LGIT - Version Control for LaTeX Directories

January 10, 2010

Contents

1 Introduction

1.1 Purpose

new bobtilde ~/gits
TEST ~/gits
~/gits /gits
~

~/gits
test 2 ~/gits
file://~/gits ~\gits The lgit collection of python programs and the git system can be used to track multiple project directories in a parallel and consistent manner. The lgit system also contains a setup program that helps to identify important LaTeX directories and then initialize the associated git repositories. The intent is to (a) be able to restore all LaTeX-related source files to a previous state so that documents can be recompiled in their original form and (b) allow for easier recovery of bad LaTeX upgrades by making it easier to return the LaTeX tree to a prior state. Other advantages of lgit are that locally-developed style files and related files can be tracked in parallel with the LaTeXtree.

Part of the challenge of using git for version control for LaTeX is that git usually puts the full repository in the directory that is being tracked, but package manager systems like MacPorts assume that the user will not alter the LaTeX source directories. If these files are altered, upgrading LaTeX programs or the source tree can be difficult. The lgit system puts the git repositories in a remote location and synchronizes repository tags to facilitate parallel operations on the working directories.

The lgit system works by using python scripts to call git and to process a list of directories that are stored in ~/.lgitconf. The system will enforce the application of consistent tags to each commit so that commits can be accessed easily.

1.2 Important Warning

Before running the lgit system, be warned that git does not track file ownership or access rights, and your LaTeX tree might be owned by the root user or other superuser ID. To avoid problems with file access rights (especially if various files are owned by different user IDs), lgit is currently coded to force the user to use the root user for some commands and then it will change file ownership to the non-root user ID. For security reasons, you should not use lgit to track a directory that contains executable files. An alternative might be to change the scripts to always require the root user ID.

2 Installation

2.1 Overview

The `lgit` system is designed for UNIX-type operating systems. It should run on various Linux, Ubuntu, and Mac computers. There are two approaches to installation that will be outlined here and that will be elaborated in the following sections: (a) default initialization and configuration and (b) manual initialization and configuration.

2.1.1 Default/Impatient Configuration

The default configuration should be useful for tracking the LaTeX distribution files. If you want to track a different set of files, then you would need to review the manual configuration procedures below.

1. Install prerequisite programs if needed (Python 3.1, git, LaTeX). Note that Python 2.X will not work.
2. Copy the `lgit` programs to a directory in the execution path. See section XXX on page XXX.
3. Create global defaults for `git` if you have never initialized git. See section XXX on page XXX.
4. Run `lgit.py setup`, which will set default options in `~/.lgitconf`, identify LaTeX directories to track, create the `git` repositories, and initialize those repositories
5. Continue with the normal version control process such as adding files and committing. See section XXX on page XXX.

2.1.2 Manual Configuration

If the default settings do not suite your needs, then the configuration can be set manually.

1. Install prerequisite programs if needed (Python 3.1, git, LaTeX). Note that Python 2.X will not work. See Section 2.2.
2. Copy the `lgit` programs to a directory in the execution path. See Section 2.4
3. Create global defaults for `git` if you have never initialized git. See Section 2.5.
4. Create the `~/.lgitconf`file and create entries for settings as described in Section 2.7.1.
5. Identify the directories that you want to track and code them in the `~/.lgitconf` file. See Section 2.6.

6. Create the `git` repository directories, which are, by default, in `~/gits`. See section XXX on page XXX.

7. Make a conventional archive copy of your computer. See Section 2.8.

8. Read the warnings in Section 2.13.

9. Fine-tune your computer by blocking indexing on the `git` repository. See Section 2.11 on page XXX.

10. Initialize the `git` repositories by running the command `sudo lgit.py init`.

11. Optionally ignore some files from the version control process (not recommended unless you are an expert). See Section 2.12.

12. Choose a nomenclature for `git` tags. See Section 2.10.

13. Continue with the normal version control process such as adding files and committing. See Section 3.

2.2 Python Prerequisites

The scripts require Python version 3 or ahigher. Python is probably on your computer already, but you might have an older version that is incompatible with Python version 3 (versions 2.5 and 2.6 will not work). Python it is available for free from `http://www.python.org/download/` or other places. If you have a Mac, you might want to consider using `http://macports.org` to install Python and related programs. It can install multiple version of Python without being confused and allow you to change the active version by running something like:

```
sudo python_select python31
```

2.3 Git Prerequisites

You must have `git`installed for the `lgit` system to work. Git is available for free from `http://git-scm.org` and other places. If you are using Mac OS X, you might want to use Mac Ports to install the program.

Any version of `git`that allows the `--no-pager`, `--allow-empty`, `--git-dir`, and `--work-tree` options will suffice. The remainder of the `git` commands will be passed from the command line to `git`.

