

An Introduction to Version Control with Git

Activity for the 2015 EarthScope Data Processing Workshop

Charles J. Ammon, Penn State

Follow along with this activity on your own.

Introduction

Goal: Demonstrate the utility of **git** sufficiently for you to recognize the importance of learning more about it.

As a scientist it is essential that you work to produce reproducible results. An important part of that process is keeping records of your work. A scientific notebook is probably the most important part of that effort. For computational work, having a record of code changes is just as important. The importance of tracking changes to computer software has led to the development of source-version-control packages that allow you to document and record changes made to software - or at this point, any text files that you might be modifying.

Whether you are writing C, python, fortran, or matlab scripts, these computer tools provide you with an opportunity to keep a history of your code modifications. They also allow you to "roll back" any changes that may not have been a good idea.

Work through the following exercise to become familiar with the basics of git (and to gain some more experience with the Atom editor).

An Introduction to Version Control with Git

Activity Setup

Open a terminal and "**cd**" to the folder containing the files for this exercise (.../Editors_and_Codes/gitActivity). The folder contains a *readme.md* file in markdown format and three C source files and a makefile that are used to build a simple command-line utility that displays a line-printer version of an earthquake focal mechanism.

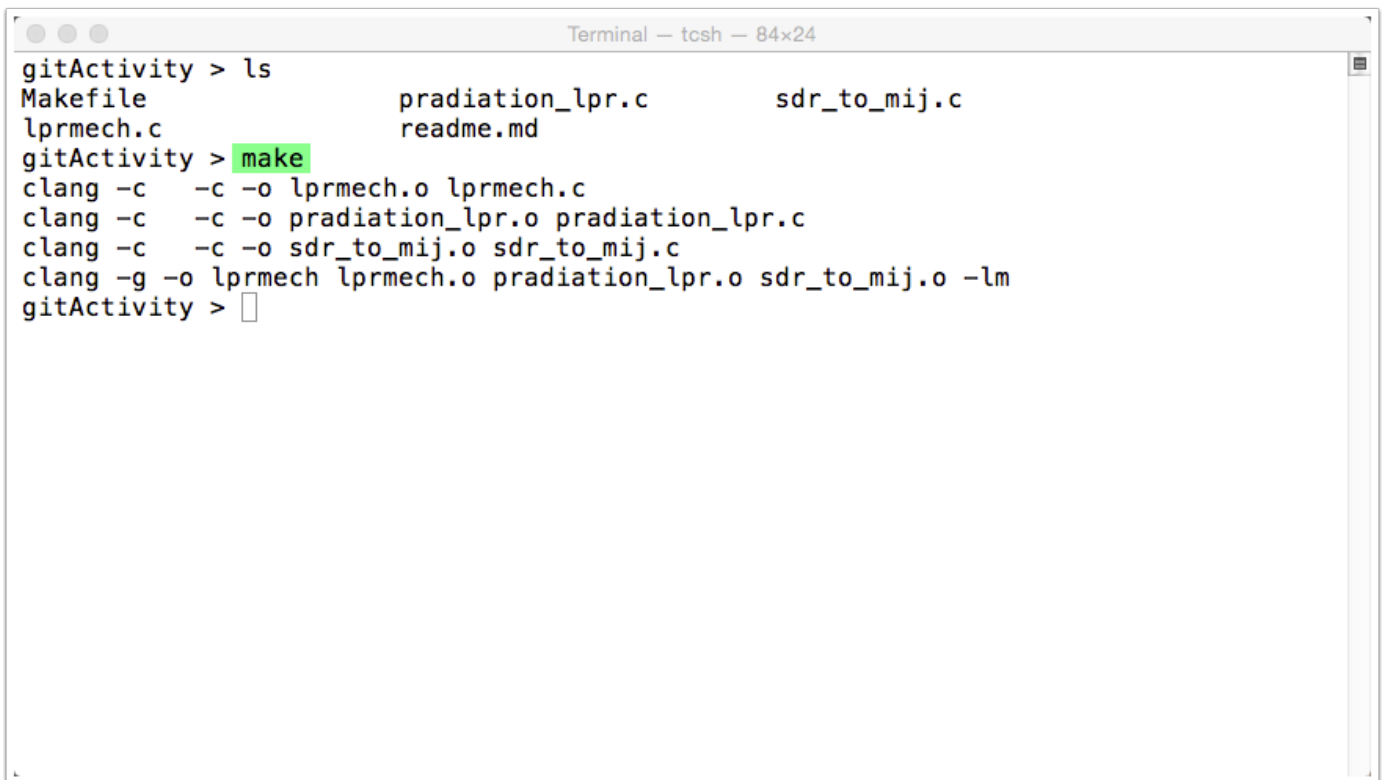


```
Terminal — tcsh — 84x24
gitActivity > ls
Makefile          pradiation_lpr.c    sdr_to_mij.c
lprmech.c         readme.md
```

An Introduction to Version Control with Git

Build the Executable

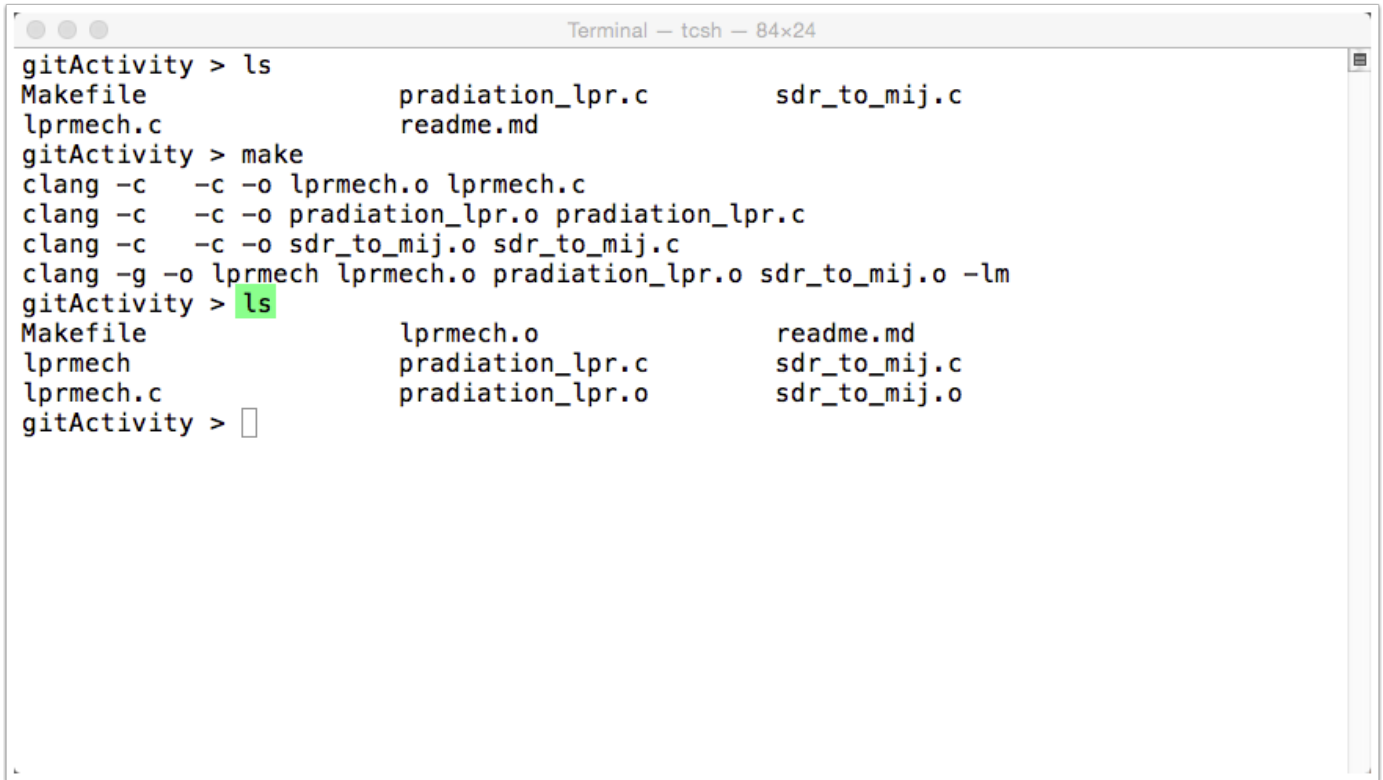
You can build the executable file using the **make** command. **Make** is a powerful tool for building codes containing multiple files (but not too many, ie. hundreds or thousands). We don't have time to get into it, but it is worth learning.



```
Terminal — tcsh — 84x24
gitActivity > ls
Makefile          pradiation_lpr.c    sdr_to_mij.c
lprmech.c         readme.md
gitActivity > make
clang -c -c -o lprmech.o lprmech.c
clang -c -c -o pradiation_lpr.o pradiation_lpr.c
clang -c -c -o sdr_to_mij.o sdr_to_mij.c
clang -g -o lprmech lprmech.o pradiation_lpr.o sdr_to_mij.o -lm
gitActivity > 
```

An Introduction to Version Control with Git

You can list the files again to see what make did. The executable is called ***lprmech***, the files that end in ".o" are compiled (binary object files) versions of the C code created with the ***clang*** compiler. The last line in the make output is the linking step that linked the object files together to create the executable.

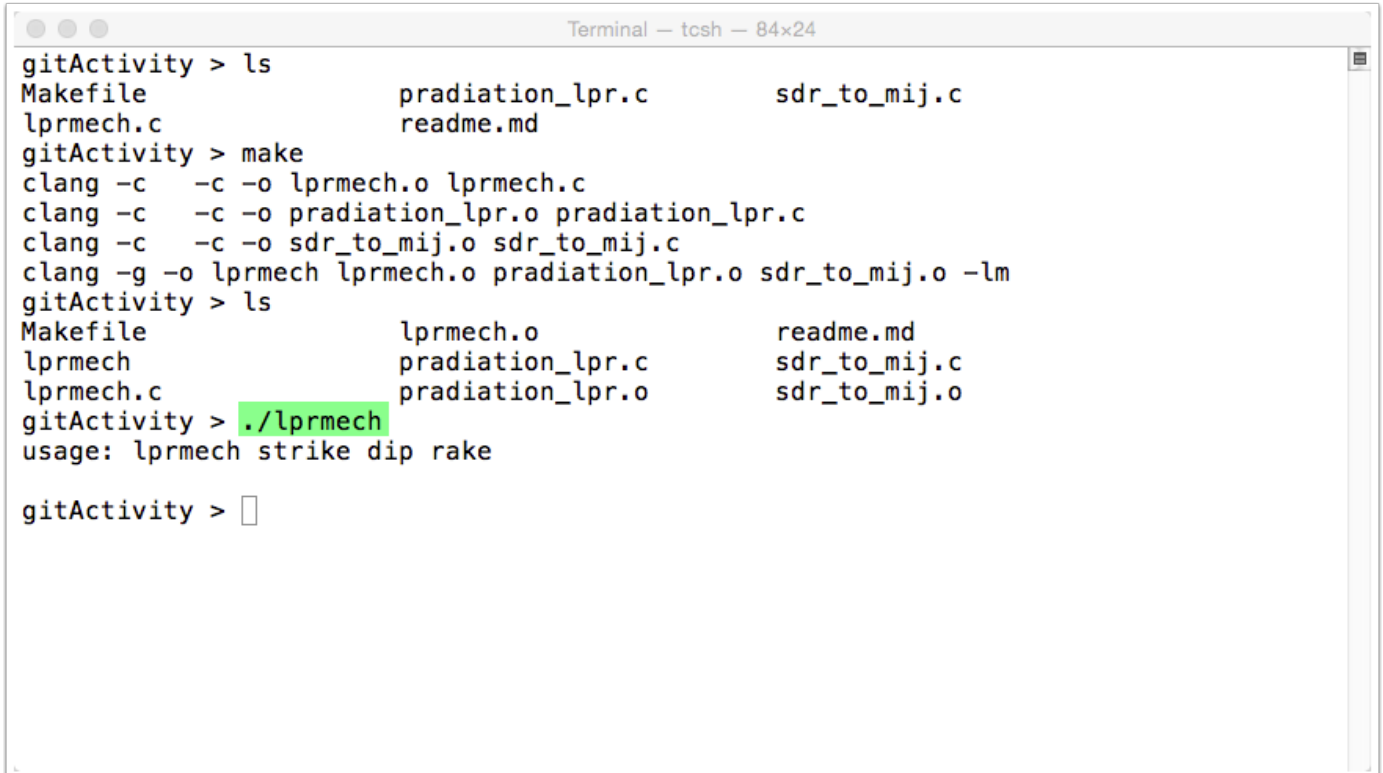
A terminal window titled "Terminal - tcsh - 84x24" showing the execution of a make command and a subsequent ls command. The output of 'make' shows the compilation of three C files into object files and their linking into an executable. The output of 'ls' shows the resulting files: Makefile, lprmech, lprmech.o, pradiation_lpr.c, pradiation_lpr.o, readme.md, sdr_to_mij.c, and sdr_to_mij.o.

```
gitActivity > ls
Makefile                pradiation_lpr.c        sdr_to_mij.c
lprmech.c                readme.md
gitActivity > make
clang -c -c -o lprmech.o lprmech.c
clang -c -c -o pradiation_lpr.o pradiation_lpr.c
clang -c -c -o sdr_to_mij.o sdr_to_mij.c
clang -g -o lprmech lprmech.o pradiation_lpr.o sdr_to_mij.o -lm
gitActivity > ls
Makefile                lprmech.o               readme.md
lprmech                 pradiation_lpr.c        sdr_to_mij.c
lprmech.c               pradiation_lpr.o        sdr_to_mij.o
gitActivity > 
```

An Introduction to Version Control with Git

Test the Executable

To run the program, enter **`./lprmech`**, you'll see a short usage statement showing you that the command accepts three command-line arguments, strike, dip, and rake.



```
Terminal — tcsh — 84x24
gitActivity > ls
Makefile          pradiation_lpr.c    sdr_to_mij.c
lprmech.c         readme.md
gitActivity > make
clang -c -c -o lprmech.o lprmech.c
clang -c -c -o pradiation_lpr.o pradiation_lpr.c
clang -c -c -o sdr_to_mij.o sdr_to_mij.c
clang -g -o lprmech lprmech.o pradiation_lpr.o sdr_to_mij.o -lm
gitActivity > ls
Makefile          lprmech.o           readme.md
lprmech           pradiation_lpr.c    sdr_to_mij.c
lprmech.c         pradiation_lpr.o    sdr_to_mij.o
gitActivity > ./lprmech
usage: lprmech strike dip rake

gitActivity > 
```

An Introduction to Version Control with Git

If you provide values for the strike, dip, and rake, you can see what this simple utility does. Here's a sample output.

```

Terminal — tcsh — 84x35
gitActivity > lprmech 0 90 0

Strike: 0 Dip 90.0 Rake: 0.0

Aki & Richards Convention
Moment Tensor:
  N      E      D
0.00    1.00    0.00
1.00    0.00    0.00
0.00    0.00   -0.00

      ----####
     ------#####
    ------#####
   ------#####
  ------#####
 ------#####
------#####
#####-----
#####-----
#####-----
#####-----
#####-----
#####-----
#####-----
#####-----
#####-----
#####-----

Harvard Convention
Moment Tensor:
  R      T      F
-0.00    0.00   -0.00
 0.00    0.00   -1.00
-0.00   -1.00    0.00

gitActivity >

```

The code is quite useful for a quick display of focal mechanisms (that you can easily paste into email messages or notes). There are some things that we should change, however. One of the changes that we want to make is highlighted above. The new

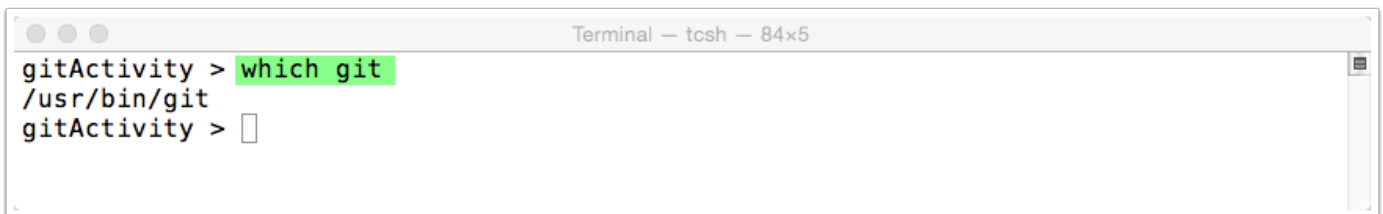
An Introduction to Version Control with Git

name of the the centroid-moment-tensor project is the Global Centroid Moment-Tensor project, no longer the Harvard Centroid Moment-Tensor project. Before we tinker with a working code, however, we should use git to save the working version and track the changes that we make in a way that we can undo our changes, in case we make an error.

Using Git

What we want to do next is create a local git repository to track changes that we are going to make in the code. **git** should be installed on the computers that you are using. If it's not on your computer, you can install it relatively simply. Follow the instructions at <http://git-scm.com>.

To see if **git** is live on the machine you are using, enter **which git** at the unix prompt.

