Managing Personal Software Installations

Afif Elghraoui

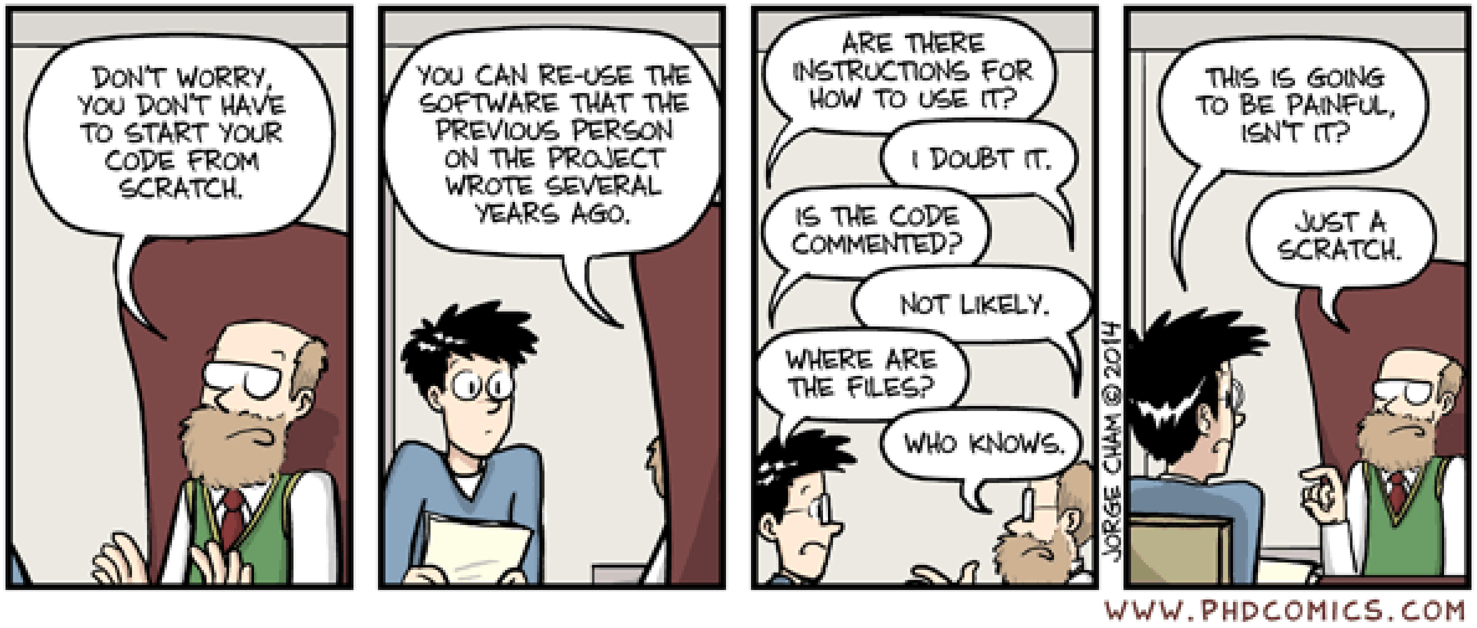
NIH HPC staff

staff@hpc.nih.gov

NIH

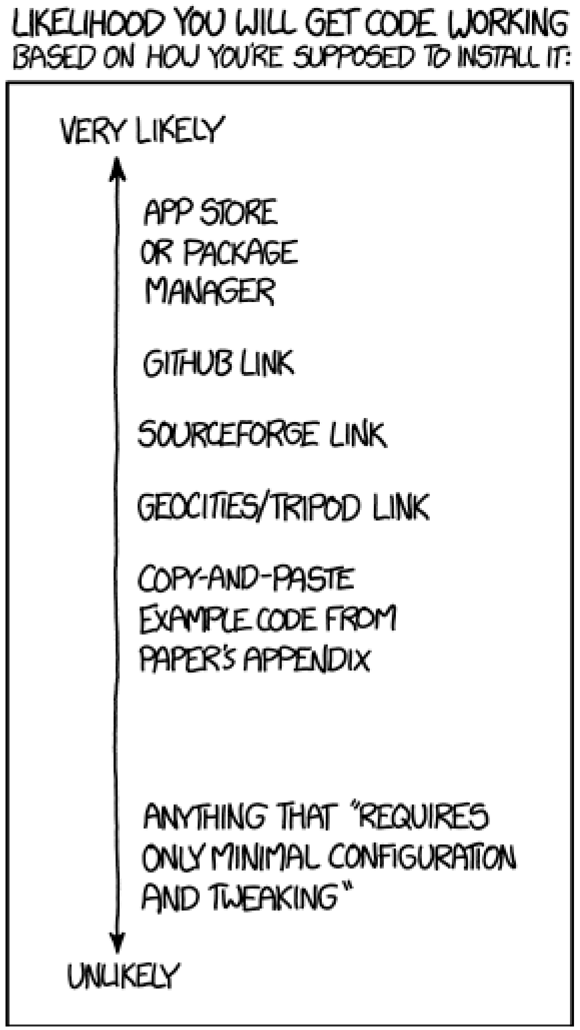
BIOWULF
HIGH PERFORMANCE COMPUTING AT THE NIH

When using code



DON'T WORRY, YOU DON'T HAVE TO START YOUR CODE FROM SCRATCH.
YOU CAN RE-USE THE SOFTWARE THAT THE PREVIOUS PERSON ON THE PROJECT WROTE SEVERAL YEARS AGO.
ARE THERE INSTRUCTIONS FOR HOW TO USE IT?
I DOUBT IT.
IS THE CODE COMMENTED?
NOT LIKELY.
