目录

摘要

《分子动力学模拟入门》第一章：从一个简单的分子动力学模拟程序开始

本章将从一个简单的分子动力学模拟程序开始，带领读者走进分子动力学模拟的世界。在以后的章节，我们将逐步深入地探讨分子动力学模拟的若干重要课题。

1 简单分子动力学模拟的基本要素

1.1 分子动力学模拟的定义

作者曾经在一篇博文给分子动力学模拟下过一个定义：

分子动力学模拟是一种数值计算方法，在这种方法中，我们对一个具有一定初始条件和边界条件且具有相互作用的多粒子系统的运动方程进行数值积分，得到系统在相空间中的一条离散的轨迹，并用统计力学的方法从这条相轨迹中提取出有用的物理结果。

在本章余下的部分，我们会一一考察上述定义中的重要概念，如初始条件、边界条件、相互作用、运动方程、数值积分等。这里，我们首先讨论上述定义中的“统计力学的方法”。

在一个特定的平衡态统计系综中，一个物理量 A 的统计平均值可表达为

$$\langle A\rangle_{\text{ensemble}} = \int f(p,q)A(p,q)dpdq \tag{1}$$

其中，$f(p,q)$ 是该系综的分布函数，q 和 p 代表广义坐标和广义动量的集合。这样的统计平均称为系综平均。然而，根据我们的定义，在分子动力学模拟中并没有使用“系综”（即系统的集合），而是仅对一个系统的时间演化过程进行分析。那么，上述定义中的“统计力学的方法”指的是什么呢?

这里的“统计力学的方法”指的是时间平均，即

$$\langle A\rangle_{\text{time}} = \lim_{t\to\infty}\frac{1}{t}\int_0^t A(t')dt' \tag{2}$$

我们知道，随着时间的推移，系统将历经一条相轨迹。因为这条相轨迹永不与自身相交，如果它不代表一个周期性运动的话，那么随着时间的增加，这条相轨迹应该历经越来越多的相点。一个自然的假设是，当时间趋近于无穷大时，这条相轨迹将遍历系统的所有相点。这就是各态历经假设。在此假设下，系综平均与时间平均等价。

从实验的角度来说，对热力学体系的实验测量采用了时间平均而非系综平均。所以，分子动力学模拟实际上比基于系综的统计力学更加贴近实验。从计算机模拟的角度来看，除了采用时间平均的分子动力学模拟，还有直接采用系综平均的蒙特卡罗模拟，但本书不讨论蒙特卡罗模拟。

根据上述定义，我们可以设想一个典型的、简单的分子动力学模拟有如下大致的计算流程：

1. 初始化。设置系统的初始条件，具体包括各个粒子的位置矢量和速度矢量。

2. 时间演化。根据系统中的粒子所满足的相互作用规律，由牛顿定律确定所有粒子的运动方程（二阶常微分方程组），并对运动方程进行数值积分，即不断地更新每个粒子

的坐标和速度。最终，我们将得到一系列离散的时刻系统在相空间中的位置，即一条离散的相轨迹。

3. 测量。用统计力学的方法分析相轨迹所蕴含的物理规律。

1.2 初始条件

初始化指的是确定一个初始的相空间点，包括各个粒子初始的坐标和速度。在分子动力学模拟中，我们需要对 $3N$（N 是粒子数目）个二阶常微分方程进行数值积分。因为每一个二阶常微分方程的求解都需要有两个初始条件，所以我们需要确定 $6N$ 个初始条件：$3N$ 个初始坐标分量和 $3N$ 个初始速度分量。

1.2.1 坐标初始化

坐标的初始化指的是为系统中的每个粒子选定一个初始的位置坐标。分子动力学模拟中如何初始化位置主要取决于所要模拟的体系。例如，如果要模拟固态氩，就得让各个氩原子的位置按面心立方结构排列。如果要模拟的是液态或者气态物质，那么初始坐标的选取就可以比较随意了。重要的是，在构造的初始结构中，任何两个粒子的距离都不能太小，因为这可能导致有些粒子受到非常大的力，以至于让后面的数值积分变得非常不稳定。坐标的初始化也常被称为建模，往往需要用到一些专业的知识，例如固体物理学（晶体学）中的知识。本章将通过一个程序介绍固态氩的建模。

1.2.2 速度初始化

我们知道，任何经典热力学系统在平衡时各个粒子的速度满足麦克斯韦分布。然而，作为初始条件，我们并不一定要求粒子的速度满足麦克斯韦分布。最简单的速度初始化方法是产生 $3N$ 个在某个区间均匀分布的随机速度分量，再通过如下几个基本条件对速度分量进行修正。

