用分子动力学模拟研究热输运

樊哲勇

July 10, 2017

1 引言

根据热力学第二定律，一个系统在不受外场作用时，若其内部有热力学性质的不均匀性，则它一定处于非平衡的状态，并有向平衡态靠近的趋势。这种由热力学性质的不均匀性导致的热力学过程叫做输运过程 (transport process)，相应的现象叫做输运现象。例如，温度的不均匀性导致能量的输运（热传导现象）；粒子数密度的不均匀性导致粒子的输运（扩散现象）。将一个系统置于两个温度不同的热源之间，最终会在系统内建立一个稳定的 (不随时间变化的) 温度分布。我们说这样的系统处于一个稳态 (steady state)，但不处于一个平衡态 (equilibrium state)。稳态和平衡态都是不依赖与时间的，但前者属于非平衡态。

上述不均匀性都是由相应的不均匀的物理量的梯度来量化的。这里，我们只研究热输运，而且假设输运方向沿着一个特定方向（假设是 x 方向）的情形。热传导现象的宏观规律由傅里叶定律描述。傅里叶定律是说热流密度 (heat flux，或者 heat current) J，即单位时间穿过单位面积的热量，在数量上正比于温度梯度 $\frac{dT}{dx}$：

$$J = -\kappa \frac{dT}{dx}. \tag{1}$$

这里的 κ 就反映了热量输运的难易程度：κ 越大代表热量越容易被输运。这样的物理量被称为输运系数 (transport coefficient)。具体到热传导，输运系数 κ 叫做热导率 (thermal conductivity)。注意等式右边有个负号，它表示热量的传导方向与温度梯度的方向相反，指向温度降低的方向 (一个物理量的梯度的方向指向它增加的方向)。在国际单位制中，温度梯度的单位为 K/m，热流密度的单位是 $\mathrm{W/m^2}$，故热导率的单位是 $\mathrm{Wm^{-1}K^{-1}}$。

必须区分热导率 (thermal conductivity) 和热导 (thermal conductance) 这两个不同的概念。考虑横截面积为 A，长度为 d 一个长方体材料，热导 K 与热导率 κ 有如下关系：

$$K = \kappa \frac{A}{d}. \tag{2}$$

更常用的是单位面积的热导，我们用符号 G 表示。对于上述长方体材料，单位面积的热导为

$$G = \kappa \frac{K}{A} = \frac{\kappa}{d}. \tag{3}$$

这样定义的 G 的单位为 $\mathrm{Wm^{-2}K^{-1}}$。为简单起见，我们以后都简称 G 为热导。我们也将简称热流密度为热流。这些都是文献中常用的简称，虽然不严谨。

有很多计算热导率和热导的方法。其中，最常用的方法是：

- 玻尔兹曼输运方程 (Boltzmann transport euqation, BTE)。玻尔兹曼输运方程是基于晶格动力学的一种方法，常用于预言新晶体材料的热导率。
  优点：
  - 声子统计可采用量子统计，适用于低温（温度远小于德拜温度）的情形。
  - 力常数可由第一性原理计算，一般来说比基于经验势的方法更准确。

  缺点：
  - 该方法的计算量对原胞的大小呈高次方关系，不适用于太复杂的纳米结构。
  - 该方法一般仅考虑三声子散射，忽略了高阶声子散射，不适用温度太高（接近熔点）的情形。
- 非平衡格林函数 (nonequilibrium Green's function, NEGF)方法。
  优点：
  - 可以和第一性原理结合。
  - 采用量子统计。

  缺点：
  - 该方法的计算量对体系的宽度成立方关系，不适用于太宽的体系。
  - 声子-声子散射很难在该方法中实现，故该方法主要用于计算界面热导，而不是材料的热导率。
- 分子动力学（molecular dynamics, MD）方法。
  优点：
  - 该方法是最通用的方法，可以计算热导率，也可以计算（界面）热导。
  - 考虑了各阶声子散射，适用于高温情形，也适用于流体系统。
  - 计算量一般仅正比于体系原子总数，故可以用于非常复杂的大尺度体系（例如有百万原子的体系）。

  缺点：
  - 声子统计为经典统计，不适用于低温（远低于德拜温度）的情形。
  - 一般采用经验势，结果的可信度往往取决于经验势的可靠程度。
  - 因为依赖于经验势，所以如果一个体系缺乏可靠的经验势，就无法进行模拟（除非自己动手开发一个经验势）。

本讲要介绍的就是用分子动力学计算热导率的理论、算法、程序、以及范例。主要有两种基于分子动力学的计算热导率的方法：

- 平衡态分子动力学（equilibrium molecular dynamics，EMD）方法，也常称为格林-久保（Green-Kubo）方法，因为该方法基于格林-久保公式 [1,2]。
  优点：
  - 该方法的尺寸效应相对较小。
  - 系统处于平衡态，相对稳定。

  缺点：
  - 该方法中没有明确的长度概念。
- 非平衡态分子动力学（nonequilibrium molecular dynamics，NEMD）方法。
  优点：
  - 该方法中有明确的长度概念。

  缺点：
  - 该方法有明显的尺寸效应。如果要模拟无限大体系的热导率，需要模拟多个不同长度的系统再外推。
  - 系统处于非平衡态，相对不稳定（热流过大可能会让体系崩溃）。

2 用平衡态分子动力学计算热导率

2.1 计算热导率的 Green-Kubo 公式

格林-久保公式实际上是一类公式，它们将非平衡过程的输运系数与平衡态中相应物理量的涨落相联系。格林-久保公式是说，输运系数等于自关联函数对关联时间的积分。例如，扩散系数是速度自关联函数的积分；粘性系数是压力自关联函数的积分；热导率是热流自关联函数的积分、等等。我推荐阅读 Haile的书 [3] 的第七章。

对热导率的计算有如下格林-久保公式：

$$\kappa_{\mu\nu}(t) = \frac{V}{k_B T^2} \int_0^t dt' C_{\mu\nu}(t'). \tag{4}$$

其中，$\kappa_{\mu\nu}(t)$ $(\mu, \nu = x, y, z)$ 是热导率张量，t' 是关联时间，k_B 是 Boltzmann 常数, T 是温度, V 是体积，$C_{\mu\nu}(t)$ 是热流自关联函数（heat current autocorrelation function，常简称为 HCACF）。上式计算的跑动热导率（running thermal conductivity）。关于跑动热导率的技术细节，将在具体的范例中讨论。

热流自关联函数的表达式如下：

$$C_{\mu\nu}(t) = \langle J_\mu(0) J_\nu(t) \rangle. \tag{5}$$

其中，尖括号表示统计平均，在分子动力学模拟中指对时间原点的平均，J_μ ($\mu = x, y, z$) 是热流。下一节讨论关联函数；下下节讨论热流的具体表达式。

如果研究的是三维的各向同性（isotropic）的系统，则热导率张量的非对角分量一定为零，并可将最终计算的热导率取为对角分量的平均值：

$$\kappa = \frac{\kappa_{xx} + \kappa_{yy} + \kappa_{zz}}{3}. \tag{6}$$

使用格林-久保方法时要注意边界条件的选取：

- 如果模拟的是三维块体（bulk）系统，则每个方向都要使用周期边界条件。
- 如果要研究准两维系统（如薄膜或者两维材料），则在垂直于薄膜的方向用自由边界条件，在平行于薄膜的方向用周期边界条件。而且，此时垂直方向热导率的计算结果无意义。
- 如果要研究的是准一维系统（如纳米线或者纳米管），则在垂直于线或者管的方向都要用自由边界条件，在平行于线或者管的方向用周期边界条件。而且，此时只有平行方向的热导率结果才有意义。

