Scientific Computing – Learning from my mistakes!

Thomas Winyard

November 12, 2021

1 Write code for people, not a computer

1. **A program should not require its readers to hold more than a handful of facts in memory at once.**

   - Humans can only hold a handful of concepts or ideas in memory at one time.
   - Each task should only require a handful of ideas.
   - Break your code into functions that are easy to understand and use only a handful of concepts.
   - base file for my code is normally 20 lines long

2. **Make names consistent, distinctive, and meaningful.**

   - don't use variable names like results and results2 results3 etc.
   - you will need to read through your own code in months and you will have forgotten why you added that variable called test.

3. **Make code style and formatting consistent.**

   - Just as if you are reading a paper and your notation changes in the middle, it is far harder to read and debug code that changes notation or style.
   - Come up with your own standardised notation and use it, it will help you understand your own code quickly.
   - e.g. matrices and arrays are capitalised but doubles and integers and lower case. My functions start lower case and each new word is capitalised, where as classes are capitalised. i,j,k,l are always integers and are always indices. The variable result is only used when a function returns a value. Systems variables are capitalised.
   - develop your own standard for file extensions and outputs. I use .out for data files.

2 Data

Ensure that raw data are backed up in more than one location. Create the data you wish to see in the world. Create analysis-friendly data.

3 Let computers do the work

1. **Make the computer repeat tasks.**

   - often a large amount of task repetition e.g. procesing large numbers of files or regenerating figures for new data etc.
   - So many of my colleagues never take the time to learn how to automate these tasks!
   - Doing it yourself increases the chance of mistakes.
   - Use a command line interface with the command history turned on. Also learn how to search your command history (changes between platform).
   - "history", "history — grep -i searchterm — less",
   - grep just searches line by line for the search term, one of the most used commands, write it on your wall
   - have a wall of commands. - not just the commands you use every day but make it inspirational, which commands would make you a better programmer?
   - learn how to use bash scripts!
   - postprocessing, so making plots, submiting multiple jobs, combining data etc. etc.
   - take the time now to learn how to do it and you will save time in the future.
   - ideal example is the compiling and linking for languages.
   - the most widely used tool here is make, I personally make use of CMake as I code in C++. Sample CMake file.
   - they define how files depend on each other e.g. if A has changed then B needs updating. etc. etc. It will decrease compile times and improve your file structure.

2. **Use a build tool to automate workflows.**

   - e.g., specify the ways in which intermediate data files and final results depend on each other, and on the programs that create them, so that a single command will regenerate anything that needs to be regenerated.
   - In order to maximize reproducibility, everything needed to re-create the output should be recorded automatically
   - remember to record the following:
   - unique identifiers and version numbers for raw data records
   - unique identifiers and version numbers for programs and libraries;
   - the values of parameters used to generate any given output;
   - the names and version numbers of programs (however small) used to generate those outputs.

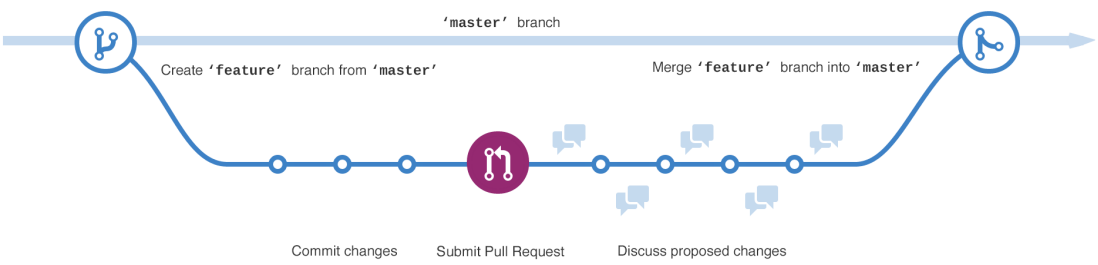
4 Incremental Changes

1. **Work in small steps with frequent feedback and course correction.**

   - The requirements and demands in our code often changes as the project develops.
   - This means the standard deign first approach doesn't work.
   - rather than trying to plan months or years of work in advance work in small self contained steps.
   - If you are working as part of a team each step should take about an hour to develop.
   - steps are then combined into an iteration or new version which should be no longer than a week (for us it may be much shorter time frame).
   - each step should be self contained and each iteration should produce working code that is testable.
   - while the time frames might not match up thinking in terms of steps and iterations is useful as it forces you to break your code down into chunks.
   - this is called agile development.

