Git Handout*

Houtan Bastani†

13 February 2018

Topics addressed in this handout are for the general `git` user. We'll go over what `git` is, setting up your environment, accessing a repository from a server and the usual workflow that you'll follow. I hope to address the most common `git` commands. After going through this handout, you should be able to handle most of your interactions with `git`. At the beginning it'll be annoying and you'll feel like `git` is slowing you down. But, with time, as you become more comfortable with the idea of `git` and the commands listed below, you'll become more accustomed to the way `git` works and be able to learn more on your own.

0.1 How To Approach Learning Git

When you begin learning `git`, things can feel a bit overwhelming. You can feel as though you're swimming in a soup of `git add`s, `commit`s, `fetch`es, `pull`s, and on and on. To avoid this "git command soup" (and the resultant frustration) make sure you understand the big picture before you focus on the details (i.e., `git` commands); try to understand why you're using `git`, the general `git` workflow, and the appropriate action to take in a given situation (which can be project dependent). After you understand what it is you're trying to accomplish and how `git` works, the commands will become much clearer and easier to pick up. And, once you have understood the commands, it will be easier to choose the GUI that best serves your needs. ☺

1 What is Git?

Git is a distributed version control system, developed by Linus Torvalds of Linux kernel fame. But what does that mean?

- **Version Control System**: Something that keeps track of the changes made to files (`commit`s)
- **Distributed**: Everyone has a copy (`clone`) of the git repository on their computer

So, a Distributed Version Control System allows you to have a copy of the complete development history of the project on your local workstation. In other words, all changes ever made in the history of the project are on your computer! This means that at any moment you can go back to the state of the project as it existed at any other point in time (`reset`).

Because of the distributed nature of git, you can develop locally (without need for an internet connection) and, when ready, send (`push`) them to the main git repository (`origin`) to share them with your team

2 When is it useful?

All the time. ☺

Really, git is useful when you are working on projects that depend on text files. So, if you're writing STATA do files, Matlab .m files, adding data to a CSV-based spreadsheet, writing a document in LaTeX, git can help you out.[1]

Organizing Work (aka Development Branches)

Git is particularly helpful when working on research projects. Having worked on a research project before the invention of git, at a certain point file organization became frustrating and rather confusing. My coauthor and myself would each work independently on different aspects of the project and then have to merge our work together. Or, we'd run into a problem, copy everything into a new folder to try an experiment to see

*This handout available at http://www.dynare.org/houtan/handouts

†Dynare Team, CEPREMAP, houtan@dynare.org

[1]Note that you can keep non text-based files in git. The thing is that git keeps track of the textual differences between files to keep the repository size small. If you keep compiled files in git (e.g. `.pdf`s, `.exe`, `.a`, etc.) and these files change often, the size of your git repository will explode. That said, there are additional programs that you can use in conjunction with git for storing these types of files such as Git LFS, but I will not cover that in this file.

if we could resolve the problem, then run into another issue and copy everything into yet another folder to try to resolve that problem. When we finally resolved the main problem, we would never delete the other folders we had used for fear of losing something that we could use later, so we ended up with a confusing layout of badly named folders that lost all meaning over the course of the research project. Git solves these problems and more ☺

In much the same way that git is useful for group-oriented work, it's also useful for working on your own. You can try diffent things on git branches, always having your main work handy. You can make changes, always certain that you can revert to a previous state of a file. Once you are at ease with the git workflow and commands, you'll find yourself using it in many aspects of your work: research, presentations, notes, etc.

Backups

In a comon setup, your local git repository is a clone of a respository that you keep on a server. So, if anything happens to your computer, you will always be able to access your code on the server. What's more, since git is distributed, if anything should happen to the server, you can always recreate an "origin" repository from your local one ☺

Finding a Bug is Easy

Using a git command called `git bisect`, you can easily track down when a bug was introduced into your code base. Imagine you added a certain functionality a year ago. Today, you realize that it no longer works properly. `git bisect` allows you to quickly track down the commit that introduced the bug by jumping to the commit half-way between a year ago and today and asking you if the bug exists. If it does, then it jumps to the point halfway between that point and a year ago and asks you again. It thus quickly zeros in on the buggy commit.

Share Work

You can easily share your work with colleagues and other interested parties. When working, all you have to do is provide the colleague with the address of your remotet repository. He or she will be able to see if you update the code and immediately have access to those updates. What's more, when you publish your paper, you'll have a version of your code that reproduces all the results in your paper. All you need to do is make this public to share it with anyone who wants to see it and reproduce the results themselves.

