TCCM homework 3: A Molecular Dynamics Code

A. Ammar, Y. Damour, P. Reinhardt, A. Scemama

November 22, 2024

You can work in groups of up to three people. We expect that you will use Git to work collaboratively on your project. We expect the following files and structure in your submission:

- A `LICENSE` file specifying the license for your code.
- An `AUTHORS` file listing the names of all contributors.
- A `README.md` file providing a brief description of the directory structure and the project.
- An `INSTALL.md` file with clear instructions on how to compile and run the program.
- A `tests` directory containing tests to ensure that the program behaves as expected.
- (Optional) A `doc` directory for additional documentation, if the project requires more detail than the `README.md` file can provide.
- A `src` directory containing all the source files of your program.

Molecular dynamics (MD) simulates the movement of atoms based on their initial positions and velocities. In this tutorial, we will develop a molecular dynamics program to illustrate key concepts in MD simulations. Our program will read force field parameters and the initial positions of atoms from an input file. After each small displacement of atoms according to their velocities, the updated coordinates will be saved to an output file, enabling the creation of a video animation with an external tool.

Throughout this tutorial, we will use the following parameters:

- ϵ : 0.0661 j/mol
- σ : 0.3345 nm

Atom coordinates will be provided in nanometers.

1 Allocating and Freeing 2D Arrays

You will need to allocate two-dimensional arrays. The following functions should help you to allocate and free 2-dimensional arrays in `C`:

```c
double** malloc_2d(size_t m, size_t n) {

  // Allocate an array of double pointers with size m
  double** a = malloc(m*sizeof(double*));
  if (a == NULL) {
    return NULL;
  }


  // Allocate a contiguous block of memory for the 2D array elements
  a[0] = malloc(n*m*sizeof(double));
  if (a[0] == NULL) {
    free(a);
    return NULL;
  }

  // Set the pointers in the array of double pointers
  // to point to the correct locations
  for (size_t i=1 ; i<m ; i++) {
    a[i] = a[i-1]+n;
  }

  return a;
}

void free_2d(double** a) {
  free(a[0]);
  a[0] = NULL;
  free(a);
}
```

In Fortran 90, 2D arrays allocation can be handled with the `allocate` and `deallocate` commands,

```fortran
! Declare the array
double precision, allocatable :: a(:,:)
```

```fortran
! To check the allocation status
integer :: i_stat

! Allocate the 2D array
allocate(a(m, n), stat=i_stat)
if (i_stat /= 0) then
    print *, "Memory allocation failed!"
    stop
end if

! Deallocate the 2D array
deallocate(a)
```

2 Describing the Atoms

We will create functions to read the atomic data from an input file called `"inp.txt"`. The input file follows this format:

- The first line contains the number of atoms (`Natoms`).
- Each subsequent line contains the x, y, and z coordinates followed by the mass of an atom.

Create a function with the following prototype which reads the number of atoms from an **opened** input file:

```c
// using C
size_t read_Natoms(FILE* input_file);
```

```fortran
! using Fortran
integer function read_Natoms(input_file) result(Natoms)
    implicit none
    integer, intent(in) :: input_file
    integer :: Natoms
    ! TODO: read Natoms
end function read_Natoms
```

Then, write a function with the following prototype:

```c
// using C
void read_molecule(FILE* input_file,
                   size_t Natoms,
                   double** coord,
                   double* mass);

! using Fortran
subroutine read_molecule(input_file, Natoms, coord, mass)
    implicit none
    integer, intent(in) :: input_file
    integer, intent(in) :: Natoms
    double precision, intent(out) :: coord(Natoms,3)
    double precision, intent(out) :: mass(Natoms)
    ! TODO: read mass and coord
end subroutine read_molecule
```

This function reads the atomic coordinates and masses, and stores the data in the arrays `coord` and `mass` provided as parameters. `coord` is a two dimensional array allocated such that `coord[4][2]` (`coord(5,3)` in `Fortran`) returns the z coordinate of atom 5.

Write another function which takes as input the array of coordinates and returns internuclear distances between each pair in a two-dimensional array `distance` of size (`Natoms` $\times$ `Natoms`):

```
// using C
void compute_distances(size_t Natoms,
                       double** coord,
                       double** distance);

! using Fortran
subroutine compute_distances(Natoms, coord, distance)
    implicit none
    integer, intent(in) :: Natoms
    double precision, intent(in) :: coord(Natoms,3)
    double precision, intent(out) :: distance(Natoms,Natoms)
    ! TODO: calculate internuclear distance
end subroutine compute_distances
```

To use the `sqrt` function in `C`, you need to include `<math.h>`, and you also need to compile using the `-lm` option to link with the `libm.so` math library. You might need to put the `-lm` option at the end of the command line (after your files).

3 The Lennard-Jones potential

Write a function which takes as input ϵ, σ, the number of atoms and the array of distances, and computes the total potential energy

$$V = \sum_{i=1}^{\texttt{Natoms}} \sum_{j>i}^{\texttt{Natoms}} V_{\mathrm{LJ}}(r_{ij})$$

where $V_{\mathrm{LJ}}(r)$ is the Lennard-Jones potential :

$$V_{\mathrm{LJ}}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right].$$

To compute x^n in `C`, you will need to call the power function located in the `libm.so` library, defined in `math.h`.

```
#include <math.h>
x_power_n = pow(x,n);
```

We will now write a function to calculate the total potential energy of the system using the Lennard-Jones potential. This function will take the following inputs: ϵ, σ, the number of atoms (`Natoms`), and the array containing the distances between each pair of atoms. The total potential energy V is given by:

$$V = \sum_{i=1}^{\texttt{Natoms}} \sum_{j>i}^{\texttt{Natoms}} V_{\mathrm{LJ}}(r_{ij}) \tag{1}$$

where $V_{\mathrm{LJ}}(r)$ is the Lennard-Jones potential:

$$V_{\mathrm{LJ}}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]. \tag{2}$$

```
// using C
double V(double epsilon,
         double sigma,
         size_t Natoms,
         double** distance);
```

```
! using Fortran
double precision function V(epsilon, sigma, Natoms, distance)
    implicit none
    double precision, intent(in) :: epsilon, sigma
    integer, intent(in) :: Natoms
    double precision, intent(in) :: distance(Natoms,Natoms)
    ! TODO: compute the total potential V
end function V
```

4 Computing the total energy

We will now write functions to compute the total kinetic energy and the total energy of the system.