2.2 时间关联函数

我再次推荐阅读 Haile的书 [3] 的第七章。

在统计力学中，设有两个依赖于时间的物理量 $A(t)$ 和 $B(t)$，我们定义这两个量之间的时间关联函数 (time correlation function) $C(t)$ 为：

$$C(t) = \langle A(t_0) B(t_0 + t) \rangle . \tag{7}$$

对这个公式的说明如下：

- 如果两个物理量相同，$A = B$，那么上式代表物理量 A 的自关联函数（auto-correlation function）。
- 上式中 t 是某个时间间隔，叫做关联时间（correlation time），而时间关联函数 $C(t)$ 是关联时间 t 的函数。
- 尖括号在统计物理中代表系综平均，但在分子动力学模拟中一般代表“时间”平均，其中“时间”指的不是上述“关联时间” t，而是“时间原点” t_0。一般将 t_0 写作 0，即常写 $C(t) = \langle A(0) B(t) \rangle$。
- 我们这里考虑的是平衡系统，即在每一个时间原点 t_0，系统都处于平衡态。所以，不同的时间原点在物理上是等价的，从而可以对时间原点求平均。

在分子动力学模拟中，我们只能得到一条离散的相轨迹（phase trajectory），故在实际计算中，与尖括号对应的时间平均将由求和方式表示。

首先，我们假设先在控制温度的情况下让系统达到平衡。然后在不控制温度的情况下（即在微正则系综）模拟了 N_p 步（下标 p 是 production 的意思），步长为 Δt。与计算静态热力学量的情形类似，我们不需要将每一步的数据都保存（因为相邻步的数据有关联性，保存得过频无益）。我们假设每 N_s (下标 s 是 sampling 的意思) 步保存一次数据，并称 N_s 为取样间隔 (sampling interval)。我们假设 N_p 是 N_s 的整数倍，记

$$\frac{N_p}{N_s} = N_d \tag{8}$$

代表记录数据的总步数（下标 d 是 data 的意思），并记

$$N_s \Delta t = \Delta\tau. \tag{9}$$

根据上面的讨论与记号，我们可以将关联时间为 t 的关联函数表达为：

$$C(t) = \frac{1}{M}\sum_{m=1}^{M} A(m\Delta\tau)B(m\Delta\tau + t). \tag{10}$$

其中，M 是求平均时用的时间原点数目。原则上，对不同的关联时间 t，可被利用的时间原点的最大数目 M 是不同的（注意：t 一定是 $\Delta\tau$ 的倍数）：

$$M = N_d - \frac{t}{\Delta\tau}. \tag{11}$$

关联时间等于零时，最大时间原点数目 $M = N_d$；关联时间等于几个时间步长，M 的值就要减少几。在实际的模拟中，要考虑的最大关联数据量 N_c (下标 c 是 correlation 的意思) 通常远小于 N_d，如果希望得到比较精确的模拟结果的话。所以，为了编程方便（特别是用并行计算时），通常将 M 的值统一取为

$$M = N_d - N_c. \tag{12}$$

这样的话，虽然浪费了一小部分数据，但好处是与每个关联时间对应的关联函数值的统计平均采用了统一的数据量（即时间原点数）。从统计的角度来看，这样做是更合理的。

我举一个非常简单的例子。假设保存了 $N_d = 100$ 组数据，并且要计算 $N_c = 10$ 个关联函数值。首先确定 $M = 100 - 10 = 90$。这 10 个关联函数值为

$$C(0) = \frac{1}{90}\sum_{m=1}^{90} A(m\Delta\tau)B(m\Delta\tau); \tag{13}$$

$$C(\Delta\tau) = \frac{1}{90}\sum_{m=1}^{90} A(m\Delta\tau)B(m\Delta\tau + \Delta\tau); \tag{14}$$

$$C(2\Delta\tau) = \frac{1}{90}\sum_{m=1}^{90} A(m\Delta\tau)B(m\Delta\tau + 2\Delta\tau); \tag{15}$$

$$C(9\Delta\tau) = \frac{1}{90}\sum_{m=1}^{90} A(m\Delta\tau)B(m\Delta\tau + 9\Delta\tau). \tag{16}$$

2.3 热流的表达式

热流定义为能量密度矩的时间导数：

$$\boldsymbol{J} = \frac{1}{V}\frac{d}{dt}\sum_i \boldsymbol{r}_i E_i. \tag{17}$$

其中，

$$E_i = \frac{1}{2}m_i\boldsymbol{v}_i^2 + U_i \tag{18}$$

是第 i 个粒子的总能量，U_i 是势能部分，$\frac{1}{2}m_i\boldsymbol{v}_i^2$ 是动能部分。

首先，用求导的莱布尼茨法则可得

$$\boldsymbol{J} = \sum_i \boldsymbol{v}_i E_i + \sum_i \boldsymbol{r}_i \frac{d}{dt}E_i \tag{19}$$

通常将上式右边的两项分别称为动能项

$$\boldsymbol{J}_{\text{kin}} = \sum_i \boldsymbol{v}_i E_i \tag{20}$$

和势能项

$$\boldsymbol{J}_{\text{pot}} = \sum_i \boldsymbol{r}_i \frac{d}{dt}E_i. \tag{21}$$

合起来，有

$$\boldsymbol{J} = \boldsymbol{J}_{\text{kin}} + \boldsymbol{J}_{\text{pot}}. \tag{22}$$

动能项不需要再推导了。利用动能定理

$$\frac{d}{dt}\left(\frac{1}{2}m_i\boldsymbol{v}_i^2\right) = \boldsymbol{F}_i \cdot \boldsymbol{v}_i, \tag{23}$$

可以将势能项写为

$$\boldsymbol{J}_{\text{pot}} = \sum_i \boldsymbol{r}_i(\boldsymbol{F}_i \cdot \boldsymbol{v}_i) + \sum_i \boldsymbol{r}_i \frac{dU_i}{dt}. \tag{24}$$

在两体势（two-body potentials）情形中，系统总势能可表达为

$$U = \frac{1}{2}\sum_i \sum_{j\neq i} U_{ij}(r_{ij}). \tag{25}$$

可以推导如下力的表达式：

$$\boldsymbol{F}_i = \sum_{j \neq i} \boldsymbol{F}_{ij}, \tag{26}$$

$$\boldsymbol{F}_{ij} = \frac{\partial U_{ij}}{\partial \boldsymbol{r}_{ij}} = -\boldsymbol{F}_{ji}. \tag{27}$$

其中，$\boldsymbol{F}_{ij}$ 是第 i 个粒子受到的来自于第 j 个粒子的力，

$$\boldsymbol{r}_{ij} \equiv \boldsymbol{r}_j - \boldsymbol{r}_i, \tag{28}$$

是从第 i 个粒子指向第 j 个粒子的位置差矢量。

练习。定义

$$U_i = \frac{1}{2} \sum_{j \neq i} U_{ij}, \tag{29}$$

推导如下公式

$$\boldsymbol{J}_{\text{pot}} = \frac{1}{2} \sum_i \sum_{j \neq i} \boldsymbol{r}_i [\boldsymbol{F}_{ij} \cdot (\boldsymbol{v}_i + \boldsymbol{v}_j)]. \tag{30}$$

在分子动力学模拟中，如果采用了周期边界条件，一个宏观物理量就不可能依赖于绝对坐标 $\boldsymbol{r}_i$, 而应该依赖于相对坐标。证明上面的热流表达式等价于

$$\boldsymbol{J}_{\text{pot}} = -\frac{1}{4} \sum_i \sum_{j \neq i} \boldsymbol{r}_{ij} [\boldsymbol{F}_{ij} \cdot (\boldsymbol{v}_i + \boldsymbol{v}_j)]. \tag{31}$$