2.4 Copy the Programs

Copy the `lgit.py`, `lgit-commit.py`, `lgitlib.py`, and `lgitw.py` programs to a directory that is in your execution path, such as `/usr/bin`. The copy command would look something like this: `sudo cp lgit.py /usr/bin`

If you do not know your search path, try running `env` and looking for the line that starts with `PATH`. The `lgitlib.py` file can be installed in an appropriate

python library, but if you do not know where that is, just copy it to the same place as the other `lgit` programs.

If the file attributes are lost somehow, you might need to set them: `sudo chmod ugo+x lgit.py`sudo chmod ugo+x lgitw.py sudo chmod ugo+x lgitlib.py

2.5 Global `git` Initialization

If you have never run `git` on your computer, you should create a `git` ID so that any changes that you make can be flagged with your name or ID.

From the command line, run these commands but use your real name and email (if you are using a Mac, look in Applications->Utilities and run the program called *Terminal* and then enter the commands in that window):

```
git config --global user.name "John Doe III"
git config --global user.email your.email@yahoo.com
git config --global color.ui true

git config --global color.diff auto
git config --global color.status auto
git config --global color.branch auto
git config --global color.interactive auto
```

2.6 Identify the Directories to Track

The `lgit` system cannot work unless it knows which directories that you want to track. To make this process easier, `lgit.py` will run a setup routine upon its first execution and it will attempt to identify the correct LaTeX directories to track and the initialize the `git` repositories. The `lgit` system can be used for things other than LaTeX, but you will need to manually enter the directories to track in the `~/.lgitconf` file.

It is recommended to check the contents of the `~/.lgitconf` file to be sure that you are tracking the correct directories. If you develop your own LaTeX style files, bibtex files, or the like, you should be sure that those are tracked. You probably should not track your ordinary LaTeX documents using `lgit`, but you could track them with regular `git`.

The setup routine will identify LaTeX directories by running the command: `texconfig conf`. You might want to run that command yourself after the `lgit` setup routine runs so that you can confirm that your LaTeX options correctly correspond to your working directories. If you run the `texconfig conf` command yourself, you can look under the "kpathsea variables" section of the output and look for entries that begin with codes like TEXMFMAIN, TEXMFDIST, TEXMF LOCAL, TEXMFSYSVAR, TEXMFSYSCONFIG, TEXMFVAR, TEXMFCONFIG, TEXMFHOME, and TEXMFHOME. You probably do not need to track VARTEXFONTS because it typically contains the compiled fonts that you already have. You might also want track additional directories that the setup routine does not automatically identify.

2.7 The Configuration File

2.7.1 Structure of the File

Options for the `lgit` system are stored in the `~/.lgitconf` configuration file. This file holds settings for general options (discussed in section XXX on page XXXX) and it also holds lists of directory-clusters to track.

The configuration file is divided into sections marked by tags that are enclosed in square brackets. In the `[lgit]` section, the entry for `gitdir` points to the root of the `lgit` repository. Under that directory will be the `git` repositories for each of the directories that `lgit` tracks. Under the `[lgitfiles]` section are key-value pairs that represent the subdirectory name for the repository and the full path to the working directories that are tracked. For example, the first entry in that section is for `texmfmain`, which means that a `git` repository will be created in `~/gits/texmfmain` and that directory will contain the version history of the working directory: `/usr/local/texlive/2008/texmf`.

If you intended to use the `push` or `pull` commands, you would also need to population the corresponding entries for the destination or origin for each working directory. If you do not want to push or pull for a given working directory, create an option entry that contains the appropriate key but with not path after it.

Here is an example listing of the `~/.lgitconf` configuration file:

```
[lgit]
requireroot = True
git_rep_root = ~/gits
unlockgitdir = True
logfile = False
promptonpushpull = True

[lgitfiles]
texmfmain = /usr/local/texlive/2008/texmf
texmfdist = /usr/local/texlive/2008/texmf-dist
texmflocal = /usr/local/texlive/texmf-local
texmfsysvar = /usr/local/texlive/2008/texmf-var
texmfsysconfig = /usr/local/texlive/2008/texmf-config
texmfvar = /Users/rehoot/.texmf-var
texmfconfig = /Users/rehoot/.texmf-config
texmfhome = /Users/rehoot/texmf
mktexcnf = /usr/local/texlive/2008/texmf/web2c

[lgitpull]
texmfmain = /q/latex/texmf
texmfdist =  /q/latex/texmf-dist
texmflocal =
texmfsysvar = /q/latex/texmf-var
texmfsysconfig = /q/latex/texmf-config
texmfvar =
texmfconfig =
texmfhome =
```

```
mktexcnf = /q/latex/texmf/web2c

[lgitpush]
texmfmain =
texmfdist =
texmflocal =
texmfsysvar =
texmfsysconfig =
texmfvar =
texmfconfig =
texmfhome =
mktexcnf =
```

2.7.2 Option Settings

The main options are under the `[lgit]` section of the `~/.lgitconf` file.