First, we will write a function to calculate the total kinetic energy of the system T. This function will take as input the number of atoms, the array of velocities ($3 \times$ `Natoms`) and masses (`Natoms`). The velocities are initialized to zero. The total kinetic energy T is given by:

$$T = \frac{1}{2} \sum_{i=1}^{\texttt{Natoms}} m_i \mathbf{v}_i^2 \tag{3}$$

```
// using C
double T(size_t Natoms,
         double** velocity,
         double* mass);
```

```
! using Fortran
double precision function T(Natoms, velocity, mass)
    implicit none
    integer, intent(in) :: Natoms
    double precision, intent(in) :: velocity(Natoms,3)
    double precision, intent(in) :: mass(Natoms)
    ! TODO: compute the total kinetic energy T
end function T
```

Next, we will write a function to compute the total energy of the system, which is the sum of the total kinetic energy T and the total potential energy V. The total energy E is given by:

$$E = T + V. \tag{4}$$

5 Computing the acceleration

The acceleration vector for each atom is given by:

$$\mathbf{a}_i = -\frac{1}{m_i} \nabla_i V$$

The analytical expression of the acceleration reads as:

$$\begin{cases} a_{xi} = -\dfrac{1}{m_i} \displaystyle\sum_{j=1}^{\texttt{Natoms}} U(r_{ij}) \dfrac{x_i - x_j}{r_{ij}} \\ a_{yi} = -\dfrac{1}{m_i} \displaystyle\sum_{j=1}^{\texttt{Natoms}} U(r_{ij}) \dfrac{y_i - y_j}{r_{ij}} \\ a_{zi} = -\dfrac{1}{m_i} \displaystyle\sum_{j=1}^{\texttt{Natoms}} U(r_{ij}) \dfrac{z_i - z_j}{r_{ij}} \end{cases} \tag{5}$$

where

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

We need to write a function that computes the acceleration vector for each atom and stores it in a double precision array.

```
// using C
void compute_acc(size_t   Natoms,
                 double** coord,
                 double*  mass,
                 double** distance,
                 double** acceleration);
```

```
! using Fortran
subroutine compute_acc(Natoms, coord, mass, distance, acceleration)
    implicit none
    integer, intent(in) :: Natoms
    double precision, intent(in) :: coord(Natoms,3)
    double precision, intent(in) :: mass(Natoms)
    double precision, intent(in) :: distance(Natoms,Natoms)
    double precision, intent(out) :: acceleration(Natoms,3)
    ! TODO: calculate acceleration
end subroutine compute_acc
```

6 Implementing the molecular dynamics

The molecular dynamics simulation is based on the Verlet algorithm, which updates the position and velocity of each atom at each time step. The algorithm consists of two main equations:

1. Position update:

$$\mathbf{r}^{(n+1)} = \mathbf{r}^{(n)} + \mathbf{v}^{(n)}\Delta t + \mathbf{a}^{(n)}\frac{(\Delta t)^2}{2} \tag{7}$$

2. Velocity update:

$$\mathbf{v}^{(n+1)} = \mathbf{v}^{(n)} + \frac{1}{2}\left(\mathbf{a}^{(n)} + \mathbf{a}^{(n+1)}\right)\Delta t \tag{8}$$

where

- (n) denotes the index of the current step.
- $\mathbf{r}$ is the position vector.
- $\mathbf{v}$ is the velocity vector.
- $\mathbf{a}$ is the acceleration vector.
- Δt is the time step.

We will write a subroutine that implements the Verlet algorithm. The process involves several steps for each iteration, starting with initial velocities set to zero and initial coordinates read from the input file.

6.1 Steps for Each Iteration:

Initialize the calculation with initial coordinates and $\mathbf{v}_i = \mathbf{0}$. Next, compute the acceleration vector using (5). At each step (n):

1. Compute the coordinates $\mathbf{r}^{(n+1)}$ using equation (7)
2. Update the part of $\mathbf{v}^{(n+1)}$ which depends on the current acceleration vector $\mathbf{a}^{(n)}$ (equation (8))
3. Update the acceleration $\mathbf{a}^{(n+1)}$ (equation (5))
4. Finalize the update of the $\mathbf{v}^{(n+1)}$ using the $\mathbf{a}^{(n+1)}$ (equation (8))

We will consider the following parameters for this simulation:

- $\Delta t = 0.2$
- Total number of steps is 1000.

Write into a file the coordinates of the trajectory in xyz format. This corresponds to writing every $M = 10$ steps:

- A line containing the number of atoms.
- A line containing a comment. This line can contain the kinetic, potential and total energy for plotting, and checking that the total energy is conserved.
- For each atom, a line containing the atomic Symbol (Ar for Argon atoms), followed by its x, y and z coordinates

The xyz can be opened with software like Molden or Jmol to play the animation of the dynamics.