Working together

You can work on the same codebase with your colleagues without fear of overwriting changes that they made. Git only allows one commit at a time and it forces you to resolve conflicts, so there is no possibility of unintentionally overwriting your co-author's changes, the way you would have if you were sharing files on, say, Dropbox.

Clear Development History

At certain times, it's nice to see how a project has progressed to its current state. By creating logical commits, you do just this. Every node of the git tree represents a version of your codebase. The power of git lies in allowing you to change back to any previous version in your development history.

Work Offline

Since git is distributed, the entire project history is on your computer. So, you can be totally offline, on a plane, on a train, in a car, and, at the drop of a dime, switch to any previous version of the project's development.

And Many more...

3 Setup

Before we tackle the main git workflow, there is a bit of housekeeping you'll need to do.

3.1 Once per computer

When setting up git on your computer for the first time, you need to tell git who you are. This will allow `commit`s you make to be associated with you.

Setup is rather simple. At the command prompt, configure your global git information by issuing the following two commands:

```
> git config --global user.name "Your Name"
> git config --global user.email you@something.something
```

That's it! ☺

3.2 Once per project

When working on a project with others, you'll need to `clone` the global git repository. This command will copy the git repository from the server to your workstation. The repository is completely self-contained and can be copied from one directory to another and from one computer to another. As mentioned above, it contains the entire development history of the project. To get a copy of the global git repository onto your computer, do the following:

```
> git clone <<repository.git>>
```

There are several different protocals that you can use to connect to a git repository (e.g., `http`, `git`, `https`, `ssh`). For example, to clone the Dynare repository, you could do:

```
> git clone https://github.com/DynareTeam/dynare.git
> git clone git@github.com:DynareTeam/dynare.git
```

where `https` is open to everyone and `ssh` is available to those who have uploaded their public key to their `github.com` account.

Of course, if you're working directly on the sevrer where the git repository lies, you can simply do:

```
> git clone /srv/git/repoName.git
```

where `/srv/git/repoName.git` is the full filename of the git repository.

If the repository makes use of `submodules`[2], instead of running

```
> git clone git@github.com:DynareTeam/dynare.git
> cd dynare && git submodule update --init --recursive
```

you can save yourself a step by simply running

```
> git clone --recurse-submodules git@github.com:DynareTeam/dynare.git
```

4 Basic Workflow

Ok, you've done it! Now you have a copy of the git repository on your computer. Now what? Well, if you're a developer, you may want to actually edit files in the repository and then share the changes you've made with the rest of the team. As these actions will comprise the majority of your interactions with git, let's start with them.

4.1 Summary

Here's an overview of the general workflow (and the following sections):

1. Update your repository: `git fetch && git rebase origin/master`
2. Edit file(s)
3. Stage file(s) according to **logical** change: `git add <<files to add>>`
4. Commit file(s) according to **logical** change: `git commit -m ''<<message>>''`
5. Share changes: `git push origin master`

4.2 Updating your repository

Now, before you begin editing a file, it's good practice to update your local copy of the repository. This allows you to always work with the latest version of the code and helps you integrate your changes with those already `push`ed to the main repository. In this way, when you're finally ready to share the changes you've been making, you won't have to worry about `merge` conflicts.

[2]These are just other repositories that your repository depends on, more on this in Section 5.3

4.2.1 Branches

Before we can talk about updating your local repository, we should talk about the concept of `branch`es within git. Remember, your local repository is a copy of the global repository. So, it is useful to know BOTH where the current state of the global repository is and where the current state of your local repository is. When you first clone the repository, you'll have:[3]

- `origin/HEAD`: tells you the branch that's checked out by default when you clone the repository (usually it points to `origin/master`)
- `origin/master`: indicates the main development branch of the repository
- `master`: is your local pointer that tracks `origin/master`. When you make changes, this pointer will advance past `origin/master`. Once your changes have been pushed to the global repository, `origin/master` will come into line with your local master branch

Note here that branch names take the form `<<remote name>>/<<branch name>>`. Here, our `remote`[4] name is `origin`.[5]

The command:

```
> git branch
```

tells you which branch you're currently on. One great benefit of git is that you can have several branches, both locally and on the server. This is useful for software releases (e.g., at Dynare, in addition to the `master` branch, we have a branch for every major release), for working on a side project that will eventually be `merge`ed back into `master`, as well as just for testing ideas out locally.

