Exercise 1

Umberto Maria Tomasini
Quantum Information Course

8 October 2018

Ex.1.2: Number precision

Testing the limits of INTEGER and REAL in Fortran.

```fortran
program int_precision
implicit none
integer*2 x,y
integer*4 z,w
real*4 a,b
real*8 c,d
x=2*(10**6)
z=2*(10**6)
y=1
w=1
a=acos(-1.0_4)*(10**32)
c=acos(-1.0_8)*(1.0d+32)
b=sqrt(2.0_4)*(10**21)
d=sqrt(2.0_8)*(1.0d+21)
x=x+y
z=z+w
a=a+b
c=c+d
print *, "The␣sum␣with␣integer*2␣is:␣", x
print *, "With␣integer*4␣is␣", z
print *, "The␣sum␣with␣real*4␣is:␣", a
print *, "With␣real*8␣is␣", c
stop
end program int_precision
```

```
umt@umt-VirtualBox:~/Desktop/QI$ gfortran ex1_2.f -fno-range-check
umt@umt-VirtualBox:~/Desktop/QI$ ./a.out
 The sum with integer*2 is:  -31615
 With integer*4 is:      2000001
 The sum with real*4 is:    3.14159278E+32
 With real*8 is:   3.1415926536039354E+032
```

The limits of INTEGER*2 are $[-2^{15}, 2^{15}-1]$. The number 2000000 exceeds those limits. As a consequence, the sum provides an error. The limits of INTEGER*4 are $[-2^{31}, 2^{31}-1]$. The number 2000000 is within those limits, so everything works.
Speaking of the REAL type, both $\pi \cdot 10^{32}$ and $\sqrt{2} \cdot 10^{21}$ stay within the limits for both precisions. As we can see frome the picture above, single precision has 8 digits of precision, while single precision has 16 digits of precision. When the sum was done in REAL*4 type, also the costants were defined in the REAL*4 type (see acos(-1.0_4)). The same applies for REAL*8 type (see acos(-1.0_8) and also 1.0d was used instead of 10** to raise a power).

As a result, the sum done with double precision is more close to reality respect to the other. In fact, in reality $\pi = 3.1415926535897932$. The result shown for REAL*8 differs from this last value in the tenth figure. This is reasonable: we are adding a number which should affect at the eleventh figure (we are adding a number of the order of 10^{21} to a number of the order of 10^{34}), but because of the carry-over the figure which is affected is the tenth.

Finally, we tried to see what happens summing two numbers of a certain type in a resulting number of a more precise type.
Summing 2000000 and 1 in two variables which were INTEGER*2 and giving the result of their sum in a INTEGER*4 variable, the result was still an error: -31615. Similarly, we have done the same thing for REAL*4 and REAL*8 types (always with $\pi \cdot 10^{34}$ and $\sqrt{2} \cdot 10^{21}$), obtaining a result in double precision which is very similar in the first eight figures to the one in single precision. By contrast, the remaining eight figures are totally different to the other number in double precision (the one obtained via the sum of two numbers saved in double precision). This suggests that those eight last figures are worthless. We can see the results printed in the figure below.

```
umt@umt-VirtualBox:~/Desktop/QI$ ./a.out
      -31615
   3.1415927798155553E+032
```

