Computer Science and Engineering Department
University Carlos III of Madrid

COMPUTER ARCHITECTURE
REDUCED GROUP 121

PROJECT 1 :

PERFORMANCE ORIENTED PROGRAMMING

24th October 2021

Team number 5

María Suevos Chinchilla - 100429902
Pablo González Vidal - 100429929
Belén Luque Calzado - 100432990
Moisés Jiménez Martínez - 100429835

Table of contents

Original Design

This project implements a gravitation simulation application for a set of objects. As stated in the project description, two versions of this simulation were made: sim_soa.cpp and sim_aos.cpp. The main difference between them is the storage of the variables: soa uses a structure of vectors of type double and aos a vector of structures (we use a vector instead of an array because we avoid having to predefine its size).

Firstly, we include some libraries such as: *iostream, fstream, random, cmath, vector, iomanip* and *cstdlib.* We must take into account that variables are mainly doubles and integers, that is the reason why using *iostream* and *iomanip* is vital. They allowed us to be able to use setprecison(), so we could ensure that we would get the three decimal precision we wanted. Then, *fstream* is used to create the files, for the arbitrary seed required we included *random,* and *cmath* in order to perform mathematical operations. Also, *cstdlib* was required to enable us using the *atoi* function, so we could transform integer parameters into strings, and the *stod* function, to convert string parameters into double ones.

Right after these libraries, we found the main difference between sim_soa.cpp and sim_aos.cpp. In soa, we declare a structure of vectors of type double called *object* and in aos we advocate a vector of structures which has the same name. Inside both of them we define the parameters required: velocity and position decomposed in the 3D axis and mass.

Afterward, we define the main function where both programs are reasonably similar, subsequently we will explain them at the same time. Right after declaring the universal gravitational constant we continue by checking the arguments introduced by the user. In the first place, we confirm that the number of parameters is six, otherwise it will be notified that there has been an error and, as requested, the program will end with error code -1. Once we convert the arguments into doubles, we check they are the type of data we want and we display each of them: *num_objects, num_iterations, random_seed, size_enclosure* and *time_step*. In case there is at least one parameter missing, we created a for loop which displays the parameter that has not been introduced. Then, we ensure that the introduced parameters are non-negative and in case any of them is negative or they are not the corresponding data type we need the program will end with error code -2.

Once we certified that all the parameters are the correct ones, the initial configuration file is created, *init_conf.txt*, where we write the size of the enclosure, the time step and the number of objects specified by the user. To achieve this we use the function setprecision() with 3 as argument, to obtain three decimals.

As it is expected, we now face a difference between the soa code and the aos one. In sim_soa.cpp a structure *objects* of vectors is created and in sim_aos.cpp a vector of structure of objects is created. After the uniform distribution function is displayed in both programs, we proceed by storing the initial values: the three coordinates for the position of the object and its velocity and the mass of the object. These values are also written in the file.

We continue by initializing to 0 the forces, which are decomposed into the three axes and we define the double variables we are going to use in the procedure. As requested, we firstly check for collisions before starting the simulation. To achieve this, we compute the relative position between objects by calculating the norm between two objects. If the norm is less than one, a collision occurs so the velocities of the objects in the three axes are added together as well as their masses and the position of the first object is keeped. Then the new object created with the sum of the masses and velocities is stored where the first object that collided was and the second object is deleted. In order to delete this object, we ensure its mass is equal to zero and then using the function *.erase()* we erase the objects with mass equal to zero and decrease the total number of objects in the system by one. We follow by computing the forces in the three coordinates for each object by applying the given formula. Afterwards we take advantage of the fact that F_{ij}=-F_{ji}; that is why we update the vector of forces in the code by multiplying by -1 the u coordinate of the force in every axis. Then, we proceed to compute the velocities corresponding to the time step and forces and consequently we update the position, which is the velocity times the time step and finally the forces are redefined to 0.

We pass on the rebound effect where we make sure that if the position is greater, in any direction, than the dimension of the enclosure we must establish the position as the cubes' dimension and set the velocities in the opposite direction. We achieve this by using a series of if statements that obey the conditions specified before. Finally we face two nested for loops which check again if there is any collision between the objects of the enclosure, following the same procedure as the one described above.

As the final step, we open a *final_conf.txt* file in which we write the final values of the parameters introduced.

Optimizations

The optimization process of this code was quite challenging. We started with high execution time and reducing them was an urgent need. We firstly thought about the techniques we previously studied in class: array merge and loop merge.

