第八章 分子动力学模拟

从分子层次研究物理系统的重要方面是对其结构和动力学行为进行表征。目前的实验技术很难同时从空间和时间上给出高分辨的测量，而分子动力学模拟能够自然地给出关于物理体系结构与动力学的精确表征，因此在物理、化学、以及生物分子研究中发挥着重要作用。本章将介绍分子动力学模拟的基本原理及其实现。

8.1 分子动力学的基本原理

描述分子体系运动的薛定谔方程如下：

$$H\Psi = E\Psi \quad (8\text{-}1)$$

其中，H和Ψ为包含了原子核和电子自由度的哈密顿量和波函数。由于原子核的质量远大于电子质量，因此原子核与电子运动的时间尺度具有很大的差别，可以分开处理。其中，在处理电子的运动时，原子核位置可以看作是固定，即玻恩-奥本海默绝热近似。由于原子核质量大，量子效应可以忽略，可由经典牛顿方程描述其运动。因此整个分子体系的运动可以表示为：

$$H_e\Psi_e = E_e\Psi_e \quad (8\text{-}2)$$

和

$$F_n = m_n a_n \quad (8\text{-}3)$$

其中，H_e和Ψ_e是电子的哈密顿量和波函数。F_n和a_n是原子核的受力和加速度。其中，原子核的受力由电子状态确定的能量面给出。这种电子自由度由量子力学方法处理，而原子核自由度由经典力学处理的方法称为量子分子动力学或第一性原理分子动力学。然而，基于量子力学方法计算绝热能量面仍然涉及复杂的计算量，从而大大限制了分子模拟能够达到的时间尺度。人们发现，绝热能量面通常能够以参数化的方式给出，即人们可针对不同的原子类型，事先通过第一性原理和实验的方法确定，并作为通用分子力场使用。这种分子力场模型可以避免在分子动力学模拟中求解电子运动的量子力学方程，极大地提高了分子模拟效率。人们通常把这类基于分子力场的分子动力学模拟方法称为经典分子动力学（简称分子动力学），是本章学习的内容。

8.2 分子动力学的基本步骤

分子动力学模拟的基本思路是根据系统中粒子的受力情况，求解牛顿方程，得到粒子坐标和动量随时间的演化，即相空间中的运动轨迹，进而基于统计物理理论得到体系的热力学量和动态行为等性质。分子动力学模拟包含如下步骤：1）初始化，2）计算力，3）积分运动方程，4）计算物理量。本节以二维原子体系为例，介绍分子动

力学模拟的基本步骤。其中，原子质量为 m,模拟体系的原子个数为 N，分布在边长为 L 的二维正方形盒子内。原子之间的相互作用为 6-12 形式的 lennard-Jones(L-J)势：

$$U(r_{ij}) = 4\epsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \quad (8\text{-}4)$$

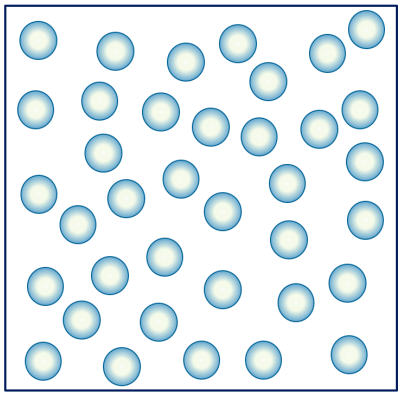

图 8.1 二维原子体系示意图。

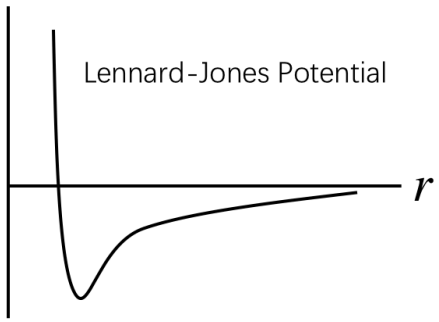

图 8.2 6-12 形式的 lennard-Jones 示意图。

牛顿方程是以二阶常微分方程给出的，因此求解牛顿方程需要给定系统所有粒子的初始位置和速度，此即为初始化过程。初始位置的设定往往依赖于具体模拟体系。对于液态或气态体系，在给定范围内随机抽取位置是通常的做法。对于已知结构的晶体或生物大分子体系，则可以通过结构数据设定初始位置。初始速度可以根据麦克斯韦速度分布律随机抽样给出。Code-8-1 中的子函数 initialposvel()给出了二维原子系统初始位置和初始速度的设置过程。

