Aladyn – A portage from the Fortran language to the Python language

Yann Abou Jaoude
ESILV, Courbevoie, France

Axel Joly
ESILV, Courbevoie, France

January 2021

Summary

Introduction

The porting of the NASA/aladyn 2.0 program, from Fortran to Python, was initiated by NEC to the group of two students from ESILV. Initially, we had to convert the Fortran program into Python, to optimize it with Pytorch and to port it in several architectures like CPU, GPU and SX-Aurora TSUBASA vector processor to benchmark it. After few weeks of work, we modified the final goals to make only the conversion, the validation of the Python version of Aladyn and to make a port documentation (this one).

This document is about our work on the Aladyn program, from its conversion in Python to its validation. We talk about the strategies, those who have been used and those who haven't, the files we made, the issues met and mostly our choices over the work to be done. We also deal with the results of the conversion and the validation, and the possible ideas to improve the Python program.

Before diving into the main subject, we want to make a disclaimer: we are students, and we cannot guarantee the accuracy of our observations and assumptions. Do not take for granted what we are going to say, especially on the last part of the document.

1. Environment

We used two different OS during the project. Windows 10 was used for developing, debugging and validating the Python part of the project. To perform these actions, we used Notepad and PyCharm to write Python code, Notepad++ to visualize Fortran sources, structure.plt and ANN.dat files, the Command Prompt and PowerShell to run the Python interpreter. Windows was also used to manage the file sharing and the information between the two authors, through Messenger, Mattermost, Teams and Github. Linux virtual machine, thanks to VMware and VirtualBox, was used to compile, launch and debug the Fortran version of Aladyn. We compiled the Fortran sources using the GNU, the Intel and the PGI compilers.

2. Conversion strategy

I overall strategies

The strategy to convert the Fortran version of Aladyn was to convert file by file, function by function, instruction by instruction into Python language. The Fortran version of Aladyn used during the whole project was the version 2.0, available on the NASA/aladyn Github repository. We mainly converted the whole Fortran source code, except the files using parallel computation libraries OpenMP and OpenACC. The files are: *aladyn_sys_ACC.f*, *aladyn_sys_OMP.f* and *aladyn_sys_NO_OMP.f*.

To convert the Fortran version of NASA/aladyn, we first divide the task in two. Thus, one author worked on the conversion of the aladyn_ANN.f file while the other author worked on the conversion of the *aladyn.f*, *aladyn_IO.f*, *aladyn_MD.f*, *aladyn_mods.f* and *aladyn_sys.f*. We converted around 5200 lines of Fortran into around 6000 lines of Python. We first thought it was a good idea to convert each module into a class. However, oriented-object approach brought unnecessary complexity to the project.

After the full conversion done in each side of the conversion project, we merged all files into the same repository on begin to work on the validation from there. See the next part for more details on the harmonization, the validation and the hotfixes.

II Array and loops

We took care about some difference between Fortran and Python. For example, the interval (or size) of an array in Fortran can be [x, y] (with x and y >0 and x < y) whereas an array in Python must begin at 0, so the interval of an array in Python compared to Fortran is [0, y]. The more x is far from 0, the more memory you burn. It was one of the conversion choices we made all along the project.

We choose to keep intact all the index and iterators are they are used sometimes in calculation. It also helps when debugging. ArrayExample[3] in python or in Fortran will give the same result, except if it doesn't exist in Fortran, it may be a 0 or a blank in python.

For this, we had to make some adaptations to the python code. In loops, we usually to the number of elements +1. Also, we kept this from Fortran: every array has its own variable to keep the number of elements it contains. We almost only use these variables instead of the python len() function.

III Variables scop, allocation & deallocations

As in Fortran, we kept most of the global variables. This causes a big change in python because for every call of one of these variables, we had to find the file where it came from. So, most of the errors we had to resolve came from a mistake here. In every function except in the aladyn_ANN.py file, we only use the useful global variables are passed into the function in the aladyn_ANN.py file, we passed all the global variable without sorting them, there is a little optimization to do here.

For the allocation and deallocation of the arrays, we absolutely followed Fortran. When Fortran allow space to an array, we filled an array with 0 or blanks. Deallocating is useless in python but still, we assigned an empty array to the deallocated array. There are some big optimizations to do here but we wanted to stick with Fortran as much as we could.

IV Import order & modification in the code structure

Importing every python file was a big challenge as python hates circular imports.

This is the final import order we choose. With this configuration we had to do only one concession.

File names	Imports
Aladyn	PROGRAM_END, aladyn_sys, sys_ACC, atoms, sim_box, constants, pot_module, node_conf, MD, aladyn_IO, aladyn_ANN
PROGRAM_END	Atoms, aladyn_ANN, sim_box, pot_module
atoms	sim_box

sim_box	PROGRAM_END
aladyn_ANN	Atoms, sim_box, pot_module
pot_module	Atoms, sim_box
aladyn_sys	
sys_ACC	
constants	
node_conf	
MD	Atoms, sim_box, constants, pot_module
aladyn_IO	Atoms, sim_box, constants, pot_module, PROGRAM_END, aladyn_ANN
Aladyn_ANN	Atoms, sim_box, pot_module

The concession was to move 4 function from the aladyn file to the aladyn_IO file.

