Software Engineering Practices for Python

Coding Experiences

- Good development practices help with the following situations:
  - *You swear that the code worked perfectly 6 months ago*, but today it doesn't, and you can't figure out what changed
  - *Your research group is all working on the same code*, and you need to sync up with everyone's changes, and make sure no one breaks the code
  - *Your code always worked fine on machine X*, but now you switch to a new system/architecture, and you code gives errors, crashes, ...
  - *Your code ties together lots of code*: legacy code from your advisor's advisor, new stuff you wrote, all tied together by a driver. The code is giving funny behavior sometime—how do you go about debugging such a beast?

Software Engineering Practices

- We'll look at some basic python style guidelines and some tools that help with the development process
  - Also helps reproducibility of your science results
  - You can google around for specific details, more in-depth tutorials, etc.

Software Engineering Practices

- Some basic practices that can *greatly* enhance your ability to write maintainable code
  - Version control
  - Documentation
  - Testing procedures
  - For compiled languages, I would add Makefiles, profilers, code analysis tools (like valgrind)
- Already discussed: PEP 8 (coding standards)
  - Helps you interact with a distributed group of developers—everyone writes code with the same convention

Python Style

- The recommended python style is described in a “Python Enhancement Proposal”, PEP-8
  - http://legacy.python.org/dev/peps/pep-0008/
  - Based on the idea that “*code is read much more often than it is written*”
- Some highlights:
  - Indentation should use 4 spaces (no tabs)
  - Lines should be less than 79 characters
    - Continuation via `'\'` or `()`
  - Classes should be capitalized of form `MyClass`
  - Function names, objects, variables, should be lower case, with `_` separating words in the name
  - Constants in `ALL_CAPS`

with

- `with` uses a context manager that has a special `__enter__` and `__exit__` function.
- Simplifies some common constructions
- Eg.

```
with open("x.txt") as f:
    data = f.read()
    # do something with data
```

  - This will open the file, return the file object `f`, allow you to work with it, and close the file automatically when this block is over

http://effbot.org/zone/python-with-statement.htm

Coding Style

- Don't make assumptions
  - For `if` clauses, have a default block (else) to catch conditions outside of what you may have expected
  - Use `try/except` to catch errors
- Use functions/subroutines for repetitive tasks
  - Check return values for errors
  - Use well-tested libraries instead of rolling your own when possible

Version Control

- Old days: create a tar file with the current source, mail it around, manually merge different people's changes…
- Version control systems keep track of the history of changes to source code
  - Logs describe the changes to each file over time
  - Allow you to request the source as it was at any time in the past
  - Multiple developers can share and synchronize changes
    - Merges changes by different developers to the same file
    - Provide mechanisms to resolve conflicting changes to files
  - Provide mechanisms to create a branch to develop new features and then merge it back into the main source.

Version Control

- Even for a single developer version control is a great asset
  - Common task: you notice that your code is giving different answers/behavior than you've seen in the past
    - Check out an old copy where you know it was working
    - Bisect the history between the working and broken dates to pin down the change
- Can also use it for papers and proposals—all the authors can work on the same LaTeX source and share chages
- All of these slides are stored in version control—let's me work on them from anywhere easily
  - Fun trick: LibreOffice files are zipped XML, but you can have it store the output as uncompressed “flat” XML files (.fodp instead of .odp)

