Documentation and Teach Cluster

Ramses van Zon

PHY1610H 2025 Winter

But first, some lessons learnt

Assignment 1

Write a program that can take 2d data sampled at discrete time points and compute a moving average of the norm of the 2d points as a function of time.

The data comes from a file with 3 columns, where the first column is time t and the second and third columns $x(t)$ and $y(t)$ are 2 coordinates. The program should compute the norm ($\sqrt{x^2+y^2}$), and then its moving average over n points. I.e. for each array element, it should compute the average of it and the preveeding n-1 elements. n will be an input parameter. Perform this moving average also for the time values. Write the result in two-column form to a file.
The program should take commmand line arguments that correspond to the input file name, the output file name and the width n of the running average.

Assignment 1 - Remarks 1/2

Keep it simple:

- if you're not using a function/class for various types, don't bother with templates.
- avoid `std::vector<std::vector<double>>` ; if your second dimension is small and fixed (e.g. 3), consider `std::vector<std::array<double,3>>`
- If your first dimension is small and fixed, an automatic array is fine: `std::vector<double> columns[3];`
- You don't have to store the n numbers, you can just index them as needed.
- Don't wrap a function around another function if it's not needed, i.e., not `double sum(vector<double>&v) { return std::reduce(v.begin(),v.end()); }`
- Use 'auto' only when the type is obvious (or obviously unimportant).

Don't reinvent the wheel:

- Needs an object to hold the `n` numbers while running over the array? Use an `std::deque`.
- The streaming operator >> can do splitting by space; no need to parse the lines yourself.

Assignment 1 - Remarks 2/2

Keep it safe:

- no “using namespace std”. You don’t know what identifiers are now defined.
- no global variables.
- check your input and arguments. If your program needs arguments but is run without out it, it should print an error message and not just crash.

Don’t waste but be aware of roundoff

- pass vectors by reference.
- don’t copy vectors if they dont change.
- If you compute the moving average by subtracting the tail value and ading the head value, roundoff error can accumulate. One bad data point (ie., a nan or inf) would ruin all subsequent points.

Assignment 2

We considered a one-dimensional variant of Conway's Game of Life, as conceived by Jonathan K. Millen and published in BYTE magazine in December, 1978.

This system is a linear set of cells that are either "alive" or "dead", and time progresses in discrete steps.

You are given a code, gameof1d.cpp (see below), that already computes the time evolution of this system, and for each time step, prints out a line with a representation of the state and fraction of alive cells. It can take parameters from the command line to set the number of cells, the number of time steps, and the initial fraction of alive cells.

Your task is to reorganize ('refactor') this code to be modular. The aim is to have separate functionalities be implemented in separate functions.

Assignment 2 - Remarks

- Don't submit object files, executables not vscode settings, _MAXOSX, .vscode, .DS_Store files.
- Create a separate header file for each module.
- Comment your code, particularly in the header
- Do not pass vectors by value.
- For consistency, include a module's header in its .cpp file
- Those who included a README file: Bravo!

In the Makefile:

- Executables depend on objects files, but not on headers or cpp files
- Object files depend on cpp and header files, not on object files
- Don't forget optimization flags
- Rules such as `$.o: %.cpp` are handy but cannot express the dependencies on header files; better to be explicit. Also, don't get too clever for simple projects.

Assignment 3

A modularized version of the hydrogen code can be found in the zip file below. Your assignment is to create tests for this code base.

For this assignment, you will have to use git and catch2. We explained in class how to get git on your computer, but you will have to install the Catch2 Libraries using cmake starting from the catch2 source code on https://github.com/catchorg/Catch2.

While proceeding with this assignment, we expect you to use git version control

Assignment 3 - Remarks

Although not everyone is done yet, a few point already came to light.

- Installing a library (here catch2) is sometimes easier said than done.
- It is okay if a test fails if that is because the code is not doing what it should.Good! Now we know what to fix.
- Compiling on different OSs and with different compilers can bring issues with the code to the fore. Good! Fixing this will make our code more portable.
- Using unsigned ints can give compilation issues (on clang for this code base).

  I usually advise against using unsigned int if any integer math is done, but I went against my own advice here.
- Don't submit object files, executables not vscode settings, _MAXOSX, .vscode, .DS_Store files.
- Codes that use 'assert' to flag an error can't be tested.
  Asserts should be used for cases that the code cannot expect, but throw-ing an exception is cleaner.
- No need to add library files like those of catch2 and rarray to your submission;
  you may assume they are there.

Documentation

Document your modules

The most unlikely pieces of code can end up being reused, so try and add at least a bit of documentation.

There are many documentation styles and philosophies:

- No documentation

  This style also often advocates no comments. The pretense is that clear code is 'self-documenting'.

  Sure, but . . . yeah, sorry, no.