These functions are:

- link_cell_setup()
- nnd_fit(nodes_on_D, iD)
- get_config(i1, i2, i3, nodes_X, nodes_Y, nodes_Z)
- node_config()

Finally, we chose to make node_conf as a class, just because it seems to be presented like a kind of struct in Fortran, but these changes have no effects.

V Final files

The final files of the Python version of Aladyn are:

- *aladyn.py* (from aladyn.f),
- *aladyn_ANN.py* (from aladyn_ANN.f),
- *aladyn_IO.py* (from aladyn_IO.f and some aladyn.f functions),
- *aladyn_sys.py* (from aladyn_sys.f), sys_ACC.py (from aladyn_sys.f),
- *atoms.py* (from aladyn_mods.f - atoms module),
- *constants.py* (from aladyn_mods.f - constants module),
- *group_conf.py* (from aladyn_mods.f - sim_box module),
- *node_conf.py* (from aladyn_mods.f - sim_box module),
- *sim_box.py* (from aladyn_mods.f - sim_box module),
- *pot_module.py* (from aladyn_mods.f - pot_module module),
- *MD.py* (from aladyn_MD.f),
- *PROGRAM_END.py* (from aladyn.f)

3. Validation strategy

Before beginning the validation work, we took care to make good basis to perform a good validation of the Python's version.

First, we made a *python-calls-graph* excel file (it is in the repo), based on a Google Sheet one we made specially for the validation. It permits to see all the. Then, we divide the work based on time: one author worked at the beginning of the week while the other worked after him during the few last days of the week.

We also made two layers of validation to the validation procedure: the first one to work on not-validated parts of the program would perform rapid and precise hotfixes to advance in the execution of the program and scout if we will encounter critical bugs. It is the first layer of validation. The other one would validate slowly and structurally the previously validated parts of Python source code to get as close as possible to an accurate execution of the original Fortran program, both logically and visual.

Memory allocation form and arithmetic computation accuracy were also very important for us during the process of validation.

4. Results & observations

I Overall results

After converting then validating the program into Python language, we have run the program. During development, one of the authors launched Python version of Aladyn using Microsoft PowerShell on a Windows 10 environment with this command:

```
PS > python aladyn.py
```

The file *aladyn.py* being the main file of the project (with *ParaGrandMC* being the main function), it launched the program. On a Linux distribution, with the required packages to run a Python program, you can run the program with the following command:

```
$ python3 aladyn.py
```

We highly recommend using Python 3 simply because we develop the program with this major version.

The launch and the entire execution of the Python version of Aladyn works nearly as the Fortran's version. The displays are also accurate to the Fortran's version, expect a few formatted lines and the structure.plt output file. You will probably notice, especially if you debug or add some prints to the two versions, that they do not have the same floating-point management or at least display. Most of the noticeable examples appears at 10^-14 approximately.

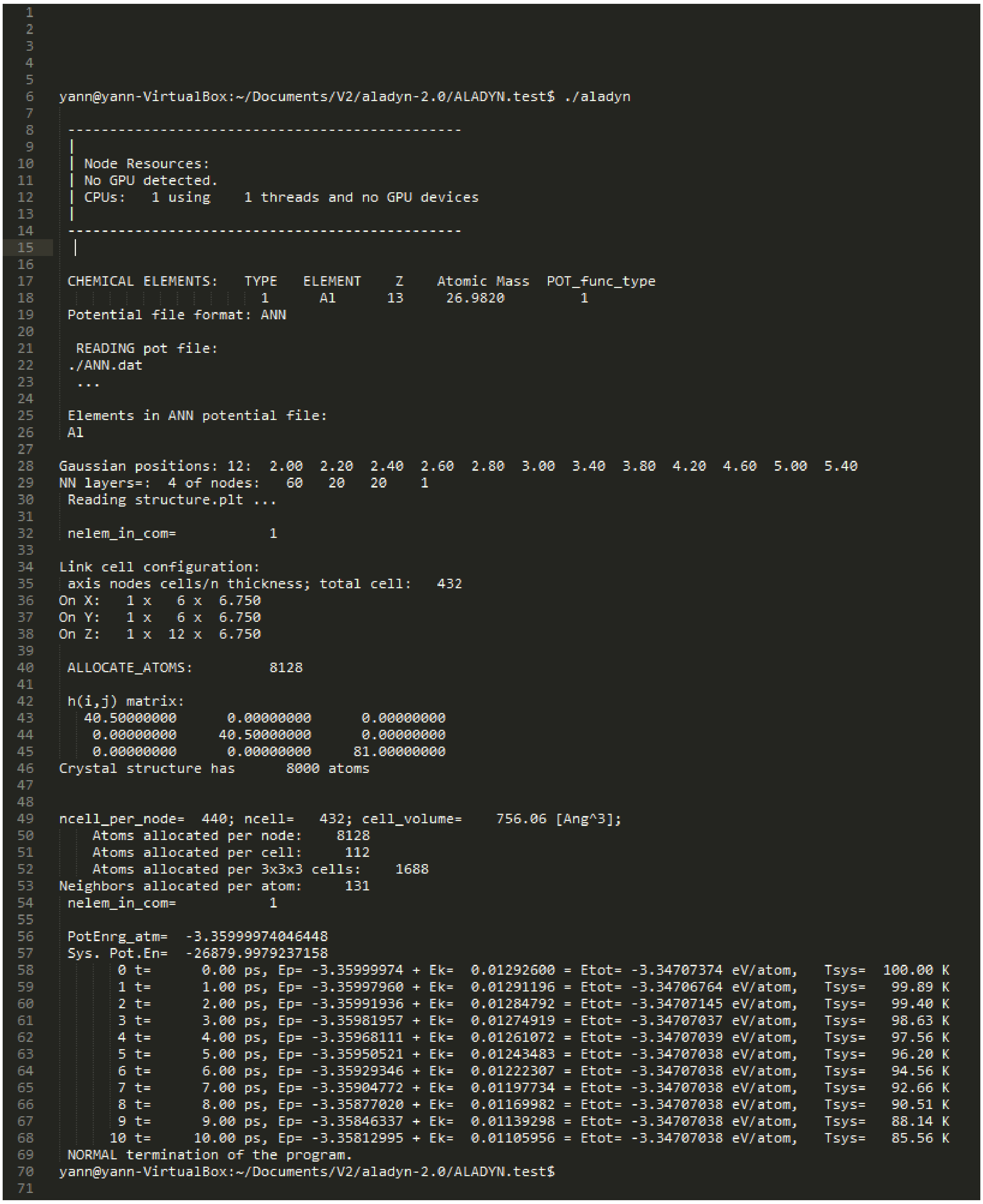
The main problem of the Python version that was raised after the first working execution of the *Frc_ANN_OMP* function (which is called each time we call *force*, which is called by force_global knowing that this function is called up to *n*2 + 2 – n* being the number of MD steps) is that the Python version of Aladyn is much slower than the Fortran version. Whereas the runtime of the Fortran version lasts 1 minute and 20 seconds, the Python version lasts between 10 and 12 hours in theory, depending on the architecture of the computer (these times have been recorded on a laptop computer running Windows 10 Pro with a i7 8750H CPU and with 16GB of RAM).

