CS 294-73
Software Engineering for Scientific Computing

Lecture 6: homework #1, git, and debugging tools.

Version Control

- An organizational protocol for keeping track of different versions of a project
- Example: You finally get part of your final project for CS294 to work
- Before moving on, you copy all of your code to a separate directory (probably called something like FINALLY_WORKING)
- The backup copy in this example is a version
- We would like to keep track of versions in a more sophisticated way

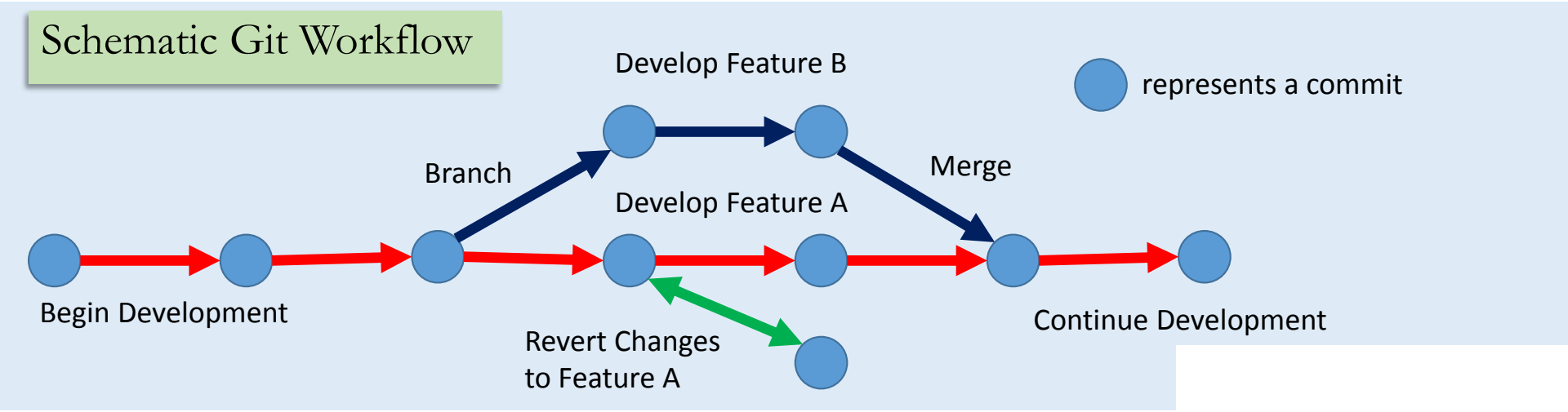
Version Control

Especially for large projects,**work does not happen in serial**

- **Example:**
  - Suppose Version 1.0 of your code works fine
  - You begin working on feature A, but realize feature B is more urgent.
  - You want to start working on B with Version 1.0 as a starting point (since you know it works) but don't want to scrap your work on feature A.
- **Other Examples:**
  - Two developers need to **work on different features separately**
  - While editing your code, it gets horribly convoluted; you would like **to return to a working version**
  - Your project partner **changed a bunch of things without documentation**

Solutions Offered By Git

- Git is a version control system that solves some of these problemsSave work in snapshots or checkpoints called **commits**
- Commits are summarized in a **log documenting modifications**
- Ability to **branch** workflow and later **merge** branches painlessly.



Using Git: getting started

- Clone or pull updates from an existing repository:
- **git clone name@host:repo_name** in this class:
- **git clone cs294-73@gilman.cs.berkeley.edu:resources** This will create a copy of the repo “resources”
- If you already have a copy of the desired repository, use:**git pull.** This will sync the local directory with the remote repo. Notably, if your local code and the repo version have diverged, git will try to merge them.

