Training Computational Scientists to Build and Package Open-Source Software

Prentice Bisbal
Princeton Plasma Physics Laboratory
Princeton, NJ
pbisbal@pppl.gov

ABSTRACT

High performance computing training and education typically emphasizes the first-principles of scientific programming, such as numerical algorithms and parallel programming techniques. However, many computational scientists need to know how to compile and link to applications built by others. Likewise, those who create the libraries and applications need to understand how to organize their code to make it as portable as possible and package it so that it is straightforward for others to use. These topics are not currently addressed by the current HPC education or training curriculum and users are typically left to develop their own approaches. This work will discuss observations made by the author over the last 20 years regarding the common problems encountered in the scientific community when developing their own codes and building codes written by other computational scientists. Recommendations will be provided for a training curriculum to address these shortcomings

KEYWORDS

Open-source Software, Computational Science, Training

1 INTRODUCTION

It has been observed that physical scientists and engineers often do not have the basic computing skills necessary to use computers effectively. This even includes computational scientists engaged in parallel computing.[15] This observation has lead to training efforts such as Software Carpentry[13], Data Carpentry[12], and HPC Carpentry[8]. These "carpentry" programs, collectively known as "The Carpentries," are workshops designed to provide training for physical scientists and engineers, data scientists, and computational scientists, respectively, in practical computer skills relevant to these groups of scientists to help them use computers effectively in their work. The purpose of these workshops is not to make the participants experts on any of the topics covered in these workshops, but to give them "good enough" skills.[14]

In the spirit of this observation and these workshops (and others like them), the author recommends training programs for computational scientists consider including training for two skills that, based on the author's experience, would be valuable to computational scientists:

(1) How to compile open-source software packages
(2) How to package open-source software for others to use

Not all computational scientists need to be proficient in both skills. All should be proficient in (1), but only those computational scientists developing codes to be used by others need to know (2).

The author basis these recommendations on his experience. The author has been a Linux system administrator and high-performance computing (HPC) specialist for the past 20 years. During this time, he has supported computational scientists in a variety of fields including plasma physics, engineering, chemistry, computational biology, astrophysics, particle physics, and weather modeling.

Most of this experience has been in smaller departments or institutions where there author was the sole HPC specialist, responsible for every aspect of HPC operations, including installing the computational software needed by the users, and providing technical support to those users. As a result, he has built and installed open-source scientific packages thousands of times, and has helped numerous users trying to compile software themselves.

For example, in the summer of 2009, the author provided HPC support for a two-week long summer program for graduate students and postdoctoral fellows in Astrophysics[9]. The instructors were accomplished computational astrophysicists from the United States and Europe. Despite the author setting up a parallel computing cluster specifically for this for this program, and installing all the computational software that the students would need for this program, which was all open-source, the instructors perferred that the students try to install all the need software themselves on their own laptops. They felt the ability for these aspiring computational scientists to install opens-source tools like this was an essential skill for their careers as computational scientists.

In order to facilitate discussion, the organizers of the program set up a group mailing list where the students could discuss the course content and ask each other for help with their homework assignments. The discussion on this list centered almost exclusively on how to compile and install the open-source tools needed for the for the program. The author was part of this mailing list, and estimates he sent over 200 hundred emails to the list during the 2-week program providing the students with assistance debugging their installations issues, and explaining the configuration and build process.

This intensive experience provided considerable insight into what practical computer skills and knowledge computational scientists are lacking at the graduate and postdoctoral levels, as well as the areas of building open-source software where most errors occur. Many of the recommendations how to teach building open-source software are based on this experience in particular.

https://doi.org/10.22369/issn.2153-4136/10/1/12

Even most though computational scientists can rely on the IT or HPC support staff at their home institution to provide most of the software they need, there are a number of reasions why computational scientists should know how to build open-source software themselves:

- Their IT or HPC department may be short-staffed, leading to delays in installing the software they need when they need it.
- To keep workloads reasonable, some HPC centers limit the software that they will support to key libraries applicable to most users or the most performance-critical libraries, leaving it to the scientists to manage the software specific to their research themselves.
- As a policy, some HPC centers will only install final release versions of a software package, so if a user needs a new feature or bug fix in a pre-release version, they may have to install it themselves.
- As laptops have become smaller and more powerful, the author has noticed more computational scientists using their laptops for small computational workloads, and wanting all the computational packages they use on their home institution's cluster on their laptop so they can work at home or when travelling to conferences.

