \tag{18}$$

其中，c_2N^2 是求能量和力的部分的，c_1N 主要对应运动方程积的部分，而 c_0 对应其它和 N 无关的部分。当 N 很大时，c_2N^2 占主导，我们说该程序的计算复杂度是 $\mathcal{O}(N^2)$，或者说具有平方标度。对于这样的算法，粒子数很大时计算量会变得很大，以至于难以承受。

我们注意到，在上一章的 findForce 函数中，即使两个原子离得很远，我们也要对它们的距离进行判断，决定是否考虑它们之间的相互作用力。如果我们事先将在某一距离范围内的原子对记录下来，并在 findForce 函数中查看记录，就可以在 findForce 函数中仅仅考虑在某一距离范围内的原子对。这个“记录”就是近邻列表（neighbor list）。这个距离叫做近邻列表的截断。它应该比是势函数的截断距离大一些，否则我们在每一步都需要先建立近邻列表，然后在 findForce 中使用。记势函数的截断距离为 R_c，近邻列表的截断距离为 R'_c，一般来说用 $R'_\mathrm{c} - R_\mathrm{c} = 1$ Å 是个不错的选择。当 $R'_\mathrm{c} > R_\mathrm{c}$ 时，一个构建好的近邻列表将在若干步内都是“安全”的，不需要每一步都更新。如果用下面将要介绍的平方标度算法，整个程序的计算复杂度依然是 $\mathcal{O}(N^2)$，但平方项的系数 c_2 将变小，从而有效地提升效率。如果用下面将要介绍的线性标度算法，整个程序的计算复杂度将变为 $\mathcal{O}(N)$，对于原子数很多的情形将具有很大的优势。我们稍后将给出具体的测试结果。

2.2 自动判断何时更新近邻列表

记 $R'_\mathrm{c} - R_\mathrm{c} = \delta$，我们可以构造如下的算法，实现近邻列表更新的自动判断：

1. 在程序的开头定义一套额外的坐标 $\{\vec{r}_i^0\}$，初始化为 0

$$\vec{r}_i^0 = \vec{0}. \tag{19}$$

2. 在积分过程的每一步，对每个原子 i 计算距离 $d_i = |\vec{r}_i - \vec{r}_i^0|$，然后计算出这些距离中的最大值 $d_{\max}$。若 $2d_{\max} > \delta$，则更新近邻列表，同时更新 $\{\vec{r}_i^0\}$：

$$\vec{r}_i^0 = \vec{r}_i. \tag{20}$$

这是一个非常保守的更新判据。读者可以思考更加好的方案（使得更新频率降低）。

2.3 构建近邻列表的平方标度算法

我们先讨论一个简单的平方标度算法。首先，我们定义近邻列表。一个近邻列表指定了研究体系中每个原子的近邻个数，即与某个中心原子距离小于 R_c' 的原子的个数。我们记原子 i 的近邻个数为 NN_i。除此以外，确定一个近邻列表还需要知道原子 i 的所有这 NN_i 个近邻的指标。我们记原子 i 的第 k 个邻居的指标为 NN_{ik}。因为我们在求力的时候将利用牛顿第三定律，所以在构建近邻列表时也要求一个原子的近邻的指标大于原子本身的指标，即 $i < \mathrm{NN}_{ik}$。这样定义的近邻列表最早由 Verlet 提出 [Verlet, 1967]，所以称为 Verlet 近邻列表。

一个很自然的构建近邻列表的算法是检验所有的粒子对的距离。这显然是一个 $O(N^2)$ 复杂度的算法。我们的 C++ 实现如下：

```
1  void findNeighborON2(Atom& atom)
2  {
3    const double cutoffSquare = atom.cutoffNeighbor * atom.cutoffNeighbor;
4    std::fill(atom.NN.begin(), atom.NN.end(), 0);
5
6    for (int i = 0; i < atom.number - 1; ++i) {
7      const double x1 = atom.x[i];
8      const double y1 = atom.y[i];
9      const double z1 = atom.z[i];
10     for (int j = i + 1; j < atom.number; ++j) {
11       double xij = atom.x[j] - x1;
12       double yij = atom.y[j] - y1;
13       double zij = atom.z[j] - z1;
14       applyMic(atom.box, xij, yij, zij);
15       const double distanceSquare = xij * xij + yij * yij + zij * zij;
```

```
16      if (distanceSquare < cutoffSquare) {
17        atom.NL[i * atom.MN + atom.NN[i]++] = j;
18        if (atom.NN[i] > atom.MN) {
19          std::cout << "Error: number of neighbors for atom " << i
20                    << " exceeds " << atom.MN << std::endl;
21          exit(1);
22        }
23      }
24    }
25  }
26 }
```