- Auto-generated documentation

  Adding specially formatted comments that a tool like doxygen uses to generate documentation.

  This is pretty decent, and a good way to keep documentation up-to-date when the code changes.

- New-user oriented

  Your code will get read by someone with (much) less understanding of what it's supposed to do than you. If you were in this situation, what documentation would you need to be able to use the module?

  If you do nothing else, at least add a README.md file.
  You can also use Sphinx on top of doxygen to write documentation.

It all starts with comments.

- Yes, code should be as clear and to-the-point as possible.
- But it cannot express why something is done.
- At some point, your code will get read by someone with less understanding than you. And this includes your future self.
- Comments will help this person to quickly recall what the code supposes to do, or get familiar with someone's else code.
- One of the things to understand is what each function does, the type and values of the arguments it take, what it returns. as well as limiting cases and restrictions.
- By adding specially formatted comments, one can use tools like doxygen to auto-generate documentation. Doxygen can read your C++ code and combine the definitions of your functions and scripts with your comments, and generate a manual for your code.
- This is a good way to keep documentation up-to-date when the code changes.

Doxygen by Example

```
/// @file   outputarray.h
/// @author Ramses van Zon
/// @date   February 6, 2025
/// @brief  Module for writing a 1d array of doubles to text and binary files.
#ifndef OUTPUTARRAYH
#define OUTPUTARRAYH
#include <string>

/// @brief Function to write an array of doubles to a binary file.
/// This function does a raw dump of the array file to file.
/// @param  s  the filename
/// @param  n  number of elements of the array to write to file
/// @param  x  pointer to the first element of the array of doubles
void writeBinary(const std::string& s, int n, const double x[]);

/// @brief Function to write an array of doubles to a text file.
/// The file will contain each element of the array on a separate line.
/// @param  s  the filename
/// @param  n  number of elements of the array to write to file
/// @param  x  pointer to the first element of the array of doubles
void writeText(const std::string& s, int n, const double x[]);
#endif
```

Doxygen by Example (continued)

1. Generate a configuration file for doxygen called Doxygen (then edit it):

```
$ doxygen -g
$ sed -i 's/PROJECT_NAME[ ]*=.*/PROJECT_NAME=Outputarray/' Doxyfile
```

(The sed command just fills in the project name for us into the Doxygen file.)

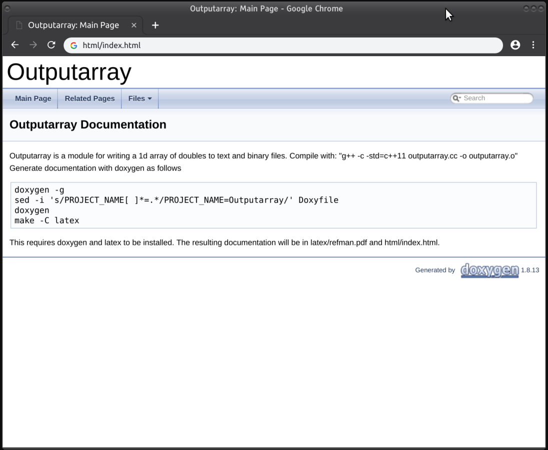
2. Create a README.md

```
[//]: # \mainpage
Outputarray is a module for writing a 1d array of doubles to text and binary files.
Compile with: "g++ -c -std=c++17 outputarray.cc -o outputarray.o"
Generate documentation with doxygen as follows
    doxygen -g
    sed -i 's/PROJECT_NAME[ ]*=.*/PROJECT_NAME=Outputarray/' Doxyfile
    doxygen
    make -C latex
This requires doxygen and latex to be installed.
The resulting documentation will be in latex/refman.pdf and html/index.html.
```

3. Generate the documentation in html and latex form:

```
$ doxygen
```

Doxygen by Example - HTML Result



Outputarray

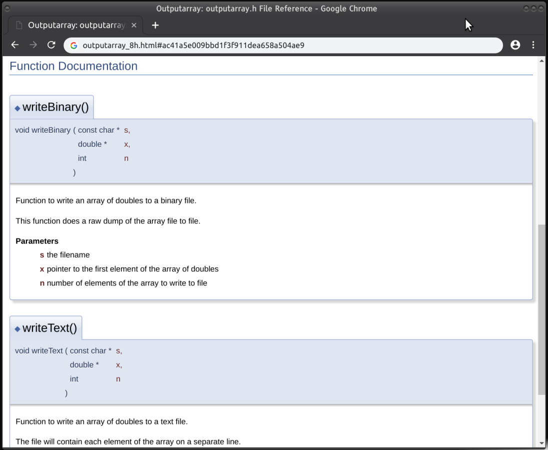
Outputarray Documentation

Outputarray is a module for writing a 1d array of doubles to text and binary files. Compile with: "g++ -c -std=c++11 outputarray.cc -o outputarray.o" Generate documentation with doxygen as follows

```
doxygen -g
sed -i 's/PROJECT_NAME[ ]*=.*/PROJECT_NAME=Outputarray/' Doxyfile
doxygen
make -C latex
```

This requires doxygen and latex to be installed. The resulting documentation will be in latex/refman.pdf and html/index.html.

