Worksheet 4: Thermostats

David Beyer, Russell Kajouri, Keerthi Radhakrishnan

December 9, 2024
Institute for Computational Physics, University of Stuttgart

Contents

1 General Remarks

- Deadline for the report is **midnight of December 22, 2024**
- On this worksheet, you can achieve a maximum of 20 points.
- The report should be written as though it would be read by a fellow student who attends the lecture, but doesn't do the tutorials.
- To hand in your report, upload it to ILIAS and make sure to add your team member to your team. If there are any issues with this, please fall back to sending the reports via email
  - David (`dbeyer@icp.uni-stuttgart.de`)
  - Russell (`russell.kajouri@icp.uni-stuttgart.de`)
  - Keerthi (`keerthirk@icp.uni-stuttgart.de`)

- For the report itself, please use the PDF format (we will *not* accept MS Word doc/docx files!). Include graphs and images into the report.
- If the task is to write a program, please attach the source code of the program, so that we can test it ourselves.
- The report should be 5–10 pages long. We recommend using LaTeX. A good template for a report is available online.
- The worksheets are to be solved in groups of two or three people.

2 Generating Pseudo-Random Numbers

Computers are deterministic machines. There is no magic algorithm to generate truly random numbers. Nevertheless, in many situations, for example in computer games, simulations or cryptography, it is desirable to have a way to obtain random numbers. One way around this problem is to use pseudo-random numbers. Although they are not truly random (as the name implies), they can be algorithmically generated and seem "sufficiently random" for many applications. In this exercise, you will learn how to generate pseudo-random numbers following different probability distributions.

In a first step, you will write your own random number generator using the linear congruential generator method (LCG). This method looks at the remainder of a division between a number and another large number. It uses the remainder both as a pseudo-random number (the outputted result) and to modify one of the original numbers such that the output will be different if the function is called a second time. If the process is iterated too many times, the numbers will eventually begin to repeat themselves. Since this is detrimental to almost all applications, the parameters from a pseudo-random number generator are chosen such that the period is large. It is important that the generated pseudo-random numbers are as uncorrelated as possible.

The LCG method employs three parameters called the *modulus* m, the *multiplier* a and the *increment* c. Starting from a number X_n, a new number can be obtained following

$$X_{n+1} = (aX_n + c) \mod m, \tag{1}$$

where all numbers considered are integers. The value of the generated X_{n+1} can go as high as $m - 1$. In order to produce a floating point random number in $[0, 1)$ one can simply perform a floating point division X_{n+1}/m. Note that for the iteration to be started, one must supply the initial condition X_0, often called the *seed*.

Task (2 points)

- Implement the linear congruential generator in python using the parameters $m = 2^{32}$, $a = 1103515245$ and $c = 12345$. The generator should return a normalized random number in the intervall $[0, 1)$.
- Generate 10^6 samples and plot a corresponding normalized histogram.
- Using your generator, perform a one-dimensional random walk with $N = 1000$ steps and a step-size of Δx in the interval (-0.5, 0.5)
- Verify that you obtain the same trajectory every time you execute your code by plotting the walks resulting from three consecutive calls of your function.
- Perform 200 different walks by using different seeds, for example based on `time.time()` or `os.getpid()` and put them into a single plot.

The Mersenne Twister is a fast, uncorrelated, random number generator with a long period. It is a popular method and is available using `numpy.random.random()`, which uses it's own self-seeding methods. From now on, you can use `numpy.random.random()` to obtain uniform random numbers. In a later task, it will turn out to be useful to know that uniform random numbers from an interval (a, b) have a standard deviation of $\sigma = \frac{b-a}{\sqrt{12}}$.

Often, one wants to obtain a random number from a Gaussian distribution. The *Box–Muller* transform allows one to obtain Gaussian random numbers from uniformly random numbers. The pitfall of this method is that it works in pairs. It needs two uniform random numbers u_1 and u_2 as an input, however, it also yields two random numbers n_1 and n_2 that follows a normal distribution.

The transform is quite straight-forward:

$$n_1 = \sqrt{-2\log(u_1)}\cos(2\pi u_2) \tag{2}$$

$$n_2 = \sqrt{-2\log(u_1)}\sin(2\pi u_2) \tag{3}$$

where n_1 and n_2 are random numbers that follow a zero-mean normal distributed with a standard-deviation of 1.

Task (2 points)

- Implement the Box-Muller transform to generate a histogram of $N = 10^5$ random numbers that follow a Gaussian distribution with a mean of $\mu = 1.0$ and a standard-deviation of $\sigma = 4.0$. Normalize your histogram and include the expected analytical curve for this distribution.
- Generate $N = 10^5$ random Gaussian velocity vectors $\mathbf{v} = (v_x, v_y, v_z)$ which have elements v_x, v_y and v_z taken from a Gaussian distribution with mean $\mu = 0$ and standard-deviation of $\sigma = 1.0$
- Plot the distribution of the speeds $v = |\mathbf{v}|$ obtained from your random vectors and compare with the analytical three-dimensional Maxwell-Boltzmann distribution.