初始化完成后，需要计算每个原子的受力，这也是分子动力学模拟的最为核心的步骤，也是最为耗时的步骤。人们对分子动力学的改进大多是从如何精确而高效地计算原子受力入手。根据模拟的体系不同，原子间的受力可以很不相同。对于诸如惰性气体原子组成的简单原子体系，主要的相互作用为范德瓦尔斯力，图 8.2 给出的 L-J 势能够很好地描述这种短程排斥，长程吸引的范德瓦尔斯力。如果模拟的体系由带电粒子组成，则还需要考虑静电相互作用力。而有机分子等多原子分子系统，原子之间存在共价键。对这类体系的描述需要引入与原子杂化环境相关的成键相互作用项，如键长、键角、二面角相关的局域相互作用项。由于经典分子动力学的能量函数是以有效力场的形式给出的，因此在使用分子动力学模拟之前，需要建立原子间相互作用的经典力场参数，如原子电荷、L-J 参数等。这些参数通常是基于量子力学方法或实验数据拟合确定的。目前，对常见的分子体系，都有成熟的分子力场供使用，且有专门的研究团队致力于发展高精度的分子力场供人们使用。例如，对水分子体系，常见的分子力场有 TIP3P，TIP4P，SPC/E 等。对生物大分子体系，常用的力场有 AMBER， CHARMM，OPLS 等。

分子动力学模拟能够包含的粒子数是有限的。依赖于计算资源，典型的模拟体系所包含的粒子数数在 10^2-10^6 之间，对应的模拟体系尺寸通常在亚微米量级，相对于宏观体系是非常小的。因此，人们通常设定周期性边界条件来克服模拟体系的有限尺寸效应（图 8.3）。在周期性边界条件下，真实模拟体系通过平移操作，向 x,y,z 方向复制延伸到无限大体系，等效于粒子从一条边离开真实模拟体系，必然从对面的另一条边以相同的速度进入模拟体系，从而维持体系的总粒子数不变。在设定的截断距离范围内(绿色圆圈)，一个给定的粒子 i 不仅与真实模拟体系中的粒子有相互作用，而且与镜像模拟体系中的粒子有相互作用。

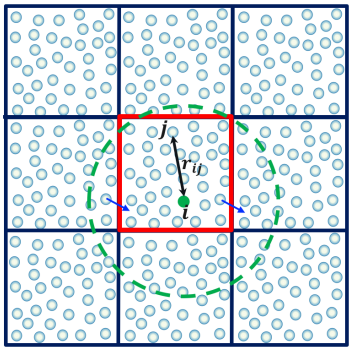

图 8.3 周期性边界条件示意图。

设定边界条件后，便可以计算截断距离内的原子对之间的距离，从而计算相互作用能和原子受力。对于相互为 L-J 势的简单原子体系，系统的总势能为：

$$E(\boldsymbol{R}) = \sum_{i<j}^{N} U(r_{ij}) = \sum_{i<j}^{N} 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] \quad (8\text{-}5)$$

其中 $\boldsymbol{R}$ 为所有原子位置坐标的集合。原子 i 在 x 和 y 方向的受力可写为：

$$\begin{aligned} f_{ix} &= -\frac{\partial E(\boldsymbol{R})}{\partial x_i} = -\sum_{i<j}^{N} \frac{\partial U(r_{ij})}{\partial x_i} = -\frac{48(x_j - x_i)}{r_{ij}^2} \sum_{j\neq i}^{N} \left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - 0.5\left(\frac{\sigma}{r_{ij}}\right)^{6} \right] \\ f_{iy} &= -\frac{\partial E(\boldsymbol{R})}{\partial y_i} = -\sum_{i<j}^{N} \frac{\partial U(r_{ij})}{\partial y_i} = -\frac{48(y_j - y_i)}{r_{ij}^2} \sum_{j\neq i}^{N} \left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - 0.5\left(\frac{\sigma}{r_{ij}}\right)^{6} \right] \end{aligned} \quad (8\text{-}6)$$

Code-8-1 中的子函数 force()给出了原子受力的计算。

确定了每个粒子的受力后，在可以求解如下牛顿方程得到粒子坐标和速度的时间演化；

$$\begin{aligned} m_i \frac{d^2 x_i}{dt^2} &= f_{ix} \\ m_i \frac{d^2 y_i}{dt^2} &= f_{iy} \end{aligned} \quad (8\text{-}7)$$