4.2.2 Updating Procedure

Now that you understand a bit more about branches in git, we're ready to look at the repository updating procedure that you'll use on a regular basis. Basically, before you start working, you should do the following:

```
> git branch
```

to ensure you're on the branch you want to be on (for our purposes, we want to be on `master`). If you're not on `master`, simply run

```
> git checkout master
```

to move back to the `master` branch. Next, to update from the `remote` repository (in our case, it's called `origin`, simply do:

```
> git fetch
> git rebase origin/master
```

So, what do these commands do? Well,

- `git fetch` brings in any changes from the global git repository, updating pointers to the global branches (i.e., `origin/master` and `origin/HEAD`)
- `git rebase origin/master` takes your local master branch pointer and places all local commits (those that have not yet been pushed to the global git repo) above the pointers to the pointers to the global branches (i.e., origin/master and origin/HEAD). In other words it changes the base of the current branch (i.e., master) to be on to of the origin/master branch.

With this procedure, every developer is responsible for ensuring that their commits fit on top of the development history. This creates a clean tree (a straight line) without unneccessary merge commits. It also forces the developer to resolve conflicts him/herself and does not rely on the handwaving that can be implicit in a `merge`. Doing this on a regular basis (before you start working for the day, or even as often as every email that notifies you that the repository has been updated) can save you a lot of pain and help you avoid conflicts.

[3] This is assuming a recently initalized repository. An older project may have other branches.

[4] A `remote` in git is just a repository. By default/convention the first repository you clone from is called `origin`. Implicit in that statement is that you can track multiple repositories. In my Dynare development for example, I track the main Dynare repository (`origin`), `https://github.com/DynareTeam/dynare.git` and I also track a personal remote that I call `houtanb`, `https://github.com/houtanb/dynare.git`. I `push` changes to `houtanb` that I want to test on a server. Once I have tested those changes and am satisfied they are correct, I `push` them to `origin` so that the other Dynare Team members can see them.

[5] Also, note that in git parlance, `origin` is just the name of a `remote` and hence that name can be changed to whatever you want.

NB: NEVER USE `git pull`; it's a high-level command (combining `fetch` and `merge`) that does some fancy stuff. Though it'll usually work, when it doesn't it becomes unneccessarily confusing / problematic / annoying / frustrating to fix it because it takes time and git know-how to find out what the problem is. It also poses a problem when git novices use it by creating unneccessary `merge` commits, making the git `tree` itself confusing, making it more difficult to use git `bisect`, and making `gitk` and other git GUIs more difficult to use as well. Avoid all updating problems by simply following the procedure outlined above.

4.3 Committing

After having made some changes, you are ready to create a commit. A commit is a logical change; it can be as short as one character or as long as you want. The point is that if it were ever to be undone (`reverted`) only that logical change should be undone. If there are more than one type of change within a commit, reverting it would undo things that you may want to keep in the repository. What's more, it helps everyone working on the project better understand exactly what was changed without being obliged to look at the code.

```
> git add <<file to be committed>>
> git commit -m "<<descriptive message>>"
```

- `git add` adds an edited file to the staging area, preparing it to be commited. NB: if you modify the file after running `git add`, you will need to add it again in order for those changes to be included in the commit
- `git commit` creates the logical commit on your current branch (`master` for us), moving it forward by one node, leaving `origin/master` behind. Your local repository is now ahead of the global repository. NB: A commit is denoted by a unique Secure Hash Algorithm (SHA).

After a commit, if you realize that you have forgotten to include a file that logically belongs there, dont worry! Just,

```
> git add <file to be committed>
> git commit --amend
```

to include it in the commit.

4.3.1 Three Types of Files In Git

- **Staged File**: Any file that you have run `git add` on. These files will be committed when you next run `git commit`
- **Unstaged File**: Any tracked file that has been modified but not yet `add`ed to the staging area
- **Untracked File**: Any file in the repository that is not explicitly ignored by a `.gitignore` file and has not previously been `commit`ted and is not currently in the staging area

Running

```
> git status
```

will tell you which files are Staged, Unstaged, and Untracked.

4.3.2 .gitignore

This is a file that tells git which untracked files to ignore. Entries might include `*.o`, `*~`, or `*.aux`, etc.

4.3.3 Related Commands

To put the tracked and edited files in a sandbox (i.e., Work in Progress (WIP) commit):

```
> git stash
```

To create a commit that undoes a previous commit (denoted by the SHA):

```
> git revert <<SHA>>
```

To delete all changes and move your current branch pointer to the commit denoted by SHA:

```
> git reset --hard <<SHA>>
```

NB: this is DANGEROUS! Use it with care not out of frustration!!

