Introduction to numerical projects

Here follows a brief recipe and recommendation on how to write a report for each project.

- Give a short description of the nature of the problem and the eventual numerical methods you have used.
- Describe the algorithm you have used and/or developed. Here you may find it convenient to use pseudocoding. In many cases you can describe the algorithm in the program itself.
- Include the source code of your program. Comment your program properly.
- If possible, try to find analytic solutions, or known limits in order to test your program when developing the code.
- Include your results either in figure form or in a table. Remember to label your results. All tables and figures should have relevant captions and labels on the axes.
- Try to evaluate the reliabilty and numerical stability/precision of your results. If possible, include a qualitative and/or quantitative discussion of the numerical stability, eventual loss of precision etc.
- Try to give an interpretation of you results in your answers to the problems.
- Critique: if possible include your comments and reflections about the exercise, whether you felt you learnt something, ideas for improvements and other thoughts you've made when solving the exercise. We wish to keep this course at the interactive level and your comments can help us improve it.
- Try to establish a practice where you log your work at the computerlab. You may find such a logbook very handy at later stages in your work, especially when you don't properly remember what a previous test version of your program did. Here you could also record the time spent on solving the exercise, various algorithms you may have tested or other topics which you feel worthy of mentioning.
- You should include tests of your algorithms. This could be represented by unit tests and/or tests of mathematical aspects of the algorithm.

Format for electronic delivery of report and programs

The preferred format for the report is a PDF file. You can also use DOC or postscript formats or as an ipython notebook file. As programming language we prefer that you choose between C/C++, Fortran2008 or Python. The following prescription should be followed when preparing the report:

- Use your github address to hand in your projects.
- Make a folder for each project. For each project you should have three folders: one for the code files, one for the report and finally a folder with specific benchmark calculations. The latter can be in the form of output from your code for a selected set of runs and input parameters.

Finally, we encourage you to work two and two together. Optimal working groups consist of 2-3 students. You can then hand in a common report.

Project 4, Molecular dynamics: atomic modeling of argon, deadline April 29

In this project we will implement a model called molecular dynamics (MD). MD allows us to study the dynamics of atoms and explore the phase space. The atoms interact through a force given as the negative gradient of a potential energy function. With this force, we can integrate Newton's laws. While exploring the phase space, we will sample statistical properties of the system like energy, temperature, pressure, diffusion constant and so on.

We will also learn how to do write object oriented code with classes and subclasses. You are given a code skeleton that contains the basic structure of an MD code, but you will have to implement many of the functions yourself. We think that *learn by example* is a great way to learn how to proper object orient your code, but you can of course start from scratch if you want. You can find the code here: `https://github.com/andeplane/molecular-dynamics-fys3150`, but you should fork[1] the repository. Go to `https://github.com/andeplane/molecular-dynamics-fys3150` (remember to be logged in) and fork the project. After pressing the Fork button, you should have a fork that you can clone with *git clone git@github.com:<username>/molecular-dynamics-fys3150.git*.

After you have downloaded the repository, start to compile and run the program, using either Qt or your preferred options. The program should create 100 argon atoms and place them uniformly inside a cubic box with sides 10 Angstroms. Each atom is given a velocity according to the Maxwell-Boltzmann distribution so that

$$P(v_i)\mathrm{d}v_i = \left(\frac{m}{2\pi k_B T}\right)^{1/2} \exp\left(-\frac{mv_i^2}{2k_B T}\right)\mathrm{d}v_i, \tag{1}$$

where m is the mass of the atom, k_B is Boltzmann's constant and T is the temperature. We recognize this as a normal distribution with zero mean and standard deviation $\sigma = \sqrt{k_B T/m}$. The program will evolve the system in time with no forces so that all atoms move in a straight line. It will create a file called *movie.xyz* containing all the timesteps ready to visualize with for example VMD[2] or Ovito[3]. You should start with the fun part - to look at the simulation in either of these programs. We then strongly recommend you to, before you start, look at the code structure and try to understand how the different classes are connected to each other and how a typical timestep is computed. You can skip understanding the contents of *unitconverter.cpp*, *vec3.cpp* and *random.cpp*.

I) These are two very important steps that come even before part a. First, spend 30 minutes figuring out what molecular dynamics is and what problems it can be used to understand. What areas of physics can it be used in? What about chemistry and biology? What you find here should be part of your introduction section in your report.

[1] A fork is a copy of a repository that will be added to your GitHub account so that you can continue working on the project. After forking a repository, you can commit, push and pull to your version.