以上二阶常微分方程可以进一步写为一阶常微分方程组的形式：

$$\begin{aligned} m_i \frac{dv_i}{dt} &= f_{ix} \\ \frac{dx_i}{dt} &= v_i \end{aligned} \quad (8\text{-}8)$$

通常地可以写出 y 方向的一阶微分方程组。式（8-8）可通过有限差分法求解，常用的方法有 verlet 算法、速度-verlet 算法以及 leapfrog 算法等。

设粒子 i 在 t 时刻的 x 坐标为$x_i(t)$，在将x_i在$t+\Delta t$时刻的值在t时刻附近作泰勒展开得：

$$x_i(t+\Delta t) = x_i(t) + v_i(t)\Delta t + \frac{f_{ix}(t)}{2m_i}\Delta t^2 + O(\Delta t^3) \quad (8\text{-}9)$$

其中，Δt为分子动力学模拟的时间步长。类似地，将x_i在$t-\Delta t$时刻的值在t时刻附近作泰勒展开得：

$$x_i(t-\Delta t) = x_i(t) - v_i(t)\Delta t + \frac{f_{ix}(t)}{2m_i}\Delta t^2 - O(\Delta t^3) \quad (8\text{-}10)$$

将式(8-9)与式(8-10)相加可得：

$$x_i(t+\Delta t) = 2x_i(t) - x_i(t-\Delta t) + \frac{f_{ix}(t)}{m_i}\Delta t^2 \quad (8\text{-}11)$$

上式即为求解运动方程的 **verlet 算法**。基于 verlet 算法，计算下一时刻的位置，需要当前时刻和上一时刻的位置信息以及当前时刻的受力。由于以上两式相加运算将$O(\Delta t^3)$抵消，因此式（8-11）给出的误差具有$O(\Delta t^4)$阶，从而给出关于位置演化的高精度估算。Verlet 算法不涉及速度，因此需要通过如下差分法由坐标估算速度：

$$v_i(t) = \frac{x_i(t+\Delta t)-x_i(t-\Delta t)}{2\Delta t} + O(\Delta t^2) \quad (8\text{-}12)$$

式（8-12）给出的速度估算精度较差，这也是 verlet 算法最大的缺陷。

为了克服 verlet 算法的缺陷，人们引入了速度-verlet 算法。类似于式（8-9），可以将$x_i(t+2\Delta t)$写为：

$$x_i(t+2\Delta t) = x_i(t+\Delta t) + v_i(t+\Delta t)\Delta t + \frac{f_{ix}(t+\Delta t)}{2m_i}\Delta t^2 + O(\Delta t^3) \quad (8\text{-}13)$$

式（8-13）与式（8-9）相减可得：

$$x_i(t+2\Delta t) + x_i(t) = 2x_i(t+\Delta t) + [v_i(t+\Delta t) - v_i(t)]\Delta t + \frac{f_{ix}(t+\Delta t) - f_{ix}(t)}{2m_i}\Delta t^2 + O(\Delta t^3)$$

(8-14)

对比式（8-14）与式（8-11）可得：

$$v_i(t+\Delta t) = v_i(t) + \frac{f_{ix}(t)+f_{ix}(t+\Delta t)}{2m_i}\Delta t \quad (8\text{-}15)$$

式（8-15）与式（8-9）一起组成了积分运动方程的**速度-verlet 算法**，即：

$$\begin{aligned} x_i(t+\Delta t) &= x_i(t) + v_i(t)\Delta t + \frac{f_{ix}(t)}{2m_i}\Delta t^2 \\ v_i(t+\Delta t) &= v_i(t) + \frac{f_{ix}(t)+f_{ix}(t+\Delta t)}{2m_i}\Delta t \end{aligned} \quad (8\text{-}16)$$

对照前面学习过的微分方程求解算法，以上速度求解使用了改进的欧拉算法。

另一类常用的积分运动方程的算法是 **Leapfrog 算法**。在 leapfrog 算法中，速度的计算与位置的计算不同步，位置的计算发生在整数时间步，而速度的计算发生在半整数时间步，即：

$$x_i(t+\Delta t) = x_i(t) + v_i(t+\Delta t/2)\Delta t$$
$$v_i(t+\Delta t/2) = v_i(t-\Delta t/2) + \frac{f_{ix}(t)}{m_i}\Delta t \quad (8\text{-}17)$$

