1. Assuming that the total energy before the first iteration is exact, present an analysis of numerical errors due to the following factors:
   1. increasing number of atoms,
   2. increasing number of iterations, and
   3. increasing timestep size.

   For each of these factors, identify and discuss the most likely source of numerical errors. ***(4 marks)***

./md_AoS	N	R	dt	iter	initial total energy	final total energy
origin1	3	1	0.001	10	-7.577011e+01	-7.577011e+01
test1	3	1	0.001	200	7.577011e+01	-7.577016e+01
test2	3	1	0.01	200	-7.577011e+01	-7.577293e+01
test3	20	1	0.01	200	-3.886382e+04	-7.577293e+01

For test1, I increased the iteration into 200. The initial total energy is -7.577011e+01. However, in the end of the iteration, the total energy is -7.577016e+01.

For test2, I increased the timestep size from 0.001 to 0.01. The initial total energy is -7.577011e+01. However, in the end of the iteration, the total energy is -7.577293e+01.

For test2, I increased the number of atoms per side from 3 to 20. The initial total energy is -3.886382e+04. However, in the end of the iteration, the total energy is -7.577293e+01.

For test1, test2 and test3 they both have round-off errors. In my code, lots of sqrt() and pow() functions will produce results that exceed the precision of the double type, the system will automatically round (or chop) those digits into a certain number of digits. Thats why when increasing the number of iterations and number of atoms will cause the difference between the initial total energy and the final total energy.

2. What is the overall execution time of your code as a function of different problem dimensions? Provide a detailed analysis of how your code scales with respect to the input parameters *N*, *R*, *dt*, and *iter* (e.g. is it *O(NlogN)*, *O(N)*?) ***(4 marks)***

./md_SoA	N	R	dt	iter	microseconds
origin1	3	1	0.001	10	6321
test1	4	1	0.001	10	24814
test2	5	1	0.001	10	54175
test3	6	1	0.001	10	135851
test4	7	1	0.001	10	341106
test5	8	1	0.001	10	753550
test6	9	1	0.001	10	1523813
test7	10	1	0.001	10	2878123

According to my code, there are two loops with n, n = N^3; In this case, the code scale is O(N^6).

./md_SoA	N	R	dt	iter	microseconds
origin1	3	1	0.001	10	6321
test1	3	10	0.001	10	6222
test2	3	20	0.001	10	6192
test3	3	40	0.001	10	6186
test4	3	80	0.001	10	6228
test5	3	100	0.001	10	6535
test6	3	2000	0.001	10	6218
test7	3	4000000	0.001	10	6277

According to the table, no matter how big the R is, the process results won't change a lot. In this case, the code scale is O(1).

./md_SoA	N	R	dt	iter	microseconds
origin1	3	1	0.1	10	6480
test1	3	1	0.01	10	5011
test2	3	1	0.001	10	6208
test3	3	1	0.0001	10	6147
test4	3	1	0.00001	10	6146
test5	3	1	0.000001	10	6143
test6	3	1	0.0000001	10	6199
test7	3	1	0.00000001	10	5902

According to the table, no matter how small the dt is, the process results won't change a lot. In this case, the code scale is O(1).

./md_SoA	N	R	dt	iter	microseconds
origin1	3	1	0.001	10	6280
test1	3	1	0.001	100	38873
test2	3	1	0.001	1000	259097
test3	3	1	0.001	10000	5820878

According to the code, iter only use once in a for loop. In this case, the code scale is O(N).

3. During each iteration, what proportion of the total execution time is spent calculating the new i) positions, ii) forces, and iii) velocities? How does this proportion change as problem dimension increases and how did you determine your answer? ***(3 marks)***

   The proportion of the total execution time for calculating the new position, forces and velocities: 1:2:1. We could look at the code to figure out the reason: for position and velocities only one for loop, and for the forces calculation has two for loops. When the problem dimension increasing, for example, increasing the iteration, N, R, or dt, the proportion is still the same because the main function will run N times update_timestep() functions, within this function the proportion is determined.

4. Does the most computationally intensive component correspond to the lines of code that is executed most often? How did you determine this? ***(3 marks)***

   Yes, the most computationally intensive component is in the calculate_pe() function and update_force() function. These two lines of code are executed most often:

   ```
   15120*:    84:     for (int r2 = 0; r2 < atoms->n; r2++)
   &
   15120:    84:     for (int r2 = 0; r2 < atoms->n; r2++)
   ```

   The result is gained from md_lib_SoA.c.gcov

   ```
   gcc -fprofile-arcs -ftest-coverage md_lib_SoA.c -o md_lib_SoA_gcov -lm
   ./md_SoA_gcov 3 1 0.001 10
   gcov -b *.gcno
   ```