A terminal window titled "Terminal — tcsh — 84x5" showing a command prompt. The prompt is "gitActivity >". The user has entered "which git" and the output is "/usr/bin/git". The prompt is now "gitActivity > " with a cursor.

```
gitActivity > which git
/usr/bin/git
gitActivity > 
```

An Introduction to Version Control with Git

git Help

You can get help on git from the unix man page, enter ***man git*** at the unix prompt or entering ***git --help*** at the prompt. There are also many web tutorials and at least one free ***git*** book available online.

```
Terminal — tcsh — 84x35
gitActivity > git --help
usage: git [--version] [--help] [-C <path>] [-c name=value]
        [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
        [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
        [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
        <command> [<args>]

The most commonly used git commands are:
  add          Add file contents to the index
  bisect       Find by binary search the change that introduced a bug
  branch       List, create, or delete branches
  checkout     Checkout a branch or paths to the working tree
  clone        Clone a repository into a new directory
  commit       Record changes to the repository
  diff         Show changes between commits, commit and working tree, etc
  fetch        Download objects and refs from another repository
  grep         Print lines matching a pattern
  init         Create an empty Git repository or reinitialize an existing one
  log          Show commit logs
  merge        Join two or more development histories together
  mv           Move or rename a file, a directory, or a symlink
  pull         Fetch from and integrate with another repository or a local branch
  push         Update remote refs along with associated objects
  rebase       Forward-port local commits to the updated upstream head
  reset        Reset current HEAD to the specified state
  rm           Remove files from the working tree and from the index
  show         Show various types of objects
  status       Show the working tree status
  tag          Create, list, delete or verify a tag object signed with GPG

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
gitActivity > 
```


An Introduction to Version Control with Git

We are not going to use most of those commands in this brief activity. We will focus on some basic frequently used commands.

An Introduction to Version Control with Git

Git Configuration

We'll want git to know who made changes and how to contact that person, so before we get too deep into the example, we'll configure that information into git. You'll want to substitute your information in place of mine. We use the git config command to set three items

- Our name
- Our email
- Our preferred editor

In the terminal, enter the commands as shown below.

A terminal window titled "Terminal — tcsh — 84x28" showing a series of git configuration commands and their output. The commands are: `git config --global user.name "Charles J. Ammon"`, `git config --global user.email "charlesammon@psu.edu"`, `git config --global core.editor "atom"`, and `git config --list`. The output of the last command lists various git configuration settings, including user name, email, editor, and merge tool settings.

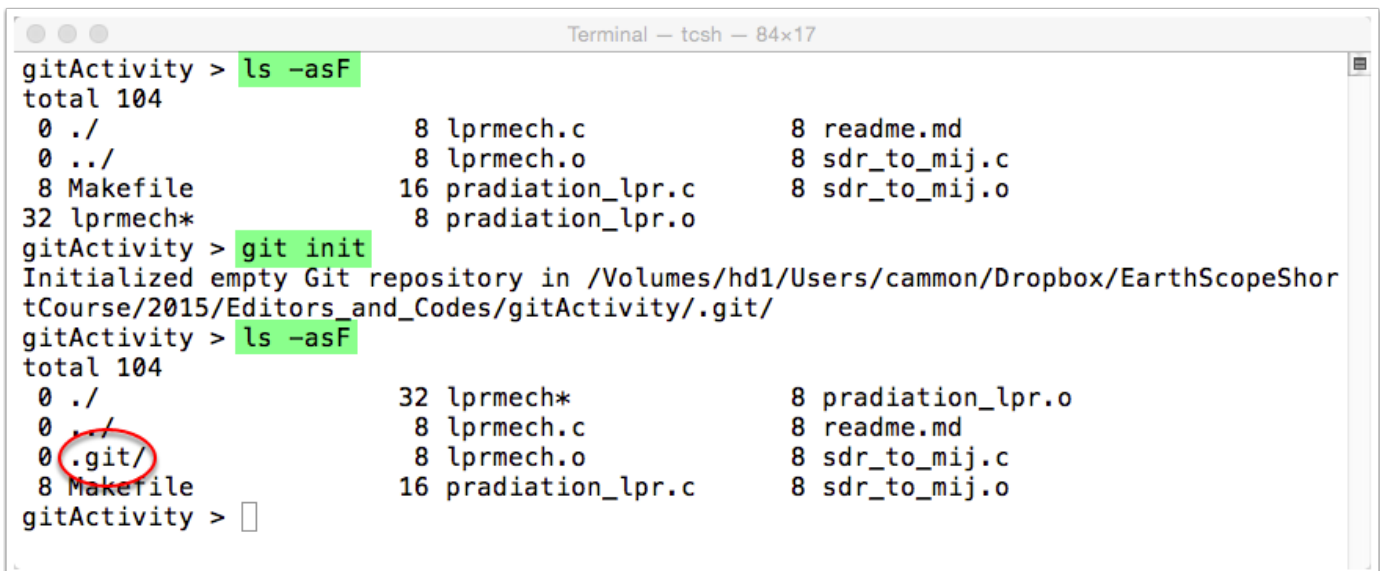
```
gitActivity > git config --global user.name "Charles J. Ammon"
gitActivity > git config --global user.email "charlesammon@psu.edu"
gitActivity > git config --global core.editor "atom"
gitActivity > git config --list
filter.media.clean=git-media-clean %f
filter.media.smudge=git-media-smudge %f
user.name=Charles J. Ammon
user.email=charlesammon@psu.edu
core.excludesfile=/Users/cammon/.gitignore_global
core.editor=atom
difftool.sourcetree.cmd=opendiff "$LOCAL" "$REMOTE"
difftool.sourcetree.path=
mergetool.sourcetree.cmd=/Applications/SourceTree.app/Contents/Resources/opendiff-w.
sh "$LOCAL" "$REMOTE" -ancestor "$BASE" -merge "$MERGED"
mergetool.sourcetree.trustexitcode=true
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
core.precomposeunicode=true
gitActivity > 
```

An Introduction to Version Control with Git

Creating the Git Repository

To create an empty git repository we simply use the `git init` command in the directory containing the files for which we plan to track changes. In our case, that's the folder that we've been working in (with the C files, makefile, and readme.md). Enter the ***git init*** command in the terminal and hit return.

Here is the result that you should see. I started with an ***ls -asF*** command so that we could see hidden files and folders in the file list. Then I entered ***git init***, and again listed the files. git created a subdirectory called ***.git*** inside the directory.

A terminal window titled "Terminal - tcsh - 84x17" showing the process of creating a git repository. The user is in a directory named "gitActivity".
First, the user runs `ls -asF`, which lists the directory contents in a long format, including hidden files. The output shows files like `./`, `../`, `Makefile`, `lprmech*`, `lprmech.c`, `lprmech.o`, `pradiation_lpr.c`, `pradiation_lpr.o`, `readme.md`, `sdr_to_mij.c`, and `sdr_to_mij.o`.
Then, the user runs `git init`. The output is "Initialized empty Git repository in /Volumes/hd1/Users/cammon/Dropbox/EarthScopeShortCourse/2015/Editors_and_Codes/gitActivity/.git/".
Finally, the user runs `ls -asF` again. The output is identical to the first listing, but now includes a new entry `.git/`, which is circled in red to highlight its creation.

```
gitActivity > ls -asF
total 104
0 ./
0 ../
8 Makefile
32 lprmech*
8 lprmech.c
8 lprmech.o
16 pradiation_lpr.c
8 pradiation_lpr.o
8 readme.md
8 sdr_to_mij.c
8 sdr_to_mij.o

gitActivity > git init
Initialized empty Git repository in /Volumes/hd1/Users/cammon/Dropbox/EarthScopeShortCourse/2015/Editors_and_Codes/gitActivity/.git/

gitActivity > ls -asF
total 104
0 ./
0 ../
0 .git/
8 Makefile
32 lprmech*
8 lprmech.c
8 lprmech.o
16 pradiation_lpr.c
8 pradiation_lpr.o
8 readme.md
8 sdr_to_mij.c
8 sdr_to_mij.o

gitActivity > 
```

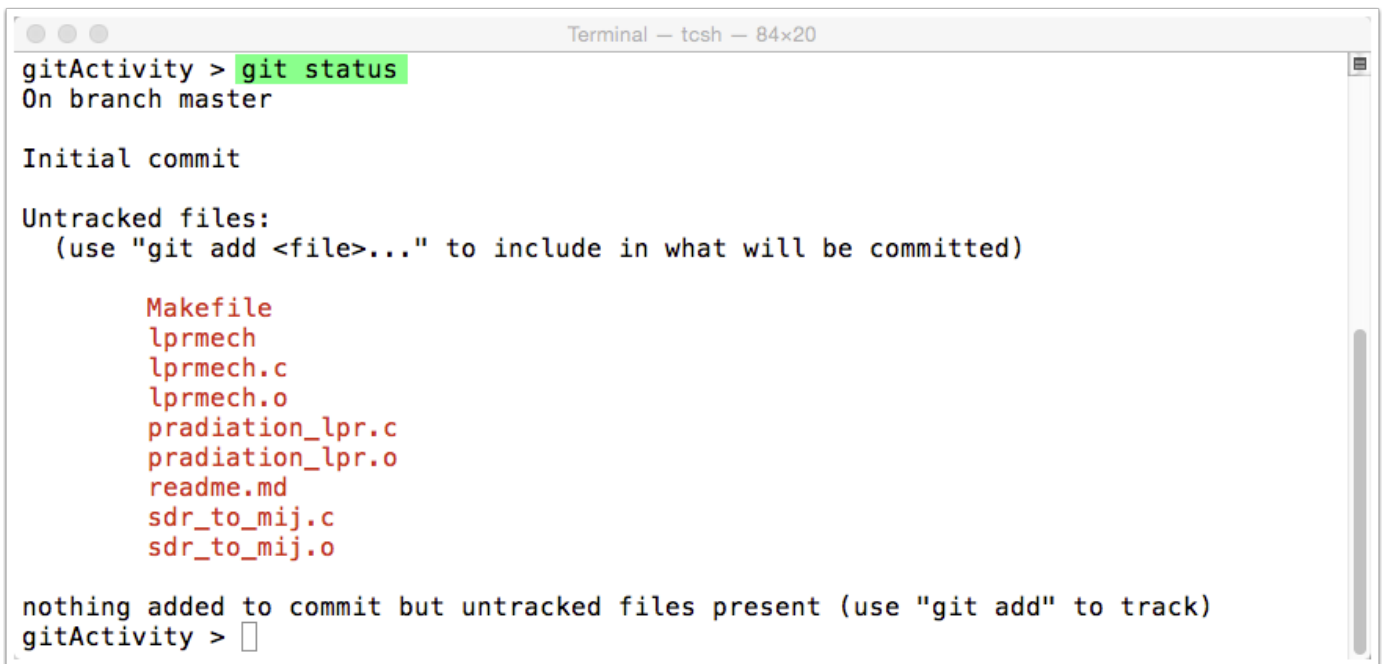
All the information needed by ***git*** will be placed inside the ***.git*** subdirectory. You can move the entire directory containing the source files and ***.git*** without affecting git's ability to track changes.

Right now, the ***.git*** folder is empty, nothing is stored within it, our first step using git is to add the files that we want to track to the repository. Adding changes to the

An Introduction to Version Control with Git

repository is a two-step process. First we **stage** the changed files (and right now all files are changes since the repository is empty), then we **commit** the changes.

We'll start with an oft-used git command, git status. That will show us what files git recognizes in our project.

A terminal window titled "Terminal — tcsh — 84x20" showing the output of the "git status" command. The output indicates an initial commit on the master branch with several untracked files listed in red text. The files are: Makefile, lprmech, lprmech.c, lprmech.o, pradiation_lpr.c, pradiation_lpr.o, readme.md, sdr_to_mij.c, and sdr_to_mij.o. The terminal also shows a message that nothing was added to the commit but untracked files are present, and a prompt for the next command.

```
gitActivity > git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    Makefile
    lprmech
    lprmech.c
    lprmech.o
    pradiation_lpr.c
    pradiation_lpr.o
    readme.md
    sdr_to_mij.c
    sdr_to_mij.o

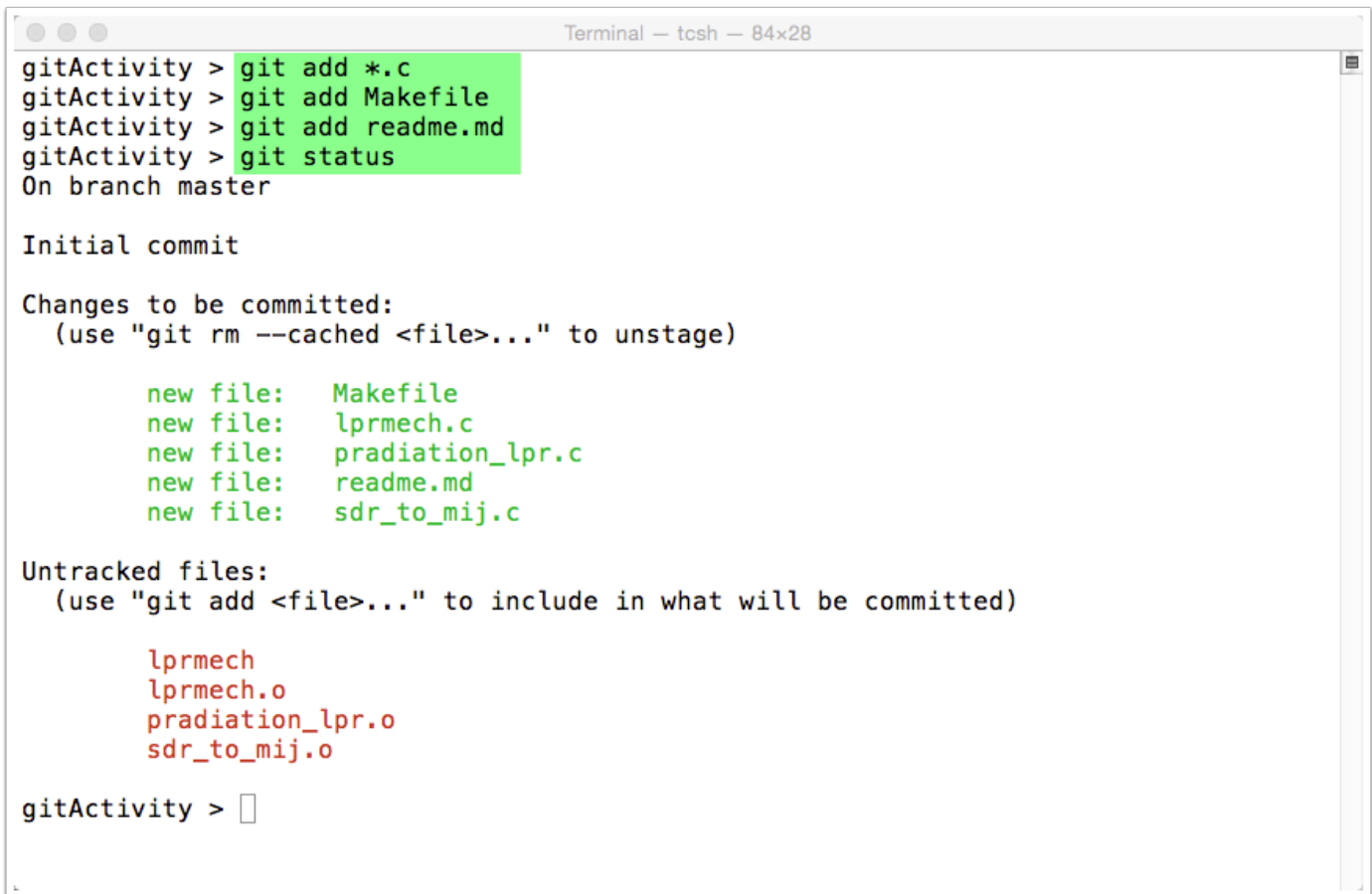
nothing added to commit but untracked files present (use "git add" to track)
gitActivity > 
```

An Introduction to Version Control with Git

Staging Files for a Commit

git tells us that there's nothing to commit at this point, but the directory contains untracked files. We want git to track the C codes, the Makefile, and the readme.md file. There's no point in tracking the *.o files since they are created from the C codes and the make file.

We stage or ready the files of interest for commit using the **git add** command.

