# Assignment 1

## Quantum Information & Computing

**Francesco Barone**
University of Padua, Department of Physics

**October 28, 2022**

# EXR 1 // warm-up: 4th order central difference

In this warm-up exercise I wish to code
a **finite difference method** to compute the derivative of a 1-D array $y_i$:

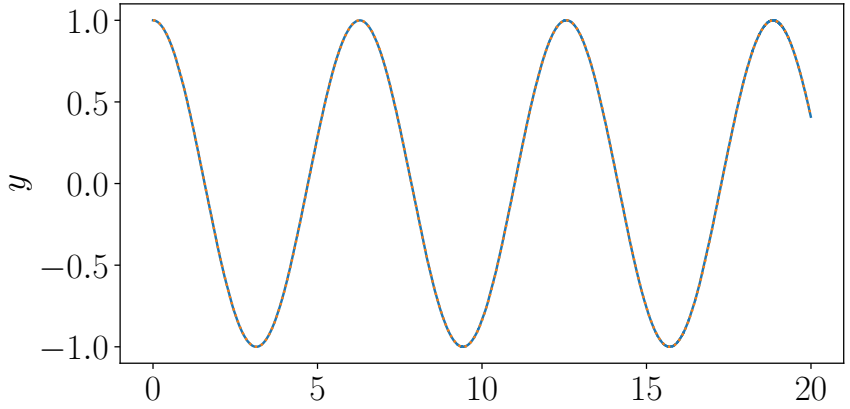
$$y'_i = \frac{y_{i-2} - 8y_{i-1} + 8y_{i+1} - y_{i+2}}{12 \cdot h} + O(h^4)$$

This method is known as **4th order central difference**.

Implementing the formula is actually really easy, and it can be done with **array slicing**.

Kreider

```
! compute 4th order central diff.
dy = 0 + y(1:nn-4)
dy = dy - 8*y(2:nn-3)
dy = dy + 8*y(4:nn-1)
dy = dy - y(5:nn)
dy = dy/(12*dx)
```




To test the formula, I've initialized $y_i$ as a `real` array with $nn = 20000$ discrete values $y_i = sin(x_i)$, computed with resolution $h = x_i - x_{i-1} = 0.001$.

The derivative is, trivially, a cosine function.

# EXR 2A // Integer overflow

**Code**

```
integer*2 :: big_int2 = 2000000
integer*2 :: res_int2
integer*4 :: big_int4 = 2000000
integer*4 :: res_int4

res_int2 = big_int2 + 1
print *, " sum with int*2 =", res_int2
res_int4 = big_int4 + 1
print *, " sum with int*4 =", res_int4
```

**Output**

```
baronefr@cayde:~/hw1$ ./exr2a.x
 twomillion increment test -------
  sum with int*2 = -31615
  sum with int*4 = 2000001

 [i] direct overflow test ------
  int*2 :  32767 + 1  !=  -32768
  int*4 :  2147483647 + 1  !=  -2147483648
```

The **signed 2-complement** representation of 2000000 is

$$(2000000)_{10} = (00000000\ \ 00011110\ \ 10000100\ \ 10000000)_2$$

However, if you try to store it as a `integer*2` **only the least significant bits will be memorized**, i.e.

$$(10000100\ \ 10000000)_2 = (-31616)_{10}\ \ .$$

Eventually, the sum of such value to one returns $-31615$, as printed in stdout.
To fix this problem we need to switch the variable type to `integer*4`.

# EXR 2B // Real precision

**Code**

```fortran
real              :: x1_single, x2_single, rs
double precision :: x1_double, x2_double, rd

! real
x1_single = 4.0*datan(1.0d0)*(10d0**32)
x2_single = sqrt(real(2d0))*(10d0**21)
rs = x1_single*x2_single

print *, "[ real ]", x1_single, x2_single
print *, "     ->  ans: ", rs

! double
x1_double = 4.0d0*datan(1.0d0)*(10d0**32)
x2_double = sqrt(real(2d0))*(10d0**21)
rd = x1_double*x2_double

print *, "[double]", x1_double, x2_double
print *, "     ->  ans: ", rd
```

**Output**

```
baronefr@cayde:~/hw1$ ./exr2b.x
 [ real ]  3.14159259E+32   1.41421360E+21
      ->  ans:  Infinity
 [double]  3.1415...E+032   1.4142...E+021
      ->  ans:  4.4428828621216639E+053
```

In this exercise, we **multiply two big real values**.

