Aerospace Computing Laboratory
Prepared by Manuel R. López & Thomas D. Economon
November 2013

HiFiLES Quick Reference Guide

Developing HiFiLES through Git + GitHub

The repository for HiFiLES is being hosted on GitHub (`https://github.com`). GitHub is simply an online project hosting service with a useful web interface and additional tools to aid code development with Git as its backbone. Git is a version control system (VCS) which is similar to SVN, Mercurial, etc., and it helps organize the development of code over time by tracking changes.

To get started, you need to create a personal user account on GitHub (free) and to follow the basic setup instructions at `https://help.github.com/articles/set-up-git`. These instructions include how to get Git installed on your local machine. To sync up your local settings with GitHub, change the user.email and user.name variables for your local git configuration with

```
git config --global user.email "enterYourEmailAddressBetweenQuotationMarks"
git config --global user.name "enterYourNameBetweenQuotationMarks"
```

Note that the email address should be the one associated with your GitHub account. If already have a separate GitHub account setup on your computer, you can change the user settings for just the HiFiLES repository by `cd`'ing to your HiFiLES working directory, and using the above commands, but without the `--global` flag.

HiFiLES is currently under a private repository (this location will change if/when the code goes publicly open-source), and the main page for the project is located here: `https://github.com/fpalacios/HiFiLES`. The web interface is useful for viewing the recent commits to the code, changes to the code over time, or creating and viewing branches, for instance. To contribute to HiFiLES, an administrator for the project must add you as a member of the developer team with push and pull privileges (ask Francisco Palacios).

Most of the day-to-day development of the code will be done on your local machine at the command line using Git. After setting up Git and gaining access to the HiFiLES repository, create a local copy of the entire repository on your machine by cloning the version that is hosted on GitHub:

```
git clone https://github.com/fpalacios/HiFiLES.git
```

After cloning, you should have a new HiFiLES/ folder in your current working directory. Move into HiFiLES/ to see the project files and to start working with the code through Git. You can see the most recent changes by typing

```
git log
```

Typical Workflow with Git

Now that you have a local copy of HiFiLES from the GitHub repository, you can begin to make changes to the codebase. This section gives an example of the typical workflow for making changes to the code, committing them locally, and then pushing your changes to the remote GitHub repository. The basic steps are as follows:

1. Make changes to the existing files (using your favorite text editor or integrated development environment, IDE) or add local files or folders to be tracked and compared against the global repo files

   ```
   git add file1.cpp file2.cpp /folder_directory1 /folder_directory2
   ```

2. Check that your changes have been registered and/or the files that you want have been added added

   ```
   git status
   ```

3. Commit the changes to your local repository (not the global repository on GitHub) and leave a short descriptive message about your change.

```
git commit -am "Added some files and folders."
```

4. Merge local and global repositories. This command will attempt to merge your version of the code with the global version. Near the end of the merger process, git will tell you if everything has been merged successfully. If there are conflicts, it will tell you the files which contain the conflicts. You must then navigate to these files, open them, and resolve the conflicts. The conflicting regions of code are delimited with chevrons like this `>>>>>>>>>>>>>>>` and `<<<<<<<<<<<<<<<`.

   ```
   git pull origin master
   ```

5. Push the final version of the code to the global repository on GitHub (the remote repository is named 'origin' by default). The changes you have made will now be available to all, and they will also be almost immediately reflected on the HiFiLES page on GitHub.

   ```
   git push origin master
   ```

Branching in Git

The ease of code branching is a major feature in Git. Branches are parallel versions of the code that allow for decentralized development of particular features or fixes. In this manner, an individual or team can easily switch between developing different features in the code and merge them into the master version when ready (can also help avoid conflicts). In fact, while not mentioned above, the master version of the code is simply a branch like any other. To see the branches in your local repository, type

```
git branch
```

The branch name with an asterisk is the current working branch. One can add branches locally or globally to the remote repository on GitHub that can be shared by all. Assume that a new branch named 'feature_new' has been created in the remote repository, either through the command line or through the GitHub web interface, and you would like to work on this feature. A typical workflow in this scenario might be:

1. Clone a fresh copy of HiFiLES or update your current version with the latest changes on the remote repository.

   ```
   git pull
   ```

   If you would just like to make your local repository aware of changes in the remote (such as the addition of the 'feature_new' branch) but don't want any changes to local files, you could use

   ```
   git fetch
   ```

2. Create a local copy of the branch that is linked to the version on the remote repository,

   ```
   git checkout -b feature_new origin/feature_new
   ```

   This command creates the local branch and switches your local working copy to the 'feature_new' branch. Note that you can not have any local edits or changes when switching between branches, so you should either make a local commit of your changes (as described above) or revert all local changes in the repository with

   ```
   git checkout -- .
   ```

3. Check that you are now in the 'feature_new' branch with

   ```
   git branch
   ```

You should notice that the local copies of your files have seamlessly switched to their state under the 'feature_new' branch.