2. **Use a version control system.**

   - If you aren't using some sort of version control, start now!
   - simplest form just number your code iterations and have different folders.
   - don't use "current", "new code", "updated code", "even newer code" etc.
   - Even better is to use a version control system or VCS.
   - keeps track of all changes so you can revert to any point in your history. It also keeps al versions true to each other and deals with changes that conflict between different copies.
   - this is particularly useful when your code has to live in multiple places.
   - I recommend git which also has a free hosting service github.
   - open source
   - commit changes, head location, conflict resolution, branching, can still use a working code while developing new features.



   - Don't use dropbox, you will end up making changes that conflict.

3. **Refer to the version control with meta data.**

    - Dont include outputs in your version control
    - update your git ignore
    - develop your own standard extensions so you can ignore them from git.
    - bad practice to produce outputs inside the code library have a projects folder where you outputs go.
    - keep all the outputs well organised have a good file structure.
    - keep the version of the code that produced the output with the output, it may be very useful if you find errors.

5 Never repeat code

- Anything that is repeated in two or more places is more difficult to maintain. Every time a change or correction is made, multiple locations must be updated.
- Use the DRY approach, DO NOT REPEAT, this goes for both data and code!

1. **Every piece of data must have a single authoritative representation in the system.**

    - Physical constants ought to be defined exactly once to ensure that the entire program is using the same value
    - raw data files should have a single canonical version

2. **Modularize code rather than copying and pasting.**

    - when a change is made or a bug is fixed, that change or fix takes effect everywhere
    - not only reduces errors but improves your confidence in your code.
    - also improves your mental picture of the code. e..g breaking it into chunks is easier
    - if you ever copy and paste code, asl yourself first, are you about to make a mistake and can this be generalised.

3. **Re-use code instead of rewriting it.**

    - At larger scales, it is good to reuse your own code.
    - also in terms of the wheel, reinvent the wheel, then burn it and buy a new one.
    - many lines of high quality libraries that are open source (stick with open source).
    - simple processes such as (e.g., numerical integration, matrix inversions, etc.) learn how it's done then use someone elses.

6 You will make mistakes

Mistakes are inevitable, so verifying and maintaining the validity of code over time is immensely challenging. Hence use defensive programming.

1. **Use assertions/**
   test inputs and parameters etc. etc. that they are of the right form, that your energy is positive and return issues if not assert len(smoothing) ¿ FILTERLENGTH,
   'Not enough smoothing parameters'
   assert 0.0 ¡ result ¡= 1.0,
   'Bradford transfer value out of legal range'

2. **Use automated testing.**

   - Automated tests can check to make sure that a single unit of code is returning correct results (unit tests),
   - that pieces of code work correctly when combined (integration tests),
   - and that the behaviour of a program doesn't change when the details are modified (regression tests).
   - These are run by the computer, it is good practice to have a command that you can run that will run all the tests and to run it everytime you make an iteration on the code.
   - e.g. my code checks that it can find the derivatives of certain functions with various boundary conditions and integrate known functions, then larger scale it finds a known energy of a particular configuration that only takes a minute to find and compares to the known energy.

3. **Use an off-the-shelf unit testing library.**

   - not something I have made any use of but they are sworn by by those that use them.
   - will initialise inputs and systematically test chunks of code.

4. **Turn bugs into test cases.**

5. **Use a proper code editor.**

   - I used to exclusively used gedit to edit code and I still often do.
   - when making large edits it is useful to have a piece of software that will keep track of all the dependencies of your code and help autocomplete function names and check your inputs. It will also try to point out errors before you compile your code. Can be really useful!
   - I currently use Clion and atom.