Once we implemented both optimizations, we realised that we were not taking advantage of the fact that F_{ij}=-F_{ji} , so we started to think of ways on how to add that to our code. We ended up achieving this by storing in our variables Fx, Fy, Fz the forces, which follow this computation: (G* mass of object 1* mass of object 2)*(position object 1 - position object 2) /norm, in each of the axis. Then we established that the force in X axis for object 1 was going to be Fx but the same force for object 2 was going to be -1*Fx. Same process for axes Y and Z. Thanks to this change our execution time was reduced from one minute to approximately 20 seconds. It made a huge impact.

All the functions were also develop in the same cpp file instead of creating them in a separate file and calling them, which would have increased the execution time.

We also realised that the order of the operations matters, as the associative property is not followed by floating points operations, so we started playing with the order of operations to try to spot the fastest way of calculating them.

Using the given optimization flags -O3 and -DNDEBUG made a difference too, as well as computing x*x*x instead of using the function pow(), which we did not expect at all to happen.

Performance evaluation

We evaluated the performance with the same computer every time. This computer has the following specifications:

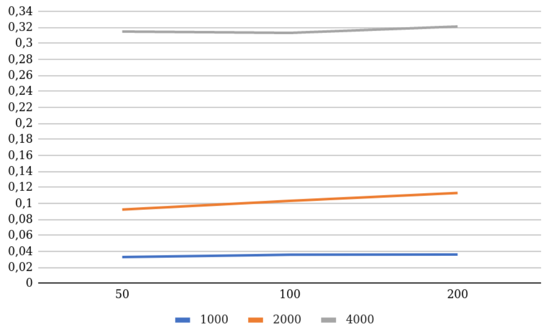
- Processor model : Intel® Core™ i7-10510U CPU @ 1.80GHz × 8
- Number of cores: core i7, 4 cores
- Main memory size: 15,5 GiB,
- Disk capacity : 512,1 GB
- Cache memory hierarchy: L1 cache: 256 KiB, L2 cache: 1 MiB, L3 cache: 8MiB.
- System software, operating system version: Ubuntu 20.04.3 LTS, 64-bit, GNOME version 3.36.8.
- Compiler version: 9.3.0

For this part, we were asked to run the experiment at least 10 or more times and obtain the average value for different numbers of objects and iterations. These are our results for the total execution time:

3.1.1. SOA

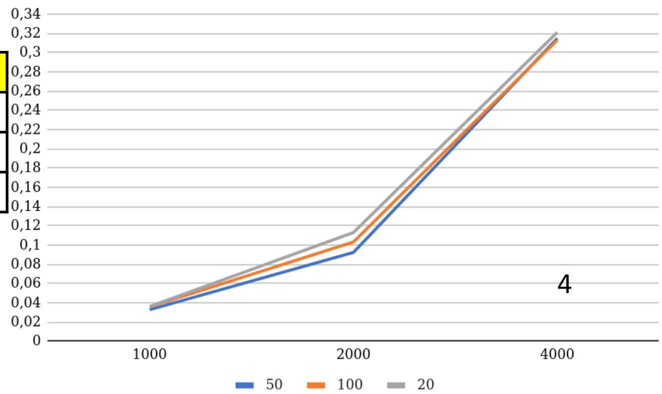
Using a random seed equal to 3, an enclosure of size 2000 and a time step of 0.5. The rows of the chart are the number of iterations and the columns the number of objects, and we have obtained the total execution time.

	1000	2000	4000
50	0,032980095	0,092535336	0,314990821
100	0,035958036	0,103378866	0,313327143
200	0,036209941	0,113267617	0,321287125



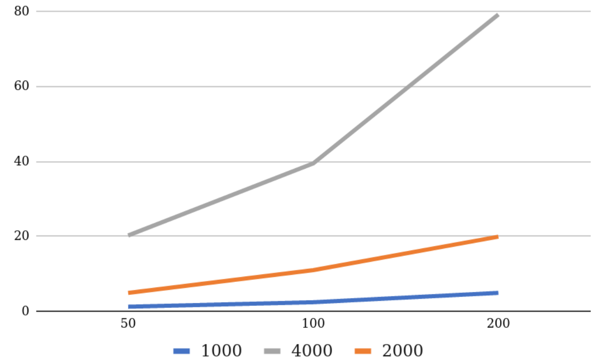
We use the same configuration as before but interchanging the rows and columns so changes can be seen better.