For computational scientists who develop software, it is in their best interest to learn how to package and distribute it in a way that makes it as easy as possible for others to install and use it. The academic research culture is often described as "publish or perish". That is to succeed you need to publish your research often to survive and get ahead. Related to this, many researchers who publish are also judged by how often their research is cited. For computational software developers, their work is often cited when other researchers use their software in their own research. These users are more likely to use a software packages that is easy for them to install, so the easier a package is to install, the more likely it is to be used, and the more it is used, the more likely it is to be cited.

Many computational software development projects are publicly funded through grants from agencies like the National Institute of Health (NIH), or the National Science Foundation (NSF), or they are developed a national lab, where they have to compete for budgeting with other projects. To continue to be funded, these projects often have to periodically demonstrate to the funding agency that the projects are worthwhile. "Worthwhile" in this case usually means that other researchers see value in this software and use it. A metric commonly used to demonstrate this is how many times that software has been cited. Making a software package easy to install can help increate this citation count and help justify continued funding of its development.

In the author's experience, open-source computational software packages are, on average, much harder to install than more general-purpose open-source packages. These difficulties are generally due to several issues with the software:

- Poor or non-existent documentation on how to build the software, either on the software's web page, or included in the source code.
- No automated configuration and/or build system is provided, leaving the user to determine the proper configuration and/or manually run the commands to compile and install the code.
- The software does not follow follow existing standards or current practices, requiring the builder to modify their procedures to accommodate these differences.

Any one of these deficiencies can create significant obstacles for experienced software builders. For computational scientists with limited experience, any one of these deficiencies can be insurmountable.

In the next two sections, the author will outline a curriculum for training computational scientists in each of these skills, including topics that should be included, and why, as well as point out topics that should be omitted to prevent overwhelming the student. Resources will be provided that provide more detail on each topic. These outlines, combined with the recommended resources, can then be used by the reader to create a complete training class.

In the spirit of Wilson, et. al.[14] the goal of this training should not be to make the trainees experts in either of these skills, but to give them skills that are "good enough" for them to to be more productive and successful.

2 HOW TO BUILD OPEN-SOURCE SOFTWARE

In the author's experience most open-source software, provides a `configure` script created with GNU Autoconf[1]. The author estimates 90% or more of the software he has installed falls into this category, so he recommends that training focus exclusively on building software that uses this tool, since this is most relevant tool due to market share, and introducing multiple configuration tools at once may take up too much time, or overwhelm the student.

Most of the errors encountered by running this script, or actually compiling the software, are the result of a file not being found. This is usually corrected by specifying the correct location of the file by defining an environment variable, or specifying the proper path with a command-line option to the configure script. Since the main goal of the build process is to compile source code, the following topics should be covered before explaining to use the configure script:

(1) The Filesystem Hierarchy Standard
(2) Environment Variables
(3) The Compiling Process

2.1 Filesystem Hierarchy Standard

The Filesystem Hierarchy Standardard[11], or FHS, is standard that specifies where certain types of files should go on a Linux filesystem. Sofware builders do not need to know the entire standard, but they should know about the directories listed below, which are relevant to the configuration and build process. When creating class materials, always consult the standard itself, and not the descriptions provided below. The descriptions below explain the directories as the author would describe them to students, and may not be 100% in agreement with the language of the standard.

/usr A major section of the filesystem. It is read-only, and holds most of the programs, libraries, and other files used by the users. The only time this section of the filesystem should

be written to is when packages provided by the operating system are added or removed by the system administrator. 3 subdirectories, important to this lesson, /usr/bin, /usr/lib, and /usr/include, are located in this directory.

/usr/bin Contains binaries and other executable commands (shell scripts, Python scripts, etc.) that any user can run (no administrator privileges necessary). In general, directories named 'bin' anywhere in the filesystem will usually contain programs to be run by non-root users.

/usr/include Contains header files for libraries stored in /usr/lib.

/usr/lib Contains shared and static library files. After the introduction of 64-bit x86 processors, it became common to use this location to store 32-bit libraries, and store 64-bit libraries in /usr/lib64

/usr/lib64 Similar to /usr/lib, but contains 64-bit libraries.