由 Leapfrog 算法可以实现对运动方程的高效率求解，同时保持合理的精确度。但是由于速度和位置的计算不同步，不能直接计算总能量。

Code-8-1 中的子函数 integrate()是基于速度-verlet 算法实现对二维简单原子体系的求解过程。基于位置和速度的时间演化信息，可以计算其他物理量，包括时间相关的动态演化过程以及通过统计平均得到的热力学量。例如，运行 code-8-1 可以得到势能、动能、以及总能量的时间演化，如图 8.4 所示，且二维简单原子将弛豫到更稳定的结构（图 8.5）。

```python
# -*- coding: utf-8 -*-
import numpy as np
import random
import matplotlib.pylab as pl
import matplotlib.animation as animation

Natom = 36 #原子数
NT = 1000 #最大时间步数
Tinit = 0.5 #初始温度
eps = 1.0 #势阱
x = np.zeros(Natom) #坐标
y = np.zeros(Natom)
vx = np.zeros(Natom) #速度
vy = np.zeros(Natom)
fx = np.zeros([Natom,2]) #力， 1）t时刻 2）t+dt时刻
fy = np.zeros([Natom,2])
L = int(1.0*Natom**0.5) #盒子边长

tt = np.arange(NT)  #时间
xc = np.zeros([Natom,NT+1]) #位置-时间
yc = np.zeros([Natom,NT+1])
EP = np.zeros(NT)  #势能-时间
EK = np.zeros(NT) #动能-时间—
ET = np.zeros(NT) #总能-时间

def initialposvel(): #初始化
    i = -1
    for ix in range(L): #按格点摆放
        for iy in range(L):
            i = i + 1
            x[i] = ix
            y[i] = iy
            vx[i] = random.gauss(0,1)
            vy[i] = random.gauss(0,1)
            vx[i] = vx[i] * np.sqrt(Tinit)
            vy[i] = vy[i] * np.sqrt(Tinit)

def forces(t): #计算力
    r2cut = 9 #截断距离平方
    PE = 0.0 #势能置零
    for i in range(0,Natom): #力置零
        fx[i][t] = 0.0
        fy[i][t] = 0.0
    for i in range(0,Natom-1):
        for j in range(i+1,Natom):
            dx = x[i] - x[j] #计算距离
            dy = y[i] - y[j]
            if (dx > 0.5*L): #周期性边界条件
                dx = dx - L
            if (dx < -0.5*L):
                dx = dx + L

            if (dy > 0.5*L):
                dy = dy - L
            if (dy < -0.5*L):
                dy = dy + L

            r2 = dx*dx + dy*dy
            if(r2 < r2cut): #截断
                invr2 = 1.0/r2
                invr6 = invr2**3
                wij = 48*eps*invr2*invr6*(invr6-0.5)
                fijx = wij*dx
                fijy = wij*dy
                fx[i][t] = fx[i][t] + fijx
                fy[i][t] = fy[i][t] + fijy
                fx[j][t] = fx[j][t] - fijx
                fy[j][t] = fy[j][t] - fijy
                PE = PE + 4.0*eps*(invr6)*(invr6-1)
    return PE

def timevolution(): #时间演化
    t1 = 0
    t2 = 1
    h = 0.01 #时间步长
    hover2 = h/2.0
    initialposvel() #调用初始化
    PE = forces(t1) #计算力与势能
    for it in np.arange (NT): #时间循环
        if np.mod(it,100) == 0:
            print('it=',it)
        PE = forces(t1) #计算力与势能
        for i in range(0,Natom):

            x[i] = x[i] + h*(vx[i] + hover2*fx[i][t1]) #速度verlet更新位置
            y[i] = y[i] + h*(vy[i] + hover2*fy[i][t1])
            if x[i] <= 0: #周期边界
                x[i]=x[i] + L
            if x[i] > L:
                x[i]=x[i] - L
            if y[i] <= 0:
                y[i]=y[i] + L
            if y[i] > L:
                y[i]=y[i] - L
            xc[i][it] = x[i] #存储位置
            yc[i][it] = y[i]

        PE = forces(t2) #计算势能与力
        KE = 0.0
        for i in range(0, Natom):
            vx[i] = vx[i] + hover2*(fx[i][t1] + fx[i][t2]) #速度verlet更新速度
            vy[i] = vy[i] + hover2*(fy[i][t1] + fy[i][t2])
            KE = KE + (vx[i]*vx[i] + vy[i]*vy[i])/2 #计算动能

        EP[it] = PE #存储势能
        EK[it] = KE #存储动能
        ET[it] = PE + KE #存储总能量

timevolution()

def init():
    d.set_data([], [])
    return d,

def update_line(num, xc,yc,dot):
    dot.set_data(xc[:,num],yc[:,num])
    return dot,

fig1 = pl.figure()
d, = pl.plot([], [], 'ro',markersize=30)
pl.xlim(-0.5, 6.5)
pl.ylim(-0.5, 6.5)
pl.xlabel('X')
pl.ylabel('Y')
pl.title('MD')
dot_ani = animation.FuncAnimation(fig1, update_line, np.arange(1000),\
fargs=(xc,yc,d),interval=20, init_func=init, blit=True)

fig2 = pl.figure()
pl.plot(tt,EP,'k-')
pl.plot(tt,EK,'r-')
pl.plot(tt,ET,'b-')
pl.xlabel('time')
pl.ylabel('E')
pl.title('MD')
pl.show()
```