再证明上式等价于下面的不怎么对称的形式：

$$\boldsymbol{J}_{\text{pot}} = -\frac{1}{2} \sum_i \sum_{j \neq i} \boldsymbol{r}_{ij} [\boldsymbol{F}_{ij} \cdot \boldsymbol{v}_i]. \tag{32}$$

在分子动力学模拟中，位力 (virial) 张量定义为

$$\mathbf{W} = \sum_i \mathbf{W}_i, \tag{33}$$

其中，$\mathbf{W}_i$ 是单粒子位力：

$$\mathbf{W}_i = -\frac{1}{2} \sum_{j \neq i} \boldsymbol{r}_{ij} \otimes \boldsymbol{F}_{ij}. \tag{34}$$

于是有：

$$\boldsymbol{J}_{\text{pot}} = \sum_i \mathbf{W}_i \cdot \boldsymbol{v}_i. \tag{35}$$

这个公式就是 LAMMPS 中用的热流公式，但它只适用于两体势，不适用于多体势（many-body potentials）。对多体势的讨论，请参看我的论文 [4]。

3 用非平衡态分子动力学计算热导率

前已提及，热传导是温度梯度引起能量传递的现象，属于非平衡过程。另一种计算热导率的方法就是基于非平衡分子动力学模拟的。在该方法中，首先通过某种方式建立一个温度梯度，并等待足够长的时间，使得系统达到稳态。在达到稳态后，对系统的温度梯度和非平衡稳态热流进行测量，然后就可以根据傅里叶定律来计算热导率。

产生温度梯度的方法有多种。在每一种方法中，都将模拟系统在热传导方向分割为若干块（blocks）。每个块的大小可以不等，但一般为了处理数据的方便，取为相等。将其中的两块分别设置为热源（heat source）区域和热汇（heat sink）区域。热源和热汇区域的位置选取与模拟的边界条件相关，具体如下：

- 如果采用周期边界条件，则块的个数一般取为偶数 $2N$（假设从 1 到 $2N$ 进行标记），并可以取第 1 块为热源，取第 $N+1$ 块为热汇，或者反过来。参见后面的范例。
- 如果采用固定边界条件，则一般将传导方向的前后两端一小层原子固定，并将固定层附近的块取为热源和热汇。此时块的个数可以不为偶数。

下面一一介绍产生温度梯度的方法。

3.1 动量交换法

该方法由 Müller-Plathe [5] 提出，其算法为：

- 找出热区中最“冷”（即速率最小）的原子，假设它的速度为$\boldsymbol{v}_c$。
- 找出冷区中最“热”（即速率最大）的原子，假设它的速度为$\boldsymbol{v}_h$。
- 用如下方式交换动量

$$\boldsymbol{v}_c' = -\boldsymbol{v}_c + 2\frac{m_c\boldsymbol{v}_c + m_h\boldsymbol{v}_h}{m_c + m_h}, \tag{36}$$

$$\boldsymbol{v}_h' = -\boldsymbol{v}_h + 2\frac{m_c\boldsymbol{v}_c + m_h\boldsymbol{v}_h}{m_c + m_h}. \tag{37}$$

当$m_c = m_h$时，上述公式就等价于交换速度

$$\boldsymbol{v}_c' = \boldsymbol{v}_h, \tag{38}$$

$$\boldsymbol{v}_h' = \boldsymbol{v}_c. \tag{39}$$

一般情况下，还是要交换动量，因为简单地交换速度可能会导致系统总动量不守恒。可以证明（练习），交换动量的公式确实满足动量守恒，即

$$m_c\boldsymbol{v}_c' + m_h\boldsymbol{v}_h' = m_c\boldsymbol{v}_c + m_h\boldsymbol{v}_h. \tag{40}$$

- 每次交换动量会导致一个能量交换。将交换的能量累加起来除以相应的时间以及系统的横截面积就可以得到非平衡热流 Q 的数值。
- 在系统达到稳态之后，开始测量系统各处的温度，通过拟合可得到一个温度梯度 $\frac{\partial T}{\partial x}$，然后根据傅里叶定律计算热导率：

$$\kappa = \frac{Q}{|\frac{\partial T}{\partial x}|}. \tag{41}$$

3.2 速度重标法

这个方法首先被 Ikeshoji 和 Hafskjold [6] 提出，后来又被 Jund 和 Jullien [7] 提出（估计后者不知道前者的工作）。该方法的算法为：

- 在系统达到平衡态之后，开始有规律地增加热源中粒子的速率，即对热源中的每一个粒子 i，作如下速度变换

$$\boldsymbol{v}_i' = \boldsymbol{v}_c + \sqrt{1 + \frac{\Delta\epsilon}{E_k^r}}(\boldsymbol{v}_i - \boldsymbol{v}_c). \tag{42}$$

其中，

$$\boldsymbol{v}_c = \frac{\sum_i m_i \boldsymbol{v}_i}{\sum_i m_i} \tag{43}$$

是热源粒子的质心速度，

$$E_k^r = \frac{1}{2}\sum_i m_i \boldsymbol{v}_i^2 - \frac{1}{2}\sum_i m_i \boldsymbol{v}_c^2 = \frac{1}{2}\sum_i m_i (\boldsymbol{v}_i - \boldsymbol{v}_c)^2 \tag{44}$$

是热源中粒子相对于它们的质心的动能，$\Delta\epsilon$ 是一个常数。可以证明（练习），该常数等于热区中能量的增量，即

$$\frac{1}{2}\sum_i m_i \boldsymbol{v}_i'^2 - \frac{1}{2}\sum_i m_i \boldsymbol{v}_i^2 = \Delta\epsilon. \tag{45}$$

还可以证明（练习），速度变换过程中热源中粒子的总动量保持不变，即

$$\sum_i m_i \boldsymbol{v}_i' = \sum_i m_i \boldsymbol{v}_i. \tag{46}$$

- 类似地，减小热汇中粒子的速率，即对热汇中的每一个粒子 i，作如下速度变换

$$\boldsymbol{v}_i' = \sqrt{1 - \frac{\Delta\epsilon}{E_k^r}}(\boldsymbol{v}_i - \boldsymbol{v}_c) \tag{47}$$

- 如果每 Δt 作一次速度变换，则产生的热流密度大小为

$$Q = \frac{\Delta \epsilon}{A \Delta t}. \tag{48}$$

  其中，A 是模拟系统在垂直于传导方向的横截面积。注意：用周期边界条件时，应该将该热流除以 2，因为注入热源的热量将从热源区域的左右两边流出，而从每一边流出的热量是总热量的一半）。

- 在系统达到稳态之后，开始测量温度梯度，然后根据傅里叶定律计算热导率。

3.3 局部热浴法

这个方法好像不归功于任何个人。我不知道是谁最先用这个方法的，但我认为这个方法是最方便的。在动量交换法中，动量交换的频率这个参数一般要用试错法来确定。交换过频，引起的温度梯度会过大。同样，在速度重标法中，每次给热区增加的能量值也一般要用试错法来确定。相比之下，在局部热浴法中，我们可以直接设置热源和热汇的温度值。例如，如果要模拟一个系统在 300 K 时的热传导，我们可以将热源的温度设置为 310 K，将热汇的温度设置为 290 K，从而直接保证温度梯度具有合理的值。

该方法的算法为：

- 在系统达到平衡态之后，撤掉整体热浴，分别对热源和热汇区域施加局部热浴，并取合适的温度值。常用的局部热浴有 Nose-Hoover (chain) 热浴和郎之万（Langevin）热浴。

- 热流大小可以通过系统和热浴交换的能量来计算（能量守恒）。

- 在系统达到稳态之后，开始测量温度梯度，然后根据傅里叶定律计算热导率。

4 程序和脚本

学习基本理论之后，还需要做一些练习。可以自己写程序，也可以使用现有的程序。当今最流行的经典分子动力学模拟程序之一是 LAMMPS [8]。作者最近也开发了一个叫做 GPUMD [9] 的高效的分子动力学模拟程序，非常适合研究某些材料，特别是由多体势描述的两维材料的热导率 [10]。该程序将很快公开。