Generated by doxygen 1.8.13



Outputarray: outputarray.h File Reference - Google Chrome
Outputarray: outputarray
outputarray_8h.html#ac41a5e009bbd1f3f911dea658a504ae9

Function Documentation

◆ writeBinary()

```
void writeBinary ( const char *  s,
                   double *      x,
                   int           n
                 )
```

Function to write an array of doubles to a binary file.

This function does a raw dump of the array file to file.

Parameters

- **s** the filename
- **x** pointer to the first element of the array of doubles
- **n** number of elements of the array to write to file

◆ writeText()

```
void writeText ( const char *  s,
                 double *      x,
                 int           n
               )
```

Function to write an array of doubles to a text file.

The file will contain each element of the array on a separate line.

In a Makefile

The command above use sed so that we can automate the documentation creation in our Makefile:

```
doc: outputarray.h
     doxygen -g
     sed -i 's/PROJECT_NAME[ ]*=.*/PROJECT_NAME=Outputarray/' Doxyfile
     doxygen
     make -C latex

.PHONY: doc
```

Teach Cluster

Compute Clusters

are needed when:

- My problem takes too long $\rightarrow$ more/faster computation
- My problem is too big $\rightarrow$ more memory
- My data is too big $\rightarrow$ more storage

This involves:

- hardware - cpus, multi-processors, network
- algorithms - parallelism, efficiency
- software - parallel programming, compilers, optimization, libraries, apps
- data management - RDM plan, data transfer, storage

Not your laptop!

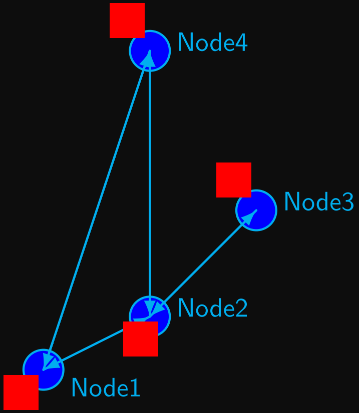
The architecture of compute clusters (a.k.a. supercomputers) is different than that of your own computer, and this matters.

There are a few prototypical architectures you should be aware of:

- Clusters
- Multi-Core Computers
- Accellerators

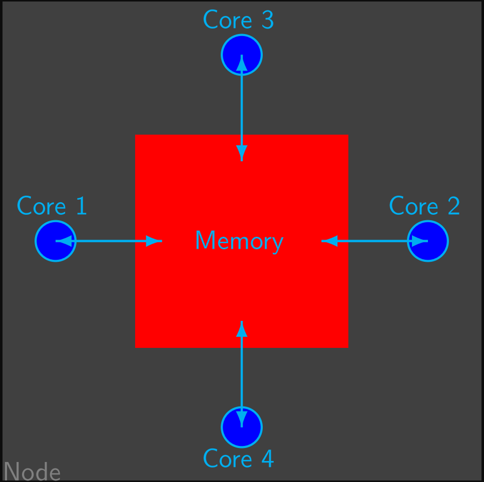
Furthermore, clusters are remote and shared resources, these machines need to be used quite differently from how you use your own computer.

Clusters



- Take existing powerful standalone computers, called nodes.
- Link them together through a network (a.k.a an *"interconnect"*).
- Easy to build and easy to expand.
- Because each node ● has its own memory a.k.a. RAM 🟥, these are called distributed memory systems.
- Nodes communicate and transfer data through messages.

Multi-Core Computers



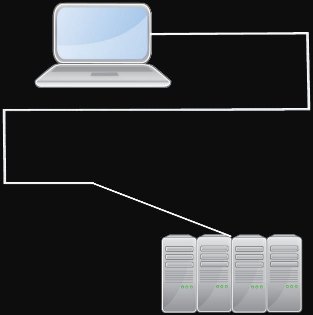
- A collection of processors on one node that can see and use the same memory.
- Limited number of cores, and much more expensive when the number of cores is large.
- Coordination/communication done through memory.
- Also known as shared-memory systems.

Your desktop, laptop and cell phone likely use this kind of architecture.

Accelerators

- Systems with accelerators have nodes which contain a device like a GPU.
- Accelerators are very fast and good at massively parallel processing (having 500-2000+ GPU cores).
- More complicated to program.