The issue is that **such operation exceeds the limits** of `real` type, being the result bigger than the upper limit of $\sim 3.40 \cdot 10^{38}$.

However, FORTRAN handles the result properly, as the return value of the operation is `Infinity`. Further operations with such value would return other Infinity values, making the problem more likely to be detected.

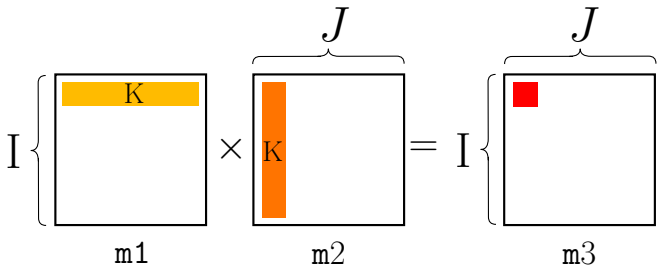
The use of `double` type, which is able to store values up to $\sim 1.79 \cdot 10^{308}$, fixes this problem easily.

# EXR 3 // Matrix product

We wish to benchmark the CPU time of a matrix product in Fortran, comparing **nested loops** with the built-in `matmul()` routine. The "row by columns rule" can be implemented with loops in two different ways, **depending on the variables used in the two external loops**.

**Remark** about the following CPU time benchmarks:

- I considered products between square matrices $N \times N$;
  - for $N = 100, 200, ..., 2000$ I took the time average of 3 measures, to make the values more stable against random fluctuations
  - for $N = 3000, 4000, ..., 10^4$ I run the benchmark only once, as the measured time is usually well above 60s of CPU time
- I've repeated the measures changing the compiler optimization flag `-O` to [0, 1, 2, 3, 5].
- The Fortran executable takes as first argument $N$. There is a **Bash script** that compiles the executable with the correct flags and accounts for all the benchmarks.




**RbC (rows by column)**

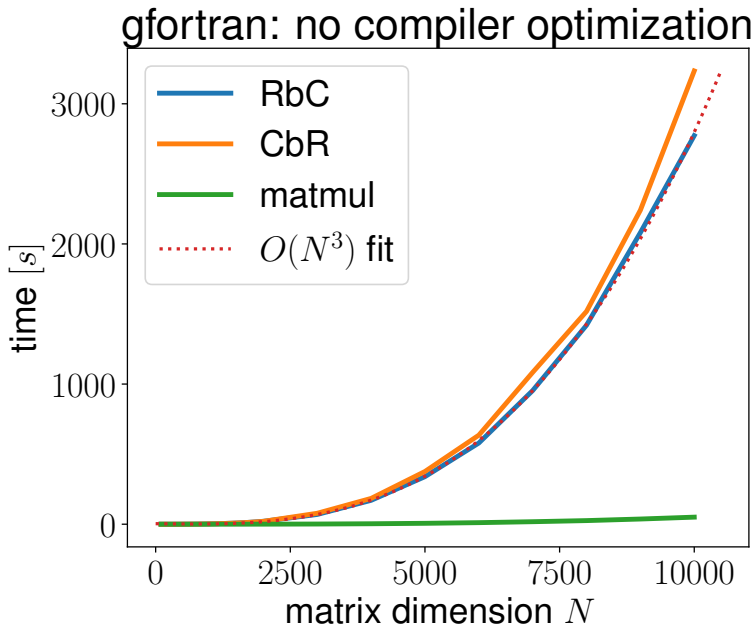
```
! m3 initialized to zero
do ii=1,I
  do jj=1,J
    do kk=1,K
      m3(ii,jj)=m3(ii,jj)+m1(ii,kk)*m2(kk,jj)
end do end do end do ! sorry for this
```

