Exercise 4: FORTRAN *and* GNUPLOT *multi-run and automated script using* PYTHON.

Iriarte Delfina (1231682)

Quantum Information

October 27, 2020

Abstract

In this report, a PYTHON code is implemented in order to perform **multi-run** and **automated** script in FORTRAN and GNUPLOT. PYTHON works as an interface among them, providing a simpler way to run them several times. In particular, a matrix multiplication program in FORTRAN, that computes the CPU times of several approaches, were implement changing the input size of the matrices by running a PYTHON code. Finally, the data were **fitted** using GNUPLOT and **optimized** in a PYTHON code.

1 Introduction

PHYTON is a programming language that enables a great **flexibility**. Unfortunately, does not always come pre-equipped with the speed necessary to perform intense numerical computation. The inflexibility enables programs to assign spots in memory to each variable that can be **accessed efficiently** and it eliminates the need to **check their type** before performing an operation on it.

However, a solution is to write the expensive steps in a fast language like FORTRAN and to keep everything else in PYTHON. Fortunately, this is a very easy task since there are a large number of modules that can be imported in PYTHON that helps to perform this.

In this report, a matrix multiplication program with several approaches built-in FORTRAN is optimized into a PYTHON code by using the MODULE SUBPROCESS. Moreover, the computational time's results obtained for the diverse models are fitted using GNUPLOT and run efficiently into a PYTHON code.

2 Code development

In FORTRAN, a matrix multiplication program is built with diverse approaches. In particular, two SUBROUTINES containing a theoretical matrix multiplication implementation made by looping through the indices, which each case corresponds to different loops, were performed. Moreover, the `matmul` function provided by FORTRAN was also implemented. The computational time was performed by using the `cpu_time` function provided by FORTRAN.

For simplicity, the computation was performed by two square matrices with the same dimension. The matrix dimension N was read from an input file as shown in Listing 1.

```
open(12, file = 'input_data.dat', status = 'old', action = 'readwrite')
read(12, *) n
```

Listing 1. FORTRAN implementation of input matrix dimension N

A PYTHON code was provided in order to work as an interface with FORTRAN. In particular, we use the MODULE SUBPROCESS provided by PYTHON in order to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.

A FUNCTION that takes as input the minimum (`N_min`) and maximum (`N_max`) of the matrix dimension, and the step size of it (`N_step`) is implemented, as shown in Listing 2. It performs a loop over the dimension of the matrix. Initially, it writes them in an output file, that will be the same one that the FORTRAN code would use to read, and then call the matrix multiplication code from FORTRAN mentioned before. The results obtained for the several multiplication approaches are saved in an array and finally in a file using the command `np.savetxt`.

```
def fortran_loop(N_min, N_max, N_step):
    #Performs the matrix x matrix fortran code and return the values into this Python file.

    results = []
    for i in range (N_min, N_max, N_step):
        fn = open("input_data.dat", "w")
        fn.write(str(i))
        fn.close()

        bash_results = subprocess.run(["./p2", str(i)], stdout=subprocess.PIPE)
        splits = bash_results.stdout.decode('utf-8').lstrip().split('\n')
        print(splits[0:3])
        to_float = np.array([float(tm) for tm in splits[0:3]])
        results.append(to_float)

    return results
```

Listing 2. PYTHON function for calling the matrix multiplication program made in FORTRAN

The function was implemented by setting the parameters `N_min = 10, N_max = 1500` and `N_step=25`, with no optimization flags and using the `-Oflag` optimization flag in FORTRAN. Furthermore, the computational time for the diverse methods was fitted using GNUPLOT as shown in Listing 3. As it can be seen, it reads input data from a file, as well as the output file name, which correspond to the `.pdf` file where the plots are going to be saved.

The fitting was performed using the command `fit` from GNUPLOT. Moreover, two fits were provided, a linear one (for the logarithmic case) and a polynomial of order 3. Finally, the obtained parameters of the fit were saved in an output file using the command `set print`.

```
set xlabel "Log(N)"
set ylabel 'Log(Cpu time)'
set title "Fitting"
set key left

fname = system("cat fit_file.txt")
oname = system("cat out_file.txt")

f(x) = a + b * x

fit f(x) fname u (log($1)):(log($2)) via a, b

plot fname u (log($1)):(log($2)) t fname w lp ls 3 lt 3 pt 7 linecolor 1, f(x) t "Linear Fit"
    w l ls 3 lt 3 linecolor 6


set print 'logparameters'.fname
print "#f(x)=a+b*x"
print a,a_err
print b,b_err

set terminal pdfcairo color
```

```
set output oname
rep
set terminal post
```