1. `requireroot`: If set to true (the default), the `lgit` commands that might change the working directory (and a few other commands) must be run from the root user ID. Harmless commands like `lgit.py status` will not require root, although the user can run such command from the root ID if desired.

2. `git_rep_root`: This option holds the path to a directory that will hold all of the `git` repositories that are tracked by `lgit`. The default value is`~/gits`.

3. `unlockgitdir`: If set to `True`, this option will check if the command is run under the root user ID, and if so it will change the ownership of the `git` repository to the regular user ID. This action will be taken only for those commands that are not marked as *safe* such as `status`, `log`, and the like. The default is `True`.

4. logfile = NOT READY YET. This option will create a log file. Default value is `False`.

5. `promptonpushpull` = If set to `True` and the command is either `push` or `pull`, the user will be prompted to confirm the push or pull foreign path and the working directory. The user can then skip that directory if desired.

2.7.3 Manual and Automated Option Settings

The setup routine will not automatically track directories where the executable programs are stored. You can track these directories, but there are some warnings to consider first. The regular `git` system does not track file ownership or access rights, so if you ever checkout an old copy of a file, there is a good chance that the file will not have the correct ownership and access rights—and this can be a security risk. Do not track executable files with `lgit` unless you modify the process to ensure proper security.

If you write your own style files or other LaTeX utilities (not your normal LaTeX documents), you should put them into the TEXMFLOCAL path as shown from the `texconfig conf|less` command. Note that the best place to put your local files is under subdirectories under the LOCAL directory: LOCAL/tex/latex/local for normal style files and LOCAL/bibtex/bib for any personal `.bib` files that you create. Note that you would track only the top of the LOCAL directory and `lgit` will capture all the files under it.

If your review of the `texconfig` output reveals duplicate directories, you should enter only one of them in `~/.lgitconf`. Also, do not track anything under the `/tmp` directory or other temporary directories. You can track directories that are under your home directory, like `~/texmf-config`, but do not track your entire home directory with `lgit` unless you are an expert.

2.8 Make a Conventional Archive

Now that you know where your LaTeX directories are, you should ensure that you have an archive copy of these in case you accidentally ruin them during your initial experimentation with `git`. Copy these files to an external hard drive, flash drive, or some other media in case of a catastrophe. You should really have a full archive of your entire hard drive and know how to restore from the archive. That functionality would be more important than continuing with `lgit`.

2.9 Create Git Repository Directories

Create a directory that will hold all of the *git* repositories. This will be the *git repository root.* The default location is `~/gits`Under that directory, create several more directories that will hold the individual repositories: `texmf`, `texmflocal`, `texmfhome`, and others that you selected to track.

2.10 Decide on a Tag Nomenclature

The `lgit` system will force you to create a tag and a commit message for every commit. You can think of a *commit* as a snapshot of your LaTeX tree that is stored in `git` format. The *tag* is a short nickname that can be used to point to old commits (snapshots) of your LaTeXtree. You might want to create tags in the form of V20100105 that would represent the version form January 5, 2010. The recommended format for tags is VYYYYMMDD format so that the tags sort in a reasonable order when listed (the *V* stands for *version* and might help to reduce confusion from other numeric strings). You use a different format for the tags if you want to, such as V11.9 for version 11.9, but you will probably forget what that means unless the code has some other significance.

2.11 Block Extraneous Processing

If you are using a Mac, you might want to prevent the Spotlight program from indexing your `git` repository because it is a waste of processing time and disk

space. To do this (on Mac only), access the System Settings program, click on the Spotlight icon, look under the Privacy tab, then click the little plus sign at the bottom of the edit box and point to your `~/gits` directory so that it appears in the list of excluded directories.

2.12 Ignoring Certain Files

The best version control approach would be to track every file in the LaTeX tree. So if you are smart, you will skip this section. If for some reason you have limited storage (or mental) capacity, you might decide to not track PDF files or other files that are often not needed to run LaTeX. Note that there might be some graphics packages that use PDF files, PostScript files, or other such things, but in general these files are not needed. Another alternative would be to write a separate script to move all of the unwanted files from the LaTeX tree, although this might cause problems if your installation program expects the LaTeX tree to remain untouched.

In general, there are three ways to tell `git` which files to ignore, and each approach has different functionality: put a `.gitignore` file in your home directory to ignore files globally, put a .gitignore file in the working directory, or put the list of files to exclude in `$GIT_DIR/info/exclude`.