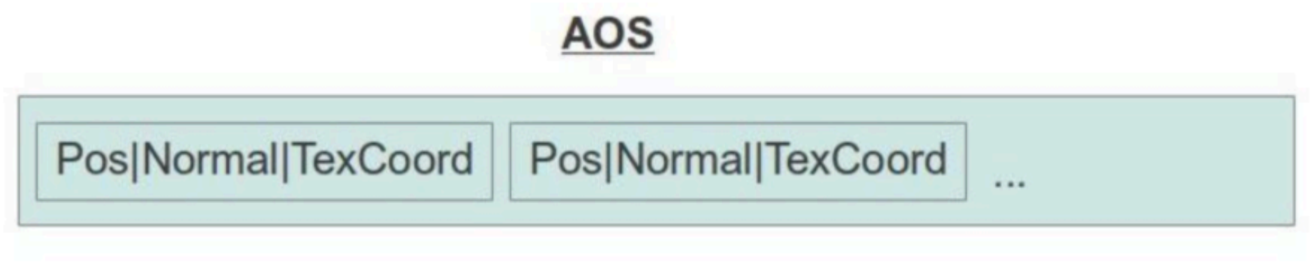
   From this file, we could clearly spot how many times that each line is called.

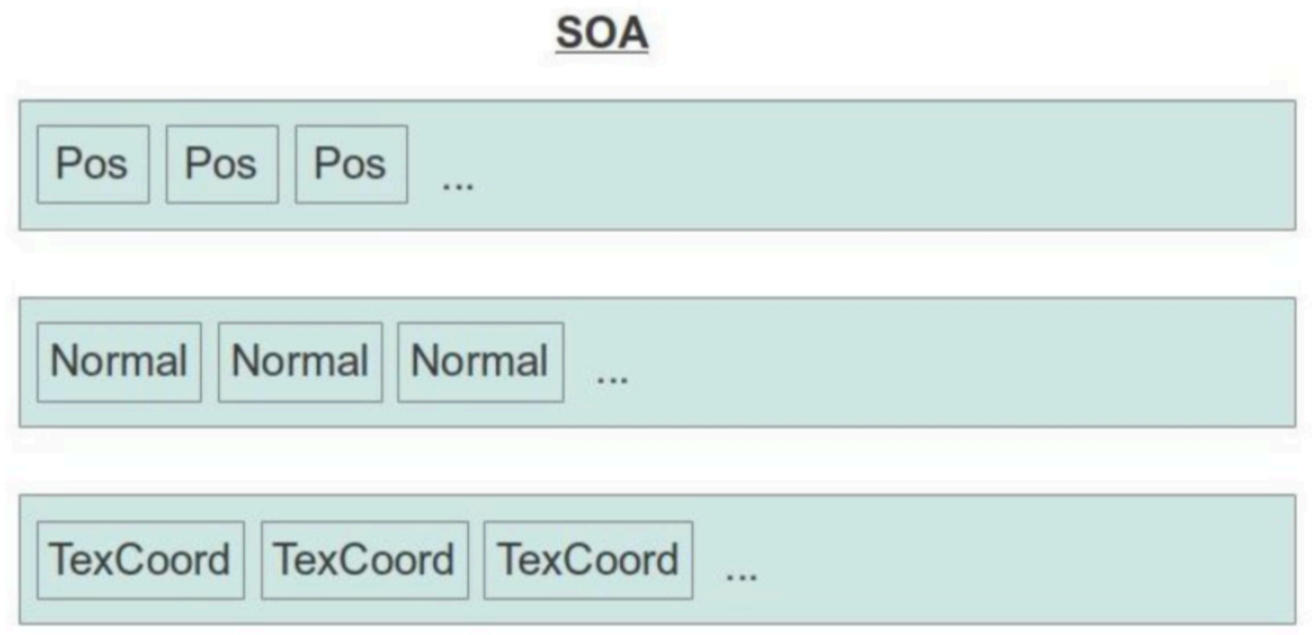
5. Compare your struct of arrays (SoA) code with your array of structs (AoS) code. Which do you expect to perform better? Which actually performs better or do they perform the same? Explain *why* you think this might be the case. What aspects of your CPU hardware would change which data structure layout is preferable? ***(4 marks)***

   My expectation is SoA better than AoS. Below is the table that I tested for SoA and AoS.

input (N R dt iter) \ struct	SoA (microseconds)	AoS (microseconds)
3 1 0.001 500	136937	133769
3 1 0.001 1000	272863	273964
10 1 0.001 20	6215541	6243024
10 1 0.001 30	9326587	9492179
10 1 0.001 40	12448155	12499327

   From the table, we can see that SoA is faster than the AoS. The reason is that SoA is stored all different atoms positions, forces, velocities and accelerations together. However, AoS will store each atom's position, force, velocity and acceleration as one array elements. Below picture can show the main difference between SoA and AoS:





In this case, when the loops are intensive access the consecutive memory addresses, its better for using SoA.

