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

Yann Abou Jaoude
ESILV, Courbevoie, France

Axel Joly
ESILV, Courbevoie, France

Summary

Introduction

1. **Environment**
2. **Conversion strategy**
3. **Validation strategy**
4. **Results & observations**
5. **Optimizations and beyond**

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

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.

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.

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.

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

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).

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.