Code-8-1

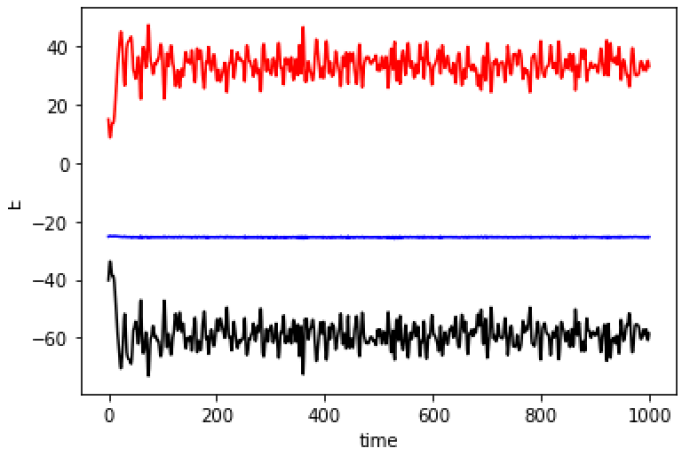

图 8.4 势能（黑色）、动能（红色）、以及总能量（蓝色）的时间演化。

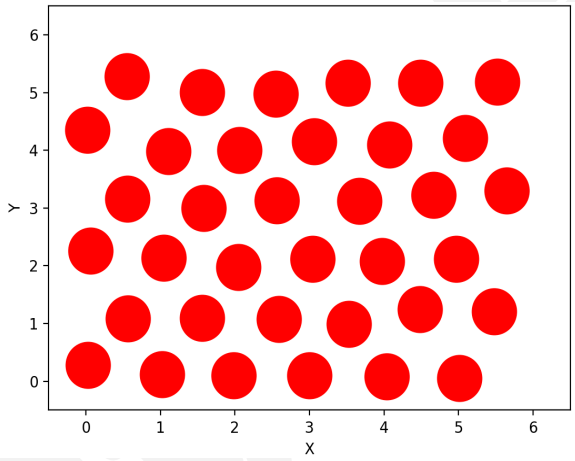

图 8.5 低温模拟得到的二维简单原子体系的代表性结构。

8.3 从分子动力学到物理观测量

通过分子动力学模拟，在给定初始条件的情况下，我们可以跟踪系统中每个粒子的速度与坐标随时间的演化。由于粒子的运动是通过求解牛顿方程来得到的，因此时间演化轨迹是物理的，从而给出从微观上观测粒子运动的一种途径，也是目前唯一可行的途径。基于分子模拟轨迹，还可以进一步提取其他实验上可以直接测量的物理量，如扩散系数、振动频率等，并与实验比较，检验理论模型或解释实验数据。上述二维简单原子体系，扩散系数可以计算如下：

$$D = \lim_{t\to\infty} \frac{1}{6N_m t} \left\langle \sum_{j=1}^{N_m} [\boldsymbol{r}_j(t) - \boldsymbol{r}_j(0)]^2 \right\rangle \tag{8-18}$$

其中，N_m为粒子数，$r_j(t)$和$r_j(0)$为粒子 j 在 t 时刻和 0 时刻的位置。扩散系数还可以通过速度自关联函数求的：

$$D = \frac{1}{3N_m}\int_0^{\infty}\left\langle\sum_{j=1}^{N_m}\boldsymbol{v}_j(t)\cdot\boldsymbol{v}_j(0)\right\rangle dt \qquad (8\text{-}19)$$

