UMEÅ UNIVERSITY
Department of Physics
Modeling and Simulation, 7.5hp

September 30, 2022

Molecular Dynamics, Stochastic simulations, and Monte Carlo

Peter Olsson

Miscellanous comments

- The following instructions are intended to be used for a Ubuntu/Linux system. If you are in the computer lab you might need to restart the computer. It will then automatically boot into Ubuntu.
- Open up the lab instructions from the browser and open up a terminal which is what you run the program from. You will now be able to copy things directly from the lab instructions to the terminal. Note that copying text is done by just pressing and moving the left mouse button, nothing more. To paste the content to the terminal: Move the mouse over the terminal, activate the terminal by left-klicking and paste by pressing the middle mouse button.
- Send the report through Canvas.
- Provide a path to the relevant code on the computers, or make it available by some other means.
- Note that there are some voluntary excercises that may give bonus points. The bonus points are determined on the basis of the first version of the lab report and are only considered if the report is handed in before deadline.
- There are functions in ran.c for getting different kinds of random numbers.

1 Introduction

A gas of particles is a very important system in physics. If the density is low, one can neglect collisions between particles. When that is a good approximation we say that the gas behaves like an ideal gas and is then well described by the ideal gas law,

$$pV = Nk_BT,$$

which is a relation between pressure p, volume V, number of particles N, temperature T, and Boltzmann's constant, $k_B \approx 1.38 \times 10^{-23}$ Joule/Kelvin.

1.1 Interaction

The fact that gases liquify at higher densities is due to interactions between the molecules. This interaction is repulsive at short distances and attractive at somewhat larger distances and is commonly modelled with the Lennard-Jones interaction,

$$U(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]. \tag{1}$$

Here the length σ and the energy ϵ are material parameters. We will however use units such that they are both equal to unity. The force between the particles is given by $F = -dU/dr$. A positive sign means a *repulsive* force.

$$F(r) = \frac{\epsilon}{r} \left[48 \left(\frac{\sigma}{r}\right)^{12} - 24 \left(\frac{\sigma}{r}\right)^{6} \right]. \tag{2}$$

The above expression gives the magnitude of the force; to capture the vector nature we write (in two dimensions)

$$\mathbf{F} = -\nabla U = -\frac{dU}{d\mathbf{r}} = -\frac{dr}{d\mathbf{r}}\frac{dU}{dr} = -\left(\frac{dr}{dx}, \frac{dr}{dy}\right)\frac{dU}{dr} = \left(\frac{x}{r}, \frac{y}{r}\right) F(r). \tag{3}$$

The last equality follows from differentiation of $r^2 = x^2 + y^2$ which gives

$$r\ dr = x\ dx + y\ dy.$$

1.2 Molecular dynamics

The dynamics of a system with N particles is governed by two coupled differential equations,

$$\begin{aligned} \dot{\mathbf{r}}_i &= \mathbf{v}_i, \\ m\dot{\mathbf{v}}_i &= \mathbf{F}_i, \end{aligned}$$

where $\mathbf{F}_i$ is the sum of the forces acting on particle i due to the interaction with the other particles. In the simulations this should be integrated with a small time step, Δt, and with the notation $\mathbf{v}_i^{(n)} = \mathbf{v}_i(n\Delta t)$ the leap-frog method becomes

$$\mathbf{v}_i^{(n+1/2)} = \mathbf{v}_i^{(n-1/2)} + \frac{1}{m}\mathbf{F}_i^{(n)}\,\Delta t, \tag{4}$$

$$\mathbf{r}_i^{(n+1)} = \mathbf{r}_i^{(n)} + \mathbf{v}_i^{(n+1/2)}\,\Delta t. \tag{5}$$

We will use simulation units which means that we set $\sigma = 1$, $m = 1$, $\epsilon = 1$ and $k_B = 1$, and let $k_B T \to T$.