For `lgit`, the best solution would be to add a list of regular expressions (file names or wildcards such as `*.pdf`) in `$GIT_DIR/info/exclude`. For example, you would populate the `exclude` file in `~/gits/texmf-dist/info/exclude`, `~/gits/hometexmf/info/exclude`, and other such places. You could edit the `exclude` file with a text editor and add a line that says `*.pdf` to exclude PDF files. If you don't know how to do this, you should probably not change the files. If you ruin the `exclude` files, you could try deleting them.

There is a global `git` setting that will allow you to ignore all files of a given type from all `git` repositories that you access from your user ID. To do this, put the list of files to ignore in `~/.gitignore`. On a Mac computer, you might want to run a command like this to ignore files that end with a tilde and to ignore files called `.DS_Store`:

```
echo "*~" >~/.gitignore
echo ".DS_Store" >>~/.gitignore
```

Note that the first command would overwrite any existing `.gitignore` file and the second command with the two right angle brackets would append to the file.

If you are using regular `git`, you would normally put the list of wildcards to exclude in a `.gitignore` file in the working directory. This approach would also cause anyone who pulls from that directory to capture that list of files to ignore. This is not a good idea for the `lgit` system because the idea is to not add any junk to the official LaTeX tree.

If you are tracking a project directory that contains a preponderance of files that you do not want to track, it would be possible to ignore all files by adding `*` to the appropriate ignore file and then use the `add -f` command to force the file into the repository. An example command would be `lgit.py add -f *.c` to add a `.c` file to the repository when the normal ignore file would ignore it.

2.13 Warnings

2.13.1 File Ownership

Note that regular `git` does not manage file ownership or file access rights. The current `lgit` system has an option that can require the use of the root user ID for commands that might alter the working directory (option: `requireroot = True` is the default). The effect of this command is that files that are extracted from a `checkout` will be owned by the root user ID.

In addition to keep the `git`repository owned by an ordinary user ID instead of the root ID. This might help to reduce file access problems in the repository itself. If the `unlockgitdir` option is set to `True`, then the `lgit` system will evaluate the `SUDO_UID` and `SUDO_GID` environmenta variables to determine the *regular* user ID and then change the ownership of the `git`repository to that ID.

Before you try to checkout an old version of the repository, you might want to check the file ownership and access rights of the tracked files and directories. You can use a command

```
ls -l
```

and read the user ID and group ID for the files that are listed. If you want to change the user ID of all files in a directory, you can use the command:

```
sudo chown -R theUID:theGROUP *
```

where you would substitute the appropriate user ID and group ID.

2.13.2 Side-Effects of Program Upgrades

You might correctly identify all of the LaTeX directories to track, but after you upgrade your LaTeX binaries there might be different directories that you should track. There will be no notification of this, so you might want to review the *Preparing to Install* directions above and check the LaTeXdirectories again.

2.13.3 Case-Sensitive File Tracking

Historically I have had some minor problems caused by changes in the upper versus lower case of some directories in the LaTeX tree. This might not or might not be a problem with your operating system depending on how it handles differences in names, but `git` will create separate object for directories with different case (I think). One of the directories might have been `/tex/latex/cjk`.

2.13.4 Tracking Binaries

Although I recommend that you not use `lgit` for tracking binary directories because of the file ownership issues, you might want to track some binaries or scripts using some technique. For example, one of the packages that I use requires the `makeglossaries` perl script. Files like this could be tracked with your normal archiving process or another version tracking process.

2.14 Repository Initialization

After you have created your repository directories (by default in `~/gits`), and after you have created your `~/.lgitconf`file, you should run the `init` command:

```
sudo lgit.py init
```

This creates a directory structure in the `~/gits` directories but does not actually save a snapshot of your files.

2.15 Converting Old `tar` Files to Git Repositories

Before you create your first snapshot of your LaTeX tree, you might want to consider adding old archives of your LaTeX tree to the `lgit` repository. To do this, you will need to restore the old copies of your LaTeX tree to a directory—either the official directory or a temporary directory that is then entered into the `~/.lgitconf` file. There are some routines for bringing CVS or SVN repositories into `git` using more automated means, but I have not used them. You can search the regular `git` manuals for information on that. If you create a `git` repository of old version of your LaTeX tree, you should be able to either copy or clone it to the `~/gits`directory.

Assuming that you have archives are of old version of your LaTeX tree (as opposed to new downloads), you should first ensure that you already created the `git` repository under `~/gits`and ran `lgit.py init`. Double check the paths in the `~/.lgitconf` file and if your old archives are in the directories listed in that file, then you can run some commands to capture those files. The commands would look something like this except you can invent your own tag and commit message:

```
lgit.py "add ."
lgit.py "add -u ."
sudo lgit-commit V20081231 "archive of Dec 31, 2008 LaTeX tree from old backup"
```

When this is done, you should put the next version of your old LaTeX tree into the working directory and run THREE commands to capture the files:

```
lgit.py "add ."
lgit.py "add -u ."
sudo lgit-commit V20090131 "archive of Jan 31, 2009 LaTeX tree from old backup"
```

The `add -u .` command will delete any files from the repository that are no longer in the working directory. You need this option so that you can restore the LaTeX tree to the exact state. Note that you do not need the -u option on the very first run, but I included it because it is a very good habit when running `lgit`.