Adding and Deleting files

- Add files to the “index” so they will be tracked by git:
- **git add file_i_just_made.txt** Add a file to the index of the repository.
- **git delete file_i_dont_want.txt** Delete a file from the index for the repository.
- **git commit –m “description of changes here”** Commit the changes in the index to the repository. After a commit, all changes are still local. To update the remote repo:
- **git push origin <branch_name>**

Add / commit / push

- Some notes on add, commit, and push:
- These commands all alter the state of the code in whichever branch you are in (more on this in a bit)
- Generally, it is good practice to group similar changes in a commit (e.g. adding a new piece of functionality or fixing a group of bugs).
- The commit message should be representative and concise (just like commenting your code... which is also good practice)
- In this class, if you try to push to a repo that you shouldn't, git won't let you. This is because we are using git with a layer of authentication on top. Normally, git is FFA.
- Be discriminating about what you add to the repo:
  - No binaries (*.o, *.exe), or files that you regenerate when building (*.d)
  - No intermediate files from latex (*.aux, *.log).
  - Be very careful about adding while using wildcards ("add *"). You can end up adding git internal files that way, and then you can get in a hopeless snarl.
  - Mac users: MAC OS X doesn't distinguish between cases. Avoid filenames that are the same except for case (Foo.H , foo.H).

Status of your git repo.

- At any point, you can check the status of your edits since the last commit with:
- **git status**
- You can get a summary of your current edits vis a vis the last commit in the branch using:
- **git diff**
- You may also view the log of previous commits in your current branch using:
- **git log**

Branching

- When a git repo is first instantiated, there is one branch: **master.** For most of your assignments in CS294-73 you will stay on the master branch
- You can create a new branch with: **git branch branch_name**
- If you aren't sure which branch you are on, simply type **git branch**
- To switch to a different branch: **git checkout branch_name**

Branching

- Some notes on branching:
- When a new branch is created, its initial state is the last commit in the branch in which it was created
- You can create a branch starting from pretty much any commit in the project tree
- If you have uncommitted changes when attempting to create a new branch, git may complain. It's best to create a new branch right after a commit (i.e. from a clean slate)
- When you switch branches, the files in your local repo will take on the state of that branch.
- There is nothing "special" about the master branch. It's just the first one in the project.

Merging

- Generally, after a project has branched into parallel versions, you will want to merge them back together. From e.g. branch_A:
- **git merge branch_B** Usually this will be fine, even if changes are made to the same file in both branches
- Occasionally there will be conflicts that git can't resolve. This usually happens when both branches alter the same line of code in different ways.
- To avoid conflicts when working in groups, communicate who is working on what part of the code.

More resources

- Very Basic Tutorial
  http://rogerdudler.github.io/git-guide/
- Interactive Tutorial; not a bad place to start
  https://try.github.io/levels/1/challenges/1
- Fairly comprehensive tutorial. – comes highly recommended.
  https://www.atlassian.com/git/tutorials/
- Then, there is always the google.

Laplacian on a Rectangle

$$\phi : [0,1] \times [0,1] \to \mathbb{R}$$

$$\Delta\phi \equiv \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2}$$

Discretize using finite differences.

$$\phi^h : \Omega^h \to \mathbb{R} \ , \ \Omega^h = [0, \ldots, N] \times [0, \ldots, N]$$

$$h = \frac{1}{N} \ , \ \phi^h_{i,j} \approx \phi(ih, jh)$$

.

$$(\Delta^h\phi^h)_{i,j} \equiv \frac{1}{h^2}(\phi^h_{i+1,j} + \phi^h_{i-1,j} + \phi^h_{i,j+1} + \phi^h_{i,j-1} - 4\phi^h_{i,j})$$

$$\Delta^h\phi^h : \Omega^h_0 \to \mathbb{R} \ , \ \Omega^h_0 = [1, \ldots, N-1] \times [1, \ldots, N-1]$$

Poisson's Equation

Want to solve

$$\Delta\phi = \rho$$

$$\phi, \rho : [0,1] \times [0,1] \to \mathbb{R}$$

$$\phi(x,0) = \phi(x,1) = \phi(0,y) = \phi(1,y) = 0$$

(we will be solving Poisson's equation in many different guises throughout the course)

Discretized form

$$\rho^h_{i,j} = \rho(ih, jh)$$

$$\Delta^h \phi^h = \rho^h \text{ on } \Omega^h_0$$

$$\phi^h_{0,j} = \phi^h_{N,j} = \phi^h_{i,0} = \phi^h_{i,N} = 0$$

.