下面的讨论基于固态氩系统。使用该系统的原因有三：

- 固态氩系统有非常简单的相互作用势，即 Lennard-Jones 势，容易编程、理解。

- 文献中有很多关于固态氩的热导率的计算结果，便于比较。例如，大家可以尝试重复作者的一篇论文 [11] 的结果。

- 将固态氩热导率的计算完全弄清楚之后，很容易将获得的经验应用于更复杂的情形。

具体地说，我们计算 60 K 的固态氩系统的晶格热导率。该温度下固态氩的晶格常数约为 5.4 Å。我将给出三个例子，它们分别

- 用自编程序和 EMD 方法；
- 用 LAMMPS 和 EMD 方法；
- 用 LAMMPS 和 NEMD 方法。

4.1 一个用平衡态方法计算固态氩热导率的 C 语言程序

该程序已经在 github 公开，网址为：

- https://github.com/brucefan1983/heat-conductivity-emd

下面是该程序的完整源代码：

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>

#define K_B                    8.617e-5 // Boltzmann's constant
#define TIME_UNIT_CONVERSION   1.018e+1 // fs     <-> my natural unit
#define KAPPA_UNIT_CONVERSION 1.574e+5 // W/(mK) <-> my natural unit


void apply_mic
(
    double lx, double ly, double lz, double lxh, double lyh,
    double lzh, double *x12, double *y12, double *z12
)
{
    if (*x12 < - lxh) {*x12 += lx;} else if (*x12 > + lxh) {*x12 -= lx;}
    if (*y12 < - lyh) {*y12 += ly;} else if (*y12 > + lyh) {*y12 -= ly;}
    if (*z12 < - lzh) {*z12 += lz;} else if (*z12 > + lzh) {*z12 -= lz;}
}


void find_neighbor
(
    int N, int *NN, int *NL, double *x, double *y, double *z,
    double lx, double ly, double lz, int MN, double cutoff
```

```
)
{
    double lxh = lx * 0.5;
    double lyh = ly * 0.5;
    double lzh = lz * 0.5;
    double cutoff_square = cutoff * cutoff;
    for (int n = 0; n < N; n++) {NN[n] = 0;}
    for (int n1 = 0; n1 < N - 1; n1++)
    {
        for (int n2 = n1 + 1; n2 < N; n2++)
        {
            double x12 = x[n2] - x[n1];
            double y12 = y[n2] - y[n1];
            double z12 = z[n2] - z[n1];
            apply_mic(lx, ly, lz, lxh, lyh, lzh, &x12, &y12, &z12);
            double  distance_square = x12 * x12 + y12 * y12 + z12 * z12;
            if (distance_square < cutoff_square)
            {
                NL[n1 * MN + NN[n1]] = n2;
                NN[n1]++;
                NL[n2 * MN + NN[n2]] = n1;
                NN[n2]++;
            }
            if (NN[n1] > MN)
            {
                printf("Error: cutoff for neighbor list is too large.\n");
                exit(1);
            }
        }
    }
}


void initialize_position
(
    int n0, int nx, int ny, int nz, double ax, double ay, double az,
    double *x, double *y, double *z
)
{
    double x0[4] = {0.0, 0.0, 0.5, 0.5}; // only for simple FCC lattice
    double y0[4] = {0.0, 0.5, 0.0, 0.5};
    double z0[4] = {0.0, 0.5, 0.5, 0.0};
    int n = 0;
    for (int ix = 0; ix < nx; ++ix)
    {
```

```
        for (int iy = 0; iy < ny; ++iy)
        {
            for (int iz = 0; iz < nz; ++iz)
            {
                for (int i = 0; i < n0; ++i)
                {
                    x[n] = (ix + x0[i]) * ax;
                    y[n] = (iy + y0[i]) * ay;
                    z[n] = (iz + z0[i]) * az;
                    n++;
                }
            }
        }
    }
}


void scale_velocity
(int N, double T_0, double *m, double *vx, double *vy, double *vz)
{
    double temperature = 0.0;
    for (int n = 0; n < N; ++n)
    {
        double v2 = vx[n] * vx[n] + vy[n] * vy[n] + vz[n] * vz[n];
        temperature += m[n] * v2;
    }
    temperature /= 3.0 * K_B * N;
    double scale_factor = sqrt(T_0 / temperature);
    for (int n = 0; n < N; ++n)
    {
        vx[n] *= scale_factor;
        vy[n] *= scale_factor;
        vz[n] *= scale_factor;
    }
}


void initialize_velocity
(int N, double T_0, double *m, double *vx, double *vy, double *vz)
{
    double momentum_average[3] = {0.0, 0.0, 0.0};
    for (int n = 0; n < N; ++n)
    {
        vx[n] = -1.0 + (rand() * 2.0) / RAND_MAX;
        vy[n] = -1.0 + (rand() * 2.0) / RAND_MAX;
```

```
        vz[n] = -1.0 + (rand() * 2.0) / RAND_MAX;

        momentum_average[0] += m[n] * vx[n] / N;
        momentum_average[1] += m[n] * vy[n] / N;
        momentum_average[2] += m[n] * vz[n] / N;
    }
    for (int n = 0; n < N; ++n)
    {
        vx[n] -= momentum_average[0] / m[n];
        vy[n] -= momentum_average[1] / m[n];
        vz[n] -= momentum_average[2] / m[n];
    }
    scale_velocity(N, T_0, m, vx, vy, vz);
}


void find_force
(
    int N, int *NN, int *NL, int MN, double lx, double ly, double lz,
    double *x, double *y, double *z, double *fx, double *fy, double *fz,
    double *vx, double *vy, double *vz, double *hc
)
{
    const double epsilon = 1.032e-2;
    const double sigma = 3.405;
    const double cutoff = sigma * 3.0;
    const double cutoff_square = cutoff * cutoff;
    const double sigma_3 = sigma * sigma * sigma;
    const double sigma_6 = sigma_3 * sigma_3;
    const double sigma_12 = sigma_6 * sigma_6;
    const double factor_1 = 24.0 * epsilon * sigma_6;
    const double factor_2 = 48.0 * epsilon * sigma_12;

    // initialize heat current and force
    hc[0] = hc[1] = hc[2] = 0.0;
    for (int n = 0; n < N; ++n) { fx[n]=fy[n]=fz[n]=0.0; }

    double lxh = lx * 0.5;
    double lyh = ly * 0.5;
    double lzh = lz * 0.5;
    for (int i = 0; i < N; ++i)
    {
        for (int k = 0; k < NN[i]; k++)
        {
            int j = NL[i * MN + k];
```