Note that it is OK if you have archives for only one of the LaTeXdirectories. The `lgit` commands will create empty commits for the directories that are empty. The empty commits might be helpful if you ever need to restore the files to an old state and then create a new branch from them.

2.16 First Commit

If you did not load old archives of your LaTeX tree into `lgit`, you can add files and run your first commit at this point. The command might run for a long time as the `git` program scans your directories and created the initial repository, which contains compressed versions of the original files:

```
lgit.py "add ."
lgit.py "add -u ."
```

The `add .` command will add your LaTeX files to a list of files that need to be saved. The command with the -u is not needed if you are starting with an empty repository, but it is a very good habit for `lgit` because it will help to delete unnecessary files from the repository, which is needed to restore the working directory to the precise state that was originally saved. Your files are not actually saved until you run `lgit-commit.py`. To commit the changes, run something like this with your own tag and commit message:

```
lgit-commit.py V20100105 "my first commit."
```

The tag here is a shortcut for the version of my LaTeX directories on January 5, 2010.

3 User Guide

Although `git` has many features, you can use it for version control using relatively few commands. Although the `git` fanatics claim that it is simple to use, a more precise assessment is that it is easy to use if you have a `git` expert next to you that can help you after you make a terrible mistake. The regular `git` program is designed to allow many people around the globe to work on the same project and keep their changes orderly. I will discuss how you can use `lgit` on a single computer for version control, and you can refer to other resources if you decide that you want to share your repository to other computers.

3.1 Querying Your LaTeX Repository

The following commands are the basics for viewing your repository. You might need this information so that you know where to get an old copy of a file.

3.1.1 Status

The *status* command shows which files have changed, which of them will be captured in the next commit, and which files are not captured or otherwise have a problem:

```
lgit.py status
```

This is a safe command and you cannot break anything by using it.

3.1.2 Log of Commits (Snapshots)

Every time you run the *commit* command, `git` scans the entire directory tree and generates a hash code using SHA1. This is a 40-byte code that is most likely unique in its representation of every file in your working directory. The output from some commands show only the first six or eight bytes of the code because that is often sufficient to uniquely identify the object. You can then use either the full 40-byte code, a shorter version of that code, or a tag to point to an old snapshot of your LaTeX directory. The tag is best for `lgit`, but if you use the `lgitw.py` command, you can use the SHA1 code.

```
lgit.py log
```

This is a safe command and you cannot break anything by using it.

3.1.3 Show Tags

To see a list of tag names without much additional information, try this:

```
lgit.py tag
```

This is a safe command and you cannot break anything by using it.

3.1.4 Show Branches

In development of a LaTeX package, you might want to create a temporary branch, edit your LaTeX files and wait until testing is finished before merging the changes into the master branch. For more information on the logic of branches, you can read the online `git` manuals. The general idea is to create and checkout a branch, make commits as needed, then checkout the main branch and merge the temporary branch. The same functionality exists with `lgit`. If you are an individual user who is not developing LaTeX packages, you might have only the master branch. If the listing says that you are not on any branch, then you might have a problem. See Section 4 on page 20 for more information. The regular listing of branches can be seen with:

```
lgit.py branch
```

This is a safe command and you cannot break anything by using it.

3.1.5 List Files in the Repository

```
# Go to the working directory where the file is located.
# Your path might be different:
cd /usr/local/texlive/2008/texmf-dist

lgitw.py "ls-tree -r  HEAD:tex/latex/base"
```

where HEAD refers to the current version of the repository, or you could enter `HEAD~3` for the third prior snapshot. You might want to look in an old snapshot using a tag or commit SHA1:

```
lgitw.py "ls-tree -r  V20091231:tex/latex/base"
```

3.1.6 Showing Changed Files Between Commits

You might want to see which files changed between the current working directory and a prior commit. The `diff` command can do this. You can see a one-line summary by using the `--shortstat` option, or a longer summary with the `--stat` option that will also show the net number of lines were added to each file.

```
sudo lgit.py "diff --stat master~1"

# Between two commits
sudo lgit.py "diff --stat master~1 master~3"
```

You can remove the `--shortstat` or `--stat` options, but you might get some gibberish from binary files that changed.

3.1.7 Blame (Show When Changes Were Made)

If a single file is broken because of a bad edit, you might be able to determine when the file change by using the *blame* command. It is often best to work on a single LaTeX directory for this command, so we will use the `lgitw.py` command (the *w* is for *working directory*).

```
# Go to the working directory where the file is located.
# Your path might be different:
cd /usr/local/texlive/2008/texmf-dist

lgitw.py "blame tex/latex/hyperref/hyperref.sty"
```

For text files, this command will show a listing of the file along with a SHA1 code, a user ID and a date when that line of the file was last changed.

