Software Engineering Practices
Class Business

- Our class website has all the lectures and examples: http://bender.astro.sunysb.edu/classes/numerical_methods/
- We will use git / github to turn in our homework (through github classroom)
  - You need to create a github account (if you don’t already have one)
- How it will work:
  - For each assignment, you will be given a link
  - Following this link will create a private repo than only you and I can see
  - Your repo will have a README.md file that gives the assignment
  - You’ll commit your homework to this repo (we’ll learn about git here) and push
  - I’ll pull your repo and grade your programs
- Blackboard will be used only for the gradebook feature in this class
Software Engineering Practices

- Some basic practices that can greatly enhance your ability to write maintainable code
  - Version control
  - Build environments
  - Testing procedures
  - Automatic code error checking
  - Profiling
  - Documentation

- There are many tools that will help you write safe code and find bugs as they are introduced. These let you focus more on the science.
Software Engineering Practices

- Main goal of this lecture is to just show you what kind of tools are out there and how they can help your workflow
  - You can google around for specific details, more in-depth tutorials, etc.
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?
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 tell you what changes have been made to each file over time
  - Allow you to request the source as it was at any time in the past
  - Allow multiple developers to all work on the same source code and share and synchronize changes
    - Merges changes by different developers to the same file
    - Provide mechanisms to resolve conflicts when two developers make incompatible changes to a file
  - 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 changes
- 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

Working copy

Add, commit

checkout, log

pull

push

pull

pull

pull

pull

push

external collaborator

Working copy

Add, commit

checkout, log

group member

Working copy

Add, commit

checkout, log

group server or github

Working copy

Add, commit

checkout, log
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 area 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, ...)
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.

- Creating a master bare repo:
  - `git init --bare --shared myproject.git`
  - `chgrp -R groupname myproject.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:

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

```bash
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 your changes locally: `git commit README`
    - Push it back to the bare repo: `git push`

- Now user A can get these 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 blame`
  - `git checkout -b experiment`
Version Control

- If you put the remote repository on a different server, then you always have a backup of your project
  - Since git is distributed, if your remote server dies, each clone is a backup of the entire repo, so you are safe both ways.
- Free (for open source), online, web-based hosting sites exist (e.g. Github)

- We'll use git to hand in our homework assignments
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

- How to contribute to someone else's project?
  - Since you are not a member of that project, you cannot push back to it
    - You don't have write access
  - Use pull requests:
    - Fork the project into your own account
    - Push back to your fork
    - Issue a **pull-request** asking for your changes to be incorporated
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.**

- Implementation:
  - Either write your own simple driver for each routine to be tested
  - Unit testing frameworks automate some tasks
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 $b = A \cdot x$
    • Compute $x = A^{-1} \cdot b$
    • Does the $x$ you get match (to machine tol) the original $x$?
Unit Testing

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

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

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.
Regression Testing

• Imagine you've “perfected” your program (simulation tool, analysis tool, etc.)
  – 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
Regression Testing

- Example 1: pyro (manual comparison tool)
- Example 2: Maestro (automated regression testing)
Compiler Checks

- Let the compiler do the work for you
- Most compilers have options to warn/abort for uninitialized variables, illegal floating point operations (NaNs and Infs), etc., array bounds checking
  - Use them!
  - These can slow down the execution of your code: make a debug version (for testing) and an optimized version (for running)
- Some examples...

- Good practice: have the ability to make either a debug or production version of your code
Coding Style

- Don't make assumptions
  
  ```
  if (x == 1) then
      ! do x = 1 stuff...
  else if (x == 2) then
      ! do x = 2 stuff...
  endif
  ...
  what if x == 3? or 0? ...
  ```

- Avoid magic numbers!
  
  - Once spent a long time scratching my head over why a code was multiplying a dividing all over by $1.239842 \times 10^{-4}$.  
  
  \[(hc \text{ in eV cm})\]
  
  - Give physical constants, etc. a named variable
Coding Style

- Use functions/subroutines for repetitive tasks
- Check return values for errors
  - Applies to your own routines, system calls (malloc), or external libraries
- Use descriptive variable names
Coding Style

- **Fortran**
  - Implicit none (one implicit none in a module carries for all the routines in it)
  - Modules give compile-time argument checking
  - When “using” data from a module, explicitly use the “only” clause
  - Use “intent” to clearly state variable intents

- **Python**
  - Try/except is a great way to catch problems at runtime

- **C/C++**
  - assert()

- Every language has a set of universally-agreed good-practices—google it.
Coding Style

- Long argument lists make it likely to mess up the orderings
  - Example: our equation of state. We switched from

    subroutine eos_old(input, dens, temp, &
    xmass, &
    pres, enthalpy, eint, &
    c_v, c_p, ne, eta, pele, &
    dPdT, dPdR, dEdT, dEdR, dPdX, dhdX, &
    gam1, cs, entropy, dsdT, dsdR, &
    do_eos_diag, pt_index)

to

    subroutine eos_new(input, eos_state, do_eos_diag, pt_index)

with

    type eos_t
    real (kind=dp_t) :: rho
    real (kind=dp_t) :: T
    real (kind=dp_t) :: xn(nspec)
    real (kind=dp_t) :: p
    ...

PHY 604: Computational Methods in Physics and Astrophysics II
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
THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"MY CODE’S COMPILING."

HEY! GET BACK TO WORK!

COMPILING!

OH. CARRY ON.

(xkcd)
Makefiles

• It is good style to separate your subroutines/functions into files, grouped together by purpose
  – Makes a project easier to manage (for you and version control)
  – Reduces compiler memory needs (although, can prevent inlining across files)
  – Reduces compile time—you only need to recompile the code that changed (and anything that might depend on it)

• Makefiles automate the process of building your code
  – No ambiguity of whether your executable is up-to-date with your changes
  – Only recompiles the code that changed (looks at dates)
  – Very flexible: lots of rules allow you to customize how to build, etc.
Makefiles

- **Basic rule:**

  target: dependencies

  |Tab| command

- Typing 'make' attempts to build the first target only

- You can create a target that depends on other targets to get several things built:

  ALL: prog1 prog2

- Easiest way to build a project consisting of many source files with dependencies among one-another.

- Can also use with LaTeX to build your papers

- Some examples...
Valgrind

- Poor memory management can lead to ill-defined behavior and crashes
- Fortran 95 automatically deallocates allocatable arrays once they go out of scope, but when using pointers and derived types, things can sometimes get missed
- C/C++ make it easy to miss things
- Valgrind is an automated tool for finding memory leaks. No source code modifications are necessary.
- Interactive example...
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.
- `gdb` is the version that works with the GNU compilers. Some graphical frontends exist.
- Lots of examples online
- Not very useful for parallel code.
Profiling

- Profilers examine your code when it runs and determine where you spend most of your time
- gprof is the standard GNU profiler
  - Just add -pg to the compile lines
  - Run as normal
  - gprof executable to get information on the subroutine/function level
  - gprof -l executable to get information on the line-by-line level
- Profiling example...
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.

- Examples:
  - Netlib
  - GAMS (NIST)
Random Bits...

- xxdiff / meld / ...

- Scripting
  - If you find yourself doing the same task more than 1 or 2 times, automate it
    - Reduces the likelihood of bugs
    - Gives you a record of what you've done
    - Applies to making plots, analyzing data, managing simulation jobs, ...

- Gnuplot
  - Installed just about everywhere
  - Great for quick-and-dirty plots (with some effort can do publication quality too)
  - Easy to do operations on columns of data
Random Bits...

- 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...