WHERE ARE THE FILES?
WHO KNOWS.
THIS IS GOING TO BE PAINFUL, ISN'T IT?
JUST A SCRATCH.
JORGE CHAM © 2014
WWW.PHDCOMICS.COM

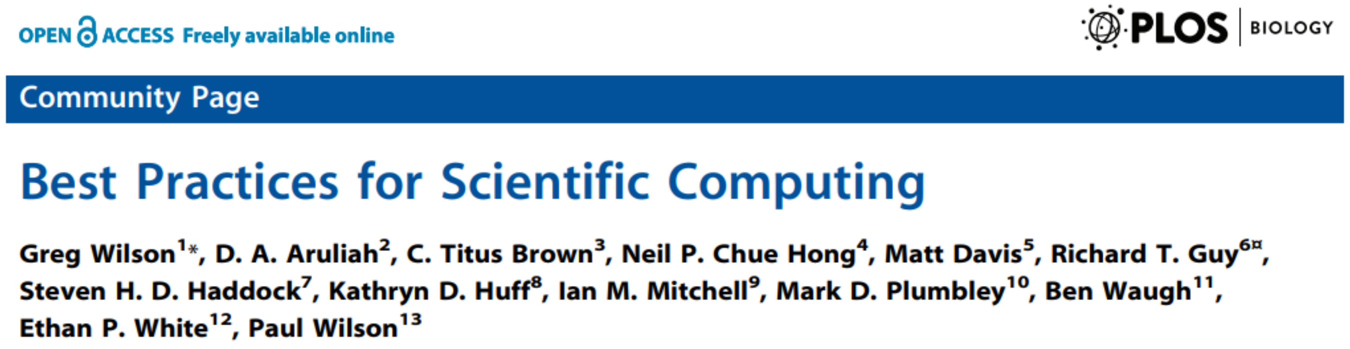
http://phdcomics.com/comics.php?f=1689



LIKELIHOOD YOU WILL GET CODE WORKING BASED ON HOW YOU'RE SUPPOSED TO INSTALL IT:
VERY LIKELY
APP STORE OR PACKAGE MANAGER
GITHUB LINK
SOURCEFORGE LINK
GEOCITIES/TRIPOD LINK
COPY-AND-PASTE EXAMPLE CODE FROM PAPER'S APPENDIX
ANYTHING THAT "REQUIRES ONLY MINIMAL CONFIGURATION AND TWEAKING"
UNLIKELY