Poisson's Equation

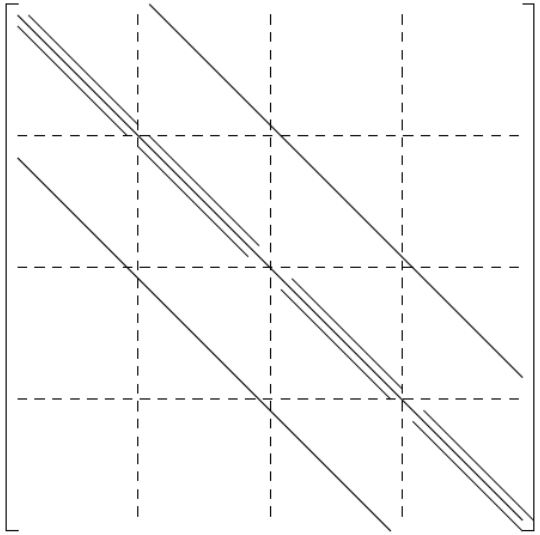
Can be written as a matrix equation

$$Au = f$$

$$u_{i+(N+1)j} = \phi_{i,j}$$

$$f_{i+(N+1)j} = \rho_{i,j} \text{ on } \Omega_0^h$$
$$= 0 \text{ on the boundary}$$

$A =$



$(N-1)^2 \times (N-1)^2$ matrix, nonzeroes along the inner tridiagonal, and two outer sub / super diagonals.

- Banded solve: $O(N^3)$ operations.
- Nested Dissection: $O(N^2 \log N)$ operations, but special to this problem.

Point Jacobi Iteration

Motivation: to solve $Au = f$ we compute it as a steady-state solution to an ODE.

$$\frac{d\tilde{u}}{dt} = A\tilde{u} - f$$

If all of the eigenvalues of *A* are negative, then

$$lim_{t\to\infty}\tilde{u}(t) = u$$

Point Jacobi: use forward Euler to solve ODE.

$$\tilde{u}^{l+1} = \tilde{u}^{l} + \mu\left(A\tilde{u}^{l} - f^{l}\right)$$

Stop when the *residual* has been reduced by a suitable amount.

$$||A\tilde{u}^{l} - f|| \leq \epsilon||f|| \;,\; ||q|| \equiv \max_{\boldsymbol{p}\in D}|q_{\boldsymbol{p}}|$$

Or stop after a fixed number of iterations, and output the norm of the residual.

Point Jacobi Iteration

Advantages:

- Simplicity: you don't have to form, store the matrix, just apply the operator.
- Generality: only depends on the eigenvalues having negative real part in order to converge (or positive real part – just can't have change of sign). Typical of operators arising from discretizing elliptic, parabolic partial differential equations.

Disadvantage: converges slowly (but we will fix that later).

Residual-Error Analysis

- Look at the equation for the error

$$\delta^l = \tilde{u}^l - u$$

$$A\delta^l = R^l = A\tilde{u}^l - Au = A\tilde{u}^l - f$$

$$\delta^{l+1} = \delta^l + \mu R^l = \delta^l + \mu A\delta^l$$

- If μ is less than the -1/(minimum eigenvalue of *A)*, then $||\delta^{l+1}|| < ||\delta^l||$. To see this, expand δ^l in the eigenvectors of *A,* and look at the evolution of each eigenmode separately. If λ is the corresponding eigenvalue, then $0 < 1 + \mu\lambda < 1$ and the amplitude of the corresponding eigenmode is decreased by that factor at each iteration. For Laplacian, eigenvalues range from -1 to -O(1/h^2), so that convergence of point relaxation is slow:
  - $1 + \mu\lambda = 1 - Ch^2$ for small eigenvalues.
  - But this is still a good starting point for better iterative methods.