2 A program for Molecular dynamics and Langevin dynamics

We are here going to simulate interacting particles in two dimensions. An important reason for looking at this problem in two dimensions is that it then becomes much easier to visualize the configurations. This is also one of the voluntary exercises.

Your task is now to complete an almost finished program.[1] The first step is to look through the program *carefully* to understand how it is meant to work. Check both the `Makefile` and all the source files. This is also a good opportunity to learn a few programming tricks. The program begins (as always) in the `main` function which is at the bottom of `sim.c`.

[1] Some of you could prefer to instead write everything from scratch and would perhaps also do that successfully. This exercise could anyway be a good one since the modification of an existing code is more common than writing things from scratch, which means that there is a fair chance that you will be asked to modify an existing program in your coming professional activities.

2.1 Getting the code

Open a terminal by pressing Ctrl-Alt-t, make a new directory

```
$ mkdir LabStoch
```

and go to that directory by writing

```
$ cd LabStoch
```

Quite a few files are needed to complete this lab and they are stored in a compressed archive at `www.tp.umu.se/modsim/files/LabStoch.tgz`. The command line can be used to download this file (though it can of course also be done with a web browser):

```
$ wget www.tp.umu.se/modsim/files/LabStoch.tgz
```

To get all the files together with some directories, execute[2]

```
$ tar xzf LabStoch.tgz
```

which is the command to e**x**tract the content from the g**z**ipped (compressed) tar **f**ile. To see what files and directories are present type

```
$ ls -l
```

which gives a listing of the directory. In the following you should edit files in `src` and compile in `lang`.

2.2 Completing the code

Note that appendix A gives a short introduction to the program.

Follow the steps below to convince yourself that your program is correct. In the source code the incomplete sections are marked with "`Fix this (1)`", where the numbers, 1–5, correspond to the numbers below. This part is only to become familiar with the workings of the program and should not be included in the report.

For editing the source code with the standard text editor, `gedit`, type

[2]In computing, tar (derived from tape archive) is both a file format (in the form of a type of archive bitstream) and the name of the program used to handle such files. ".tgz" is the same as ".tar.gz" – a gzipped tar file.

```
$ gedit src/sim.c &
```

This will open a new window that shows `sim.c`. The & character is needed to be able to continue using the terminal for other commands. The shell (which is the program running in the terminal and accepting input) will otherwise wait for `gedit` to terminate before showing the prompt, "$", which shows that it accepts input. Then go to `lang`

```
$ cd lang
```

to be ready for the following steps:

1. Complete the function `measure` (in `sim.c`) which is necessary to perform the measurements in the program. Appendix A gives some information about the layout of the array `atoms` used for storing both position and velocity coordinates. Compile the program by typing `make` in the terminal. If you now do "`$ ls -l`" you will see that there is now a file `sim` which is "executable" which is the meaning of the "x" in "-rwxr-xr-x". Now run the program by typing

   ```
   ./sim N=64 rho=0.5 T=1.0 read=0064_r0.500_T1.000_start run
   ```

   The program should now give the energies for the starting configuration:

   ```
   Potential energy = -1.33933
   Kinetic energy   = 1.04104
   ```

   Note that these are intensive values, energy per particle.

2. Next, remove the `exit(EXIT_SUCCESS)` in `sim.c` that terminated the program. Study the function `step` and complete `force_magnitude` and `one_force` in `common.c` by using the equations above. Also fix the function `vel_from_force` to implement the leap-frog method. To do this you will have to look through the program to understand how it is intended to work.

   If not specified otherwise the friction parameter is $\alpha = 0$, which means that the program by default does molecular dynamics. With the same command line as before you should now get the following output for the positions and velocities for particle number 0:

```
x, y, vx, vy = 3.78663  9.78626  -0.299201  -1.60687
```

3. Remove the print statement for `x, y, vx, vy = ...` and the `exit` statement. The program will then instead run `par->ntherm` units of time for equilibrium. The program should now be ready for long runs.