```
            if (j < i) { continue; } // will use Newton's 3rd law

            double x_ij = x[j] - x[i];
            double y_ij = y[j] - y[i];
            double z_ij = z[j] - z[i];
            apply_mic(lx, ly, lz, lxh, lyh, lzh, &x_ij, &y_ij, &z_ij);

            double r_2 = x_ij * x_ij + y_ij * y_ij + z_ij * z_ij;
            if (r_2 > cutoff_square) { continue; }

            double r_4  = r_2 * r_2;
            double r_8  = r_4 * r_4;
            double r_14 = r_2 * r_4 * r_8;
            double f_ij = factor_1 / r_8 - factor_2 / r_14;
            fx[i] += f_ij * x_ij; fx[j] -= f_ij * x_ij; // Newton's 3rd law
            fy[i] += f_ij * y_ij; fy[j] -= f_ij * y_ij;
            fz[i] += f_ij * z_ij; fz[j] -= f_ij * z_ij;
            double f_dot_v
                = x_ij*(vx[i]+vx[j])+y_ij*(vy[i]+vy[j])+z_ij*(vz[i]+vz[j]);
            f_dot_v *= f_ij * 0.5;
            hc[0] -= x_ij * f_dot_v; // calculate heat current
            hc[1] -= y_ij * f_dot_v;
            hc[2] -= z_ij * f_dot_v;
        }
    }
}


void integrate
(
    int N, double time_step, double *m, double *fx, double *fy, double *fz,
    double *vx, double *vy, double *vz, double *x, double *y, double *z,
    int flag
)
{
    double time_step_half = time_step * 0.5;
    for (int n = 0; n < N; ++n)
    {
        double mass_inv = 1.0 / m[n];
        double ax = fx[n] * mass_inv;
        double ay = fy[n] * mass_inv;
        double az = fz[n] * mass_inv;
        vx[n] += ax * time_step_half;
        vy[n] += ay * time_step_half;
        vz[n] += az * time_step_half;
```

```
        if (flag == 1)
        {
            x[n] += vx[n] * time_step;
            y[n] += vy[n] * time_step;
            z[n] += vz[n] * time_step;
        }
    }
}


void find_hac
(
    int Nc, int M, double *hx, double *hy, double *hz, double *hac_x,
    double *hac_y, double *hac_z
)
{
    for (int nc = 0; nc < Nc; nc++) // loop over the correlation time points
    {
        for (int m = 0; m < M; m++) // loop over the time origins
        {
            hac_x[nc] += hx[m] * hx[m + nc];
            hac_y[nc] += hy[m] * hy[m + nc];
            hac_z[nc] += hz[m] * hz[m + nc];
        }
        hac_x[nc] /= M; hac_y[nc] /= M; hac_z[nc] /= M;
    }
}


static void find_rtc
(
    int Nc, double factor, double *hac_x, double *hac_y, double *hac_z,
    double *rtc_x, double *rtc_y, double *rtc_z
)
{
    for (int nc = 1; nc < Nc; nc++)
    {
        rtc_x[nc] = rtc_x[nc - 1] + (hac_x[nc - 1] + hac_x[nc]) * factor;
        rtc_y[nc] = rtc_y[nc - 1] + (hac_y[nc - 1] + hac_y[nc]) * factor;
        rtc_z[nc] = rtc_z[nc - 1] + (hac_z[nc - 1] + hac_z[nc]) * factor;
    }
}


void find_hac_kappa
```

```
(
    int Nd, int Nc, double dt, double T_0, double V,
    double *hx, double *hy, double *hz
)
{
    double dt_in_ps = dt * TIME_UNIT_CONVERSION / 1000.0; // ps
    int M = Nd - Nc; // number of time origins

    double *hac_x = (double *)malloc(sizeof(double) * Nc);
    double *hac_y = (double *)malloc(sizeof(double) * Nc);
    double *hac_z = (double *)malloc(sizeof(double) * Nc);
    double *rtc_x = (double *)malloc(sizeof(double) * Nc);
    double *rtc_y = (double *)malloc(sizeof(double) * Nc);
    double *rtc_z = (double *)malloc(sizeof(double) * Nc);
    for (int nc = 0; nc < Nc; nc++)
    {hac_x[nc] = hac_y[nc] = hac_z[nc] = 0.0;}
    for (int nc = 0; nc < Nc; nc++)
    {rtc_x[nc] = rtc_y[nc] = rtc_z[nc] = 0.0;}

    find_hac(Nc, M, hx, hy, hz, hac_x, hac_y, hac_z);
    double factor = dt * 0.5 *  KAPPA_UNIT_CONVERSION / (K_B * T_0 * T_0 * V);
    find_rtc(Nc, factor, hac_x, hac_y, hac_z, rtc_x, rtc_y, rtc_z);

    FILE *fid = fopen("kappa.txt", "a");
    for (int nc = 0; nc < Nc; nc++)
    {
        fprintf
        (
            fid, "%25.15e%25.15e%25.15e%25.15e%25.15e%25.15e%25.15e\n",
            nc * dt_in_ps,
            hac_x[nc], hac_y[nc], hac_z[nc],
            rtc_x[nc], rtc_y[nc], rtc_z[nc]
        );
    }
    fclose(fid);

    free(hac_x); free(hac_y); free(hac_z);
    free(rtc_x); free(rtc_y); free(rtc_z);
}


int main(int argc, char *argv[])
{
    srand(time(NULL));
    int nx = 4; // number of unit cells in the x-direction
```

```
int ny = 4; // number of unit cells in the y-direction
int nz = 4; // number of unit cells in the z-direction
int n0 = 4; // number of particles in the unit cell
int N = n0 * nx * ny * nz; // total number of particles
int Ne = 20000;   // number of steps in the equilibration stage
int Np = 20000;   // number of steps in the production stage
int Ns = 10;      // sampling interval
int Nd = Np / Ns; // number of heat current data
int Nc = Nd / 10;   // number of correlation data
int MN = 100;     // maximum number of neighbors for one particle

// See Fig. 5(a) in [Computer Physics Communications 184, 1414 (2013)]
// for lattice constants at different temperatures
double T_0 = 60.0;    // temperature prescribed
double ax = 5.4;      // lattice constant in the x direction
double ay = ax;       // lattice constant in the y direction
double az = ax;       // lattice constant in the z direction
double lx = ax * nx;  // box length in the x direction
double ly = ay * ny;  // box length in the y direction
double lz = az * nz;  // box length in the z direction
double cutoff = 10.0; // cutoff distance for neighbor list

double time_step = 10.0 / TIME_UNIT_CONVERSION; // time step

// fixed neighbor list
int *NN = (int*) malloc(N * sizeof(int));
int *NL = (int*) malloc(N * MN * sizeof(int));

// major data for the particles
double *m  = (double*) malloc(N * sizeof(double)); // mass
double *x  = (double*) malloc(N * sizeof(double)); // position
double *y  = (double*) malloc(N * sizeof(double));
double *z  = (double*) malloc(N * sizeof(double));
double *vx = (double*) malloc(N * sizeof(double)); // velocity
double *vy = (double*) malloc(N * sizeof(double));
double *vz = (double*) malloc(N * sizeof(double));
double *fx = (double*) malloc(N * sizeof(double)); // force
double *fy = (double*) malloc(N * sizeof(double));
double *fz = (double*) malloc(N * sizeof(double));
double *hx = (double*) malloc(Nd * sizeof(double)); // heat current
double *hy = (double*) malloc(Nd * sizeof(double));
double *hz = (double*) malloc(Nd * sizeof(double));

// initialize mass, position, and velocity
for (int n = 0; n < N; ++n) { m[n] = 40.0; } // mass for argon atom
```

```
initialize_position(n0, nx, ny, nz, ax, ay, az, x, y, z);
initialize_velocity(N, T_0, m, vx, vy, vz);

// initialize neighbor list and force
find_neighbor(N, NN, NL, x, y, z, lx, ly, lz, MN, cutoff);
double hc[3]; // heat current at a specific time point
find_force
(N, NN, NL, MN, lx, ly, lz, x, y, z, fx, fy, fz, vx, vy, vz, hc);

// equilibration
clock_t time_begin = clock();
for (int step = 0; step < Ne; ++step)
{
    integrate(N, time_step, m, fx, fy, fz, vx, vy, vz, x, y, z, 1);
    find_force
    (N, NN, NL, MN, lx, ly, lz, x, y, z, fx, fy, fz, vx, vy, vz, hc);
    integrate(N, time_step, m, fx, fy, fz, vx, vy, vz, x, y, z, 2);
    scale_velocity(N, T_0, m, vx, vy, vz); // control temperature
}
clock_t time_finish = clock();
double time_used = (time_finish - time_begin) / (double) CLOCKS_PER_SEC;
fprintf(stderr, "time use for equilibration = %f s\n", time_used);