4. Make some changes to the code and commit your changes to your local copy of the 'feature_new' branch as usual

   ```
   git commit -am "Updates."
   ```

5. Merge local and remote versions of the branch and fix any conflicts if necessary,

   ```
   git pull origin feature_new
   ```

6. Push the final version of the code to the remote branch on GitHub to make it available to all,

   ```
   git push origin feature_new
   ```

It is often the case that you would like to merge your branches back into the master branch after completing work on your new feature or bug fix. When developing features that may take an extended amount of time, it is a good idea to update your branch frequently with the recent changes in the master. This will make it much easier to merge the branches eventually and will help avoid conflicts and headaches when the time comes. Let's assume you are about to work on the 'feature_new' branch again, but would like to update it with the most recent work in the master branch. You could do the following (there are multiple ways to push/pull things between branches):

1. Move back over into your local copy of the master branch

   ```
   git checkout master
   ```

2. Check that you are back in the master branch with

   ```
   git branch
   ```

3. Pull the latest and greatest from the remote master branch on GitHub. Your local copy of the master branch now has all recent changes that can be shared with other local branches.

   ```
   git pull origin master
   ```

4. Switch back over to your local copy of the 'feature_new' branch

   ```
   git checkout feature_new
   ```

5. Merge the local version of the master branch that you just updated into your local version of 'feature_new' and fix any conflicts if necessary.

   ```
   git merge master
   ```

6. Make code changes, merge with the remote 'feature_new' branch, and push to the remote 'feature_new' branch like usual

   ```
   git commit -am "More updates."
   git pull origin feature_new
   git push origin feature_new
   ```

Note that this could also be process could be more direct by pulling from the remote master straight from within your local copy of the 'feature_new' branch. Again, there are multiple ways of updating and merging which involve pushing and pulling between different branches that may be local or remote.

Lastly, once you are finished developing your new feature and have merged your work from the 'feature_new' branch into the master, you can delete the branch from the remote repository with

```
git push origin :feature_new
```

Compiling and running HiFiLES

Needs to be updated...

To specify whether HiFiLES will be compiled for running on CPUs or GPUs, go to `sd_b/bin`, open `Makefile`, look for the line that says `NODE=` and set it to `NODE=CPU` or `NODE=GPU`.

To compile, go to directory `sd_b/bin` and run

```
make -f Makefile
```

To run sd++, go to directory `sd_b/bin` and run

```
./sd++_b INPUTFILE
```

For example

```
./sd++_b ../examples/input_sphere_hex_inv
```

Some nomenclature

`dis`:	discontinuous	`con`:	continuous
`u`:	solution	`f`:	flux
`t`:	transformed	`l,r`:	left, right
`s`:	shape	`norm`:	normal
`inv`:	inviscid	`vis`:	viscous
`grad`:	gradient	`detjac`:	determinant of Jacobian
`pts`:	points	`in`:	input
`loc`:	location	`cub`:	cubature –multidimensional integration
`bdy`:	boundary of domain	`int`:	interior of domain
`inters`:	interfaces	`delta`:	change
`mag`:	magnitude	`dot`:	dot product
`mul`:	multiplication	`jac`:	jacobian
`lut`:	look up table	`temp`:	temporary
`v,vert`:	vertex	`ptr`:	pointer
`ppt`:	plot points		

Main Classes

Class:	**Description**
`array< typename T >`:	stores information in arrays; can be resized and printed
`input`:	reads and stores values from input file
`mesh`:	indirectly stores solution and flux values using `array<eles_type>`; indirectly stores data related to the interfaces using class `inters`; stores information about the mesh; advances the solution in time
`eles`:	stores all solution and flux values directly using `array< double >`; calculates residual in each element (contains the functions that implement the Flux Reconstruction approach)
`inters`:	stores data related to each interface using class `array< double >`; calculates interface fluxes

Contents of folders in sd++

Folder	Contents
`sd_b/bin`	`./sd++_b` executable, Makefile, compiled libraries
`sd_b/docs/html`	Doxygen documentation; open any html file to navigate
`sd_b/examples`	sample input files; mesh files; location of the mesh file is specified in the input file
`sd_b/include`	the `.h` files contain class definitions; the name of each file is the name of the class it defines
`sd_b/lib`	libraries for Tecplot 360 and MPI
`sd_b/src`	`driver.cpp` contains `main()`; the other `.cpp` files contain the implementation of the member functions