quantique-UdeS repositories v1.1

Maxime Charlebois
Sherbrooke
September 2017

Contents

1 Objective

Every year many students graduate or leave Université de Sherbrooke with all their newly acquired knowledge. The problem is that most of the time, they want to leave their codes to the group but they go away leaving behind to their supervisor a source code directory impossible to understand and to work with. This natural erosion of knowledge happens in all universities.

The present procedure is an attempt to slow down or even stop this erosion. The solution proposed is a peer reviewing publication service of source code supported by the Institut Quantique of Université de Sherbrooke. We want to create a service that every professor will be able to suggest to their programming students who are willing to leave their code to the group before they go away.

The procedure is designed to be simple and straightforward to apply. The goal is to make the code permanent through simple and systematic requirements that will be evaluated. Having the code approved by the service means that it is compliant to every "source code requirements" defined in the next section; it is a quality seal that insures reusability.

This procedure also details how the reviewing process should happen, what to put in and what not to put in the publication repository, and finally what happens when someone wants to update this procedure.

Throughout the document:

- "Programmer" refers to the individual that wants to submit and archive the code,
- "Reviewer" refer to the individual responsible for evaluating and make the code compliant with every step of this procedure,
- "Requirements" are the elements that the source code submission should contain to be compliant.
- "procedure", the set of systematic rules that are contained in this document.

2 Source code requirements

The requirements defined here are chosen to be the simplest and most essential elements you need to successfully download, compile and run a source code. The requirements are chosen so that they mimic some well established open-source software packages (Lapack, GSL, Python, CMake, Gnuplot, Git). Of course, there is no absolute standard, but there are a few things one needs to know before being able to compile any of these software packages. Based on these library standards, we define here our standard. To be compliant with the present procedure, the source code directory MUST contain:

1. a README file,
2. one or multiple useful case examples.

It CAN also contain:

1. a testing program,
2. an external documentation,
3. a LICENSE file.

2.1 README

Everything here revolves around the README file. The root directory of the source code must have a README or README* file that specifies or links every needed information. It must be short (around 200 lines, under 400 lines), in order to provide a good overview of all the important information. It can refer to an outside file, like an "INSTALL" file that specifies a longer installation procedure, for example.

The README file MUST contain:

1. the dependencies (libraries, versions),
2. the installation procedure on every system supported (Linux, Mac, MP and/or MS),
3. the relative path to the example directory(ies) and documentation.

The README file CAN contain:

1. a development repository link,

2. the installation procedure for a testing program,
3. the documentation on how to run the testing program,
4. the relative path of external documentation,
5. anything else judged relevant to the code.

The reviewer job will be to certify that the README file contains at least the minimal information required and also (and this is important) that every claim in the README file is true and consistent. Most of the time, when a code evolves over time, the comments, the examples, the README, the documentation become obsolete, false or incomplete. Here, the reviewer will have the job to verify every point of the README file in order to grant the seal of approval.

The more claims there are, the more time the reviewer will work to test every point of the README file.

2.2 Example directories

At least one example of usage and expected results must be provided. It is important to understand that it is possible that the program can do much more what is presented in the README file and example(s). By publishing the code through this procedure, you advertise which task the software can accomplish. The only task that will be taken in consideration are the ones with examples. If the program can do more than what is advertised from the README and examples, it will not be considered during the reviewing process. The knowledge on how to run the software can then remain hidden from the README file, but this is exactly the opposite of what we try to accomplish here. Indeed if, on the other hand, the README or the documentation linked in the README describe the program and what it is capable of calculating, but there is no example directory and files that show an example of this calculation (but only some other calculation, not related to this), this will be considered inconsistent and faulty.

The example can be the testing program, as long as it is sufficiently self-explicit, documented, and that the user can run every part of the test in a simple way.

Every example must be self-explanatory and pedagogical (it can be accompanied with a local README file). This is the main tool through which an external user of the published code will learn how to use the code. This also offers a great way to start to understand the source code itself, since the user can simply look at the output generated, find where this output is produced in the code, and work his way around from there.