**CbR (column by rows)**

```
do jj=1,J
  do ii=1,I
    do kk=1,K
      m3(ii,jj)=m3(ii,jj)+m1(ii,kk)*m2(kk,jj)
end do end do end do ! sorry for this
```

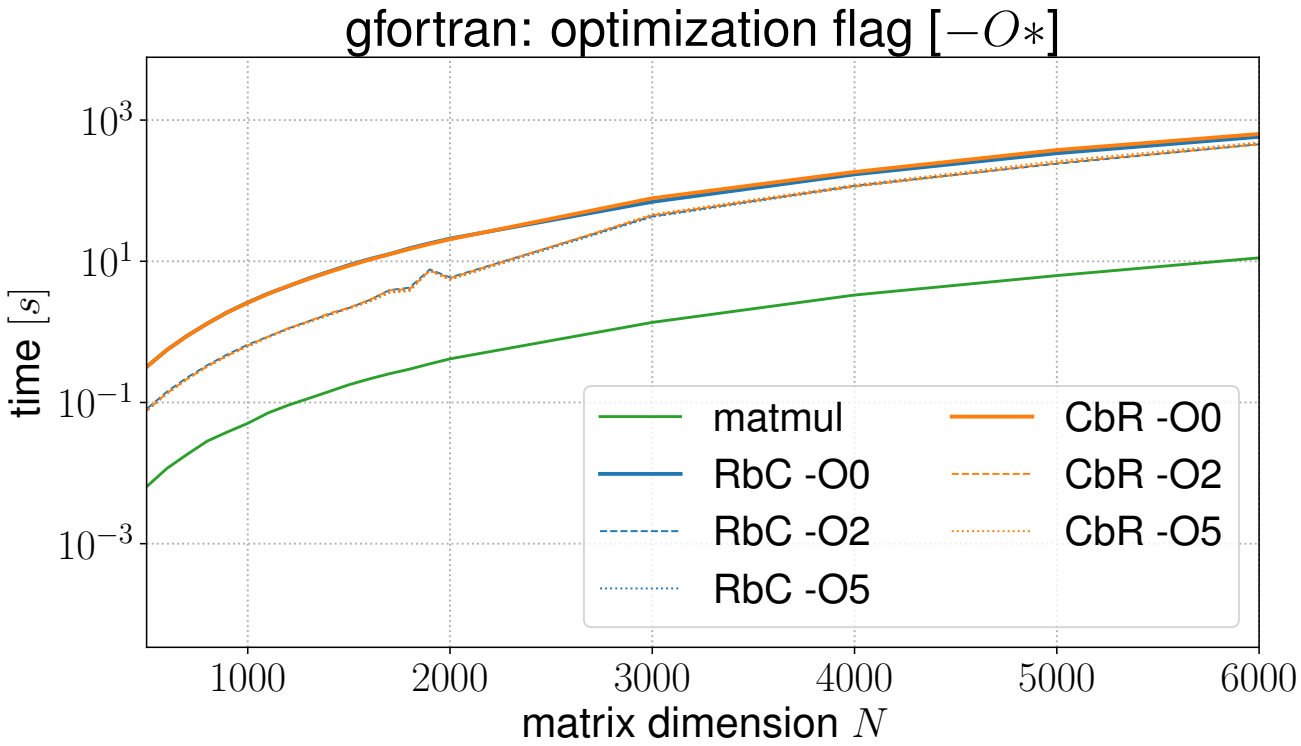
Let us call **RbC** the loop over *Rows and Columns*. Similarly, we call **CbR** the loop over *Columns and Rows*, which is a simple swap of loops over `ii` and `jj`.

# EXR 3 // Matrix product




⚠ We know that the computational complexity of the Loop algorithm scales as $O(N^3)$. The benchmarks are consistent with such statement.

Unluckily, but not surprisingly, even **with compiler optimization** both the permutations `RbC` and `CbR` **are really slow** compared to an optimized routine, such as `matmul()`.




The plot shows that the benchmark using gfortran's optimization flag O2 overlaps with flag O5, hinting that the **compiler optimization is not able to push further down the CPU time**.

