A guide to using GIT for CASINO

v1.1 – November 19, 2008 – Pablo López Ríos[1]

1 Overview

GIT is the revision control system we have chosen for CASINO. This guide provides an introduction to using GIT for common tasks.

1.1 Notation

Notation rules:

- Commands and filenames will be written in a `fixed-width font`.
- When commands and their output are shown in the same block of text, a dollar sign `$` is shown before command lines.
- Optional arguments will be enclosed in `[square brackets]`.
- Argument descriptions, which are meant to be substituted with whatever the description indicates, will be hyphenated together for clarity and enclosed in `<angle-brackets>`[2].

1.2 GIT basics

GIT commands are of the form

```
git [<global-options>] <command> [<command-options>]
```

Most of the commands have to be executed while sitting in a directory inside the distribution tree –any directory– in order to work; the commands in this guide assume this, unless otherwise stated/implied.

There are plenty of different GIT commands, 128 by my count. They are divided into "porcelain" (high level), "ancillary" (extras) and "plumbing" (low level). Most likely you will never want to use any of the "plumbing" commands, which reduces the list to 72 commands, of which you will use about 5–10 regularly. I strongly recommend defining aliases or functions in your `~/.bashrc` or `~/.cshrc` to quickly access the commands you use the most.

To see a list of GIT commands, type

```
man git
```

To get help on one of the commands, type

```
git help <command>
```

This will display the manpage of the command you have chosen. The same syntax also provides access to other GIT-related manpages, in particular you are encouraged to have a look at

```
git help gitglossary
git help gittutorial
git help gittutorial-2
```

Finally, you can find tons of information online. Do try the manpages first, they usually contain what you are looking for.

We will assume GIT version 1.5.0 (Feb 2007)[3] or newer in this guide.

[1] Email: `pl275@cam.ac.uk` – please email any suggestions regarding this document.

[2] Hyphenation doesn't imply that the argument is required to be a single word, though.

[3] Available on the repositories of modern LINUX distributions. "Enterprise" distributions (e.g., those provided by SUSE and RED HAT) are an exception, since they usually get outdated very quickly. If this is your case, you can compile GIT from the sources; visit `http://www.git.or.cz` to get them.

2 Getting started

2.1 CASINO development and the official GIT repository

CASINO has an *official* version which resides in a GIT repository in TCM. This repository can be read by developers and collaborators, but it is only modifiable by the CASINO maintainers (currently this includes Mike Towler[4] only).

Each developer will have their own GIT repository to work on. Any changes that you wish to contribute to the official version must be sent to one of the maintainers[5] who will review the changes and incorporate them when they are ready.

Developers are also welcome to interchange patches among themselves as they see fit, and there is GIT functionality to help do this. Bear in mind that source code can only be interchanged between registered developers, as per the developer license agreement.

2.2 Initial setup

2.2.1 Clean up any previous installation

If you already have a `~/CASINO` directory, remove it by deleting it (`rm -rf ~/CASINO`) or, if you want to keep it for any reason, by moving it elsewhere (e.g. `mv ~/CASINO ~/CASINO_save`).

2.2.2 Install CASINO

Now clone the repository into your home directory. If you are in TCM, type

```
cd
git clone ~casino/git/CASINO
```

If you are on a computer outside TCM (e.g., Darwin) but you have a TCM account, type

```
cd
git clone ssh://<user>@pc52.tcm.phy.cam.ac.uk/u/tcmsf1/casino/git/CASINO
```

or, if you want to synchronize you external repository with your TCM repository rather than with the main one (which is a good idea), type

```
cd
git clone ssh://<user>@pc52.tcm.phy.cam.ac.uk:/u/tcmsf1/<user>/CASINO
```

If you are an external collaborator, type[6]

```
cd
git clone <address-to-be-published-online>
```

This will set up your repository and working tree (see Section 2.3), and will also configure a *remote-tracking repository* for the official repository under the name `origin`. In the rest of this guide this is assumed to be configured; use `git remote` to achieve the same effect if you decide to set up the repository otherwise (e.g. with `git init` and `git pull`).

Tip for TCM users: To make your directory readable by other people in the `casino` Unix group, for collaborative purposes, type

```
cd
chgrp -R casino CASINO
chmod o-rx,g+rxs `find CASINO -type d`
```

[4] Email: `mdt26@cam.ac.uk`

[5] Use `git format-patch` as explained later in this guide, and attach the patch to your email.

[6] Check `http://tcm.phy.cam.ac.uk/~mdt26/casino` for the correct address, which is unknown as of this writing.

