Quantum information and Computing
Ex. 04

Andrea Nicolai - 1233407

November 2, 2020

Abstract

In this work we were asked to modify our previous *FORTRAN* code in order make it able to receive an input, namely the matrix size from a file. The value contained by this file was to be written via a Python script that, for each value, would run the executable and compute the times spent in multiplication and store them in different files according to the matricial product used. Finally, two gnuplot scripts were run in order to plot these values and then perform a fit of these times by the mean of a cubical curve, keeping the coefficients obtained distinct according to the file they refer to.

Theory

When writing a program one must take into account that the code must be readable and comfortable to use. That is to say that it must successfully interact with other programming languages via some sort of scripts, automatizing as much as possible all the instructions to be done. This is why we were asked to implement a multi script code that performs many actions in different environments, taking into account that each of them has its own features, with its own pros and cons and roles.

Code Development

We were asked to change the program, already implemented for previous homework, in order to give the executable the opportunity to read from a file the integer size of the matrices to be initialized in the program. Therefore the following code was written.

```
IF(COMMAND_ARGUMENT_COUNT() < 1) THEN
  !set a default size
  size_matrix = 100
  PRINT*, "No␣arguments␣passed,␣setting␣default␣size␣(100)"
ELSE
```

```
  CALL GET_COMMAND_ARGUMENT(1, input_name)
  OPEN (UNIT = 1, FILE = TRIM(input_name))
  READ (UNIT = 1, FMT=*) size_matrix
  PRINT*,"Getting input from", TRIM(input_name), &
  & "matrix size is ", size_matrix
  !PRE-CONDITIONS step0
  !Check whether the size_matrix is either int4 or int8
   IF (DEBUG) THEN
      PRINT*, "Check whether the size_matrix is of type int4"
      PRINT*, assert_is_type(size_matrix, "int4", &
      & DEBUG_PRINT_OUTPUT, EXIT_ON_ERROR)
   END IF
END IF
```

If the executable is called passing a parameter with the filename, the program looks for that specific file. Otherwise, if no arguments are given, it sets the size of the matrix to a predefined value, namely 100. Later, it reads from the file, when given, the integer that will become the size of the matrices used in the program. Finally, it checks whether the number read is an integer as declared. The following one is the Python script implemented to create and deal with the array of values to be used as matrix size:

```
if ( len(sys.argv) >= 4) :
        N_min   = int(sys.argv[1])
        N_max   = int(sys.argv[2])
        N_count = int(sys.argv[3])
else:
        N_min   = 50
        N_max   = 1500
        N_count = 50
array_sizes = np.logspace( np.log10(N_min+1), np.log10(N_max+1),
num=N_count, base=10, dtype=int)
```

Here we see that if we call run the executable with 3 arguments, the first two will define the interval $[N_{min}; N_{max}]$, while the third one, namely N_{count}, will describe how many values we want to take that belong to this interval. Note that the values, which we specify to be integers, are spaced evenly on a log scale. We then compile the code via this script, and purge the environment to delete all files with the same name as the output files. This is done in order to eliminate files that are not involved with this instance of the program and might have been involved in previous runs. The following code describes how the script generates the data:

```
#compile the Fortran script
os.system(compile_program)

#start the script and produce data
for ii in array_sizes:

    istream = open(filename_input_size, "w")
```

```
    istream.write(str(ii))
    istream.close()
    sub.call([run_executable + filename_input_size],
    shell = True)
```

As one can see, we iterate over the array *array_sizes* and write each single value to the file "*sizes.dat*", and later call the executable we have previously compiled, passing it the name of the file as argument. As output of the executable, we will obtain three different files with different names: "definition.dat", "transposed.dat" and "intrinsic.dat". They will actually contain the size of the matrix and the time spent in doing the matricial product.
Later on, we ask Python to run the script "timeplots.gplot" that plots the execution times for different sizes of the matrix. Here we can see a short part of the code, that iterates over the three files containing the data after setting the y scale to logarithmic for a better display.

```
set logscale y
array data[3]
data[1] = "definition.dat"
data[2] = "transposed.dat"
data[3] = "intrinsic.dat"
plot for [i=1:3] data[i]
    using 1:2 title data[i] with linespoints lw 2
```