https://xkcd.com/1742/

When writing code



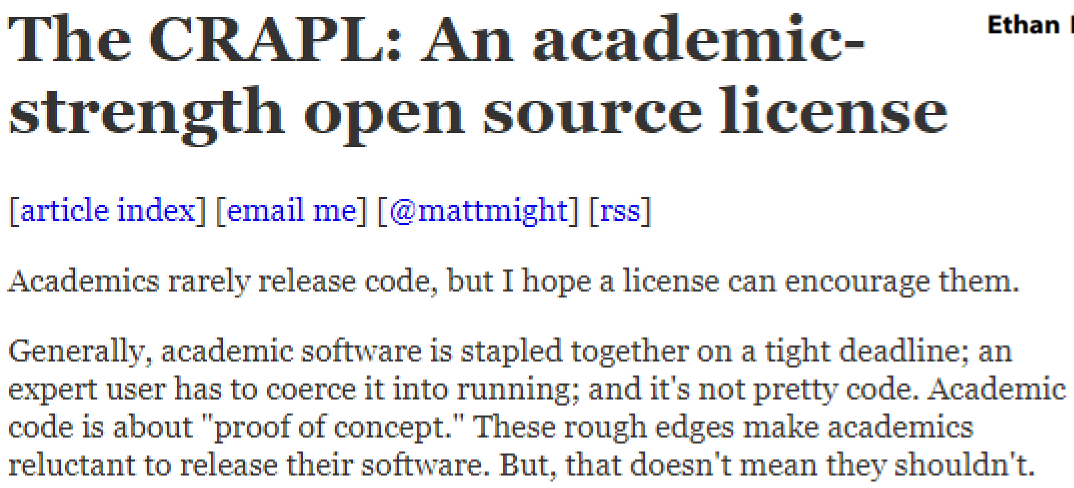
OPEN ACCESS Freely available online

PLOS | BIOLOGY

Community Page

Best Practices for Scientific Computing

Greg Wilson[1]*, D. A. Aruliah[2], C. Titus Brown[3], Neil P. Chue Hong[4], Matt Davis[5], Richard T. Guy[6¤], Steven H. D. Haddock[7], Kathryn D. Huff[8], Ian M. Mitchell[9], Mark D. Plumbley[10], Ben Waugh[11], Ethan P. White[12], Paul Wilson[13]

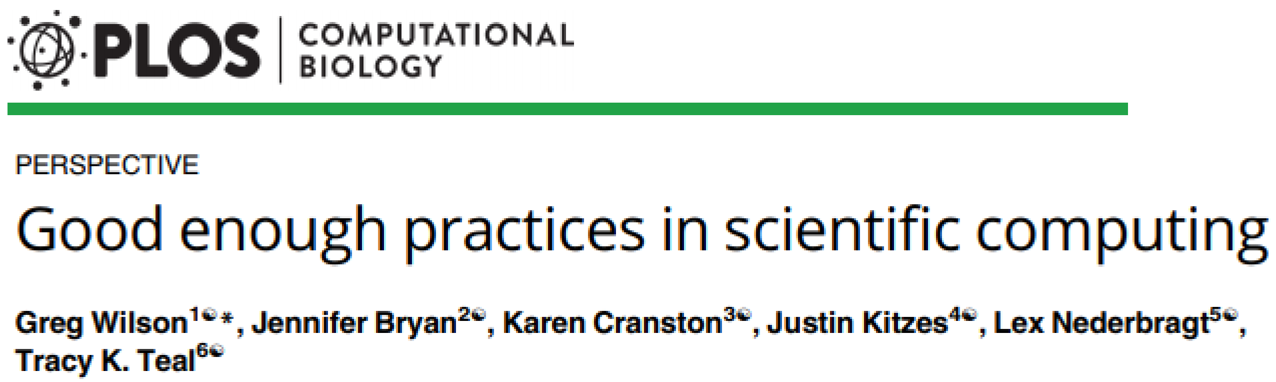


The CRAPL: An academic-strength open source license

[article index] [email me] [@mattmight] [rss]

Academics rarely release code, but I hope a license can encourage them.

Generally, academic software is stapled together on a tight deadline; an expert user has to coerce it into running; and it's not pretty code. Academic code is about "proof of concept." These rough edges make academics reluctant to release their software. But, that doesn't mean they shouldn't.



PLOS | COMPUTATIONAL BIOLOGY

PERSPECTIVE

Good enough practices in scientific computing

Greg Wilson[1]*, Jennifer Bryan[2], Karen Cranston[3], Justin Kitzes[4], Lex Nederbragt[5], Tracy K. Teal[6]