A terminal window titled "Terminal - tcsh - 84x28" showing a series of git commands and their output. The commands are: git add *.c, git add Makefile, git add readme.md, and git status. The output shows the current branch is master, it's an initial commit, and lists the files to be committed (Makefile, lprmech.c, pradiation_lpr.c, readme.md, sdr_to_mij.c) and untracked files (lprmech, lprmech.o, pradiation_lpr.o, sdr_to_mij.o).

```
gitActivity > git add *.c
gitActivity > git add Makefile
gitActivity > git add readme.md
gitActivity > git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   Makefile
    new file:   lprmech.c
    new file:   pradiation_lpr.c
    new file:   readme.md
    new file:   sdr_to_mij.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    lprmech
    lprmech.o
    pradiation_lpr.o
    sdr_to_mij.o

gitActivity > 
```

An Introduction to Version Control with Git

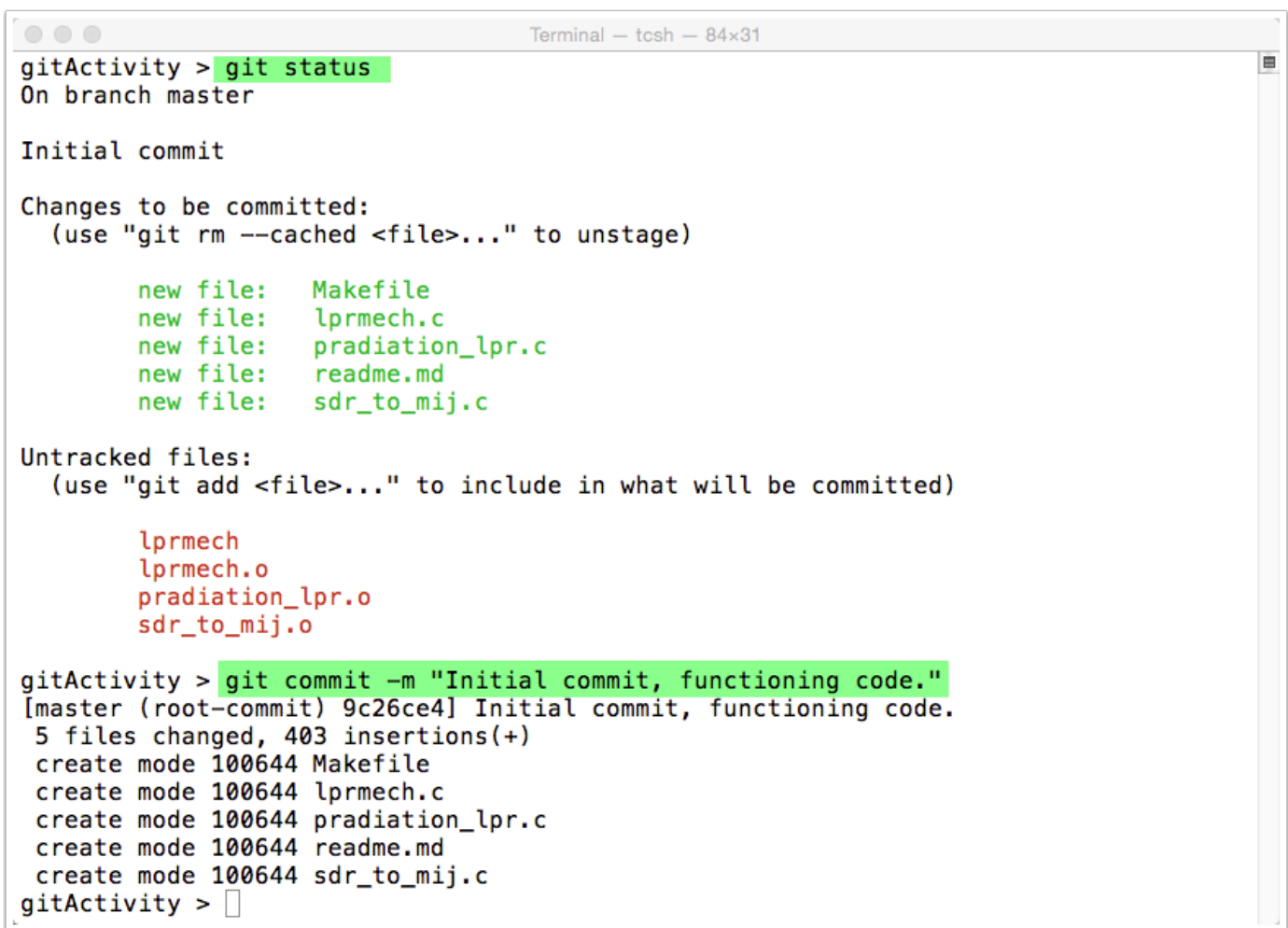
Again, **git** provides some hints for what we might want to do next in the status command. If we wanted to unstage a file we can use **git rm --cached <file>** or we can add some of the untracked files.

An Introduction to Version Control with Git

Committing Changes to the Repository

We are now ready to create the initial commit of the source codes to the repository. An important part of committing changes is describing those changes. This can be done on the command line using

git commit -m "description ..."

A terminal window titled "Terminal - tcsh - 84x31" showing the output of git status and git commit. The git status command shows an initial commit on the master branch with five new files staged for commit: Makefile, lprmech.c, pradiation_lpr.c, readme.md, and sdr_to_mij.c. It also lists untracked files: lprmech, lprmech.o, pradiation_lpr.o, and sdr_to_mij.o. The git commit -m "Initial commit, functioning code." command is then executed, resulting in a new commit 9c26ce4 with 5 files changed and 403 insertions. The commit message is "Initial commit, functioning code." and the files are created with mode 100644.

```
gitActivity > git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   Makefile
        new file:   lprmech.c
        new file:   pradiation_lpr.c
        new file:   readme.md
        new file:   sdr_to_mij.c

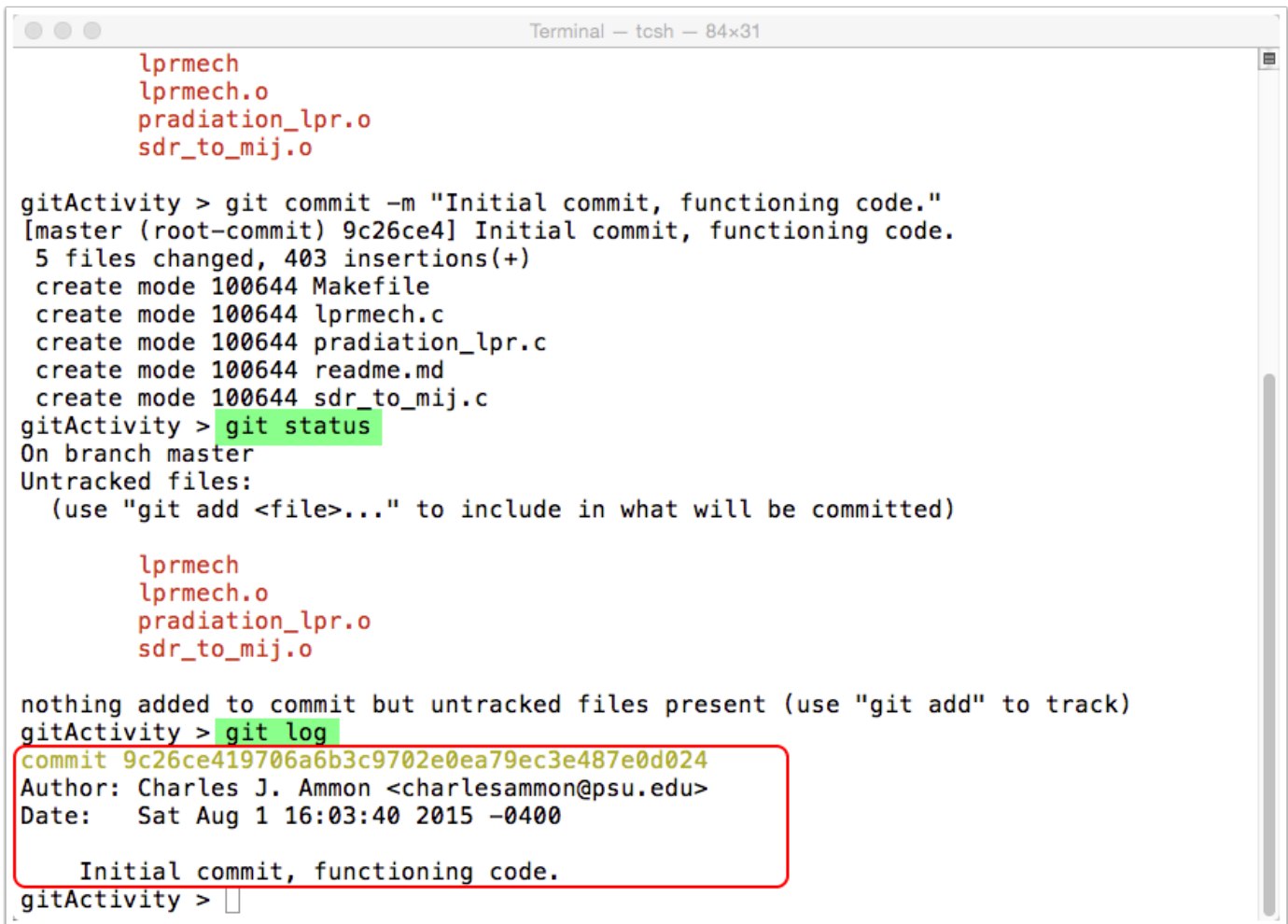
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        lprmech
        lprmech.o
        pradiation_lpr.o
        sdr_to_mij.o

gitActivity > git commit -m "Initial commit, functioning code."
[master (root-commit) 9c26ce4] Initial commit, functioning code.
 5 files changed, 403 insertions(+)
 create mode 100644 Makefile
 create mode 100644 lprmech.c
 create mode 100644 pradiation_lpr.c
 create mode 100644 readme.md
 create mode 100644 sdr_to_mij.c
gitActivity > 
```

An Introduction to Version Control with Git

We can check the status of our repository using the **git status** command, and then check the history of the repository using the **git log** command.

A terminal window titled "Terminal - tcsh - 84x31" showing a series of commands and their outputs. The user lists files (lprmech, lprmech.o, pradiation_lpr.o, sdr_to_mij.o), runs 'git commit -m "Initial commit, functioning code."', and then 'git status'. The status output shows untracked files. Finally, the user runs 'git log', and the output for the initial commit is highlighted with a red box.

```
lprmech
lprmech.o
pradiation_lpr.o
sdr_to_mij.o

gitActivity > git commit -m "Initial commit, functioning code."
[master (root-commit) 9c26ce4] Initial commit, functioning code.
5 files changed, 403 insertions(+)
create mode 100644 Makefile
create mode 100644 lprmech.c
create mode 100644 pradiation_lpr.c
create mode 100644 readme.md
create mode 100644 sdr_to_mij.c
gitActivity > git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    lprmech
    lprmech.o
    pradiation_lpr.o
    sdr_to_mij.o

nothing added to commit but untracked files present (use "git add" to track)
gitActivity > git log
commit 9c26ce419706a6b3c9702e0ea79ec3e487e0d024
Author: Charles J. Ammon <charlesammon@psu.edu>
Date:   Sat Aug 1 16:03:40 2015 -0400

    Initial commit, functioning code.
gitActivity > 
```

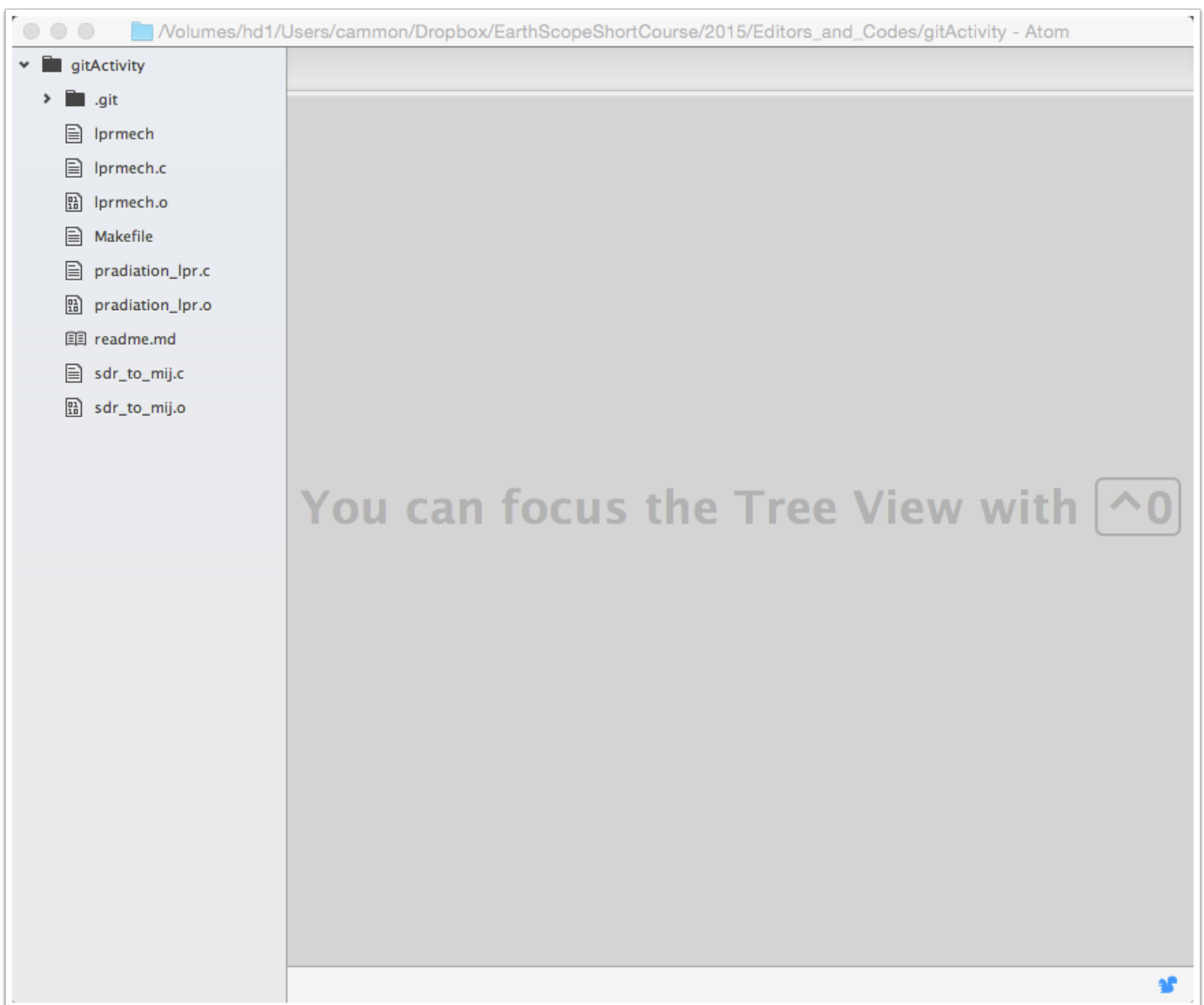
The **git log** command shows us the "hash" (the long string of hexadecimal numbers), the author, the date, and the message describing this commit. If at some point we want to revert back to this state of code, we would use the hash to identify this particular snapshot of the codes.

An Introduction to Version Control with Git

Changing the Codes

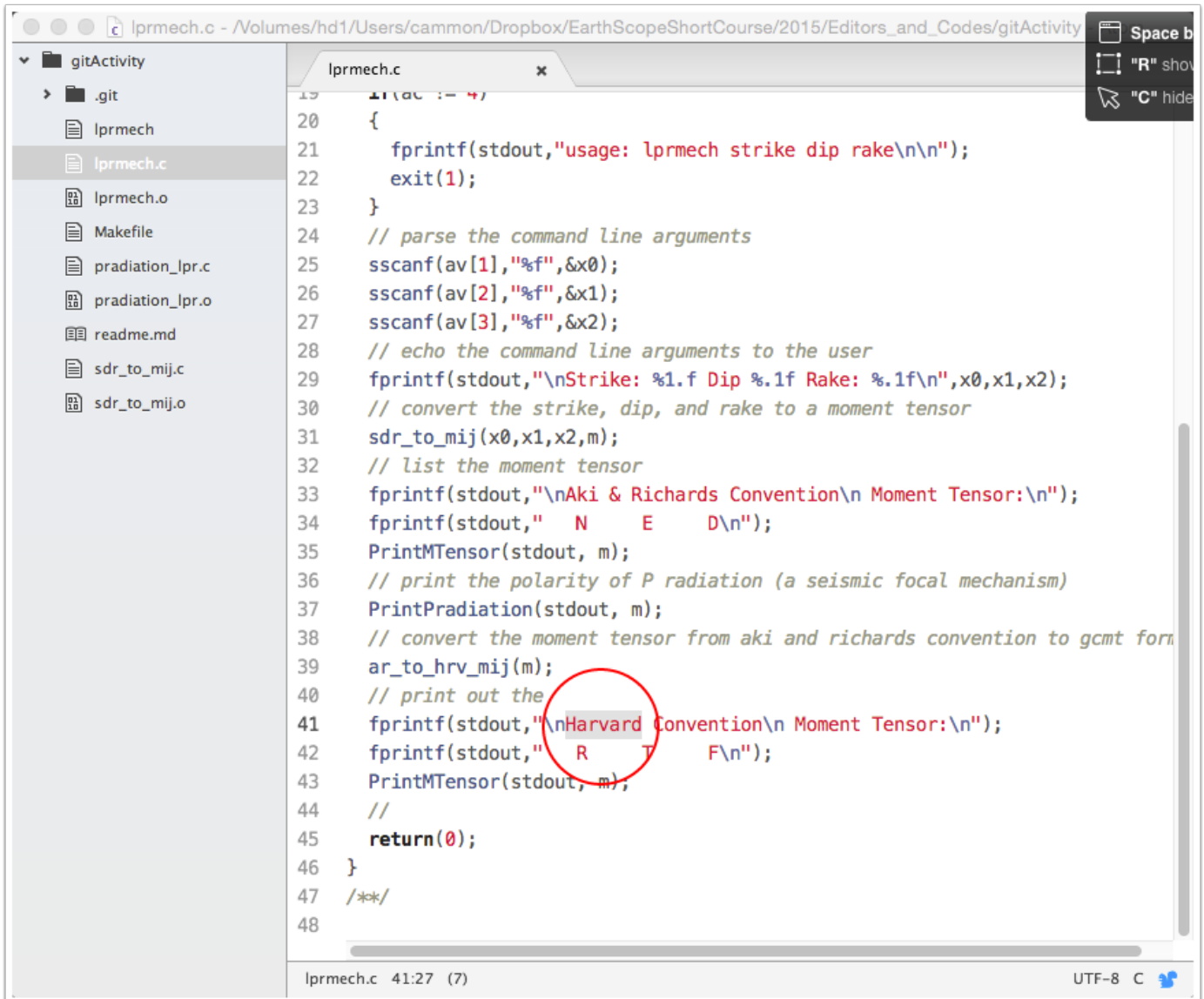
Now that we have the original codes safely recorded in the git repository, we can begin to modify them without fear of losing the original working version (and without producing a suite of directories called "original", "new", "newer", etc.

Start up Atom and open the directory containing the *lprmech* source codes.



An Introduction to Version Control with Git

Single-click the `lprmech.c` file so that we can change the output statement to include "GCMT Convention" in place of "Harvard Convention". The change should be made on line 41. Select the word Harvard and replace it with GCMT.