第一个条件是让系统的总动量为零。也就是说，我们不希望系统的质心在模拟的过程中跑动。分子间作用力是所谓的内力，不会改变系统的整体动量，即系统的整体动量是守恒的。只要初始的整体动量为零，在分子动力学模拟的时间演化过程中整体动量将保持为零。如果整体动量明显偏离零（相对于所用浮点数精度来说），则说明模拟出了问题。这正是判断程序是否有误的标准之一。

第二个条件是系统的总动能应该与所选定的初始温度对应。我们知道，在经典统计力学中，能量均分定理成立，即粒子的哈密顿量中每一个具有平方形式的能量项的统计平均值都等于 $k_{\mathrm{B}}T/2$。其中，k_{B} 是玻尔兹曼常数，T 是系统的绝对温度。所以，在将质心的动

量取为零之后就可以对每个粒子的速度进行一个标度变换，使得系统的初始温度与所设定的温度一致。假设我们设置的目标温度是 T_0，那么对各个粒子的速度做如下变换即可让系统的温度从 T 变成 T_0：

$$\vec{v}_i \to \vec{v}_i' = \vec{v}_i \sqrt{\frac{T_0}{T}} \tag{3}$$

容易验证（习题），在做上式中的变换之前，如果系统的总动量已经为零，那么在做这个变换之后，系统的总动量也为零。

第三个可选的条件是角动量的初始化。我们将在一个习题中研究这个问题。

1.3 边界条件

在我们对分子动力学模拟的定义中，除了初始条件，还提到了边界条件。边界条件对常微分方程的求解并不是必要的，但在分子动力学模拟中通常会根据所模拟的物理体系选取合适的边界条件，以期得到更合理的结果。边界条件的选取对粒子间作用力的计算也是有影响的。常用的边界条件有好几种，但我们这里只先讨论其中的一种：周期边界条件。在计算机模拟中，模拟的系统尺寸一定是有限的，通常比实验中对应的体系的尺寸小很多。选取周期边界条件通常可以让模拟的体系更加接近于实际的情形，因为原本有边界的系统在应用了周期边界条件之后，“似乎”没有边界了。当然，并不能说应用了周期边界条件的系统就等价于无限大的系统，只能说周期边界条件的应用可以部分地消除边界效应，让所模拟系统的性质更加接近于无限大系统的性质。通常，在这种情况下，我们要模拟几个不同大小的系统，分析所得结果对模拟尺寸的依赖关系。

在计算两个粒子，如粒子 i 和粒子 j 的距离时，就要考虑周期边界条件带来的影响。举个一维的例子。假设模拟在一个长度为 L_x 的模拟盒子中进行，采用周期边界条件时，可以将该一维的盒子想象为一个圆圈。假设 $L_x = 10$（任意单位），第 i 个粒子的坐标 $x_i = 1$，第 j 个粒子的坐标 $x_j = 8$，则这两个粒子的距离是多少呢？如果忽略周期边界条件，那么答案是 $|x_j - x_i| = 7$，而且 j 粒子在 i 粒子的右边（坐标值大的一边）。但是，在采取周期边界条件时，也可认为 j 粒子在 i 粒子的左边，且坐标值可以平移至 $8 - 10 = -2$。这样，j 与 i 的距离是 $|x_j - x_i| = 3$，比平移 j 粒子之前两个粒子之间的距离要小。在我们的模拟中，总是采用最小镜像约定：在计算两个粒子的距离时，总是取最小的可能值。定义

$$x_j - x_i \equiv x_{ij} \tag{4}$$

则这个约定等价于如下规则：如果 $x_{ij} < -L_x/2$，则将 x_{ij} 换为 $x_{ij} + L_x$；如果 $x_{ij} > +L_x/2$，则将 x_{ij} 换为 $x_{ij} - L_x$。最终效果就是让变换后的 x_{ij} 的绝对值不大于 $L_x/2$。

很容易将上述讨论推广到二维和三维的情形。例如，在二维的情形中，可以将一个周期的模拟盒子想象为一个环面，就像一个甜甜圈或一个充了气的轮胎的表面。在三维的情

形中，可以将一个周期的模拟盒子想象为一个三维环面，而最小镜像约定可以表达为：

- 如果 $x_{ij} < -L_x/2$，则将 x_{ij} 换为 $x_{ij}+L_x$；如果 $x_{ij} > +L_x/2$，则将 x_{ij} 换为 $x_{ij}-L_x$。
- 如果 $y_{ij} < -L_y/2$，则将 y_{ij} 换为 $y_{ij}+L_y$；如果 $y_{ij} > +L_y/2$，则将 y_{ij} 换为 $y_{ij}-L_y$。
- 如果 $z_{ij} < -L_z/2$，则将 z_{ij} 换为 $z_{ij}+L_z$；如果 $z_{ij} > +L_z/2$，则将 z_{ij} 换为 $z_{ij}-L_z$