Every example or test should also run in a small amount of time. It is supposed to be a sample of what the code is capable of doing. Hence, the example should run with a small sample of computation. However, every example should still produce a valid physical or numerical result. For example, if the code calculates an integral, and to minimize the calculation time of the example, you reduce the precision of the integral to a point where the result is false, the example is not valid. This can be very hard to test for the "reviewer" since he most likely does not know enough about the physics of every software to discriminate which result is valid and which is not. But the programmer is expected to generate examples that he claims to be physically correct (like any scientific publication). Now, if for other parameters or options, the same task presented in examples produces a wrong result, this is not the same. It suffices that examples chosen by the programmer produce valid results, to the best of his knowledge.

2.3 Minimal working examples (MWE)

You can find two minimal examples of approved and published source codes on the site:

www.physique.usherbrooke.ca/quantique-udes/

More specifically, the links are:

www.physique.usherbrooke.ca/quantique-udes/afmconductivities.tar.gz
www.physique.usherbrooke.ca/quantique-udes/onebody.tar.gz

The codes are designed to be simple to read from start to finish. They provide some kind of MWE that anyone can read to understand the basis of the procedure. You will see that it does not require a lot to be compliant.

Now, in order to remain simple, we also chose the simplest way to comply with every requirement. For example, in both cases, the installation procedure is a basic makefile with minimal information. It does not mean that an installation procedure with "CMake" or "Autotools" is not compliant. Indeed, it is probably better than the MWE provided. But information in both MWE is the minimum you must provide with a makefile approach. You don't absolutely need a makefile approach if you have a more elaborate approach. It suffices that it works when the reviewer follows the installation procedure. The same remark applies to every other requirements.

2.4 License

"*When you make a creative work (which includes code), the work is under exclusive copyright by default. Unless you include a license that specifies otherwise, nobody else can use, copy, distribute, or modify your work without being at risk of take-downs, shake-downs, or litigation. Once the work has other contributors (each a copyright holder), "nobody" starts including you.*"[1] This is a consequence of the Berne Convention, which most countries have signed, including Canada.[2]

So, if you make a code available publicly, it might be preferable to put a license on it. Licenses in general can be a very messy business as it involves law and lawyers at some point. It has to be compatible with the other license of the libraries you use. The simplest example is that you cannot use a "GPL License" library to create a monetized software. But you can use a more permissive licensed library like the "MIT License" for example. This last license is the most popular open source license since it give almost any right to the user, except that the programmer cannot be considered guilty of anything.

There is a lot more license options out there, very well documented, with every possible variant. You can also create your own license. In any case, if you want a license, you just need to create and edit the `LICENSE` text file in the root directory of your source code.

Finally, note that if the code you want to publish already contains a private license or ownership that prevents such a publication process (public or private), the procedure will abort.

3 Code submission procedure

3.1 Programmer

It is a little bit like the peer reviewing process in science publication like PRB except that the reviewer is known. The programmer checks all the different requirements by reading this procedure and checking the minimal working examples (see section 2.3). Once he/she feels like the code is up to date and compliant with the requirements, he/she submits the code (in a compressed .tar.gz archive) to the reviewer via email. The submission email should not contain information crucial to the code compilation, since every important

[1] https://choosealicense.com/
[2] https://en.wikipedia.org/wiki/Public-domain_software

detail should be in the README file. The only information necessary should be the programmer's name and email (obtained from the email itself) and the type of hosting targeted: private or public.

3.2 Reviewer

Once the reviewer obtains the first submission, depending on the README claims, the source code should be compiled on a specified OS.

3.2.1 Linux

If Linux is specified, the reviewer must install a new virtual machine of the last Ubuntu LTS release. It is easy to create and install a benchmark virtual machine and copy paste it in Virtualbox for example. The configuration of a new virtual machine is not something to repeat for every evaluation of a program. The Linux installations must be new in order to avoid some already installed libraries. Of course some basic software like `gcc` or `make` will be installed, but this is why we specify the version of Linux of the tests.