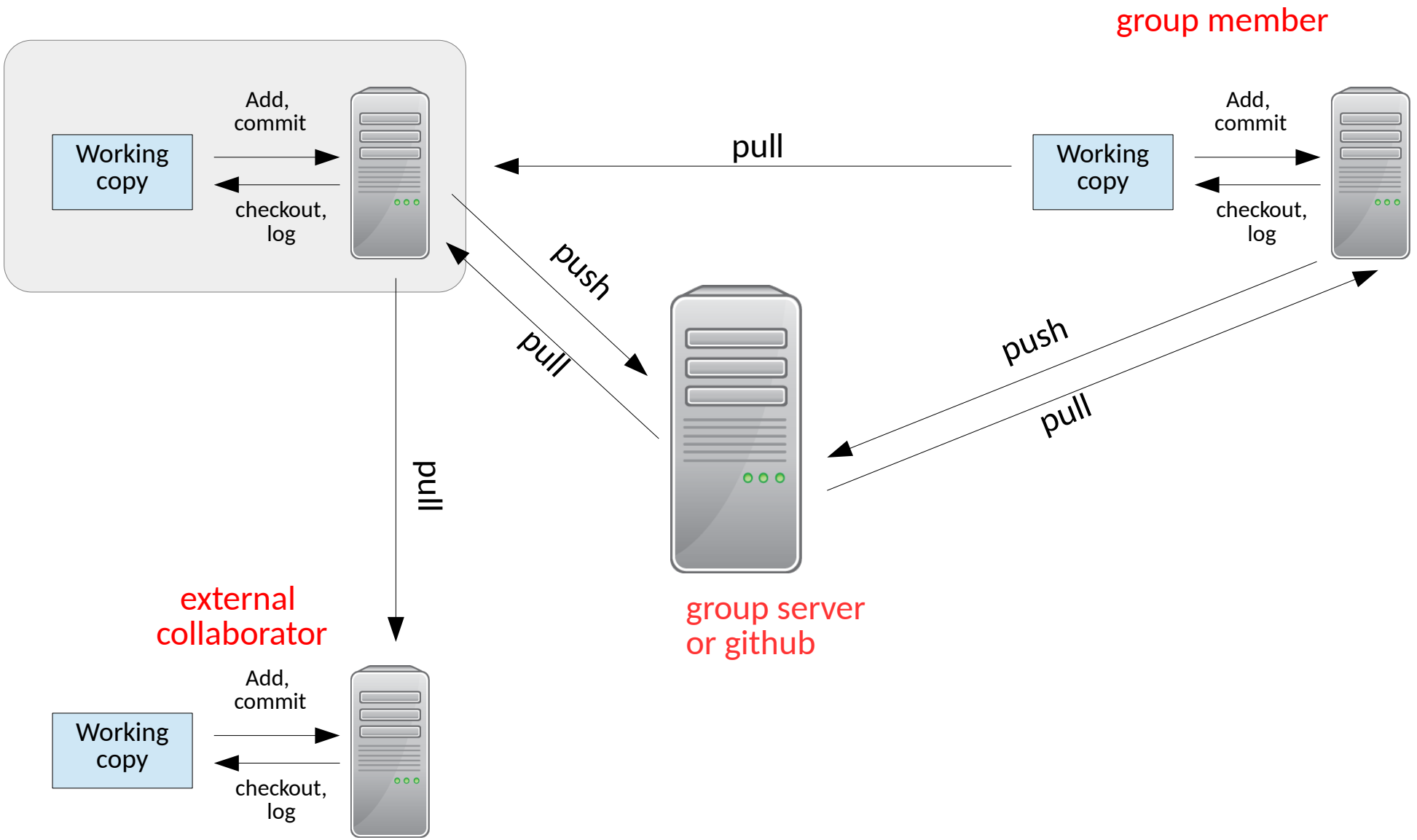
Centralized vs. Distributed Version Control

- Centralized (e.g. CVS, subversion)
  - Server holds the master copy of the source, stores history, changes
  - User communicates with server
    - Checkout source
    - Commit changes back to the source
    - Request the log (history) of a file from the server
    - Diff your local version with the version in the server
  - Doesn't scale well for very large projects
  - “Older” style of version control

- Distributed (e.g. git, mercurial)
  - Everyone has a full-fledged repository
  - You clone another person's repo
  - Commits, history, diff, logs are all local operations (these operations are faster)
  - You push your changes back to others.
  - Each copy is a backup of the whole history of the project
  - Easier to fork—just clone and go

Any version control system is better than none!

Distributed Version Control



Distributed Version Control

- In an ideal world, people only pull from others, never push.
  - See, e.g. http://bitflop.com/document/111
- Github/bitbucket provide a centralized repo built around *pull requests*

Version Control

- Note that with git, every change generates a new “hash” that identifies the entire collection of source.
  - You cannot update just a single sub-directory—it's all or nothing.
- Branches in a repo allow you to work on changes in a separate are from the main source.
  - You can perfect them, then merge back to the main branch, and then push back to the remote.
- LOTS of resources on the web.
- Best way to learn is to practice.
- There is more than one way to do most things
- Free (for open source), online, web-based hosting sites exist (e.g. Github, BitBucket, …)

Git


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

(xkcd)