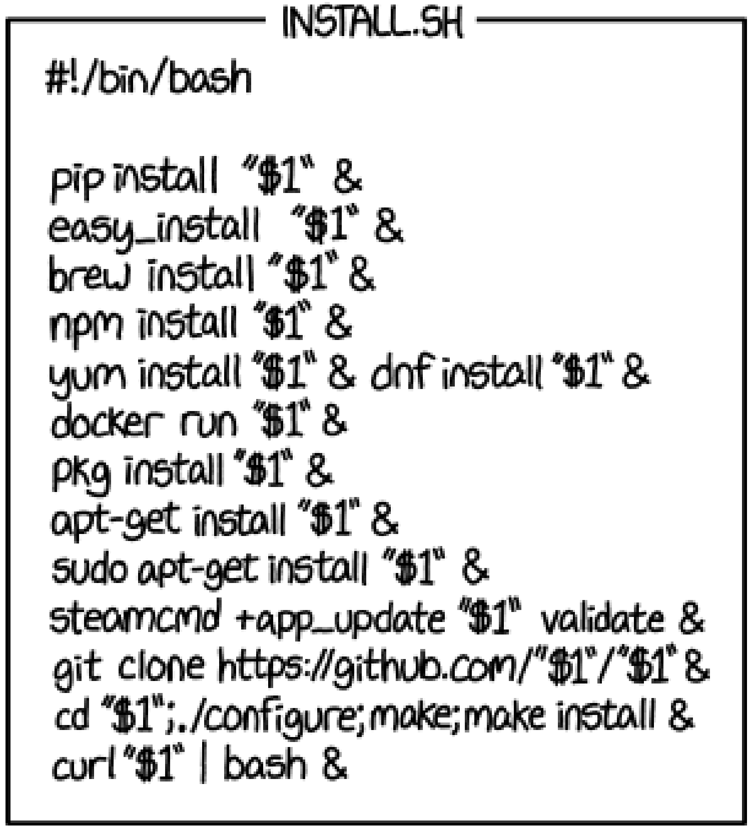
Application support on Biowulf

The HPC Staff will maintain software installations if

- we expect that they'll be useful to more than one or two people – not obscure/unpublished/obsolete
- the application can be run without elevated privileges (like requiring write access to the installation directory).

Installation Methods

- System Package Manager (apt, yum, dnf, …)
  - Packages are built with a consistent set of libraries
  - Potentially limited selection of packages and package versions.
  - Requires root access in most cases (but becomes an option if you're using containers!)
- User-level Package Managers (conda, homebrew, nix, guix, …)
  - No special permissions needed
  - Automatically install almost any version of any package…and maybe get a bunch of conflicts as a result of the complexity.
  - Some are language-specific (pip, gem, cpan, …) and won't handle dependencies if they're implemented in a different language.
- Manual
  - "Dependency hell"
  - Can be messy and cause interesting problems if you're not careful.



```
#!/bin/bash

pip install "$1" &
easy_install "$1" &
brew install "$1" &
npm install "$1" &
yum install "$1" & dnf install "$1" &
docker run "$1" &
pkg install "$1" &
apt-get install "$1" &
sudo apt-get install "$1" &
steamcmd +app_update "$1" validate &
git clone https://github.com/"$1"/"$1" &
cd "$1";./configure;make;make install &
curl "$1" | bash &
```

https://xkcd.com/1654

Existing Biowulf guides to user-level package management and installation

- Conda

  Language-independent, but started with Python.