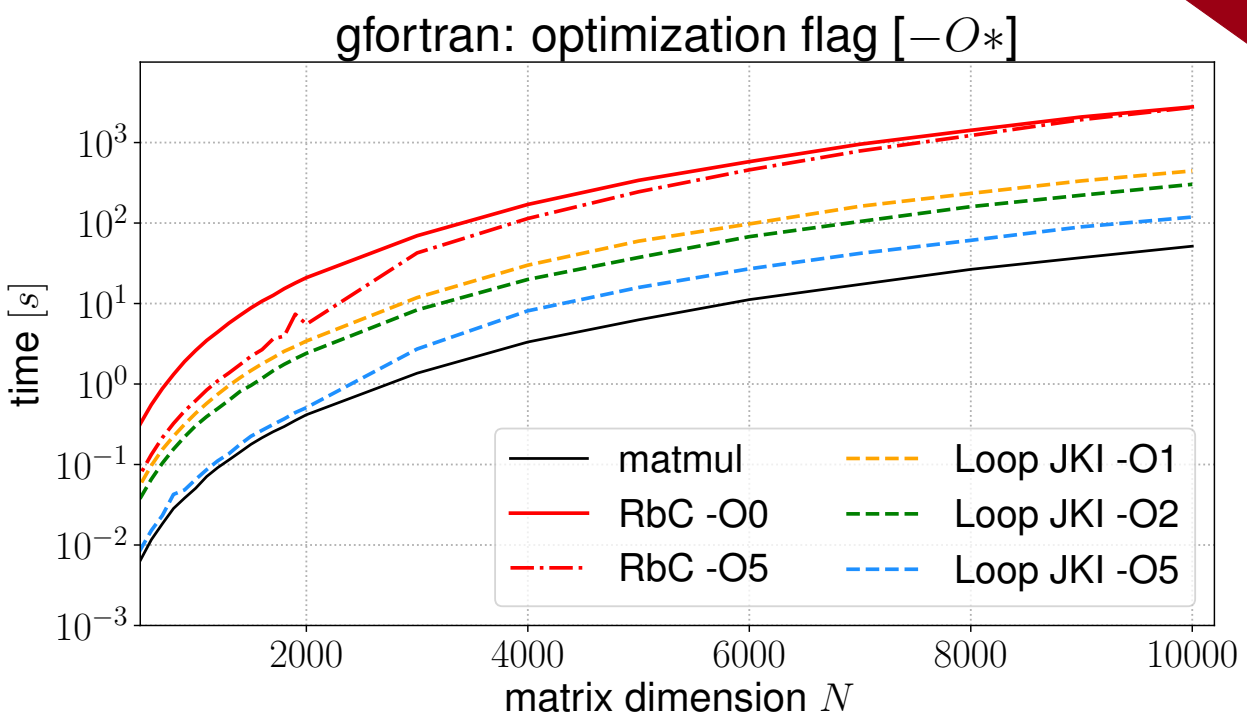
# EXR3 // Matrix product - the better way

However, **there exists a permutation for which gfortran is able to do a much better optimization**:

**Loop JKI**

```
do jj=1,J
  do kk=1,K
    do ii=1,I
      m3(ii,jj) = m3(ii,jj) + m1(ii,kk)*m2(kk,jj)
end do end do end do ! sorry for this
```

As a result, **higher optimization flags on `Loop JKI` are able to push down the CPU time** by orders of magnitude, reaching performances in the same order of `matmul()`.




This new layout of the loops allows the code to be better executed with **vector instructions**.
To confirm the use of vector instructions, I have dumped the **optimized tree** from the compiler (use the flag `-fdump-tree-optimized`).

**Tree dump for -O5 flag**

```
<bb 22> [local count: 44957436]:
...
vect__222.119_435 = MEM <vector(4) real(kind=4)>
    [(real(kind=4) *)_567 + ivtmp.176_401 * 1];
vect__223.120_436 = vect_cst__411 * vect__222.119_435;
vect__224.121_437 = vect__95.111_400 + vect__223.120_436;
```