其中，$v_j(t)$和$v_j(0)$为粒子 j 在 t 时刻和 0 时刻的速度。

除以上动态信息外，分子动力学模拟还可以作为一种空间采样的手段，来计算物理量的系综平均值。系综是均有相同宏观条件的大量系统的集合。根据统计物理的知识，实验上的测量得到的物理量通常对应的是系综平均（或称为热力学平均），即对系综中所有大量系统的平均值。例如对正则系综，物理量的热力学平均由下式给出：

$$\langle Q\rangle_{系综} = \frac{\int A(p^N,r^N)e^{-\beta E(p^N,r^N)}dp^N dr^N}{\int e^{-\beta E(p^N,r^N)}dp^N dr^N} \qquad (8\text{-}20)$$

而分子动力学模拟通常只针对一个系统进行长时间的演化。这种同一个系统的长时间轨迹中不同时刻的平均值称为动力学平均（或称为时间平均），即：

$$\langle Q\rangle_{时间} = \frac{1}{M}\sum_{i=1}^{M}Q(p^N(t_i),r^N(t_i)) \qquad (8\text{-}21)$$

其中t_i为分子动力学模拟轨迹中的采样点。如果模拟轨迹足够长，则可以认为系统是各态历经的（遍历性假设）。在这种情况下，热力学平均和动力学平均是等价的，即可以通过分子模拟的时间平均来近似代替物理量的系综平均，从而给出一种基于微观分子模拟计算物理量平均值的方法。

8.4 不同系综下的分子动力学模拟

根据宏观条件的不同，可以划分为微正则系综、正则系综、等温等压系综等。微正则系综要求系统与外界没有能量和物质交换，因此对应能量 E 守恒、粒子数 N 守恒、且体积 V 不变,也称为 EVN 系综。以上 8.2 节介绍的分子模拟，满足 EVN 系综条件。如果系统可以与外界交换热量，从而温度维持在环境温度附近，而粒子数和体积维持不变，则对应正则系综（TVN）。同样地，如果维持系统的温度、压强、和粒子数不变，则对应等温-等压系综（TPN）。因此，为了模拟正则系综和等温-等压系综条件，需要在分子模拟中实现对温度和压强的控制。以下将介绍常用的等温和等压条件的实现算法。

在介绍等温条件实现算法之前，需要先介绍分子模拟中的温度的计算。根据能均分定理，系统中粒子的总动能$E_K = \sum_{i=1}^{N} m_i v_i/2$平均分配到$3N$个自由度，且每个自由度对应的动能为$k_B T/2$。其中$k_B$为玻尔兹曼常数。因此，我们可以根据系统中粒子的总动能来计算温度，即

$$T = \sum_{i=1}^{N} \frac{m_i v_i}{3Nk_B} \quad (8\text{-}22)$$

系统中粒子与外界交换热量是通过包含环境分子、容器壁、以及系统分子的碰撞实现的。当然分子模拟中无法直接包含环境分子以及容器壁。因此我们可以通过模拟分子碰撞的效果而非直接模拟碰撞过程来实现热量的交换。**Andersen 热浴法**即是通过模拟虚拟环境分子与系统分子碰撞的一种算法。具体实现步骤如下：1）设定每一个系统粒子被环境分子碰撞的频率为ν，则经过一个分子动力学时间步长Δt,每个粒子被碰撞的概率为$\nu\Delta t$。2)以均匀的概率随机选取一个粒子i，并产生一个（0，1）之间均匀分布的随机数ξ，若$\xi < \nu\Delta t$成立，则粒子i被碰撞成功。3）碰撞的结果是粒子i丢失原有速度信息，重新按给定温度 T 下的麦克斯韦速度分布律重新抽取速度。对所有粒子循环以上 1）-3）步，则系统会维持在给定温度 T 附近涨落。其中碰撞频率ν决定了系统粒子与外界热浴耦合的强度。可以证明，Andersen 浴法能够给出正则系综对应的分布。