// production
time_begin = clock();
int count = 0;
for (int step = 0; step < Np; ++step)
{
    integrate(N, time_step, m, fx, fy, fz, vx, vy, vz, x, y, z, 1);
    find_force
    (N, NN, NL, MN, lx, ly, lz, x, y, z, fx, fy, fz, vx, vy, vz, hc);
    integrate(N, time_step, m, fx, fy, fz, vx, vy, vz, x, y, z, 2);
    if (0 == step % Ns)
    { hx[count] = hc[0]; hy[count] = hc[1]; hz[count] = hc[2]; count++; }
}
time_finish = clock();
time_used = (time_finish - time_begin) / (double) CLOCKS_PER_SEC;
fprintf(stderr, "time use for production = %f s\n", time_used);

// calculate HCACF and thermal conductivity
find_hac_kappa(Nd, Nc, time_step * Ns, T_0, lx * ly * lz, hx, hy, hz);

free(NN); free(NL); free(m);  free(x);  free(y);  free(z);
free(vx); free(vy); free(vz); free(fx); free(fy); free(fz);
free(hx); free(hy); free(hz);
```

```
    system("PAUSE"); // For DEV-C++ in Windows;

    return 0;
}
```

编译上述程序后，可以运行多次。每次运行的结果将追加到同一个输出文件。可用如下 MATLAB 脚本分析结果。我得到的结果见图 1。得到的结果可报道为 $\kappa = 0.28 \pm 0.02$ $\mathrm{Wm^{-1}K^{-1}}$。

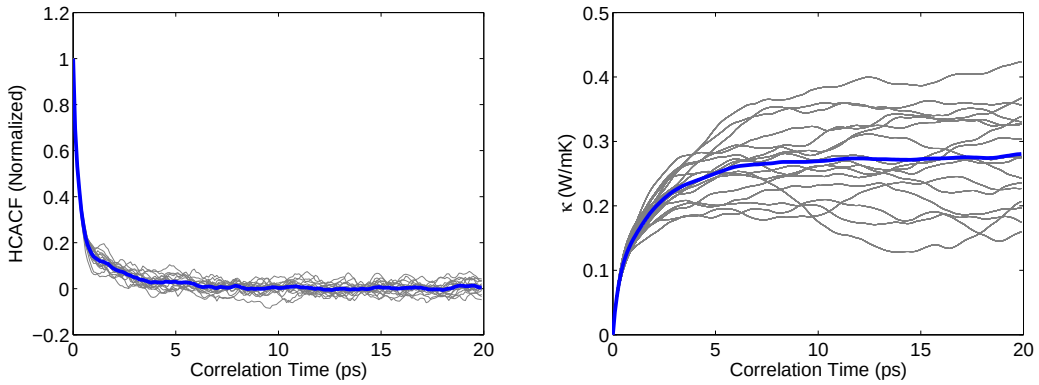

Figure 1: 左图：归一化的热流自关联函数。右图：跑动热导率。细线代表每个独立计算的结果；粗线代表平均结果。模拟体系为 60 K 的固态氩。计算程序为自编 C程序。

```
clear;close all; font_size=15;
load kappa.txt; % Data from the C code

% number of correlation data for one simulation;
% should be consistent with the calculations using the C code
M = 200;

% I usually use 10-100 simulations to get high-quality data
number_of_simulations = size(kappa, 1) / M

t = kappa(1 : M, 1); % correlation time

% For isotropic 3D systems, k = (kx + ky + kz) / 3
hac = reshape(mean(kappa(:, 2:4), 2), M, number_of_simulations);
rtc = reshape(mean(kappa(:, 5:7), 2), M, number_of_simulations);

% ensemble average
hac_ave = mean(hac, 2);
```

```
rtc_ave = mean(rtc, 2);

% Normalized heat current auto-correlation function
figure
for n = 1 : number_of_simulations
    plot(t, hac(:, n) / hac(1, n), 'color', 0.5 * [1 1 1]);
    hold on;
end
plot(t, hac_ave ./ hac_ave(1), 'linewidth', 3);
xlabel('Correlation Time (ps)', 'fontsize', font_size);
ylabel('HCACF (Normalized)', 'fontsize', font_size);
set(gca,'fontsize', font_size);

% Running thermal conductivity
figure
for n = 1 : number_of_simulations
    plot(t, rtc, 'color', 0.5 * [1 1 1]);
    hold on;
end
plot(t, rtc_ave, 'linewidth', 3);
xlabel('Correlation Time (ps)', 'fontsize', font_size);
ylabel('\kappa (W/mK)', 'fontsize', font_size);
set(gca,'fontsize', font_size);

% Average over an appropriate time block after visual inspection
kappa_converged = mean(rtc(M/2+1 : M, :));

% Report the mean value and an error estimate (standard error)
kappa_average = mean(kappa_converged)
kappa_error = std(kappa_converged) / sqrt(number_of_simulations)
```

4.2 一个用 EMD 方法计算固态氩热导率的 LAMMPS 脚本

下面是一个用 EMD 方法计算固态氩热导率的 LAMMPS 脚本。

```
# 用Green-Kubo方法计算固态氩热导率的LAMMPS脚本

# 采用metal单位系统（记住：能量eV；长度：A；时间：ps）
units       metal

# 定义几个参数
variable    T  equal 60      # 温度为 60 K
variable    A  equal 5.4     # 晶格常数大概为 5.4 A
variable    DT equal 0.01    # 积分步长为 10 fs
```

```
# 初始化坐标、势函数、速度等数据
lattice      fcc ${A}               # 固态氩具有面心立方结构
region       BOX block 0 4 0 4 0 4  # 一共有 4*4*4*4=256 原子
create_box   1 BOX                  # 先造一个盒子
create_atoms 1 box                  # 再填满原子
mass         1 40                   # 质量就是原子量
pair_style   lj/cut 10.0            # LJ 势的截断半径取10 A
pair_coeff   * * 1.032e-2 3.405     # epsilon 和 sigma
velocity     all create ${T} 12345  # 根据温度初始化速度

# 在NVT系综下平衡体系
fix          NVT all nvt temp ${T} ${T} 1 # Nose-Hoover 热浴
timestep     ${DT}                        # 设置积分步长
thermo_style custom step temp press       # 观察温度和压强的变化
thermo       1000                         # 1000 步输出一次

# 根据以上设置跑若干步，目的是使系统达到热力学平衡态
run          20000                        # 跑 20000 步

# 在产出阶段将用NVE系综
unfix          NVT            # 撤掉之前的 NVT 系综
fix            NVE all nve    # 换上 NVE 系综

# 在产出阶段计算热流
compute      KE all ke/atom                # 计算单原子动能
compute      PE all pe/atom                # 计算单原子势能
compute      V all stress/atom NULL virial # 计算单原子位力
compute      J all heat/flux KE PE V       # 计算单原子热流

# 在产出阶段计算热流自关联函数 <J(0)J(t)>
variable    Ns equal 10             # 取样间隔
variable    Nc equal 200            # 关联数据量（最大关联时间为 Ns*Nc*DT）
variable    Np equal ${Ns}*${Nc}*10 # 产出步数（这是一个好的选择）
fix         HAC all ave/correlate ${Ns} ${Nc} ${Np} c_J[1] c_J[2] c_J[3] &
            type auto file hac.txt

variable    M equal 10          # 热流自关联函数的个数
variable    N equal ${M}*${Np} # 产出步数