第 4 行调用 C++ 标准库的 `std::fill()` 函数将每个原子的近邻个数置零，为后面的累加做准备。第 17 行的语句等价于如下两句：

```
atom.NL[i * atom.MN + atom.NN[i]] = j;
atom.NN[i]++;
```

第 18-22 的语句对近邻列表的存储空间进行判断，如果任何原子的近邻个数超出了设定的最大值 `atom.MN`，就报告一个错误消息并退出程序。该最大值在该程序中设置为 1000，但读者可以适当地改动。这个数值设置得过大，就会浪费内存，设置得过小，就容易引发此处的错误。读者可以思考如下问题：如何修改程序，使得程序能够自动地调整相关数组的内存分配量，并总是让程序顺利地执行，而不会因为 `atom.MN` 设置得过小而退出程序，也不会因为 `atom.MN` 设置得过大而浪费太多的内存？

2.4 构建近邻列表的线性标度算法

上述简单的近邻列表算法已经能够加速程序了，但它还是一个 $O(N^2)$ 复杂度的算法。本节介绍一个 $O(N)$ 复杂度的算法，即所谓的线性标度算法。该算法的主要思想有以下两点：（1）整个模拟盒子被划分为一系列立方体小胞（cell），每个胞的边长不小于近邻列表的截断距离；（2）对于每个原子，只需要在 27 个小胞（一个是原子所在的胞，另外 26 个是与该胞紧挨着的）中寻找近邻。在分子动力学模拟中最早提出此类方法的可能是 Quentrec 和 Brot [Quentrec and Brot, 1973]。

先看第一点。首先，我们需要确定将整个体系划分为多少个小盒子。我们针对一般的三斜盒子来讨论。对于三斜盒子，我们需要计算三个厚度，然后除以近邻截断，并向下取整，得到每个盒子矢量方向的小盒子个数：

$$N_a = \lfloor (V/A_{bc})/R'_c \rfloor; \tag{21}$$

$$N_b = \lfloor (V/A_{ca})/R'_c \rfloor; \tag{22}$$

$$N_c = \lfloor (V/A_{ab})/R'_c \rfloor. \tag{23}$$

总的小盒子个数为 $N_{\text{cell}} = N_a N_b N_c$.

计算盒子厚度的 C++ 函数如下：

```
1  float getArea(const double* a, const double* b)
2  {
3    const double s1 = a[1] * b[2] - a[2] * b[1];
4    const double s2 = a[2] * b[0] - a[0] * b[2];
5    const double s3 = a[0] * b[1] - a[1] * b[0];
6    return sqrt(s1 * s1 + s2 * s2 + s3 * s3);
7  }
8
9  void getThickness(const Atom& atom, double* thickness)
10 {
11   double volume = abs(getDet(atom.box));
12   const double a[3] = {atom.box[0], atom.box[3], atom.box[6]};
13   const double b[3] = {atom.box[1], atom.box[4], atom.box[7]};
14   const double c[3] = {atom.box[2], atom.box[5], atom.box[8]};
15   thickness[0] = volume / getArea(b, c);
16   thickness[1] = volume / getArea(c, a);
17   thickness[2] = volume / getArea(a, b);
18 }
```

第 11 行计算盒子的体积。第 12-14 行获得三个盒子矢量 $\vec{a}$、$\vec{b}$、$\vec{c}$。第 15-17 行用体积除以面积得到对应的厚度。由矢量 $\vec{a}$ 和 $\vec{b}$ 构成的四边形的面积为 $|\vec{a}\times\vec{b}|$，如第 3-6 行所示。

在确定小盒子的个数之后，我们就可以确定每个原子处于哪个小盒子了。对应的 C++ 函数如下：

```
1  void findCell(
2    const double* box,
3    const double* thickness,
4    const double* r,
5    double cutoffInverse,
6    const int* numCells,
7    int* cell)
```

```
8  {
9    double s[3];
10   s[0] = box[9] * r[0] + box[10] * r[1] + box[11] * r[2];
11   s[1] = box[12] * r[0] + box[13] * r[1] + box[14] * r[2];
12   s[2] = box[15] * r[0] + box[16] * r[1] + box[17] * r[2];
13   for (int d = 0; d < 3; ++d) {
14     cell[d] = floor(s[d] * thickness[d] * cutoffInverse);
15     if (cell[d] < 0)
16       cell[d] += numCells[d];
17     if (cell[d] >= numCells[d])
18       cell[d] -= numCells[d];
19   }
20   cell[3] = cell[0] + numCells[0] * (cell[1] + numCells[1] * cell[2]);
21 }
```

