Fortran programming and molecular dynamics simulation of argon module

Teacher Guide

Introduction

In this module students will learn basic programming skills using the Fortran programming language while exploring kinetic molecular theory and molecular dynamics (MD) simulations. Students will create hypotheses about how changing the parameters in their programs will affect the simulation. The data produced from the simulations such as temperature, potential energy, and coordination, will be graphed and analyzed by the students and compared to their hypotheses.

All of the Fortran files referenced in this guide can be downloaded from our website by cliking the "Sample files and fortran codes" link under the "Fortran programming and molecule dynamics simulation of liquid argon" section.

Basic programming

Introduction to Fortran and "average program"

In the first part of the module, we introduce the Fortran programming language and teach students how to write a basic program that reads a list of numbers from a file, calculates the average, display it on the screen, and also writes that average to a new file.

To write and run programs in Fortran you need to use a simple text editor to write the code and a compiler to convert it into an executable that your computer can run. With Internet access, you can use the Tutorials Point Fortran development environment, which allows you to write Fortran code, compile it, and run it in a single browser window. You can also install a Fortran compiler on your computer; see the Appendix for details.

Let's look at a basic Fortran program (`ft_average.f95`, included as a sample file) that calculates the average of the numbers in a file called `numbers.txt`. Below the code is a line-by-line annotation. Note that in Fortran any line beginning with the character `!` is a "comment," which is not translated into the executable program by the compiler. These comments are simply to help the human reader of the code understand it better. All of our code examples include comments to help make them more human readable.

```
1   program average
2   implicit none
3   integer :: i
4   integer, parameter :: numlines = 10
5   real :: run_sum
6   real :: x
7
8   ! open the file for reading
9   open(unit = 11, file = 'numbers.txt')
10
11  ! initialize the running sum
12  run_sum = 0.0
13
14  ! read the number on each line into x,
15  ! add to the running sum
16  do i = 1, numlines
17     read(11, *) x
18     run_sum = run_sum + x
19  end do
20
21  close(11)
22  ! calculate average by dividing running sum by number of lines
23  run_sum = run_sum / real(numlines)
24
25  ! open a file for writing and write the average to it
26  open(unit = 91, file = 'out.txt')
27  write(91, *) run_sum
28  close(91)
29  print*, run_sum
30
31  end program average
```

Here is a line-by-line annotation, explaining the function of this program:

- Line 1: Every Fortran program begins with the statement `program` followed by the name of the program.
- Line 2: `implicit none` prevents the use of any undeclared variables. It is recommended to always use this statement.
- Lines 3–6: Variable declaration. A variable is simply a container for a value. We use them throughout the program to store different values, such as numbers. We must

"declare" variables before we use them. Variables have types which dictate what can be stored in them. For example, in line 3, `i` is declared as an integer, so it stores a number in $\ldots, -3, -2, -1, 0, 1, 2, 3 \ldots$ On the other hand, `run_sum` and `x` are declared as `real`, so these can be used to store decimal (floating point) numbers, such as 3.2 or 1.6. The `parameter` statement in line 4 allows us to specify a value, 10, that the variable `numlines` will have for the entire program. Attempting to set `numlines` to any other value later in the program will result in an error.

- Line 9: The `open` statement opens a file for reading and assigns it a number. In this case, we assign the file `numbers.txt` the number `11`. We can read or write to this file using this number from now on.

- Line 12: We *initialize* the variable `run_sum` to the value `0.0`. This is because when we declare the variable, it will automatically take on a (seemingly) random value from memory. Since we need to use this variable to calculate a running sum, we will not get the right answer unless we set it to 0.0 before hand. Note that 0.0 is not the same as 0 (0.0 is of type `real`, and 0 is of type `integer`

- Lines 16–19: This is a type of loop called a `do` loop in Fortran. The code in between `do i = 1, numlines` and `end do` is executed `numlines` times. In line 17, we use the `read` statement to read everything (`*`) from file `11`, and store what we read into the variable `x`. In line 18, we set `run_sum` to whatever its current value is plus the current value of `x`. As the code repeats in the loop, the `run_sum` variable stores a cumulative sum of all the numbers in the file, while `x` only stores the number that was last read from the file.

- Line 21: The `close` statement closes a file. We do this because we are done with the file.