[2] Download VMD from `http://www.ks.uiuc.edu/Research/vmd/`

[3] Download Ovito from `http://www.ovito.org/index.php/download2`

II) Run the program, visualize the output with VMD or Ovito (as mentioned above). Now, spend 30 minutes looking at the code and understand how the output is produced and try to understand every step in the code. The best way to do this is to start at the top of the *main()* function and follow every line and every function call, step by step. **As a part if your report, you should explain the code structure and draw a flow chart (a diagram showing how the program is executed).** This part is extremely important and will save you a lot of time. And of course, please ask us teachers if you have any questions!

a) When working with MD, the system size is usually limited by available computer resources. A typical large MD simulation (with a parallelized code) contains a few million atoms corresponding to a system much smaller than a cubic micron. In order to get rid of boundary effects, we usually apply periodic boundary conditions so that we simulate a system of infinite size. You need to implement the *applyPeriodicBoundaryConditions* function in the *System* class. After doing so, run the program again and notice how the atoms now are contained inside a box. Discuss the benefits of this strategy.

b) The atoms are usually given velocities according to the Maxwell-Boltzmann distribution (you should discuss why this is the case) which will result in a nonzero net momentum in the system. Implement the *removeMomentum* function in the *System* class so that the net momentum is zero.

c) The atoms are now uniformly distributed in space. This is not very physical, so we should place the atoms in a crystal structure. When we later implement the Lennard-Jones potential, we will see that the face-centered cubic (FCC) lattice is a stable structure for the potential (it is actually the crystalline structure of argon). A lattice is built up by *unit cells* - a group of atoms - so that a larger system can be created by repeating these cells in space. An FCC lattice unit cell of size b Å (this is the lattice constant) consists of four atoms with local coordinates

$$\mathbf{r}_1 = 0\hat{\mathbf{i}} + 0\hat{\mathbf{j}} + 0\hat{\mathbf{k}}, \tag{2}$$

$$\mathbf{r}_2 = \frac{b}{2}\hat{\mathbf{i}} + \frac{b}{2}\hat{\mathbf{j}} + 0\hat{\mathbf{k}}, \tag{3}$$

$$\mathbf{r}_3 = 0\hat{\mathbf{i}} + \frac{b}{2}\hat{\mathbf{j}} + \frac{b}{2}\hat{\mathbf{k}}, \tag{4}$$

$$\mathbf{r}_4 = \frac{b}{2}\hat{\mathbf{i}} + 0\hat{\mathbf{j}} + \frac{b}{2}\hat{\mathbf{k}}. \tag{5}$$

You can now create $N_x \times N_y \times N_z$ such unit cells next to each other to form a larger system so that the origin of unit cell (i, j, k) is

$$\mathbf{R}_{i,j,k} = i\hat{\mathbf{u}}_1 + j\hat{\mathbf{u}}_2 + k\hat{\mathbf{u}}_3, \tag{6}$$

where $i = 0, 1, ..., N_x - 1$, $j = 0, 1, ..., N_y - 1$, $k = 0, 1, ..., N_z - 1$. The unit vectors of the unit cells are scaled with the lattice constant b so that

$$\hat{\mathbf{u}}_1 = b\hat{\mathbf{i}}, \quad \hat{\mathbf{u}}_2 = b\hat{\mathbf{j}}, \quad \hat{\mathbf{u}}_3 = b\hat{\mathbf{k}}. \tag{7}$$

Implement the function *createFCCLattice(int numberOfUnitCellsEachDimension, double latticeConstant)* in the *System* class. Remember to remove the 100 atoms that are

created as the code is right now. Use lattice constant $b = 5.26$ Å. Remember to update the system size so the *applyPeriodicBoundaryConditions* function works properly. If you now visualize the result of the program, you should see the nice crystallic structure in the beginning of the simulation. What is the density ρ in your system?

d) While this looks nice, we need to implement the computation of forces in order to see interesting physics. In this project, we will use the Lennard-Jones potential[4] which calculates the energy between two atoms i and j as

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

where $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ is the distance from atom i to atom j, ϵ is the depth of the potential well (units energy) and σ is the distance at which the potential is zero. For argon, optimal values of the parameters are:

$$\frac{\epsilon}{k_B} = 119.8 \text{ K}, \sigma = 3.405 \text{ Å}. \tag{9}$$

In our code, we use the so called MD units (see appendix B). This potential actually reproduces equilibrium thermodynamic properties that are in good agreement with experimental values for argon. The equation of state for this system is the van der Waals equation of state. Phases such as solid, liquid and gas are reproduced from this simple model with phase transitions and everything which is remarkable.