# 跑产出阶段
run         ${N}
```

运行上述 LAMMPS 脚本后用手动的方式或通过其它程序改变输出文件格式（具

体细节课堂讲解）。然后，可用如下 MATLAB 脚本分析结果。我得到的结果见图 2。得到的结果可报道为 $\kappa = 0.30 \pm 0.02$ $\mathrm{Wm^{-1}K^{-1}}$，与前面用自编程序算出的结果在误差范围内是一致的。

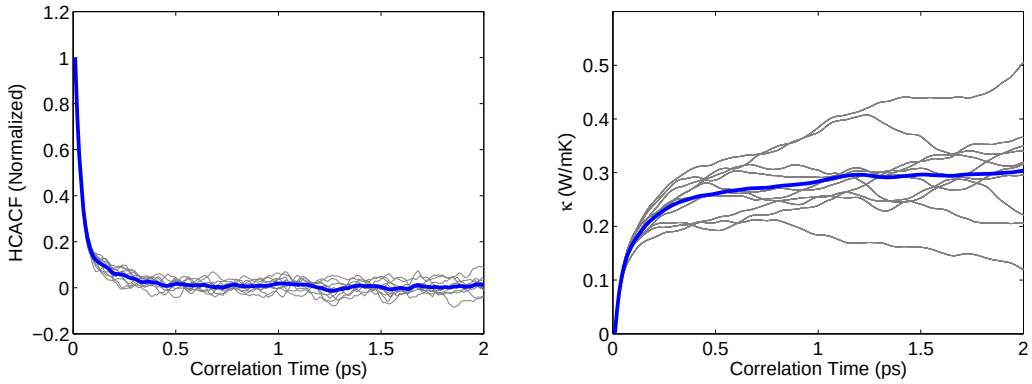

Figure 2: 左图：归一化的热流自关联函数。右图：跑动热导率。细线代表每个独立计算的结果；粗线代表平均结果。模拟体系为 60 K 的固态氩。计算程序为 LAMMPS。

```
clear;close all; font_size=15;

file ='hac.txt';

% Nc = 一次模拟的关联数据量
[Tstart Nc] = textread(file, '%n%n', 1, 'headerlines', 3);
[index TimeDelta Ncount c_1 c_2 c_3] ...
    = textread(file, '%n%n%n%n%n%n', 'headerlines', Nc+4, 'emptyvalue', 0);
Ns = TimeDelta(3,1) - TimeDelta(2, 1);  % Ns = 取样间隔
hac = [index TimeDelta Ncount c_1 c_2 c_3];
nall = length(hac) / (Nc+1);
clear index TimeDelta Ncount c_1 c_2 c_3;
for i=1:nall
    hac((i-1)*Nc+1, :) = [];
end

%根据MD模拟确定一些参数
dt = 0.01;       % 积分步长，单位为 ps
T = 60;          % 单位为 K
kB = 8.617e-5;   % 玻尔兹曼常数，单位为 eV/K
a = 5.4;         % 晶格常数，单位为 A
V = (4*a)^3;     % 体积
scale = 1.6e+3 / (kB*T*T*V) * (Ns*dt); % 转换因子
```

```
% 自动确定模拟次数
M = size(hac, 1) / Nc

% 关联时间：
t = (1 : Nc) * dt; % correlation time

% 计算三维系统的热流自关联和跑动热导率
hac = reshape(mean(hac(:, 4:6), 2), Nc, M);
rtc =  scale * cumtrapz(hac);

% 系综平均（对独立模拟的平均）
hac_ave = mean(hac, 2);
rtc_ave = mean(rtc, 2);

% 画出归一化的热流自关联
figure
for n = 1 : M
    plot(t, hac(:, n) / hac(1, n), 'color', 0.5 * [1 1 1]);
    hold on;
end
plot(t, hac_ave / hac_ave(1), 'linewidth', 3);
xlabel('Correlation Time (ps)', 'fontsize', font_size);
ylabel('HCACF (Normalized)', 'fontsize', font_size);
set(gca,'fontsize', font_size);

% 画出跑动热导率
figure
for n = 1 : M
    plot(t, rtc, 'color', 0.5 * [1 1 1]);
    hold on;
end
plot(t, rtc_ave, 'linewidth', 3);
xlabel('Correlation Time (ps)', 'fontsize', font_size);
ylabel('\kappa (W/mK)', 'fontsize', font_size);
ylim([0, 0.6]);
set(gca,'fontsize', font_size);

% 报道结果
kappa_converged = mean(rtc(Nc/2+1 : Nc, :)); %具体问题具体分析
kappa_average = mean(kappa_converged)
kappa_error = std(kappa_converged) / sqrt(M)
```

4.3 一个用NEMD方法计算固态氩热导率的 LAMMPS脚本

```
# 用速度重标法计算固态氩热导率的LAMMPS脚本

# 采用metal单位系统（记住：能量eV；长度：A；时间：ps）
units       metal

# 定义几个参数
variable    T     equal 60   # 温度为 60 K
variable    A     equal 5.4  # 晶格常数大概为 5.4 A
variable    DT    equal 0.01 # 积分步长为 10 fs
variable    POWER equal 0.05 # 与热流对应的功率 0.05 eV/ps

# 初始化坐标、势函数、速度等数据
lattice      fcc ${A}               # 固态氩具有面心立方结构
region       BOX block 0 20 0 4 0 4 # 一共有 20*4*4*4=1280 原子
create_box   1 BOX                  # 先造一个盒子
create_atoms 1 box                  # 再填满原子
mass         1 40                   # 质量就是原子量
pair_style   lj/cut 10.0            # LJ 势的截断半径取10 A
pair_coeff   * * 1.032e-2 3.405     # epsilon 和 sigma
velocity     all create ${T} 12345  # 根据温度初始化速度

# 在NVT系综下平衡体系
fix          NVT all nvt temp ${T} ${T} 1 # Nose-Hoover 热浴
timestep     ${DT}                        # 设置积分步长
thermo_style custom step temp press       # 观察温度和压强的变化
thermo       1000                         # 1000 步输出一次

# 根据以上设置跑若干步，目的是使系统达到热力学平衡态
run          20000                        # 跑 20000 步

# 定义热源（heat source）和热汇（heat sink）
region          SOURCE block  0   1 INF INF INF INF #  第 1个block是热源
region          SINK   block  10 11 INF INF INF INF #  第11个block是热汇

# 在热源注入热量，在热汇导出热量，使系统达到非平衡稳态
unfix           NVT             # 撤掉之前的 NVT 系综
fix             NVE all nve     # 换上 NVE 系综
fix             HEAT_IN   all ehex 1  ${POWER} region  SOURCE # 注入热量
fix             HEAT_OUT  all ehex 1 -${POWER} region  SINK   # 导出热量
# 可将ehex换成heat。使用前者能量守恒满足得更好。

# 计算单原子“温度”
compute    KE all ke/atom           # 单原子动能
variable   KB equal 8.625e-5        # 玻尔兹曼常数，eV/K
variable   TEMP atom c_KE/1.5/${KB} # 单原子温度
```

```
# 下面两个命令是联合起来计算块温度（block temperatures）的。
# 首先定义一个计算，将体系在x方向平均分为1/0.05=20块，
# 然后计算每个块的温度的时间平均，将结果记录到文件。
compute BLOCKS all chunk/atom bin/1d x lower 0.05 units reduced
fix     T_PROFILE all ave/chunk 10 1000 10000 BLOCKS v_TEMP file temp.txt

