Introduction to HPC — Assingment #1

September 11, 2025

Assignment #1

Total: **10 points**

Due date: **September 26th**

*This assignment must be completed **individually**.*

I.e. you should work alone without any help or collaboration from others.

Please provide detailed answers to the following questions.

Please READ carefully!

- Solutions must include proper modular implementations, best practices in software development, comments and documentation.
- You must also **profile** and indicate the most "costly" part(s) in each problem ($\approx 20\%$ of each problem).
- **Your solutions (code) MUST compile in the lab.machines or Teach cluster!** Code that does NOT compile will get a 0!
-

Problems 1–4 [each problem will account for 2 points each]

Solve the problems listed below which are the set of exercises from the first part from our draft for *"Practical Scientific Computing" (Van Zon, Ponce)* available on Quercus.

Solutions must include proper **modular** implementations, best practices in software development, comments and documentation.

Coding Exercises for Scientific Computing Best Practices

In this part, we list the assignments given in the course throughout the years, which are a good way to exercise the material presented in the text. We've also included a number of further exercises.

General Remarks A few suggestions that are recommended, in particular with respect to code development and coding style:

- Try to avoid duplicating the specific values when implementing your code (i.e., -5, 5 or any other quantities you might end up using repeatedly). E.g. storing them, once, in a variable, and use that variable; this will result in your code to have wider applicability.
- Give your variables understandable and meaningful (ie. representative) names.
- Do NOT use global variables!
- Indent your code properly. Mostly, that means that you should indent consistently and in such a way that all code in the same block has the same indentation.
- Comment and document your code!
- In general: try and write a code that your future self could still read and understand it in, say, 1 year.
- Put your code under Version Control and create a repository to host it.

Ex.1 Moving average

The goal of this is assignment is to write a C++ program that computes the *moving average* of a given vector.

Ex.1.1 Part I

The program should accept a command line number which will indicate how many points are going to be considered in the vector. The program will generate a table with three columns, one column with the original values of the vector defined as indicated below, the second and third columns will be the computation of the moving average using a window of 5 and 10 elements respectively.

Moving Average definition: in its simplest interpretation the moving average of a vector of N elements considering a window of m elements, is given by:

$$\langle y_i \rangle = \frac{1}{m} \sum_{j=i}^{i+m-1} y_j \tag{7.1}$$

Notice that for each element of the vector, you will have a "moving average" value associated. Additionally, one should implement special considerations when dealing with the "latest" elements of the vector, i.e. the points where $i > N - m$.

In particular, your program should:

- Contain at least two functions (besides `int main`), one that generates the actual values of the vector y. And a second function which should implement the computation of the moving average.
- For generating the vector y, consider generating a vector x taken from N evenly spaced values from the interval $[-5, 5]$ (including the end points), where N is given by the command line argument passed to the program; the vector y then is defined as the cosine of the elements from the vector x.
- It should write out the results in tabular form (i.e. three-column ASCII text) to the screen. Don't try to add lines to the table, just use spaces and newlines to format the text.

Compile and run your code using `g++` with the `-std=c++14` flag. Capture the output for the case $N = 100$, for instance if you named your code `movingAverage`, you can run it in this way

```
$ ./movingAverage 100 > averages.txt
```

this should generate a text file named "averages.txt" containing a vector with 100 elements and 3 columns, for the vector y, and the moving averages for windows of 5 and 10 elements respectively. Inspect these values by plotting them and comparing to what you would expect from the standard average computation.

Ex.1.2 Part II

Depending on the strategy you used for the first part of this exercise you can have implemented the computation in different ways.

For instance, if we consider the definition previously given for the moving average, see Eq.(7.1), this definition assumes a **forward** consideration of the elements, i.e. it goes from $j = i$ up to $j = i + m - 1$.

One could have defined this as a **backwards** or **centered** implementation, where in each respective case the range of j will go from $i - (m - 1)$ up to i or $i - (m - 1)/2$ up to $i + (m - 1)/2$ respectively.

Taking this into consideration, add an additional argument to your moving average function indicating which type of moving average computation (forward, backward or centered) you want to compute and implement this particular feature.

Another element that was left open to interpretation was how to deal with the final elements of the vector, i.e. when the index i was larger than $N - m$, when N is the size of the vector and m the length of the window where to compute the moving average.

Consider now adding another additional argument to the moving average function, where depending on the following values, is how the computation will be done:

- if the argument is **restricted**, the elements considered when reaching the element $N - m$ will shrink by one, so that the window is reducing so that it does not go beyond the last element of the array, i.e. when $j = N$ then $m = 1$.

- if the argument is **cyclic**, then the elements of the beginning of the array will be used to complete the m elements in the window. I.e. if $j = N - m + 2$ then the element $j = 0$ and 1 should be considered in the average.

- if the argument is **padding**, then additional elements with the same value as the last element, will be added to the array so the computation can be continued with them.

Ex.2 Lissajous

The goal of this exercise is to write a C++ program that generates a table of data as indicated below.