3.1.8

3.2 Day-to-Day Git Operations

Commands issued through `lgit` correspond to regular `git` commands. The most basic set of `git` commands would include some of the operations in the previous section, such as `status` and `log`, plus a set of repository commands discussed in this section.

A typical course of events might be as follows:

1. Add, delete, or modify files in a working directory. This could mean that you download new LaTeX packages, you synchronize with another LaTeX tree, you edit a style file that you wrote or similar actions if you are using `lgit` for something other than LaTeX.

2. Tell `git`about the files that were changed. This is done with the `add` command

3. Check the status of the working directory using the `status` command.

4. Commit changes, which reads the list of files that were added to the pending list and then puts those files into the `git` repository.

A sequence of `lgit` commands might be as follows:

`lgit.py add .` This will tell `git` to identify files that have been added or changed. Those files will be added to a list of pending transactions. Depending on your options, you might have to preface this command with `sudo␣` so that it is executed under the root user ID.

`lgit.py add -u .` The `-u` option tells `git` to register files that have been deleted. This command is very useful for tracking LaTeX source distributions because sometimes many files are deleted when you synchronize with CTAN. The alternative would be to run `git rm` commands, which are exceedingly slow.

`lgit.py status` This command lists all the changes that `git` is prepared to make and it also lists changes that it will not make. If you set your global options according to the instructions in Sections 2.5, then the `status` command should show filenames in green if `git` will capture the changes or red if `git` will not capture the changes. If there are any red filenames, they will be at the bottom of each status listing, but note that `lgit` will cycle through many directories, so you will have to scan the output carefully.

`lgit-commit.py V20091231 ''my commit message''` The commit takes two arguments, a tag name and a commit message. If the commit is successful, then the list of pending changes will be integrated into the repository and you will have a backup copy of your working directory.

3.2.1 Add

If you add, delete, or modify files in a working directory, those changes are not capture until you tell `git` to identify and capture them. This process can imply one or two essential steps plus some commands to check your work. In many cases, this command will add all the files that you want to add:

```
lgit.py add .
```

In some cases, you will be prompted to run the command under the root ID, in which case you would run:

```
sudo lgit.py add .
```

The proper way to delete files from a `git` working directory is to run the `lgit.py rm` *filename* command, but the command is slow. If you deleted many files from a working directory, you can tell git to reflect those changes by use the `-u` option, which is much faster than the `rm` command:

```
lgit.py add -u .
```

If for some reason you told `git` to ignore some files, you might need to force a file into the repository. Use the `-f` option:

```
lgit.py add -f *.py
```

3.2.2 Tag

Because `lgit` is designed to process multiple working directories in parallel, it is important to use tags to facilitate actions such as `checkout` and `branch`. To see a list of tags that are available, run:

```
lgit.py tag
```

Sometimes it is useful to see the tag information in a graphical context, the the interface on several of the `git` front-ends does not support remote repositories very well. One approach that does seem to work is to change the current directory to the git repository (such as `~/gits`/texmflocal) and then run:

```
gitk
```

(do not run `lgitw.py` for this command).

3.2.3 Commit

3.2.4 Graphical Interfaces

Sometimes it is useful to see the commit histories or other repository features in a graphical context, but the the interface on several of the `git` front-ends does not support remote repositories very well. One approach that does seem to work is to change the current directory to the git repository (such as `~/gits/texmflocal`) and then run:

```
gitk
```

(do not run `lgitw.py` for this command).

If you are desperate to use a graphical front-end that does not seem to work with remote repositories, you could try renaming the repository directory (e.g. `~/gits/texmflocal`) to `.git` and then run the application from the `~/gits` directory. This is an emergency measure only, and be sure to rename the directory when you are done.

3.3 Updating the LaTeX Tree

After your `lgit` repository has been fully initialized and you made your first commit, you might download some LaTeX packages or syncronize with another LaTeX tree. You should then add the new files and commit the changes so that you have a snapshot of the LaTeX files.

```
# Show a list of files that changed:
lgit.py status

# Add files that changed to the list of files
# that will be captured in the next commit:
lgit.py "add ."
```

```
# Identify files that need to be removed from the repository
# and schedule them for removal on the next commit:
lgit.py "add -u ."

# Double check the changed files to see if you
# captured them all.  Files should be green.  If they
# are red, then they have not been captured.
lgit.py status

sudo lgit-commit.py V20100131 "downloaded the new hyperref package"
```

3.4 Operations on Only One of the LaTeX Directories

Sometimes you will want to examine files or directories that are in only one of the directories that you are tracking with `lgit`. Depending on the command, you might be able to run the regular `lgit.py` command and simply ignore output from the unwanted directories, or you can change your directory to the working directory in question and use the `lgitw.py` command (the *w* is for working directory). The `lgitw.py` command will detect the current directory, scan the `~/.lgitconf` file to see if that director is listed (while properly detecting if the current directory is a link) and then perform the `git` command only on that directory. You can issue most of the regular `git` commands.