这里，我们假设了三维模拟盒子中 3 个共点的边的长度分别为 L_x、L_y 和 L_z，且两两相互垂直（所谓的正交模拟盒子）。如果有任意两个共点的边不是相互垂直的，情况就要复杂一些。本章仅讨论正交盒子的情形，以后再讨论非正交盒子的情形。

1.4 相互作用

宏观物质的性质在很大程度上是由微观粒子之间的相互作用力决定的。所以，对粒子间相互作用力的计算在分子动力学模拟中是至关重要的。粒子间有何种相互作用不是分子动力学模拟本身所能回答的；它本质上是一个量子力学的问题。在经典分子动力学模拟中，粒子间的相互作用力常常由一个或多个经验势函数描述。经验势函数能够在某种程度上反映出某些物质的某些性质。近年来，机器学习也广泛地用于构造更加准确的势函数。在本章，我们只介绍一个称为 Lennard-Jones 势的简单势函数（简称为 LJ 势）。在以后的章节中，我们将介绍更多的经验势函数以及机器学习势函数。

考虑系统中的任意粒子对 i 和 j，它们之间的相互作用势能可以写为

$$U_{ij}(r_{ij}) = 4\epsilon\left(\frac{\sigma^{12}}{r_{ij}^{12}} - \frac{\sigma^{6}}{r_{ij}^{6}}\right). \tag{5}$$

其中，ϵ 和 σ 是势函数中的参数，分别具有能量和长度的量纲；$r_{ij} = |\vec{r}_j - \vec{r}_i|$ 是两个粒子间的距离。

LJ 势比较适合描述惰性元素组成的物质。它是最早提出的两体势函数之一。所谓两体势，指的是两个粒子 i 和 j 之间的相互作用势仅依赖于它们之间的距离 r_{ij}，不依赖于系统中其他粒子的存在与否及具体位置。本章只讨论两体势，后续的章节会讨论一些多体势，即非两体势。对于两体势函数，我们可以将整个系统的总势能 U 写为

$$U = \sum_{i=1}^{N} U_i; \tag{6}$$

$$U_i = \frac{1}{2}\sum_{j\neq i} U_{ij}(r_{ij}). \tag{7}$$

将以上两式合起来，可以写成

$$U = \frac{1}{2}\sum_{i=1}^{N}\sum_{j\neq i} U_{ij}(r_{ij}) \tag{8}$$

上面的 U_i 可以称为粒子 i 的势能。也可以将总势能写为如下形式：

$$U = \sum_{i=1}^{N}\sum_{j>i} U_{ij}(r_{ij}) \tag{9}$$

根据力的定义可得（习题），粒子 i 所受总的力为

$$\vec{F}_i = \sum_{j\neq i} \vec{F}_{ij} \tag{10}$$

$$\vec{F}_{ij} = \frac{\partial U_{ij}(r_{ij})}{\partial r_{ij}}\frac{\vec{r}_{ij}}{r_{ij}} \tag{11}$$

其中，我们定义了一个表示粒子间相对位置的符号

$$\vec{r}_{ij} \equiv \vec{r}_j - \vec{r}_i \tag{12}$$

显然，牛顿第三定律成立：

$$\vec{F}_{ij} = -\vec{F}_{ji}. \tag{13}$$

进一步推导可得 LJ 势中力的明确表达式：

$$\vec{F}_{ij} = \left(\frac{24\epsilon\sigma^6}{r_{ij}^8} - \frac{48\epsilon\sigma^{12}}{r_{ij}^{14}}\right)\vec{r}_{ij}. \tag{14}$$

通常，为了节约计算，我们会对势函数进行一个截断，即认为当两个原子之间的距离大于某个截断距离 R_c 时，它们之间的相互作用势能和力都是零：

$$U_{ij}(r_{ij}) = 0 \quad (r_{ij} > R_\mathrm{c}) \tag{15}$$

$$\vec{F}_{ij} = \vec{0} \quad (r_{ij} > R_\mathrm{c}) \tag{16}$$

1.5 运动方程的数值积分

我们知道在经典力学中，粒子的运动方程可以用牛顿第二定律表达。例如，对于第 i 个粒子，其运动方程为

$$m_i\frac{d\vec{r}_i^2}{dt^2} = \vec{F}_i \tag{17}$$

这是一个二阶常微分方程，我们可以把它改写为两个一阶常微分方程：

$$\frac{d\vec{r}_i}{dt} = \vec{v}_i \tag{18}$$

$$\frac{d\vec{v}_i}{dt} = \frac{\vec{F}_i}{m_i} \tag{19}$$

对运动方程进行数值积分的目的就是在给定的初始条件下找到各个粒子在一系列离散的时间点的坐标和速度值。我们假设每两个离散的时间点之间的间隔是固定的，记为 Δt，称为时间步长。在分子动力学模拟中使用的数值积分方法有很多种，本章只介绍所谓的“速度-Verlet”积分方法，其推导过程见习题。在该方法中，第 i 个粒子在时刻 $t+\Delta t$ 的速度 $\vec{v}_i(t+\Delta t)$ 和位置 $\vec{r}_i(t+\Delta t)$ 分别由以下两式给出：