/usr/share This part of the /usr filesystem contains architecture-independent files that can be shared between systems with different processor architectures. In practice this is where a lot of non-executable text files are stored, including software documentation (/usr/share/doc) and man pages (/usr/share/man)

/usr/local This directory is similar in purpose and organization to /usr, and is meant as a place where the system administrator can install additional software on the system without interfering with the software provided by the operating system.

/home This directories contains subdirectories named after each user account. These subdirectories are known as *home directories*. They are owned by the user they are named after, and the user has full read-write-execute privileges over the entire contents of this directory. Personal settings are stored here, and users can save their files here, and install and run software here, too.

The main items to emphasize here are the following:

(1) Header files for libraries included with the operating system are located in /usr/include, and others may be installed in /usr/local/include, or possibly other places, depending on where the system administrator decided to install additional software. Knowing the possible location of headers is essential to building software.
(2) Library files included with the operating system are located in /usr/lib and /lib. For 64-bit systems, libraries may also be located in /usr/lib64 and /lib64. They may be located /usr/local/lib or /usr/local/lib64 or other locations depending on where the system administrator has installed additional software. Knowing the location of libraries is essential to building software.
(3) /usr/local is a traditional location for installing additional software that is not included with the operating system. Unless another location is specified, many open-source packages will try to install in /usr/local. This means header files will be installed in /usr/local/include, library files in /usr/local/lib or /usr/local/lib64, executables in /usr/local/bin, and man pages and documentation in /usr/share.
(4) Users may install and run software from their home directory. The author often gets inquiries from users asking if it's okay or safe for them to install software they need in their own home directories. For most users, if they are installing software themselves on a shared system, like their departmental or campus cluster, they will want to install the software into their home directory.

2.2 Environment Variables

When providing this training the topic of environment variables should be discussed: what they are, and how to set and unset them. While this may seem like a very basic topic to experience Linux users, the author has seen even experienced users who have misunderstandings about how environment variables work. Some misconceptions he has encountered:

- Once an environment variable is set in one shell, it affects all other shells
- Once an environment variable is set, it is persistent and doesn't need to be set again

Whether or not this topic needs to be included depends on the knowledge level of students in the class, and me be skipped if the instructor doesn't feel this needs to be covered.

Setting certain environment variables will be helpful to the build process, and it's necessary to set environment variables when the software installation is complete in order to update PATH, MANPATH, LD_LIBRARY_PATH and other environment variables to use the new software.

As stated in the previous section, knowing the location of header files and libraries is important to the build process. If a header is located in a standard location such as /usr/include or /usr/local/include, or a library is located in /usr/lib, /usr/lib64, /lib or /lib64, they should be correctly detected during the build process and no further steps are required.

However, when software is installed in another location, builders should know that they can set environment variables to ensure they are detected properly. For GCC, the following environment variables can be used to specify the location to include directories in non-standard locations:

CPATH A list of directories to be searched for header files, independent of the programming language

C_INCLUDE_PATH A list of directories to be searched for header files, but only when compiling files written in C.

CPLUS_INCLUDE_PATH A list of directories to be searched for header files, but only when compiling files written in C++.

For library files there is only one variable to set, LIBRARY_PATH.

The values these environment variables are set to are lists of directories, using a colon (:) as the separator, and the directories are searched in order from left to right. For example, to look for C include files in $HOME/include and /opt/include before the standard locations, you can set C_INCLUDE_PATH with the following command (bash shell syntax shown).

```
$ export C_INCLUDE_PATH:$HOME/include:/opt/include
```

It's important to note that these environment variables add to the default search locations - they do not overwrite them, but they do supercede them.

The export command above is needed to make that variable available to the commands called by that shell. This is needed for

the compiler, which will be called later, to access the values of these variables.

2.3 The Compiling Process

The final prerequisite before discussing the configuration and build process is to briefly explain the compiling process. Technically, the compiling process has 4 steps:

(1) The preprocessor step
(2) The compiling step
(3) The assembly step
(4) The linking step

However, for the purpose of this instruction it might be better to simplify to 3 steps:

(1) The processor step
(2) The compiling step
(3) The linking step

The author feels this simplication is justified, since to the user, the assembly step is not normally visible, and any errors during the build process typically do not occur during the compiling or assembly steps, so no information is lost that the students would need.