Quick git Example

- We'll look at the example of having people work with a shared remote repository—this is common with groups.
  - Each developer will have their own clone that they interact with, develop in, branch for experimentation, etc.
  - You can push and pull to/from the remote repo to stay in sync with others
  - You probably want to put everyone in the same UNIX group on the server
- We'll start by creating a shared master bare repo:
  - `git init --bare --shared myproject.git`
  - `chgrp -R groupname myrepo.git`

Note the permissions set the sticky bit for the group (guid)

Quick git Example

- This repo is empty, and bare—it will only contain the git files, not the actual source files you want to work on
- Each user should clone it
  - In some other directory. User A does:
    - `git clone /path/to/myproject.git`
  - Now you can operate on it
    - Create a file (README)
    - Add it to your repo: `git add README`
    - Commit it to your repo: `git commit README`
    - Push it back to the bare repo: `git push`
  - Note that for each commit you will be prompted to add a log message detailing the change

* older versions of git won't know where push to. Instead of this, you can tell git to use the proposed new (git 2.0) behavior by doing:

```
git config --global push.default simple
git push
```

Quick git Example

- If you get confused about where the remote repo you are working with is, you can do:
  - `git remote -v`

Quick git Example

- Now user B comes along and wants to play too:
  - In some other directory. User B does:
    - `git clone /path/to/myrepo.git`
  - Note that they already have the README file
    - Edit README
    - Commit you changes locally: `git commit README`
    - Push it back to the bare repo: `git push`
- Now user A can get this changes by doing: `git pull`
  - Note that I did this on my laptop for demonstration, but the different users can be on completely different machines (and in different countries), as long as they have access to the same server
  - In general, you can push to a *bare repo*, but you can pull from anyone

Quick git Example

- You can easily look at the history by doing: `git log`
- You can checkout an old version using the hash:
  - `git checkout hash`
  - Make changes, use this older version
  - Look at the list of branches: `git branch`
  - Switch back to the tip: `git checkout master`
- Other useful commands:
  - `git diff`
  - `git status`
  - Branching
    - `git branch experiment`
    - `git checkout experiment`

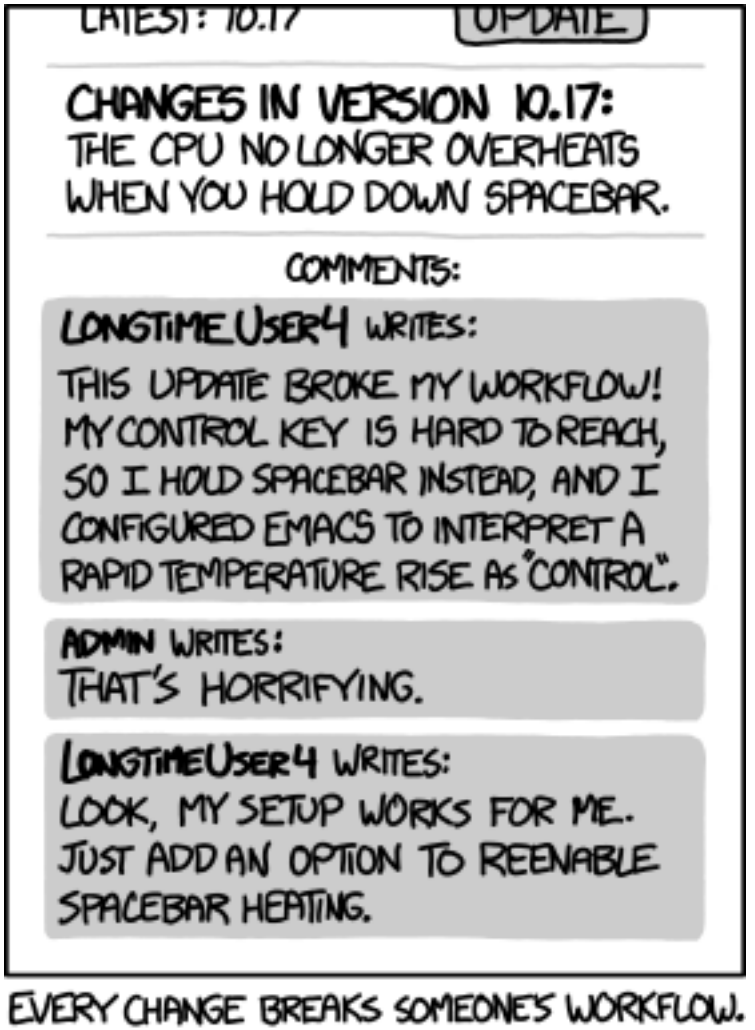
    } `git checkout -b experiment`
  - `git blame`