$$\vec{v}_i(t+\Delta t) = \vec{v}_i(t) + \frac{1}{2}\frac{\vec{F}_i(t)+\vec{F}_i(t+\Delta t)}{m_i}\Delta t \tag{20}$$

$$\vec{r}_i(t+\Delta t) = \vec{r}_i(t) + \vec{v}_i(t)\Delta t + \frac{1}{2}\frac{\vec{F}_i(t)}{m_i}(\Delta t)^2 \tag{21}$$

由以上两式可以看出，$t+\Delta t$ 时刻的坐标仅依赖于 t 时刻的坐标、速度和力，但 $t+\Delta t$ 时刻的速度依赖于 t 时刻的速度、力及 $t+\Delta t$ 时刻的力。所以，从算法的角度来说，以上两式应该对应如下的计算流程：

第一步：部分地更新速度并完全地更新坐标（注意，我们引入了一个中间的速度变量 $\vec{v}_i(t+\Delta t/2)$）：

$$\vec{v}_i(t) \to \vec{v}_i(t+\Delta t/2) = \vec{v}_i(t) + \frac{1}{2}\frac{\vec{F}_i(t)}{m_i}\Delta t \tag{22}$$

$$\vec{r}_i(t) \to \vec{r}_i(t+\Delta t) = \vec{r}_i(t) + \vec{v}_i(t+\Delta t/2)\Delta t \tag{23}$$

第二步：用更新后的坐标计算新的力

$$\vec{F}_i(t) \to \vec{F}_i(t+\Delta t) \tag{24}$$

第三步：用更新后的力完成速度的更新：

$$\vec{v}_i(t+\Delta t/2) \to \vec{v}_i(t+\Delta t) = \vec{v}_i(t+\Delta t/2) + \frac{1}{2}\frac{\vec{F}_i(t+\Delta t)}{m_i}\Delta t \tag{25}$$

完成上述计算之后，粒子的坐标、速度、和力都从 t 时刻的更新为 $t+\Delta t$ 时刻的。这就是一个时间步的计算。反复执行这样的计算流程，系统的微观状态就会不断地随时间变化，从而得到一条相空间的轨迹。系统所有的宏观性质都包含在相轨迹中。

2 用 C++ 开发一个简单的分子动力学模拟程序

本节给出一个简单的用 C++ 开发的分子动力学模拟程序。我们选择用 C++ 语言开发程序，因为这是作者最熟悉的编程语言。因为分子动力学模拟一般较为耗时，所以用高效的编译型语言开发较为合适。除了 C++ 语言，C 语言和 Fortran 语言也很高效。以后我们还将介绍分子动力学模的 CUDA 编程实现。另外，在完成分子动力学模拟的计算后，我们往往需要对得到的数据进行可视化，此时使用一门解释性语言更为方便。作者使用 Matlab 语言进行数据的后处理和可视化。

2.1 程序中使用的单位制

我们的分子动力学模拟程序只涉及经典力学和热力学，故只需要用到 4 个基本物理量的单位。我们选择如下 4 个基本单位来确定各个物理量的数值：

1. 能量：电子伏特（记号为 eV），约为 1.6×10^{-19} J。
2. 长度：埃（angstrom，记号为 A），即 10^{-10} m。
3. 质量：原子质量单位（atomic mass unit，记号为 amu），约为 1.66×10^{-27} kg。
4. 温度：开尔文（记号为 K）。

用这样的基本单位，可使程序中大部分物理量的数值都接近 1。我们称这样的单位为该程序的“自然单位”。

从以上基本单位可以推导出程序中其他相关物理量的单位：

1. 力。因为力乘以距离等于功（能量），故力的单位是能量单位除以长度单位，即 eV A^{-1}。
2. 速度。因为动能正比于质量乘以速度的平方，故速度的单位是能量单位除以质量单位再开根号，即 $\mathrm{eV}^{1/2}$ $\mathrm{amu}^{-1/2}$。
3. 时间。因为长度等于速度乘以时间，故时间的单位是长度单位除以速度单位，即 A $\mathrm{amu}^{1/2}$ $\mathrm{eV}^{-1/2}$，约为 1.018051×10^{1} fs（fs 指飞秒，即 10^{-15} s）。
4. 玻尔兹曼常数 k_{B}。这是一个很重要的常数，它在国际单位制中约为 1.38×10^{-23} J/K，对应于程序自然单位制的 8.617343×10^{-5} eV/K。