- Line 23: To calculate the average, we simply divide the sum of all the numbers by the count of numbers in the file. Note that `numlines` is originally declared as an integer, so to ensure we get a decimal result we use `real(numlines)` to turn it into a `real`.

- Line 26: We open a file called `out.txt` and assign it the `unit` number `91`.

- Line 27: We write the average to the file with `unit 91`.

- Line 28: Now we close file `91` because we will not write anything else to it.

- Line 29: `print*,` prints all of the variables listed after it to the console.

- Line 31: We end the program.

Now, let's run the program. If you're using the Tutorials Point online environment:

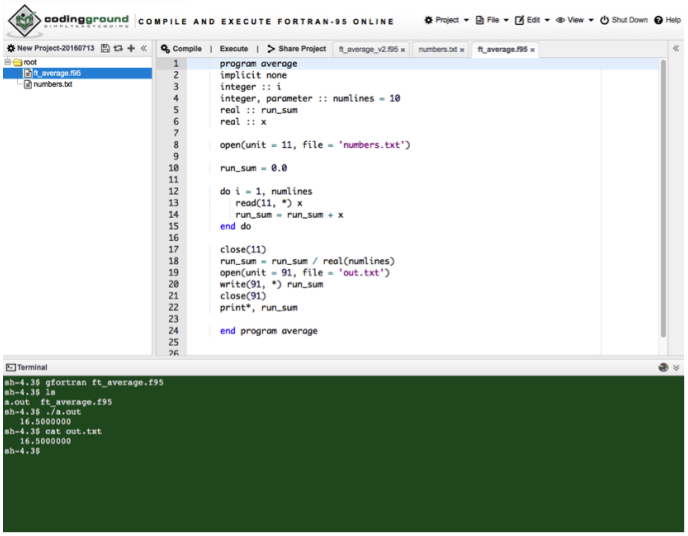

Figure 1: The Coding Ground Fortran online environment. The left pane shows the list of files and directories, the right pane is the text editor, and the bottom green pane is the Terminal.

1. Click on the `root` folder. Then click "File → Upload File" and select `ft_average.f95` and `numbers.txt` from wherever they are on your computer.

2. Note the split of the environment into several panes. The left contains a list of your files, the bottom is the Terminal, the center is a text editor, and the right contains links to other tutorials (Figure 2). The Terminal is where you can run the compiler to make your program and then run it. In the Terminal, type `gfortran ft_average.f95` and hit the enter/return key.

3. If all is successful, the compiler will not give you any warnings or errors and generated an executable, which is by default called `a.out`. To check that this file exists, type `ls` in the terminal and hit enter to see a list of all files in your current directory. You should see that `ft_average.f95`, `a.out`, `numbers.txt`, and perhaps the default Tutorials Point `main.f95` are all there.

4. Finally, run the program. Type `./a.out` and hit enter. The average of the ten numbers in `numbers.txt` will be printed to the screen. Type `ls` and hit enter to see the files in the directory now. An additional file called `out.txt` should have been created. Type `cat out.txt` to put the contents of that file onto the screen and verify that it also contains the average of the ten numbers in your file.

Now, you can verify that if you change the ten numbers in `numbers.txt` to a different ten numbers, you'll get the average of those numbers. If you change one of the lines to

something that isn't a number, the program will crash. If you add more than ten numbers to `numbers.txt` but don't modify the code, those numbers won't be included in the average. Lastly, if your `numbers.txt` has fewer than ten numbers, the program will also crash.

What if you want to calculate the average of more than ten numbers? In the code, we specified

```
integer, parameter :: numlines = 10
```

as a parameter. If we change `10` to some other number, we can then read that number of lines from a file.

Writing slightly more interesting programs

So far, we have a program that reads in a certain amount of numbers and tells us the average. In that program, we read from files and wrote to files, did some calculations, and used a loop. We now also introduce another programming structure, called a conditional. Let's say we do not know how many numbers are in `numbers.txt`, and instead of specifying the number of lines as a parameter in our file, we just want to read until its end.

In the last program, we used the read function as

```
read(11, *) x
```

to store a number in the variable `x`. An alternate way of using the read function is:

```
read(11, *, iostat=iostatus) x
```

When read is used in this way, instead of the programming crashing when there are no more lines or something of the wrong type is read, it puts a number in the variable `iostatus`. If that number is 0, then read worked fine. If the number is negative, `read` reached the end of the file. If the number is greater than zero, then `read` didn't work for some other reason (for example, in our case maybe one of the lines wasn't a number but a word).