Tip for TCM users: This is a good moment to ensure that your binaries will not overflow the tight disk quotas we have in TCM. Type

```
mkdir -p /rscratch/$USER/tempdirs/bin_qmc
mkdir -p /rscratch/$USER/tempdirs/src_zlib
mkdir -p /rscratch/$USER/tempdirs/utils_zlib
ln -s /rscratch/$USER/tempdirs/bin_qmc ~/CASINO/bin_qmc
ln -s /rscratch/$USER/tempdirs/src_zlib ~/CASINO/src/zlib
ln -s /rscratch/$USER/tempdirs/utils_zlib ~/CASINO/utils/zlib
```

2.2.3 Configure your details

To set up your details, type

```
git config user.name "<your-full-name>"
git config user.email "<your-email-address>"
```

2.3 Concepts

We are ready to start using GIT, but we should introduce a few concepts beforehand.

2.3.1 Repository and working tree

If you look at the output of `ls -lA` under `~/CASINO`, you will notice the presence of a `.git` subdirectory. This subdirectory contains the GIT *repository*, while the rest of `~/CASINO` is called the *working tree*.

The working tree represents a state of the distribution (past or current), including any changes you may have made to it.

The GIT repository contains your very own copy of the CASINO history. Moreover, it may contain any number of *branches* (i.e., modified versions), along with their own history. Each branch has a "current state", which is called the *head* of the branch. Only one branch is allowed to be active at any given time. Therefore the "current state" of the repository is defined by the current branch and its head.

You can make any change to the working tree and store it in the GIT repository by *committing* your change. Conversely, you can restore any previous state of the working tree from the repository by *resetting* your working tree to the state of your choice, or by *checking out* a different branch from the repository.

2.3.2 Flexibility, robustness and unreferenced objects

The necessity for allowing unreferenced objects in a repository arise from the combination of two factors,

- GIT is flexible: it offers the ability to modify history by removing, reordering and inserting commits anywhere in a branch.
- GIT is robust: it is designed so that you will *never* lose anything once it is commited (unless you really want to).

Clearly, the ability to remove commits is somewhat incompatible with the principle that no commits be lost. What really happens when you delete a commit is that the history stops pointing at it, but the commit still exists in the database. Such a commit is a unreferenced object.

You can recover data from unreferenced objects at any time, and may re-incorporate them into a branch if necessary. Unreferenced objects have an expiry date (30 days from their creation by default), after which they *may* be removed (if you actively tell GIT to do so) to reclaim the disk space.

While this may sound a bit obscure to start this guide with, it is important to keep these notions in mind, especially when you do something wrong –your work is probably recoverable.

2.4 A look at the repository

To see what commits are recorded in your repository, type

```
git log
```

Note that by default the output is paged using `less`; press `space` to advance a page and `q` to quit.[7]

The output of `git log` looks like this:

```
$ git log --pretty=oneline
7710ca9bfb1b7aab1bcc25d8756df60bb48a7b73 [NDD] Bug fix for 2D MPC.
865af2e9ce13dc7e822e1f0d5d7f72cbbb05555d (NDD) Minor no-importance edit to eval
44d1b3de248cb58126edcf3a477bc36509ed8886 [NDD] Minor mods to blip treatment of
88d36c1a75d9ba7adfaec0ed4f5bd84760bcbbe7 Forgot to git commit DIARY changes. Don
86fd4e335d64a4cb918a9eafed6f21d2e7b6c31a Fixes for the MPC interaction in slab g
82432ef8e6e065daed47e580170b84f38160a234 Increase tolerance for lattice generato
9ba0e16efa8557a2fdd9e5b9c6b88f4b40290213 Automatically flip time reversed k-poin
a4e9db9e61c34600ae6462dd58dfac61461cffba Slightly clarify output of varmin.
a1474946ab72ef55c5144e3a856c261b61812bac Interpret wdmcmin and wdmcmax as relati
4231af173050cb97fb4a18f08321c1d1230674db Add timing information for the numerica
```

The commits are in reverse order: the one at the top represents the current state of the branch (the head).

The forty-figure hexadecimal number displayed on each line is a SHA1 hash. Each commit is identified by its SHA1 hash. The advantage of this is that SHA1 is a cryptographically secure hash, which implies that commits can be safely identified uniquely by their hash, which makes comparing commits very easy. This not only applies to commits, but also to every other object in the GIT database (trees, files, diffs).

While extremely useful, SHA1 keys only get in the way of conciseness when we are just trying to see the log. Now let's try the following:

```
$ git log --pretty=tformat:"%h [%an] %s"
7710ca9 [Mike Towler] [NDD] Bug fix for 2D MPC.
865af2e [Mike Towler] (NDD) Minor no-importance edit to eval_geometry.f90
44d1b3d [Mike Towler] [NDD] Minor mods to blip treatment of MPC interaction (i.e.
88d36c1 [Mike Towler] Forgot to git commit DIARY changes. Done.
86fd4e3 [Norbert Nemec] Fixes for the MPC interaction in slab geometry.
82432ef [Norbert Nemec] Increase tolerance for lattice generator to 1e-6. The prob
9ba0e16 [Norbert Nemec] Automatically flip time reversed k-points in blip wave fun
a4e9db9 [Norbert Nemec] Slightly clarify output of varmin.
a147494 [Norbert Nemec] Interpret wdmcmin and wdmcmax as relative towards the curr
4231af1 [Norbert Nemec] Add timing information for the numerical derivatives to va
```

This looks cleaner. The SHA1 hash has been truncated to 7 characters, which is OK. Actually, not much information is lost: you may refer to a commit by any portion of its SHA1 hash as long as there is no other hash in the database which contains the same sub-string.