Quick git Example

- You can also put a link to your bare repo on the web and people can clone it remotely
  - Note you need to do `git update-server-info -f` after each change to the repo

Community

- Github / bitbucket provide tools to engage with your community
- Issue tracking
- Pull requests



(xkcd)

Github example

- Don't want to use your own server, use github or bitbucket
  - Free for public (open source) projects
  - Pay for private projects
- Create a github account (free)
- These class notes are on github:
  - `git clone https://github.com/sbu-python-class/python-science`
- Github is great for managing a community of developers outside your organization
  - You don’t have to give everyone write permission
  - Normal interaction is through *pull request* and *issues*

Github example

- Our class test repo: https://github.com/sbu-python-class/test-repo
  - *Fork* the project into your own account
  - Use git to clone your fork and interact with it—*you own it, so you can push changes back to your fork*
  - Issue a *pull-request* to the main (upstream) project asking for your changes to be incorporated
  - Feel free to try this workflow out with the class repo!

Slack Integration

- You can integrate your research group's github into your slack
  - This is done in our git channel
  - Any changes, PRs, issues, etc. will be reported

Version Control

- There's no reason not to use version control
  - Pick one (git or hg) and use it
- Even if you are working alone
  - Each clone has all the history of the project
  - Cloning on different machines means you have backups
  - Allows you to sync up your work between home and the office

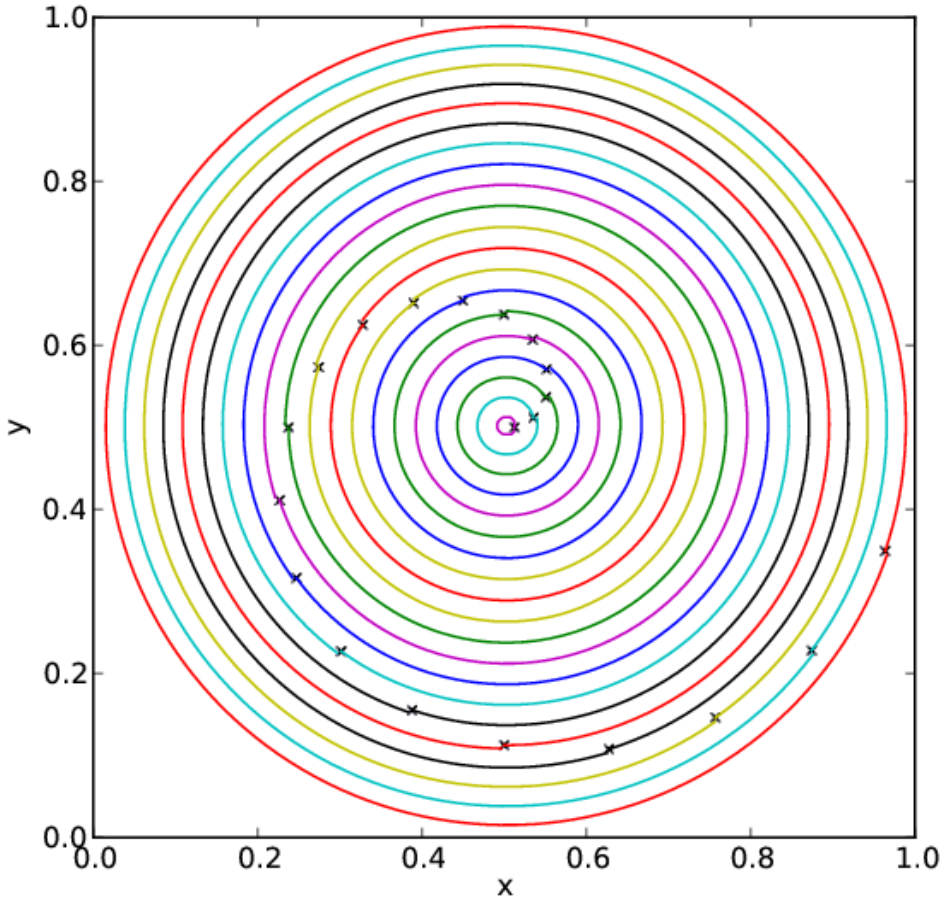
Unit Testing

- When writing a complex program (e.g. a simulation code), there can be many separate steps / solvers involved in getting your answer.
  - Finding out the source of errors in such a complicated code can be tough.
- Unit testing is the practice in which each smallest, self-contained unit of the code is tested independently of the others.
  - You could, for example, start by writing tests for each of the major physics units, and then worry about lower levels
- Implementation:
  - Either write your own simple driver for each routine to be tested
  - Unit testing frameworks automate some tasks
    - nose is a very popular one—we'll see some examples of this on the discussion board