At this point we were asked to write a Gnuplot script that, when run, would fit the data obtained so far. We chose to fit them using a cubical curve, namely one of this kind:

$$f(x) = ax^3 + bx^2 + cx + d \tag{1}$$

The last part of our Python script involves performing and automatizing fits by iterating over the list of aforementioned filenames.

```
#need to initialize some filenames using gnuplot format
data_files_gnuplot = ['\'definition.dat\'',
'\'transposed.dat\'', '\'intrinsic.dat\'']
output_png_files_gnuplot = ['\'def.png\'',
'\'tra.png\'' , '\'int.png\'']
#need to call the gnuplot fit commands
#and initialize some useful strings
gnuplot_call_w_arguments = "gnuplot -e \""
gnuplot_datafile_command = "datafile="
gnuplot_output_png_command = "outputname="

#gnuplot fit commands
for single_input_data, output_image in
zip(data_files_gnuplot, output_png_files_gnuplot):
    sub.call([gnuplot_call_w_arguments +
              gnuplot_datafile_command + single_input_data+
              "; "+ gnuplot_output_png_command + output_image +
              "\" " + "fit_cubic.gplot"], shell = True)
```

As one can easily see, some useful strings are initialized in order to pass the correct commands to the fit script. In particular, we call gnuplot with the "$-c$" command that accepts a string with some variable declarations and initializations useful for the execution. For this specific case, we pass the filename where to extract data from and the filename where to plot the ".png" output. The fit is therefore performed for every input file specified in the Python program.

The GNUPLOT script that is run is very easy, and is the following for fitting:

```
cubic(x) = a*x**3 + b*x**2 + c*x + d
fit cubic(x) datafile using 1:2 via a,b,c,d
plot cubic(x) title 'Cubic fit',  datafile using 1:2 title
```

Moreover, it produces a plot in .png format for each single fit.

Results

We run the script via using the following bash command and keeping disabled the DEBUG mode:

```
python python_script.py 50 2000 50
```

We therefore expected a series of number evenly logarithmically spaced, between 50 and 2000. After some time, the first gnuplot script returns the figure 1), with the execution times for all the different matricial multiplications

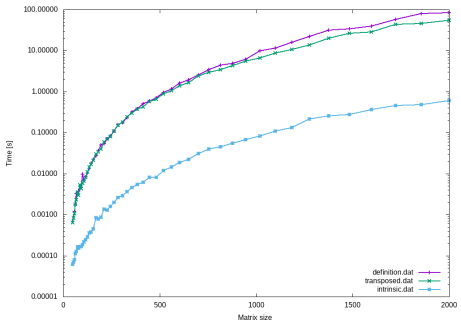
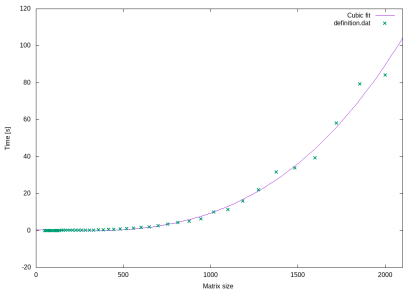


Figure 1: Plot of all different execution times for different kind of matricial product

Successively, the gnuplot script that performs the fit via a cubical function is run.

As an example, we show only one plot (fig. 2a) that is related to the data acquired using the definition of matricial product. In addition we show the output seen in terminal (fig. 2b), that is present in the file "fit.log" as well after the fit has been performed.



(a) Fitting cubical curve for the times spent by the program using the definition of matricial product.

```
After 13 iterations the fit converged.
final sum of squares of residuals : 0.00503778
rel. change during last iteration : -3.27125e-15

degrees of freedom    (FIT_NDF)                        : 46
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.010465
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.000109517

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                  = 1.05288e-10   +/- 1.069e-11    (10.16%)
b                  = -6.16119e-08  +/- 3.033e-08    (49.22%)
c                  = 3.4351e-05    +/- 2.284e-05    (66.48%)
d                  = -0.00322197   +/- 0.003817     (118.5%)

correlation matrix of the fit parameters:
                a      b      c      d
a               1.000
b              -0.984  1.000
c               0.905 -0.963  1.000
d              -0.664  0.743 -0.854  1.000
```

(b) Output for the fit with curve coefficients. These lines can be found in the "fit.log" file that gnuplot creates by default

Self Evaluation

In this homework we learnt how to automatize as much as possible some operations, by the mean of a single script to be run. Actually, this script might have been written in any language, but Python was actually the most handy one (moreover we were requested to do so).
Some useful further implementations for this program are, as an example, the faculty of exporting the coefficients found by the fitting script to another file. Moreover we could use other functions to fit our data with: actually we gave it a try by using a parabula, but the cubical curve gave better results, so we decided keep only the latter analysis and omit other curves in the report. Anyways, nothing prevents us to try for other curves and write more scripts.
Another improvement we may think of is to pass a single file, with multiple rows, to the Fortran program and store the outputs temporarily into an array that is exported into a file only at the end of the execution. However, this approach might be either too much difficult to implement or have less performance than what we have implemented so far, therefore is neglected.