2.2 本章程序的源代码解析

本章的程序很简单，一共只有 200 行代码，故将所有代码写在一个源文件 `ljmd.cpp`。下面，我们详细地讲解该程序。

2.2.1 主函数

我们从该文件的主函数 `main()` 看起。下面是主函数的全部代码：

```
1  int main(int argc, char** argv)
2  {
3    if (argc != 4) {
4      printf("usage: %s numSteps temperature timeStep\n", argv[0]);
5      exit(1);
6    }
7    const int numSteps = atoi(argv[1]);
8    const double temperature = atof(argv[2]);
9    double timeStep = atof(argv[3]);
10   timeStep /= TIME_UNIT_CONVERSION; // from fs to natural unit
11
12   Atom atom;
13   initializePosition(atom);
14   initializeVelocity(temperature, atom);
15
16   const clock_t tStart = clock();
17   std::ofstream ofile("energy.txt");
18   for (int step = 0; step < numSteps; ++step) {
19     integrate(true, timeStep, atom); // step 1 in the book
20     findForce(atom);                 // step 2 in the book
21     integrate(false, timeStep, atom); // step 3 in the book
22     if (step < numSteps / 2) {     // equlibration
23       scaleVelocity(temperature, atom); // control temperature
24     } else if (step % Ns == 0) {   // production
25       ofile << findKineticEnergy(atom) << " "
26             << std::accumulate(atom.pe.begin(), atom.pe.end(), 0.0)
```

```
27            << std::endl;
28      }
29    }
30    ofile.close();
31    const clock_t tStop = clock();
32    const float tElapsed = float(tStop - tStart) / CLOCKS_PER_SEC;
33    std::cout << "Time used = " << tElapsed << " s" << std::endl;
34
35    return 0;
36  }
```

首先，程序从命令行读入三个参数，分别是整个分子动力学模拟的步数（numSteps）、体系的目标温度（temperature）和数值积分的时间步长（timeStep）。在读入时间步长后，立刻将其单位从输入的 fs 转换至程序的自然单位。这种单位转换只需要在处理输入和输出时实施，在程序的其他地方任何物理量的单位都将是我们定义的自然单位。

接着，程序定义了一个结构体 Atom 的变量 atom。该结构体类型中定义了程序中用到的大部分数据。具体定义如下：

```
1  struct Atom {
2    int number; // 总的粒子数（原子数）
3    double box[6]; // 前三个数是三个盒子长度；后三个数是盒子长度的一半
4    // 质量、坐标、速度、力、势能：
5    std::vector<double> mass, x, y, z, vx, vy, vz, fx, fy, fz, pe;
6  };
```

我们这里用了 C++ 标准模板库中的 std::vector 来表示一些数组。

在定义 atom 后，我们调用函数 initializePosition() 读取一个名为 xyz.in 的文件，初始化体系的坐标（具体内容见后）等，然后调用函数 initializeVelocity() 初始化体系的速度。

接下来，做一个次数为 numSteps 的循环进行时间演化。在该循环中，我们实现前面讲述的速度-Verlet 积分算法，一共三个步骤：

- 语句 integrate(true, timeStep, atom); 实现速度的部分更新和坐标的完全更新。

- 语句 `findForce(atom);` 实现力的更新。
- 语句 `integrate(false, timeStep, atom);` 实现剩下的速度更新

前一半和后一半的步数分别对应于平衡阶段和产出阶段。在平衡阶段，我们每一步都对各个粒子的速度进行一个标度变换，从而控制体系的温度。在产出阶段，我们不再控制体系的温度，但每 `Ns=100` 步计算一次体系的总动能和总势能并输出到文件 `energy.txt`。

程序将对演化过程计时，在结束程序之前报道演化过程所花的总时间。

2.2.2 输入处理

程序将从一个名为 `xyz.in` 的文件读取粒子的坐标和其它相关信息。该功能由如下函数实现：

```
1  void initializePosition(Atom& atom)
2  {
3    std::ifstream ifile("xyz.in");
4    ifile >> atom.number;
5    atom.mass.resize(atom.number, 40.0); // argon mass
6    atom.x.resize(atom.number, 0.0);
7    atom.y.resize(atom.number, 0.0);
8    atom.z.resize(atom.number, 0.0);
9    atom.vx.resize(atom.number, 0.0);
10   atom.vy.resize(atom.number, 0.0);
11   atom.vz.resize(atom.number, 0.0);
12   atom.fx.resize(atom.number, 0.0);
13   atom.fy.resize(atom.number, 0.0);
14   atom.fz.resize(atom.number, 0.0);
15   atom.pe.resize(atom.number, 0.0);
16   ifile >> atom.box[0] >> atom.box[1] >> atom.box[2];
17   atom.box[3] = atom.box[0] * 0.5;
18   atom.box[4] = atom.box[1] * 0.5;
19   atom.box[5] = atom.box[2] * 0.5;
20   for (int n = 0; n < atom.number; ++n) {
21     ifile >> atom.x[n] >> atom.y[n] >> atom.z[n];
```

```
  }
  ifile.close();
}
```