Unit Testing

- Simple example: matrix inversion
  - Your code have a matrix inversion routine that computes A^{-1}
  - A unit test for this routine can be:
    - Pick a vector x
    - Compute b = A x
    - Compute x = A^{-1} b
    - Does the x you get match (to machine tol) the original x?
- More complicated example: a hydro program may consist of
  - Advection routines, EOS calls (and inverting the EOS), Particles, Diffusion, Reactions
  - Each of these can be tested alone
- There is a python unit testing framework called nose—we can explore that in the discussion forum if there is interest.

Unit Testing



Test of particle advection in our low Mach hydro code, Maestro

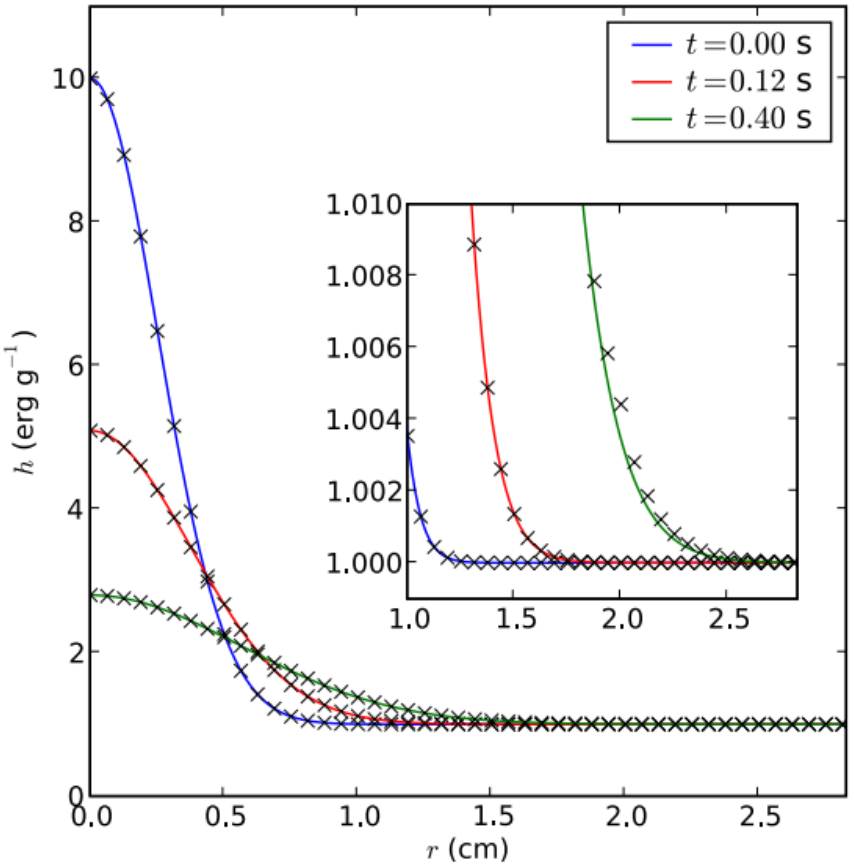


Figure 18. Average of enthalpy as a function of radius from the center, $(x, y) = (2.0, 2.0)$, of a two-dimensional Gaussian pulse. The $\times$'s are data from the numerical solution at the shown times. The lines represent the analytic solutions as given by Equation (A9). The numerical solution tracks the analytic solution very well except when the pulse has diffused enough that it begins to interact with the boundaries of the computational domain as seen in the inset plot.