	50	100	200
1000	0,032980095	0,035958036	0,036209941
2000	0,092535336	0,103378866	0,113267617
4000	0,314990821	0,313327143	0,321287125



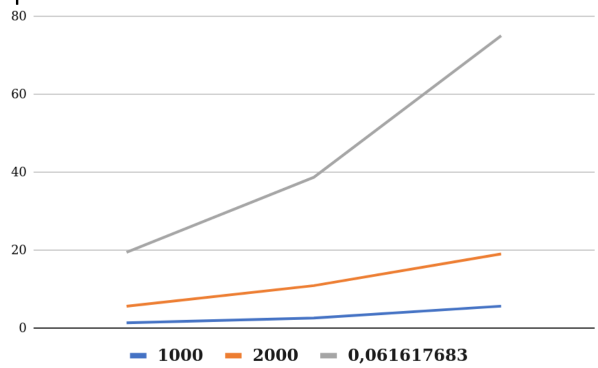
Using a random seed equal to 3, an enclosure of size 80000 and a time step of 0.5. Same configuration of rows and columns as in the first graph.

	1000	2000	4000
50	1,236934716	4,91999312	20,25330638
100	2,430043458	10,99359946	39,53185036
200	4,927210509	19,94457949	79,29845205



Using a random seed equal to 3, an enclosure of size 80000 and a time step of 2. Same configuration of rows and columns as in the first graph.

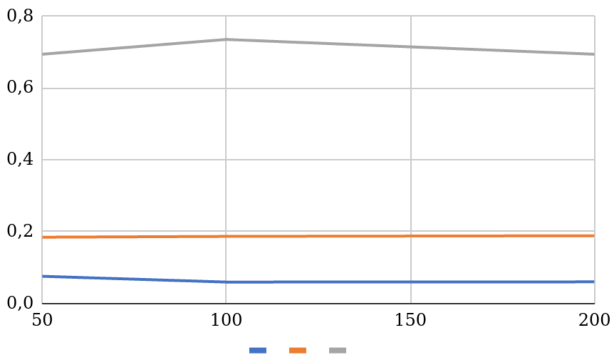
	1000	20000	40000
50	1,395242969	5,655975437	19,50468264
100	2,613444892	10,94121344	38,77157738
200	5,67211637	19,101754	75,15324012



3.1.2. AOS

Using a random seed equal to 3, an enclosure of size 2000 and a time step of 0.5. Same configuration of rows and columns as before.

	1000	2000	4000
50	0,07497224	0,183894295	0,693692854
100	0,058904645	0,186145582	0,73486866
200	0,059514124	0,188002185	0,693650013

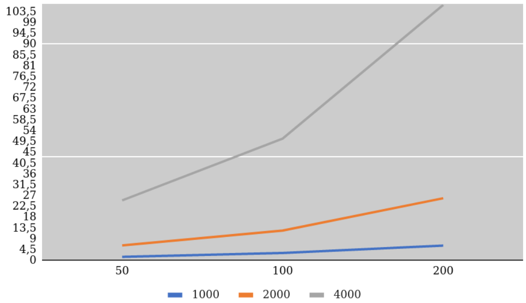


We use the same configuration as before but interchanging the rows and columns so changes can be seen better.

	50	100	200
1000	0,07497224	0,058904645	0,059514124
2000	0,183894295	0,186145582	0,188002185
4000	0,693692854	0,73486866	0,693650013

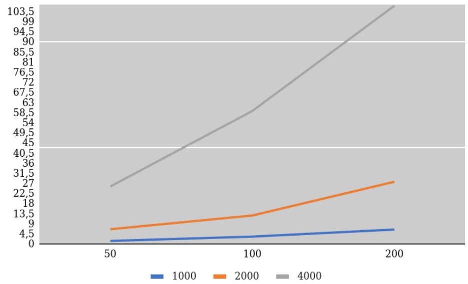
Using a random seed equal to 3, an enclosure of size 80000 and a time step of 0.5. Same configuration of rows and columns as in the first graph.

	1000	2000	4000
50	1,572695368	6,354225338	25,06622826
100	3,186690288	12,49837648	50,78906373
200	6,261955101	25,96275571	106,3732805



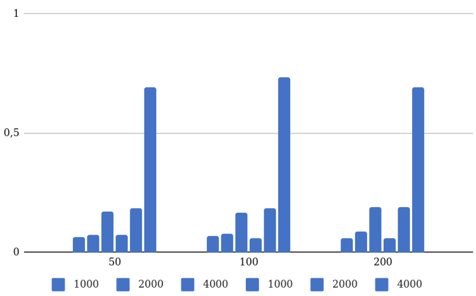
Using a random seed equal to 3, an enclosure of size 80000 and a time step of 2. Same configuration of rows and columns as in the first graph.