Compute Clusters Use Remote Access



- You're at your computer ("terminal")
- The cluster is in a data centre somewhere ("server").
- You must connect remotely using `ssh` ("secure shell").
- You must interact with the cluster using the command line.

Logging in to Teach

- You will receive an email with your user account and how to set your first-time password.
- To log in, type on the command line (could be in a local terminal in MobaXTerm in Windows):

```
$ ssh -Y USERNAME@teach.scinet.utoronto.ca
```

  and type the password.

Transfering files to Teach

- To download files to *Teach* when logged in, use

```
$ wget URL
```

- To copy files **from your computer** to the *Teach Cluster*

```
$ scp  filename  USERNAME@teach.scinet.utoronto.ca:path/filename
$ scp  USERNAME@teach.scinet.utoronto.ca:path/filename  filename
```

  For whole directories. add the `-r` option.

- For code, use git clone/push/pull!

Teach is a Shared Systems

- You're now on a login node **together with all other folks** working on the Teach Cluster.
- All other nodes of a cluster like this are compute nodes.
- The login node is for developing, compiling testing, and preparing compute jobs.
- To run on compute nodes, you need to create a job script that contains a request for specific resources for a specific time.
- You pass this job script to the scheduler.
- The scheduler used on the *Teach Cluster*, and on many other clusters, is called SLURM.
- The scheduler keeps the queue and allocates compute nodes to jobs in due time.
- Compute nodes see the same home directory as the login nodes.

Developing on Teach

- Most software is installed as **modules**.
- In particular, the C++ compilers are also in modules.So before you can compile, you need to do

```
teach-login01:~$ module load gcc
```

- For testing, `catch2` is also installed as a module:

```
teach-login01:~$ module load catch2
```

- Rarray is available a module as well.
- For documentation, doxygen and latex are already available without loading a module

```
teach-login01:~$ doxygen --version
```

  To view the resulting pdf, you can use `mupdf`.
- For editing, the path of least resistance is to use a terminal editor like **nano**, **vi**, or **emacs**.

Job Submission on Teach

- Log in to the *Teach cluster*

```
$ ssh  USERNAME@teach.scinet.utoronto.ca
```

- Copy the material needed for this example

```
$ wget https://pages.scinet.utoronto.ca/~rzon/ia.tgz
$ tar xzvf ia.tgz
```

- Change to the newly created directory

```
$ cd ia
```

- Submit the job ‘sweep_bondbreak.sh’

```
$ sbatch sweep_bondbreak.sh
```

- Check the status of your job(s)

```
$ squeue --me
```

- Once running/completed, check the output in `slurm-*.out`

```
$ less slurm-*.out
```

Job script: sweep_bondbreak.sh

```
#!/bin/bash
```

$\longleftarrow$ First line makes this a bash shell script

```
#SBATCH --ntasks=1
#SBATCH --time=01:00:00
#SBATCH --mem=1000M
```

$\}$ #SBATCH lines request 1 core, for 1 hour (on teach –mem is ignored)

Rest of script kept to run on allocated compute node.

```
module load python/3.10.2 scipy-stack/2023a
```

$\longleftarrow$ Most software requires module commands

```
TEMP=2.2      # temperature value
NUMSEEDS=500  # number of seeds
OUTPUT=output-$TEMP-$SLURM_JOB_ID
mkdir -p $OUTPUT
```

$\longleftarrow$ Setup up parameters

```
# Run multiple cases with different random seeds
for SEED in $(seq $NUMSEEDS) ; do
   echo "Simulation $SEED of $NUMSEEDS"
   ./bondbreak -t $TEMP -s $SEED -f $OUTPUT/$TEMP-$SEED.dat -l $OUTPUT/$TEMP-$SEED.log
done
```

$\longleftarrow$ A loop in the bash shell

$\nwarrow$ Pass parameters to bondbreak app

```
# Extract the breakage times from the logs
awk '/BREAKAGE DETECTED/{print $8}' $OUTPUT/$TEMP-*log > $OUTPUT/breaktimes.dat
```

$\swarrow$ Collect breakage times

Using the Scheduler on Teach

Main commands to interact with the scheduler

`sbatch`	submit job
`squeue`	see queued jobs and their status
`scancel`	cancel a job
`debugjob`	get short interactive job on a compute node

Common sbatch parameters for *Teach*:

`-t`	`--time`	amount of time
`-N`	`--nodes`	number of nodes
`-n`	`--ntasks`	number of tasks
	`--ntasks-per-node`	number of tasks per node
`-c`	`--cpus-per-task`	number of threads per task

More Information on Teach

- There is a wiki page on how to use *Teach*:

  https://docs.scinet.utoronto.ca/index.php/Teach

- Because the software stack is the same as on the Digital Research Alliance Clusters, the Alliance documentation can be helpful as well:

  https://docs.alliancecan.ca