Test of diffusion in our low Mach hydro code, Maestro
(Malone et al. 2011)

Unit Testing

- Whenever you important some legacy code into your project, write a unit test
  - Verifies that it performs as the authors intend
  - Allows for tests to interface changes you make to the code

Regression Testing

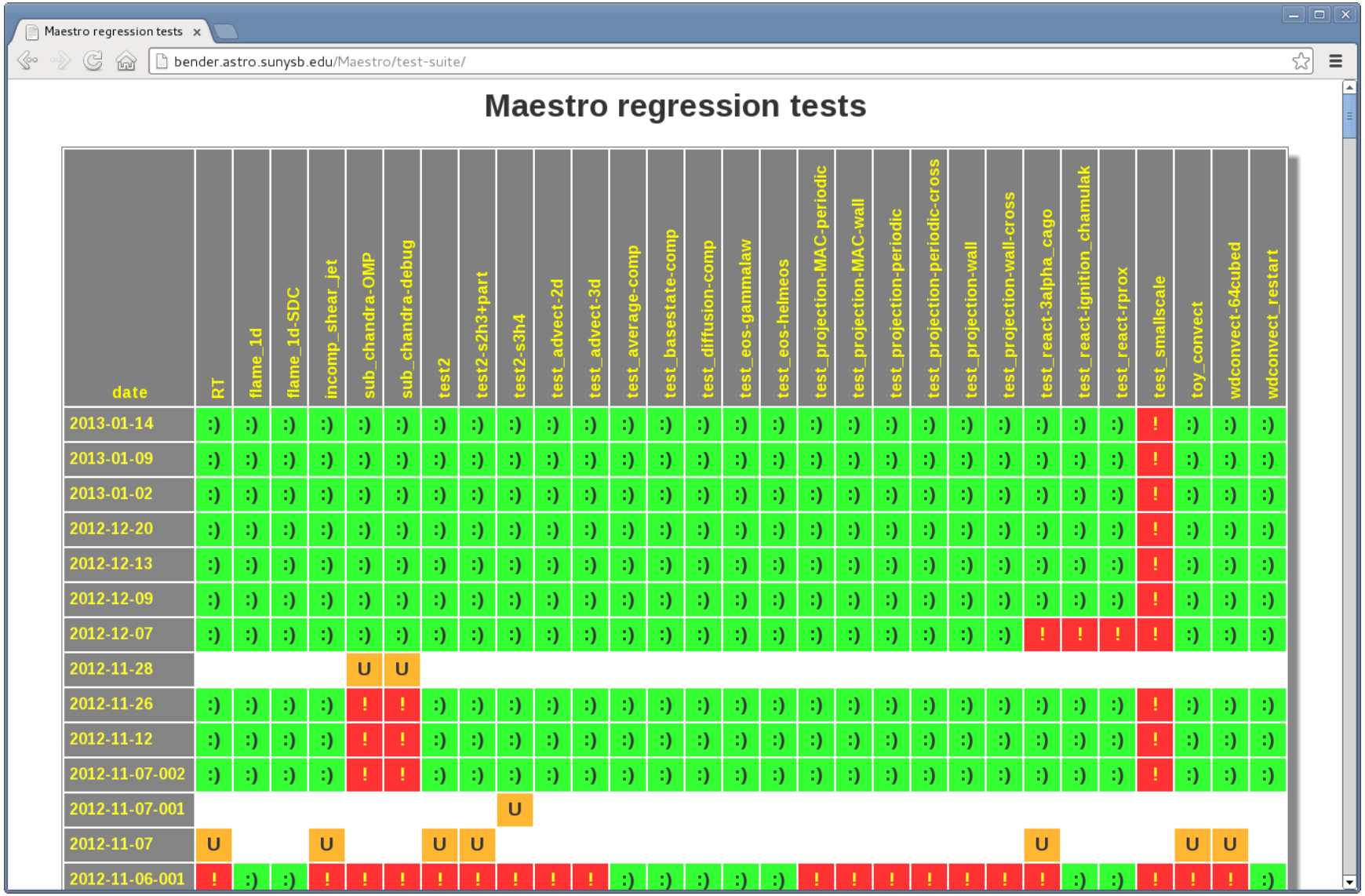
- Imagine you've “perfected” your program
  - You are confident that the answer it gives is “right”
  - You want to make sure that any changes you do in the future do not change the output
  - Regression testing tests whether changes to the code change the solution
- Regression testing:
  - Store a copy of the current output (a benchmark)
  - Make some changes to the code
  - Compare the new solution to the previous solution
  - If the answers differ, either:
    - You've introduced a bug → fix it
    - You've fixed a bug → update your benchmark

