Multi-run script and automated fits

Abstract

The first aim of this exercise is to implement the first week exercise on matrix multiplication defining the matrix dimension as an input to be read from a file and to run it using a Python script. Then, gnuplot will be used to plot the CPU times. The second aim is to fit the scaling of the time needed for the different multiplication methods as a function of the input size.

1 Theory

Matrix multiplication is a binary operation that produces a matrix from two matrices. If A is an $m \times n$ matrix and B is an $n \times p$ matrix, the matrix product C = AB is defined to be the $m \times p$ matrix such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}, \tag{1}$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, p$.

The simplest algorithm to compute this multiplication uses three loops over the indices i from 1 through m and j from 1 through p, computing Eq. (1) using a nested loop. The pseudo-code is the following.

```
Input: matrices A and B
Let C be a new matrix of the appropriate size
For i from 1 to n:
    For j from 1 to p:
        Set sum = 0
        For k from 1 to m:
            Set sum = sum + A_ik × B_kj
    Set C_ij = sum
Return C
```

This algorithm takes time $\mathcal{O}(nmp)$, but if we deal with square matrices of size $N \times N$ the running time is $\mathcal{O}(N^3)$.

2 Code development

This exercise requires to write a lot of code, so it is not convenient to show it entirely in this report. Having said that, we explain which is the philosophy of the code and we report only the most crucial parts.

2.1 Multi-run script

In this first part we want to implement the Fortran code for matrix multiplication defining the matrix dimension N as an input to be read from a file (in this exercise the matrices are assumed to be square). Then this code will be run by a Python script, which will generate different values of N and write them in the input file for the Fortran code.

In the Fortran code, the matrix dimension N is loaded by means of a subroutine `LoadDimensions`, which is contained in the `mult` module.

```
    subroutine LoadDimensions(filename, dimensions)
        ! this subroutine is used to load the dimension of the two
        ! matrices from a txt files
        implicit none
        ! name of the file
        character(*), intent(in) :: filename
        ! variable to store the dimension
        integer :: dimensions
        ! value for the iostat specifier
        integer :: iostat
        open(unit = 10, file = filename, status = "old")
        ii = 1
        do while(ii < 2)
            read(10, *, iostat=iostat) dimensions
            if(iostat < 0) then
                write(6, '(A)') "Warning: the file containts no entries."
                exit
            else if(iostat > 0) then
                write(6, '(A)') 'Error reading file.'
                stop
            end if
            ii = ii + 1
        end do
        close(10)
    end subroutine LoadDimensions
```

Notice that when we open the file, the status is set to `"old"`, so the text file must already exist. The file reading is handled through the `iostat` specifier:

- if it is zero the `read` was executed flawlessly;
- if it is negative this means that the end of the input has reached;
- if it is positive the `read` has encountered some problem.

Notice also that this subroutine can be easily adapted to load the dimension of rectangular matrices: it is sufficient to set `dimensions` to be a `integer, dimension(4)` and extend the while to `ii < 5`.

The Python script asks the user to enter the minimum and the maximum dimension of the matrices, `N_min` and `N_max`, using a `try/except` block to ensure to get correct values (that is both integers and $N_{max} > N_{min}$).

```
    N_min = 0
    while (N_min <= 0):
        try:
            N_min = int(input("Please enter the minimum dimension N_min: "))
        except ValueError:
            print("Error! The input must be an integer.")
    N_max = 0
    while (N_max <= 0 or N_max < N_min):
        try:
            N_max = int(input("Please enter the maximum dimension N_max: "))
        except ValueError:
            print("Error! The input must be an integer.")
        if N_max < N_min:
            print("Error! N_max must be greater than N_min.")
```