6. Show some data obtained using hardware performance counters comparing your SoA and AoS code. This may include metrics such as cycle counts, L1/L2 cache miss rates, instructions per cycle, branch mispredictions, or whatever appears worthwhile to you. Explain the significance of each metric that you have chosen, analyse how they vary with different problem dimensions, and discuss how they correlate with observed execution times (if applicable). Does this explain the performance difference (if any) between your two codes? [*You must do this on Gadi*] ***(5 marks)***

   Below table is showing the difference between SoA and AoS in respect to cycle counts.

   ```
   ./md_SoA 3 1 0.001 10
   ./md_AoS 3 1 0.001 10
   ```

Counters	SoA	AoS
Exec. time (us)	5359	4461
PAPI_L1_LDM (Level 1 load misses)	602	546
PAPI_L1_STM (Level 1 store misses)	93	47
PAPI_L1_TCM (Level 1 cache misses)	4420	4043
PAPI_L2_TCA (Level 2 total cache accesses)	8000	7323
PAPI_L2_TCM (Level 2 cache misses)	1199	1079
PAPI_L3_TCA (Level 3 total cache accesses)	1199	1079
PAPI_L3_TCM (Level 3 cache misses)	1039	920
L2 miss rate	14.99%	14.73%
L3 miss rate	86.66%	85.26%

For L1 cache, only some counters are avaliable. We can't know the total cache accesses for L1 cache. However, for L2 and L3 cache rates, AoS miss rates are lower than SoA.

```
./md_SoA 10 1 0.001 10
./md_AoS 10 1 0.001 10
```

Counters	SoA	AoS
Exec. time (us)	2849280	2828061
PAPI_L1_LDM (Level 1 load misses)	5323901	19254518
PAPI_L1_STM (Level 1 store misses)	24168	6841
PAPI_L1_TCM (Level 1 cache misses)	6146662	27168558
PAPI_L2_TCA (Level 2 total cache accesses)	5358917	19269829
PAPI_L2_TCM (Level 2 cache misses)	3479	4071
PAPI_L3_TCA (Level 3 total cache accesses)	3479	4071
PAPI_L3_TCM (Level 3 cache misses)	2033	2407
L2 miss rate	0.065%	0.021%
L3 miss rate	58.44%	59.13%

For this case, I increased the N from 3 to 10. The L2 miss rate SoA is bigger that the AoS. However, For L3 miss rate, AoS is bigger than the SoA one.

7. Choose either your SoA or AoS code and create a new program `md_Opt` by copying the files `md_XoX.c`->`md_Opt.c` and `md_lib_XoX.c`->`md_lib_Opt.c` respectively. Modify the `md_Opt.c`, `md_lib_Opt.c`, `Makefile`, and `md.py` files accordingly so that you can run the new code with either `./md_Opt N R dt iter` or `python3 md.py N R dt iter -c Opt`. Develop a modification to this code (e.g. change the access pattern, code structure or add an additional optimization) and compare the performance with the previous version using the tools you have been introduced to during the labs and lectures. ***(5 marks)***

   ```
   cat /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size
   ```

   We could get Gadi L1 cache line size: 64, which means that each cache line could hold 8 double numbers.

   In the structure of AoS, each atoms contains position (x, y, z), velocity (x, y, z), acceleration (x, y, z) and old_acceleration (x, y, z).

   `lscpu` we could get L1 data cache size is 32K (512 cache lines = 4096 double numbers). In order to maximum utilize cache, I choose to do the loop unrolling.

   I have applied 2 times loop unrolling into update_position, update_force and update_velocities functions. Compare the running time could see from the below table:

Inputs	SoA	AoS	Opt
3 1 0.001 10	112135	130762	96538
3 1 0.001 100	105920	147779	90957
3 1 0.001 1000	342391	347487	357443
3 1 0.001 10000	6521995	4652430	4690386
6 1 0.001 10	222477	230140	191166
10 1 0.001 10	3072078	3014552	2947948
15 1 0.001 10	33501625	32940440	32696456

   We can see that most of cases, Opt is fastest compared to SoA and AoS.

8. Suppose now you want to convert your optimized molecular dynamics (`md_Opt`) code into a *benchmark* similar to the LINPACK benchmarks. Research and briefly describe the implementation of the LINPACK benchmark, and how the TOP500 list is compiled using LINPACK. ***(1 mark)***

   LINPACK benchmarks will test a system's FLOPs (floating point computing power). It let the system to solve linear equation, for example: Ax = b. In this case, md_Opt need to have a set of inputs for different systems to accept test. For example, pre-define the N, R, dt and iter with different combinations, and record the peak performance.

9. Considering how LINPACK calculates the number of floating point operations (FLOP) as a model, devise a method for measuring the MFLOP/s in relation to your MD code. Specifically, outline how you will calculate the number of FLOPs for a given problem dimension. ***(2 marks)***

In my code, first thing I need to change is the type, I need to change the type from double into float. Then, I need to calculate how many float point operations in total (+, -, *, /). After all of that, I also need to record the time for executing the iterations. In the end, using the float point operations divide the time we will get the FLOPs.