	1000	2000	4000
50	1,583890989	6,77359602	25,80713814
100	3,499169354	12,89922526	59,41494702
200	6,620939816	27,92866582	106,280375

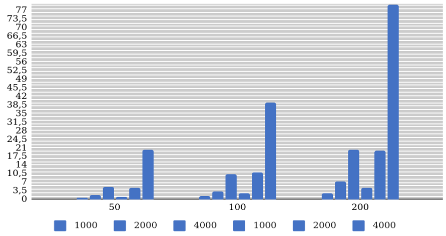


Then , we proceeded to compare our results with the one that v8-log provided. We organize the obtained data in graphs where the three leftmost columns inside a number of iterations (50,100,200) are the v8-log results and the three leftmost columns are ours.

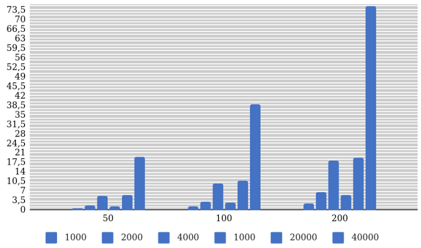
The following three graphs are the ones corresponding to SOA



Here we represent the results from using a random seed equal to 3, an enclosure of size 2000 and a time step of 0.5. We did this for 50, 100 and 200 iterations and also with 1000,2000,4000 objects.

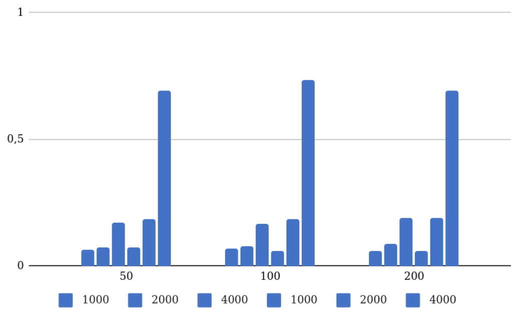


Here we represent the results from using a random seed equal to 3, an enclosure of size 80000 and a time step of 0.5.

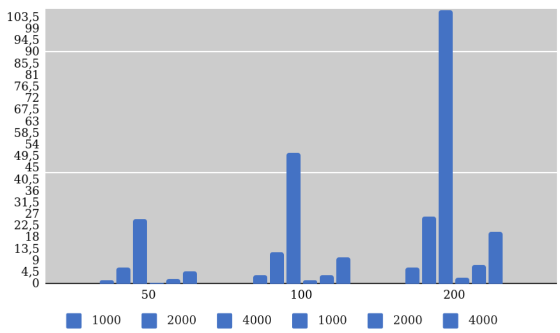


Here we represent the results from using a random seed equal to 3, an enclosure of size 80000 and a time step of 2.

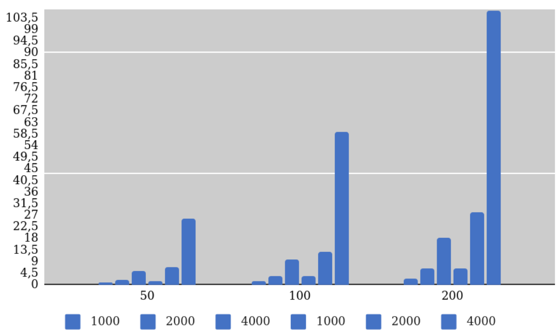
The following three graphs are the ones corresponding to AOS



Here we represent the results from using a random seed equal to 3, an enclosure of size 2000 and a time step of 0.5. We did this for 50, 100 and 200 iterations and also with 1000,2000,4000 objects.



Here we represent the results from using a random seed equal to 3, an enclosure of size 80000 and a time step of 0.5.

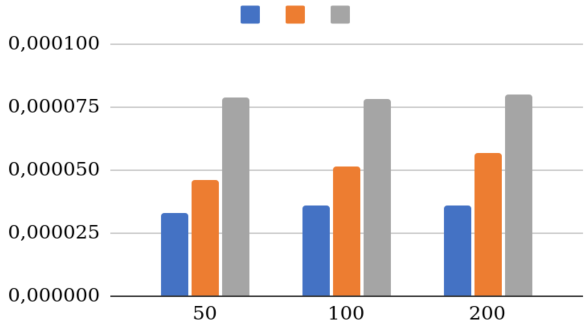


Here we represent the results from using a random seed equal to 3, an enclosure of size 80000 and a time step of 2.

Now, we will display the results of the average time per iteration.