In the author's experience most build errors occur for one of the following reasons:

(1) A header file cannot be found by the preprocessor
(2) A library file cannot be found by the linker
(3) The linking stage results in unresolved symbol errors, caused by libraries listed in the wrong order or a necessary library not specified on the command-line.

Errors do not normally occur during the compiling step. And when they do, they are normally harder to solve: the code is using a version of the syntax that is either too new or too old for the compiler to understand, or the code is using language extensions supported by a compiler other than the one being used. Not only are errors at this step relatively rare, they require knowledge of the programming language used, and teaching programming languages is beyond the scope of this training.

There are numerous resources that cover the compiling process, so the details of the different steps will not be discussed here, instead let's focus on how to address those 3 common errors mentioned above:

If a header file cannot be found, determine the correct location of the header file, then specify it's location on the compiler command line with the -I switch. For example, if the correct header is in $HOME/include you would specify that like this:

```
$ gcc -I $HOME/include ... ...
```

This additional directory can also be added to the preprocessor environment variables (CPATH, C_INCLUDE_PATH, etc.) as described in the previous section. However, the author recommends using the command-line whenever possible, since that consolidates all your settings in a single command. Since environment variables only affect the shell they're issued in, if a user has a lot of terminal windows open, it's very easy to type the command in a terminal window where the environment variables are not set correctly leading to unexpected errors.

For a library that cannot be found by the linker, the solution is almost the same as for a missing header file: determine the correct location of the library file using the -L switch. Assuming the library file is located in $HOME/lib, that would look like this:

```
$ gcc -L $HOME/lib ... ...
```

The LIBRARY_PATH environment variable can also be set as decribed in the previous section.

The linker can also report unresolved symbol errors. This means that either the file being compiled makes references to a function that is not provided by one of the libaries, or one of the libraries being linked to relies on a function provided by another library that is not included. There are two possible causes for this:

The first is that the libraries are not listed in the compiler command with the -l switch in the correct order. The libraries must be listed in the correct order for the linker to resolve the symbols correctly. The libraries are searched in order from left to right as they appear on the command-line specified with the -l switch. The library needing the function needs to be listed before the library providing the function. This issue can be tested easily be changed the order of the -l options and seeing if that eliminates the error. Sometime you can do an Internet search of the unresolvable symbols to determine which library provides it, and use that information to correct the order of the libraries.

Unresolvable symbol errors can also occur during linking if the library providing a symbol is omitted from the list of libraries to link. If you're not sure what library needs to be added to the command, an Internet search can often provide useful clues.

For instructional purposes, it is helpful to create simple examples to illustrate these errors, and show the student how to fix them in real time.

2.4 The Configure Script

Now that we have covered some important prerequisites to help the students understand the build process, we can take a look at how to use the `configure` script to configure the software with the correct settings for building.

As stated earlier, this `configure` script is created by the software's developer using GNU Autoconf[1]. When the source archive is uncompressed, this script will usually be located at the top level of that resulting directory.

When executed, this script will run a number of small tests to probe the environment the software is being built it to determine if the prerequisites for mandatory features and optional features are present, as well as determine how to optimize the code for the environment. For example, if it detects the processor supports FMA4 or AVX2 instructions, it may enable optimization to take advantage of those processor features.

An example of an optional feature would be a command-line program that has an optional GUI interface that uses X11. The program can still perform all of it's functions from the command-line, without the GUI. If configure can't detect the X11 headers or libraries on the system, it will print out a brief message stating that, and continue on. On the other hand, if the user specified the option to build the GUI using the appropriate switch to the `configure` script, and the X11 libraries were not found, configure would halt with an error, since `configure` would cosider this an error.

When `configure` runs, it executes a number of small, simple tests and uses the results of those tests to determine if certain features are present on the operating system. These tests include attempting to execute a command to see if it's present, trying to compile a simple program to see if the header file in the program is found by the processor, and so on.

The details of how configure works under the hood aren't critical to computational scientists at this level. What is important is how to invoke it with the correct options to build the application to meet their needs, so most of the instructional time should be spent focusing on this.