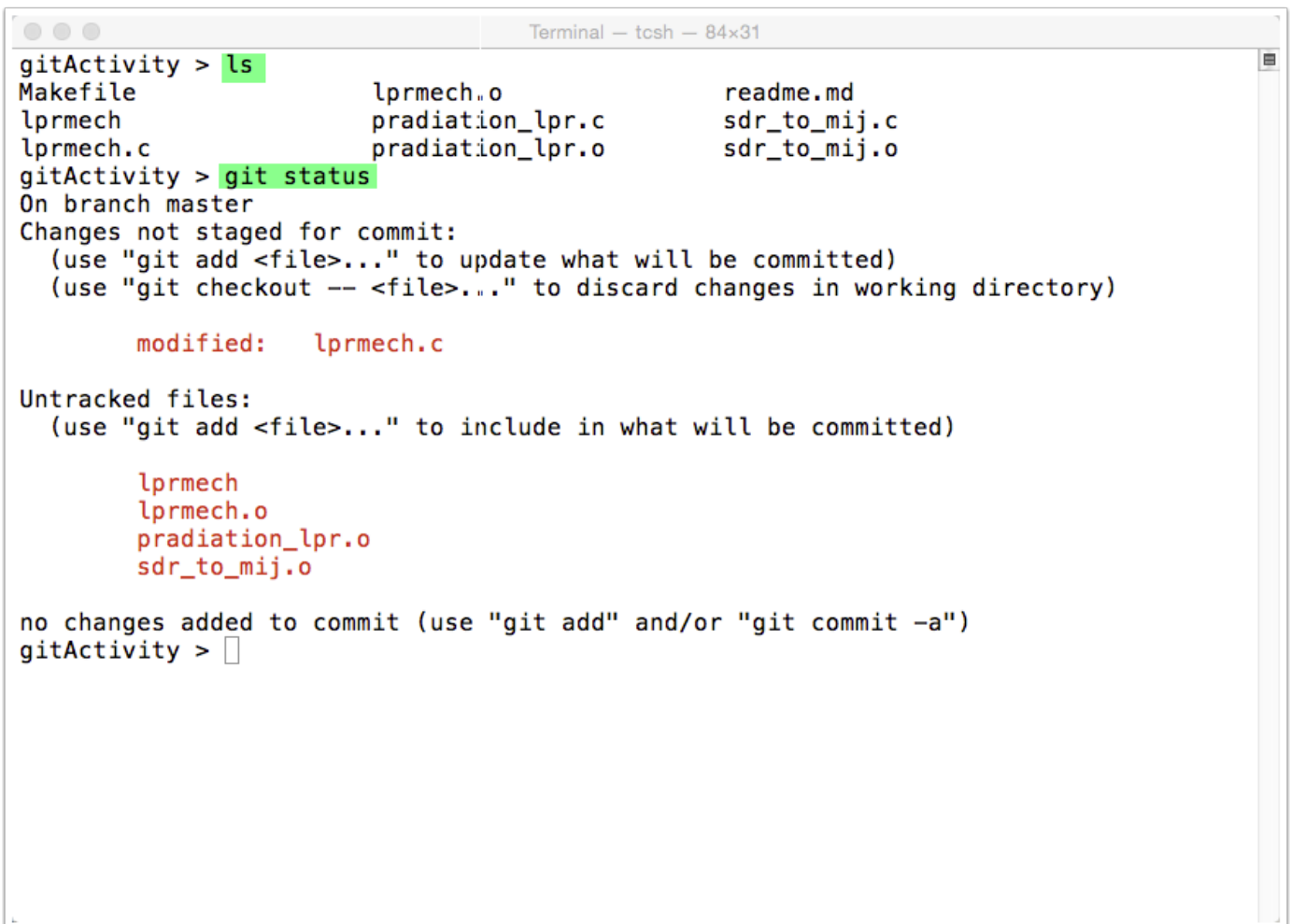


```
19  int main(int argc, char **argv)
20  {
21      fprintf(stdout, "usage: lprmech strike dip rake\n\n");
22      exit(1);
23  }
24  // parse the command line arguments
25  sscanf(argv[1], "%f", &x0);
26  sscanf(argv[2], "%f", &x1);
27  sscanf(argv[3], "%f", &x2);
28  // echo the command line arguments to the user
29  fprintf(stdout, "\nStrike: %1.f Dip %1.f Rake: %1.f\n", x0, x1, x2);
30  // convert the strike, dip, and rake to a moment tensor
31  sdr_to_mij(x0, x1, x2, m);
32  // list the moment tensor
33  fprintf(stdout, "\nAki & Richards Convention\n Moment Tensor:\n");
34  fprintf(stdout, "  N      E      D\n");
35  PrintMTensor(stdout, m);
36  // print the polarity of P radiation (a seismic focal mechanism)
37  PrintPradiation(stdout, m);
38  // convert the moment tensor from aki and richards convention to gcmt format
39  ar_to_hrv_mij(m);
40  // print out the
41  fprintf(stdout, "\nHarvard Convention\n Moment Tensor:\n");
42  fprintf(stdout, "  R      T      F\n");
43  PrintMTensor(stdout, m);
44  //
45  return(0);
46  }
47  /**/
48
```

An Introduction to Version Control with Git

Save the change to `lprmech.c` and go back to the terminal and execute a ***git status*** command.

You will see that git has recognized a change to the tracked file ***lprmech.c***, as shown below.

A terminal window titled "Terminal — tcsh — 84x31" showing the output of the `git status` command. The window has a light gray title bar with three window control buttons on the left. The terminal text is as follows:

```
gitActivity > ls
Makefile          lprmech.o          readme.md
lprmech           pradiation_lpr.c   sdr_to_mij.c
lprmech.c         pradiation_lpr.o   sdr_to_mij.o
gitActivity > git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   lprmech.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)

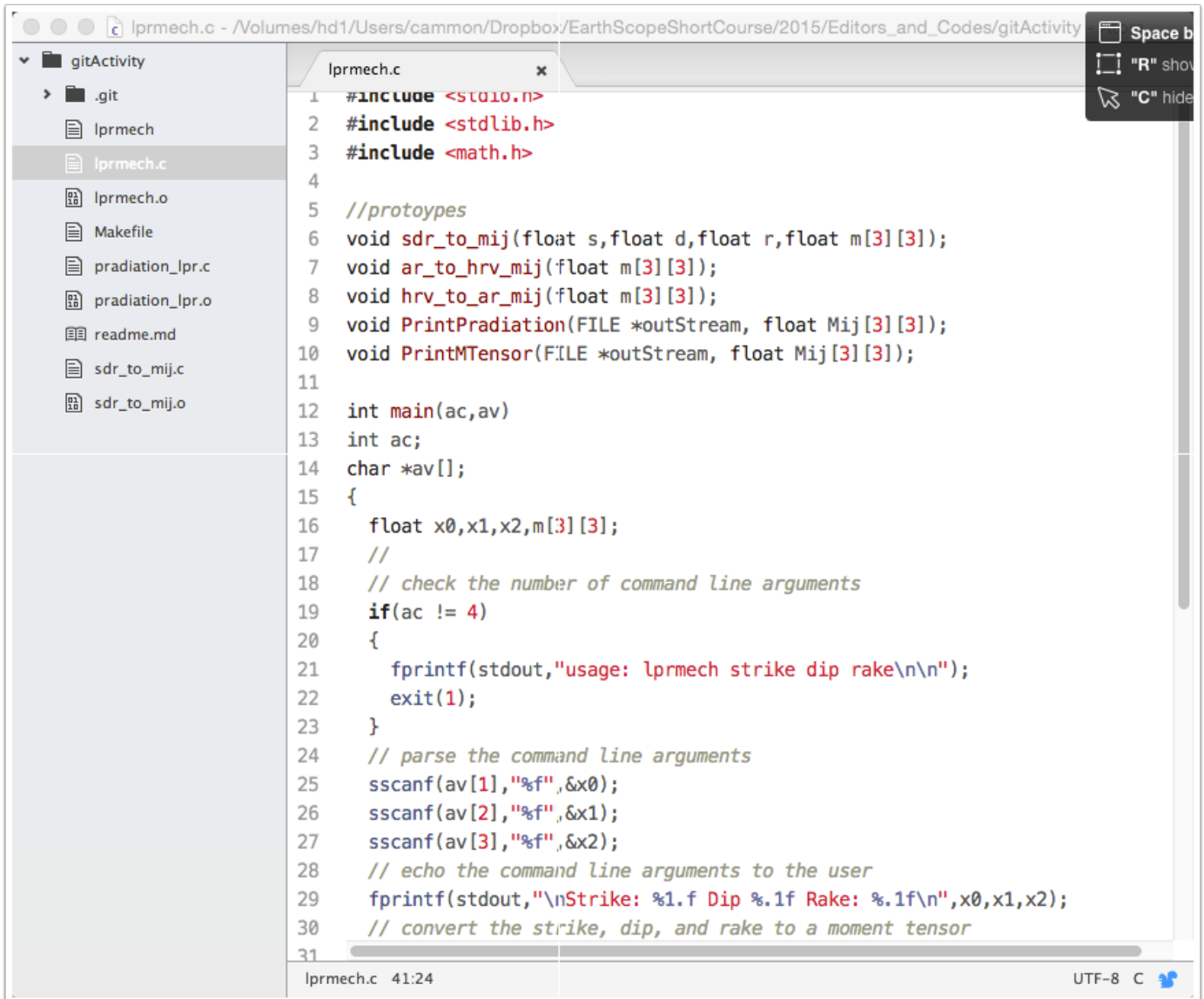
        lprmech
        lprmech.o
        pradiation_lpr.o
        sdr_to_mij.o

no changes added to commit (use "git add" and/or "git commit -a")
gitActivity > 
```

Recognize the change is all git has done so far. If we want to commit this change, we need to stage the modified file with ***git add***, and then commit the changes with ***git commit***. We are not done yet.