From the interval $[N_{min}, N_{max}]$ we take fifty values and we use the Python library `subprocess` to run the Fortran code for each of these values, creating the corresponding input file each time. Then, we also run the gnuplot script to plot the CPU time as a function of N.

```
    # compile the Fortran code
    subprocess.call(["gfortran", fileSource, "-o", fileExec, "-O3"])
    # store the dimensions in the 'dimensionsFile' and launch the Fortran program
    for d in dimensions:
        with open(dimensionsFile, "w+") as inputfile:
            print("Computing for N =", str(d))
            inputfile.write(str(d) + '\n')
        subprocess.run("./" + fileExec)
    ...
    # execute the gnuplot script
    plotFile = "ex04plot.gnu"
    subprocess.call(["gnuplot", plotFile])
```

Notice that the Fortran code is compiled using the option `-O3`, which – according to the results of the first week exercise – leads to the best optimization.

2.2 Automated fits

In this second part we write a Python script that – by running a gnuplot script – will automatically fit the CPU time needed for the different multiplication methods as a function of the input matrix size. The gnuplot script is the following.

```
    set terminal png size 1024, 768 font "Verdana, 25"
    set output sprintf("%s%s%s", "CPU_time_fit_", name, ".png")
    set title "Matrix multiplication - CPU times" font "Verdana, 27"
    set xlabel "Matrix dimension"
    set ylabel "CPU time (s)"
    set grid
    set logscale y
    set key bottom right box height 1.7
    set format y '%2.0t*10^{%T}'
    fileLog = name."_fit.log"
    set fit logfile fileLog
    a = 1e-9
    b = 3
    f(x)= a*x**b
    fit f(x) filename using 1:2 via a, b
    plot f(x) title 'f(x)=ax^b' lw 2, filename using 1:2 title name with
    linespoints lw 2
```

Notice that the two variables `name` and `filename` must be given as input with the `-e` option when we run the script, in the following way.

```
    $ gnuplot -e "filename='foo.txt'" -e "name='foo'" ex04fit.gnu
```

where `foo.txt` is the file containing the data and `foo` is an identifying name for the dataset that will be used to write the file name (`CPU_time_fit_foo.png`). This command is implemented in the Python script defining a list of strings corresponding to the various keywords for each plot we want to do. The run is done through the `subprocess.Popen` class, which is more flexible than the usual `subprocess.run` function (in this case the problem is related to the large presence of inverted commas).

```
    # gnuplot script
    plotFile = "ex04fit.gnu"
    # write the command for each file
    command_not_opt = ["gnuplot", "-e", "filename='not-optimized.txt'", "-e",
                       "name='not-optimized'", plotFile]
    command_opt = ["gnuplot", "-e", "filename='optimized.txt'", "-e",
                   "name='optimized'", plotFile]
    command_matmul = ["gnuplot", "-e", "filename='matmul.txt'", "-e",
```

```
                    "name='matmul'", plotFile]
    commands = [command_not_opt, command_opt, command_matmul]
    # run the commands
    for i in commands:
        subprocess.Popen(i)
```

3 Results

The plot of the CPU time as a function of the input matrix dimension is shown in Figure 1.

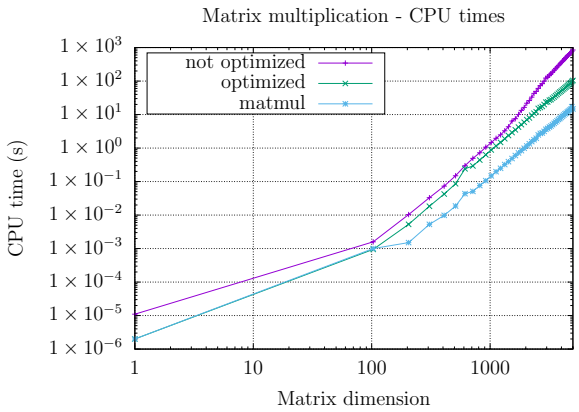
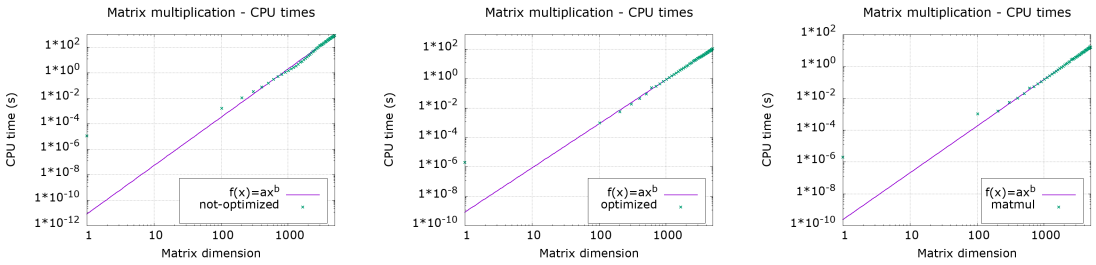