3.2.1. SOA

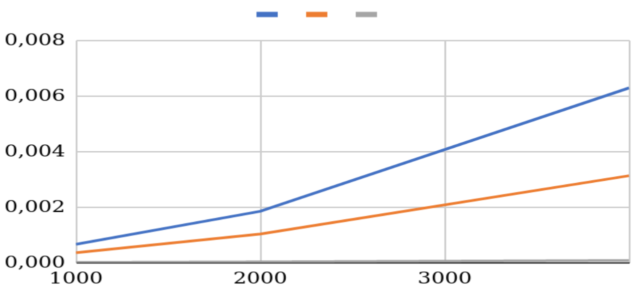
Using a random seed equal to 3, an enclosure of size 2000 and a time step of 0.5. The rows of the chart are the number of iterations and the columns the number of objects, and we have obtained the total execution time. In the following graphs, blue is 1000 objects, orange is 2000 and grey is 4000.

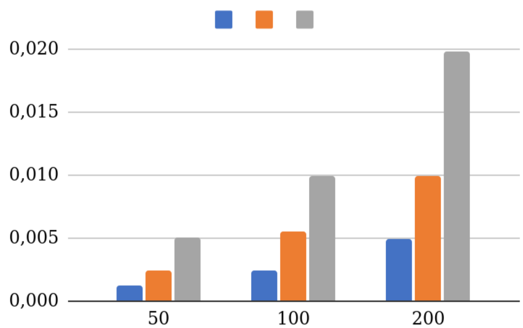


	1000	2000	4000
50	0,000032980095	0,000046267668	0,00007874770525
100	0,000035958036	0,000051689433	0,00007833178575
200	0,000036209941	0,0000566338085	0,00008032178125

Same parameters but interchanging rows and cols:

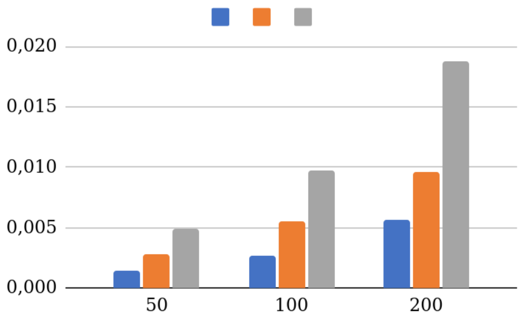
	50	100	200
1000	0,0006596019	0,00035958036	0,00000905248525
2000	0,00185070672	0,00103378866	0,00002831690425
4000	0,00629981642	0,00313327143	0,00008032178125





Using a random seed equal to 3, an enclosure of size 80000 and a time step of 0.5. Same configuration of rows and columns as in the first graph.

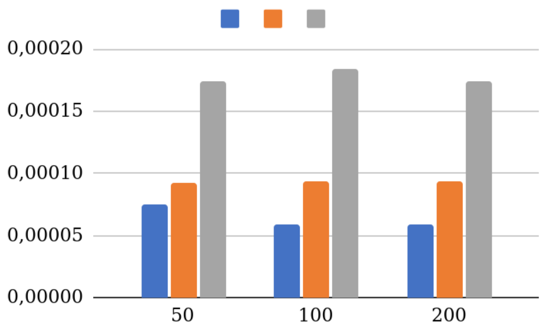
	1000	2000	4000
50	0,001236934716	0,00245999656	0,005063326595
100	0,002430043458	0,005496799729	0,009882962589
200	0,004927210509	0,009972289743	0,01982461301



Using a random seed equal to 3, an enclosure of size 80000 and a time step of 2. Same configuration of rows and columns as in the first graph.

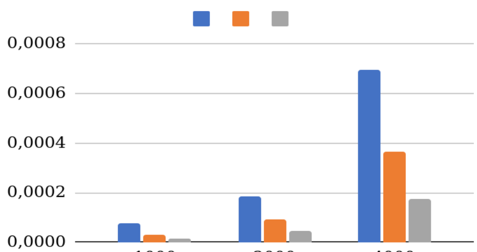
	1000	20000	40000
50	0,001395242969	0,002827987719	0,00487617066
100	0,002613444892	0,005470606721	0,009692894344
200	0,00567211637	0,009550877002	0,01878831003

3.2.2. AOS



Using a random seed equal to 3, an enclosure of size 2000 and a time step of 0.5. The rows of the chart are the number of iterations and the columns the number of objects, and we have obtained the total execution time. In the following graphs, blue is 1000 objects, orange is 2000 and grey is 4000.