An Introduction to Version Control with Git

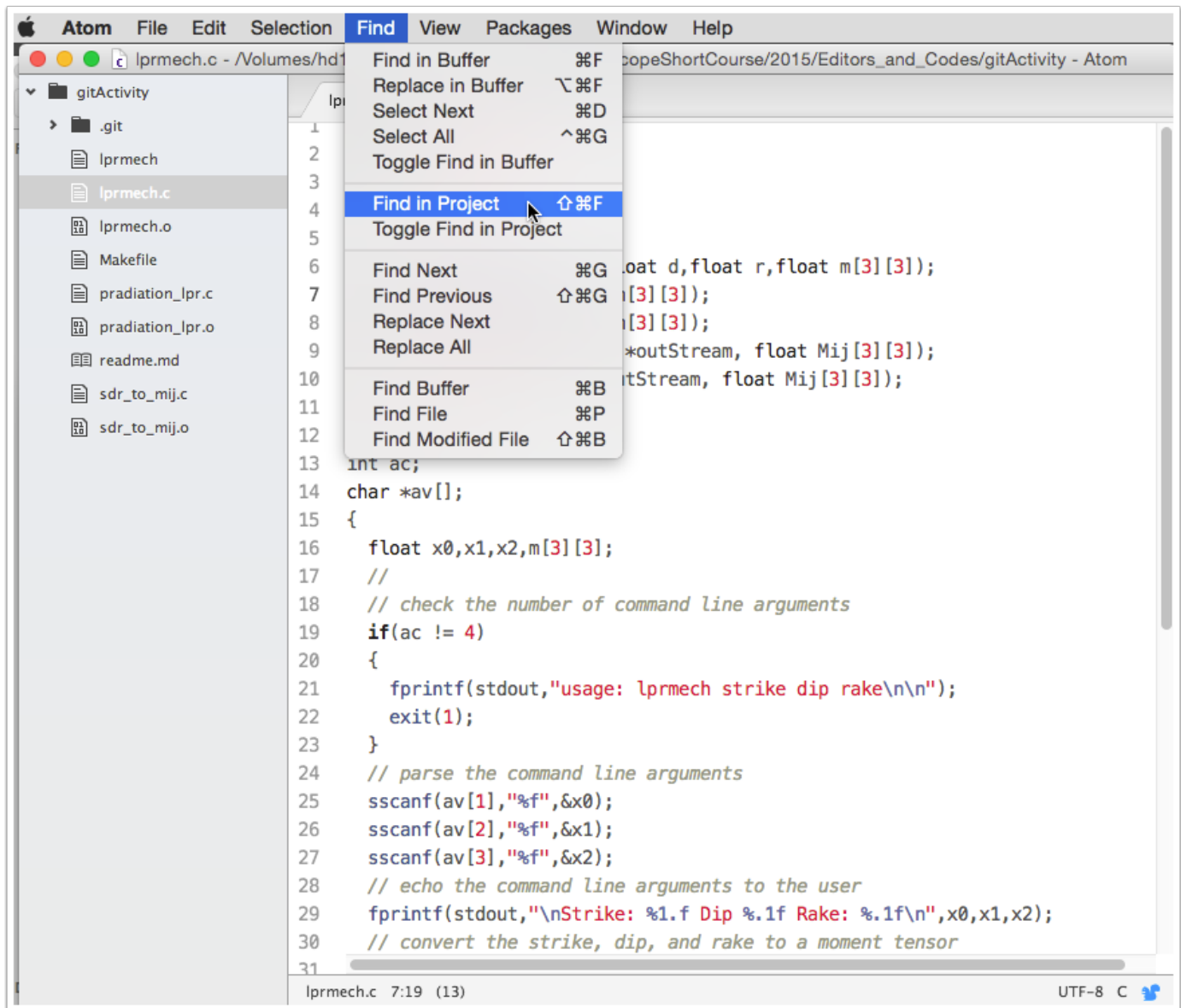
Two other references to the Harvard CMT remain in the code. Look at lines 7 and 8, which contain function names that include the abbreviation hrv. We might as well change those as well.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 //prototypes
6 void sdr_to_mij(float s,float d,float r,float m[3][3]);
7 void ar_to_hrv_mij(float m[3][3]);
8 void hrv_to_ar_mij(float m[3][3]);
9 void PrintPradiation(FILE *outStream, float Mij[3][3]);
10 void PrintMTensor(FILE *outStream, float Mij[3][3]);
11
12 int main(ac,av)
13 int ac;
14 char *av[];
15 {
16     float x0,x1,x2,m[3][3];
17     //
18     // check the number of command line arguments
19     if(ac != 4)
20     {
21         fprintf(stdout,"usage: lprmech strike dip rake\n\n");
22         exit(1);
23     }
24     // parse the command line arguments
25     sscanf(av[1],"%f",&x0);
26     sscanf(av[2],"%f",&x1);
27     sscanf(av[3],"%f",&x2);
28     // echo the command line arguments to the user
29     fprintf(stdout,"\nStrike: %1.f Dip %1.f Rake: %1.f\n",x0,x1,x2);
30     // convert the strike, dip, and rake to a moment tensor
31
```

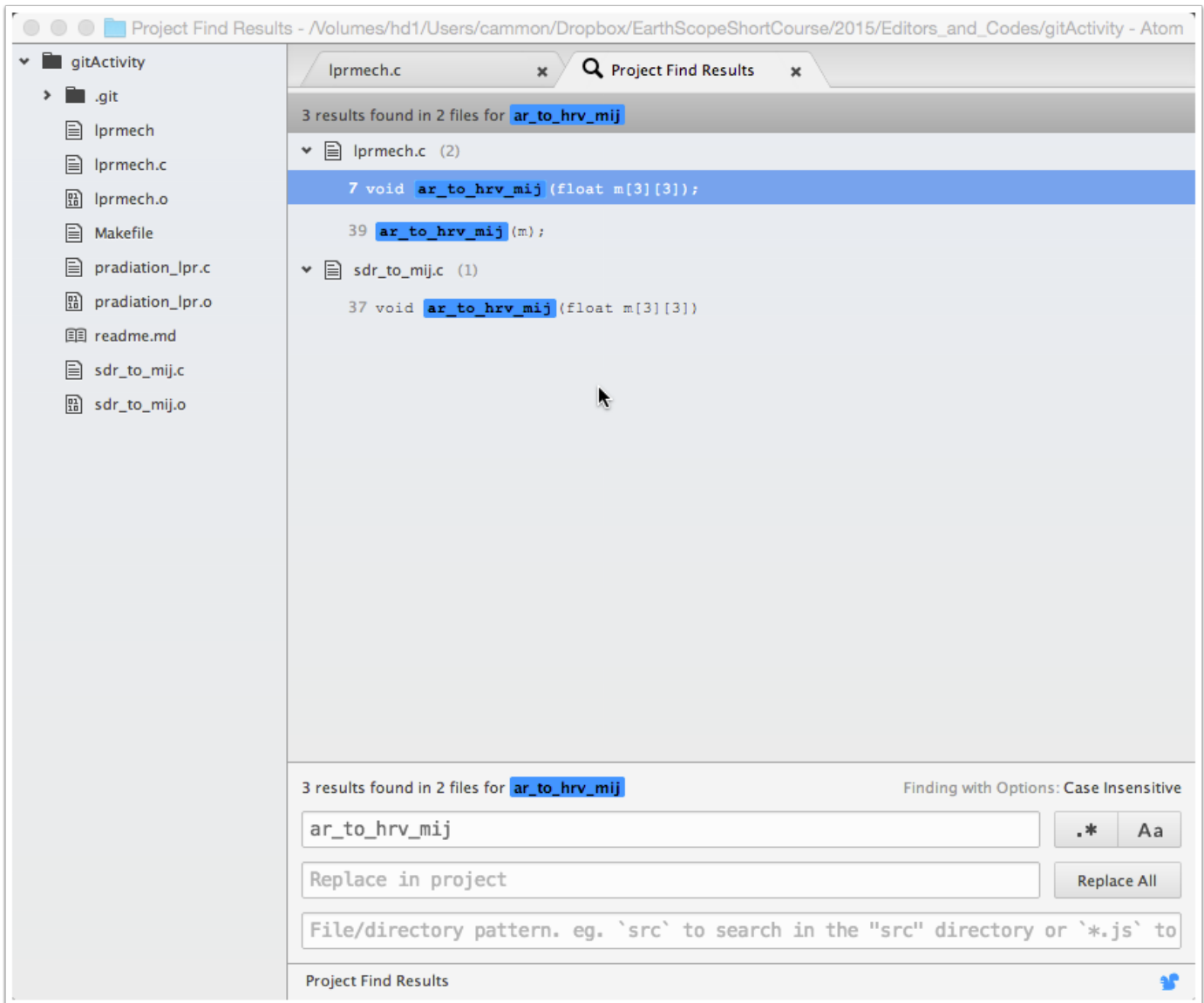
An Introduction to Version Control with Git

However, these function names appear in several files in our project. We'll have to change them in multiple files. To find all the instances of *ar_to_hrv_mij* in our project, select **Find in Project** from the **Find Menu**.



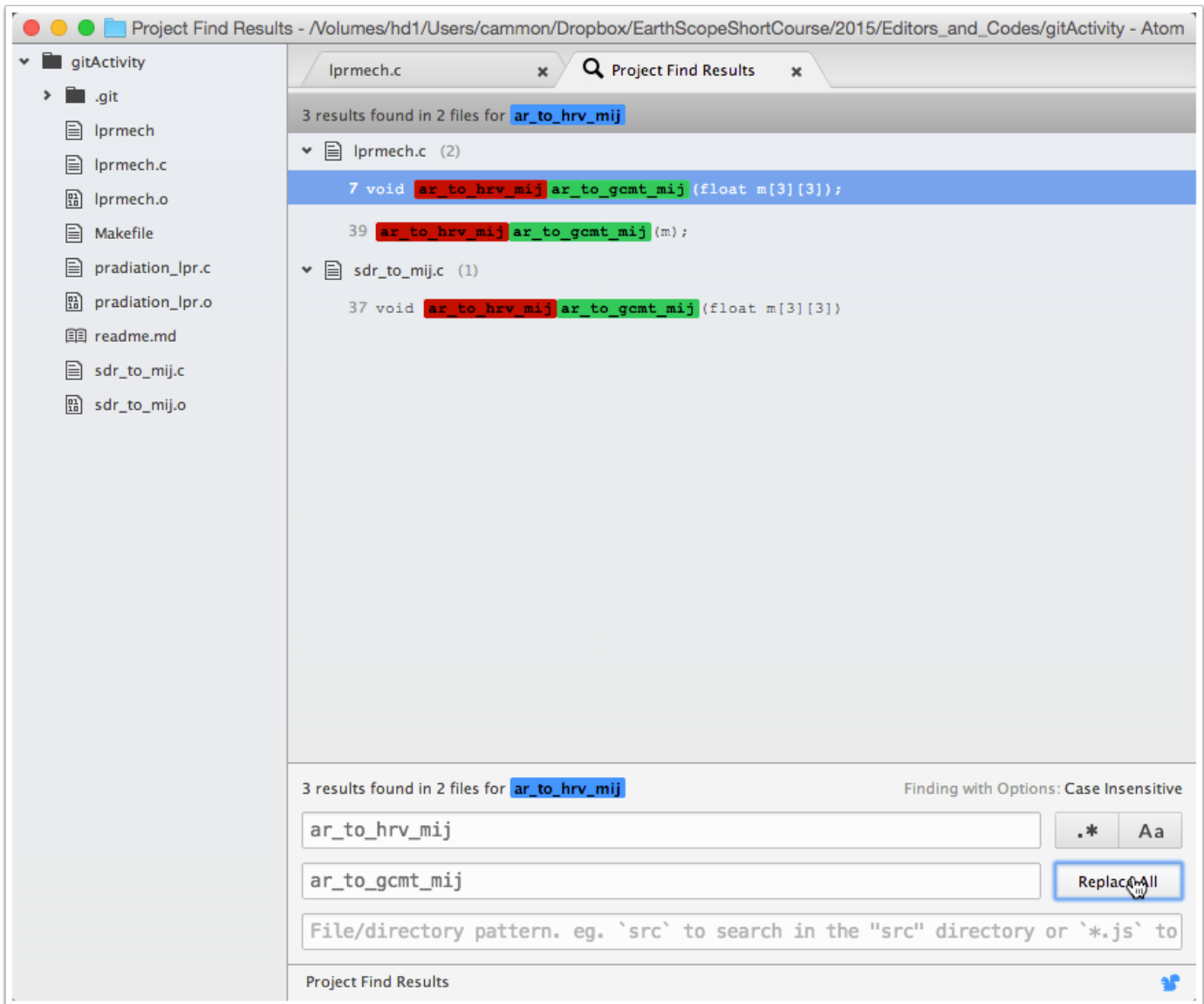
An Introduction to Version Control with Git

Here's the results of the search. The function name occurs in three places over two files.



An Introduction to Version Control with Git

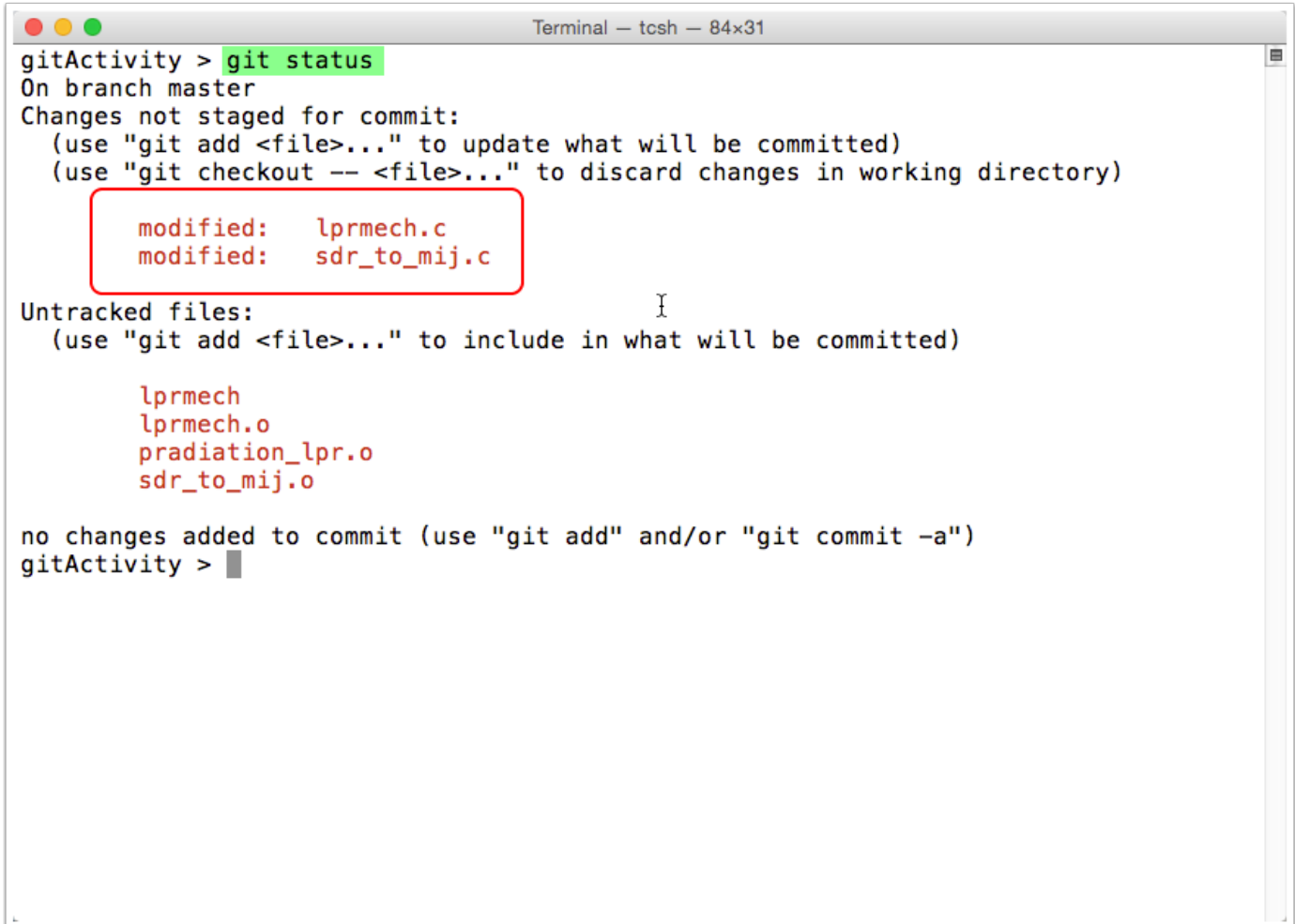
To change all occurrences of the function name, use the form at the bottom of the **Project Find Results** tab.



Repeat the procedure to change **hrv_to_ar_mij** to **gcmt_to_ar_mij**. Dismiss the find subwindow using the **Toggle Find in Project** menu item in the **Find Menu**.

An Introduction to Version Control with Git

The git status should now show two modified files.

A terminal window titled "Terminal - tcsh - 84x31" showing the output of the "git status" command. The output indicates that the user is on the "master" branch and that there are changes not staged for commit. Two files, "lprmech.c" and "sdr_to_mij.c", are listed as modified. Below this, untracked files are listed: "lprmech", "lprmech.o", "pradiation_lpr.o", and "sdr_to_mij.o". The terminal prompt is "gitActivity >".

```
gitActivity > git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   lprmech.c
        modified:   sdr_to_mij.c

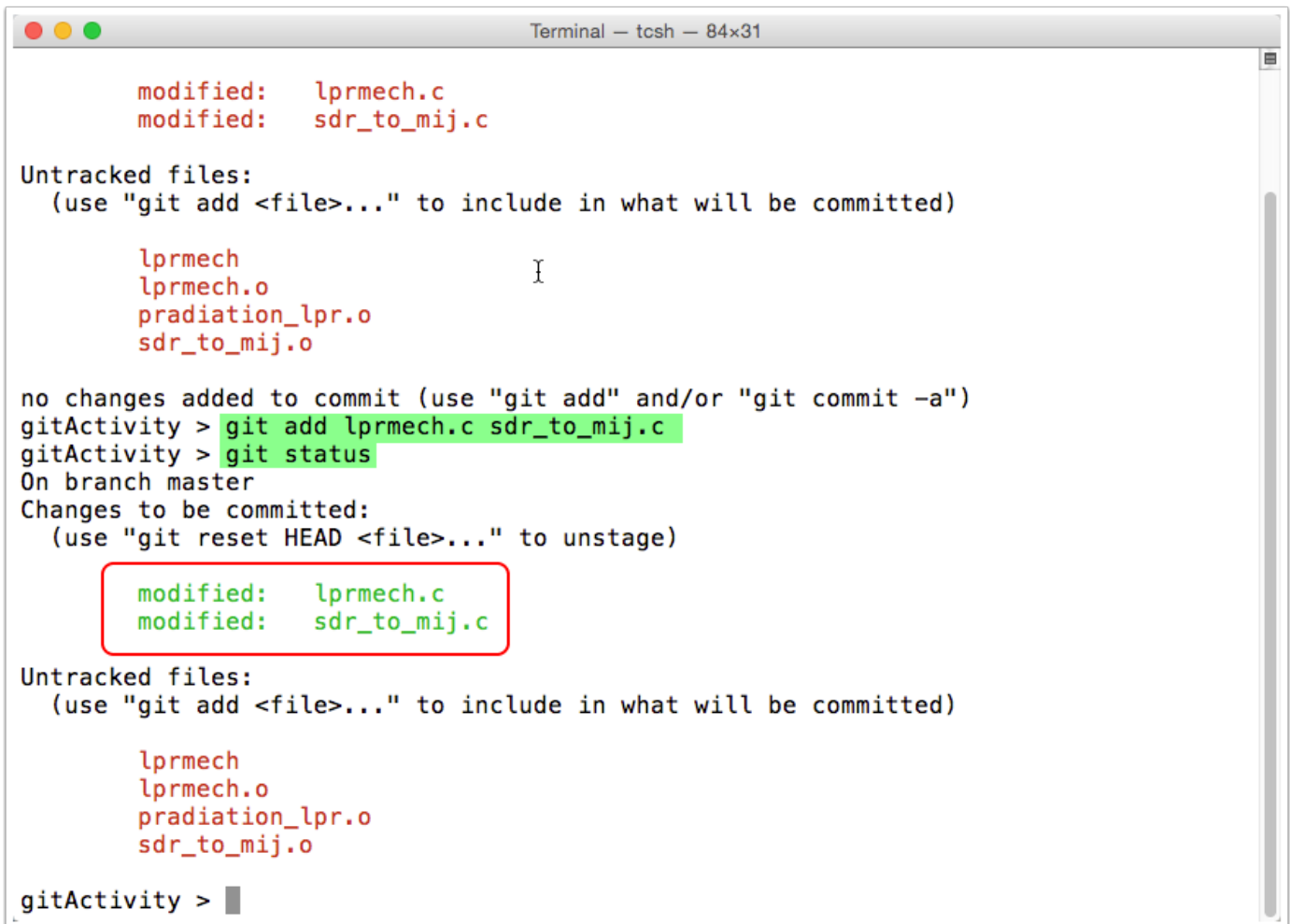
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        lprmech
        lprmech.o
        pradiation_lpr.o
        sdr_to_mij.o

no changes added to commit (use "git add" and/or "git commit -a")
gitActivity >
```


An Introduction to Version Control with Git

Before staging and committing the changes, **make** the executable and test it. You should see a correct output, including the label "GCMT" in place of Harvard. Once you are sure that the codes work, stage the files for the commit using **git add**.

A terminal window titled "Terminal — tcsh — 84x31" showing the output of a 'git status' command. The output indicates that 'lprmech.c' and 'sdr_to_mij.c' are modified files. It lists untracked files: 'lprmech', 'lprmech.o', 'pradiation_lpr.o', and 'sdr_to_mij.o'. The user then runs 'git add lprmech.c sdr_to_mij.c', which stages the modified files. The subsequent 'git status' command shows that 'lprmech.c' and 'sdr_to_mij.c' are now staged for commit. The terminal prompt is 'gitActivity >'.

```
Terminal — tcsh — 84x31

modified:   lprmech.c
modified:   sdr_to_mij.c

Untracked files:
(use "git add <file>..." to include in what will be committed)

lprmech
lprmech.o
pradiation_lpr.o
sdr_to_mij.o

no changes added to commit (use "git add" and/or "git commit -a")
gitActivity > git add lprmech.c sdr_to_mij.c
gitActivity > git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   lprmech.c
    modified:   sdr_to_mij.c

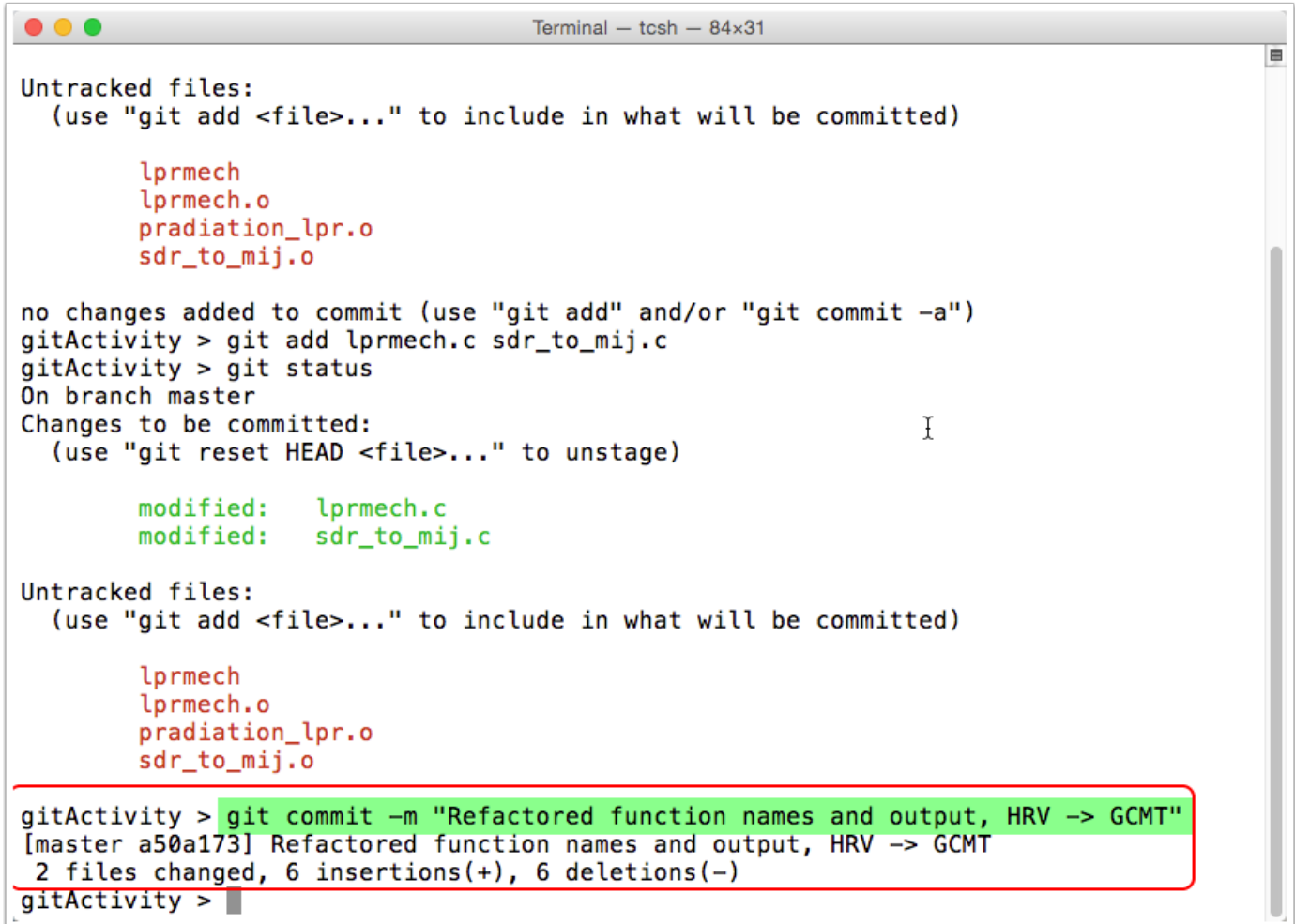
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    lprmech
    lprmech.o
    pradiation_lpr.o
    sdr_to_mij.o

gitActivity > 
```

An Introduction to Version Control with Git

Then commit the changes using ***git commit -m "description..."***.



