Panel discussion on “Legacy codes in astrophysics”

- tactics (the lecturers)
  - the branch of military science dealing with detailed maneuvers to achieve objectives set by strategy
- strategy
  - the branch of military science dealing with military command and the planning and conduct of a war
- anecdotes from Peter
- complaints
  - what are some of the bad features of the codes used in this school?

NORMAN

- Developing it
  - Don’t do it unless you’re committed to supporting it
  - Identify your feature set and design goals
  - Find a meticulous grad and leave him/her alone
  - Encourage “best practices” coding (exemplars)
  - Elegance vs. performance trade-offs (you can have both. See http://www.c3.lanl.gov/poosc/)
  - Modular design; unit testing
  - Verification test suite
  - 2-yr to develop; rewrite after 10
- Supporting it
  - Don’t expect grant funding specifically for development and support (piggy back on science grants)
  - good documentation mitigates support burden
  - Establish user and developer email lists
  - Establish mechanisms for uptake of user-developed code
  - Nightly regression testing (http://lca.ucsd.edu/projects/lcatest)

PRETORIUS

- Keep it simple
  - computational science uses code as a means-to-an-end; the code is not the end
    - avoid the temptation of using the newest languages and/or bleeding-edge features of compilers, forcing the code to adhere to strict programming paradigms, etc.
  - write new code in pieces as small as possible, thoroughly testing them before continuing to the next piece
    - never add a new feature/parameter/etc. that you will not be able to test and use immediately

RICHARDSON

- Do…
  - Use comments. Liberally. Despite what Brian Kernighan says. Really.
  - When hacking (yes, it happens), ALWAYS add a searchable comment, like /*DEBUG!*/.
  - Use #include and make, even for a simple code.
- Don't…
  - Use global variables. Ever. This will help you program in modular style. Trust me.
  - Forget to back up your source code regularly. Twice. On disks that are far away from each other. Preferably in different countries.

Oh yeah, and when testing, use a *soft link* to the executable, or force make install…

SPITKOVSKY

- Keep a record of what you tried. Use a rich text document (or Word, or PPT, etc), where you can quickly paste a screenshot of the plot window. This is the fastest way to leave a trace. Try Evernote.com for diary-type notes.

- Use meaningful names for directories, more descriptive than run1/ or even run35.2/ .

- Don’t be too evangelical. There are much worse things in life than a few global variables.

- Think before relying on external libraries. You will be recompiling them a lot on many platforms.

- Explain how to use the code to people and ask your first users to write the user manual as a wiki. This makes for an amusing and educational read.

SPRINGEL

- Use a highly portable language (C comes to mind), and stick to its standard.
- Always compile with “full warnings” (-Wall) enabled, and address all issues until the code compiles with no warnings.
- Use a version control system (e.g. subversion). The repository should be on a server in your institute that has nightly back-ups.
- For science production runs, always create a separate copy of the source code and treat it as part of the input/output data. This source is part of the simulation, and should not be changed unless you discard the simulation. (long-term reproducibility!)
- Do not rely on compiler optimizations to make your code fast – rather focus on writing intrinsically efficient code.
- Always put in error checks when operating system functions are called, especially for dynamic memory allocation and I/O.
- Use meaningful variable names, and a consistent indention convention.
- Difficult code is best written incrementally, interleaved with frequent tests.
- This may seem very obvious... work with an editor that does syntax highlighting.

STONE + TEUBEN

- Modularity: makes extensions to code easier (also think of other codes using some of your engines; python hooks, e.g. MUSE)
- Ease-of-use:
  - adopt portable configuration tools (configure, etc.)
  - flexible variety of output files (that don't depend on external libraries!)
  - Input files have intuitive format enabled by special-purpose parser.
- Portability ensured by:
  - Strict adherence to ANSI standards (don't use language extensions!)
  - No reliance on external libraries (except when absolutely necessary, e.g. parallelization with MPI)
- Performance: is memory or cpu (including cache access) limiting factor?
- Testing: regressions testing and test problems (benchmark). Also good to add benchmark/tracers in your code in DEBUG mode. Handy to have sample data.
- Source Code Management (SCM): **SVN** with **trac** integrated very useful. Also handles dealing user support, e.g. bugs.
- Documentation: Users and/or Programmers Guide. Sometimes self-generated via e.g. doxygen.

“Five Golden Rules of Installing Software”

or:

why can't they just not use some self-extracting and installing binary blurb?

Or:

why don't they just write this for windows?

1. don't stress your sysadmin

- RTFM (Read The Friendly Manual)
- GIYF (Google Is Your Friend)

2. Don't read the last error

- You should have scrollbars or look at the log file where the S*&$^@! hit the fan

(t)csh shell: command >& logfile

(ba)sh shell: command > logfile 2>&1

3. Parsing Errors the Right Way

- “/usr/bin/ld: cannot find -lfoobar”
  - libfoobar.a or libfoobar.so not found or not in the right path of one of your -L compiler directives
  - Use “locate libfoobar” to see if you have it
  - Maybe incomplete install (e.g. -devel missing)
    - (debian) dpkg -S /usr/lib/libfoobar.so
    - (redhat) rpm -qf /usr/lib/libfoobar.so
    - (mac) 1-800-eat-apple

4. Environment Variables

- $PATH:
- $LD_LIBRARY_PATH (linux, solaris) or $DYLD_LIBRARY_PATH (darwin)
- $CFLAGS, $FFLAGS
- $CC, $CXX, $F77 (configure uses them)

5. Unix Commands

- ldd
- nm
- hexdump -C
- lfind : alias lfind 'find . -name *\!** -print'
- “gcc –version” or “gcc --help”

Anecdotes

(hey, it works for me.....)

- Space in a directory name: configure really bombed out with a seeminly innocent statement
- (on a mac) incompatible format for .o file: turns out the intel and gcc compiler were mixed and matched wrong.
- Older version of same library in /usr/local/lib which now was in /usr/lib; same for headers
- **ld: command not found.** You are probably on a spanking new mac and did not load Xcode!

Strategic issues

- is writing and maintaining a legacy code bad for your career?
- what is the right balance between code development and science?
- how do you make sure that your code is widely used?
- at what stage should a code be made public?
  - should funding agencies require this?
- who owns the code?
  - the author?
  - the employer?
  - the funding agency?
- is there adequate testing and validation of codes in astrophysics? If not, how can the situation be fixed?