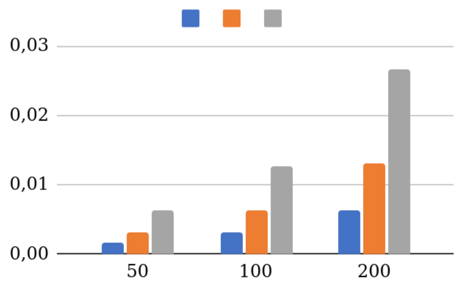
	1000	2000	4000
50	0,00007497224	0,0000919471475	0,0001734232135
100	0,000058904645	0,000093072791	0,000183717165
200	0,000059514124	0,0000940010925	0,0001734125033



Same parameters but interchanging rows and cols:

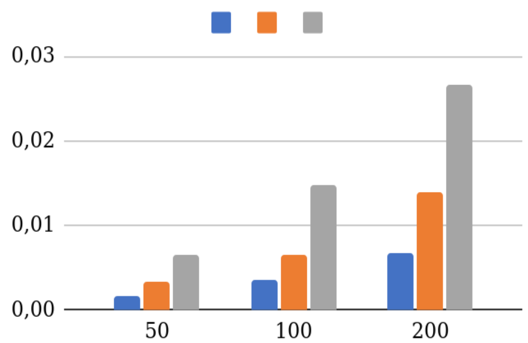
	50	100	200
1000	0,00007497224	0,0000294523225	0,000014878531
2000	0,000183894295	0,000093072791	0,00004700054625
4000	0,000693692854	0,00036743433	0,0001734125033

Using a random seed equal to 3, an enclosure of size 80000 and a time step of 0.5. Same configuration of rows and columns as in the first graph.



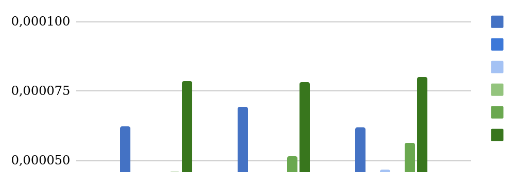
	1000	2000	4000
50	0,001572695368	0,003177112669	0,006266557065
100	0,003186690288	0,006249188239	0,01269726593
200	0,006261955101	0,01298137786	0,02659332014

Using a random seed equal to 3, an enclosure of size 80000 and a time step of 2. Same configuration of rows and columns as in the first graph.



	1000	2000	4000
50	0,001583890989	0,00338679801	0,006451784535
100	0,003499169354	0,006449612628	0,01485373676
200	0,006620939816	0,01396433291	0,02657009375

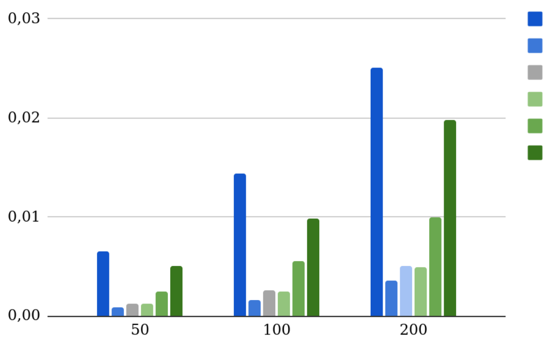
Then , we proceeded to compare our results with the one that v8-log provided and represent them in these graphs. Blue parts are v8-log results and green are ours.



The following three graphs are the ones corresponding to SOA:

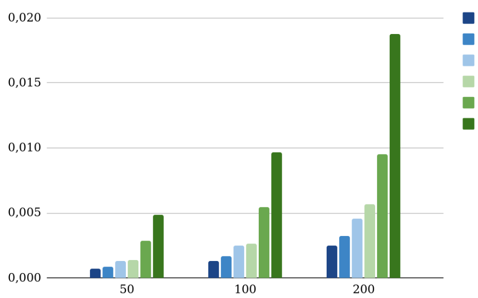
Here we represent the results from using a random seed equal to 3, an enclosure of size 2000 and a time step of 0.5. We did this for 50, 100 and 200 iterations and also with 1000,2000,4000 objects.

	1000	2000	4000	1000	2000	4000
50	0,000062449684	0,000037141344	0,00004258953775	0,000032980095	0,000046267668	0,00007874770525
100	0,000069324651	0,000038798649	0,00004201228325	0,000035958036	0,000051689433	0,00007833178575
200	0,000062078604	0,000045016701	0,000047017643	0,000036209941	0,0000566338085	0,00008032178125



Here we represent the results from using a random seed equal to 3, an enclosure of size 80000 and a time step of 0.5.