If you change one file in one directory, you should not try to commit that one directory unless you are an expert. If one file changed and you want to commit the changes to the repository, run the `lgit-commit.py` command to apply the new commit and tag to all of the tracked repositories. Some of the directories will have no changes, but that is not a problem. The `lgit-commit.py` command includes the `--allow-empty` option so that tags will be unambiguous as opposed to having multiple tags associated with a single commit in one repository but having the same tags associated with multiple commits in a different repository.

You can use `gitk` when you have sets of related repositories, but you have to use it on one repository at a time. Change your directory to the repository directory (in `~/gits` where the `.git` directory was moved) and then run `gitk`.

3.5 SHA1

The `git` program codes each file and directory with a hash code that helps to ensure that the version in the repository is the same as the version in the working directory. The code is 40 bytes long, and is shown in many `git` screens. If you want to check the code for a single file, you could run something like this:

```
# Go to the working directory where the file is located.
# Your path might be different:
cd /usr/local/texlive/2008/texmf-dist

openssl dgst -sha1 ./tex/latex/hyperref/hyperref.sty
```

If you do not have the `openssl` program and you use a Mac, you might want to get MacPorts and get `openssl` from there.

3.6 Restoring One File or Directory from an Old Snapshot

If you realize that one of the files in your LaTeX tree is bad, you have a few options. You could checkout the old snapshot (checkout an old commit using the tag name such as V201001015) or you could grab the one file or directory from the old snapshot and bring it into your working directory. To grab an old file from the previous commit, try something like this:

```
# Go to the working directory where the file is located.
# Your path might be different:
cd /usr/local/texlive/2008/texmf-dist

lgitw.py "checkout --  tex/latex/hyperref/hyperref.sty"
```

Note the double quotes around the command that is passed to the `lgit` command. Use the `lgitw.py` command (with a *w* for *working directory*) to operate on just one of the LaTeXdirectories.

If the good file is in an old commit, you could try

```
# Go to the working directory where the file is located.
# Your path might be different:
cd /usr/local/texlive/2008/texmf-dist

lgitw.py "checkout  V20091231 -- tex/latex/hyperref/hyperref.sty"
```

Where V20091231 is a tag, or you could use the commit ID that is visible from the `lgit.py log` command.

3.7 Working with Branches

If you are using `lgit` to track the LaTeX distribution files, you probably do not need to create any branches and you could probably skip this section. If you are using `lgit` to track a set of projects that are under development, you might want to create branches to test an idea and, if the test works, merge the test branch into the main branch. You must be careful to double check your work whenever you run branch commands in `lgit` because if the branch command fails on one of the working directories, then you might be in an inconsistent state. Use `lgit.py branch` to assess the current situation both before and after you perform major branch operations. If one working directory is unsynchronized with the others, you might have to change your working directory to that directory and then use the `lgitw.py` command to operate on that directory only.

3.7.1 Creating Branches

3.7.2

3.7.3

3.7.4

3.7.5

3.7.6

3.8 Copying Git Repositories Between Computers

Git was originally intended to manage distributed editing of the Linux core. Some users of `lgit` will likely use it to manage personal copies of the LaTeX tree. If this is the case, you might want to copy your `git` repository from one computer to another. The traditional way to do this would be for you to go to the new computer, use a network connection to connect to the computer that already has the repository, and use the `git clone` command to grab the repository either from a network-mapped drive or from a `git` server. Some users might not be familiar with these techniques or they might have other needs to physically copy `git` repositories between computers.

Note that using the `git clone` command is best way to copy a git repository from one computer to another because it automatically records the origin of the clone and thereby makes it easier to pull from that same source. That being said, it is possible to copy the .git directories from one computer to another. The `git` repositories in the `lgit` system will be under `~/gits` directory and will have names like `~/gits/texmf` and `~/gitstexmf/texmf-var`. These can be copied between computers, but then you need to ensure that the `~/.lgitconf` file contains correct information on the destination computer. After you copy the `git` repositories and fix the `~/.lgitconf`, you can run `lgit.py "checkout master"` to load files from the repository into your working directories.