Tip: Try the following command,

```
git --no-pager log -10 --pretty=tformat:"%Cred%h%Cblue%an%n%Cgreen%s%n%Creset%b"
```

See `git help log` for more possible formats. Pick your favourite and create an alias for it in your shell's `rc` file as a shortcut.

So for example we may ask for more information about the second-to-last commit:

[7] You can prevent a pager from being used by typing `git --no-pager log`, or permanently by typing `git config --global core.pager cat`.

```
$ git show 865af2e
commit 865af2e9ce13dc7e822e1f0d5d7f72cbbb05555d
Author:  Mike Towler <mdt26@cam.ac.uk>
Date:    Fri Nov 14 13:07:37 2008 +0000

     (NDD) Minor no-importance edit to eval_geometry.f90

diff --git a/examples/atom/neon_n/awfn.data.gz b/examples/atom/neon_n/awfn.data.gz
new file mode 100644
index 0000000..c4bd5cd
Binary files /dev/null and b/examples/atom/neon_n/awfn.data.gz differ
diff --git a/src/eval_geometry.f90 b/src/eval_geometry.f90
index 623e2b2..e6a89d6 100644 --- a/src/eval_geometry.f90
+++ b/src/eval_geometry.f90
@@ -129,9 +129,9 @@ CONTAINS

 !  Inverses of matrices of lattice vectors.
  painv=transpose(pbmat)*one_over_twopi
- pbinv=transpose(pamat)*twopi
+ pbinv=transpose(pamat)*one_over_twopi
  ainv=transpose(bmat)*one_over_twopi
- binv=transpose(amat)*twopi
+ binv=transpose(amat)*one_over_twopi

 !  Check orientations in 1D and 2D systems
```

A rather interesting view of the history may be produced if you run `gitk`, which is a graphical utility[8], see Fig. 1. There is another graphical utility in the GIT suite, `git gui`. Both are out of the scope of this guide.

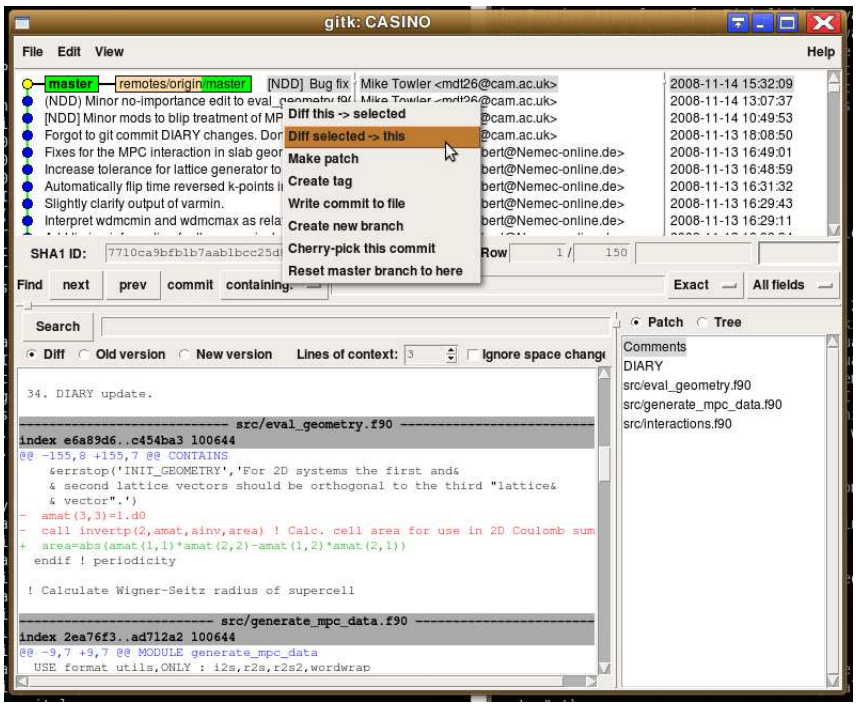


Figure 1: `gitk`, a graphical utility distributed with GIT.

[8]Best seen (with antialiased text) if you have TCL/TK version 8.5 or later.