Homework #1

- First part: implement `RectMDArrayImplem.H` corresponding to the `RectMDArray.H` provided to you. Compile and run the test code in `/exec1` for `DIM = 2,3.`
- The second part is to use Point Jacobi to compute an approximate solution to Poisson's equation on the unit square, with the discretization given above. The right-hand side f is given by

$$f(r) = \left(cos\left(\pi \frac{r}{2r_0}\right)\right)^6 \text{ if } r \le r_0 \qquad r = \left(\sum_{d=0}^{D-1} (x_d - .5)^2\right)^{\frac{1}{2}}$$

$$= 0 \text{ otherwise}$$

$$\text{Note: } D = \texttt{DIM}$$

Where r_0 = .25 . You should write this in a dimension-independent fashion, needing only to switch `DIM` from 2 to 3 in the makefile to change the dimensionality of the problem. Also, when you think the code is correct, you should make sure you have recompiled and run with −O3 optimization.

Dimension-independent code

Laplacian is a sum of second partial derivatives – we could have programmed it that way. Here is a slightly different version from `/exec1 .`

```
for
(Point p=D0.getLowCorner();D0.notDone(p);D0.increment(p))
   {
      LOfPhi[p] = 0.;
      for (int dir = 0;dir < DIM; dir++)
        {
          LOfPhi[p] += phi[p+getUnitv(dir)] +
            + phi[p-getUnitv(dir)];
         }
       LOfPhi[p] -=2*DIM*phi[p];
       LOfPhi[p] /= h*h;
       LOfPhi[p] -= f[p];
       maxL = max(abs(LOfPhi[p]),maxL);
    }
// Note: getUnitv(int a_dir), getOnes(), getZeros().
```

Dimension-independent code

What about the relaxation parameter ?

$$\mu = \frac{h^2}{4D}$$

i.e. (-1) times half the inverse of the diagonal contribution to the operator. With that choice

$$\delta_{\boldsymbol{p}}^{l+1} = \frac{1}{2}\left(\delta_{\boldsymbol{p}}^{l} + \frac{1}{4D}\sum_{d=0}^{D-1}(\delta_{\boldsymbol{p}+\boldsymbol{e}^d}^{l} + \delta_{\boldsymbol{p}-\boldsymbol{e}^d}^{l})\right)$$

i.e. the error in max norm is always nonincreasing (the new error at a point is the average of the old error at a point and the average of the old error at the nearest neighbors).

Run-Time Errors and Debugging

- *Run-time errors* only occur when you *run* a program, and thus, they can only occur if there is a program to run (i.e., it must have compiled and linked without errors).
- When you run the executable and something goes wrong then we call it a *run-time error*. There are two main types of run-time errors:
- **Fatal Errors**
  - A *fatal error* is basically when the executable crashes.
  - **Example 1**: The program divided by zero, as in:
    - int scores = 500; int num = 0; int avg; avg = scores / num; The
    - program would crash saying: `Floating exception`
  - **Example 2**: `Segmentation fault, Bus error.`
    - These occur when you try to access memory that your program is not allowed to use or that doesn't exist in the computer (i.e., there is only so much memory in the computer).

gdb / lldb

- We need to find out where the errors are occurring, and why they are occurring.
- Interactive debuggers: gdb (Linux / g++) or lldb (Mac / clang++).
  - Allows you to step through the program, line by line, as if it were being run under an interpreter.
  - Find the arithmetic errors, logic errors.
- Main capabilities:
  - Execution commands: `run, next, step, finish, continue.`
  - Setting breakpoints: telling the program to stop at a given point.
  - Setting watchpoints: telling the program to stop when a variable changes.
  - Examining variables and calling functions (including member functions): `print.`

- https://lldb.llvm.org/lldb-gdb.html is a cheat sheet for lldb / gdb (both!).

