

Physics 115/242

Molecular dynamics project: sample answer

Peter Young

I took $N = 20$ particles coupled with anharmonic springs as discussed in the text. Periodic boundary conditions were used so each particle has two neighbors. I used the leapfrog algorithm with timestep $h = 0.02$. I let the system equilibrate for a time $t_{\text{equil}} = 500$ and then measured the velocity of each particle at integer times during the subsequent period of $t_{\text{meas}} = 5000$ (perhaps an overkill). I formed bins of size 0.1 to create the histogram.

The mean square velocity was also computed gave the temperature of the system to be

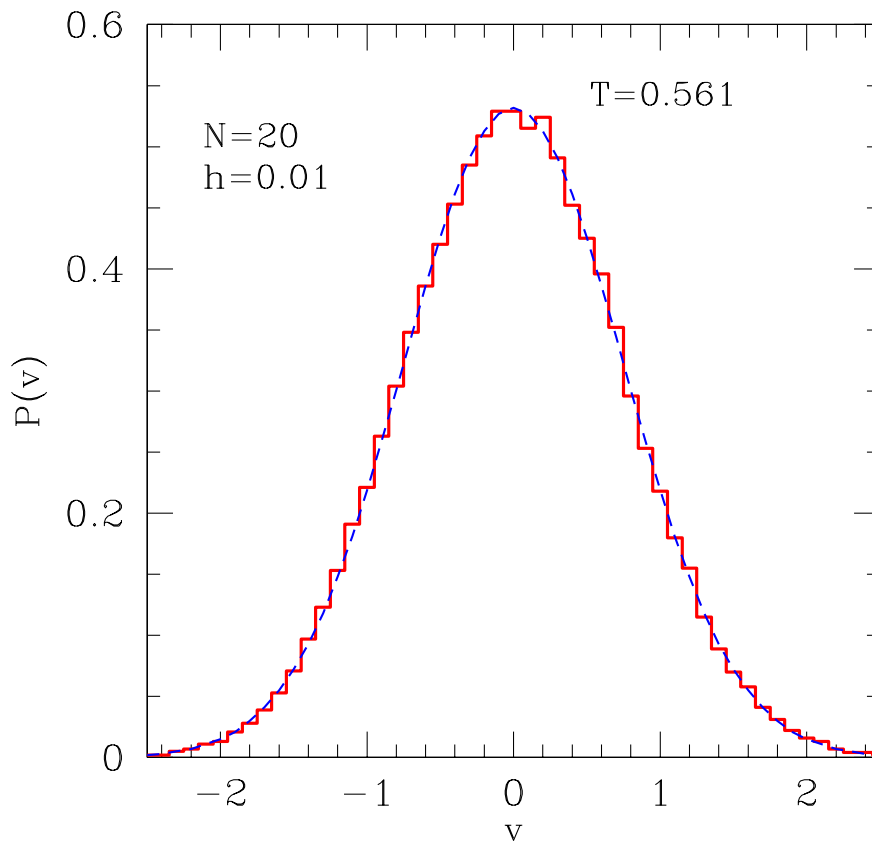
$$\langle v^2 \rangle = T = 0.561. \quad (1)$$

Hence the distribution of velocities should be given by

$$P(v) = \frac{1}{\sqrt{2\pi T}} \exp\left(-\frac{v^2}{2T}\right) \quad (2)$$

with T given by Eq. (1).

The dashed line in the figure below shows the expected distribution of velocities given by Eq. (2). The histogram in the figure is the (normalized) histogram of velocities obtained in the simulation. It is seen that the two agree very well.



To get good agreement I found it was necessary to use a fairly small time step h in the leapfrog algorithm. With a much larger h I found that the computed distribution was a little too low around $v = 0$.