This table should have three columns, one column with x double precision values ranging from -5 to 5, a second column with values $y = \sin(2x)$, and a third column with values $z = cos(3x)$.

In particular, your program should:

1. Contain at least two functions (besides "`int main`"), called "f" and "g". The function "f" should take x as an argument and return the value of $\sin(2x)$, while the function "g" should that take x as an argument and return $\cos(3x)$.

2. The program should take 101 evenly spaced x-values from the interval $[-5, 5]$ (including the end points).

3. It should use the functions f and g to compute the values of those functions for those 100 x-values.

4. It should write out the x, f, and g values in tabular form (i.e. three-column ascii text), either to a file called "lissajous.txt". Don't try to add lines to the table, just use spaces and newlines to format the text.

Compile and run your code using `g++` with the `-std=c++14` flag, and capture the output. Inspect the output by plotting the columns, in particular, g vs f; you should be able to recognize the so-called Lissajous shapes.

Ex.3 Arrays

Write a program which:

- Dynamically allocates an array of 100,000 integers.
- Fills the array such that element i has the value $9 * \sin(i)$, rounded down to an integer.
- Counts the number of times the value -9 occurs in the array, how often -8 occurs, -7, -6, -5, -4, -3, -2, -1 ,0, 1, 2, 3, 4, 5, 6, 7, 8, 9. (I.e., for each value separately, how often does it occur in the array?)
- Print these values and counts out to screen.
- Deallocate any allocated memory
- Returns 0 (why?)

Problem #5: *Numerical Linear Algebra* [2 points]

For the following problems, you can consider using basic C/C++ array structures or take a look at *template* library called `rarray` (https://github.com/vanzonr/rarray).

a) Implement a program that numericaly calculates a *matrix-matrix multiplication*. For starting you can consider an integer 100×100 matrix filled with arbitrary values.
Profile the program and determine the most expensive (slowest) part of the code.
What would happen if the the matrices were $1,000,000 \times 1,000,000$ elements-long?

b) Eigenvalues (and their eigenvectors) are fundamental properties of the matrix and play a role in many linear algebra applications. In case your linear algebra is a little rusty, eigenvalues of a matrix are those special values λ for which one can find a vector x that satisfies $Ax = \lambda x$. There are at most n eigenvalues if A is an $n \times n$ matrix. A vector x_i that satisfies the relation $Ax_i = \lambda_i x_i$ is called an eigenvector corresponding to the eigenvalue λ_i.
For this exercise, consider the following 10×10 symmetric real matrix,

$$A = \begin{pmatrix} 1 & 11 & 7 & 9 & 7 & 11 & 7 & 9 & 2 & 11 \\ 11 & 4 & 10 & 10 & 6 & 2 & 9 & 6 & 10 & 0 \\ 7 & 10 & 3 & 5 & 4 & 4 & 4 & 4 & 6 & 10 \\ 9 & 10 & 5 & 3 & 8 & 8 & 3 & 5 & 1 & 8 \\ 7 & 6 & 4 & 8 & 8 & 10 & 5 & 6 & 10 & 0 \\ 11 & 2 & 4 & 8 & 10 & 9 & 4 & 3 & 5 & 11 \\ 7 & 9 & 4 & 3 & 5 & 4 & 3 & 10 & 7 & 2 \\ 9 & 6 & 4 & 5 & 6 & 3 & 10 & 11 & 1 & 7 \\ 2 & 10 & 6 & 1 & 10 & 5 & 7 & 1 & 10 & 5 \\ 11 & 0 & 10 & 8 & 0 & 11 & 2 & 7 & 5 & 1 \end{pmatrix}$$

c) Using the *power method* (see https://en.wikipedia.org/wiki/Power_iteration), compute the largest eigenvalue of the matrix A. How many iterations are necessary to converge up to 10^{-5}?

d) **Profile** your implementation, and determine the most expensive parts of it.

I. OPTIONAL EXERCISES

i. *Sockets revisited*

In B09 and D58, we have implemented client-server applications using `sockets`.

For this exercise, re-implement the client-server app. but in a *concurrent* fashion. I.e. do this by attempting two possible mechanisms:

(a) using multiple threads,

(b) using multiple processes;

for receiving/sending and processing/displaying messages and data.

Validate the implementation monitoring the processes/threads launched during execution.

ii. *Game of Life in 3D*

Most likely you are already familiar with *Conway's Game of Life* (https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life) cellular automaton. In this problem, we will generalize it to 3D –see Bays (1998), Bays (2006)–. There are different ways in which the *rules* can be generalizes leading to different results. You may consider different set of rules, or consider to use the ones given below:

(a) *resurrect* dead cells if they have exactly 5 neighbors

(b) *kill* live cells if they have less than 5 or exactly 8 neighbors.

– OR –

(a) *resurrect* dead cells if they have from 14 to 19 neighbors

(b) *kill* live cells if they have less than 13 neighbors.

In any case, any given cell has 26 neighbors to check – i.e. it will be a homogeneous 3-dim Cartesian grid[1].

---

[1] There have been other interesting implementations in more complex geometries and manifolds but we will use the simplest possible one for our case.