Ex.1.3: Test Performance

The following code does a matrix-matrix multiplication following three different methods. The first two rely on an explicit code, written differently (the difference is in the order of the loops: one is column by column and the other one is row bu row). The third way uses the already made Fortran function *matmul*. The first matrix is such that in the entry (i,j) there is the value i+j, whilst the second has the value i*j in the entry (i,j) (it was chosen to create integer matrices by means of a simple rule, in order to focus only on test perfomance). For sake of simplicity, the matrices are squared.

```
program test_performance
implicit none
integer*2, dimension(4,4) :: m1
integer*2, dimension(4,4) :: m2
integer*2, dimension(4,4) :: mris1, mris2, mris3
integer ii, jj, kk, nn
nn=4

c    inserting values in m1: in the entry (i,j)
c        the value i+j is assigned
do ii=1,nn
do jj=1,nn
```

```
m1(ii,jj)=ii+jj
end do
end do

c     inserting values in m2, in the entry (i,j)
c     the value i*j is assigned
do ii=1,nn
do jj=1,nn
m2(ii,jj)=ii*jj
end do
end do


c     matrix1-matrix2 multiplication,
c     first mode: column by column
do ii=1,nn
do jj=1,nn
do kk=1,nn
mris1(ii,jj)=mris1(ii,jj)+m1(ii,kk)*m2(kk,jj)
end do
end do
end do

c     matrix1-matrix2 multiplication,
c     second mode: row by row
do ii=1,nn
do jj=1,nn
do kk=1,nn
mris2(jj,ii)=mris2(jj,ii)+m1(jj,kk)*m2(kk,ii)
end do
end do
end do

c     matrix1-matrix2 multiplication,
c     using the function
mris3=matmul(m1,m2)



c     printing m1
print *, "Matrix␣1"
do ii=1,nn
print*, (m1(ii, jj), jj = 1, nn)
end do
```

```
c    printing m2
print *, "Matrix␣2"
do ii=1,nn
print*, (m2(ii, jj), jj = 1, nn)
end do


c    printing mres1
print *, "Resultant␣matrix␣1"
do ii=1,nn
print*, (mris1(ii, jj), jj = 1, nn)
end do

c    printing mres2
print *, "Resultant␣matrix␣2"
do ii=1,nn
print*, (mris2(ii, jj), jj = 1, nn)
end do

c    printing mres3
print *, "Resultant␣matrix␣3,␣with␣the␣function"
do ii=1,nn
print*, (mris3(ii, jj), jj = 1, nn)
end do


stop
end program test_performance
```

```
umt@umt-VirtualBox:~/Desktop/QI$ ./a.out
 Matrix 1
           2           3           4           5
           3           4           5           6
           4           5           6           7
           5           6           7           8
 Matrix 2
           1           2           3           4
           2           4           6           8
           3           6           9          12
           4           8          12          16
 Resultant matrix 1
          40          80         120         160
          50         100         150         200
          60         120         180         240
          70         140         210         280
 Resultant matrix 2
          40          80         120         160
          50         100         150         200
          60         120         180         240
          70         140         210         280
 Resultant matrix 3, with the function
          40          80         120         160
          50         100         150         200
          60         120         180         240
          70         140         210         280
```

Code performance for different matrix's size

Calling the function *cpu_time* at the beginning and at the end of a part of a code, we can infer the time in seconds spent by the CPU on esecuting that part. This procedure was executed three times, one for each way to compute the matrix-matrix multiplication (cutting the part dedicated to printing the matrices, in order to avoid the ineffecient printing of huge matrices). This was done for different size of the matrices (encoded by the variable nn).

```
program test_performance_cpu
implicit none
integer*2, dimension(4,4) :: m1
integer*2, dimension(4,4) :: m2
integer*2, dimension(4,4) :: mris1, mris2, mris3
integer ii, jj, kk, nn
real :: finish1, start1, finish2, start2
real :: finish3, start3
nn=4

c     inserting values in m1: in the entry (i,j)
c         the value i+j is assigned
do ii=1,nn
do jj=1,nn
m1(ii,jj)=ii+jj
end do
```

```
end do

c    inserting values in m2, in the entry (i,j)
c    the value i*j is assigned
do ii=1,nn
do jj=1,nn
m2(ii,jj)=ii*jj
end do
end do


c    matrix1-matrix2 multiplication,
c    first mode: column by column
call cpu_time (start1)
do ii=1,nn
do jj=1,nn
do kk=1,nn
mris1(ii,jj)=mris1(ii,jj)+m1(ii,kk)*m2(kk,jj)
end do
end do
end do
call cpu_time (finish1)
print *, "Time␣in␣seconds=␣", finish1-start1

c    matrix1-matrix2 multiplication,
c    second mode: row by row
call cpu_time (start2)
do ii=1,nn
do jj=1,nn
do kk=1,nn
mris2(jj,ii)=mris2(jj,ii)+m1(jj,kk)*m2(kk,ii)
end do
end do
end do
        call cpu_time (finish2)
print *, "Time␣in␣seconds=␣", finish2-start2

c    matrix1-matrix2 multiplication,
c    using the function
call cpu_time (start3)
mris3=matmul(m1,m2)
        call cpu_time (finish3)
print *, "Time␣in␣seconds=␣", finish3-start3

stop
```

```
end program test_performance_cpu
```