4. We want both averages and standard errors from the collected data and that is prepared for in the code. Note the meaning of `v1sum[]`, and `v2sum[]` and complete the function `print_standard_error` to print out the mean and the estimated standard error according to the following formulas:

$$X = \frac{1}{\text{nblock}} \sum_i x_i, \quad X_2 = \frac{1}{\text{nblock}} \sum_i x_i^2, \quad \text{stderr}(X) = \sqrt{\frac{X_2 - X^2}{\text{nblock} - 1}}.$$

   This is a quantity that decreases when the run is longer, i.e. the number of blocks is bigger.

   It is also sometimes interesting to determine the typical size of the fluctuations which is given by

$$\sqrt{X_2 - X^2}.$$

   As this is a measure of the size of the fluctuations it shouldn't (and doesn't) decrease with an increasing number of blocks.

5. The program should now perform (deterministic) Molecular dynamics. To change this to the stochastic Langevin dynamics you need to complete `langevin_forces` (`common.c`) and calculate the magnitude of the noise (at the beginning of `run_simulation` in `sim.c`) correctly. See Sec. 3.2 for details. Note that $\langle \xi_\pm^2 \rangle = 1/3$ if $\xi_\pm$ is from a uniform distribution between -1 and 1. (This is the kind of random numbers returned by `dran_sign()`.) Adjust the magnitude of the noise and run the program with $\alpha = 0.1$ to check that the kinetic energy per particle is equal to T, which is $T/2$ per degree of freedom (= dimension). If that is not so, please contemplate the meaning of $\langle \xi_\pm^2 \rangle = 1/3$, before asking the lab tutor.

The program should now be ready for the two following exercises.

3 Exercises

3.1 Molecular dynamics and the time step

In molecular dynamics it is important to check that the time step is not too big. To that end one looks at the total energy which should be a constant. Do a number of runs, starting from `conf/0064_r0.500_T1.000_start`, with $\alpha = 0$ and time steps, $\Delta_t = 0.006$, 0.008, 0.010, 0.012 0.014, 0.016, 0.018, and 0.020. Use at least 50 blocks (`nblock=50`) in each run. Consider both *stability* and *precision*. The files in directory `efile` contain time and total energy per particle, $E = (U + K)/N$.

Exercise 3.1:

1. What happens for large time steps?
2. Show the data for all but the largest time steps by plotting the total energy per particle vs Δ_t. Show the standard errors with error bars.
3. There could be a spurious dependence of energy on Δ_t, in your data, which is due to the fact that energy is not conserved exactly, which means that the temperature is changing. (This is in contrast to e.g. Langevin simulations which have an built-in thermostat that keeps the temperature at the desired value.) To examine if this dependence could explain the different E for different Δ_t, determine the temperature from the kinetic energy per particle, $T = K/N$, and plot E vs T.

Comment on the result.

3.2 Langevin dynamics – the effect of a finite α

In Langevin dynamics the total energy fluctuates and it is interesting to examine how these fluctuations depend on α. In this dynamics both damping and noise give changes to the velocity:

$$m\mathbf{v}_i(t+\Delta_t) = m\mathbf{v}_i(t) + (\mathbf{F}_i - \alpha\mathbf{v}_i + \boldsymbol{\zeta}_i)\Delta_t. \tag{6}$$

Here $\boldsymbol{\zeta}_i$ is a vector noise where each component has the magnitude

$$\langle\zeta^2\rangle = \frac{2\alpha T}{\Delta_t}.$$

Exercise 3.2: The question in focus is now what changes and what remains the same when we change α. Do runs with $\alpha =$ 0.01, 0.1, and 1.0, density $\rho = 0.6$ and temperature $T = 1.0$ and plot energy vs time. Remove the statement `read=conf/0064_r0.500_T1.000_start`, since that is meant to work with $\rho = 0.5$. It is now OK with shorter runs, say 10 blocks. We take $\Delta_t = 0.01$ which gives a good precision (as shown above), and is also the default value of the time step in the code. Consider both the size and the velocity of the energy fluctuations:

1. Use the data in the files in directory `efile` to calculate the size of the fluctuations from

   $$\sigma_E = \sqrt{\langle E^2\rangle - \langle E\rangle^2}.$$

   Do the size of the fluctuations in energy change with α? What do you expect from theory?