6. **Use a symbolic debugger.**

    - a debugger allows users to pause a program at any line (or when some condition is true), inspect the values of variables, and walk up and down active function calls to figure out why things are behaving the way they are.
    - using print statements has it's useful and I tend to have two running modes for my code debug and nondebug.
    - adding and removing print statements is bad practice (though I do it all the time).
    - debuggers are ussually more productive.

7 Optimize once you are ready

The time cost of a project is not the run time of your code. It is the development time + debug time + the run time. Get your code working first! Then test it to within an inch of it's life. Then and only then, if you feel it is worth your time optimising the code, the optimize it.

1. **Use optimisation flags.**

    - most software, highlevel and low level programming langauges can be set to release code that is more streamlined. It often taks longer to compile and performs less checks on what you are doing (so make sure it works first!)
    - -O1, -O2, -O3, -Ofast flags for c++ makes the compiler optimise more
    - beware of -Ofast, I only use -O3.

2. **How much do you need to speed up your code?**

    - improve method.
    - memory management and resource management
    - openMP
    - openMPI
    - CUDA

3. **Have built in timers in your code as standard.**

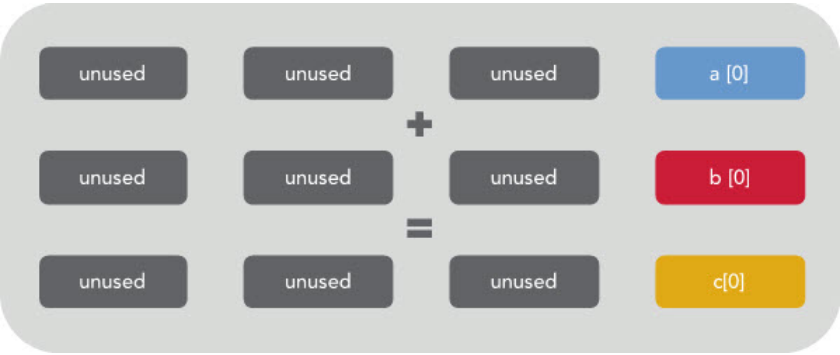
    - always time your code and check those times.
    - do the following things scale as expected:
    - how does the time taken scale with more threads?
    - how does the time taken scale with more nodes?
    - how do your different methods affect the time taken.
    - how does changing your accuracy scale the time taken.

4. **Use a profiler to identify bottlenecks.**

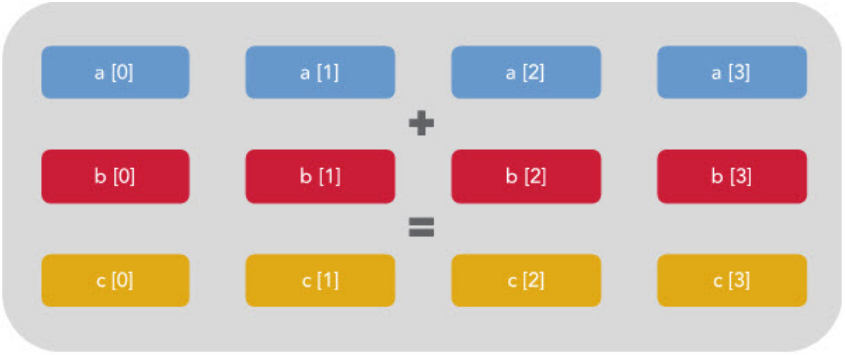
- Can automate the finding of performance bottlenecks
- – inefficient programming
- – memory, I/O bottlenecks
- – vectorization
- – parallel scaling
- TYPES:
- Hardware counters
  – count events from CPU perspective (no of flops, memory loads, etc) – usually need Linux kernel module installed
- Statistical profilers (sampling)
  – interrupt program at given intervals to find what routine/line the program is in
- Event based profilers (tracing)
  – collect information on each function call
- Intel's performance profiler VTune™ Amplifier XE

5. **vectorisation.**

- Vectorization is the process of converting an algorithm from operating on a single value at a time to operating on a set of values at one time.
- Modern CPUs provide direct support for vector operations where a single instruction is applied to multiple data (SIMD).
  for$(i = 0; i < 4; i++)$
  $c[i] = a[i] + b[i];$
  In a serial calculation, the individual vector (array) elements are added in sequence. The additional register space in modern CPUs is unused.