This `iostat` variable combined with a programming structure called conditionals will allow us to implement our goal of taking the average of several numbers without knowing how many there are beforehand. A conditional in Fortran is shown below:

```
if ( 6 > 5 ) then
   print*, "Hey there"
end if
```

The statement in parenthesis is called the condition. First, the program evaluates whether the condition is true or false (in this case $6 > 5$ is always true). If it is, it will print `Hey there!` Otherwise, it does nothing. The condition can contain variable names, in which case it references that value of the variable. Here's a more complicated example. Let's say your program already has a grade for a student's assignment stored in the variable `grade`

```
if (grade == 100) then
   print*, "Perfect score!"
else if (grade > 90) then
   print*, "A"
else if (grade > 80) then
   print*, "B"
else if (grade > 70) then
   print*, "C"
else if (grade > 65) then
   print*, "D"
else
   print*, "F"
end if
```

In this case, first the program checks to see if the grade is 100. If it is, it prints `Perfect score!` and does nothing else. If the grade is not 100, it goes to the first `else if` statement and checks if the grade is over 90. If it is, it then prints `A` and does nothing else. If all of the conditions (i.e. == 100, > 90, >80, >70,>65) are false, it simply prints `F`. Below is a new version of the average program, `ft_average_v2.f95`, which uses conditionals and the `iostat` part of the read statement to read `numbers.txt` until the end of the file. Many of the lines are exactly the same as original version, but the changes are detailed below.

```
1        program average
2        implicit none
3        integer :: numlines
4        integer :: iostatus
5        real :: run_sum
6        real :: x
7
8        open(unit = 11, file = 'numbers.txt')
9
10       ! initialize running sum and number of lines to 0
11
12       run_sum = 0.0
13       numlines = 0
14
15       do
16          read(11, *, iostat=iostatus) x
17          ! if read was successful
18          if (iostatus == 0) then
19             run_sum = run_sum + x
```

```
20              numlines = numlines + 1
21           ! if read failed
22           else if (iostatus > 0) then
23              print*, "Something is wrong with input...exiting..."
24              ! STOP quits the program
25              STOP
26           ! if read found end of file
27           else
28              ! exit terminates the do loop
29              EXIT
30           end if
31        end do
32
33
34        close(11)
35        run_sum = run_sum / real(numlines)
36        open(unit = 91, file = 'out.txt')
37        write(91, *) run_sum
38        close(91)
39        print*, run_sum
40
41        end program average
```

- Line 3: In the first program `numlines` was a parameter, but in this case we will update `numlines` each time we read a line.
- Line 13: We will essentially count the number of lines we read in, so `numlines` has to be set to 0 to begin with.
- Line 15–31: `do`, with nothing following says to repeat lines between `do` and `end do` infinitely (either until something ends the loop from within or you kill the program).
- Line 16: Read the contents of a line into x, and also store the `iostat` into `iostatus`.
- Lines 18–20: If `iostatus` is 0, that means that the read of the line completed successfully, so we'll add the number into our running sum, and also add 1 to `numlines`.
- Lines 22–25: If `iostatus` is > 0 then there was some error. In this case we just print a message and crash the program with the `STOP` command.
- Lines 27–29: The only other option for `iostatus` is to be negative if the program gets to this point. If `iostatus` is negative, then we know `read` got to the end of the file. The `EXIT` command kills the loop.

Compile and run this program as you did before by uploading it to the Coding Ground site, typing `gfortran ft_average_v2.f95` to compile, and `./a.out` to run. Play around with changing the number of lines in the `numbers.txt` file; you will notice that the program will work now regardless of how many numbers you give it.

A final example

After writing the average program, students can write slightly more complicated programs. One example we suggest is a program that reads in a list of names with grades. If a student's grade is higher than some threshold, the program prints their name, followed by "Yay!" This code example is included as `ft_grades.f95`.

