LAMMPS for Dummies - (how-to's)

F. Cornes*
Departamento de Física, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Pabellón I, Ciudad Universitaria, 1428 Buenos Aires, Argentina.

G.A. Frank†
Universidad Tecnológica Nacional, Facultad Regional Buenos Aires, Av. Medrano 951, 1179 Buenos Aires, Argentina.
(Dated: March 11, 2016)

This article is intended as a starting point in the art of simulation. We will try to make it easy...

PACS numbers: DuMMy, 1.2.me

I. HOW TO CREATE A COMPLEX REGION

Up to now we simulated particles inside in a square region. Let's see how we can make a region with an aperture, like a door. In LAMMPS the apertures are made superposing regions. What? (you probably says) Let's take a look to the following instructions:

```
region   zone1 block 0 30 0 30 -1 1 units box
region   zone2 block 30.12 60 0 30 -1 1 units box
region   zone3 block 29 31 14.4 15.6 -1 1 units box
region   zone4 union 3 zone1 zone2 zone3
```

The first three are regions with different sizes that overlap each other (this is important). The command `region` has the option (`union`) that joins regions. If you use this option you have to tell the amount of regions to be joined (3 in this case) and then enumerate each one by its name (label). In this case, the regions are `zone1`, `zone2` and `zone3`. That is all! A door is simply a narrow region (`zone3`) that overlaps with the other teo (`zone1` and `zone2`).

Now, we have to tell LAMMPS where the simulation will take place. Notice that this step is necesary since the above definitions do not mention the role that the regions will play during the process. Furthermore, we did not specify (untill now) where the atoms (or molecules or particles) will be placed. The following instructions do these jobs

```
create_box       1 zone4
create_atoms     1 region zone1
```

With the `create_box` instruction we are telling LAMMPS the region where we want to run the simulation (that is, inside `zone4`). With `create_atoms` we say where the atoms will be placed. The number `1` in both instructions correspond to the label of the `box` and the label of the `atoms`. The `box` and the `atoms` do not need to have the same label. You can even label them with more explicit names like `mybox` or `myatoms`, or whatever. Future references will use these labels.

II. HOW TO MAKE A MOVIE

What is a simulation without a video? ...nothing. To make a movie we have to install the package FFMPEG. It can be download by writting in the "terminal" (Ubuntu is quite simply)

```
sudo apt-get install ffmpeg
```

After that you need to make some aditional changes to LAMMPS in order to tell it that FFMPEG is available in the system. The changes should be done to the file `Makefile.mpi` located in the following directory

```
/home/you/Documents/lammps-10Aug15/src/MAKE
```

where `you` should be replaced with your user name in the system. `lammps-10Aug15` is the current version of LAMMPS, but you may have an other one. Please check that!

Open the file `Makefile.mpi` with your prefered text editor (typically `gedit`) and look for the line

```
LMP_INC = -DLAMMPS_GZIP -DLAMMPS_JPEG -DLAMMPS_PNG
```

Add `-DLAMMPS_FFMPEG` at the end of the lines (leave space between `PNG` and `-DLAMMPS`...).

Perhaps you are a cautious person and you want to keep things well done for the future (...just in case...). So, you can apply the same changes to the file `Makefile.serial`. This is not necesary, but you can leave it done for the future.

Finally, go back to the `src` directory and re-build LAMMPS. Type the following in the command-line

```
cd ..
make clean-all
make mpi
```

* fercornes@gmail.com
† guillermo.frank@gmail.com

(this will take a few minutes). Notice that the instruction `make clean-all` deletes any previous configuration, including the additional LAMMPS packages. Thus, you will need to install them again. For example, if you installed the `granular` package in the past, now you will need to type `make yes-granular`.

Let's return to the script. Making a movie is very similar to making an image. This will be done by writting the following instruction (in a single line)

```
dump myfirstmovie all movie 100 myfirstmovie.mp4
type type
```

This `dump` has the label `myfirstmovie`. The `all` option means that we want all the atoms (or molecules or particles) to be displayed. The image sequences will be updated every `100` timesteps and the movie will be saved as `myfirstmovie.mp4`. It is not necesary for the filename to be the same as the label.

Other available formats are .avi, .mpg, .m4v, .mp4, .mkv, .flv, .mov, .gif. Now is time to enjoy the movie!

III. HOW TO PRINT OUT THE ATOMS POSITIONS AND FORCES

LAMMPS has an option that allows us to know several properties of the molecules at each timestep. For example, you can try the following

```
dump myreport all custom 100 in.myreport id x y fx fy
```

This instruction makes a report of the `x` and `y` positions (for all the molecules), and the corresponding forces `fx` and `fy`. This `dump` is labelled as `myreport` and reports the requested magnitudes every `100` timesteps. We had to tell LAMMPS that the wanted magnitudes are those written at the end of the instruction, and that we also wanted them in that order. This was done by means of the `custom` option.

Let's take a look of the report file `in.myreport` at the zero timestep.

```
ITEM: ATOMS id       x       y            fx           fy
             1 1.50008 1.50008  -1.40417e-07 -1.40417e-07
             2 4.50023 1.50008             0 -1.40417e-07
            12 4.50023 4.50023             0            0
```

The first displayed magnitude is the atom `id`, that is, the label of each atom. LAMMPS usually prints the `id`'s in consecutive order. However, you can change that by assigning different `id`'s to the molecules throughout the process.

Notice that the first molecule received a force `fx=-1.40417e-07` in the x-direction and `fy-1.40417e-07` in the y-direction. The second molecule only received a force in the y-direction. The 12 molecule did not interact at all.

The complete report prints the positions and forces at each timestep. This is what we call the configuration information. Although this king of reports eats the disk space, it is necesary for making movies with other programs such as `VMD`.