Regression Testing

- Simplest requirements:
  - You just need a tool to compare the current output to benchmark
  - You can build up a more complex system from here with simple scripting
- Big codes need a bunch of tests to exercise all possible options for the code
  - If you spend a lot of time hunting down a bug, once you fix it, put a test case in your suite to check that case
  - You'll never have complete coverage, but your number of tests will grow with time, experience, and code complexity
- Simple example with my python hydro code
  - `./pyro.py --compare_benchmark advection smooth inputs.smooth`

Regression Testing

- Maestro (automated regression testing)



date	RT	flame_1d	flame_1d-SDC	incomp_shear_jet	sub_chandra-OMP	sub_chandra-debug	test2	test2-s2h3+part	test2-s3h4	test_advect-2d	test_advect-3d	test_average-comp	test_basestate-comp	test_diffusion-comp	test_eos-gammalaw	test_eos-helmeos	test_projection-MAC-periodic	test_projection-MAC-wall	test_projection-periodic	test_projection-periodic-cross	test_projection-wall	test_projection-wall-cross	test_react-3alpha_cago	test_react-ignition_chamulak	test_react-rprox	test_smallscale	toy_convect	wdconvect-64cubed	wdconvect_restart
2013-01-14	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	!	:)	:)	:)
2013-01-09	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	!	:)	:)	:)
2013-01-02	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	!	:)	:)	:)
2012-12-20	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	!	:)	:)	:)
2012-12-13	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	!	:)	:)	:)
2012-12-09	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	!	:)	:)	:)
2012-12-07	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	!	!	!	!	:)	:)	:)
2012-11-28					U	U																							
2012-11-26	:)	:)	:)	:)	!	!	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	!	:)	:)	:)
2012-11-12	:)	:)	:)	:)	!	!	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	!	:)	:)	:)
2012-11-07-002	:)	:)	:)	:)	!	!	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	:)	!	:)	:)	:)
2012-11-07-001									U																				
2012-11-07	U			U			U	U															U				U	U	
2012-11-06-001	!	:)	:)	!	!	!	!	!	!	!	!	:)	:)	:)	:)	:)	!	!	!	!	!	!	!	:)	:)	!	!	!	:)

2013-01-14 / test2-s2h3+part

build directory: Exec/TEST_PROBLEMS/test2/

Execution Time (seconds) = 1649.596245

input file: inputs_2d.s2h3

aux1File: model.hse.cool.coulomb

dimensionality: 2

Compilation Successful

Compliation command:
gmake -j1 BOXLIB_HOME=/home/zingale/gfortran-testing/BoxLib/ ASTRODEV_DIR=/home/zingale/gfortran-testing/AstroDev/ NDEBUG=t MPI= OMP= COMP=gfortran >& /home/zingale/gfortran-testing/Maestro-tests/2013-01-14/test2-s2h3+part//test2-s2h3+part.make.out

make output

Comparison Successful

Execution command:
./main.Linux.gfortran.exe inputs_2d.s2h3 --plot_base_name test2-s2h3+part_plt --check_base_name test2-s2h3+part_chk --chk_int 0 >& test2-s2h3+part.run.out

execution output

```
../fcompare.exe -n 0 --infile1 /home/zingale/gfortran-testing/Maestro-benchmarks/test2-s2h3+part_plt00139 --infile2 test2-s2h

           variable name        absolute error        relative error
 ---------------------------------------------------------------------
 level =  1
     x_vel                    0.000000000           0.000000000
     y_vel                    0.000000000           0.000000000
     density                  0.000000000           0.000000000
     rhoh                     0.000000000           0.000000000
     h                        0.000000000           0.000000000
     X(C12)                   0.000000000           0.000000000
     X(O16)                   0.000000000           0.000000000
     X(Mg24)                  0.000000000           0.000000000
     w0_x                     0.000000000           0.000000000
```

General Rules

- When you write code, think to yourself: “if I come back to this 6 months from now, while I understand what I've done?”
  - If not, take the time now to make things clearer, document (even a simple README) what you've done, where the equations come from, etc.
  - You'll be surprised and how long your code lives on!
- Some languages let you do cute tricks.
  - Even if they might offer a small speed bump, if they complicate the code a lot to the point that it is hard to follow, then they're probably not worth it.
- Get things working before obsessing on performance