```
Terminal — tcsh — 84x31

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    lprmech
    lprmech.o
    pradiation_lpr.o
    sdr_to_mij.o

no changes added to commit (use "git add" and/or "git commit -a")
gitActivity > git add lprmech.c sdr_to_mij.c
gitActivity > git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   lprmech.c
    modified:   sdr_to_mij.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    lprmech
    lprmech.o
    pradiation_lpr.o
    sdr_to_mij.o

gitActivity > git commit -m "Refactored function names and output, HRV -> GCMT"
[master a50a173] Refactored function names and output, HRV -> GCMT
 2 files changed, 6 insertions(+), 6 deletions(-)
gitActivity >
```

An Introduction to Version Control with Git

Now we have two commits (or versions of our codes stored within the git repository). you can see them with ***git log***.

```
Terminal — tcsh — 84x31
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   lprmech.c
    modified:   sdr_to_mij.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    lprmech
    lprmech.o
    pradiation_lpr.o
    sdr_to_mij.o

gitActivity > git commit -m "Refactored function names and output, HRV -> GCMT"
[master a50a173] Refactored function names and output, HRV -> GCMT
 2 files changed, 6 insertions(+), 6 deletions(-)
gitActivity > git log
commit a50a1732ef27236f022d3a9669297ba81ed0826f
Author: Charles J. Ammon <charlesammon@psu.edu>
Date:   Sat Aug 1 17:49:37 2015 -0400

    Refactored function names and output, HRV -> GCMT

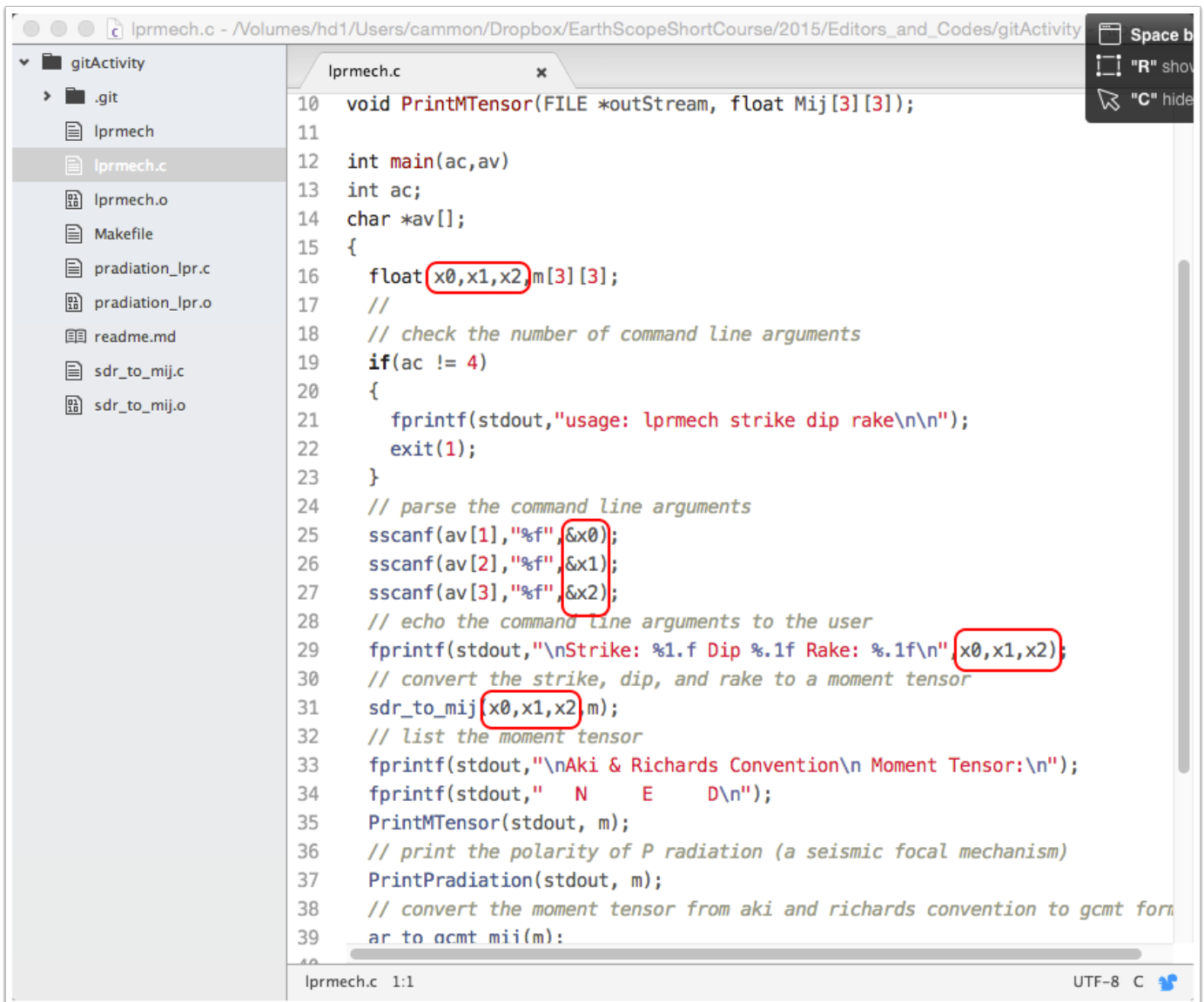
commit 9c26ce419706a6b3c9702e0ea79ec3e487e0d024
Author: Charles J. Ammon <charlesammon@psu.edu>
Date:   Sat Aug 1 16:03:40 2015 -0400

    Initial commit, functioning code.
gitActivity >
```

An Introduction to Version Control with Git

Multiple Cursors in Atom

Let's make one more change to the codes. If you look closely, the variables that are used to represent strike, dip, and rake are the uninformative x0, x1, and x2. This is not a good programming practice.



```
10 void PrintMTensor(FILE *outStream, float Mij[3][3]);
11
12 int main(ac,av)
13 int ac;
14 char *av[];
15 {
16     float x0,x1,x2,m[3][3];
17     //
18     // check the number of command line arguments
19     if(ac != 4)
20     {
21         fprintf(stdout,"usage: lprmech strike dip rake\n\n");
22         exit(1);
23     }
24     // parse the command line arguments
25     sscanf(av[1],"%f",&x0);
26     sscanf(av[2],"%f",&x1);
27     sscanf(av[3],"%f",&x2);
28     // echo the command line arguments to the user
29     fprintf(stdout,"\nStrike: %1.f Dip %1.f Rake: %1.f\n",x0,x1,x2);
30     // convert the strike, dip, and rake to a moment tensor
31     sdr_to_mij(x0,x1,x2,m);
32     // list the moment tensor
33     fprintf(stdout,"\nAki & Richards Convention\n Moment Tensor:\n");
34     fprintf(stdout,"  N    E    D\n");
35     PrintMTensor(stdout, m);
36     // print the polarity of P radiation (a seismic focal mechanism)
37     PrintPradiation(stdout, m);
38     // convert the moment tensor from aki and richards convention to gcmt format
39     ar_to_acmt_mij(m);
40 }
```

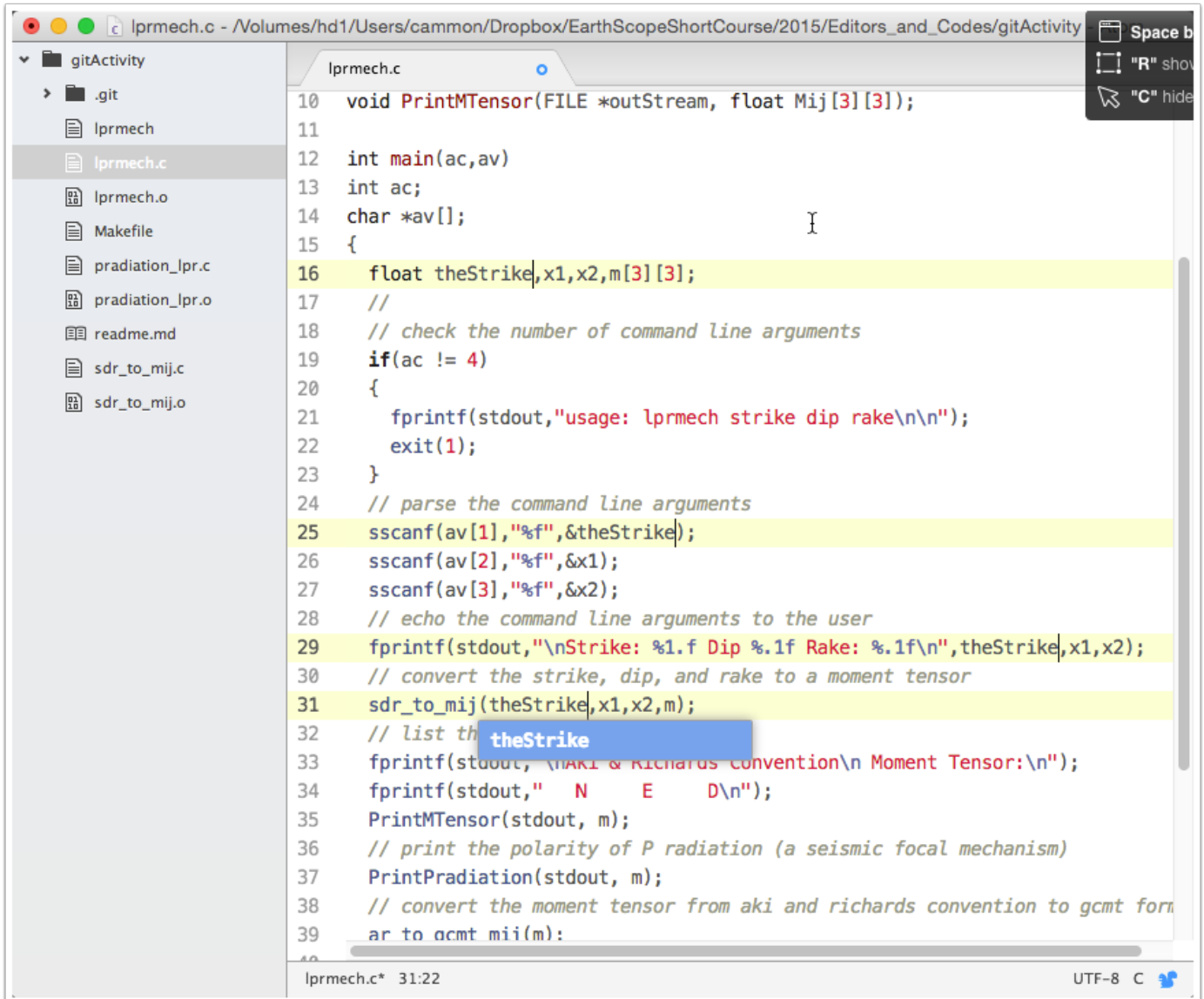
An Introduction to Version Control with Git

We'll use the multiple cursor feature in Atom to change these variable names to something more informative, like `theStrike`, `theDip`, `theRake`.

In the editor, select **`x0`** in line 16. Then enter the key combination **`ctrl-cmd-g`** (hold the control and command keys down and then press g). This will select all instances of **`x0`** in the file.

When all the **`x0's`** are selected, enter **`theStrike`**, to replace all instances of **`x0`** with **`theStrike`**.

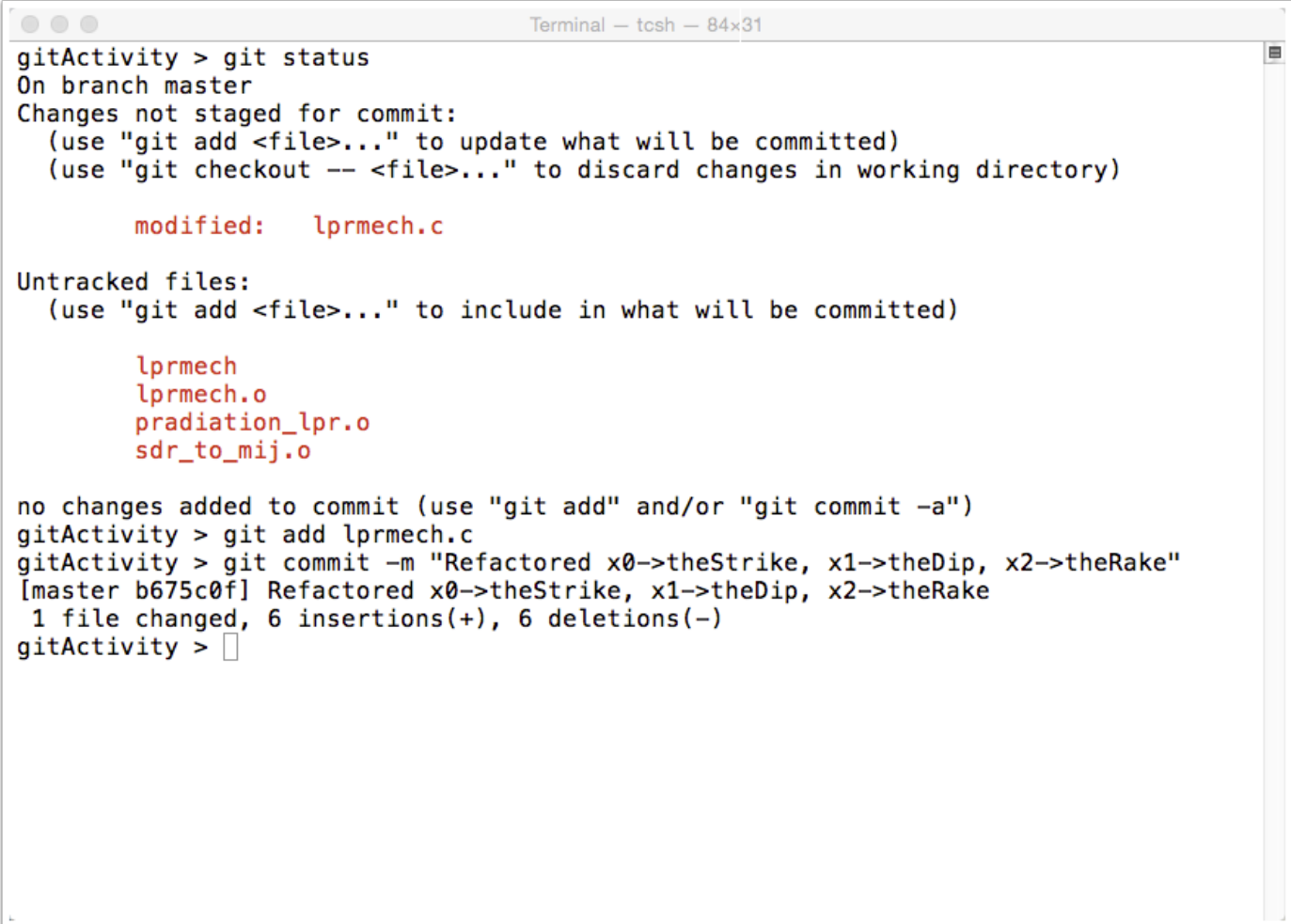
An Introduction to Version Control with Git



```
10 void PrintMTensor(FILE *outStream, float Mij[3][3]);
11
12 int main(ac, av)
13 int ac;
14 char *av[];
15 {
16     float theStrike, x1, x2, m[3][3];
17     //
18     // check the number of command line arguments
19     if(ac != 4)
20     {
21         fprintf(stdout, "usage: lprmech strike dip rake\n\n");
22         exit(1);
23     }
24     // parse the command line arguments
25     sscanf(av[1], "%f", &theStrike);
26     sscanf(av[2], "%f", &x1);
27     sscanf(av[3], "%f", &x2);
28     // echo the command line arguments to the user
29     fprintf(stdout, "\nStrike: %1.f Dip %1.f Rake: %1.f\n", theStrike, x1, x2);
30     // convert the strike, dip, and rake to a moment tensor
31     sdr_to_mij(theStrike, x1, x2, m);
32     // list the moment tensor
33     fprintf(stdout, "\nAKI & Richards convention\n Moment Tensor:\n");
34     fprintf(stdout, "    N    E    D\n");
35     PrintMTensor(stdout, m);
36     // print the polarity of P radiation (a seismic focal mechanism)
37     PrintPradiation(stdout, m);
38     // convert the moment tensor from aki and richards convention to gcmt format
39     ar_to_acmt_mii(m);
40 }
```

An Introduction to Version Control with Git

Repeat the procedure for the x1 and x2 variables and save the results. The stage the file, and commit the change into the git repository.