2. For this point you will then need to change the writing to the the energy file (`efile`) such that it happens after each measurement, and not only after every 100th measurement, which is what "`if (isamp % 100 == 0)`" before the call to `energy_print` does. We do, however, not always like to print each measured energy value to file, as it tends to produce big files.

   Plot E vs t. Comment on the energy fluctuations—are they rapid or slow?—and relate them to the fact that m/α has the dimension of time.

4 Brownian dynamics

Go to directory `brown` to use for Brownian dynamics and copy the files:

```
$ cd ..
$ cp lang/define.h lang/Makefile brown
$ cd brown
```

You then need to change in the new `define.h`:

- Remove `#define VEL` since Brownian dynamics only operates on the positions.
- Instead include `#define BROWN` which should be used in the code (mostly in the function `step` in `common.c`) for selecting statements to use for Brownian dynamics.

The dynamics is given by

$$\mathbf{r}_i(t + \Delta_t) = \mathbf{r}_i(t) + \mathbf{F}_i \frac{\Delta_t}{\alpha} + \boldsymbol{\eta}_i \Delta_t,$$

where each component of the noise is characterized by

$$\langle \eta^2 \rangle = \frac{2T}{\alpha \Delta_t}.$$

As a quick test that the program is correct, check that the obtained potential energy when using $\alpha = 10.0$ and $\Delta_t = 0.001$ gives about the same results as the Langevin program.

5 Monte Carlo

The last exercise is to make another program that does Monte Carlo. Go to the `mc` directory, copy `define.h` and `Makefile`, and put `#define MC` in `define.h`.

The changes to implement Monte Carlo will mainly be to write a function that performs a Monte Carlo sweep. Note that you should also keep track of the acceptance ratio and that you will need the parameter b that specifies the maximum distance to move the particles.

A Monte Carlo sweep is done by looping over the particles i and for each i do the following operations:

1. Suggest a new position: $\mathbf{r}_i' = \mathbf{r}_i + b\boldsymbol{\delta}$, where $\boldsymbol{\delta}$ is a random vector.

2. Use ν for the original configuration and μ for the suggested one, and calculate the energy difference $\Delta U = U_\mu - U_\nu$ and

$$\alpha_{\nu\to\mu} = \min\left(1, e^{-\Delta U/T}\right).$$

3. To accept with this probability, generate a random number ξ and accept if $\xi < \alpha_{\nu\to\mu}$, which means that we let $\mathbf{r}_i'$ be the new $\mathbf{r}_i$.

5.1 Comparison

We will now demonstrate that Brownian motion gives the same results as Monte Carlo in the limit of small Δ_t/α:

> **Exercise 5.1:** Make a long run with the Monte Carlo program to get good precision for the potential energy with the parameters $N = 64$, $\rho = 0.3$, and $T = 1.0$. Run the Brownian dynamics program with $(\alpha, \Delta_t) = (10.0, 0.001)$, (10.0, 0.002), (10.0, 0.003), and (10.0, 0.004). Plot the potential energy *with error bars* vs Δ_t/α. Put the Monte Carlo data at $\Delta_t/\alpha = 0$. Be sure to run long enough to get reasonably precise data, which in this case means that the statistical errors ought to be < 0.002. It should be possible to see a clear trend in the data.