Molecular dynamics

Theoretical background

Molecular dynamics (MD) simulations are a type of computer simulation that allow scientists to observe the movements of atoms or molecules in nanoscale systems. At the beginning of a simulation ($t = t_0$), we specify the positions and velocities of every particle in our system. We define a potential function that allows us to calculate the energy of interaction between particles as a function of their positions. From classical mechanics, force can be calculated as the derivative of potential energy with respect to distance, and the force acting on each particle in the system can be determined. By Newton's second law, we can determine the acceleration of each particle, and from the equations of motion we can then calculate velocity and position of every particle at $t = t_0 + \Delta t$. In practice, Δt (the "timestep") is a small number, typically on the order of femtoseconds (1 fs = 10^{-15} s).

The simulation program used in this module was written to closely reflect one of the first published MD simulations. In 1964, Aneesur Rahman published a report of a simulation of liquid argon at 94.4 K with 864 atoms [1]. The parameters used in our code come from this paper.

Potential function: Lennard-Jones potential

Rahman's 1964 simulation used a pairwise potential function to model the interaction between atoms called the Lennard-Jones potential. This function is given by the equation:

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

where r_ij where r_{ij} is the distance between atoms i and j and ϵ and σ are potential parameters specific to the atoms being modeled. As shown in Figure **??** the Lennard-Jones

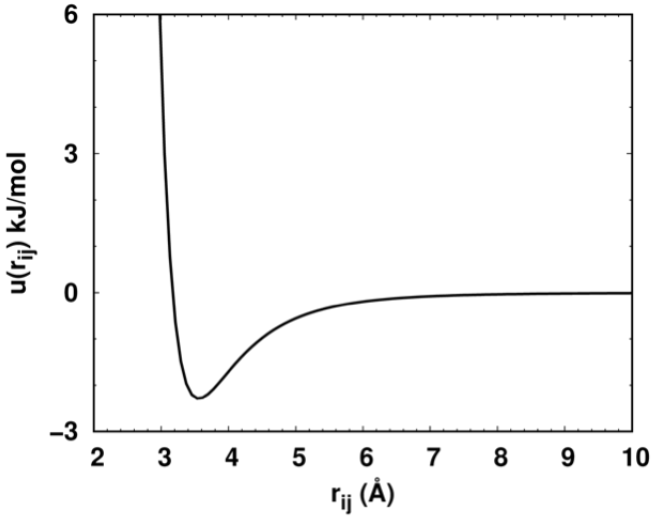

Figure 2: Plot of a Lennard-Jones potential function.

potential is repulsive (positive potential) at short distances but enters into an attractive regime (negative potential) prior to decaying to zero. The ϵ parameter has units of energy and is the depth of the potential well; thus, the minimum of the Lennard-Jones potential occurs at $-\epsilon$. σ has distance units and is the distance at which the potential function $U(r_{ij}) = 0$.

An excellent resource for explaining the Lennard-Jones potential is the PhET Atomic Interactions simulation. This small applet depicts two particles moving back and forth in one dimension while simultaneously showing the corresponding potential on a plot of the Lennard-Jones potential function. Options are even available to display force vectors on the atoms and manually adjust potential parameters to see their effects.

The simulation algorithm and its implementation

The first part of each timestep involves the calculation of the forces on all the atoms. First, the distances between all atoms in the system are calculated. If the distance between two atoms is within a defined cutoff, the derivative of equation (1) with respect to r is used to calculate the force acting on the atoms. The total force acting on each atom is calculated, and then used to determine each atom's acceleration vector. From the acceleration vector, a numerical integration is performed to propel each atom's position and velocity one timestep forward.

We have provided three Fortran files to conduct the simulations: `ft_main.f`, `ft_mdroutines.f`, and `ft_parameters.f`. The following is a brief description of the files:

- `ft_mdroutines.f` contains subroutines (essentially small, callable pieces of code that perform specific operations) that are essential for MD simulations. Among these are the calculation of forces and the integration of motion. This file is commented, so take a look at it; with the programming knowledge from the previous part of this guide you might be able to understand much of this file.
- `ft_parameters.f` contains all of the constant value parameters needed for the simulation, such as the size of the box, number of atoms in the simulation, mass of the atoms, Lennard-Jones potential parameters, physical constants, and various parameters that control how the simulation is run.
- `ft_main.f` contains the "main" MD program. This program performs the simulation by calling the subroutines from `ft_mdroutines.f` according to parameters specified in `ft_parameters.f`

Additionally, we provide an input configuration of 864 argon atoms called `Ar_Lattice.gro`. The `ft_parameters.f` file is preconfigured for an equilibration simulation of argon.