```
Terminal — tcsh — 84x31
gitActivity > git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

       modified:   lprmech.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)

       lprmech
       lprmech.o
       pradiation_lpr.o
       sdr_to_mij.o

no changes added to commit (use "git add" and/or "git commit -a")
gitActivity > git add lprmech.c
gitActivity > git commit -m "Refactored x0->theStrike, x1->theDip, x2->theRake"
[master b675c0f] Refactored x0->theStrike, x1->theDip, x2->theRake
 1 file changed, 6 insertions(+), 6 deletions(-)
gitActivity > 
```

An Introduction to Version Control with Git

Now we have three commits in the git repository.

A terminal window titled "Terminal - tcsh - 84x31" showing the output of the 'git log' command. The output lists three commits in reverse chronological order. Each commit entry includes a commit hash, the author's name and email, the date and time, and a description of the changes. The first commit (top) is for refactoring variables, the second for refactoring function names, and the third is the initial commit.

```
gitActivity > git log
commit b675c0f509183dbb9ad1deb60ed2df85ac01edb0
Author: Charles J. Ammon <charlesammon@psu.edu>
Date: Sat Aug 1 18:03:11 2015 -0400

    Refactored x0->theStrike, x1->theDip, x2->theRake

commit a50a1732ef27236f022d3a9669297ba81ed0826f
Author: Charles J. Ammon <charlesammon@psu.edu>
Date: Sat Aug 1 17:49:37 2015 -0400

    Refactored function names and output, HRV -> GCMT

commit 9c26ce419706a6b3c9702e0ea79ec3e487e0d024
Author: Charles J. Ammon <charlesammon@psu.edu>
Date: Sat Aug 1 16:03:40 2015 -0400

    Initial commit, functioning code.
gitActivity > 
```

Undoing

Just about anything in the git repository can be undone and you can recover earlier versions or the code, but you have to be careful when doing so. You can revert any file that has not yet been committed using

git checkout <filename>

An Introduction to Version Control with Git

If you've already committed changes that you want to undo, then you use

git reset <last good commit hash>

Be careful with that, it will revert to the referenced commit. Of course if you have to recover from a reset, you can, but things can get confusing. You can use ***git reflog*** and ***git reset*** or ***git checkout***. For a complete description of the process, see

<https://github.com/blog/2019-how-to-undo-almost-anything-with-git>.

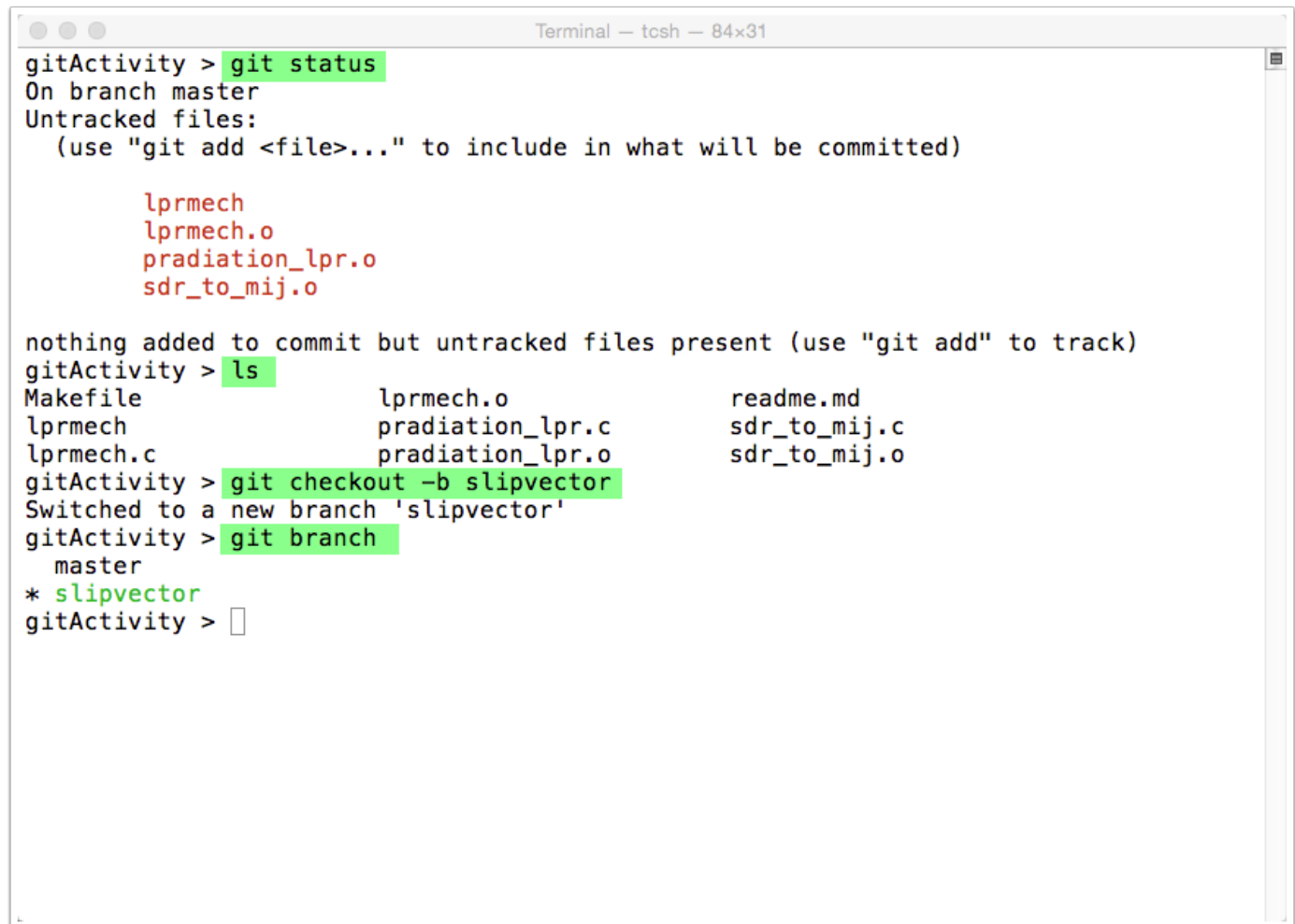
Right now, our changes are probably worth keeping, so we won't revert anything. But suppose that we wanted to extend the program, say by adding a function to compute the stress axes and slip vector for the specified strike, dip, and rake.

An Introduction to Version Control with Git

Branches

Git provides a superb facility for experimenting with the code. We can create a new branch of the code, make changes, test them, decide whether to keep them, and then later merge the experimental branch with the "master" branch that we just created.

We start by "checking out" the existing code into a new branch, which we'll call **slipvector**. Specifically, we create the new branch with the **git checkout** command, and then list the branches with the **git branch** command. The asterisk indicates that we are now working with the **slipvector** branch.

A terminal window titled "Terminal - tcsh - 84x31" showing a series of Git commands and their outputs. The user is in a directory named "gitActivity". The commands and outputs are as follows:

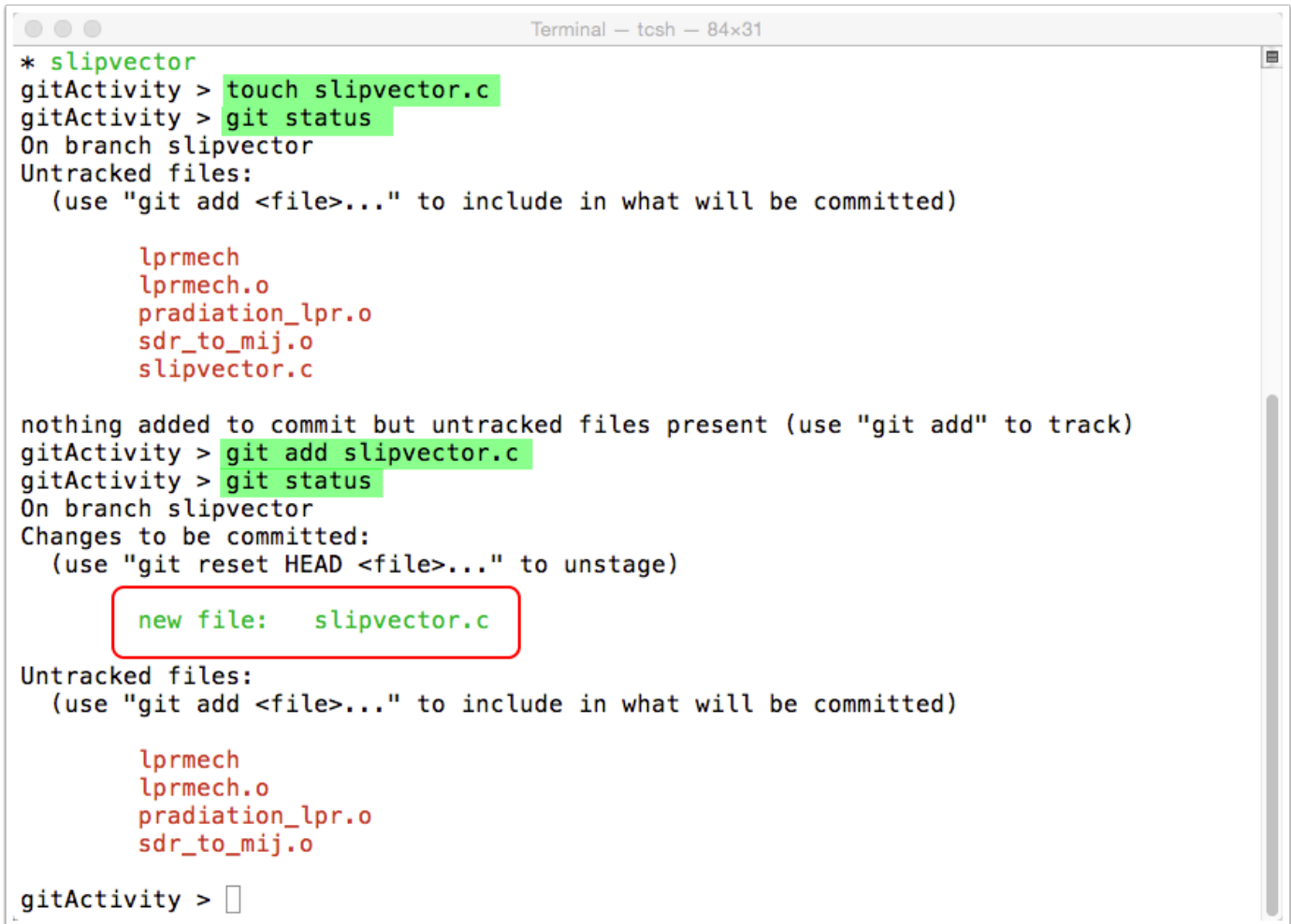
```
gitActivity > git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    lprmech
    lprmech.o
    pradiation_lpr.o
    sdr_to_mij.o

nothing added to commit but untracked files present (use "git add" to track)
gitActivity > ls
Makefile                lprmech.o               readme.md
lprmech                  pradiation_lpr.c        sdr_to_mij.c
lprmech.c                pradiation_lpr.o        sdr_to_mij.o
gitActivity > git checkout -b slipvector
Switched to a new branch 'slipvector'
gitActivity > git branch
  master
* slipvector
gitActivity > 
```

An Introduction to Version Control with Git

We can create a file called ***slipvector.c*** and add it to the slipvector branch of the code using ***git add***. Use ***git status*** to check the state of the repository.

A terminal window titled "Terminal - tcsh - 84x31" showing a series of Git commands and their outputs. The user is on the 'slipvector' branch. They create a new file 'slipvector.c' using 'touch'. Then they run 'git status', which shows 'slipvector.c' as an untracked file. Next, they run 'git add slipvector.c' to stage the file. A second 'git status' shows 'slipvector.c' as a change to be committed. The output 'new file: slipvector.c' is highlighted with a red box. Finally, they run 'git status' again, which shows no changes to be committed, only untracked files.

```
Terminal - tcsh - 84x31
* slipvector
gitActivity > touch slipvector.c
gitActivity > git status
On branch slipvector
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    lprmech
    lprmech.o
    pradiation_lpr.o
    sdr_to_mij.o
    slipvector.c

nothing added to commit but untracked files present (use "git add" to track)
gitActivity > git add slipvector.c
gitActivity > git status
On branch slipvector
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   slipvector.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    lprmech
    lprmech.o
    pradiation_lpr.o
    sdr_to_mij.o

gitActivity > 
```

An Introduction to Version Control with Git

We can then commit the change (the addition of the slipvector.c file to the slipvector branch).

A terminal window titled "Terminal - tcsh - 84x30" showing a series of Git commands and their outputs. The user performs a commit with the message "Starting slipvector development.", then runs 'git log' to view the commit history. The log shows four commits by Charles J. Ammon, with the most recent being the initial commit. The terminal output is as follows:

```
gitActivity > git commit -m "Starting slipvector development."
[slipvector 7b7df24] Starting slipvector development.
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 slipvector.c
gitActivity > git log
commit 7b7df2417411398d497dc2b57dc13aef05453736
Author: Charles J. Ammon <charlesammon@psu.edu>
Date: Sat Aug 1 20:14:56 2015 -0400

    Starting slipvector development.

commit b675c0f509183dbb9ad1deb60ed2df85ac01edb0
Author: Charles J. Ammon <charlesammon@psu.edu>
Date: Sat Aug 1 18:03:11 2015 -0400

    Refactored x0->theStrike, x1->theDip, x2->theRake

commit a50a1732ef27236f022d3a9669297ba81ed0826f
Author: Charles J. Ammon <charlesammon@psu.edu>
Date: Sat Aug 1 17:49:37 2015 -0400

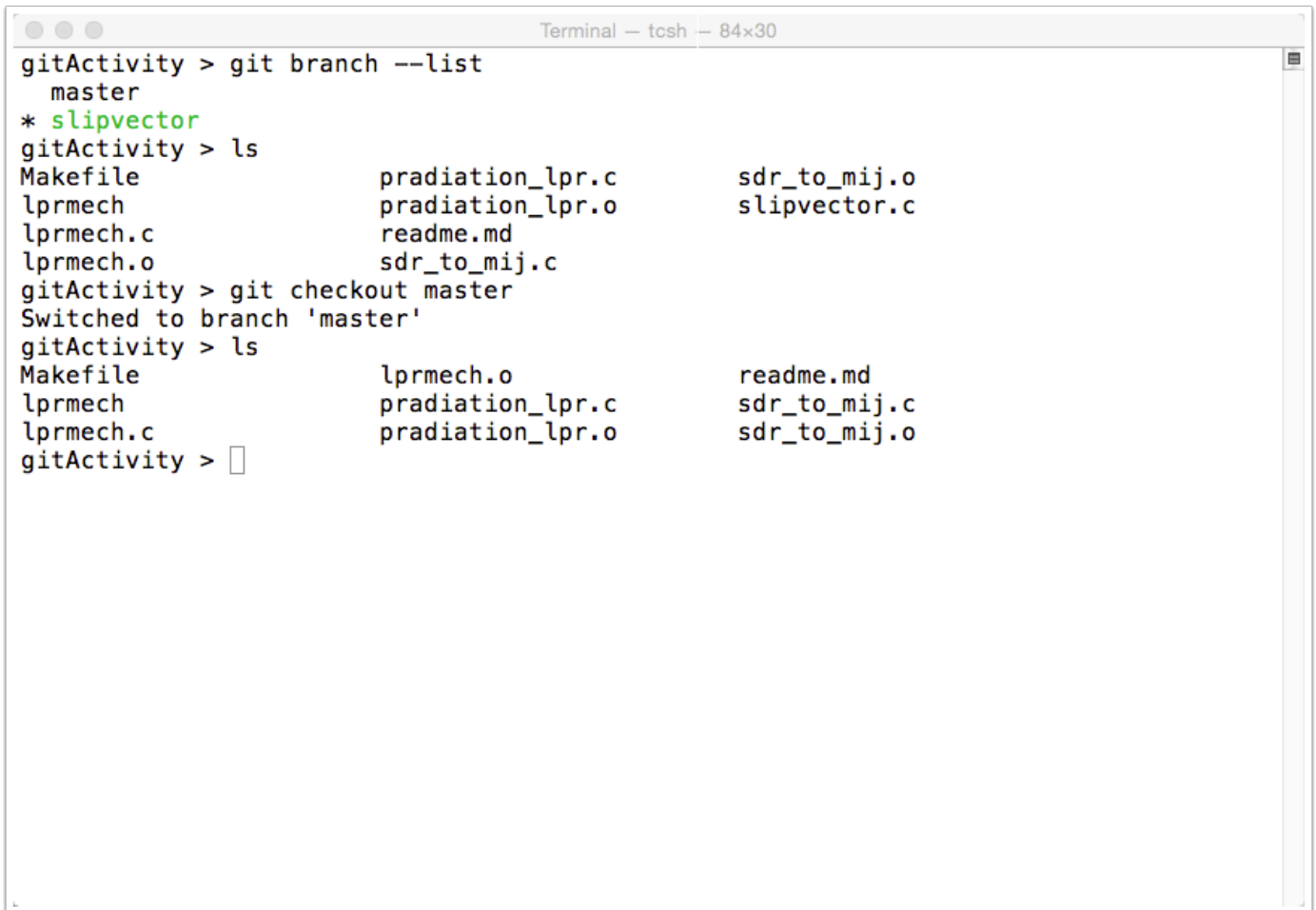
    Refactored function names and output, HRV -> GCMT

commit 9c26ce419706a6b3c9702e0ea79ec3e487e0d024
Author: Charles J. Ammon <charlesammon@psu.edu>
Date: Sat Aug 1 16:03:40 2015 -0400

    Initial commit, functioning code.
gitActivity > 
```

An Introduction to Version Control with Git

We can switch back and forth between the branches using ***git checkout*** to checkout a particular branch. Note that when I return to the master branch, the file ***slipvector.c*** is not in the working directory.