6 Voluntary exercises

For (at least some of) these voluntary exercises it could be convenient not to write out the result from the simulations to the terminal, only, but to also write them to files, e.g. in a directory `res`, with file names given by the parameters, just as the final configuration is written to `conf/`*filename*. When doing this, be sure to write both the parameters of the run and the average values (and perhaps also the standard error) to the files and to make files that are easily read into your plotting program (e.g. `matlab`).

6.1 Velocity correlation

Whereas static quantities—the ones that may be determined from configuration snapshots—should be the same independent of α dynamic quantities should be different for different α. The task is now to determine the velocity auto-correlation function $g_v(t)$

$$g_v(t) = \langle \mathbf{v}_i(t') \cdot \mathbf{v}_i(t'+t) \rangle, \tag{7}$$

Exercise 6.1: Determine $g_v(0.0)$, $g_v(0.1)$, $g_v(0.2)$,... $g_v(5.0)$ at $\rho = 0.2$ (a lower density than before), $T = 1.0$, and $\alpha = 0.01$, 0.1, and 1.0 and show them in the same plot. (1p)

Hints: Use a circular buffer to store the velocity vector for 50 earlier times and create a file to write out the velocity correlations. It could be convenient to put the functions for initialisation, accumulation, and for writing out results related to the velocity correlation in a file `vcorr.c` together with a header file `vcorr.h`, which should be included from both `sim.c` and `vcorr.c`

6.2 Different phases

At high temperatures (here, around or above $T = 1$) the system is a gas with a uniform density. When the temperature is lowered the attractive interaction starts to become more important and the particles prefer to be close to one another. This will then lead to a denser region (a liquid) coexisting with the region with lower density (a gas). The liquid is however about as disordered as the gas. If the temperature is lowered even further the system will try hard to get the lowest possible energy, which is obtained by forming an ordered crystal. The transition between such phases may be studied with

advanced methods, but we will here just look at snapshots from the stored configuration files.

> **Exercise 6.2:** Do Monte Carlo at temperatures $T = 1.0$, 0.9, 0.8,... 0.1 on a larger system with $N = 200$ and $\rho = 0.25$. Be sure to start the run at the lower temperature from the configuration of the next higher one. Show representative pictures of (1) gas, (2) gas-liquid coexistence, and (3) solid phase (well, gas-solid coexistence), and try to approximately locate the phase transition temperatures. (1p)

6.3 Pressure curves

The interactions make the pressure differ from the ideal gas law, which with our units, with k_B absorbed into the temperature, becomes $p = NT/V = T\rho$. To get best possible precision, we here use Monte Carlo. Do rather long runs, `nblock=100`, with $N = 64$ particles at temperatures $T = 0.5$, 0.6,... 1.0 and densities $\rho = 0.15$, 0.20, 0.25, 0.30, 0.40, 0.50, 0.60, 0.70, and 0.80. Note that the equilibrium state for some parameters at low temperatures is a phase separated state, with regions of higher and lower density, which could be difficult to reach. One therefore needs to use many sweeps for thermalization (e.g. `10000`) and also to use the stored configuration from a higher temperature as a starting point for the next lower temperature. This may be achieved with

```
./sim N=64 rho=0.500 T=1.000 read T=0.900 nblock=100 run
```

since the `read` command reads the configuration associated with $T = 1.0$ from an earlier run, if such a run has been done before.

> **Exercise 6.3:** Plot p vs $1/\rho$ for $T = 0.5$ through 1.0. Also include two dashed lines with $p_{\text{ideal}} = T\rho$ for $T = 0.5$ and 1.0. (1p)

6.3.1 Running several simultaneous jobs

Below are some hints for speeding things up. You don't need to follow them.