在打开 xyz.in 文件后，首先读入粒子数 atom.number，然后将相关数组分配内存并初始化。其中，对于质量数组，每个元素初始化为氩原子的质量（40 amu），对其他数组，每个元素初始化为零。接着，读入模拟盒子在三个方向的长度，然后计算盒子长度的一半（在实施最小镜像约定时将用到）。最后，读入各个粒子的坐标，并关闭文件。

2.2.3 速度初始化

下面是速度初始化的函数 initializeVelocity()。在该函数中，首先利用随机数获得分布在 -1 到 1 之间的随机速度分量，同时计算体系的平均动量。注意，函数 rand() 返回一个从零到 RAND_MAX 之间的整数。接着，对速度进行修正，使得体系的整体动量为零。最后，调用 scaleVelocity() 函数对速度进行标度变换，使得体系的温度达到目标值 T0。

```
void initializeVelocity(const double T0, Atom& atom)
{
  double momentumAverage[3] = {0.0, 0.0, 0.0};
  for (int n = 0; n < atom.number; ++n) {
    atom.vx[n] = -1.0 + (rand() * 2.0) / RAND_MAX;
    atom.vy[n] = -1.0 + (rand() * 2.0) / RAND_MAX;
    atom.vz[n] = -1.0 + (rand() * 2.0) / RAND_MAX;

    momentumAverage[0] += atom.mass[n] * atom.vx[n] / atom.number;
    momentumAverage[1] += atom.mass[n] * atom.vy[n] / atom.number;
    momentumAverage[2] += atom.mass[n] * atom.vz[n] / atom.number;
  }
  for (int n = 0; n < atom.number; ++n) {
    atom.vx[n] -= momentumAverage[0] / atom.mass[n];
    atom.vy[n] -= momentumAverage[1] / atom.mass[n];
    atom.vz[n] -= momentumAverage[2] / atom.mass[n];
  }
```

```
18   scaleVelocity(T0, atom);
19 }
```

下面是函数 scaleVelocity() 的定义。在该函数中，首先调用函数 findKineticEnergy() 根据当前的速度计算当前的动能，进而得到当前的温度 temperature，然后根据公式计算速度标度变换的因子 scaleFactor，对速度进行标度变换。

```
1  void scaleVelocity(const double T0, Atom& atom)
2  {
3    double temperature =
4      findKineticEnergy(atom) * 2.0 / (3.0 * K_B * atom.number);
5    double scaleFactor = sqrt(T0 / temperature);
6    for (int n = 0; n < atom.number; ++n) {
7      atom.vx[n] *= scaleFactor;
8      atom.vy[n] *= scaleFactor;
9      atom.vz[n] *= scaleFactor;
10   }
11 }
```

下面是通过速度计算动能的函数 findKineticEnergy()。注意，该函数计算的结果将通过一个值返回。

```
1  double findKineticEnergy(const Atom& atom)
2  {
3    double kineticEnergy = 0.0;
4    for (int n = 0; n < atom.number; ++n) {
5      double v2 = atom.vx[n] * atom.vx[n] + atom.vy[n] * atom.vy[n] +
6                  atom.vz[n] * atom.vz[n];
7      kineticEnergy += atom.mass[n] * v2;
8    }
9    return kineticEnergy * 0.5;
10 }
```

2.2.4 运动方程的数值积分

我们用函数 `integrate()` 来实现速度-Verlet 积分算法的两个步骤。该函数的第一个输入参数 `isStepOne` 是一个布尔型变量。当该变量为真时，就实行速度-Verlet 积分算法的第一个步骤，即部分地将速度更新，并完全地将坐标更新。当该变量为假时，就实行速度-Verlet 积分算法的第二个步骤，只更新速度，不再更新坐标。

```
1  void integrate(const bool isStepOne, const double timeStep, Atom& atom
       )
2  {
3    double timeStepHalf = timeStep * 0.5;
4    for (int n = 0; n < atom.number; ++n) {
5      double mass_inv = 1.0 / atom.mass[n];
6      double ax = atom.fx[n] * mass_inv;
7      double ay = atom.fy[n] * mass_inv;
8      double az = atom.fz[n] * mass_inv;
9      atom.vx[n] += ax * timeStepHalf;
10     atom.vy[n] += ay * timeStepHalf;
11     atom.vz[n] += az * timeStepHalf;
12     if (isStepOne) {
13       atom.x[n] += atom.vx[n] * timeStep;
14       atom.y[n] += atom.vy[n] * timeStep;
15       atom.z[n] += atom.vz[n] * timeStep;
16     }
17   }
18 }
```