3 A hands-on tour of GIT

Let's play a bit with the repository to simulate real-life situations. Issue the following command

```
$ git reset --hard HEAD~2
HEAD is now at 44d1b3d [NDD] Minor mods to blip treatment of MPC
interaction (i.e. patch 2.2.209).
```

This has reverted your working directory and your `HEAD` to two commits before its former position[9], which you can check using `git log` as explained above. We are pretending we are in a situation where the main repository has been updated but we still don't know.

3.1 Update without local changes

Since you haven't done any changes to the distribution, there will be nothing to worry about. To see if anything has changed in the main repository, type

```
git fetch
git log master..origin
```

The first command retrieves any new remote information[10,11], and the second gives a log of what has happened between the tip of our `master` branch and the now-updated `origin` remote-tracking branch.[12] The log should obviously contain just the two commits we have removed.

So now we decide to update our distribution. Type

```
$ git pull
Updating 44d1b3d..7710ca9
Fast forward
 DIARY                              |   12 ++++++++++++
 examples/atom/neon_n/awfn.data.gz |  Bin 0 -> 9699 bytes
 src/eval_geometry.f90              |    7 +++----
 src/generate_mpc_data.f90          |   30 ++++++++++++++++++++------------
 src/interactions.f90               |    2 +-
 5 files changed, 34 insertions(+), 17 deletions(-)
 create mode 100644 examples/atom/neon_n/awfn.data.gz
```

So now we're back where we were; check the log to make sure. Fair enough, but let's go back and try a different method. Type:

```
$ git reset --hard HEAD~2
$ git rebase origin
First, rewinding head to replay your work on top of it...
Fast-forwarded master to origin.
```

The effect is the same as that of `git pull`, as you can see in the log. We will see a slight difference in the next part.

Before continuing, reset the `HEAD` two commits back again.

3.2 Making changes

Now let's make some trivial change. For example, type

```
cd ~/CASINO
echo "All the above is a lie." >> README
```

[9] See "Specifying Revisions" in `git help rev-parse` to learn more about specifying commits.

[10] In this case we already have the remote information in the local repository, because our remote-tracking branch (`origin`) is fully up-to-date. The `git reset` command has had no effect on it.

[11] Notice that from remote locations you seem to need to type your SSH password twice. I don't know why this is.

[12] See `git help rev-list` to learn more about specifying commit ranges.

Now type

```
$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   README
#
no changes added to commit (use "git add" and/or "git commit -a")
```

This command tells you the difference between the current `HEAD` and the working tree, and is very useful before committing changes. To select the `README` file for the commit, type

```
git add README
```

You would do this for all the files you want to add to the commit. If you make up your mind you may type `git reset` to "un-add" all the files and start again (with no change to your working tree, of course). If you want to discard your modifications altogether, type `git reset --hard`.

Once you've `git add`-ed the file, type

```
git commit
```

You will be prompted for a commit message in a text editor[13]. The editor is pre-filled with commented-out information about what files you are commiting; this will not be part of the message. The structure of the commit message is:

- **subject**: single line summarizing the changes very briefly
- **blank line** to signal the end of the subject
- **body**: full description of the changes, taking as many lines (and paragraphs) as you need

For example:

```
Lie flagged in README

Having found that the entire contents of the CASINO/README file are false,
I have flagged them as such. Hopefully users will read the last line in the
file before reading the rest.
# Please enter the commit message for your changes.
# (Comment lines starting with '#' will not be included)
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   README
#
```

If you choose to leave the commit message empty (e.g., close without saving), the commit will be aborted.

Once you have done the commit, you will see it in the log:

```
$ git log -5 --pretty=tformat:"%h [%an] %s"
72e8924 [Pablo Lopez Rios] Lie flagged in README
44d1b3d [Mike Towler] [NDD] Minor mods to blip treatment of MPC interaction (i.
88d36c1 [Mike Towler] Forgot to git commit DIARY changes. Done.
86fd4e3 [Norbert Nemec] Fixes for the MPC interaction in slab geometry.
82432ef [Norbert Nemec] Increase tolerance for lattice generator to 1e-6. The pr
```

[13]The editor is determined by your `EDITOR` environment variable, which you can set in your `~/.bashrc` file.

Just for practice, let's modify further. Type

```
cd ~/CASINO
echo "And I don't like lies" >> README
```

You can check that `git status` tells you the file has been modified again. Instead of producing a new commit, this time we shall modify the previous commit to include this new change. Type:

```
git add .
git commit --amend
```

Note the different syntax used, `git add .`, which adds any changes to the current directory and its subdirectories –used here just to show how it works.