Running a simulation

Before we run a simulation, let's look at our initial structure. Open VMD and select "File → New Molecule" in the "VMD Main" window. Click "Browse..." to find the `end_structure.gro` file. Click "Open" in the "Choose a molecule file" window and then "Load" to load in the file. Adjust the representation so that you can better visualize the system. First select "Graphics → Representations" in the VMD Main window, and under the "Drawing Method" dropdown menu select "VDW." You will notice that the structure is simply an organized lattice of atoms, each a fixed distance apart. In particular, you might notice that this lattice is not very liquid-like. Since our goal is to simulate and calculate the properties of liquid argon, we must do something to get our system to actually look like liquid argon. Thus, the first thing we will do is run an **equilibration** simulation. In this simulation, we will start with the lattice-like structure provided, and simulate it at a constant temperature (94.4 K, at which argon is in the liquid state) until the structure more closely resembles liquid argon.

Now, close VMD and upload the MD Fortran files and `Ar_Lattice.gro` file to Coding Ground. Double click on the `ft_parameters.f` file. This file contains lines that control various simulation parameters, shown below:

```
1       MODULE PARAMETERS
2       IMPLICIT NONE
3       ! Number of atoms in system
4       integer, parameter              :: natoms = 864
5       ! Number of timesteps to run simulation
6       integer, parameter              :: nsteps = 2000
```

```
7         ! Atom name for output trajectory
8         character (len = 2), parameter :: aname  = 'Ar'
9         ! Lennard-Jones parameter sigma, nm
10        real, parameter                :: sigma  = 0.34
11        ! LJ cutoff, nm
12        real, parameter                :: R      = 0.765
13        ! Lennard-Jones parameter epsilon, J
14        real, parameter                :: eps    = 1.657E-21
15        ! Box dimension, nm
16        real, parameter                :: box    = 3.47786
17        ! Mass of each atom, kg
18        real, parameter                :: mass   = 6.6335209E-26
19        ! Timestep for simulation, ps
20        real, parameter                :: dt     = 1E-2
21        ! Whether or not to use thermostat; .TRUE. or .FALSE.
22        logical, parameter             :: Tcoupl = .TRUE.
23        ! Reference temperature for thermostat
24        real, parameter                :: Ref_T  = 94.4
25        ! Frequency to write to trajectory and log files
26        integer, parameter             :: stride = 10
27        ! Name of input file for simulation
28        character (len=*), parameter   :: inputfile = 'Ar_Lattice.gro'
```

Look at lines 21–24. Line 22 tells the program whether to use a **thermostat** to control the temperature of the system. When this is activated (`Tcoupl = .TRUE.`) the speed of every atom is rescaled to a target speed for the specified temperature, which is determined by the equipartition theorem:

$$v = \sqrt{\frac{N_{dof} k_B T}{m}} \tag{2}$$

where N_{dof} is the number of degrees of freedom (3 for single atoms, corresponding to translation in x, y, and z), k_B is the Boltzmann constant, T is the absolute temperature, and m is the particle mass. For argon, equation 2 gives a speed of 0.242 773 nm/ps. If we did not use a thermostat, the temperature would immediately "explode" due to the very close distances between lattice atoms causing large, repulsive forces. The atoms then accelerate quite quickly and reach a speed away from our target, even though their initial velocities correspond to the target temperature.

The temperature coupling used here is very basic; while it does allow us to run a simulation and get a reasonable starting structure for a "production run," the trajectory of this run is not suitable for analysis. The main reason for this is that this thermostat makes every atom in the system move at the same exact speed (note the difference between speed and velocity; each atom has the same speed, but every atom is moving in a different

direction, so they do not have the same velocities). The distribution of speed in this system is unrealistic, and if we were in fact to plot it it would be a vertical line.