  Guide to using conda on Biowulf: https://hpc.nih.gov/apps/python.html#envs

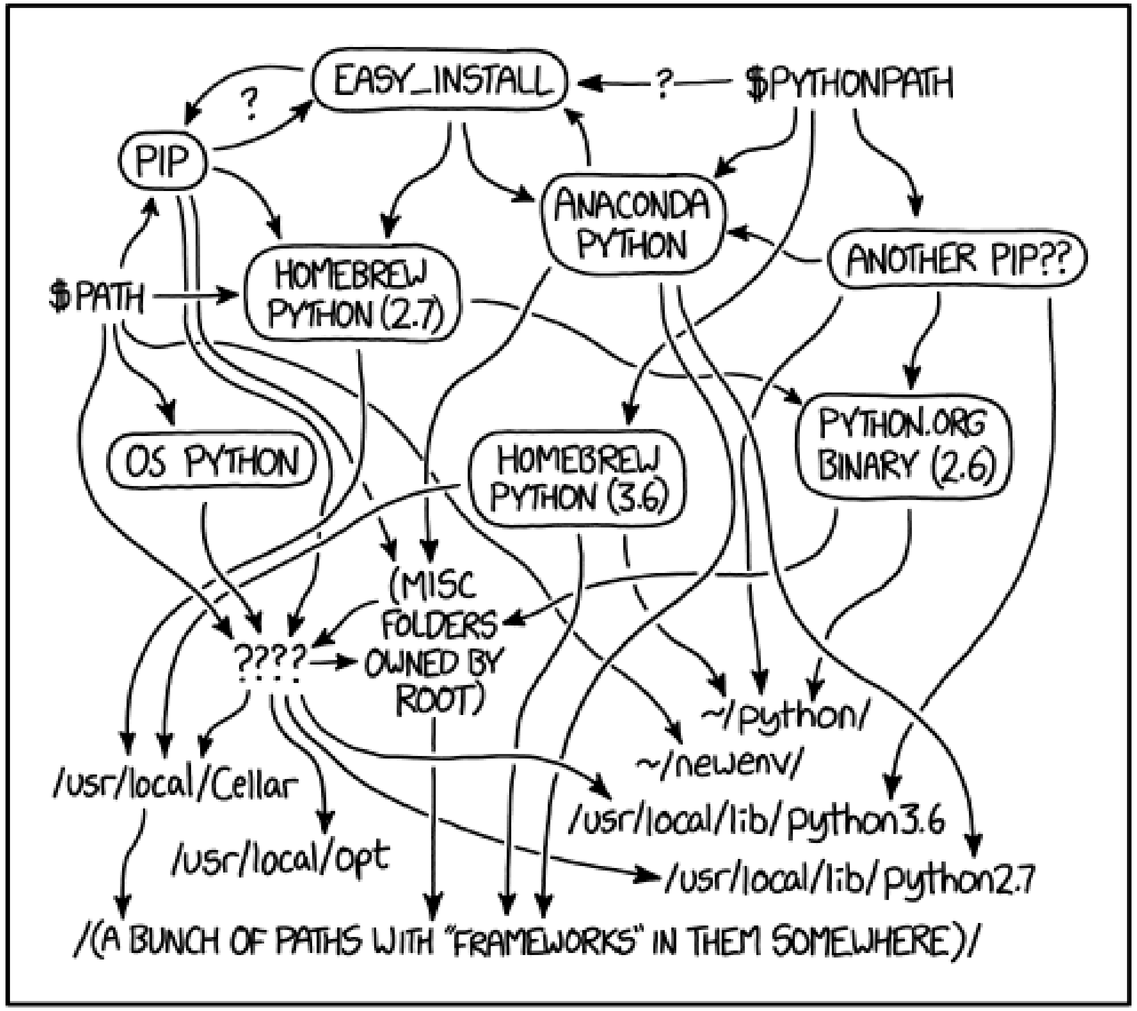
- Personal R package installations

  R's `install.packages()` can be used as a regular user– it will fall back to installing in your home directory after it finds out it's not allowed to write in the system directory.

  Personal R packages on Biowulf: https://hpc.nih.gov/apps/R.html#install

- Guide to personal environment modules: https://hpc.nih.gov/apps/modules.html#personal

Keeping things organized



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

https://xkcd.com/1987/

Filesystem Hierarchy Standard (FHS) and the installation prefix

- Specifies layout of the system directory tree.
- Executive summary, as pertains to software installation:

  Most of these directories are seen in `/`, `/usr/`, and `/usr/local/`
  - `bin/` – executables
  - `libexec/` – helper commands (run by commands in bin/, not by users directly)
  - `include/` – header files (like for C, C++ libraries)
  - `lib/` – software libraries
  - `etc/` – configuration files
  - `share/` – architecture-independent files
    - man/ – manual pages

https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html

Review(?) - environment variables

- See them with the `env` or `printenv` commands.
- Some are used by the system, like `$PATH`, and others, like `$http_proxy`, are conventions and might be respected or ignored depending on the program.

```
$ MYVAR=foo
$ echo $MYVAR
foo
$ ./printmyvar  # if you create this script that just runs ‘echo $MYVAR’
$
$ unset MYVAR
$ echo $MYVAR
$ MYVAR=foo ./printmyvar
foo
$ echo
```