Note that the source code below is divided into subroutines (functions) which do a particular job, for example initializing the velocities, and integrating one time-step. This makes the program easier to read. It is also easier to debug if there is one piece of code to do one job, which is then called at different points in the program, rather than having copies of the code at each place where the job has to be done. In the latter case, changes have to be made in each copy, which is more error prone.

```

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

double f(double x)                // Computes the force
{
    return -x - x*x*x;
}

/*****
 *
 * Does 1 time step of position Verlet. Includes periodic boundary conditions
 *
 *****/
void update_1_step(int N, double h, double x[], double v[]) // position Verlet
{
    double f(double x);
    int i;

    for (i = 0; i < N; i++)        // half-step in x
    {
        x[i] += 0.5 * h * v[i];
    }
    //
    // full-step in v; v[0] and v[N-1] treated separately because of
    // periodic boundary conditions.
    //
    v[0] += h * (f(x[0] - x[1]) + f(x[0] - x[N-1]));
    for (i = 1; i < N-1; i++)
    {
        v[i] += h * (f(x[i] - x[i+1]) + f(x[i] - x[i-1]));
    }
    v[N-1] += h * (f(x[N-1] - x[0]) + f(x[N-1] - x[N-2]));

    for (i = 0; i < N; i++)        // half-step in x
    {
        x[i] += 0.5 * h * v[i];
    }
}

/*****
 *
 * Creates a randomized array of initial velocities of either -1 or 1,
 * with N/2 of each. I choose N/2 sites randomly. Sites in this list
 * are given velocity +1, the others -1.
 *
 * Original version by Deva O'Neil. Modified by Peter Young.
 *
 *****/
void random_vs (int N, double v[])
{
    int k, num;
    double sum;
    int taken[N] ;                // shows whether a given number is already taken

```

```

srand(time(NULL)); // Seed the random number generator

for(k=0; k<N; k++)
{
    v[k] = -1.0; // Initially set all the velocities to be -1
    taken[k] = 0; // Indicates that all sites are initially not taken
}

for(k=0; k<N/2; k++) // Main loop gives k/2 random sites with +1 velocities
{
    while(1) // Continue until get a number that isn't taken
    {
        num = rand()%N;
        if (taken[num] == 0) // We have a number that isn't taken
        {
            taken[num] = 1; // num is now taken
            break; // Exit from the loop
        }
    }
    v[num] = 1.0; // Set velocity of site "num" to be +1
} // End of main loop
// Omit the rest from here once the routine works
/* sum = 0; // Test if it worked. Print out v[k]'s and the sum.
for(k=0; k<N; k++)
{
    printf("v[%2d]= %10.4f\n",k,v[k]); // Print out the velocities
    sum += v[k];
}
printf("sum = %10.4f\n", sum); // Print out the sum; should be zero.
*/
}

main()
{
    static int N=20, NH=50;
    double x[N], v[N];
    double h, t_for_equil, t_for_meas, sumv, binsize, v2, u;
    int i, time, n_for_equil, n_of_meas, n_bet_meas, istep, i_of_meas;
    int ihist, iv;
    double hist[2*NH + 1];
    void random_vs(int N, double v[]);
    void update_1_step(int N, double h, double x[], double v[]);

    binsize = 0.1;
    t_for_equil = 500;
    t_for_meas = 5000;
    h = 0.01; // step size
    n_for_equil = t_for_equil / h;
    n_of_meas = t_for_meas ;
    n_bet_meas = 1 / h; // only measure v every unit of time,
    v2 = 0; // not every step

    for (i = 0; i < 2*NH + 1; i++) // Initialize histogram
    {
        hist[i] = 0;
    }
    random_vs(N, v); // Initialize v's, 1/2 are +1, 1/2 are -1
    for (i = 0; i < N; i++) // Initialize x's

```

```

{
    x[i] = 0;
}

for(istep = 0; istep < n_for_equil; istep++) // steps for equilibration
{
    update_1_step(N, h, x, v);
}

// steps for measurement, loop over meas.
for (i_of_meas = 0; i_of_meas < n_of_meas; i_of_meas++)
{
    for (istep = 0; istep < n_bet_meas; istep++) // steps between meas.
    {
        update_1_step(N, h, x, v);
    }
    for (i = 0; i < N; i++)
    {
        v2 += v[i]*v[i]; // collect data for <v^2>
        iv = NH + round(v[i] / binsize); // add to histogram
        hist[iv] += 1;
    }
}

v2 /= n_of_meas * N; // Average v^2 is the temperature
printf (" Temperature = %8.3f \n\n", v2);
// print the histogram
printf ("      v      P(v)  P(v)(Boltz) \n");
for (ihist = 0; ihist < 2 * NH + 1; ihist++)
{
    if (hist[ihist] != 0) // only print out non-zero entries
    {
        hist[ihist] /= binsize*n_of_meas * N; // normalize the histogram
        // print histogram and expected Gaussian
        u = pow((ihist-NH)*binsize, 2);
        printf (" %8.2f %8.3f %8.3f \n", binsize*(ihist-NH), hist[ihist],
            exp(-0.5*u/v2) / (sqrt(2*3.14159*v2)));
    }
}
}

```