However, after we do run the equilibration, we can use the structure it generates to run a production run and analyze the trajectory. Let's run the equilibration now:

1. Upload the four files detailed above to the Coding Ground environment.
2. Compile the program with the following line in the Coding Ground Terminal pane: `gfortran ft_parameters.f ft_mdroutines.f ft_main.f -o equilibration`
3. Run the simulation by executing the newly compiled program with: `./equilibration`

Assuming you have not modified the `nsteps` parameter, the simulation should run very quickly. After you have run the simulation you will have generated several files (in Coding Ground, you may have to click the refresh button in order for the files to appear):

- `trajectory.xyz` contains the coordinates of all of the atoms in the system at different time points in the simulation. Because this file can get rather large quickly, the program only writes the coordinates every 10 steps. You can modify this behavior by changing the `stride` parameter.
- `temperature.log` contains the temperature of the system at different time points. The frequency of output is also controlled by the `stride` parameter.
- `energy.log` contains the kinetic, potential, and total energy of the system at different time points in the system. The output frequency is also controlled by the `stride` parameter.
- `end_structure.gro` is a file that contains the coordinates and velocities of every atom in the system at the very end of the simulation.

Download `trajectory.xyz` to your computer and open it in VMD as you did before with the `Ar_Lattice.gro` file. By dragging the slider in the "VMD Main" window, you will be able to watch the movement of the atoms over 20 ps. Now, let's run the production run:

1. Before beginning, make sure you have downloaded all of the files from Coding Ground.
2. Rename the `end_structure.gro` to some other name like `equil_structure.gro`. Be sure not to use any spaces in the name and to maintain the ".gro" file extension. We will use this file as the input for the production run.
3. Open `ft_parameters.f` and change the following parameters:
    - First change `inputfile` to the name of your equilibrated structure (e.g. `equil_structure.gro`).

- Next change `Tcoupl` to `.FALSE.` Note the periods around the word FALSE. This will turn off temperature coupling in your simulation. We can do this because we are starting with a structure that already looks liquid like and has velocities that correspond to a reasonable liquid temperature (94.4 K. Since the structure is equilibrated, the temperature should stay close to our target without artificially maintaining it there algorithmically; moreover, our system will have a Maxwell-Boltzmann distribution of speed.
- We'll also change the duration for which we run the simulation. We previously ran the equilibration for 20 ps. We?ll run the production run for 50 ps. Change nsteps to 5000.
- Lastly, because we?re running the simulation for a much longer time, we should try to ensure that we don?t write files that are too massive. The stride to parameter tells the program how often to write coordinates to the trajectory file. For example, if stride is 10, the resulting trajectory file has coordinates for every 10 timesteps. Let?s set stride to 25.

4. Save and close the file. Compile and run the program with:
   ```
   gfortran ft_parameters.f ft_mdroutines.f ft_main.f -o production
   ./production
   ```

After running the simulation, you'll generate the same files as you did before. At this point, if you're using the Coding Ground environment, download all of the generated files to your desktop from both the equilibration and production runs and close the environment.

Simulation visualization

We can now view the production simulations trajectory in VMD. Load `trajectory.xyz` from the production run as before into VMD. Play around with the representations to get your scene to look the way you'd like. You can render an image of the display window and save it for later. To do this, click "File → Render..." to open the "File Render Controls" window (Figure 3). To make a good quality image, you should use the option "Tachyon (internal, in-memory rendering)." Click the "Browse..." button to select where you want to save your image. Click the "Start Rendering" button to make the image file.

Coordination structure

The radial distribution function, $g(r)$, is a function that describes the probability of finding two atoms in a system a distance r apart from each other. From a simulation trajectory, it can be easily calculated by determining the distances between each pair of atoms and counting how many of the pair distances fall in a certain range. This count is then normalized and expressed relative to the number of atoms one would expect to find in a spherical shell of radius r with the system's average density.

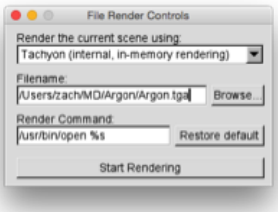

Figure 3: The VMD render controls GUI window.

From the radial distribution function (RDF), we can easily determine the radius of argon's first coordination shell; it is simply the distance to the first minimum of the RDF. Additionally, the integral of the RDF with respect to r gives us the coordination number, or number of argon atoms that are on average within a distance r. In this sense, if we integrate the RDF and find the value of its integral at the RDF?s first minimum, we know the number of atoms that are on average within the first coordination sphere of an argon atom.