2.2.5 求势能与力

函数 `findForce()` 负责求体系中各个粒子的势能和受到的力。在该函数的开头，我们定义了若干常量。这种常量的计算将在编译期间就完成。在循环之前尽可能多地计算常量可以省去很多不必要的计算。我们用了固态氩的 LJ 参数 $\epsilon = 0.01032e$ eV，$\sigma = 3.405$ A。并将截断距离取为 $R_c = 10$ A。

接着，我们将每个原子的势能和受力初始化为零，因为在后面的循环中我们将对势能和力进行累加。

接下来，是一个两重循环，因为们要计算每一对粒子之间的相互作用力。不过这里的两重循环有些特殊，排除了 `i >= j` 的可能性，这就是利用牛顿第三定律节约一半的计算量。

在循环体中，首先计算相对位置 $\vec{r}_{ij}$ 并对其实施最小镜像约定。紧接着计算两个粒子距离的平方并忽略截断距离之外的粒子对。最后，通过非常节约的方式计算两个粒子之间的势能和相互作用力，并存储在对应的数组中。这部分的计算要避免使用耗时的 `sqrt()` 函数和 `pow()` 函数。在 LJ 势的编程中，虽然从公式来看好像需要使用，但是仔细思考后会发现这些都是可以避免的。还有一点值得注意，那就是除法运算大概是乘法运算的几倍耗时，所以在编写程序时，要将除法运算的个数最小化。

```
1  void findForce(Atom& atom)
2  {
3    const double epsilon = 1.032e-2;
4    const double sigma = 3.405;
5    const double cutoff = 10.0;
6    const double cutoffSquare = cutoff * cutoff;
7    const double sigma3 = sigma * sigma * sigma;
8    const double sigma6 = sigma3 * sigma3;
9    const double sigma12 = sigma6 * sigma6;
10   const double e24s6 = 24.0 * epsilon * sigma6;
11   const double e48s12 = 48.0 * epsilon * sigma12;
12   const double e4s6 = 4.0 * epsilon * sigma6;
13   const double e4s12 = 4.0 * epsilon * sigma12;
14   for (int n = 0; n < atom.number; ++n)
15     atom.fx[n] = atom.fy[n] = atom.fz[n] = atom.pe[n] = 0.0;
16
17   for (int i = 0; i < atom.number - 1; ++i) {
18     for (int j = i + 1; j < atom.number; ++j) {
19       double xij = atom.x[j] - atom.x[i];
20       double yij = atom.y[j] - atom.y[i];
21       double zij = atom.z[j] - atom.z[i];
```

```
22      applyMic(atom.box, xij, yij, zij);
23      double r2 = xij * xij + yij * yij + zij * zij;
24      if (r2 > cutoffSquare)
25        continue;
26
27      double r2inv = 1.0 / r2;
28      double r4inv = r2inv * r2inv;
29      double r6inv = r2inv * r4inv;
30      double r8inv = r4inv * r4inv;
31      double r12inv = r4inv * r8inv;
32      double r14inv = r6inv * r8inv;
33      double f_ij = e24s6 * r8inv - e48s12 * r14inv;
34      atom.pe[i] += e4s12 * r12inv - e4s6 * r6inv;
35      atom.fx[i] += f_ij * xij;
36      atom.fx[j] -= f_ij * xij;
37      atom.fy[i] += f_ij * yij;
38      atom.fy[j] -= f_ij * yij;
39      atom.fz[i] += f_ij * zij;
40      atom.fz[j] -= f_ij * zij;
41    }
42  }
43 }
```

最小镜像约定的实施由如下两个函数实现。注意，这里的函数参数用了 C++ 里面的引用（Reference）。

```
1 void applyMicOne(const double length, const double halfLength, double&
      x12)
2 {
3   if (x12 < -halfLength)
4     x12 += length;
5   else if (x12 > +halfLength)
6     x12 -= length;
```

```
7  }
8
9  void applyMic(const double box[6], double& x12, double& y12, double&
       z12)
10 {
11   applyMicOne(box[0], box[3], x12);
12   applyMicOne(box[1], box[4], y12);
13   applyMicOne(box[2], box[5], z12);
14 }
```

2.2.6 程序的编译与运行

本书所开发的 C++ 程序都可以在 Linux 和 Windows 操作系统使用。我们推荐使用 GCC 工具。在命令行可以用如下方式编译本章的程序：

```
$ g++ -O2 ljmd.cpp -o ljmd
```