```
Terminal — tcsh — 84x30
gitActivity > git branch --list
  master
* slipvector
gitActivity > ls
Makefile      pradiation_lpr.c      sdr_to_mij.o
lprmech       pradiation_lpr.o      slipvector.c
lprmech.c     readme.md
lprmech.o     sdr_to_mij.c
gitActivity > git checkout master
Switched to branch 'master'
gitActivity > ls
Makefile      lprmech.o      readme.md
lprmech       pradiation_lpr.c  sdr_to_mij.c
lprmech.c     pradiation_lpr.o  sdr_to_mij.o
gitActivity > 
```

Once we complete development of the `slipvector.c` code, test it, and are satisfied that it works, we can ***git merge*** the `slipvector` branch with the `master` branch. We can always checkout the `master` branch if someone needs the code before we complete the addition of the `slipvector` feature (or we mess everything up, we can delete the experimental `slipvector` branch and start over).

An Introduction to Version Control with Git

Exploring git and <http://github.com>

We have barely scratched the surface of git's capabilities and applications. You can use it with latex documents (it's not the most efficient situation, but it will work), or any other text files that you maintain.

You should have no problem finding tutorials and information on git. Explore them because git will help you stay organized, track your software development and help you create more reproducible results.

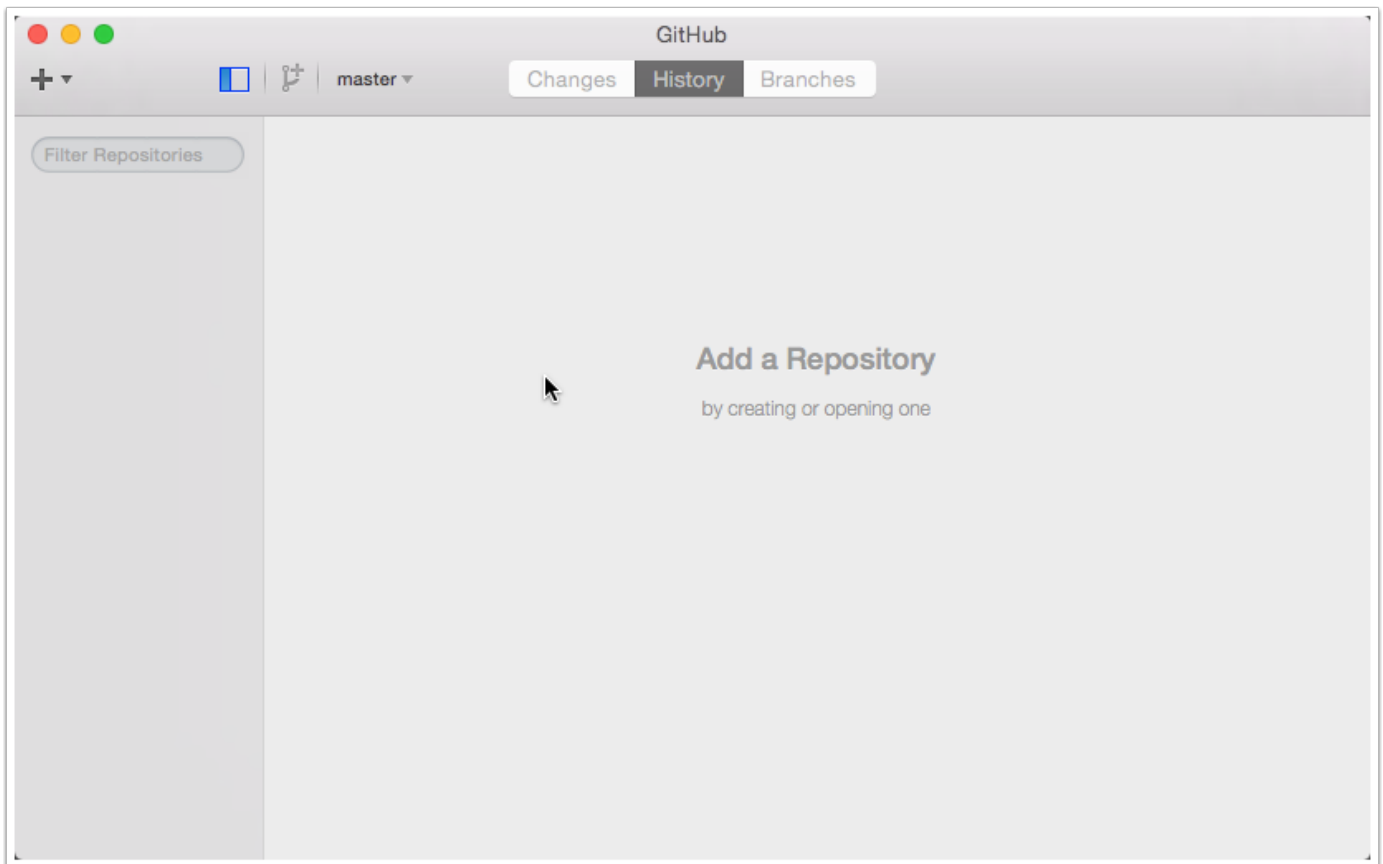
We have discussed local repositories that are located on the computer that we are using. The broad computing community has largely adopted cloud-based repositories that form a valuable resource, and help make sure the codes are backed up, and easy to install on new systems. Browse or search <http://github.com> for an incredible number of projects. If you haven't visited that site, you may be surprised to see how much code is being shared with git repositories.

An Introduction to Version Control with Git

The GitHub App

Using the command line is a good way to start using git, and may be the only way to use it when you are remotely logged into a machine. However, there are GUI tools that can allow to use git more conveniently.

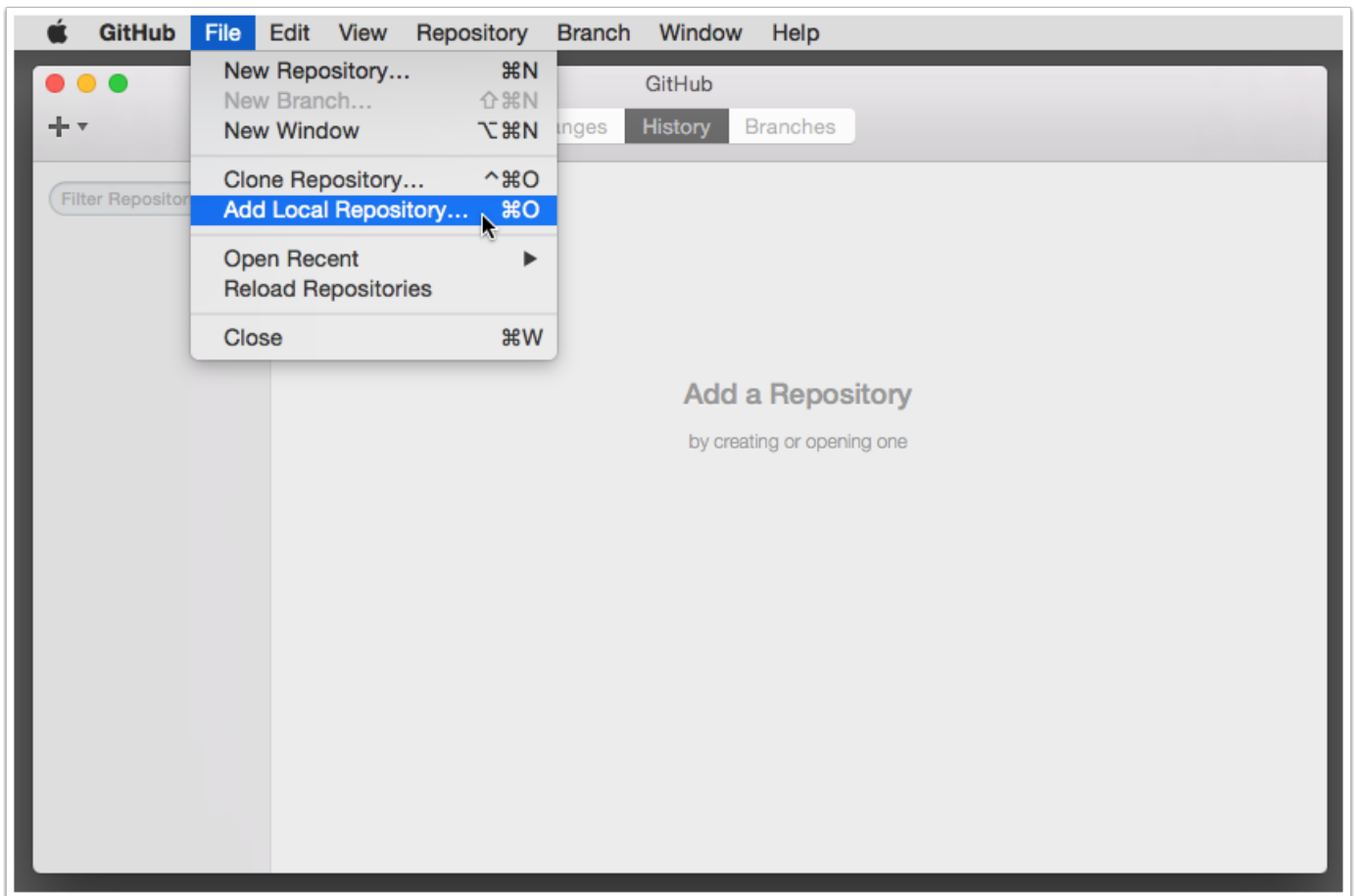
Connect to <http://mac.github.com> and download GitHub for the mac. Then launch the application.



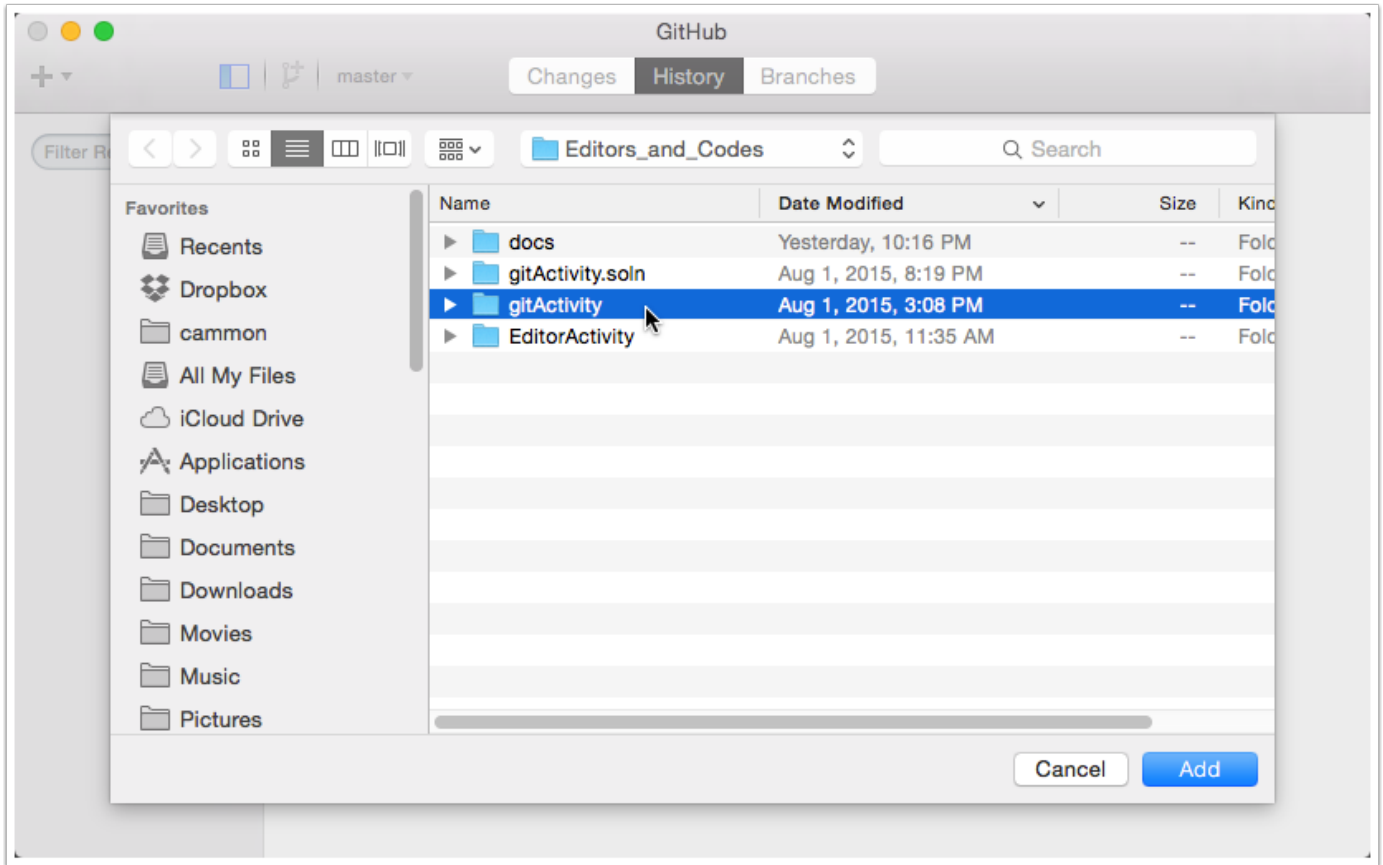
An Introduction to Version Control with Git

Open the `lprmech` git repository that you just created.

Select Add Local Repository from the File Menu. Then navigate to the folder containing your recently created `lprmech` repository. I was working in the folder `gitActivity`, so I selected that folder.



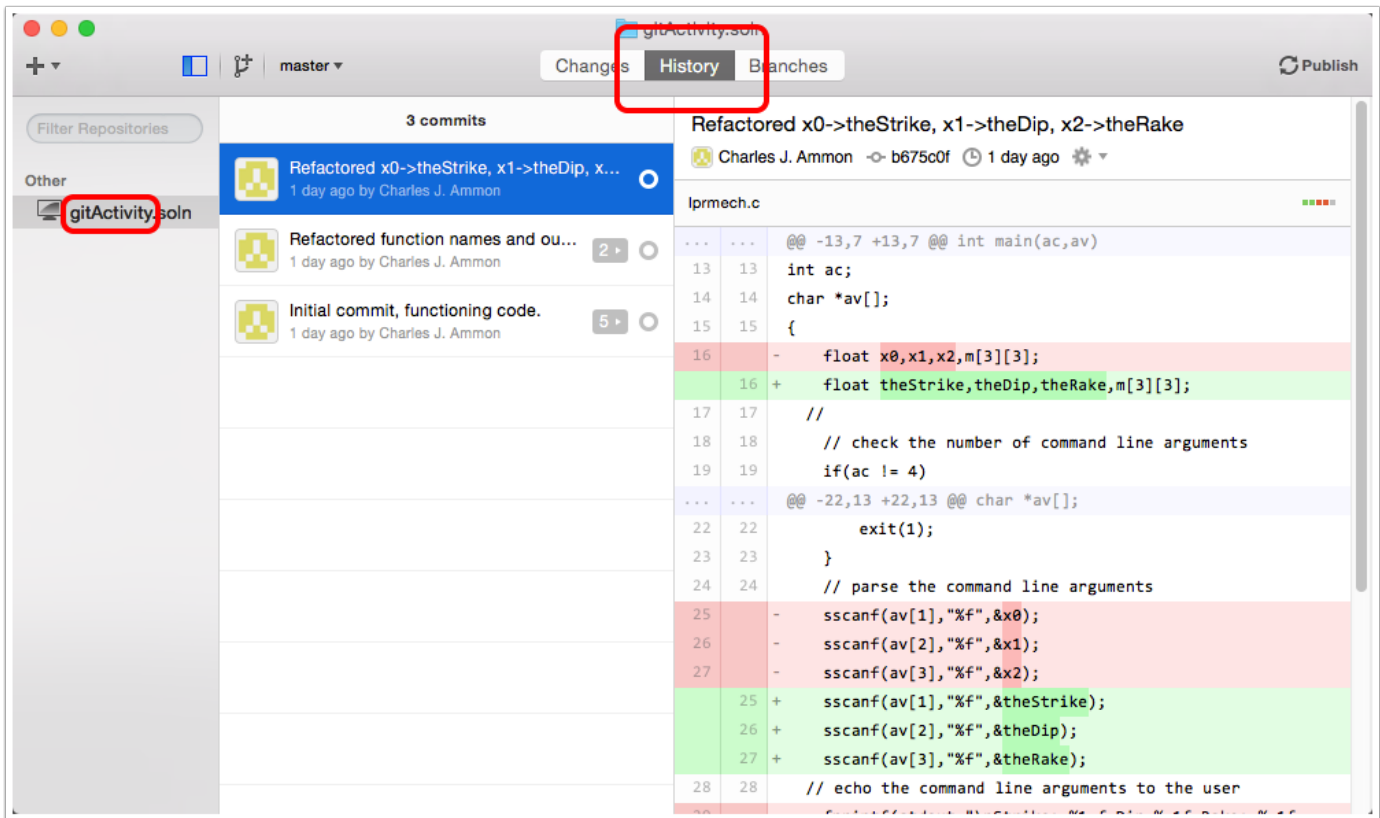
An Introduction to Version Control with Git



An Introduction to Version Control with Git

You'll see a list of repositories on the left, the list of commits in the selected repository in the middle, and then a comparison of the changes included in the selected commit on the right. You can browse the history of the file quite conveniently with this tool.

Select some of the commits to review your activity.



The GitHub app also lets you publish your repository to <http://github.com>, which allows you to share your work with others and back up your work to the cloud. github memberships are free, and although most repositories are public, they allow a few private repositories for educational work, and more if you are willing to pay a small annual fee.

An Introduction to Version Control with Git

Penn State runs their own online git service based on the <http://git.psu.edu> software that members of Penn State community can use to create and store repositories. I am sure that your institution likely has a similar resource.

We don't have time to explore the many features and uses of github, but I recommend you make some time when you can. To get started, go to github.com and search for **user:USGS**, you'll see many software items being distributed by one part of our community. For general exploration, click the **explore** link on the github home page.