Method	Size	Time [10^{-5} s]
column	4×4	0.200002
row	4×4	0.100001
matmul	4×4	0.899995
column	8×8	0.500004
row	8×8	0.400003
matmul	8×8	1.199998
column	15×15	1.800002
row	15×15	1.800002
matmul	15×15	1.199998
column	100×100	$7.796000 \cdot 10^2$
row	100×100	$7.719001 \cdot 10^2$
matmul	100×100	$4.599988 \cdot 10^1$

Method	Size	Time [s]
column	1000×1000	6.725492
row	1000×1000	7.035482
matmul	1000×1000	0.496622
column	2000×2000	$1.166687 \cdot 10^2$
row	2000×2000	$1.233601 \cdot 10^2$
matmul	2000×2000	3.958939

As we can see from the table, the execution time rises sharply increasing the size. Theorically, the computational time spent on a matrix calculation goes like n^3, due to the fact that for each one of the n^2 entries of the resulting matrix, n multiplications have to be computed. In order to verify that, we should do a cubic fit ($ax^3 + bx^2 + cx + d$) on the data. Another test could be considering only very high numbers, in order to discard the smallest terms. This last method needs a lot of time. Just for 10000×10000 matrices the execution took so long that it had to be terminated.

We can roughly try to do that for numbers which have an acceptable execution time, such as 1000 and 2000 expecting some errors. We can solve the below sistem of equations, with $x_1 = 1000$ and $x_2 = 2000$. y_1 and y_2 are the time of execution, which are different acording to the method.

$$\begin{aligned} ax_1^3 + d = y_1 \\ ax_2^3 + d = y_2 \end{aligned} \tag{1}$$

We will find three different couples (a,d) for three different methods.

Method	a [s]	d [s]
column	$1.5706 \cdot 10^{-8}$	-8.9805
row	$1.6618 \cdot 10^{-8}$	-9.5825
matmul	$4.94617 \cdot 10^{-10}$	0.002

Then we try to infer the execution time for the 1500×1500 case. In the table below we compare the results with the true ones.

Method	True [s]	Projected [s]
column	$3.644118 \cdot 10^1$	$4.402725 \cdot 10^1$
row	$4.627720 \cdot 10^1$	$4.650325 \cdot 10^1$
matmul	1.755074	1.671332

We can notice that the projected results are of the same order of the true results (in two cases are very similar). This sugegst that the compilation time is really of the order of n^3.

Another fact worth noticing is that the function *matmul* provides the lowest execution time. This is not true very small sizes of the matrices, whilst it is true for bigger sizes. It is also noticeable that the column by column method have a minor execution time than the row by row method. This difference becomes clear for bigger sizes.

Optimization flags

Fixing the size at 1000×1000, we have applied some optimization flags on the execution. The table below provides a list of possible optimization flags and some information about the consequences of their use on execution time, code size, memory usage and compile time.

option	optimization level	execution time	code size	memory usage	compile time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

+increase ++increase more +++increase even more -reduce --reduce more ---reduce even more

Method	Flag	Time [s]
column	none	6.725492
row	none	7.035482
matmul	none	0.496622
column	-O0	6.863408
row	-O0	7.138404
matmul	-O0	0.502725
column	-O1	1.264411
row	-O1	1.303978
matmul	-O1	0.593714
column	-O	1.311188
row	-O	1.319187
matmul	-O	0.492852
column	-O2	1.268427
row	-O2	1.283581
matmul	-O2	0.497744
column	-O3	1.441108
row	-O3	0.265098
matmul	-O3	0.500692
column	-Os	1.747055
row	-Os	1.736168
matmul	-Os	0.525609
column	-Ofast	1.440345
row	-Ofast	0.241460
matmul	-Ofast	0.495933

As it is clear from the table, the use of optimization flag reduced the execution time in all cases, except for the flag -O0. Due to the fact that the program is not extremely long, the compile time was not sensibly affected. The same applies on the code size and on the memory usage.