第 9-12 行计算与原子坐标 $\vec{r}$ 对应的分数坐标 $\vec{s}$。第 14 行计算原子在某个盒子矢量方向的小盒子指标。第 15-18 行保证计算的小盒子指标不超出界限。最后，第 20 行根据三维的小盒子指标计算一个一维指标。

有了以上准备之后，我们考虑第二点，即在 27 个小盒子内寻找一个原子的近邻。实现该部分的 C++ 函数如下。第 5-6 行计算三个盒子厚度。第 8-12 行计算每个盒子矢量方向的小盒子个数。第 17 行定义一个数组，代表每个小盒子中的原子个数 $C_n(0 \leq n \leq N_{\text{cell}}-1)$，起计算过程在第 20-24 行。第 18 行定义一个数组 $S_n(0 \leq n \leq N_{\text{cell}}-1)$，其元素定义为：

$$S_0 = 0; \tag{24}$$

$$S_n = \sum_{m=0}^{n-1} C_m \quad (1 \leq n \leq N_{\text{cell}}-1). \tag{25}$$

用计算机的术语来说，数组 S_n 是数组 C_n 的前缀和（prefix sum），其计算过程在第 26-28 行。第 29 行将数组 C_n 的元素置零，因为后面又要对它进行累加。第 32-39 行计算根据小盒子指标排列的原子指标，保存为一个长度为原子数 N 的数组 I。其中，从 I_{S_n} 到 $I_{S_n+C_n-1}$ 的元素就是处于小盒子 n 的原子的指标。第 41-85 行根据以上几个辅助数组的信息构建 Verlet 近邻列表，其算法和前面的 $\mathcal{O}(N)$ 算法差不多，只不过我们只需要考虑 27 个小盒子，而不是整个大盒子。

```
1  void findNeighborON1(Atom& atom)
2  {
```

```
3  const double cutoffInverse = 1.0 / atom.cutoffNeighbor;
4  double cutoffSquare = atom.cutoffNeighbor * atom.cutoffNeighbor;
5  double thickness[3];
6  getThickness(atom, thickness);
7
8  int numCells[4];
9
10 for (int d = 0; d < 3; ++d) {
11   numCells[d] = floor(thickness[d] * cutoffInverse);
12 }
13
14 numCells[3] = numCells[0] * numCells[1] * numCells[2];
15 int cell[4];
16
17 std::vector<int> cellCount(numCells[3], 0);
18 std::vector<int> cellCountSum(numCells[3], 0);
19
20 for (int n = 0; n < atom.number; ++n) {
21   const double r[3] = {atom.x[n], atom.y[n], atom.z[n]};
22   findCell(atom.box, thickness, r, cutoffInverse, numCells, cell);
23   ++cellCount[cell[3]];
24 }
25
26 for (int i = 1; i < numCells[3]; ++i) {
27   cellCountSum[i] = cellCountSum[i - 1] + cellCount[i - 1];
28 }
29
30 std::fill(cellCount.begin(), cellCount.end(), 0);
31
32 std::vector<int> cellContents(atom.number, 0);
33
34 for (int n = 0; n < atom.number; ++n) {
35   const double r[3] = {atom.x[n], atom.y[n], atom.z[n]};
36   findCell(atom.box, thickness, r, cutoffInverse, numCells, cell);
37   cellContents[cellCountSum[cell[3]] + cellCount[cell[3]]] = n;
38   ++cellCount[cell[3]];
39 }
40
41 std::fill(atom.NN.begin(), atom.NN.end(), 0);
```

```
42
43  for (int n1 = 0; n1 < atom.number; ++n1) {
44    const double r1[3] = {atom.x[n1], atom.y[n1], atom.z[n1]};
45    findCell(atom.box, thickness, r1, cutoffInverse, numCells, cell);
46    for (int k = -1; k <= 1; ++k) {
47      for (int j = -1; j <= 1; ++j) {
48        for (int i = -1; i <= 1; ++i) {
49          int neighborCell = cell[3] + (k * numCells[1] + j) * numCells[0] + i;
50          if (cell[0] + i < 0)
51            neighborCell += numCells[0];
52          if (cell[0] + i >= numCells[0])
53            neighborCell -= numCells[0];
54          if (cell[1] + j < 0)
55            neighborCell += numCells[1] * numCells[0];
56          if (cell[1] + j >= numCells[1])
57            neighborCell -= numCells[1] * numCells[0];
58          if (cell[2] + k < 0)
59            neighborCell += numCells[3];
60          if (cell[2] + k >= numCells[2])
61            neighborCell -= numCells[3];
62
63          for (int m = 0; m < cellCount[neighborCell]; ++m) {
64            const int n2 = cellContents[cellCountSum[neighborCell] + m];
65            if (n1 < n2) {
66              double x12 = atom.x[n2] - r1[0];
67              double y12 = atom.y[n2] - r1[1];
68              double z12 = atom.z[n2] - r1[2];
69              applyMic(atom.box, x12, y12, z12);
70              const double d2 = x12 * x12 + y12 * y12 + z12 * z12;
71              if (d2 < cutoffSquare) {
72                atom.NL[n1 * atom.MN + atom.NN[n1]++] = n2;
73                if (atom.NN[n1] > atom.MN) {
74                  std::cout << "Error: number of neighbors for atom " << n1
75                            << " exceeds " << atom.MN << std::endl;
76                  exit(1);
77                }
78              }
79            }
80          }
```

```
81          }
82        }
83      }
84    }
85  }
```