Demo lldb

Print inside of the debugger.

`print var , var` is POD (prints value) or a pointer to POD (prints address).

`print foo, foo` is an object – prints the member data. Sometimes it will only give you addresses of member data (e.g. it would automatically dereference pointers).

`print func(args)` – will call the function, and print the return value.

`print foo.m_bar` – will print member datum `m_bar` of object `foo.`

`print foo.memberfcn(...)` will call the member function on the object, and print the return value.

Coding Standards

- When two or more people work on a code, there can be ill feelings about how the other codes things.
- Ignoring these issues leads to slowly building resentment
  - It also leads to functionally inert code changes for style reasons, that look like development in the git logs
- So, Coding Standards.
  - You can get mad at the standard, and want to change the standard, but it isn't about your co-workers.
- Goals of a Coding Standard
  - Good formatting accompanies good structure (reference)
  - Easier to read code is often easier to understand (important for this class, i.e. me and Max).
- This is a pretty minimal coding standard − for a more elaborate one, see https://llvm.org/docs/CodingStandards.html

Highlights from Coding Standard

- Headers

```
#ifndef _EBAMR_H_  //multi-include guard
#define _EBAMR_H_
#include “yourincludesGoHere.H”  // using “
#include <systemHeadersHere.H>  // using < >
#include <cheaders>   // C++-style libC headers
class   BillyBob
{


};

#endif
```

Source files: .cpp

```
#include "BitSet.H"
#include "SPMD.H"
#include "MayDay.H"
#include <cstdlib>
#include <cstdio>

{
   .
}
.
.
```

Doxygen comments

- Code comments should be recognizable by doxygen
  - `/// assignment operator.  copies pointer member`
  - `/** copies pointer member and integer pointer, decreases refcount of rhs member  before assignment.  this refcount increased my one. */`
  - `inline const RefCountedPtr<T>& operator =(const RefCountedPtr<T>& rhs);`
  - `/// dereference access operator.  use like a pointer derefence access function.`
  - `inline T* operator ->();`
  - When the `make doxygen` target is executed this will generate html documentation, or LaTeX reference manual material

Names

- Variable names follow the convention:
  - Member variables begin with an m as in `m_memVar.`
  - Argument variables begin with an a as in a_argVar.
  - Static variables begin with an s as in s_statVar.
  - Global variables begin with an g as in g_globVar.
    - Note, the use of global variables is usually discouraged!
- Function names are `likeThis()`
  - This applies to class member functions and stand-alone functions.
- Function arguments are named.
  - For example, this is o.k.: `int cramp(int a_base, int a_power);`, but this is not: `int cramp(int, int);`.
- Argument names are identical in definitions and declarations.
- All modified arguments come before all unmodified arguments. Default values should be used sparingly, and be documented.

Indentation and Alignment

```
void AMRPoissonOp::applyOpNoBoundary(LevelData<FArrayBox>&       a_lhs,
                                     const LevelData<FArrayBox>& a_phi)
{

  LevelData<FArrayBox>&    phi = (LevelData<FArrayBox>&)a_phi;
  const DisjointBoxLayout& dbl = a_lhs.disjointBoxLayout();
  DataIterator dit = phi.dataIterator();

  phi.exchange(phi.interval(), m_exchangeCopier);

  for (dit.begin(); dit.ok(); ++dit)
    {
      const Box& region = dbl[dit];
      FORT_OPERATORLAP(CHF_FRA(a_lhs[dit]),
                       CHF_CONST_FRA(phi[dit]),
                       CHF_BOX(region),
                       CHF_CONST_REAL(m_dx),
                       CHF_CONST_REAL(m_alpha),
                       CHF_CONST_REAL(m_beta));

    }
}
```

scope token comes on own line

align arguments and successive variables

2 spaces in new scope