NIH

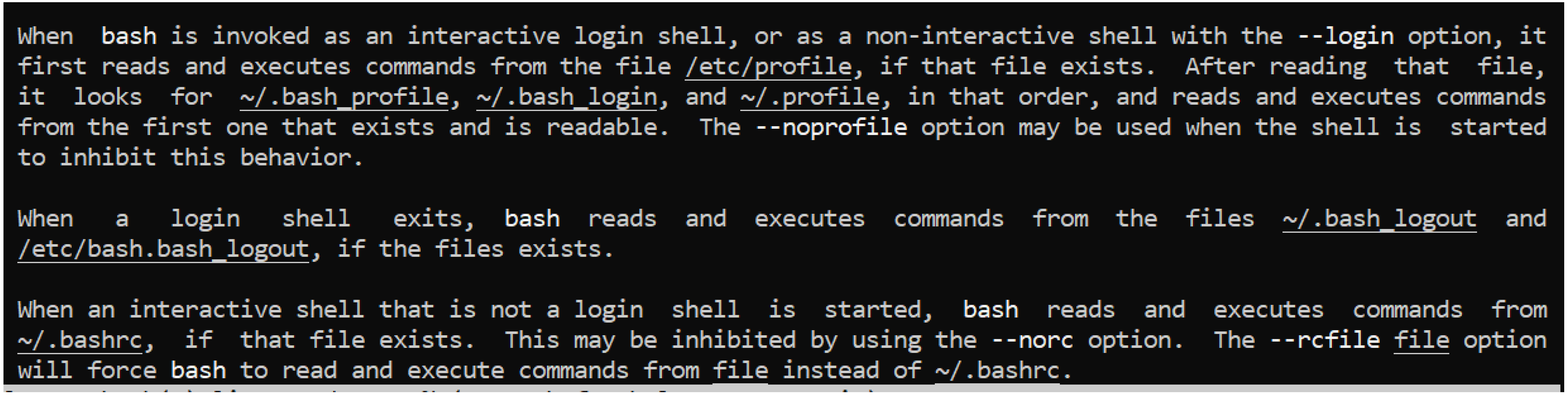
BIOWULF
HIGH PERFORMANCE COMPUTING AT THE NIH

“dotfiles”

- Personal configurations stored in files/directories beginning with a dot (making them hidden) in your home directory.
  - Some applications started following the [XDG base directory specification](), which specifies `~/.config/` as the default directory for such files.
- Aside – ideas for keeping your dotfiles under version control: https://dotfiles.github.io
- **Making persistent customizations to your environment will involve editing your shell’s resource file– `~/.bashrc` in the case of bash, the default shell.**

~/.bashrc

- From `bash(1)`, section INVOCATION:



```
When  bash is invoked as an interactive login shell, or as a non-interactive shell with the --login option, it
first reads and executes commands from the file /etc/profile, if that file exists.  After reading  that  file,
it  looks  for  ~/.bash_profile, ~/.bash_login, and ~/.profile, in that order, and reads and executes commands
from the first one that exists and is readable.  The --noprofile option may be used when the shell is  started
to inhibit this behavior.

When   a   login   shell   exits,  bash  reads  and  executes  commands  from  the  files  ~/.bash_logout  and
/etc/bash.bash_logout, if the files exists.

When an interactive shell that is not a login  shell  is  started,  bash  reads  and  executes  commands  from
~/.bashrc,  if  that file exists.  This may be inhibited by using the --norc option.  The --rcfile file option
will force bash to read and execute commands from file instead of ~/.bashrc.
```

Special Environment Variables for Running Applications

- `PATH`

  List of directories to look for executables
- `LD_LIBRARY_PATH`

  List of directories to look for shared libraries. Not needed if library paths were built into the software was built with rpath
- `MANPATH`

  List of directories to look for man pages
- Some language specific variables:
  - `R_LIBS`, `R_LIBS_USER`
  - `PYTHONPATH`
  - `PERL5LIB`
  - …

Special Environment Variables for Building Applications

- `LIBRARY_PATH`
  search path for libraries to link to
- `LD_RUN_PATH`
  Library paths to hard-code into the resulting binary as rpath. Alternative to setting them via the command-line flag `-Wl,-rpath`, as some build systems do. The variable is ignored if the command-line flag is used [citation-needed].
- `CPATH`
  search path for header files
- `CFLAGS CXXFLAGS`
  C and C++ compiler flags
- `CPPFLAGS`
  C pre-processor flags. (Include paths *could* be passed here as `-I/path/to/include` if the build system honors it, rather than using `CPATH`).
- `LDFLAGS`
  linker flags. (Library paths *could* be passed here as `-L/path/to/lib` if the build system honors it, rather than using `LIBRARY_PATH`).

https://gcc.gnu.org/onlinedocs/gcc/Environment-Variables.html

Build Systems - Autotools

- Generally the most straightforward for users to deal with.
- Characterized by the existence of a `configure` script and a template Makefile, `Makefile.am`.
- Environment variables previously mentioned are respected.
- General process

```
./configure [configure options]
make
make check # if test suite exists
make install
```