The total potential energy V in the system is computed by summing over all pairs of atoms (counting each pair only once)

$$V = \sum_{i>j} U(r_{ij}). \tag{10}$$

The force is calculated by taking the negative gradient of the potential

$$\mathbf{F}(r_{ij}) = -\nabla U(r_{ij}), \tag{11}$$

giving the x-component (the other components are calculated the same way)

$$F_x(r_{ij}) = -\frac{\partial U}{\partial r_{ij}} \frac{\partial r_{ij}}{\partial x_{ij}}, \tag{12}$$

where x_{ij} is the x-component of $\mathbf{r}_{ij}$.

Calculate (analytically) the force and implement the *calculateForces* function in the LennardJones class. You will have to take care of the periodic boundary conditions[5]. Now run the simulation with 5 unit cells in each dimension (500 atoms) for different initial temperatures. Can you tell if the system is in a solid state just by looking at the system in VMD/Ovito? At what initial temperature is the crystal melting?

[4] See `http://en.wikipedia.org/wiki/Lennard-Jones_potential` for more details.

[5] You could use the minimum image convention as described here `http://en.wikipedia.org/wiki/Periodic_boundary_conditions`.

e) The time integrator used in the skeleton code is the Euler-Cromer method. In MD, one usually uses the Velocity Verlet integrator. It is a so called symplectic integrator. Other algorithms tend to drift the energy over time which is something we do not want. The Velocity Verlet algorithm consists of three steps (in addition to computing the forces)

$$\mathbf{v}(t + \Delta t/2) = \mathbf{v}(t) + \frac{\mathbf{F}(t)}{m}\frac{\Delta t}{2} \tag{13}$$

$$\mathbf{r}(t + \Delta t) = \mathbf{r}(t) + \mathbf{v}(t + \Delta t/2)\Delta t \tag{14}$$

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t + \Delta t/2) + \frac{\mathbf{F}(t + \Delta t)}{m}\frac{\Delta t}{2}. \tag{15}$$

Implement this algorithm in the *integrate(System *system, double dt)* function in the *VelocityVerlet* class and use it instead of the *EulerCromer* integrator. Note that the first step assumes that you already have computed the forces, so you need to take care of that. Compare the two integrators and how the fluctuations in the total energy scales as the timestep Δt (plot the standard deviation of the total energy as a function of timestep, $\sigma_E(\Delta t)$).

f) We now have a working Molecular dynamics code! Congratulations, well done. The next step is to calculate some physical properties of the system. The easiest properties to measure are the kinetic and potential energy (calculated in the LennardJones class). The kinetic energy is of course defined as

$$E_k = \sum_{i=1}^{N_{\text{atoms}}} \frac{1}{2} m_i v_i^2, \tag{16}$$

where m_i and v_i is the mass and the speed of atom i. You can use the function *atom->velocity.lengthSquared()* to calculate the dot product of the velocities.

You can also calculate an estimate of the temperature through the equipartition theorem[6]

$$\langle E_k \rangle = \frac{3}{2} N_{\text{atoms}} k_B T. \tag{17}$$

We can use this to define an *instantaneous* temperature

$$T = \frac{2}{3}\frac{E_k}{N_{\text{atoms}} k_B}. \tag{18}$$

All of these quantities should be calculated in the *StatisticsSampler* class. All the statistical quantities should be saved to a file which needs to be implemented in the *saveToFile* function in the same class.

As you will see, since the temperature is proportional to the kinetic energy, its initial value will drop in the beginning when the system is initialized in the crystal structure. Why does this happen? What happens to the total energy? In order to have a system with final temperature T, what initial temperature T_i is needed? What is the ratio T/T_i? Again visualize with *VMD* or *Ovito* and use your eyes to determine at what temperature the system actually melts at. After the temperature drops, it will reach a more or less stable value. The system is then said to be in an equilibrium state.

[6] See for example D. Schroeder's *An Introduction to Thermal Physics* for details.

g) The last exercise is to compute the diffusion constant and use this to measure the melting temperature. We use the Einstein relation that relates the so called mean square displacement (MSD) to the diffusion constant

$$\langle r^2(t) \rangle = 6Dt, \tag{19}$$

where D is the diffusion constant, t is time. The mean square displacement for atom i is calculated as

$$r_i^2(t) = |\mathbf{r}_i(t) - \mathbf{r}_i(0)|^2, \tag{20}$$

where $\mathbf{r}_i(t)$ is the position of atom i at time t. When you implement this you need to add the initial position as a new property in the *Atom* class. You also need to update the *applyPeriodicBoundaryConditions* so that $r_i^2(t)$ is the actual displacement as if no periodic boundary condition has occured. Why is this important?