Automating Reproducibility

- Store meta-data in your output files that tell you where, when, what, and how the data was produced.
  - Already saw the example of the git hash in the makefile examples
  - Maestro example…

```
================================================================================
 Job Information
================================================================================
job name:
inputs file: inputs_2d

number of MPI processes      1
number of threads            1

CPU time used since start of simulation (CPU-hours)     0.1473997255E-01


================================================================================
 Plotfile Information
================================================================================
output date:              2013-05-08
output time:              11:39:43
output dir:               /home/zingale/development/MAESTRO/Exec/TEST_PROBLEMS/test2
time to write plotfile (s)    0.5483140945


================================================================================
 Build Information
================================================================================
build date:    2013-05-08 11:38:04.553714
build machine: Linux nan.astro.sunysb.edu 3.8.9-200.fc18.x86_64 #1 SMP Fri Apr 26 12:50:07 UTC 2013 x8
6_64 x86_64 x86_64 GNU/Linux
build dir:     /home/zingale/development/MAESTRO/Exec/TEST_PROBLEMS/test2
BoxLib dir:    /home/zingale/development/BoxLib/

MAESTRO    git hash: bad8ea8d66871a2172dcb276643edce53f739695
BoxLib     git hash: febf34dba7cc701a78c73e16a543f062cf36d587
AstroDev   git hash: 5316edb829577f80977fd2db8a113ccc4da42e02

modules used:
  Util/model_parser
  Util/random
  Util/VODE
  Util/BLAS
  Source
  ../../../Microphysics/EOS
  ../../../Microphysics/EOS/helmeos
  ../../../Microphysics/networks/ignition_simple
```

```
FCOMP:           gfortran
FCOMP version:   gcc version 4.7.2 20121109 (Red Hat 4.7.2-8) (GCC)

F90 compile line: mpif90  -Jt/Linux.gfortran.debug.mpi/m -It/Linux.gfortran.debug.mpi/m -g -fno-range-
check -O1 -fbounds-check -fbacktrace -Wuninitialized -Wunused -ffpe-trap=invalid -finit-real=nan  -I..
/../../Microphysics/EOS/helmeos  -c

F77 compile line: gfortran   -Jt/Linux.gfortran.debug.mpi/m -It/Linux.gfortran.debug.mpi/m -g -fno-ran
ge-check -O1 -fbounds-check -fbacktrace -Wuninitialized -Wunused -ffpe-trap=invalid -finit-real=nan  -
I../../../Microphysics/EOS/helmeos  -c

C compile line:   gcc  -std=c99 -Wall -g -O1 -DBL_Linux -DBL_FORT_USE_UNDERSCORE   -c

linker line:      mpif90  -Jt/Linux.gfortran.debug.mpi/m -It/Linux.gfortran.debug.mpi/m -g -fno-range-
check -O1 -fbounds-check -fbacktrace -Wuninitialized -Wunused -ffpe-trap=invalid -finit-real=nan  -I..
/../../Microphysics/EOS/helmeos


================================================================================
 Grid Information
================================================================================
 level:            1
    number of boxes =           60
    maximum zones   =          384         640

 Boundary Conditions
   -x: periodic
   +x: periodic

   -y: slip wall
   +y: outlet


================================================================================
 Species Information
================================================================================
index                 name           short name        A        Z
--------------------------------------------------------------------------------
    1     carbon-12                       C12      12.00     6.00
    2     oxygen-16                       O16      16.00     8.00
    3     magnesium-24                    Mg24     24.00    12.00
```

+ values of all runtime parameters…

Debuggers

- Simplest debugging: lots of prints!
- Interactive debuggers let you step through your code line-by-line, inspect the values of variables as they are set, etc.
- pdb is the python debugger
- If you just want to know how the code gets to a certain function:

```
import traceback
traceback.print_stack()
```

Commenting and Documentation

- The only thing worse than no comments are wrong comments
  - Comments can easily get out of date as code evolves
- Comments should convey to the reader the basic idea of what the next set of lines will accomplish.
  - Avoid commenting obvious steps if you've already described the basic idea
- Many packages allow for automatic documentation of routines/interfaces using pragmas put into the code as comments.

Source Code Libraries

- There are many sources for open, well-tested, published codes that may already do what you want.
  - This makes it easier to get going, may offer better algorithms than you were prepared to code.
  - Benefits from a community of developers and maturity
  - Still need to test, examine return codes, etc.
- Many of these mature codes are already wrapped for you in SciPy.
- For other codes, we'll look next at how to extend python in Fortran and C

Summary

- Some basic coding practices can greatly improve the reliability of your code
  - Frees you to do science
- Small learning curve is greatly offset by the improved productivity and stability