The Box-Muller transform introduced above allows us to sample random numbers from a non-uniform probability distribution, namely the Gaussian one. There also exists an alternative method to sample from non-uniform distributions, the so-called *inverse transformation sampling* method, which can in principle be applied to a broad number of distributions. Consider a positive definite ($p(x) > 0$) probability distribution that is normalized, i.e.

$$\int_{-\infty}^{\infty} p(x)\,\mathrm{d}x = 1, \tag{4}$$

from which we would like to sample ($p(x)$ can also have a compact support, we will simply use the interval $(-\infty, \infty)$ in the following). We can now introduce the *cumulative* distribution $P(x)$, which is defined by

$$P(x) = \int_{-\infty}^{x} p(x')\,\mathrm{d}x'. \tag{5}$$

Due to the normalization and the fact that $p(x) > 0$, we clearly have $0 \leq P(x) \leq 1$, i.e. $P(x)$ is a probability distribution. Furthermore,

$$\frac{\mathrm{d}}{\mathrm{d}x}P(x) = \frac{\mathrm{d}}{\mathrm{d}x}\int_{-\infty}^{x} p(x')\,\mathrm{d}x' = p(x) > 0, \tag{6}$$

i.e. $P(x)$ is strictly monotonous.

The basic idea of the inverse distribution method is the following: let us assume that we know a method to produce samples of x according to the probability distribution $p(x)$. If we produce a large number of samples and calculate $P(x)$ for each of them, the values

of $P(x)$ will follow a uniform distribution in the interval $[0, 1]$. To see that this indeed the case, consider the probability distribution $\mathcal{P}$ of $P(x)$:

$$\mathcal{P}\left(P(x)\right) = \int_{-\infty}^{\infty} \delta\left(P(x) - P(x')\right) p(x')\,\mathrm{d}x' \tag{7}$$

We can evaluate this integral by using the substitution $x' \to P(x')$ (which is unique because $P(x)$ is strictly monotonous) and the identity $p(x')\,\mathrm{d}x' = dP(x')$ introduced above:

$$\mathcal{P}\left(P(x)\right) = \int_{0}^{1} \delta\left(P(x) - P(x')\right) dP(x') = 1, \tag{8}$$

i.e. $P(x)$ follows a uniform distribution in the interval $[0, 1]$. This insight is crucial, as we know how to produce uniformly distributed samples. Thus, we can simply sample a uniformly distributed random variable y in the interval $[0, 1]$ (corresponding to $P(x)$) and obtain the corresponding x by inverting $P(x)$:

$$x = P^{-1}(y). \tag{9}$$

This provides us a simple way to sample from a non-uniform distribution, given that we can calculate $P^{-1}(y)$.

Task (2 points)

- Implement the inverse distribution method for the probability distribution $p(x) = 3x^2$ with $x \in [0, 1]$ (and thus $P(x) = x^3$ and $P^{-1}(y) = y^{1/3}$). Generate 10^5 random samples, plot a histogram and compare it to the analytical probability distribution.
- How could one proceed if $P^{-1}(y)$ cannot be inverted analytically? What if even $P(x)$ cannot be evaluated analytically?

3 Langevin Thermostat

As we have seen on the last worksheet, even though the *velocity re-scaling thermostat* is able to keep the temperature constant, this is not actually the same as simulating the canonical (N, V, T)-ensemble, as the Maxwell-Boltzmann distribution of the velocities is destroyed.

A thermostat that does allow to simulate the canonical ensemble is the *Langevin thermostat*. In the Langevin thermostat, at each time step every particle is subject to a

random (stochastic) force and to a frictional (dissipative) force. There is a precise relation between the stochastic and dissipative terms, which comes from the so-called fluctuation-dissipation theorem, and ensures sampling of the canonical ensemble. The equation of motion of a particle is thus modified to

$$m\mathbf{a}_i = \mathbf{F}_i - m\gamma\mathbf{v}_i + \mathbf{W}_i(t), \tag{10}$$

with the introduction of a friction coefficient γ that has the units of an inverse time and a random force $\mathbf{W}_i$ acting on particle i that is uncorrelated in time and between particles, and otherwise characterized by its variance:

$$\langle \mathbf{W}_i(t) \cdot \mathbf{W}_j(t') \rangle = \delta_{ij}\delta\left(t - t'\right) 6k_B m T\gamma. \tag{11}$$

The modified Velocity-Verlet algorithm for a Langevin thermostat is

$$\mathbf{x}_i(t + \Delta t) = \mathbf{x}_i(t) + \Delta t\mathbf{v}_i(t)(1 - \Delta t\frac{\gamma}{2}) + \frac{\Delta t^2}{2m}\mathbf{G}_i(t) \tag{12}$$

$$\mathbf{v}_i(t + \Delta t) = \frac{\mathbf{v}_i(t)(1 - \Delta t\frac{\gamma}{2}) + \frac{\Delta t}{2m}(\mathbf{G}_i(t) + \mathbf{G}_i(t + \Delta t))}{(1 + \Delta t\frac{\gamma}{2})} \tag{13}$$

where $\mathbf{G}_i$ is the total force: $\mathbf{G}_i = \mathbf{F}_i + \mathbf{W}_i$.

