# Astro 585 Lab/Homework #3 (Version Control, Testing, Documentation, Disk I/O)

In this lab exercise, you'll first experiment with different ways of reading and writing data to/from disk. However, the main goal of this lab exercise is to get you into the habit of: 1) using version control, 2) writing small testable functions that work together to accomplish a larger task, 3) writing tests for each of your functions and 4) writing useful documentation for each function.

**1. Start using git for *Version Control***

1a. Setup git. If the computer you're using doesn't have git installed, download and install it. It's probably easiest if you follow the tutorial at https://help.github.com/articles/set-up-git to set your default username and password, optionally enable password caching. The directions here are for the command line interface, but there are also free GUI interfaces for Windows and Mac that you could use.

[When creating your github account, you are welcome to use a code name (rather than a username that gives away your real name) if you'd like to remain anonymous when we discuss each other's code.]

Proceed to the next page to initialize a local git repository, commit a file to your local repository, create a remote git repository, set your local repository to be linked to the remote repository, push your local commit to the remote repository. Now use the web interface to find your repository and verify that the file you committed appears there. For example,

```
my_root)M-Osmond216-220:~ ebf11$ git config --global user.name "AstroCat17"
(my_root)M-Osmond216-220:~ ebf11$ git config --global user.email "astrocat17@astro585.edu"
(my_root)M-Osmond216-220:~ ebf11$ git config --global credential.helper osxkeychain
Create Repository at https://github.com/new (you may need to create account first)
(my_root)M-Osmond216-220:Hello-World ebf11$ vi README
(my_root)M-Osmond216-220:Hello-World ebf11$
(my_root)M-Osmond216-220:HelloWorld ebf11$ git add README.md
(my_root)M-Osmond216-220:HelloWorld ebf11$ git commit -m "first commit"
(my_root)M-Osmond216-220:HelloWorld ebf11$ git remote add origin https://github.com/eford/HelloWorld.git
(my_root)M-Osmond216-220:HelloWorld ebf11$ git push -u origin master
```

[Again, you may want to give something other than your real name and email when setting up git. You can specify your code name and a fake email like in the example above. See last page for more notes about setting up git]

1b. Place the source code for functions that you write for this (and future) assignments into files that are part of a git repository. Edit the text files with a text editor like emacs, AquaMacs, or the editor built into JuliaStudio. Load those into your IJulia or JuliaStudio session with the include("path/filename.jl"); command. As you work on the rest of the assignment, add files for each new set of functions and commit code changes to your local repository often. At least once a day (and when you've submitted the assignment), push the commits from your local repository to github (or you're welcome to use some other git server like bitbucket.org, if you prefer).

If you want to try something out that you may want to undo later, then create a branch. Once your code passes tests, a good strategy is to keep the master branch passing those tests, and let new code be localized to branches. If you decide it wasn't a good idea, go back to the master branch. Basically, get in the habit of using version control for your assignments, especially your class project, and hopefully for the

rest of your life.
Let me know github repository that you'll be using for this assignment, (so I can see that you're actually using it for version control, rather than just posting the completed assignment :).

**2. File I/O.**

Often you will perform one large calculation that generates lots of data and then repeatedly reanalyze it several different ways. That requires that your first large simulation writes lots of data to disk. Later you want to be able to read the data from disk and process it. In this exercise, you will write functions that print formatted data to the screen, write data to disk (either in ASCII, raw binary or structure binary formats) and read the data back from disk.

2a) Generate a vector of 1024 floating point numbers. Print an ASCII representation of them to STDOUT and report how long that takes. (You'll likely use the print or println functions and/or the @printf macro.)

2b) Write a function that opens a file, writes the array in an ASCII format to the file and closes the file. Write a second function that does the same, but reads in the array. (You could use the same functions as above, plus the functions open, readline, convert and close (or replace close with the end of a do block). Alternatively, you could use the writedlm or writecsv functions.) Remember to place these functions in a file that is part of a git repository. To load the functions in the file into IJulia, include("path/filename.jl").

2c) Time how long reading and writing the ASCII files takes (separately). How does writing to a file compare to printing to STDOUT (i.e., the default way command line programs write output text data to your screen)? How does the time required to read a file change if you read the same file multiple times in a row? Increase the size of the array to 1024*1024 floating point numbers and retime the writing and reading to/from a file.