Some of the possible magnitudes that can be reported are `x y z` (positions), `vx vy vz` (velocities), `fx fy fz` (forces) and `px py pz` (pressures). As mentioned above, the `id`'s can be changed by computing other ones such as the cluster id, or user defined id's.

IV. HOW TO MAKE ARITHMETIC OPERATIONS

LAMMPS is straight foward for running a bare simulation. You only need a small script! But sometimes you will like to make some kind of arithmetic operations during the simulation process. There are two main reasons for this: you might not want to write a program from scratch to get an indirect magnitude, or, you can not afford storing big data in your hard disk. If you feel that you fit into these categories, continue reading.

We are going to explain to how obtain arithmetic results from a simulation done with LAMMPS. You first need to differentiate between a `variable` and a `compute`. For example, during the simulation process LAMMPS computes positions, velocities, etc. for each atom. These are atom properties since each atom has its own value for the position, velocity, etc. On the contrary, a `variable` stores values (scalar or vector) and are not necesary related (univocally) to atom values.

The starting point for retrieving data is telling LAMMPS which magnitudes are you interested in. For example, the following instruction tells LAMMPS that we are interested in the `x` position for each atom

```
compute  1 all property/atom x
```

We have asked LAMMPS to `compute` the x-position `property` for `all` the atoms. We tagged these computation with the number `1`. From now on the computation of the x-positions will be mentioned for short as `c_1`.

Suppose, for example, that we want to know how many atoms are located at positions $x > 10$. Then, we should ask whether `c_1>10.0`. If this is true the answer will be `1`, while if not the answer will be `0`. We want to keep this answers, and thus, we need to define a variable for storing this information. We can do this as follows

```
variable b atom c_1>10.0
```

The above instruction creates the variable named `b` to store the "per atom" results for the comparison `c_1>10.0`. `b` can be visualized as the vector having `1` or `0` whether $x > 10$.

If we sum up all the 1's in `b` we will know how many atoms belong to the region $x > 10$. The summation corresponds to a `compute` operation as follows

```
compute  mycompute all reduce sum v_b
```

Once again, the computation labeled `mycompute` sums the values in `b`. Notice that we have written `v_b` instead of `b`. This is necesary to inform LAMMPS that the values are stored in a previously defined variable.

It looks strange that we summation is invoked es `reduce sum`. We can explain that by saying that the summation "reduces" many values to a single one. However, this explanation is not satisfactory because it is ambiguous and obscure. The "true" reasons for writing `reduce sum` are programing reasons (out of the scope of this text).

Finally, we can store the computation `mycompute` (now `c_mycompute`) in the variable `s`. Thus, the hole sequence of instructions is

```
compute  1 all property/atom x
variable b atom c_1>10.0
compute  mycompute all reduce sum v_b
variable s equal c_mycompute
```

V. HOW TO MAKE LOOPS

A loop is no more than a comparison and a jump. That is, the comparison checks for the condition to be true. If this happens, the consequently action is a jump to a specific line in the LAMMPS script.

To achieve a jump we need to tag the place where we want to jump. For example, suppose that the following instructions appear at some place in the script

```
label myfirstloop

...
...

jump SELF myfirstloop
```

(the `...` lines indicate any instruction)

The `label` instruction indicates a specific place in the script. The `jump SELF` is the way we say "at this point make a jump back to the place where `myfirstloop` appears". This is an unconditional jump since no condition is required for the jump. This kind of loops are never ending loops, and thus, not practical. But its a good example to begin with.

Suppose we want to introduce a counter inside the loop. That is, we would like to count how many jumps back we have done. In order to achieve this, we first need to define a new variable, say `i` and increase its value each time we jump back. The following code illustrates how can this be done

```
variable i loop 100
label myfirstloop

...
...
next i
jump SELF myfirstloop
```

The first line defines the variable `i`, but IT DOES NOT make `i equal 100`. Instead, it says `i loop 100`, meaning that `i` will store the set 1, 2,..., 100. At first `i` is set to `1`.

The loop starts at the second line and it ends with the `jump` instruction. Immediately before the `jump` instruction we included the instruction `next i`. This instruction increases the `i` variable by one unit, and this operation is always done before jumping back to the `label` line. However, if `i` is increased to the "out of range" value 101, a break in the loop occurs. The `jump` instruction is omitted, and the script continues executing the instruction out of the loop. Notice that the variable is able to take the values 1...100, so the instructions inside the loop are swept a hundred times.

It is a good programing practice to delete used variables, although this is not mandatory. The loop variable `i` can be deleted after the loop. The way to do that is simply typing

```
variable i delete
```

Loops can be terminated under special circumstances. Suppose, for example, that there is a variable tagged `j` inside the loop, and something is going wrong if `j` exceeds `0.5`. We would like to finish the job if $j > 0.5$. Thus, we need to compare the value stored in `j` at each loop cycle, and break the loop if `0.5` is exceeded. We can do this as follows

```
variable i loop 100
label myfirstloop

...
...
if "$j > 0.5" then "jump SELF myescape"
...
...
next i
jump SELF myfirstloop

label myescape
```

The comparison between the value stored in `j` and `0.5` is done by the expression `"$j > 0.5"`. The `$j` means "the value stored in j", not just the "tag j". The `if` instrucction evaluated the expression and executes a `jump` if the expression becomes true. Otherwise, the `jump SELF myescape` will not be used. In this case, we have done a `jump` forward to the `label myescape`.

Be careful on the way you indicate a variable! We mentioned above that `j` is the name of the variable, but `$j` means the value stored in the variable. Also, be aware that the names that include more than one character need to be enclosed between braces. For example, the variable value stored in `myvar` should be invoked as `${myvar}`. You can also invoke `${j}` if you like.