You may leave the commit message as it is, or modify it. Notice that if you now type

```
$ git log -5 --pretty=tformat:"%h [%an] %s"
d666d3e [Pablo Lopez Rios] Lie flagged in README
44d1b3d [Mike Towler] [NDD] Minor mods to blip treatment of MPC interaction (i.
88d36c1 [Mike Towler] Forgot to git commit DIARY changes. Done.
86fd4e3 [Norbert Nemec] Fixes for the MPC interaction in slab geometry.
82432ef [Norbert Nemec] Increase tolerance for lattice generator to 1e-6. The pr
```

the SHA1 key of your commit has changed. As it should. Probably you will also notice that your SHA1 keys differ from mine; the name of a commit's author affects the hash.

3.3 Update with non-conflicting changes

The change we have introduced will not present a problem for an update, since no-one else has modified the `README` file. Let's type

```
$ git pull
Merge made by recursive.
 DIARY                              |   12 ++++++++++++
 examples/atom/neon_n/awfn.data.gz |  Bin 0 -> 9699 bytes
 src/eval_geometry.f90              |    7 +++----
 src/generate_mpc_data.f90          |   30 ++++++++++++++++++++------------
 src/interactions.f90               |    2 +-
 5 files changed, 34 insertions(+), 17 deletions(-)
 create mode 100644 examples/atom/neon_n/awfn.data.gz
```

If we now check the log, we will see:

```
$ git log -5 --pretty=tformat:"%h [%an] %s"
4eb03c4 [Pablo Lopez Rios] Merge branch 'master' of /home/pablo/git-play/fakerep
d666d3e [Pablo Lopez Rios] Lie flagged in README
7710ca9 [Mike Towler] [NDD] Bug fix for 2D MPC.
865af2e [Mike Towler] (NDD) Minor no-importance edit to eval_geometry.f90
44d1b3d [Mike Towler] [NDD] Minor mods to blip treatment of MPC interaction (i.
```

The third and fourth lines are the commits we had deleted, and the second is our change to the `README` file. So far so good, this is what we intended. However the last commit is unexpected. Running `gitk` (see Fig. 2) you can see what has happened: you have merged the two diverging sets of changes back into one branch.

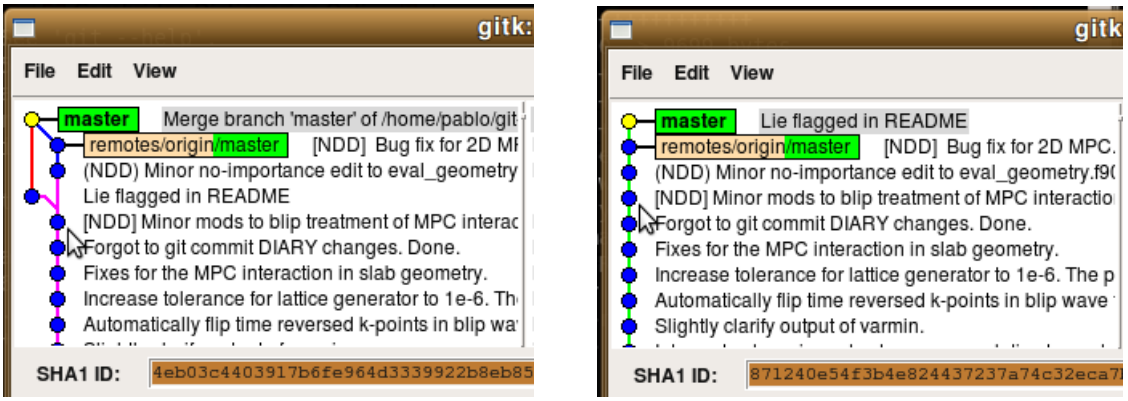
This is a good reason for using the `git rebase` method of updating your tree. Type

```
$ git reset --hard HEAD@{1}
```

which puts the `HEAD` back where it was before it was last moved (we will see more about this in section **??**. Then type

```
$ git rebase origin
```

As you can see from `git log` and from Fig. 2, this is probably more like what you would have expected: your change has been re-applied on top of the remote `HEAD`.



(a) `git pull` (b) `git rebase`

Figure 2: `gitk` representation of the state of your `master` branch after a `git pull` (left) and a `git rebase` (right), both of which can be used to update your sources.

I recommend using `git rebase` to update your branches, to avoid unnecessarily complicating your history.

3.4 Retracing your steps

Later on I want to illustrate an update with conflicting changes, so now we want to go back to the state we were at before attempting to update our distribution. You can check the history of the position of the `HEAD` with

```
$ git reflog
871240e... HEAD@{0}: rebase: Lie flagged in README
7710ca9... HEAD@{1}: checkout: moving from master to 7710ca9bfb1b7aab1bcc25d8756
d666d3e... HEAD@{2}: HEAD@{1}: updating HEAD
4eb03c4... HEAD@{3}: pull : Merge made by recursive.
d666d3e... HEAD@{4}: commit (amend): Lie flagged in README
72e8924... HEAD@{5}: commit: Lie flagged in README
44d1b3d... HEAD@{6}: HEAD~2: updating HEAD
```

This shows a list of past states of the repository in reverse chronological order. Note that this is not the same as a normal log of commits, since the commits may be reordered at will. In our case, we want to go back to our ammended commit, marked as `HEAD@{4}` in the `reflog`, so we type

```
git reset --hard HEAD@{4}
```

This type of action is useful in panic situations. If you are unhappy about a merge you can go back to where you were and deal with it at a later time.