VMD can calculate the RDF for us. To use this feature, load the trajectory into VMD as before. Select "Extensions → Tk Console." In the Tk Console, type `pbc set {34.7786 34.7786 34.7786} -all` and press Enter. Close the Tk Console and now select "Extensions → Analysis → Radial Pair Distribution Function g(r)." Select your trajectory in the "Use Molecule" dropdown menu. Type "all" in both the "Selection 1 and Selection 2" fields. Ensure the "Use PBC" checkbox is checked. Click "Compute g(r)" to have VMD calculate the RDF. It will open in a separate window.

You can save the file containing the function data by choosing "Export to ASCII matrix ..." from the "File" dropdown menu in the plot window. This exports the file as a text file that can be imported into nearly any data analysis software (e.g. Microsoft Excel, Numbers, OpenOffice, Mathematica, etc.)

Looking at temperature and energy of the system

When you consider the temperature and energy log files and the file from the RDF calculation calculation, there are several text files with simulation data in them. A convenient way to look at the results of the simulation would be to plot the data. As mentioned earlier, nearly any data analysis software including Microsoft Excel is capable of doing this. To import a given file into Excel,

1. Select "File → Import ..." to open the import window.

2. Click on the "Text file" radio button and then click "Import"

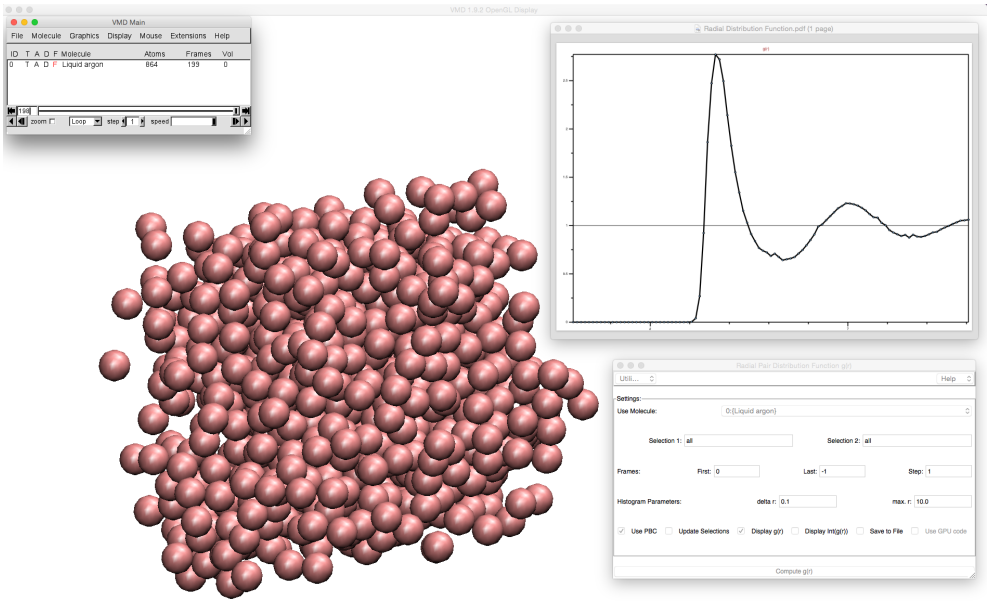

Figure 4: Liquid argon structure shown in VMD with plot of the radial distribution function and the calculate radial distribution function dialog box.

3. Select your text file. Because the files generated by the simulation have extension ".log" and the VMD files have extension ".dat," you may need to change the selection in the "Enable" dropdown menu from "Text Files" to "All Files." Click "Get Data" to open the Text Import Wizard.

4. Ensure the "Delimited" radio button is selected and click "Next >"

5. Under "Delimiters," uncheck "Tab" and check "Space." Ensure that "Treat consecutive delimiters as one" is checked. Click "Next >"

6. Use the "General" column data format and click "Finish."

7. Select the place in your Excel sheet you want to put the data and click "OK."

In the `energy.log` and `temperature.log` files, the first column and resulting x-axis of the graphs corresponds to time, in picoseconds (1 ps $= 10^{-12}$ s). In `energy.log`, the second, third, and fourth columns correspond to potential, kinetic, and total energies, respectively. These values are in joules. The second column of `temperature.log` is temperature in Kelvin.