Autotools package example: GNU hello

```
wget http://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz
tar -xf hello-2.10.tar.gz
cd hello-2.10
mkdir -p /data/$USER/opt/hello
module load gcc    # use a modern compiler
./configure --prefix /data/$USER/opt/hello/2.10
make
make check
make install
```

Private modulefile for `hello`

Create the file `~/modulefiles/hello/2.10.lua` with the following contents:

```
local basedir = “/data/” .. os.getenv(“USER”) .. “/opt/” .. myModuleFullName()

prepend_path(“PATH”, basedir .. “/bin”)
prepend_path(“MANPATH”, basedir .. “/share/man”)
```

Using your personal modules

```
module use ~/modulefiles    # can add this line to ~/.bashrc

module load hello

module list # see what's been loaded

man hello

which hello

hello
```

Build Systems - Cmake

- Cross-platform Make. It can set up builds for native Windows, too, unlike the Autotools.
- Characterized by the existence of a `CMakeLists.txt` file.
- Needs the `cmake` program installed to be able to configure the build (Autotools just uses the shell).
- General procedure:

```
mkdir build && cd build
cmake [config options] ..
make
make test # if test suite exists
make install
```

Cmake example (with a twist): kallisto

```
wget https://github.com/pachterlab/kallisto/archive/v0.46.0.tar.gz
tar –xf v0.46.0.tar.gz
cd kallisto-0.46.0
module purge # start fresh following the previous example
mkdir /data/$USER/opt/kallisto # create our separate installation prefix for this program
module load gcc cmake
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=/data/$USER/opt/kallisto/0.46.0 ..
```

Cmake example (with a twist): kallisto

Not so fast

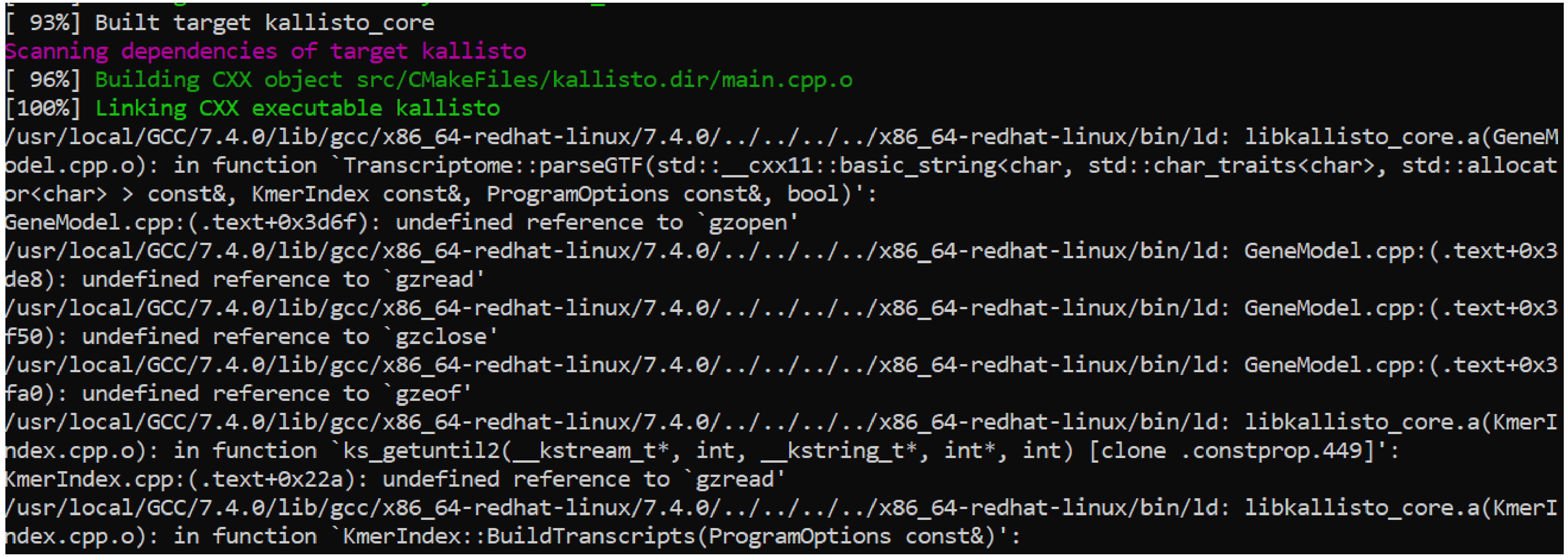
```
CMake Error at /usr/local/Cmake/3.9.5/share/cmake-
3.9/Modules/FindPackageHandleStandardArgs.cmake:137 (message):        Could NOT find HDF5 (missing:
HDF5_LIBRARIES HDF5_INCLUDE_DIRS)                                     Call Stack (most recent call first):
/usr/local/Cmake/3.9.5/share/cmake-3.9/Modules/FindPackageHandleStandardArgs.cmake:377
(_FPHSA_FAILURE_MESSAGE)       /usr/local/Cmake/3.9.5/share/cmake-3.9/Modules/FindHDF5.cmake:839
(find_package_handle_standard_args)              src/CMakeLists.txt:30 (find_package)
```