In order to force a clash between our changes and those we are going to incorporate from the remote repository later on, let's have a look at

```
git show origin/HEAD~1
```

This will display the same information we got in section 2.4 for the then-second-to-last commit.

To force a clash, simply edit `src/eval_geometry.f90`, locate line 132,

```
pbinv=transpose(pamat)*twopi
```

and make it look like

```
pbinv=transpose(pamat)*2.d0*pi
```

After the edit, you can type `git diff` to see what you have just done. Now commit the change,

```
git commit -a -m 'Twopi -> 2*pi'
```

Here we use `git commit -a` to show its existance; it's equivalent to `git add .` followed by `git commit`. The `-m` option can be used to give the commit message without going into the editor –not very useful in general, you probably want to be a little more verbose about your changes than what reasonably fits in a single command line.

In `git log`, you should now see

```
$ git log -5 --pretty=tformat:"%h [%an] %s"
9e2e936 [Pablo Lopez Rios] Twopi -> 2*pi
d666d3e [Pablo Lopez Rios] Lie flagged in README
44d1b3d [Mike Towler] [NDD] Minor mods to blip treatment of MPC interaction (i.
88d36c1 [Mike Towler] Forgot to git commit DIARY changes. Done.
86fd4e3 [Norbert Nemec] Fixes for the MPC interaction in slab geometry.
```

3.5 Reordering commits

Just for practice, let's suppose we want to swap our two commits around, for example because we want to work a bit more on the previous one, and `git commit --amend` only works on the last commit. To do this, type

```
git rebase -i HEAD~5
```

This allows you to change the order of the last 5 commits (`-i` is for "interactive"). You will be presented with an editor containing

```
pick 86fd4e3 Fixes for the MPC interaction in slab geometry.
pick 88d36c1 Forgot to git commit DIARY changes. Done.
pick 44d1b3d [NDD] Minor mods to blip treatment of MPC interaction (i.
pick d666d3e Lie flagged in README
pick 9e2e936 Twopi -> 2*pi

# Rebase 82432ef..9e2e936 onto 82432ef
#
# Commands:
# p, pick = use commit
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

which is a list of the 5 last commits with the instruction `pick` in front –notice that the order of the commits is reversed with respect to that used in `git log`. The commented-out instructions are very clear: swapping lines swaps commits, and deleting lines deletes commits. What we want to do

here is swap the last two lines around and exit[14]. Once you're done, the log shows

```
$ git log -5 --pretty=tformat:"%h [%an] %s"
98fc1ba [Pablo Lopez Rios] Lie flagged in README
7785d07 [Pablo Lopez Rios] Twopi -> 2*pi
44d1b3d [Mike Towler] [NDD] Minor mods to blip treatment of MPC interaction (i.
88d36c1 [Mike Towler] Forgot to git commit DIARY changes. Done.
86fd4e3 [Norbert Nemec] Fixes for the MPC interaction in slab geometry.
```

Notice the change in the SHA1 keys for the two commits. A commit's hash depends on which commit it is based on.

Now we would be ready to do further work on the first set of changes. But we've already seen how to do this, so let's just try to update our distribution with a clashing change.

3.6 Update with conflicting changes

As usual, we would type

```
$ git rebase origin
First, rewinding head to replay your work on top of it...
Applying Twopi -> 2*pi
error: patch failed: src/eval_geometry.f90:129
error: src/eval_geometry.f90: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merged src/eval_geometry.f90
CONFLICT (content): Merge conflict in src/eval_geometry.f90
Failed to merge in the changes.
Patch failed at 0001.

When you have resolved this problem run "git rebase --continue".
If you would prefer to skip this patch, instead run "git rebase --skip".
To restore the original branch and stop rebasing run "git rebase --abort".
```

So we are told that `src/eval_geometry.f90` has problems. `git status` and `git diff` will give you more information. If you edit `src/eval_geometry.f90` you will see the clash marked with diff-like symbols,

```
!  Inverses of matrices of lattice vectors.
 painv=transpose(pbmat)*one_over_twopi
<<<<<<< HEAD:src/eval_geometry.f90
 pbinv=transpose(pamat)*one_over_twopi
=======
 pbinv=transpose(pamat)*2.d0*pi
>>>>>>> Twopi -> 2*pi:src/eval_geometry.f90
 ainv=transpose(bmat)*one_over_twopi
 binv=transpose(amat)*one_over_twopi
```

You are expected to remove the markings and produce a merged result, which in this case is

```
!  Inverses of matrices of lattice vectors.
 painv=transpose(pbmat)*one_over_twopi
 pbinv=transpose(pamat)*one_over_twopi
 ainv=transpose(bmat)*one_over_twopi
 binv=transpose(amat)*one_over_twopi
```

i.e., we've removed our modification. Save the file, exit the editor and type

[14] In VIM you can cut a line by typing `dd` and paste it pressing `P`.