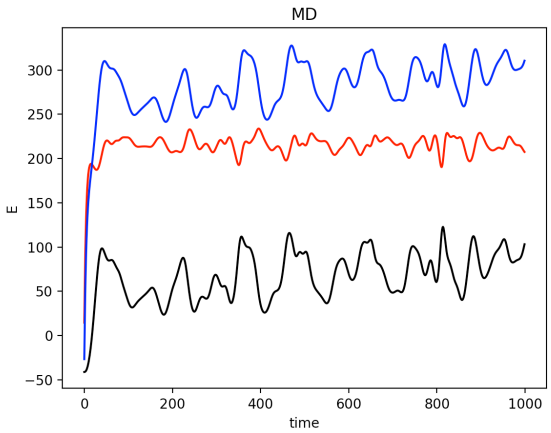

图 8.6 势能（黑色）、动能（红色）、以及总能量（蓝色）的时间演化。

Berendsen 弱耦合算法是另一类常用的热浴。设分子模拟的参考温度为T_0,由式（8-22）给出的瞬时温度为T，则对粒子的瞬时速度按因子$\lambda = \sqrt{T_0 / T}$进行重新标度，即$v_i^{new} = \lambda v_i^{old}$，则系统的温度始终维持在参考温度。这种强耦合算法也称为等动能模拟。实际应用中，可以使用弱耦合算法实现将温度维持在参考温度附近涨落。弱耦合算法中，温度不是每一步都标度到参考温度，而是根据下式逐步向参考温度弛豫：

$$\frac{dT(t)}{dt} = \frac{1}{\tau}(T - T_0) \quad (8\text{-}23)$$

即瞬时温度朝参考温度弛豫的速度正比于温度差，其中参数τ控制耦合的强度。对式（8-23）建立差分格式，得：

$$\Delta T = T(t+\Delta t) - T(t) = \frac{\Delta t}{\tau}(T_0 - T(t)) \qquad (8\text{-}24)$$

进而得到标度因子λ如下：

$$\lambda = \sqrt{T(t+\Delta t)\,/\,T(t)} = \sqrt{1 + \frac{\Delta t}{\tau}(T_0\,/\,T(t) - 1)} \qquad (8\text{-}25)$$

则对粒子的瞬时速度按以上标度因子进行重新标度，即$v_i^{new} = \lambda v_i^{old}$，则系统的温度维持在参考温度附近涨落。这种弱耦合算法最早由 Berendsen 于 1984 年提出，称为 Berendsen 热浴，具有计算效率高的优点，且能够近似给出正确的正则分布律。Code-8-2 中的子函数 105-109 行实现了 Berendsen 热浴,其能量随时间演化如图 8.6 所示。可见在等温条件下，总能量不再守恒，但总动能为此在一定范围能涨落。

以上通过对速度进行重标度，能够维持系统温度在参考温度附近涨落。类似地，可以对系统的体积进行标度，实现对压强的控制。而体积的标度可以由坐标的标度来实现。算法如下：

$$\frac{dP(t)}{dt} = \frac{1}{\tau_P}(P_0 - P(t)) \qquad (8\text{-}26)$$

其中参数τ_P控制压强耦合的强度。$P(t)$为瞬时压强，由如下 Virial 公式给出：

$$P(t) = \frac{2}{3V}\left(\sum_{i=1}^{N} m_i v_i/2 + \sum_{i<j}^{N} r_{ij} f_{ij}/2\right) \qquad (8\text{-}27)$$

上式中f_{ij}为粒子对(i,j)之间的相互作用力。对（8-26）建立差分格式得到位置坐标的标度因子为：

$$\lambda_P = \sqrt[3]{V(t+\Delta t)\,/\,V(t)} = \sqrt[3]{1 - \kappa\frac{\Delta t}{\tau}(P - P_0)} \qquad (8\text{-}28)$$

其中$\kappa = \frac{dP}{dV}$为系统的压缩系数。对粒子的瞬时位置标标按以上标度因子进行重新标度，即$x_i^{new} = \lambda x_i^{old}$，则可实现系统的压强维持在参考压强附近涨落。