Load hdf5 and try again—

```
module load hdf5
cmake -DCMAKE_INSTALL_PREFIX=/data/$USER/opt/kallisto/0.46.0 ..
make
```

Cmake example (with a twist): kallisto

More trouble!



```
[ 93%] Built target kallisto_core
Scanning dependencies of target kallisto
[ 96%] Building CXX object src/CMakeFiles/kallisto.dir/main.cpp.o
[100%] Linking CXX executable kallisto
/usr/local/GCC/7.4.0/lib/gcc/x86_64-redhat-linux/7.4.0/../../../../x86_64-redhat-linux/bin/ld: libkallisto_core.a(GeneM
odel.cpp.o): in function `Transcriptome::parseGTF(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocat
or<char> > const&, KmerIndex const&, ProgramOptions const&, bool)':
GeneModel.cpp:(.text+0x3d6f): undefined reference to `gzopen'
/usr/local/GCC/7.4.0/lib/gcc/x86_64-redhat-linux/7.4.0/../../../../x86_64-redhat-linux/bin/ld: GeneModel.cpp:(.text+0x3
de8): undefined reference to `gzread'
/usr/local/GCC/7.4.0/lib/gcc/x86_64-redhat-linux/7.4.0/../../../../x86_64-redhat-linux/bin/ld: GeneModel.cpp:(.text+0x3
f50): undefined reference to `gzclose'
/usr/local/GCC/7.4.0/lib/gcc/x86_64-redhat-linux/7.4.0/../../../../x86_64-redhat-linux/bin/ld: GeneModel.cpp:(.text+0x3
fa0): undefined reference to `gzeof'
/usr/local/GCC/7.4.0/lib/gcc/x86_64-redhat-linux/7.4.0/../../../../x86_64-redhat-linux/bin/ld: libkallisto_core.a(KmerI
ndex.cpp.o): in function `ks_getuntil2(__kstream_t*, int, __kstring_t*, int*, int) [clone .constprop.449]':
KmerIndex.cpp:(.text+0x22a): undefined reference to `gzread'
/usr/local/GCC/7.4.0/lib/gcc/x86_64-redhat-linux/7.4.0/../../../../x86_64-redhat-linux/bin/ld: libkallisto_core.a(KmerI
ndex.cpp.o): in function `KmerIndex::BuildTranscripts(ProgramOptions const&)':
```

Cmake example (with a twist): kallisto

Lots of gz errors. Googling the errors indicates that the problem is that our zlib is too old.

```
module load zlib
cd .. && rm -rf build && mkdir build && cd build  # start a fresh build
cmake \
  -DCMAKE_INSTALL_PREFIX=/data/$USER/opt/kallisto/0.46.0 \
  `# $ZLIB_LIBS is defined by the biowulf zlib module. You can see this by running module show zlib` \
  -DCMAKE_EXE_LINKER_FLAGS="$ZLIB_LIBS" \
 `# setting rpath on the command line removes LD_RUN_PATH from consideration` \
  -DCMAKE_SKIP_RPATH=YES \
  ..
make
make install
```

Cmake example (with a twist): kallisto

Create our modulefile (almost identically as before, but there are no manpages this time) `~/modulefiles/kallisto/0.46.0.lua` with the following contents:

```
local basedir = “/data/” .. os.getenv(“USER”) .. “/opt/” ..
myModuleFullName()

prepend_path(“PATH”, basedir .. “/bin”)
```

Cmake example (with a twist): kallisto

Kallisto is already installed centrally on biowulf, but your personal module will be given precedence since it appears first in the `MODULEPATH`.

```
# get out of the build directory and unload the build modules
cd; module purge

module avail kallisto # which versions do you see?

module load kallisto
module list    # which version was loaded?
which kallisto # where is our installed kallisto?
kallisto -h
```

Congratulations!