其中，`-O2` 选项表示启动二级优化。编译完成后，将生成名为 `ljmd` 的可执行文件（在 Windows 中为 `ljmd.exe`）。

然后，就可以在命令行使用该程序。为此，先建立一个文件夹，在里面准备好一个 `xyz.in` 文件，然后从命令行进入该文件夹，用如下方式运行程序：

```
$ path/to/ljmd  numSteps temperature timeStep
```

如果使用时忘了给命令行参数，程序会提示正确的用法。以上执行命令中的斜杠 `/` 在 Windows 中需要换成反斜杠 `\`。

3 测试与应用程序

3.1 能量守恒的测试

本章程序不是一次性开发出来的，而是一点一点地写出来的，而且作者在编写的过程中做过很多测试，也犯过很多错误。这个开发过程是很难在书中体现出来的。我们首先测试该程序是否能通过能量守恒的测试。

程序在产出阶段每隔 100 步输出系统的总动能 $K(t)$ 和总势能 $U(t)$，它们都是时间 t（从产出阶段开始计时）的函数。对于大小有限的体系，它们都是随时间 t 涨落的。然而，

根据能量守恒定律，系统动能和势能的和，即总能量 $E(t) = K(t) + V(t)$，应该是不随时间变化的。当然，我们的模拟中使用了具有一定误差的数值积分方法，故总能量也会有一定大小的涨落。这个涨落主要与积分的时间步长有关系。一般来说，积分的时间步长越大，总能量的涨落越大。

图 xxx（a）给出了系统的总动能、总势能和总能量在产出过程中随时间变化的情况。可以看出动能是正的，势能是负的，涨落相对较大；总能是负的，但在该图中看不出有涨落。图 xxx（b）给出了 $(E(t) - \langle E\rangle)/|\langle E\rangle|$，即总能的相对涨落值。可见总能量确实也有涨落，相对涨落值在 10^{-5} 量级。对于很小的体系来说，这是一个合理的值。

4 习题

1. 假设一个体系有 N 个粒子，它们的速度是 $\{\vec{v}_i\}_{i=1}^N$，对应的体系温度是 T。请验证，当对速度做标度变换

$$\vec{v}_i \to \vec{v}_i' = \vec{v}_i\sqrt{\frac{T_0}{T}}. \tag{26}$$

之后，体系的温度将变成 T_0。提示：从温度的微观定义出发。

2. 证明：如果在做速度标度变换

$$\vec{v}_i \to \vec{v}_i' = \vec{v}_i\sqrt{\frac{T_0}{T}}. \tag{27}$$

之前，系统的总动量已经为零，那么在做这个变换之后，系统的总动量也是零。

3. 本书所讨论的力都是保守力。保守力可以表达为势能的负梯度。例如，在具有 N 个粒子的系统中，粒子 i 的受力可以写成如下形式

$$\vec{F}_i = -\nabla_i U \tag{28}$$

这里，U 是整个系统的总势能。请由此推导正文中两体势函数体系中力的表达式：

$$\vec{F}_i = \sum_{j\neq i}^{N} \frac{\partial U_{ij}(r_{ij})}{\partial r_{ij}} \frac{\vec{r}_{ij}}{r_{ij}} \tag{29}$$

4. 对能量守恒的检验只是判断一个分子动力学模拟程序是否正确的方法之一。一个分子动力学模拟程序通过了能量守恒的检验，不代表就没有错误了。请读者修改本章的程序，每 100 步输出各个粒子的速度，然后检验如下两点：

1. 体系的动量受否守恒?

2. 粒子的速度是否满足麦克斯韦速度分布规律?

5 习题解答

1. 假设系统中粒子的质量为 $\{m_i\}_{i=1}^N$，则它们的速度为 $\{\vec{v}_i\}_{i=1}^N$ 时，体系具有总动能

$$E_k = \frac{1}{2}\sum_i \left(m_i \vec{v}_i^2\right) = \frac{3}{2}Nk_BT \tag{30}$$

体系的温度和总动能有如下关系（能量均分定理）：

$$\frac{3}{2}Nk_BT = E_k \tag{31}$$

于是有

$$T = \frac{1}{3Nk_B}\sum_i \left(m_i \vec{v}_i^2\right) \tag{32}$$

当速度变为 $\{\vec{v}_i'\}_{i=1}^N$ 之后，体系的温度将变为

$$T' = \frac{1}{3Nk_B}\sum_i \left(m_i \vec{v}_i'^2\right) = \frac{1}{3Nk_B}\sum_i \left(m_i \vec{v}_i^2 \frac{T_0}{T}\right) = \frac{1}{3Nk_B}\sum_i \left(m_i \vec{v}_i^2\right)\frac{T_0}{T} = T\frac{T_0}{T} = T_0. \tag{33}$$

证毕。