We can use the diffusion constant to find the melting temperature. Atoms in a solid will not diffuse much, so a solid will have a diffusion constant close to zero. Now solve the Einstein relation for the diffusion constant D and add a function in *StatisticsSampler* measuring this quantity. Plot the diffusion constant as a function for different temperatures T and use this to find an estimate for the melting temperature. Use what you found in f) to make sure you simulate temperatures both above and below the melting temperature. Also remember not to use the initial temperature, but the measured temperature after the system has reached equilibrium. You can use the builtin *UnitConverter* to convert the temperature to SI units as illustrated in the top of the *main* function.

If you want (**not required, but interesting if you have time**), compare the melting temperature to experimental values for argon. Note that the lattice constant b we have used corresponds to a very high density. You may want to change the lattice constant so that the density matches the experimental setup you found.

Appendices

A FYS2160 - what is really going on here?

The idea of an MD simulation is to sample microstates from a statistical ensemble. In our simulation, we have a constant number of atoms N, a constant volume V and a (more or less, depending on our integrator) constant energy E. This corresponds to the microcanonical ensemble (NVE).

Although we produce the dynamics of atoms, we should not see MD as a model that will give the true trajectories of the atoms, but rather a model that allows us to explore the phase space according to the probabilities we can calculate with statistical mechanics. A fundamental assumption in any MD simulation is the ergodic hypothesis. It states that[7] *"over long periods of time, the time spent by a system in some region of the phase space of microstates with the same energy is proportional to the volume of this region, i.e., that all accessible microstates*

[7] `http://en.wikipedia.org/wiki/Ergodic_hypothesis`.

are equiprobable over a long period of time". This means that by evolving the atoms through time will visit microstates of the ensemble according to the true probabilities given by the ensemble.

B Units

The program uses by default a set of units where we have chosen the following four units

$$\begin{aligned} 1 \text{ unit of mass } &= 1 \text{ a.m.u} = 1.661 \times 10^{-27} \text{ kg}, && (21)\\ 1 \text{ unit of length } &= 1.0 \text{ Å} = 1.0 \times 10^{-10} \text{ m}, && (22)\\ 1 \text{ unit of energy } &= 1.651 \times 10^{-21} \text{ J}, && (23)\\ 1 \text{ unit of temperature} &= 119.735 \text{ K}. && (24) \end{aligned}$$

Boltzmann's constant is then equal to one (convince yourself about this), and other units can be derived by using known relations. We can for example find the unit of time by using $E = mc^2$. We have that

$$\text{Energy} = \text{Mass} \times \frac{\text{Length}^2}{\text{Time}^2}, \tag{25}$$

which can be solved for time

$$\text{Time} = \text{Length} \times \sqrt{\frac{\text{Mass}}{\text{Energy}}}. \tag{26}$$

By inserting the values above, we get

$$1 \text{ unit of time } = 1.0 \times 10^{-10} \sqrt{\frac{1.661 \times 10^{-27}}{1.651 \times 10^{-21}}} \text{ s} = 1.00224 \times 10^{-13} \text{ s}. \tag{27}$$

This way, we can continue working out the values of the other units. What are the units of force? And pressure? In the project, a class that calculates these values is given.

C Visualizations in VMD

C.1 Installation

C.1.1 OS X

The installation on Mac OS X should be fairly simple. Just visit `http://www.ks.uiuc.edu/Development/Download/download.cgi?PackageName=VMD` and download the version (the 1.9.1 version works, but feel free to try the 1.9.2 beta) called *MacOS X OpenGL, CUDA (32-bit Intel x86)*. You will need to create an account, but don't worry. They won't send you any spam. Once you have downloaded the file, open the dmg-file and copy *VMD 1.9.1* into your Applications folder and start the program.

C.1.2 Linux (computer lab)

The installation on Linux requires that we compile the source code. Since we don't have sudo access on the computer lab machines we need to configure the installation path. To simplify this process, we have written an installation script in Python that you can find here: `http://folk.uio.no/anderhaf/fys3150/install_vmd_linux.py`. Just download the script and run it with *python install_ vmd_ linux.py*. When you run the script, *VMD* will be installed in a folder called *VMD* in the directory you run the script from. The same script should work on Ubuntu. The compiled executable is then put in *./VMD/bin/* so that you can run vmd with the command *./VMD/bin/vmd*.

C.2 Usage

VMD is a great tool to look at MD simulations. When you open VMD, you can load the file *movie.xyz* by clicking *File, New Molecule* and browsing to your file. Click *Load* and the file is loaded into VMD. The default representation of atoms isn't very beautiful, so go to *Graphics, Representations* and choose for example *VDW* from the *Drawing Method* list. You can make the spheres smaller by choosing a smaller *Sphere Scale*.