‘

```python
# -*- coding: utf-8 -*-
import numpy as np
import random
import matplotlib.pylab as pl
import matplotlib.animation as animation

Natom = 36 #原子数
NT = 1000 #最大时间步数
Tinit = 0.5 #初始温度
T0 = 4.0 #平衡温度
eps = 1.0 #势阱
tau = 0.005 #Brenderson
x = np.zeros(Natom) #坐标
y = np.zeros(Natom)
vx = np.zeros(Natom) #速度
vy = np.zeros(Natom)
fx = np.zeros([Natom,2]) #力, 1) t时刻 2) t+dt时刻
fy = np.zeros([Natom,2])
L = int(1.0*Natom**0.5) #盒子边长
tt = np.arange(NT)  #时间
xc = np.zeros([Natom,NT+1]) #位置-时间
yc = np.zeros([Natom,NT+1])
EP = np.zeros(NT)  #势能-时间
EK = np.zeros(NT) #动能-时间
ET = np.zeros(NT) #总能-时间

def initialposvel(): #初始化
    i = -1
    for ix in range(L): #按格点摆放
        for iy in range(L):
            i = i + 1
            x[i] = ix
            y[i] = iy
            vx[i] = random.gauss(0,1)
            vy[i] = random.gauss(0,1)
            vx[i] = vx[i] * np.sqrt(Tinit)
            vy[i] = vy[i] * np.sqrt(Tinit)

def forces(t): #计算力
    r2cut = 9 #截断距离平方
    PE = 0.0 #势能置零
    for i in range(0,Natom): #力置零
        fx[i][t] = 0.0
        fy[i][t] = 0.0
    for i in range(0,Natom-1):
        for j in range(i+1,Natom):
            dx = x[i] - x[j] #计算距离
            dy = y[i] - y[j]
            if (dx > 0.5*L): #周期性边界条件
                dx = dx - L
            if (dx < -0.5*L):
                dx = dx + L

            if (dy > 0.5*L):
                dy = dy - L
            if (dy < -0.5*L):
                dy = dy + L

            r2 = dx*dx + dy*dy
            if(r2 < r2cut): #截断
                invr2 = 1.0/r2
                invr6 = invr2**3
                wij = 48*eps*invr2*invr6*(invr6-0.5)
                fijx = wij*dx
                fijy = wij*dy
                fx[i][t] = fx[i][t] + fijx
                fy[i][t] = fy[i][t] + fijy
                fx[j][t] = fx[j][t] - fijx
                fy[j][t] = fy[j][t] - fijy
                PE = PE + 4.0*eps*(invr6)*(invr6-1)
    return PE

def timevolution(): #时间演化
    t1 = 0
    t2 = 1
    h = 0.001 #时间步长
    hover2 = h/2.0
    initialposvel() #调用初始化
    PE = forces(t1) #计算力与势能
    for it in np.arange (NT): #时间循环
        if np.mod(it,100) == 0:
            print('it=',it)
        PE = forces(t1) #计算力与势能
        for i in range(0,Natom):
            x[i] = x[i] + h*(vx[i] + hover2*fx[i][t1]) #速度verlet更新位置
            y[i] = y[i] + h*(vy[i] + hover2*fy[i][t1])
            if x[i] <= 0: #周期边界
                x[i]=x[i] + L
            if x[i] > L:
                x[i]=x[i] - L
            if y[i] <= 0:
                y[i]=y[i] + L
            if y[i] > L:
                y[i]=y[i] - L
            xc[i][it] = x[i] #存储位置
            yc[i][it] = y[i]

        PE = forces(t2) #计算势能与力
        KE = 0.0
        for i in range(0, Natom):
            vx[i] = vx[i] + hover2*(fx[i][t1] + fx[i][t2]) #速度verlet更新速度
            vy[i] = vy[i] + hover2*(fy[i][t1] + fy[i][t2])
            KE = KE + (vx[i]*vx[i] + vy[i]*vy[i])/2 #计算动能

        Tnow = 2*KE/Natom/3 #瞬时温度
        lamda = np.sqrt(1+h/tau*(T0/Tnow-1)) #Brenderson速度标度因子
        for i in range(0, Natom): #速度重标度
            vx[i] = vx[i]*lamda
            vy[i] = vy[i]*lamda

        EP[it] = PE #存储势能
        EK[it] = KE #存储动能
        ET[it] = PE + KE #存储总能量

timevolution()

def init():
    d.set_data([], [])
    return d,

def update_line(num, xc,yc,dot):
    dot.set_data(xc[:,num],yc[:,num])
    return dot,

fig1 = pl.figure()
d, = pl.plot([], [], 'ro',markersize=30)
pl.xlim(-0.5, 6.5)
pl.ylim(-0.5, 6.5)
pl.xlabel('X')
pl.ylabel('Y')
pl.title('MD')
dot_ani = animation.FuncAnimation(fig1, update_line, np.arange(1000),\
fargs=(xc,yc,d),interval=20, init_func=init, blit=False)

fig2 = pl.figure()
pl.plot(tt,EP,'k-')
pl.plot(tt,EK,'r-')
pl.plot(tt,ET,'b-')
pl.xlabel('time')
pl.ylabel('E')
pl.title('MD')
pl.show()
```

Code-8-2