4 If You Are Stuck With No Branch

If you did something in the past that caused you to be not on any `git`branch, your best bet is to bring the current version into a branch and then examine the situation:

```
# confirm that there are no pending changes
lgit.py status

# If there are pending changes, execute a git-add and
# a git-commit here.

# Check your current tag names so you can pick a unique
# name
lgit.py tag
```

```
# First tag your current commit:
lgit.py "tag Stray001"

# Show the commit history and save it so you know
# your situation (use double >> so each output appends
# to the previous output).
lgit.py "log --decorate=full" >> ~/StrayBranchLog.txt

# Check your current branch names so you can pick a unique
# name
lgit.py branch

# Now create a new branch called "newbranch" that is
# based on StrayBranch20091015 and check it out.
sudo lgit.py "checkout -b newbranch Stray001"

# Confirm that you are on a real branch and that there are
# no pending changes.  Run
#   lgit.py add
# and
#   lgit-commit.py
# whenever you find uncommitted changes.
lgit.py branch
lgit.py status

# If your new branch is active and everything looks OK,
# then checkout the master branch
sudo lgit.py "checkout master"

# If the checkout of master fails, you might need
# to add the checkout -f  option to force it, but it is
# best to examine the situations to see if you need to
# add files or execute another commit before switching branches.

# If you want the changes that are in "newbranch" only,
# then after the master branch is active you could run:
#   lgit.py "merge newbranch"

# If the merge has many conflicts, verify that you have
# no pending changes and commit them before trying again.
# You could  also erase the contents of the working directories, checkout
# the files from newbranch, then run
#   lgit.py "add -u ."
# and
#   lgit.py "add -A ."
# and commit (as is done in my test1.sh from Oct 2009).
```

5 Upgrading LaTeX distribution

1. Commit existing changes before upgrading programs or source tree.
2. For upgrading programs using Debian or MacPorts, it might be best to checkout a new branch, checkout the commit associated with the prior install CHECK THIS.
3. Upgrade the LaTeX programs via the package manager (such as Debian or MacPorts).
4. Run `sudo texhash` and `sudo updmap --syncwithtrees` (or use your methods for updating ls-R files and font maps)
5. Run `lgit.py "add -u ."`
6. Run `lgit.py "add -A ."`
7. Those commands should have told `git` to remove any deleted files from the index and add any new files, and update any changed files.
8. Run `lgit.py "status"` and ensure that all of the changes, additions, and deletions have been properly recognized.
9. Commit changes: `lgit-commit.py V20091014upgrade "upgraded to the new version of texlive version 2007-6"`
10. Sync to `tug` or checkout specific files from a prior commit so that you have all of the new LaTeX source files that you need.
11. Run `sudo texhash` and `sudo updmap --syncwithtrees` (or use your methods for updating ls-R files and font maps)
12. Commit again after you have all the files that you need.

A Change Working Directory Problem

Running `git` version 1.6.0.1 on the MacBook Pro, I get these errors about "`Could not jump back into original cwd`" that also occur when I run on individual directories. The tag operation also fails in these cases. If found some code at http://github.com/git/git/blob/5ad9dce7e691106fecde413de8cc321b937367a6/setup.c that suggests that the error message above might be generated in the event that the path name is too long or of there is some other error in calling `getcwd()`.

The problem might be that my local `texmf` directory was deleted*

The problem might be that somehow my home `texmf` folder was converted to a link to /opt/local/share/texmf, which is problematic for some reason. The direct entry for /opt/local/share/texmf also failed, but I don't know why. The file permissions look OK (after being wrong yesterday).

```
[rehoot:/texlocal] # sudo lgit-commit.py V20090920sync "sync with CTAN as of Sep 20, 2009"
```

```
Processing the work trees and git directories that are listed in /Users/rehoot/lgit01files.txt

Created commit 69ea531: sync with CTAN as of Sep 20, 2009
The commit returned a good code, so I will tag the commit
Created commit f7c0ff2: sync with CTAN as of Sep 20, 2009
The commit returned a good code, so I will tag the commit
Created commit 28ed3d0: sync with CTAN as of Sep 20, 2009
The commit returned a good code, so I will tag the commit
Created commit 643c0fe: sync with CTAN as of Sep 20, 2009
The commit returned a good code, so I will tag the commit
Created commit 8a0bfe0: sync with CTAN as of Sep 20, 2009
The commit returned a good code, so I will tag the commit
Created commit 0fecc15: sync with CTAN as of Sep 20, 2009
The commit returned a good code, so I will tag the commit
fatal: Could not jump back into original cwd
Created commit 4b6734f: sync with CTAN as of Sep 20, 2009
The commit returned a good code, so I will tag the commit
Created commit 016e6ed: sync with CTAN as of Sep 20, 2009
The commit returned a good code, so I will tag the commit
Created commit a7448c1: sync with CTAN as of Sep 20, 2009
The commit returned a good code, so I will tag the commit
Created commit 8c5d540: sync with CTAN as of Sep 20, 2009
The commit returned a good code, so I will tag the commit
fatal: Could not jump back into original cwd
```