	1000	2000	4000	1000	2000	4000
50	0,00650963371	0,000896403129	0,001266817538	0,001236934716	0,00245999656	0,005063326595
100	0,01433646566	0,001670605409	0,002546019337	0,002430043458	0,005496799729	0,009882962589
200	0,02507884118	0,003634786025	0,005094150619	0,004927210509	0,009972289743	0,01982461301

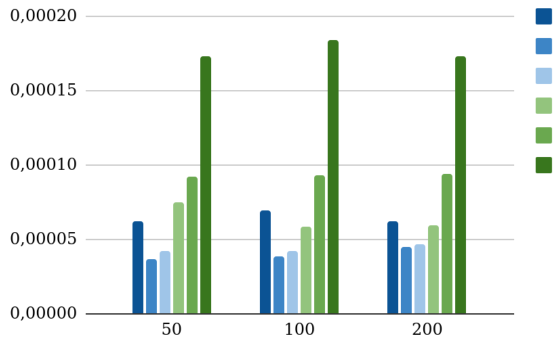


Here we represent the results from using a random seed equal to 3, an enclosure of size 80000 and a time step of 2.

	1000	2000	4000	1000	20000	40000
50	0,00070230749	0,0008763927655	0,001296325264	0,001395242969	0,002827987719	0,00487617066

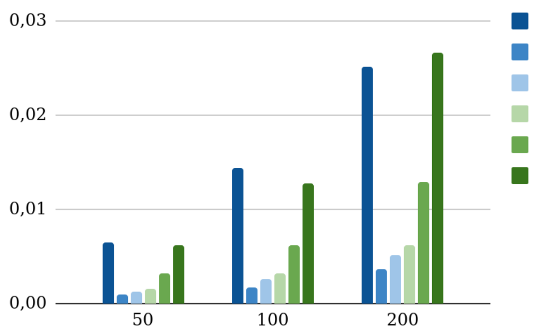
100	0,001298635196	0,001643204067	0,002456812778	0,002613444892	0,005470606721	0,009692894344
200	0,002496768736	0,0032454 7745	0,004580544526	0,00567211637	0,009550877002	0,01878831003

The following three graphs are the ones corresponding to AOS:



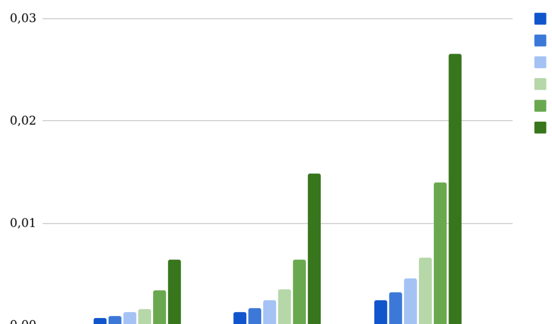
Here we represent the results from using a random seed equal to 3, an enclosure of size 2000 and a time step of 0.5. We did this for 50, 100 and 200 iterations and also with 1000,2000,4000 objects.

	1000	2000	4000	1000	2000	4000
50	0,000062449684	0,000037141344	0,00004258953775	0,00007497224	0,0000919471475	0,0001734232135
100	0,000069324651	0,000038798649	0,00004201228325	0,000058904645	0,000093072791	0,000183717165
200	0,000062078604	0,000045016701	0,000047017643	0,00005951124	0,0000940010925	0,0001734125033



Here we represent the results from using a random seed equal to 3, an enclosure of size 80000 and a time step of 0.5.

	1000	2000	4000	1000	2000	4000
50	0,00650963371	0,000896403129	0,001266817538	0,001572695368	0,003177112669	0,006266557065
100	0,01433646566	0,001670605409	0,002546019337	0,003186690288	0,006249188239	0,01269726593
200	0,02507884118	0,003634786025	0,005094150619	0,006261955101	0,01298137786	0,02659332014



Here we represent the results from using a random seed equal to 3, an enclosure of size 80000 and a time step of 2.

	1000	2000	4000	1000	2000	4000
50	0,00070230749	0,0008763927655	0,001296325264	0,001583890989	0,00338679801	0,006451784535
100	0,001298635196	0,001643204067	0,002456812778	0,003499169354	0,006449612628	0,01485373676
200	0,002496768736	0,00324547745	0,004580544526	0,006620939816	0,01396433291	0,02657009375

Analyzing the results, we can observe the simulator runs faster by implementing a structure of arrays (soa) than implementing an arrays of structures (aos). This can be observed mainly in the first tables of both soa and aos where executing the soa program with 4000 iterations, 200 objects, 3 seed, 2000 size enclosure and 0.5 time step it takes 0,00008032178125 seconds while running aos takes 0,0001734125033 which is 21 times faster.