Each temperature and energy file from both equilibration and production runs can be imported and plotted into Excel. Notice the differences between the temperature and

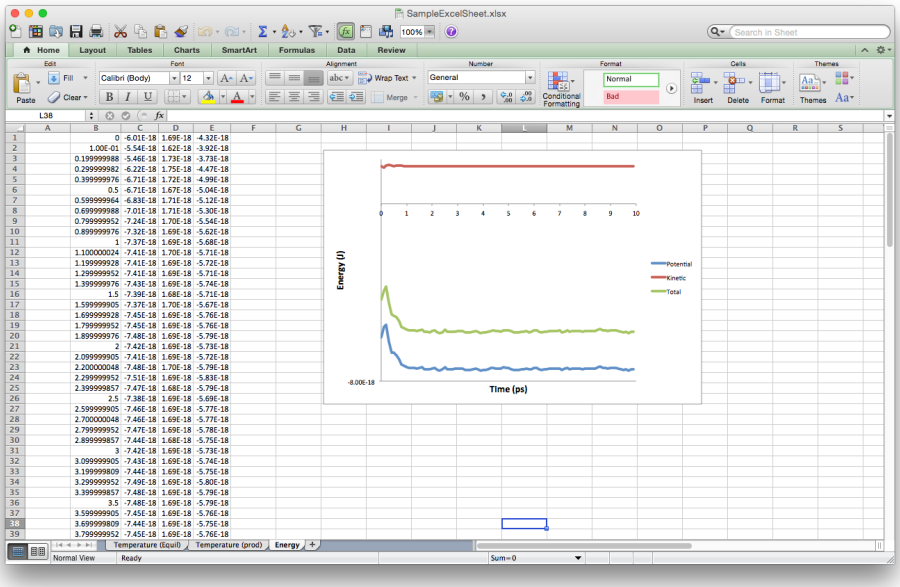

Figure 5: Microsoft Excel with energy file from the equilibration simulation imported. A plot of the energy evolution with time is shown. As a thermostat was used to maintain the temperature, the kinetic energy is nearly constant, but total energy is not conserved.

energy plots from the equilibration and production runs. In the equilibration, the temperature is constant. Likewise, the kinetic energy is constant (because temperature and kinetic energy are related quantities), while total and potential energy fluctuate (Figure 5). Recall that in the equilibration, we purposely maintained the temperature at 94.4 K. As a result, the kinetic energy remains nearly constant, but total energy is not conserved. On the other hand, during the production run you will notice the temperature fluctuates around 94.4 K. In the energy plot kinetic and potential energy will fluctuate, but total energy is conserved.

Modifying the parameters

Due to the modular nature of the MD code, it is rather easy to modify specific simulation parameters and tinker with the potential function or simulate a different system. For example, if we alter the particle mass and Lennard-Jones potential parameters, we would expect different properties in the resulting trajectories. Students can try to alter the parameters themselves and form hypotheses on the effects of these alterations, or they can use Lennard-Jones parameters for similar atoms such as neon and attempt to simulate a system of neon atoms by editing the appropriate parameters. Table 1 shows some Lennard-

Jones parameters and the corresponding modifications that would need to be made in `ft_parameters.f` to simulate other group 18 atoms. Students can look up phase diagrams for these elements to determine appropriate simulation temperatures for liquid.

Table 1: Masses and Lennard-Jones parameters for some group 18 elements [2].

Element `aname`	Mass (kg) `mass`	σ (nm) `sigma`	ϵ (J) `epsilon`
Ne	$3.350\,91 \times 10^{-26}\,\mathrm{kg}$	$0.275\,\mathrm{nm}$	$4.915\,11 \times 10^{-22}\,\mathrm{J}$
Kr	$1.391\,53 \times 10^{-25}\,\mathrm{kg}$	$0.383\,\mathrm{nm}$	$2.264\,27 \times 10^{-21}\,\mathrm{J}$
Xe	$2.180\,22 \times 10^{-25}\,\mathrm{kg}$	$0.406\,\mathrm{nm}$	$5.605\,44 \times 10^{-21}\,\mathrm{J}$

References

[1] Rahman, A. *Phys. Rev.* **1964**, *136*, A405–A411.

[2] McQuarrie, D.A.; Simon, J.D. *Physical Chemistry: A Molecular Approach*, 1st ed.; University Science Books, 1997.