```
$ git add src/eval_geometry.f90
$ git rebase --continue
Applying Twopi -> 2*pi
No changes - did you forget to use 'git add'?

When you have resolved this problem run "git rebase --continue".
If you would prefer to skip this patch, instead run "git rebase --skip".
To restore the original branch and stop rebasing run "git rebase --abort".
```

This now tells us that the merged file is identical to the original, and no-action patches are not allowed, so we need to type

```
$ git rebase --skip
HEAD is now at 7710ca9 [NDD] Bug fix for 2D MPC.
Applying Lie flagged in README
```

This skips the conflicting patch and applies the other one. You can check in the log that the clashing patch is now gone –this is because it became a no-action patch, else it would have remained.

3.7 Sharing patches

Let's now export our remaining patch, presumably to send it to somebody else. Type

```
$ git format-patch -k -o $HOME HEAD~1
/home/pablo/0001-Lie-flagged-in-README.patch
```

This has produced a file under your `$HOME` containing the patches applied since `HEAD~1`, this is, your patch. If you examine the file you will see

```
$ cat $HOME/0001-Lie-flagged-in-README.patch
From 0a38d10f49fd6bbcd8c9c97bbe4498293aadce3d Mon Sep 17 00:00:00 2001
From: Pablo Lopez Rios <pl275@cam.ac.uk>
Date: Tue, 18 Nov 2008 06:10:18 +0000
Subject: Lie flagged in README

Having found that the entire contents of the CASINO/README file
are false, I have flagged them as such.  Hopefully people will
read the last line in the file before reading the rest.
---
 README |    2 ++
 1 files changed, 2 insertions(+), 0 deletions(-)

diff --git a/README b/README
index 9a19d0b..9e248d1 100644
--- a/README
+++ b/README
@@ -116,3 +116,5 @@ manual   : the CASINO manual (also available on the web at
A bin_qmc directory will automatically appear on compilation of CASINO or the
utility programs. This directory will contain separate binaries in
subdirectories corresponding to different machines and compilation level.
+All the above is a lie.
+And I don't like lies
--
1.5.6.3
```

Note that the patch is formatted like an email message. It can actually be sent as such, although it's OK if when you send it to the repository maintainer you attach the `.patch` file to the email.

Let's now apply this patch. Start by removing the commit from your repository,

```
git reset --hard HEAD~1
```

and then do

```
git am -k $HOME/0001-Lie-flagged-in-README.patch
```

You can check in the log that the patch is back there. Notice that its SHA1 hash has changed: this is because the commit date is taken into account in the SHA1 as well. This is not a problem in any practical situation, so you can safely ignore this.

3.8 Congratulations!

You've completed the hands-on tour of GIT. You should be ready to use GIT for common tasks. There is more to learn, though. For example, we haven't covered branches, which allow you to maintain different development versions simultaneously. But you have the manpages to look that up, and you should have the basic notions to learn to use other features quickly. You may also find some additional information in the next section, which attempts to provide a reference list of commands for a variety of situations.

4 Cheatsheet

This section contains a list of useful actions and associated commands in their simplest form –see the manpages if you need more options. This list is probably somewhat incomplete –email Pablo with suggestions.

4.1 Initialization

- Cloning an existing repository:

```
git clone <repository-location> [<new-directory>]
```

- Creating a new repository with the contents of the current directory:

```
git init
git add .
git commit -m "Initial commit."
```

4.2 Querying the history

- Displaying the last `<n>` (default is all) commits in branch `branch` (default is the current branch):

```
git log [-<n>] [<branch>]
```

- Displaying full details about a commit:

```
git show <commit-id>
```

- Displaying the changes to a file (default is all files) that happened between two commits:

```
git diff <first-commit>..<second-commit> [<file>]
```

- Displaying the changes to a file (default is all files) that happened between the last commit and the working tree:

```
git diff [<file>]
```

4.3 Making changes

- Displaying the changes to the working directory eligible for a commit:

```
git status
```

- Selecting files for a commit:

```
git add <files>
```

- "Un-adding" all selected files:

```
git reset
```

- Removing a file and selecting the removal for a commit:

```
git rm <file>
```

- Selecting a file removal for a commit (after having removed it by hand):