Listing 3. GNUPLOT for fitting data

The GNUPLOT code was optimized by implementing it in a PHYTON code as shown in Listing 4: it loops over the different files, previously generated when running the FORTRAN code, and call the GNUPLOT process by using the `subprocess` module.

```
#implemented the gnuplot FIT in ptyhon with the different methods with cubic fit.
in_files = ["matmul.txt", "mat_1.txt", "mat_2.txt"]
out_files = ["matmul_fit.pdf", "mat_1_fit.pdf", "mat_2_fit.pdf"]

for i in range(3):
    fl = open("fit_file.txt", "w+")
    fl.write(in_files[i])
    fl.close()

    fn = open("out_file.txt", "w+")
    fn.write(out_files[i])
    fn.close()
    subprocess.call(["gnuplot","Ex4_fit.gp"])
```

Listing 4. PYTHON code optimization for fitting data

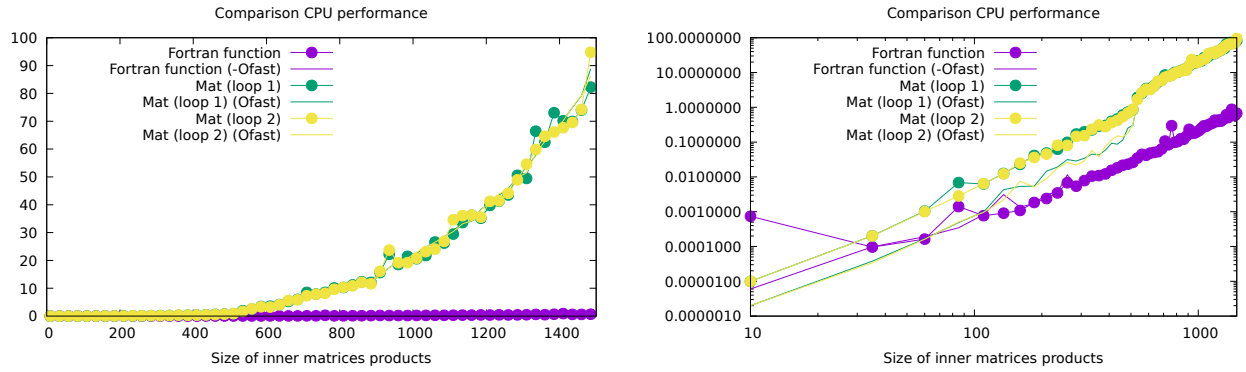
Finally, with the parameters obtained by the fitting, a function `param(x, model, log = False)` is defined as shown in Listing 5. It takes as input the matrix multiplication model, a logical value log (which allows enabling the liner fit or the polynomial), and an array x. The functionality is to return the fitting function obtained with the corresponding parameter of a given model by reading them in their files.

```
def param(x, model, log = False):

    #function that returns the parameters obtained by fitting in gnuplot.
    if model == matmul: data = np.loadtxt('parametersmatmul.txt')
    elif model == mat_1: data = np.loadtxt('parametersmat_1.txt')
    elif model == mat_2: data = np.loadtxt('parametersmat_2.txt')

    a = data[0][0]
    b = data[1][0]
    c = data[2][0]
    d = data[3][0]

    result = a +b *x + c * x**2 + d * x **3

    if log == True:
        if model == matmul: data = np.loadtxt('logparametersmatmul.txt')
        elif model == mat_1: data = np.loadtxt('logparametersmat_1.txt')
        elif model == mat_2: data = np.loadtxt('logparametersmat_2.txt')
        a = data[0][0]
        b = data[1][0]

        result = a +b *x

    return result
```

Listing 5. PYTHON function for getting the parameters of the fitting and return the fit function for any given value x

3 Results

In Figure 1 the CPU time comparison between the multiplication approaches with and without optimization flags are shown. As it can be seen from Figure 1b the `matmul` function is more efficient than the other ones made "by hand." On the other hand, the order of the loop did not affect the performances, however, it can be seen that by using the optimization flag `-Oflag` the performance of the `matmul` function has not been improved, but it did for the others methods.