  In a vectorized calculation, all elements of the vector (array) can be added in one calculation step.



- Issues that impact vectorizing your code:
- Loop Dependencies (Avoid read-after-write)
- Indirect Memory Access (Use loop index directly. Seek unit loop stride)
- Non 'Straight line' code (function calls, conditions, unknown loop count)

```
for (i = 1; i < end; i++)
  f[i] = f[i-1] + b[i-1];
```

```
for (i = 0; i < end; i++)
  c[idxC[i]] = a[i] + b[i];
```

```
for (i = 0; i < CalcEnd(); i++)
{
  if (DoJump())
    i += CalcJump();
  c[i] = a[i] + b[i];
}
```

6. **Write code in the highest-level language possible.**

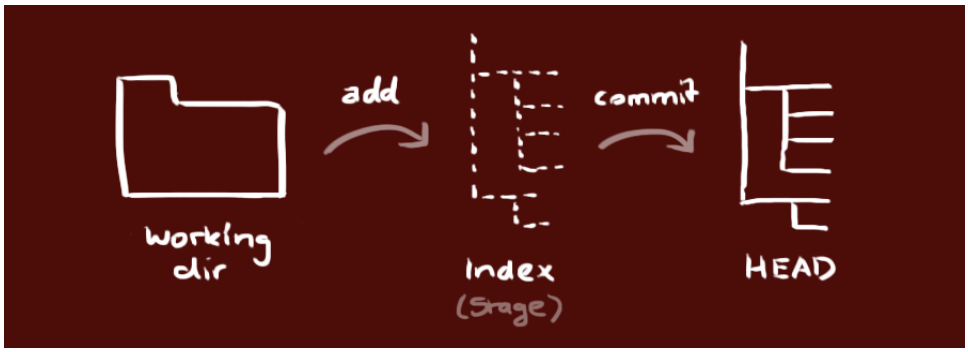
8 Collaborate

Peers, share code, the internet.

9 Git and Github

- git clone /path/to/repository

- your local repository consists of three "trees" maintained by git. the first one is your Working Directory which holds the actual files. the second one is the Index which acts as a staging area and finally the HEAD which points to the last commit you've made.



9.1 add and commit

- You can propose changes (add it to the Index) using
  git add <filename>
  git add *
  git add -A

- This is the first step in the basic git workflow.

- To actually commit these changes use
  git commit -m "Commit message"
  Now the file is committed to the HEAD, but not in your remote repository yet.

9.2 pushing

- pushing changes

- Your changes are now in the HEAD of your local working copy. To send those changes to your remote repository, execute
  git push origin master

- Change master to whatever branch you want to push your changes to.

9.3 branches

- Branches are used to develop features isolated from each other.

- The master branch is the "default" branch when you create a repository.

- Use other branches for development and merge them back to the master branch upon completion.

- create a new branch named "featureX" and switch to it using
  git checkout -b featureX

- switch back to master
  git checkout master
- and delete the branch again
  git branch -d featureX
- a branch is not available to others unless you push the branch to your remote repository
  git push origin ¡branch¿

9.4 update and merge

- to update your local repository to the newest commit, execute
  git pull
  in your working directory to fetch and merge remote changes.
- to merge another branch into your active branch (e.g. master), use
  git merge <branch>
  in both cases git tries to auto-merge changes.
- Unfortunately, this is not always possible and results in conflicts. You are responsible to merge those conflicts manually by editing the files shown by git. After changing, you need to mark them as merged with
  git add <filename>
  before merging changes, you can also preview them by using
  git diff <sourceBranch> <targetBranch>

9.5 When you screw up – replace local changes

- In case you did something wrong, you can replace local changes using the command
  git checkout - - <filename>
- this replaces the changes in your working tree with the last content in HEAD.
- Changes already added to the index, as well as new files, will be kept.
- If you instead want to drop all your local changes and commits, fetch the latest history from the server and point your local master branch at it like this
  git fetch origin
  git reset - -hard origin/master

9.6 git .ignore

Make use of this file and always add any output files with wildcards and any output files from compilations etc.