2d) Write functions that open a file, write/read the data in binary to the file, close the file. Remember to place these functions in a file that is part of a git repository. Time how long it takes for both reading and writing of the 1024 and/or 1024^2 size arrays. If it's practical on your computer, try a significantly larger size like $10*1024^2$ or $100*1024^2$. (You'll likely use open, close (or a do block), and either read and write or serialize and deserialize.) How do these times compare to the time required for ASCII files? How does the resulting file size compare?

2e) Repeat 2d, but have the functions read and write the data to a binary file in a HDF5 format. For this, you'll need to use Julia's HDF5 package:

```
Pkg.add("HDF5")
using HDF5, JLD
```

Then you can either use the @save and @load macros (simplest) or one of the more flexible set of functions (e.g., jldopen, write, read, close; h5write and h5read; h5open and dictionary-like interface). Remember to place these functions in a file that is part of a git repository. How does the time required and file sizes compare to the other file formats?

2f) Write functions that let the caller read the value of any single array element from a file for each of the file formats above. Remember to place these functions in a file that is part of a git repository. When reading an element that's roughly mid-way through the file, how does the time required compare for the different methods?

2g) Under which circumstances would you be likely to use each file formats?

2h) [Optional] Perform similar performance tests for a structured text file format like XML or YAML and report how they compare in terms of performance. [Warning: I added this at the last minute in response to a student questions, so I haven't tried these myself, and you may run into troubles. My guess is that you'll want avoid the largest file sizes, as I suspect these will result in significantly larger file. Finish the rest of the assignment before spending time on part 2h. If even one or two students do this, I'll share the results with the class.]

**3. *Tests*:**
Import the test module that is part of the standard Julia language via “using Base.Test”.
Learn how the @test macro works by running code such as

```
@test 1==2
@test_approx_eq_eps(1.0,1.001,0.01)
@test_approx_eq_eps(1.0,1.001,0.0001)
```

For further information, see http://docs.julialang.org/en/latest/stdlib/test/

3a) Design and write a set of unit tests for each of your functions from part 2 of this assignment. Remember to place these functions in a file that is part of a git repository. Run your tests. Do your functions pass all of them? If not, correct the function (and tests if necessary) and rerun the tests. Remember to commit the changes to any functions to your git repository.

3b) What are the pre-conditions and post-conditions for each of the functions you used above?
Do your tests check the behavior when the pre-conditions are not satisfied? Do your tests check that the post-conditions are satisfied? If not, write more tests. Remember to commit any additions/changes to any functions to your git repository.

3c) Modify the above functions to include assertions that enforce the pre-conditions are met (and post-conditions, if appropriate). Retest your functions.

3d) Write a function that calculates the variance of a large array of numbers, using a one-pass algorithm when practical, but reverting to a two pass algorithm when there is a clear advantage. Note that the functions isnan(x), isinf(x) and isfinite(x) allow you to test whether a variable is set to NaN (or -NaN), infinity (or -infinity) or a finite value. For this part, use standard if statements rather than an error handler. (Hopefully, you can reuse functions written for a previous assignment, rather than reimplementing them. In the process, you may realize that you should add to the documentation, assertions and tests).

3e) Write one or more regression tests for the function in 3d. Run your tests. Does your function pass all of them? If not, correct the function (and tests if necessary) and rerun the tests.

3f) [Optional] Write another function that does the same thing, but using an error handler or a try...catch statement.

3g) Compare the performance of the original functions (from previous assignment) and the new functions you wrote for part e (and f, if you were able to do that).

3h) What are the implications for whether, when and how you plan to lookout for issues like NaN’s in your code?

**4  Documentation.**
4a.  Add in-line documentation to the functions that you used in exercise 2 and 3, following the guidelines discussed in the reading and class discussion.  (Don't forget to commit the changes to your git repository.)

4b. Rerun your tests from exercise 3.  If your functions no longer pass all the tests, then correct the problem.  If you're unsure what change caused things to break, use git to restore one or more previous versions to figure out exactly where the error first crept in.  In retrospect, is there anything you should have done differently to reduce the chance of introducing a bug?

## Notes about git:

If you've already setup a git repository from a different computer, then you can clone it to your local computer using a command like

```
git clone https://github.com/username/Spoon-Knife.git
```

where you replace the url with that of your repository (little box on the right below settings).
You'll likely still need to run the git config commands on the computer your setting up to accesss your new repository.