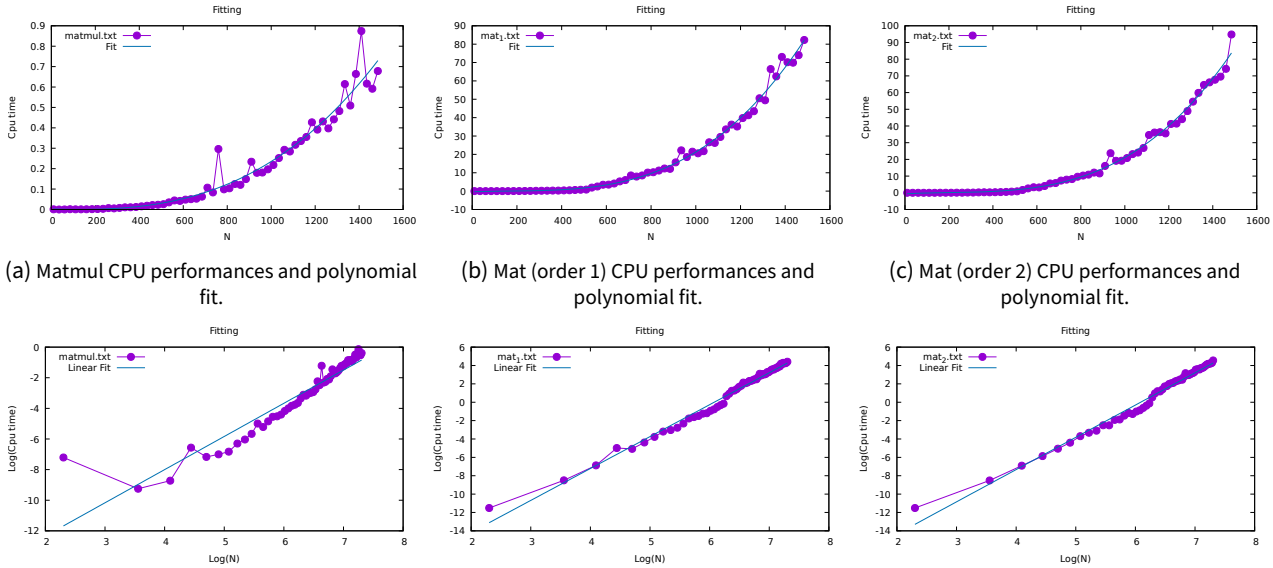


(a) Original scaling.

(b) Logarithmic scaling.

Figure 1. CPU performances for the several approach with and without optimization flags.

Figure 2 shows the original data with their correspond fit obtained by using GNUPLOT for the polynomial and linear case.



(a) Matmul CPU performances and polynomial fit.

(b) Mat (order 1) CPU performances and polynomial fit.

(c) Mat (order 2) CPU performances and polynomial fit.

(d) Matmul CPU performances in logarithmic scale and linear fit.

(e) Mat (order 1) CPU performances in logarithmic scale and linear fit.

(f) Mat (order 2) CPU performances in logarithmic scale and linear fit.

Figure 2. Fitted CPU performances for the diversal approaches

Finally, Figure 3 shows the linear fit model of the 3 different approaches with the obtained parameters of the fitting by GNUPLOT. As can be seen, the slope of the MATMUL function is lower than for the other ones, where the computational time increases very fast making them less feasible to compute.

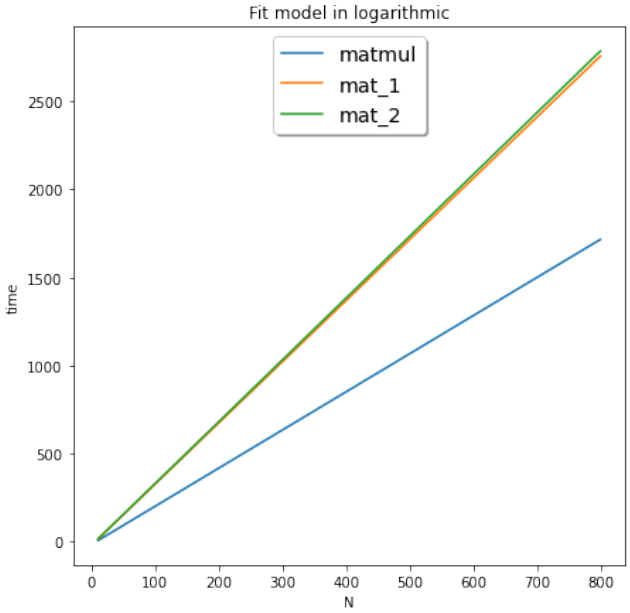


Figure 3. Linear fits obtained for the several models

4 Conclusion

A multi-run script in PYTHON was successfully implemented in a FORTRAN program that computes several approaches of matrix multiplication. It provides a fast and easier way to run FORTRAN and GNUPLOT codes in an automated way without the necessity of multiple files (one per each approach).