Every `configure` script has a number of options. These options will be unique to the package being built. The best way to see what configuration options are available is to run the configure script with the `--help` switch, like this:

```
$ ./configure --help | less
```

In the above example, the output of the command is piped into *less* in order to facilitate scrolling through the output, since most configure scripts will print out a lot of options. In general there are three types of configuration options available:

(1) Options to specify where the software is installed. Where library and header files will be installed, for example.
(2) Options that specify how the software is built and what features are enabled or disabled, like whether to build shared libraries or not.
(3) Environment variables that control the behavior of the configure script. These variables can be used to specify what compiler to use, or what flags should be passed to the preprocessor.

In a classroom environment, this would be a good time to display the output of a `configure --help` command for some open-source package, and discuss what some of them mean. Due to the amount of output, it's not really possible to show an example of this output in this article.

There are some configuration options that are common to all configure scripts. The most important of these is the `--prefix` option. This option tells configure in what directory the software should be installed. If this is not specified, the default is used, which is usually /usr/local. All other directories and files are then installed under here. For example if the default is used, all header files will be installed in /usr/local/include, all libraries will be installed in /usr/local/lib, and all executables will be installed in /usr/local/bin.

This is typically not what you want, since this will make it harder to keep track of what files in those directories belong to which application, and prevents having multiple versions of an application installed, since the files from whatever version is installed last will overwrite the versions installed earlier.

For software that's installed manually like this, it's much easier to put each application in it's own directory, in a path that makes it easy to understand what application is installed where. For example, if you want to install versions 1.1 and 2.2 of an application named "example", you might install them in /usr/local/example-1.1 and /usr/local/example-2.2, respectively, or /usr/local/example/1.1 and /usr/local/example/2.2, respectively.

For users installing software in their home directory, it is recommended they create a directory named 'apps' or 'software' in their home directory, and then install everything under that. For example, using the previous example but installing it in $HOME, those versions could be installed in $HOME/apps/example-1.1 or $HOME/apps/example/2.2.

Some other common options that the author likes to set are:

–disable-silent-rules This enables verbose output from the make process, which makes debugging problems much easier
–enable-shared Build shared libraries. This is usually the default, but not always, so it's easier to be explicit every time.
–enable-static Build static libraries. This is usually not the default. Since the author often build libraries for a number of users, some of whom may need/prefer static libraries, he always specifies this.

It's not possible or practical to discuss all the options specific to any software package, such as what features to enable or disable, but it would be good in a classroom environment. For example here's some of the options from the configure script for FFTW 3.3.8[6]

–enable-single compile fftw in single precision
–enable-float synonym for –enable-single
–enable-long-double compile fftw in long-double precision
–enable-quad-precision compile fftw in quadruple precision if available
–enable-sse enable SSE optimizations
–enable-sse2 enable SSE/SSE2 optimizations
–enable-avx enable AVX optimizations
–enable-avx2 enable AVX2 optimizations
–enable-avx512 enable AVX512 optimizations
–enable-avx-128-fma enable AVX128/FMA optimizations
–enable-kcvi enable Knights Corner vector instructions optimizations
–enable-altivec enable Altivec optimizations
–enable-vsx enable IBM VSX optimizations
–enable-neon enable ARM NEON optimizations

At the end of the `--help` output environment variables will be listed that can be used to influence the behavior of `configure`. This list will be different for every package, but ther are some that are common to just about every package, such as these:

CC C compiler command
CFLAGS C compiler flags
LDFLAGS linker flags, e.g. -L<lib dir> if you have libraries in a nonstandard directory <lib dir>
LIBS libraries to pass to the linker, e.g. -l<library>
CPP C preprocessor
CPPFLAGS (Objective) C/C++ preprocessor flags, e.g. -I<include dir> if you have headers in a nonstandard directory <include dir>

It is always a good idea to use these environment variables to specify which compilers you want to use, such as a C compiler with CC. This makes sure you are using the desired compiler. This is especially critical in environments where you have more than one compiler installed (Intel and GCC, for example). If you have different versions of the same compiler, specify the full path to the correct version in CC to make sure you are using the correct version. To install version 2.2 of package "example" in $HOME/apps/example/2.2,

using the gcc compiler in /usr/local/bin, and enabling some of the common options the author recommends, the configure command would look like this:

```
./configure \
--prefix=$HOME/apps/example/2.2 \
--disable-silent-rules \
--enable-shared \
--enable-static \
CC=/usr/local/bin/gcc
```

Please note in the above example the backslashes at the end of each line are to escape the newline character at the end of each line. This enables the shell to treat those multiple lines as if they are one line. There can be nothing after those backslashes other than the newline character for this to work. The author prefers this syntax, since it allows long configure lines with many options to be easier to read.