2.5 程序速度测试

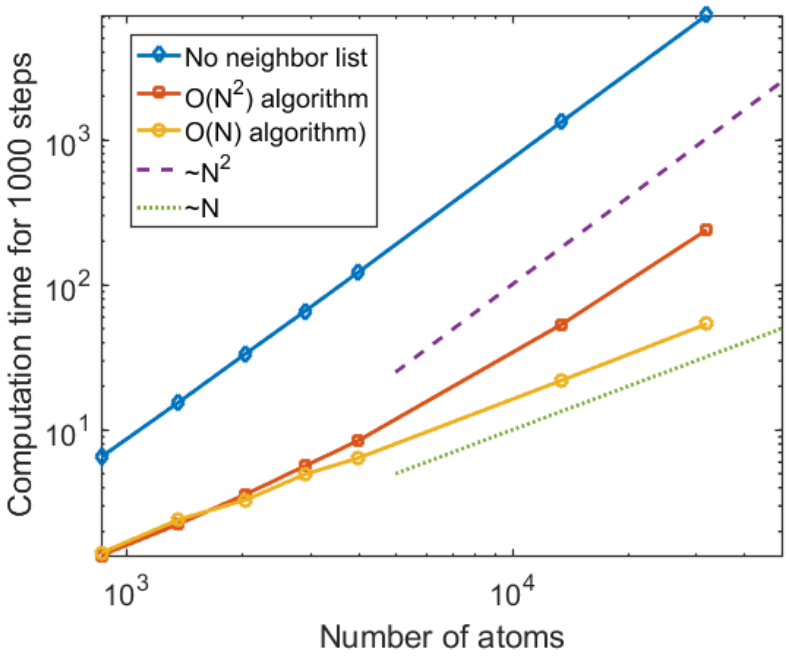

图 1: 不同近邻列表方案下的程序速度对比。

用本章的程序 md2.cpp 进行测试，得到如图 1 的结果。此图对比了三种近邻列表方案下程序跑 1000 步所花的时间和体系原子数的关系。这里的三种近邻列表方案分别是：(1) 不使用近邻列表 (类似于上一章的程序，但注意我们从本章开始使用了三斜的盒子)；(2) 使用 $\mathcal{O}(N^2)$ 计算复杂度的近邻列表构建方法；(3) 使用 $\mathcal{O}(N)$ 计算复杂度的近邻列表构建方法。该图的结果是符合预期了，它展示了如下特征：

- 不使用近邻列表时，程序的计算量正比于原子数的平方，拥有典型的 $\mathcal{O}(N^2)$ 计算复杂度。

- 使用 $\mathcal{O}(N^2)$ 计算复杂度的近邻列表构建方法时，程序的计算量在小体系的极限下正比于原子数，但在大体系的极限下正比于原子数平方。
- 使用 $\mathcal{O}(N)$ 计算复杂度的近邻列表构建方法时，程序的计算量始终正比于原子数，具有 $\mathcal{O}(N)$ 复杂度。

所以，在研究较大体系时，一定要采用 $\mathcal{O}(N)$ 计算复杂度的近邻列表构建方法。

参考文献

[Quentrec and Brot, 1973] Quentrec, B. and Brot, C. (1973). New method for searching for neighbors in molecular dynamics computations. *Journal of Computational Physics*, 13(3):430–432.

[Verlet, 1967] Verlet, L. (1967). Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Phys. Rev.*, 159:98–103.