Figure 1: CPU time vs. matrix dimension for the three different matrix multiplication methods (log-log scale).

As we can see, the three functions are almost equivalent until $N = 100$. For larger values of N, we see that the intrinsic `matmul` function performs way better – approximately one order of magnitude – then the two defined by the programmer.
Regarding the two non-intrinsic functions `mult1` (non-optimized) and `mult2` (optimized) it turns out that, as we expected, the optimized one is faster.

Moving on to the fits, the plots obtained by running the Python script for the automated fits are shown in Figure 2.



(a) Non-optimized function (`mult1`). (b) Optimized function (`mult2`). (c) Intrinsic function (`matmul`).

Figure 2: CPU time vs. matrix dimension fit with the function $f(x) = ax^b$ for the three different matrix multiplication methods (log-log scale). These graphs are obtained using the automated fit Python script.

The function which is used to fit the data is

$$f(x) = ax^b, \tag{2}$$

where a and b are the parameters to determine. In particular, we are interested in b, which represents the scaling exponent of the CPU time as a function of the dimension N. If we inspect the log files generated by gnuplot, we find that

$$b_{non-opt} = 3.79 \pm 0.02, \qquad b_{opt} = 3.00 \pm 0.02, \qquad b_{matmul} = 2.93 \pm 0.05.$$

These results are in line with what we know from the theory, i.e. that the matrix-matrix multiplications scales as $\mathcal{O}(N^3)$. As we expect, the scaling for the non-optimized function is worse than the one for the other two. This confirms the fact that by exploiting the way in which Fortran stores matrices we get better performances.

For the sake of completeness, in Figure 3 we show the three fits in linear scale in a single plot.

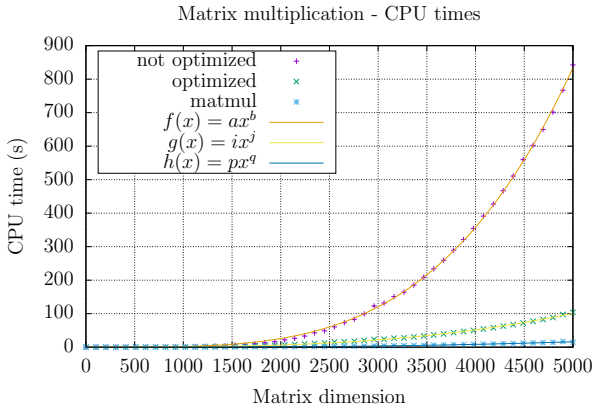


Figure 3: CPU time vs. matrix dimension fit with the function $f(x) = ax^b$ for the three different matrix multiplication methods.

4 Self evaluation

This exercise was very instructive since it made me learn how to run a program written in another language using Python by means of the `subprocess` package and how to use operating system dependent functionalities with the `os` package.

Moreover, I expanded my knowledge of gnuplot. In particular, I learned how to use the `-e` option to enter "dynamically" the name of the file which contains the data to fit and plot and a label.