```
git rm --cached <file>
```

- Moving a file and selecting the move for a commit:

```
git mv <file-from> <file-to>
```

- Doing a new commit with selected changes:

```
git commit
```

  then edit the commit message (subject line, blank line and body).

- Automatically selecting modified files and commiting them (this will **not** take care of additions or removals):

```
git commit -a
```

- Adding selected changes to the last commit, and/or modifying the commit message, and/or reassigning the patch to someone else:

```
git commit --amend [--author="<full-name> <<email-address>>"]
```

- Undoing the last commit without altering the working tree:

```
git reset HEAD^
```

- Undoing the last commit in both the repository and the working tree:

```
git reset --hard HEAD^
```

4.4 Sharing changes

- Exporting the last `<n>` commits in the current branch to `<directory>`:

```
git format-patch HEAD~<n> -o <directory>
```

- Importing the patch in `<file>` on top of the current branch as a new commit:

```
git am <file>
```

4.5 Solving conflicts

- Solving conflicts on `<files>` and continuing a failed rebase:

```
<edit-<files>-looking-for-"<<<<<<",-"======",-and-">>>>>>">
git add <files>
git rebase --continue
```

- Discarding conflicting patch and continuing a failed rebase:

```
git rebase --skip
```

4.6 Modifying history

- Interchanging the order of (some of) the last `<n>` commits on the current branch:

```
git rebase -i HEAD~<n>
<edit-file:-swapping-lines-swaps-commits,-deleting-lines-deletes-commits>
```

4.7 Branches

- Displaying branches:

  ```
  git branch
  ```

- Creating a branch from arbitrary commit (default is current HEAD):

  ```
  git branch <branch> [<commit-id>]
  ```

- Creating a branch from arbitrary commit (default is current HEAD) and switching to it:

  ```
  git checkout -b <branch> [<commit-id>]
  ```

- Switching to different branch:

  ```
  git checkout <branch>
  ```

- Renaming a branch (default is current branch):

  ```
  git branch -m [<old-branch-name>] <new-branch-name>
  ```

- Deleting a branch:

  ```
  git branch -d <branch-to-delete>
  ```

- Updating current branch from `<other-branch>` (default is branch where current was originally created from):

  ```
  git rebase [<other-branch>]
  ```

4.8 Working with unreferenced objects

- Displaying unreferenced objects in a repository

  ```
  git fsck --unreachable --no-reflogs
  ```

- Pointing `HEAD` at place other than tip of a branch:

  ```
  git checkout <commit-id-to-switch-to>
  ```

  Note: when you move the `HEAD` to anywhere other than the tip of a branch, including unreachable commits, the repository is regarded not to be on any branch. This state behaves exactly like a branch, except its unnamed and no other object points at it. I foresee this may be useful for quickly retreiving an old version of CASINO (and committing a few changes if needed) in order to compile it with

  ```
  make debug EXECUTABLE=casino_old
  ```

  and switch back to the development branch to debug some inconsistency between the two versions.

- Going back to a "normal" state:

  ```
  git chechout <proper-branch-which-actually-exists>
  ```

- Turning current unreachable state into a proper branch:

  ```
  git checkout -b <new-branch-name>
  ```

- Displaying the history of positions pointed at by `HEAD`:

  ```
  git reflog
  ```

  Note: this will usually show whatever actions have lead you to lose a commit.

- Forcing expiration of unreachable objects:

  ```
  git reflog expire --expire-unreachable=0 --all
  ```

- Doing away with expired unreachable objects:

  ```
  git prune
  ```

4.9 Extracting specific file versions

- Extracting arbitrary version (default is current `HEAD`) of arbitrary file to working tree

  ```
  git checkout [<commit-id>:]<file>
  ```

- Extracting arbitrary version (default is current `HEAD`) of arbitrary file to arbitrary place

  ```
  git show [<commit-id>:]<file> > <resulting-file>
  ```

5 Using GIT for things other than CASINO

Learning to use GIT requires a bit of effort, so you may as well put the knowledge to a broader use! A revision control system has many uses apart from managing software projects. These are a few ideas:

- Keep revisions of LaTeX documents (like this one). Simply type

  ```
  git init
  echo '<output>.pdf' >> .gitignore
  git add .
  git commit -m "Initial commit."
  ```

  and you are set.

- Do collaborative work on LaTeX documents.

- Keep regular backups of your computer's configuration files (`/etc`) using `cron`, e.g.,

  ```
  cd /etc
  sudo git init
  echo '0 */6 * * * root cd /etc && git add . && git commit -m "Cron"' |
      sudo tee -a /etc/crontab
  sudo git add .
  sudo git commit -m "Initial commit."
  ```

  for quarter-daily backups (you may need to use `su -c` instead of `sudo` depending on your distribution's conventions).

- Keep manual backups of your entire `$HOME`.

- ...