CC can be defined as an environment variable before running configure, or put on the command-line before the configure command instead of after it, but the style shown above, where CC (and other environment variables) are defined on the command-line after the configure command, is actually recommended in Section 7.1 of the GNU Coding Standards[4]

Actually running configure can take several minutes, depending on how large and complicated the package being configured is. When configure completes, it will create a number of makefiles which will the guide the actual compiling and installation of all the files with the correct settings as determined by configure.

2.5 Make

The actual command that compiles and installs the software is make[3]. Make is tool that automates the compiling of software based instructions provided to it in *makefiles*. Make is a very powerful tool that deserves it's own training session. Knowing how it works is not really relevant to students in this training, but they should have a basic understanding of what it does at a high-level, so they have an idea of what they are doing when they run the make commands below.

To actually build the software, at this point, simply run the `make` command:

```
$ make
```

When the above command completes, the next step is to actually install the software, which means to copy the files to the correct locations and make sure ownership and permissions are set correctly. That is done with the `make install` command, like this:

```
$ make install
```

2.6 Post-install tasks

`make install` is the final step in *installing* an open-source software package, but there are couple steps that need to be completed before this software can actually be used. Environment variables such as PATH and LD_LIBRARY_PATH may need to be updated to include the installation locations of the the executables and libraries. Other variables thay may need to be updated include CPATH, C_INCLUDE_PATH, and MANPATH

2.7 Resources

The Filesystem Hierarchy Standard[11] is the definitive document for where files and directories should be located on any Linux operating system. As far as standards go, it's relatively brief, and easy to understand. Since it's freely available in PDF from from the FHS website, it's recommended to distribute it to students when this topic is covered in training as part of the instructional materials.

Environment variables are a feature of the shell. The bash is the most commonly used shell on Linux. There are numerous resources online and in print covering the bash shell, but the author is not familiar enough with any of them to recommend them as a resource.

Brian Gough's *Introduction to GCC*[7] provides an excellent overview of how to use the GCC compilers with many easy to understand examples, including how to use -I, -L and -l flags as mentioned above. For an instructor preparing a course on building open source software, this is a good resource for refreshing their knowledge of GCC if necessary. It is also suitable for distributing to the students as an instructional material.

Chapter 7 of the GNU Coding Standards[4] includes a section "How Configuration Should Work" which supplements the information provided here. It provides more detail than what is provided here, which could be useful to an instructor preparing to teach this topic. This same chapter includes information on make, which may be useful if an instructor would like to cover `make` as part of this curriculum.

GNU's Autoconf Manual[2] provides some introductory material that an instructor might find helpful when preparing their curriculum. Since GNU Autoconf is a tool for developers more than users, there is no need to be an expert on using GNU Autoconf to teach this material.

While the author chose not to discuss make in this curriculum, other instructors may feel differently, GNU's online documenation for `make`[3] is an excellent source for information about make.

3 HOW TO PACKAGE OPEN SOURCE SOFTWARE

The author is admittedly not an expert on packaging open-source software himself. In fact, he's never done it, but as someone who has built many open-source packages over the years, he has seen what makes one package easier or harder to install than another. In the remainder of this section, he recommends some best practices that should be employed by computational scientists when developing software to make it as easier as possible for other to use their software to make its use as widespread as possible.

All packages should have a clearly defined version number. This version information should be easily identifiable on the website for the software, in the source code archive, and after installation by running some command with the `--version` option. It is very difficult for users to know if they're using the latest version or not without this information. This information is often necessary when reporting a bug or determining if the current version in use has certain features. There are different version numbering conventions in use, and pros and cons of each convention could be a topic of discussion in the training.

Software developers should make sure the files from their software packages are put in locations that are consistent with with

current standards and conventions. The FHS[11], discussed in the previous section, is the definitive source for this information.

How to properly build the software package should be well documented both on the website for the software, as well as in a README or INSTALL file included in the source code archive. It should never be assumed that it is obvious how a package should be built.