4.4 Pushing

Well, well, well. Now you've updated your repository, committed some changes and are ready to share those changes with everyone else. Before you do so, update your repository again! Do this just to make sure that in the period between when you last updated your repository and when you want to push your changes, someone else hasn't introduced changes to the repository. Now, after doing another round of `git fetch` and `git rebase origin/master`, do:

```
> git push
```

and your changes will be sent to the global repository, updating the `origin/HEAD` and `origin/master` pointers. ☺

5 Other Useful Commands....just not used as regularly

5.1 `git rebase -i`

Since the idea of git is to maintain the entire development history of a project, we should be wary of overwriting said history. That said, as when developing locally we may run into situations where we want to reorder our commits before sharing them with others in the project. This may happen when, for example, you have made two bug fix commits and are about to push them to the shared repository when you notice that your first bug fix commit didn't actually fix the bug. You edit the code, commit it, but now have a tree that looks like this:

```
o [master] bug fix 1 update - timestamp today 12:01 (SHA sha3)
o bug fix 2                 - timestamp today 12:00 (SHA sha2)
o bug fix 1                 - timestamp today 11:59 (SHA sha1)
o [origin/master] msg       - timestamp whenever    (SHA sha0)
```

This, however, breaks the defining characteristic of a commit: that it reprepsents a **logical** change to the code because you have two commits that fix bug 1, whereas you should only have one commit that fixes bug 1. Before pushing to the shared repository, we can fix this issue by using an interactive rebase (command: `git rebase -i <<SHA>>` or `git rebase -i origin/master`), which will yield a screen akin to:

```
pick fb65f2d bug fix 1
pick f7f45c6 bug fix 2
pick 853c3f0 bug fix 1 update

# Rebase 81d5998..853c3f0 onto 81d5998
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

To modify the commit history you can then reorder lines, delete lines and/or modify the word `pick` according to the `Commands` section. To accomplish what we originally wanted in this example, we would change those lines to be:

```
pick fb65f2d bug fix 1
f 853c3f0 bug fix 1 update
pick f7f45c6 bug fix 2
```

This would result in the tree:

```
o [master] bug fix 2        - timestamp today 12:00 (SHA sha2)
o bug fix 1                 - timestamp today 11:59 (SHA sha1)
o [origin/master] msg       - timestamp whenever    (SHA sha0)
```

and now we would be ready to share these commits with the other team members. ☺

5.2 `git merge`

`git merge` is a command that combines two or more development histories (branches). You'll need it when deciding to merge some experimental branch back into the main branch. Or, you might need it if you're responsible for merging developers' commits into the main git repository.

To merge two branches, first checkout the branch you want to merge into, then merge in the other branch:

```
> git checkout master
> git merge development
```

Easy as that! Just, not really. If it is, breathe a sigh of releif. If it's not, well, you have some more work to do. Git will let you know there's a problem, displaying a message such as:

```
Auto-merging main.cc
CONFLICT (add/add): Merge conflict in main.cc
Automatic merge failed; fix conflicts and then commit the result.
```

while the file(s) with the conflicts will be marked up like:

```
<<<<<<< HEAD
// My first version of main.cc
=======
// My second version of main.cc
>>>>>>> trash1
```

You'll run into this problem when the same line was modified in both branches. In this case, git doesn't know which modification to keep and hence asks you to figure things out. Git helps you out by marking up the file in the location where things diverged, but that's all it can do. The rest is up to you. You have to open the file, go to the section marked up by git, and modify the lines to contain the good version of the code. You can then add and commit it as:

```
> git add main.cc
> git commit
```

git remembers that you are in the middle of a merge and will set the commit message for you, which, of course, you can edit as you please.

If you find yourself in the middle of a merge that you do not want to complete, simply run either `git merge --abort` or `git reset --hard [<<SHA>>]`.

Now, note that git has a bunch of what they call "merge strategies." The details of these strategies are not useful for the average small or medium-sized project so they won't be addressed here.

5.3 `git submodule`

Sometimes your project will depend on another project's code. You may want to compile their code directly into your executables (if it's C-based project for example) or you may want to distribute their functions alongside your own (if their project is Matlab-based for example). If they use git as well, you can store their project within your own.

More to come here...

6 Group Project with Git: Workflow Setup

6.1 Everyone can `push`

This is the basic setup where all team members can write directly to the repository. This works best for small teams with separate tasks. Basically, everyone clones from the same remote repository, `origin`. They do their work and push their changes to the repository. Management is minimal and overall project stability is not important; people can break parts of the code and fixing it is not urgent.

6.2 Gatekeeper

In this model, there is an assigned gatekeeper, usually a project manager. This person ensures the global coherence of the project. Individual team members have two remotes, `origin` and a personal one that is a fork of the main repository. They make changes as needed and push to their fork of the main repository. They then create a pull request, asking the gatekeeper to merge their changes into the main repository. The gatekeeper is responsible for ensuring that the changes brought into the repository do not interfere with other parts of the project code. This setup is best for medium-size projects with many moving parts where code stability, even in development, is important.

7 Presentation Example

General Workflow

Setup `origin`

1. `mkdir presentation-origin-repo.git`
2. `cd presentation-origin-repo.git`
3. `git init --bare --shared=group`
4. `git config receive.denynonfastforwards false`
5. `cd ..`

Clone

1. Command line
   (a) `git clone presentation-origin-repo.git command_line_user`
   (b) `cd command_line_user`
   (c) `git config user.name "Command Line User"`
   (d) `git config user.email "command@line.user"`
   (e) `git commit -m "Initial Commit" --allow-empty`
   (f) `git push origin master`
2. SmartGit
   (a) Repository → Clone...
   (b) Repository → Settings; change Name and Email

General Workflow (without the `git log` and `git status`)

1. Command line
   (a) `git status`
   (b) `git add function1.m`
   (c) `git status`
   (d) `git commit -m adding "function1.m"`
   (e) `git status`
   (f) `git log`
   (g) `git push origin master`
   (h) `git status`
   (i) `git log`
2. SmartGit user
   (a) Remote → Pull... → Rebase
   (b) Right click on file → Stage
   (c) Local → Commit... → "Adding function2.m"
   (d) Remote → Push...

Conflict on `git pull --rebase`

When you have already made a commit

1. Command line
   (a) `git pull --rebase`
   (b) edit lines 1 & 2 of function1.m
   (c) `git add function1.m`
   (d) `git commit -m "this was a logical commit"`
   (e) `git push origin master`
2. SmartGit user

(a) edit lines 1 & 13 of function1.m
(b) Right click on file → Stage
(c) Local → Commit... → "this was a logical commit too"
(d) Remote → Push...
(e) Remote → Fetch
(f) Right click on `origin/master` → Rebase HEAD (master) to...

When you have staged files

1. Command line
   (a) `git reset --hard HEAD~1`
   (b) edit lines 1 & 2 of function1.m
   (c) `git add function1.m`
   (d) `git commit -m "this was a logical commit"`
   (e) `git push origin master`
2. SmartGit user
   (a) edit lines 1 & 13 of function1.m
   (b) Right click on file → Stage
   (c) Remote → Pull... → Rebase
   (d) Right click on Stash → Apply Stash...

Branch

1. Command line
   (a) `git fetch`
   (b) `git rebase origin/master`
   (c) edit lines 1 & 2 of function1.m
   (d) `git add function1.m`
   (e) `git commit -m "this was a logical commit on the master branch"`
   (f) `git push origin master`
2. SmartGit user
   (a) Branch → Add Branch... → development → Add Branch & Checkout
   (b) edit lines 1 & 2 of function1.m
   (c) Right click on file → Stage
   (d) Local → Commit... → "a logical change on my development branch"
   (e) Remote → Pull... → Rebase
   (f) edit lines 4 & 5 of function1.m
   (g) Right click on file → Stage
   (h) Local → Commit... → "one more logical change before merge"
   (i) Right click on `origin/master` → Check out... → master
   (j) Click on `development` → Branch → Merge...

Interactive Rebase

1. SmartGit user
   (a) edit line 1 of function1.m (type "bug fox 1" somewhere)
   (b) Right click on file → Stage
   (c) Local → Commit... → "bug fix 1"
   (d) edit lines 4 of function1.m (type "bug fix 2" somewhere)
   (e) Right click on file → Stage
   (f) Local → Commit... → "bug fix 2"

(g) edit lines 1 of function1.m (change to “bug fix 1”)

(h) Right click on file → Stage

(i) Local → Commit... → “bug fix 1 update”

(j) Drag 3rd update on top of 1st update → Rebase 1 commit from master to ¡¡SHA¿¿

(k) Highlight the two commits, right click → Squash Commits...

(l) Click on “bug fix 2” commit and drag on top of this new commit → Cherry-pick selected commits to master

(m) Remote → Push...