Other than this problem, there are no known bug for the moment in the Python version and the program is working correctly (one could rather say functional, because unfortunately it is very slow).

II Comparison of the outputs with Fortran

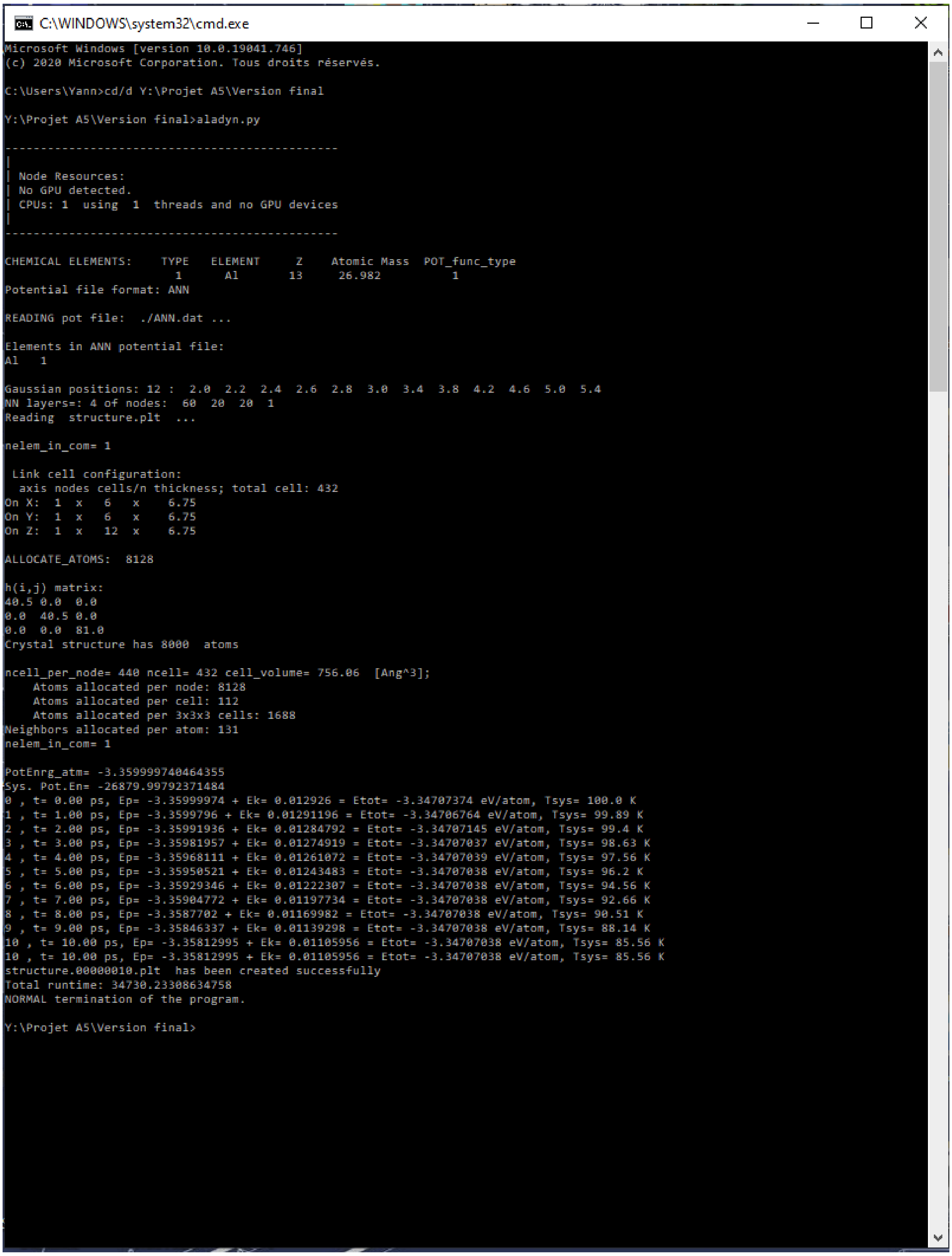
This is a comparison between the Fortran original compilation as furnish on the official GitHub of Aladyn and our submitted python version. We choose to describe here the most basic execution. We give no parameters, so the default options are used (m=1, n=10). The plt file is Al_N8000.plt from the Aladyn GitHub too:

Fortran console output:



```
yann@yann-VirtualBox:~/Documents/V2/aladyn-2.0/ALADYN.test$ ./aladyn

 -----------------------------------------------
 |
 | Node Resources:
 | No GPU detected.
 | CPUs:    1 using    1 threads and no GPU devices
 |
 -----------------------------------------------


 CHEMICAL ELEMENTS:   TYPE   ELEMENT    Z    Atomic Mass  POT_func_type
                        1      Al      13     26.9820         1
 Potential file format: ANN

  READING pot file:
 ./ANN.dat
  ...

 Elements in ANN potential file:
 Al

Gaussian positions: 12:  2.00  2.20  2.40  2.60  2.80  3.00  3.40  3.80  4.20  4.60  5.00  5.40
NN layers=:  4 of nodes:   60   20   20    1
 Reading structure.plt ...

 nelem_in_com=           1

Link cell configuration:
 axis nodes cells/n thickness; total cell:   432
On X:    1 x    6 x  6.750
On Y:    1 x    6 x  6.750
On Z:    1 x   12 x  6.750

 ALLOCATE_ATOMS:         8128

 h(i,j) matrix:
   40.50000000      0.00000000      0.00000000
    0.00000000     40.50000000      0.00000000
    0.00000000      0.00000000     81.00000000
Crystal structure has      8000 atoms


ncell_per_node=  440; ncell=   432; cell_volume=    756.06 [Ang^3];
    Atoms allocated per node:    8128
    Atoms allocated per cell:     112
    Atoms allocated per 3x3x3 cells:    1688
Neighbors allocated per atom:     131
 nelem_in_com=           1

 PotEnrg_atm=  -3.35999974046448
 Sys. Pot.En=  -26879.9979237158
       0 t=      0.00 ps, Ep= -3.35999974 + Ek=  0.01292600 = Etot= -3.34707374 eV/atom,   Tsys=  100.00 K
       1 t=      1.00 ps, Ep= -3.35997960 + Ek=  0.01291196 = Etot= -3.34706764 eV/atom,   Tsys=   99.89 K
       2 t=      2.00 ps, Ep= -3.35991936 + Ek=  0.01284792 = Etot= -3.34707145 eV/atom,   Tsys=   99.40 K
       3 t=      3.00 ps, Ep= -3.35981957 + Ek=  0.01274919 = Etot= -3.34707037 eV/atom,   Tsys=   98.63 K
       4 t=      4.00 ps, Ep= -3.35968111 + Ek=  0.01261072 = Etot= -3.34707039 eV/atom,   Tsys=   97.56 K
       5 t=      5.00 ps, Ep= -3.35950521 + Ek=  0.01243483 = Etot= -3.34707038 eV/atom,   Tsys=   96.20 K
       6 t=      6.00 ps, Ep= -3.35929346 + Ek=  0.01222307 = Etot= -3.34707038 eV/atom,   Tsys=   94.56 K
       7 t=      7.00 ps, Ep= -3.35904772 + Ek=  0.01197734 = Etot= -3.34707038 eV/atom,   Tsys=   92.66 K
       8 t=      8.00 ps, Ep= -3.35877020 + Ek=  0.01169982 = Etot= -3.34707038 eV/atom,   Tsys=   90.51 K
       9 t=      9.00 ps, Ep= -3.35846337 + Ek=  0.01139298 = Etot= -3.34707038 eV/atom,   Tsys=   88.14 K
      10 t=     10.00 ps, Ep= -3.35812995 + Ek=  0.01105956 = Etot= -3.34707038 eV/atom,   Tsys=   85.56 K
 NORMAL termination of the program.
yann@yann-VirtualBox:~/Documents/V2/aladyn-2.0/ALADYN.test$
```

Python console output:



C:\WINDOWS\system32\cmd.exe

```
Microsoft Windows [version 10.0.19041.746]
(c) 2020 Microsoft Corporation. Tous droits réservés.

C:\Users\Yann>cd/d Y:\Projet A5\Version final

Y:\Projet A5\Version final>aladyn.py

-----------------------------------------------
|
| Node Resources:
| No GPU detected.
| CPUs: 1  using  1  threads and no GPU devices
|
-----------------------------------------------

CHEMICAL ELEMENTS:    TYPE   ELEMENT     Z    Atomic Mass  POT_func_type
                        1      Al       13     26.982          1
Potential file format: ANN

READING pot file:  ./ANN.dat ...

Elements in ANN potential file:
Al   1

Gaussian positions: 12 :  2.0  2.2  2.4  2.6  2.8  3.0  3.4  3.8  4.2  4.6  5.0  5.4
NN layers=: 4 of nodes:  60  20  20  1
Reading  structure.plt  ...

nelem_in_com= 1

 Link cell configuration:
  axis nodes cells/n thickness; total cell: 432
On X:  1  x   6   x    6.75
On Y:  1  x   6   x    6.75
On Z:  1  x   12  x    6.75

ALLOCATE_ATOMS:  8128

h(i,j) matrix:
40.5 0.0  0.0
0.0  40.5 0.0
0.0  0.0  81.0
Crystal structure has 8000  atoms

ncell_per_node= 440 ncell= 432 cell_volume= 756.06  [Ang^3];
    Atoms allocated per node: 8128
    Atoms allocated per cell: 112
    Atoms allocated per 3x3x3 cells: 1688
Neighbors allocated per atom: 131
nelem_in_com= 1

PotEnrg_atm= -3.359999740464355
Sys. Pot.En= -26879.99792371484
0 , t= 0.00 ps, Ep= -3.35999974 + Ek= 0.012926 = Etot= -3.34707374 eV/atom, Tsys= 100.0 K
1 , t= 1.00 ps, Ep= -3.3599796 + Ek= 0.01291196 = Etot= -3.34706764 eV/atom, Tsys= 99.89 K
2 , t= 2.00 ps, Ep= -3.35991936 + Ek= 0.01284792 = Etot= -3.34707145 eV/atom, Tsys= 99.4 K
3 , t= 3.00 ps, Ep= -3.35981957 + Ek= 0.01274919 = Etot= -3.34707037 eV/atom, Tsys= 98.63 K
4 , t= 4.00 ps, Ep= -3.35968111 + Ek= 0.01261072 = Etot= -3.34707039 eV/atom, Tsys= 97.56 K
5 , t= 5.00 ps, Ep= -3.35950521 + Ek= 0.01243483 = Etot= -3.34707038 eV/atom, Tsys= 96.2 K
6 , t= 6.00 ps, Ep= -3.35929346 + Ek= 0.01222307 = Etot= -3.34707038 eV/atom, Tsys= 94.56 K
7 , t= 7.00 ps, Ep= -3.35904772 + Ek= 0.01197734 = Etot= -3.34707038 eV/atom, Tsys= 92.66 K
8 , t= 8.00 ps, Ep= -3.3587702 + Ek= 0.01169982 = Etot= -3.34707038 eV/atom, Tsys= 90.51 K
9 , t= 9.00 ps, Ep= -3.35846337 + Ek= 0.01139298 = Etot= -3.34707038 eV/atom, Tsys= 88.14 K
10 , t= 10.00 ps, Ep= -3.35812995 + Ek= 0.01105956 = Etot= -3.34707038 eV/atom, Tsys= 85.56 K
10 , t= 10.00 ps, Ep= -3.35812995 + Ek= 0.01105956 = Etot= -3.34707038 eV/atom, Tsys= 85.56 K
structure.00000010.plt  has been created successfully
Total runtime: 34730.23308634758
NORMAL termination of the program.

Y:\Projet A5\Version final>
```

We will always describe using the file in Fortran.

Line 18: Fortran have a defined number of significant figures (4) whereas python have not.

Lines 21 -23: Python have no line break here.

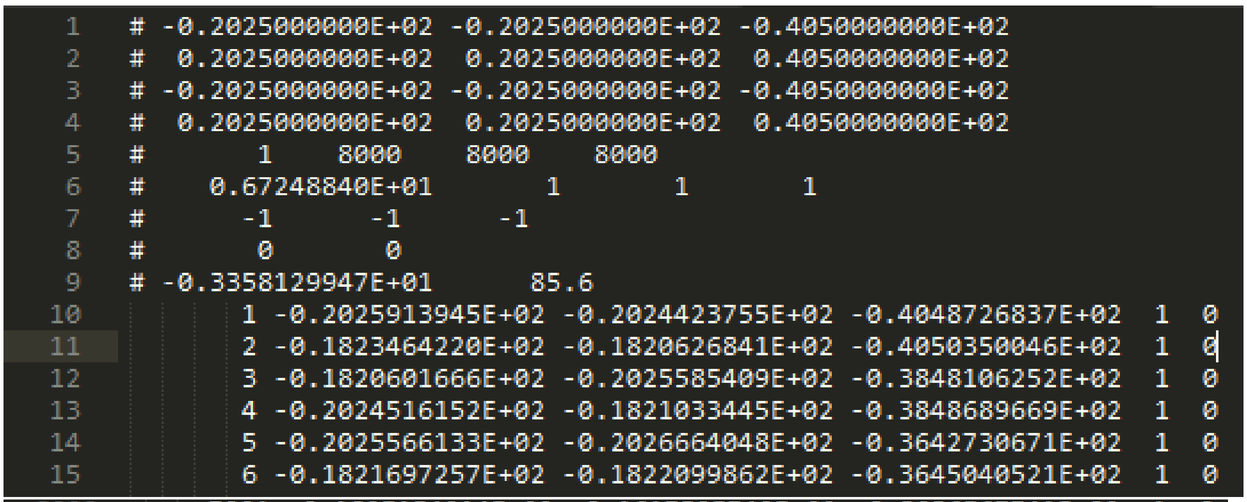
Line 26: We choose to write again the Type number of the element. This information is redundant in the plt file and in the ANN file. But a difference in the Type numbers may cause unexpected behaviours

Line 28, 36 - 38, 43 - 45: Fortran have a defined number of significant figures (8) whereas python have not, but at least one number after the comma.