Task (4 points)

- For the purposes of this worksheet, the random vector $\mathbf{W}_i(t)$ can follow a uniform distribution. Make sure the mean is zero and that each of the three components have a standard deviation of $\sigma = \sqrt{2mk_{\mathrm{B}}T\gamma/\Delta t}$ where Δt is the simulation time-step.
- Extend the template `ex_3_1.py` to implement the Langevin thermostat, you can use $\gamma = 1.0$. Write a function `step_vv_langevin()` that implements Eq. (12) and (13). Remember to update the force between the two-half steps. The function `step_vv()` gives you a rough guide how to proceed.
- Extend the template by writing the `compute_instaneous_temperature` function, which computes the instantaneous temperature from the velocities.
- Run the script for temperature $T = 1.0$, particle number $N = 100$ and time step $\Delta t = 0.01$ for a total of 2000 time units (i.e. $2 \cdot 10^5$ time steps).
- Plot the instantaneous temperature vs. time and explain what you see.
- Plot the distribution of the absolute value of the average particle velocity $P(|\vec{v}|)$ and compare with the three-dimensional Maxwell-Boltzmann distribution.

4 Diffusion Coefficients

In this analysis section you will need simulation trajectories for the Langevin thermostat. E-mail your tutor for sample trajectories if you were unable to perform them.

In this task, you will be asked to calculate the diffusion coefficient by averaging across the trajectories for non-interacting particles.

One method consists of measuring the mean-squared-displacement (MSD) and performing a fit using

$$\langle \Delta x^2 \rangle := \langle \Delta \left| \vec{x} \right|^2 \rangle = 2Dd\Delta t, \tag{14}$$

where D is the diffusion coefficient and d the dimensionality of the system. Note that here Δt is not related to the integrator's time step.

The MSD can be obtained from a single trajectory covering T time by subdividing it into N sub-trajectories of duration $\Delta t = \frac{T}{N}$ each. At each time subdivision $\Delta t \in (0, T]$,

$$\langle \Delta x^2 \rangle_i = \left| \vec{x}(i\Delta t) - \vec{x}((i-1)\Delta t) \right|^2 \tag{15}$$

is averaged over the N sub-trajectories to give

$$\langle \Delta x^2 \rangle (\Delta t) = \frac{1}{N} \sum_{i=1}^{N} \langle \Delta x^2 \rangle_i. \tag{16}$$

For example, there are $N = 5$ sub-trajectories of length $\Delta t = 10$ that can be extracted from a single trajectory of length $T = 50$ or $N = 7$ at $\Delta t = 7$ (discarding the last few time steps if N does not divide T without remainder).

Your task is to determine the squared displacements $\langle \Delta x^2 \rangle (\Delta t)$ that occur during a time Δt. Find the diffusion coefficient by fitting the data to the diffusion model.

Task (4 points)

- Open the simulation trajectories and plot the MSD resulting from the Langevin thermostats. Include error bars for the average from Eq. (16). The trajectories written by the template consist of $3N$ columns, where N is the number of particles. Calculate the MSD for each column separately, then average over all columns.
- Perform a fit in the linear region to determine the diffusion coefficient.
- What is the cause for the non-linear behaviour for short time lags?

An alternate way to obtain a diffusion coefficient from a dynamical simulation is from the velocity auto-correlation function (VACF) via the Green-Kubo relation:

$$D = \int_0^\infty \langle v(t) \cdot v(0) \rangle \, \mathrm{d}t. \tag{17}$$

Numerically, you can determine auto-correlation functions either directly via `numpy.convolve` or `scipy.signal.convolve` or spectrally via `numpy.fft` or `scipy.signal.fftconvolve`.

Task (3 points)

- Open the velocity data-file generated by the Langevin thermostat and plot the VACF. Take care of the scaling of the VACF by ensuring that the value for zero time lag $\langle v(0)^2 \rangle$ is consistent with the chosen temperature.
- Numerically integrate the VACF to determine the diffusion coefficient. Compare with the expected result for the chosen temperature and friction.

5 Andersen Thermostat

The Langevin thermostat we have explored in the previous exercises is not the only stochastic thermostat that can be used in a molecular dynamics simulation. An alternative is the so-called Andersen thermostat, which simulates collisions with a heat bath in a stochastic manner: In addition to the temperature T, the heat bath is thus characterized by a collision frequency ν. In practice, after an integration step, each particle undergoes a collision with the heat bath with probability $\nu \mathrm{d}t$. Such a collision means that the particle is assigned a new velocity, which is randomly chosen from the Boltzmann distribution at the desired temperature.

Task (3 points)

- Copy your code from exercise 3 and add a function `step_vv_andersen()` that performs a Velocity-Verlet integration step with a subsequent Andersen thermostatting.
- Run the simulation for the same parameters as in exercise 3 and with $\nu = 0.01, 0.1$. As in exercise 3, plot the speed distribution and the time evolution of the instantaneous temperature. How does the collision frequency influence the results?
- Is the Andersen thermostat a reasonable choice to calculate dynamical properties in a simulation?