Adhering to standards or commonly used conventions can make maintaining code easier, and make collaboration easier. If responsibility for maintaining the code is ever transferred to someone else, adhering to well-known conventions or standards will make that transition easier. The GNU Coding Standards[4] is one set of standards that could be used for this.

Automatic configuration tools like GNU Autoconf[1] should employed to make it easier for users to build the software correctly and as easily as possible. The previous section of this paper focused on building code with an GNU Autoconf-generated configure script because it is by far the most common tool for doing this. However, there are other tools out there that serve the same purpose, such as CMake[10] and Scons[5]. These tools can be discussed, but the author recommends focusing on GNU Autoconf, since that is currently the *de facto* standard for this.

4 CONCLUSION

This paper has identified building open-source software as a skills gap for computational scienists. It has provided an outline of what topics need to be taught to computational scientists in a logical order to train them to do this. The author has provided references to some of the topics discussed that could be used to develop training materials, or distributed directly to students as part of the training materials.

For computational scientists developing software, packaging this software in a way that makes it easier for users to build that software has also been identified as skills gap. Although the author is not an experienced software developer himself, he provided several best practices that should be taught to make it easier for others to build and use their software.

The author has successfully used the outline presented here to teach building of open-source software to junior coworkers. He hopes to continue refining this instruction and eventually include it HPC training workshops, such as the Software or HPC Carpentry programs mentioned here.

REFERENCES

[1] Free Software Foundation. 2009. GNU Autoconf. (2009). Retrieved September 28, 2018 from https://www.gnu.org/software/autoconf/autoconf.html

[2] Free Software Foundation. 2012. GNU Autoconf - Creating Automatic Configuration Scripts. (2012). Retrieved September 28, 2018 from https://www.gnu.org/software/autoconf/manual/index.html

[3] Free Software Foundation. 2016. GNU Make. (May 2016). Retrieved September 28, 2018 from https://www.gnu.org/software/make/

[4] Free Software Foundation. 2018. GNU Coding Standards: Configuration. (Aug. 2018). Retrieved September 28, 2018 from https://www.gnu.org/prep/standards/

[5] SCons Foundation. 2018. SCons: A software construction tool. (2018). Retrieved September 28, 2018 from https://socns.org/

[6] Matteo Frigo and Steven G. Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[7] Brian Gough. 2004. An Introduction to GCC. (Feb. 2004). Retrieved September 28, 2018 from http://www.network-theory.co.uk/docs/gccintro/

[8] https://hpc carpentry.github.io. 2017. About HPC Carpentry. (2017). Retrieved September 28, 2018 from https://hpc-carpentry.github.io/about

[9] https://sns.ias.edu. 2009. Prospects in Theoretical Physics: Computational Astrophysics. (July 2009). Retrieved September 28, 2018 from http://www.sns.ias.edu/pitp2/index2009final.html

[10] Kitware. 2018. CMake. (2018). Retrieved September 28, 2018 from https://cmake.org/

[11] Daniel Quinlan Rusty Russell and Christopher Yeoh. 2004. Filesystem Hierarchy Standard. (Jan. 2004). Retrieved September 28, 2018 from http://www.pathname.com/fhs/pub/fhs-2.3.pdf

[12] Trace K. Teal, Karen A. Cranston, Hilmar Lapp, Ethan White, Greg Wilson, Karthik Ram, and Aleksandra Pawlik. 2015. Data Carpentry: Workshops to Increase Data Literacy for Researchers. *International Journal of Digital Curation* 10 (02 2015).

[13] Greg Wilson. 2006. Software Carpentry: Getting Scientists to Write Better Code by Making Them More Productive. *Computing in Science & Engineering* 8, 6 (November–December 2006), 66–69. https://doi.org/10.1109/MCSE.2006.122 Summarizes the what and why of Version 3 of the course.

[14] Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K. Teal. 1996. Good enough practices in scientific computing. *IEEE Computational Science and Engineering* 3, 2 (Summer 1996), 46–65. https://doi.org/10.1109/99.503313

[15] G.V. Wilson, R.H. Landau, and S.McConnell. 1996. What Should Computer Scientists Teach to Physical Scientists and Engineers? *IEEE Computational Science and Engineering* 3, 2 (Summer 1996), 46–65. https://doi.org/10.1109/99.503313