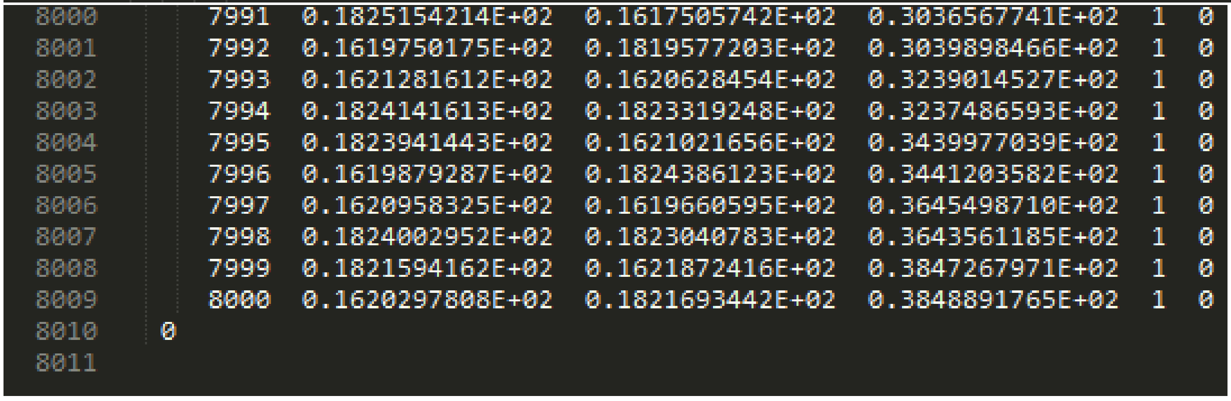
Line 58 – 68: Fortran have a defined number of significant figures (8) and values are aligned. Python have a maximum number after the comma (8) and value are not aligned.

Line 69: We added execution time in sec and the name of the created plt file in the working directory.

Fortran plt result:

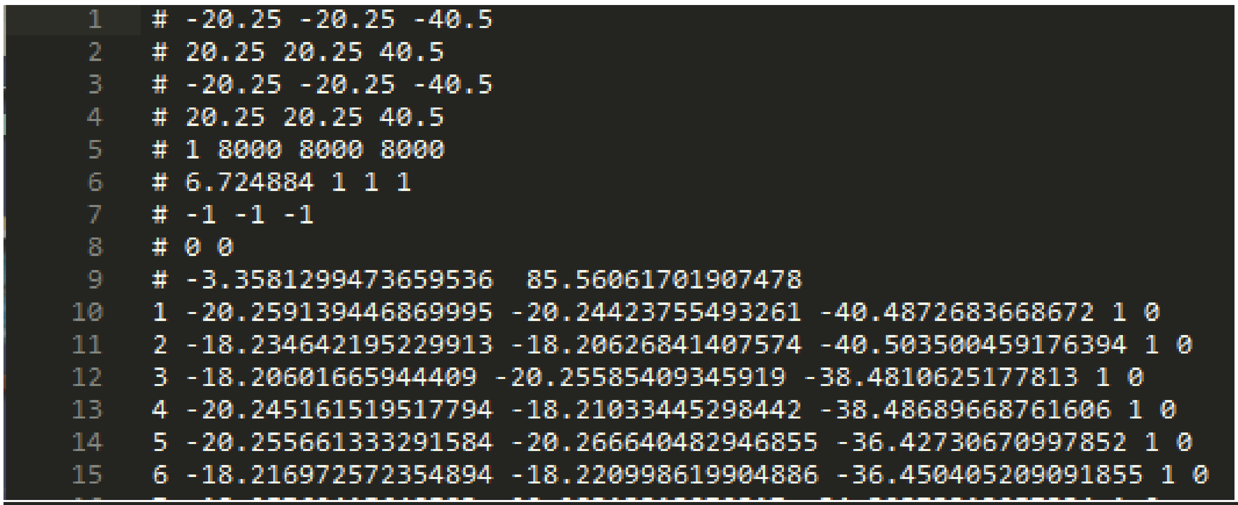


```
# -0.2025000000E+02 -0.2025000000E+02 -0.4050000000E+02
#  0.2025000000E+02  0.2025000000E+02  0.4050000000E+02
# -0.2025000000E+02 -0.2025000000E+02 -0.4050000000E+02
#  0.2025000000E+02  0.2025000000E+02  0.4050000000E+02
#        1    8000     8000     8000
#    0.67248840E+01        1        1        1
#      -1       -1       -1
#       0        0
# -0.3358129947E+01       85.6
           1 -0.2025913945E+02 -0.2024423755E+02 -0.4048726837E+02  1  0
           2 -0.1823464220E+02 -0.1820626841E+02 -0.4050350046E+02  1  0
           3 -0.1820601666E+02 -0.2025585409E+02 -0.3848106252E+02  1  0
           4 -0.2024516152E+02 -0.1821033445E+02 -0.3848689669E+02  1  0
           5 -0.2025566133E+02 -0.2026664048E+02 -0.3642730671E+02  1  0
           6 -0.1821697257E+02 -0.1822099862E+02 -0.3645040521E+02  1  0
```