3.2.2 Mammouth

On mammouth (MS or MP), the reviewer should unload every module (by commenting or deleting the `.modules` or `.modules_mp2` files) and resetting his `PATH` and his `LD_LIBRARY_PATH` environment variables to the default values (by commenting every export of these variables in the `~\.bash_profile` file and `~\.bashrc` .

3.2.3 macOS

On Macintosh, it is not possible, through legal ways, to obtain virtual machines of OSX. Hence the reviewer will use an existing Macintosh personal computer.

3.2.4 Steps

Once the system is ready to test the source code, the reviewer opens the submitted .tar.gz archive. Guided by the README, the reviewer then follows these 4 steps:

1. Installing every required library. If no version of the libraries is specified, the reviewer then tests the default (or most recent version) as well as the oldest version that can be installed in order to verify that the results do not depend on the version. Most of the time, it is preferable to write a version than let this parameter free.
2. Compiling the code and checking if the binary is generated (if it is a compiled language).
3. Run the software in order to reproduce the examples contained in the example directories.
4. Verify that the examples give the same results as the ones in the subdirectory results.

3.3 Fail

If any of the steps above fails, the reviewer needs to note that what went wrong, and what needs to be corrected. As far as possible, the reviewer must then try to get around the obstacle in order to continue to follow the rest of the steps (by installing a more recent library version than specified or by running the code with the correct parameters instead of the ones specified from the README, for example). Once the reviewer is unable to move forward anymore, the submission is returned with the verdict, and the cause of failure (as well as any other point noted along the way).

3.4 Iterations

The programmer can then take note of the failed result and try to make corrections in order to submit again. Most of the problems might come from versions of the libraries needed or the OS version to compile or run. A simple workaround would be to specify a more restraining version of the libraries or OS needed to compile. The resubmission can simply be a reply to the "Failed submission" received. Technically, there can be an infinite number of resubmissions, but in order to speed up the process on both parts, it is better to avoid submitting unless fairly certain that the submission is compliant with the requirements.

3.5 Success

Once everything is corrected and up-to-date, the reviewer can approve the submission. The reviewer then adds to the end of the README file: his/her name, the date of approval, the version of the procedure followed and the OS on which the code was tested.

The reviewer can then proceed to publish the source code.

Note that in the case of an update of an already existing publication, the source code must go through the same submission process. It should be faster, since it has already been published before, but it is necessary to verify that every claim still holds.

4 Publication

A successful reviewing process of a source code will result with a publication of a release version on the department website:

www.physique.usherbrooke.ca/quantique-udes/.

Everything will be moved to one directory and compressed in a `tar.gz` file that will contain the minimum needed to compile the code. It does not contain the binary compilation of the code, nor any external libraries, since they can be obtained from other sources.

Any unused file (backups, temporary directories, unnecessary non-text files, etc.) of any kind will be deleted prior to the publication, in order to clean up the root directory of the published code. This includes at least all the binary and other files generated from the compilation and the usage of the code (apart from the expected results from examples).

The whole publication process does not require to use any source control software, like git or mercurial, but if it does, the `tar.gz` must use the `.hgignore` or `.gitignore` information not to include these files. This can be done with the options `--exclude-vcs-ignores --exclude-vcs` of the `tar` command (see the MWEs).

Once the `tar.gz` file will be created, it will be added to the website (modifying the index.html) and it will be given a version number. If there is a public development repository, a tag of the release of the version will be added to the repository.

5 Change of reviewer

This procedure is defined in a way that does not depend on the experience of the reviewer with the code to archive. At any point, the reviewer can change and the procedure will still hold. There are a few details to address when the time for a change in the reviewer comes. The new reviewer must gain permission to modify the publication website and an email must be sent to tell the department he is now in charge of the reviewing process.

6 Update this procedure

The procedure described here can be viewed as too (or not enough) constraining. It is suggested that it should be updated in order to improve it.

This is the main reason to put a version number to the procedure (present in the title of the document). If the procedure is updated, it does not imply that every code already published needs to be updated. Indeed, the published code already contains the version number of the procedure. However, every version of the procedure must be archived and available.

When someone updates the procedure, the example codes (MWE) on the website must be updated in order to make everything compliant.