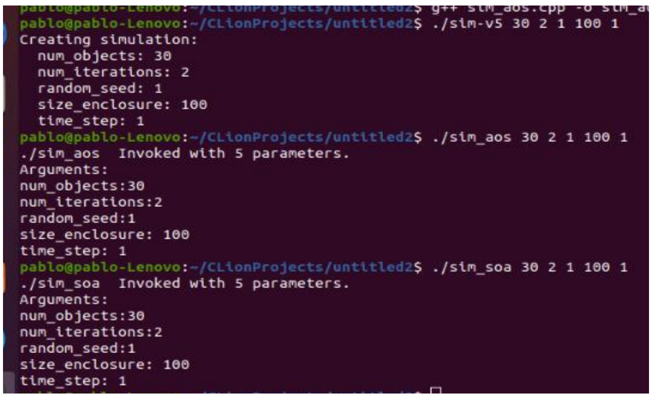
Also, we can deduce that as we increase the number of iterations the execution time of the program increases faster than when we increase the number of objects, which can be observed in the first two tables and graphs of both soa and aos.

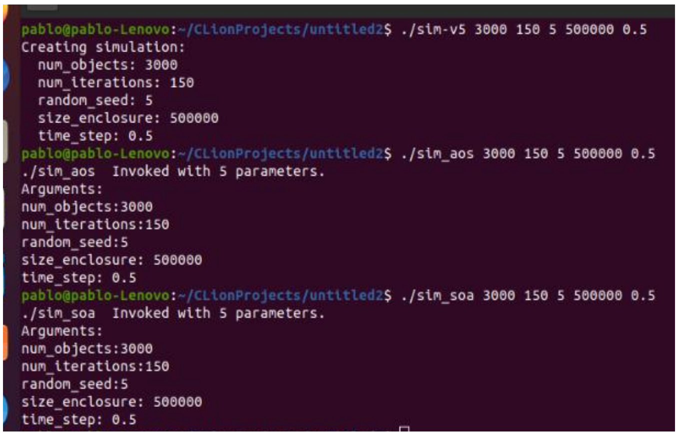
Below those graphs, we can see the change in the time execution and average time in each iteration when changing the size enclosure, so that objects don't collide so easily, and the time step.

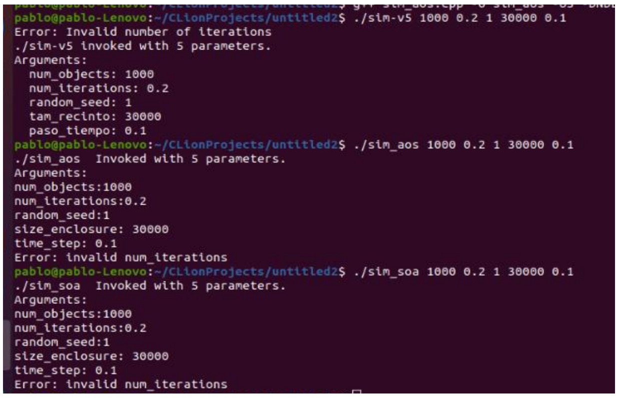
Finally, the comparison between the execution time and the average execution time per iteration of soa and aos with the sim-v8log executable denotes that soa and aos are slightly faster when running them with 3 seed, 2000 size enclosure and 0.5 time step while executing it with different number of iterations and objects but they seem slower when we increase the size enclosure and the time step.

4. Tests

Here we do some test in order to check the correct functionality of our program. We introduce an invalid number of arguments, negative arguments or floats numbers in data that must be integers in order to know if our program is able to detect the problem and stop. We also perform a test where the arguments are correct to show that it works.







5. Conclusions

Doing this project we have realized the importance of trying to minimize the number of operations, for example we decrease the time to almost to the half when we were able to reuse the negative forces.

Also we have tried to reduce the number of operations getting a common factor, reusing some results of previous iterations or storing results that are the same for every operation but each time that we do some of these modifications, we get a higher difference in the results from the teacher simulation. This is something that surprised us a lot, we couldn’t even imagine that changing the order or the form of the equations could change so much the result.

Due to the importance of the order of the operations we focus on trying to simulate exactly the same order of equations described in the pdf, but we weren’t be able to get the same output so because we have a huge number of objects and iterations we get bigger differences between our simulation and the reference simulation.

One of our main problems was trying to be as exact as possible with respect to the reference but we realize that in each single sum, division, multiplication... we were getting a little error and this error becomes bigger as more objects and iterations we have, in fact as more operations that we perform. We conclude that this was because of the policy of how c++ rounds off floats and how the computer stores them which explains why when we don’t have a big number of operations we get the same output but when we increase the number of iterations and objects we get a lot of mistakes.