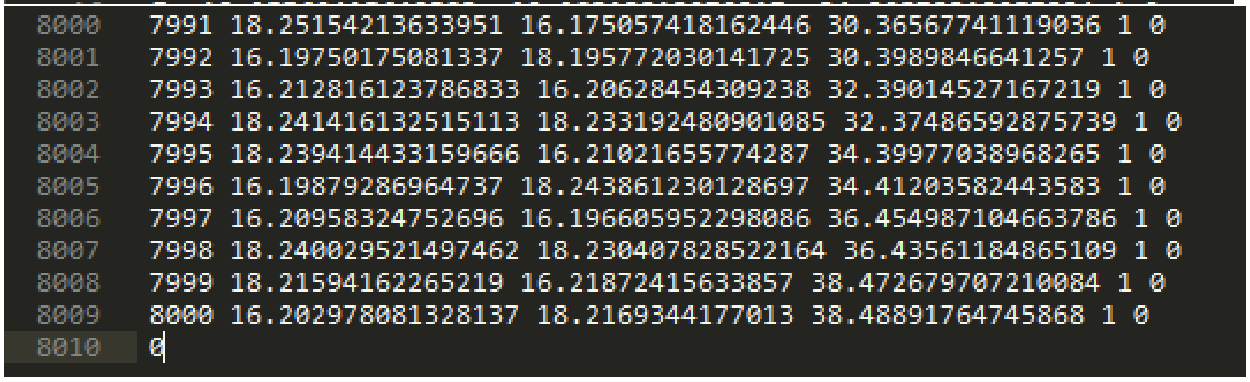


```
        7991  0.1825154214E+02  0.1617505742E+02  0.3036567741E+02  1  0
        7992  0.1619750175E+02  0.1819577203E+02  0.3039898466E+02  1  0
        7993  0.1621281612E+02  0.1620628454E+02  0.3239014527E+02  1  0
        7994  0.1824141613E+02  0.1823319248E+02  0.3237486593E+02  1  0
        7995  0.1823941443E+02  0.1621021656E+02  0.3439977039E+02  1  0
        7996  0.1619879287E+02  0.1824386123E+02  0.3441203582E+02  1  0
        7997  0.1620958325E+02  0.1619660595E+02  0.3645498710E+02  1  0
        7998  0.1824002952E+02  0.1823040783E+02  0.3643561185E+02  1  0
        7999  0.1821594162E+02  0.1621872416E+02  0.3847267971E+02  1  0
        8000  0.1620297808E+02  0.1821693442E+02  0.3848891765E+02  1  0
 0
```

Python plt result:



```
 1   # -20.25 -20.25 -40.5
 2   # 20.25 20.25 40.5
 3   # -20.25 -20.25 -40.5
 4   # 20.25 20.25 40.5
 5   # 1 8000 8000 8000
 6   # 6.724884 1 1 1
 7   # -1 -1 -1
 8   # 0 0
 9   # -3.3581299473659536  85.56061701907478
10   1 -20.259139446869995 -20.24423755493261 -40.4872683668672 1 0
11   2 -18.234642195229913 -18.20626841407574 -40.503500459176394 1 0
12   3 -18.20601665944409 -20.25585409345919 -38.4810625177813 1 0
13   4 -20.245161519517794 -18.21033445298442 -38.48689668761606 1 0
14   5 -20.255661333291584 -20.266640482946855 -36.42730670997852 1 0
15   6 -18.216972572354894 -18.220998619904886 -36.450405209091855 1 0
```



```
8000   7991 18.25154213633951 16.175057418162446 30.36567741119036 1 0
8001   7992 16.19750175081337 18.195772030141725 30.3989846641257 1 0
8002   7993 16.212816123786833 16.20628454309238 32.39014527167219 1 0
8003   7994 18.241416132515113 18.233192480901085 32.37486592875739 1 0
8004   7995 18.239414433159666 16.21021655774287 34.39977038968265 1 0
8005   7996 16.19879286964737 18.243861230128697 34.41203582443583 1 0
8006   7997 16.20958324752696 16.196605952298086 36.454987104663786 1 0
8007   7998 18.240029521497462 18.230407828522164 36.43561184865109 1 0
8008   7999 18.21594162265219 16.21872415633857 38.472679707210084 1 0
8009   8000 16.202978081328137 18.2169344177013 38.48891764745868 1 0
8010   0
```

The values are always strictly the same.

Fortran use scientific notation when python just writes doubles. The python separator is always a simple space, whereas it is about an alignment in FORTRAN. Finally, the Fortran file have an empty line at 8011.