# 在施加热流的情况下跑若干步
run 100000
```

运行上述 LAMMPS 脚本后用手动的方式或通过其它程序改变输出文件格式（具体细节课堂讲解）。然后，可用如下 MATLAB 脚本分析结果。我得到的结果见图 3。得到的结果可报道为 $\kappa = 0.26 \pm 0.02$ $\mathrm{Wm^{-1}K^{-1}}$。注意：这个热导率是针对长度为 11 nm 的体系的。固态氩在 60 K 的声子平均自由程远小于该体系长度，所以该热导率已经很接近 EMD 方法得到的结果。

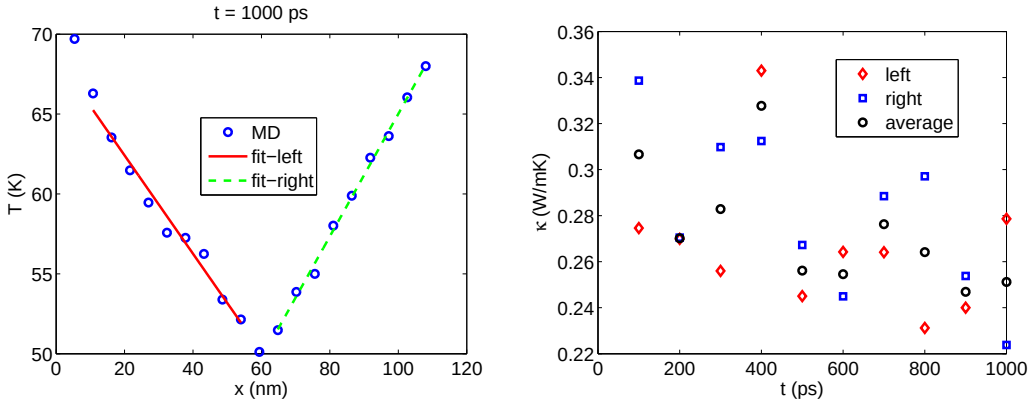

Figure 3: 左图：温度分布。右图：热导率随模拟时间点的变化。模拟体系为 60 K 的 11 nm 长的固态氩。计算程序为 LAMMPS。

```
clear; close all; font_size=15;

file = 'temp.txt';

%根据MD模拟确定一些参数
num_blocks = 20;          % 块的个数
num_outputs = 10;         % 输出块温度的次数
Nx = 20;                  % 输运方向晶胞个数
a = 5.4;                  % 晶格常数，单位为 A
Lx = a * Nx;              % 输运方向长度
dx = Lx / num_blocks;     % 每个块的长度
power = 0.05 * 1.6e-7;    % 功率 eV/ps -> W
```

```
A = (4*a)^2 * 1.0e-20;  % 横截面积 A^2 -> m^2
Q = power / A / 2;      % 热流密度 W/m^2
t0 = 100;               % 每次输出数据增加的模拟时间，单位为  ps

% 非平衡模拟时间
t= t0 * (1 : num_outputs);

% 块坐标
x = (1 : num_blocks) * dx; % 一行
x = x.';                   % 一列

%从LAMMPS输出文件中提取块温度
[Chunk Coord1 Ncount v_TEMP] ...
    = textread(file, '%n%n%n%n', 'headerlines', 3, 'emptyvalue', 0);
temp = [Chunk Coord1 Ncount v_TEMP]; nall = length(temp) / (num_blocks+1);
clear Chunk Coord1 Ncount v_TEMP;
for i =1:nall
    temp((i-1)*num_blocks+1,:)=[];
end
temp = reshape(temp(:, end), num_blocks, num_outputs);

% 对输出次数作循环，计算温度梯度和热导率
for n = 1 : num_outputs

    % 确定拟合区间（具体问题具体分析）
    index_1 = 2 : num_blocks/2;              % 左边除去热源后的块指标
    index_2 = num_blocks/2+2 : num_blocks;   % 右边除去热汇后的块指标

    % 拟合温度分布
    p1 = fminsearch(@(p) ...
        norm(temp(index_1, n) - (p(1)-p(2)*x(index_1)) ), [60, -1]);
    p2 = fminsearch(@(p) ...
        norm(temp(index_2, n) - (p(1)+p(2)*x(index_2)) ), [60, 1]);

    % 得到温度梯度
    gradient_1 = p1(2) * 1.0e10; % K/A -> K/m
    gradient_2 = p2(2) * 1.0e10; % K/A -> K/m

    % 得到热导率
    kappa_1(n) = Q / gradient_1
    kappa_2(n) = Q / gradient_2

    % 画温度分布以及拟合曲线
    figure;
    plot(x, temp(:, n), 'bo', 'linewidth', 2);
```

```
    hold on;
    x1=x(index_1(1)) : 0.1 : x(index_1(end));
    x2=x(index_2(1)) : 0.1 : x(index_2(end));
    plot(x1, p1(1)-p1(2)*x1, 'r-', 'linewidth', 2);
    plot(x2, p2(1)+p2(2)*x2, 'g--', 'linewidth', 2);
    legend('MD', 'fit-left', 'fit-right');
    set(gca, 'fontsize', font_size);
    title (['t = ', num2str(t0 * n), ' ps']);
    xlabel('x (nm)');
    ylabel('T (K)');
end

% 热导率的时间收敛测试
kappa = (kappa_1 + kappa_2) / 2;
figure;
plot(t, kappa_1, ' rd', 'linewidth', 2);
hold on;
plot(t, kappa_2, ' bs', 'linewidth', 2);
plot(t, kappa, ' ko', 'linewidth', 2);
set(gca, 'fontsize', font_size);
xlabel('t (ps)');
ylabel('\kappa (W/mK)');
legend('left', 'right', 'average');

% 报道结果（具体问题具体分析）
kappa_average = mean( kappa(5:end) )
kappa_error   = mean( 0.5*abs( kappa_1(5:end)-kappa_2(5:end) ) )
```

References

[1] M. S. Green. Markoff random processes and the statistical mechanics of timedependent phenomena. II. Irreversible processes inuids, The Journal of Chemical Physics, **22**, 398, (1954).

[2] R. Kubo, Statistical-mechanical theory of irreversible processes. I. General theory and simple applications to magnetic and conduction problems, Journal of the Physical Society of Japan, **12**, 570, (1957).

[3] J. M. Haile, *Molecular Dynamics Simulation: Elementary Methods*, John Wiley Sons, Inc., New York, (1992).

[4] Z. Fan, L. F. C. Pereira, H.-Q. Wang, J.-C. Zheng, D. Donadio, and A. Harju, Force and heat current formulas for many-body potentials in molecular dynamics

simulations with applications to thermal conductivity calculations, Physical Review B **92**, 094301 (2015).

[5] F. Müller-Plathe, A simple nonequilibrium molecular dynamics method for calculating the thermal conductivity, Journal of Chemical Physics **106**, 6082 (1997).

[6] T. Ikeshoji and B. Hafskjold, Non-equilibrium molecular dynamics calculation of heat conduction in liquid and through liquid-gas interface, Molecular Physics **81**, 251 (1994).

[7] P. Jund and R. Jullien, Molecular-dynamics calculation of the thermal conductivity of vitreous silica, Physical Review B **59**, 13707 (1999).

[8] S. Plimpton, Fast Parallel Algorithms for Short-Range Molecular Dynamics, Journal of Computational Physics, **117**, 1 (1995). http://lammps.sandia.gov.

[9] Z. Fan, W. Chen, V. Vierimaa, and Ari Harju. Efficient molecular dynamics simulations with many-body potentials on graphics processing units, Computer Physics Communications **218**, 10 (2017).

[10] Z. Fan, L. F. C. Pereira, P. Hirvonen, M. M. Ervasti, K. R. Elder, D. Donadio, T.o Ala-Nissila, and A. Harju, Thermal conductivity decomposition in two-dimensional materials: Application to graphene, Physical Review B **95**, 144309 (2017).

[11] Z. Fan, T. Siro, and A. Harju, Accelerated molecular dynamics force evaluation on graphics processing units for thermal conductivity calculations, Computer Physics Communications, **184**, 1414, (2013).