With 8 different densities and 6 temperatures these runs could be rather cumbersome. Since modern computers typically have four cores it is however possible to speed things up by running several jobs at the same time. (This is even better done on the server, `sesam`, which has 12 cores. From the

computer lab, log in with "`ssh sesam`".) To run several jobs at the same time one can make use of the built-in loop feature in the `bash` shell. As a simple demonstration try

```
for r in 0.15 0.20 0.25; do echo $r; done
```

In our case this can be combined with the line above to give (but change "`...`" and keep it on a single line)

```
for r in 0.15 0.20 ... 0.80; do ./sim N=64 rho=0.$r
  T=1.000 read T=0.900 nblock=100 run & done
```

(The construct "`./sim...run & done`" starts the runs in parallel. If one had instead written "`./sim...run ; done`" the runs would be started sequentially, one after the other, which is not what we want.)

A drawback with this approach is that it mixes the output from the programs. To avoid that one can redirect the output from each run to a separate file:

```
for r in 0.15 0.20 ... 0.80; do ./sim N=64 rho=0.$r T=1.000
  read T=0.900 nblock=100 run > log/0064-r${r}_T0.900 & done
```

A Program overview and some hints for the coding

A.1 Data storage

There are several different ways to store the positions and velocities for a set of particles, but we here go for the simplest which is just to use arrays of type double. One reason for this choice is that it is then easy to write code with or without velocity coordinates which can be used in both two and three dimensions.

With the variable `pos` for the positions, the x and y, coordinates of the first particle, particle number zero, are then `pos[0]` and `pos[1]`, and it follows that the storage for the second particle start at `pos[2]`, or, more generally, with `D` for the dimensionality, storage for particle `i` starts at `pos[D * i]`.

Since some of our simulation methods uses velocity variables whereas others do not, we want the velocity variable, `vel`, to be present only in certain versions of the program. We then let the first half of the array `atoms` contain the position array whereas the second part contains the velocities. With the number of particles in `par->n`, a part of the code may then be written

```
double *atoms;
atoms = malloc(par->n * 2 * D * sizeof(double));

double *pos = atoms;
double *vel = atoms + par->n * D;
```

The notation in "`double *pos = atoms`" is confusing, but the declaration should be taken to mean that `pos` is of type `double *`, a pointer to a double. This is thus actually the same as

```
double *pos;
pos = atoms;
```

To illustrate this we show the code to advance the position of the particles one time step. We can then either write

```
for (i = 0; i < par->n; i++) {
```

```
    for (d = 0; d < D; d++)
      pos[D * i + d] += par->deltat * vel[D * i + d];
  }
```

or, by just combining in a single loop

```
  for (id = 0; id < D * par->n; id++)
    pos[id] += par->deltat * vel[id];
```

or, by using `ipos` and `ivel` for position and velocity of particle `i`,

```
  for (i = 0; i < par->n; i++) {
    double *ipos = pos + D * i;
    double *ivel = vel + D * i;
    for (d = 0; d < D; d++)
      ipos[d] += par->deltat * ivel[d];
  }
```

Note that the code above is a not complete, as the particles will now and then move outside the simulation box. One therefore needs to include a check that the new position is within the range $[0, L)$, and otherwise move it to the image position that is inside the simulation box. This test should be done each time a particle is moved.

A.2 Code overview

Sketch of the contents of `sim.c`:

```
void run_simulation(Par *par)
{
    .                // Initialize
    .
  for (iblock = 0; iblock < par->nblock; iblock++) {
    for (isamp = 0; isamp < par->nsamp; isamp++) {
      for (istep = 0; istep < nstep; istep++)   // Advance one time unit
        step(par, atoms, force);                // Take one leap-frog step

      measure(par, atoms, vblock);        // Measure quantities, store in vblock
    }
  }
    .
}

// read_args interprets the commands on the command line
int read_args(Par *par, char *arg)
{
    .
  if (!strcmp(arg, "run")) {
    .
    run_simulation(par);
  }
    .
}

// The main program. Execution starts here.
int main(int argc, char *argv[])
{
  for (iarg = 1; iarg < argc; iarg++)
    if (!read_args(&par, argv[iarg]))
      exit(EXIT_FAILURE);
  .
  exit(EXIT_SUCCESS);
}
```