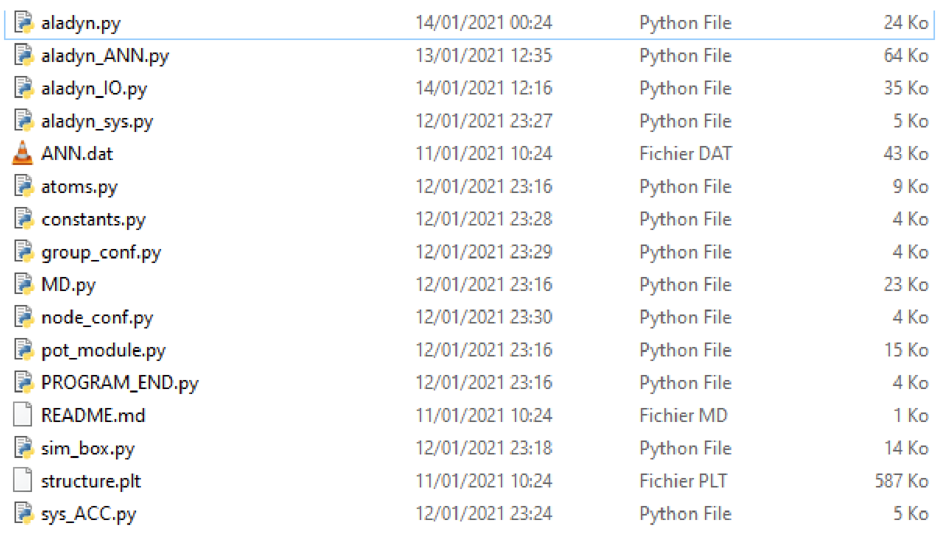
For these reasons, the resulting python plt file is approximately 10% lighter than the Fortran one. The Aladyn program take 2 entries and some options. One of this entry is a plt file. Therefore, a generated plt file should also fit in Aladyn as an input. We verified that the python file can be used as an input plt.

So these two resulting files remain interchangeable and none of its differences disturb the execution of the program.

III How to launch & options

Only requirement is to install python 3. We use no library except sys, operator, os and math, which are native library of python.

Verify that your working directory contains all these files:



aladyn.py	14/01/2021 00:24	Python File	24 Ko
aladyn_ANN.py	13/01/2021 12:35	Python File	64 Ko
aladyn_IO.py	14/01/2021 12:16	Python File	35 Ko
aladyn_sys.py	12/01/2021 23:27	Python File	5 Ko
ANN.dat	11/01/2021 10:24	Fichier DAT	43 Ko
atoms.py	12/01/2021 23:16	Python File	9 Ko
constants.py	12/01/2021 23:28	Python File	4 Ko
group_conf.py	12/01/2021 23:29	Python File	4 Ko
MD.py	12/01/2021 23:16	Python File	23 Ko
node_conf.py	12/01/2021 23:30	Python File	4 Ko
pot_module.py	12/01/2021 23:16	Python File	15 Ko
PROGRAM_END.py	12/01/2021 23:16	Python File	4 Ko
README.md	11/01/2021 10:24	Fichier MD	1 Ko
sim_box.py	12/01/2021 23:18	Python File	14 Ko
structure.plt	11/01/2021 10:24	Fichier PLT	587 Ko
sys_ACC.py	12/01/2021 23:24	Python File	5 Ko

Note that you can change the plt file by another. You just have to replace de "structure .plt" file. Be sure to keep the same name. You can now use your terminal to go to your working directory and type python aladyn.py to execute the program. To execute the program with options, you can for example type python

```
aladyn.py -m 100 -n 10
```

N or n represent the number MD steps (default 10). M or m represent the measure step (default 1)/

IV tested options and results

We tested all the STR files from Test folder of this aladyn github: https://github.com/nasa/aladyn

All the execution has been done one windows 10 using python 3.8.6 with a CPU intel i7 3790K with 10% OC and 24go RAM

We used default option so m = 1 and n =10

All tests have been finished with the message "NORMAL termination of the program".

Here are the executions times:

File	days	hours	mins	sec
Al_N4000.plt	0	4	24	12
Al_N8000.plt	0	10	11	23
Al_N16000.plt	0	23	16	04
Al_N32000.plt	1	17	58	58
Al_N64000.plt	3	22	23	10
Al_N128000.plt	7	12	47	59
Al_N192000.plt	11	6	54	41
Al_N256000.plt	15	1	20	15

We have also done 2 tests:

First, we took the result plt file from the precedent test using Al_N8000.plt. This result plt file is almost the same as the original Al_N8000.plt file, so had exactly the same result, this is what we expected. Execution time was similar too.

Finally, we tested the program with options m=100 and n=10 we used Al_N32000.plt. We had a normal termination of the program after 8 days. To conclude, we have not yet detected any case where the execution in python has failed on this configuration. We had some crash on lower configuration, essentially memory errors, probably caused by the excessive RAM consumption of this program (up to 17 go)

5. Optimizations and beyond

The problem of slowness lies on the language itself – Python – and very possibly on our conversion choices compared to Fortran. Keep in mind that the Python's version runtime will take approximately 300 more time than the Fortran version, knowing that the Fortran's version did not use the OpenMP and OpenACC during the time comparisons we made.

Nonetheless, we have several ideas to offer if the program is subsequently used to be optimized and then used for benchmarking:

- Compare the differences in logic and memory usage between Fortran and Python. For example: calls to variables according to the scopes (file, class); arrays of size [X-Y] in Fortran while there are arrays of size [0-Y] in Python; etc.
- The use of parallel computation libraries like OpenMP and OpenACC (there is a very large number of assignments in nested loops, of vector / matrix computations to parallelize).
- Optimization of Python source code compilation via the interpreter. I suppose there must be some debugging options on the Python interpreter to do profiling, code coverage, or an option identical to -O2 / -O3 used with GCC.
- The use of Pytorch. This was what was agreed if we had more time for this project.
- Delete the variables declared but not used, and the variables declared, used but useless (not necessarily visible directly by the IDE or the compiler). Most of the output variables for several functions are useless with Python, whereas they were primordial for Fortran. It consumes memory so it could be a good idea to get rid of these variables and the way the functions return them.