Journal of Data Science 0 (0), 1–19
??? 2025
DOI: 10.6339/25-JDS1202
Data Science Reviews

Version Control Systems: Fundamentals for Beginners

Marius Hofert[1,*]

[1] *The University of Hong Kong, Department of Statistics and Actuarial Science, Hong Kong*

Abstract

When collaborating with students, colleagues and practitioners, one soon realizes the lack of efficiency when sending around emails with multiple attachments, especially if changes are made in several types of documents (for example, text, code, PDF) and simultaneously by several collaborators. Using a version control system (VCS) can largely improve joint workflows, from file sharing, including merging changes from different collaborators, to providing access to past versions of the shared work, while allowing each collaborator to work under her/his preferred setup (for example, text editor or file manager). There exists lots of technical or specialized information and literature about VCSes online, but, as often, this is rather overwhelming for beginners. Knowing the basics well is more important than getting lost in the vast amount of possible options VCSes offer. Also, the basics are sufficient to enjoy using VCSes and to see their value in collaborative work, additional features can still be picked up along the way once necessary. We focus on such fundamentals of the centralized VCS SVN and the distributed VCS Git. We explain in simple terms how these systems can be set up and interacted with to increase efficiency in collaborative workflows.

Keywords *centralized; distributed; Git; GitHub; SVN; tips*

1 Introduction

Alice and Bob collaborate on a project and frequently share details in a report (.tex file(s)), develop code together (.R file(s)) or collect others' findings (.pdf file(s)). To avoid having to send files back-and-forth by email all the time, while risking that Alice and Bob both modify the lines in the same files and thus create a *conflict*, the two could use a server for sharing the files. We refer to the main project directory containing all related files (including further sub-directories) as a *repository*. Storing this repository on a server allows Alice and Bob to each *pull* the latest version of all files from the server or *push* their modified versions to the server. However, if the server just contains the repository, this procedure would still lead to conflicts, for example if Alice pushes a modified file to the server that Bob is still modifying locally on his computer. When Bob pushes his changes to the server, the server does not know whether to accept Bob's version or to continue to serve Alice's.

A *version control system (VCS)* keeps track of such file changes, can merge two versions, allows to revert to previous versions, and more; see also Zolkifli et al. (2018). Conflicts are not entirely avoidable, though, think of Bob modifying a file that Alice has already deleted (it is unclear whether the file should then continue to exist or not). Nevertheless, using a VCS can largely simplify joint scientific work. It can even be useful when developing larger projects alone,

* Email: mhofert@hku.hk.

Received July 2, 2025; Accepted October 6, 2025

for the purpose of keeping track of file changes and being able to revert to previous versions if a certain change in the project does not turn out to be fruitful.

The VCS so far described is a *centralized VCS*, there is a server on which the repository resides, one thus also speaks of the repository as a *remote repository* (or simply *remote*). An example of a centralized VCS is Apache Subversion (SVN; subversion.apache.org) with main command `svn`. For example, the sources of the free software environment for statistical computing and graphics R (r-project.org) are jointly developed with SVN. An example server for SVN projects such as contributed R packages is R-Forge (r-forge.r-project.org).

Suppose Alice has no internet connection to the server (for example while traveling, because of an outage, maintenance downtime of the server, etc.). A centralized version control system with a remote repository is only of limited use to her in these situations. She can neither push changes to the server, nor revert to previous versions in case necessary, as the remote repository is not reachable. In these scenarios, it would be good if Alice *also* had a *local repository* (or simply *local*) on her computer. She could then register file changes with the local repository, so *commit* changes to the local repository. Or she could revert to previous versions based on the local repository. When the remote repository is reachable again, Alice could then simply push the changes to the remote repository at that point. This is the main idea behind a *distributed VCS*, namely that there is both a remote and a local repository. An example of a distributed VCS is Git (git-scm.com), originally developed by Linus Torvalds for joint work on the Linux kernel with several thousand collaborators. Popular example servers for Git projects are GitHub (github.com), GitLab (gitlab.com) or Bitbucket (bitbucket.org); we focus on the former in what follows, basic workflows are essentially the same.

Figure 1 highlights the differences between SVN as a centralized and Git as a distributed VCS, including the pull and push operations involved. As introduced above, we refer to the operations interacting with the server as "pull" and "push" throughout the article, even though specific VCSes may have their own terminology for these operations. For example, as can be seen from Figure 1, SVN uses "commit" (instead of "push") in the command to push changes to the server and "update" (instead of "pull") to pull changes from the server.

In Section 2, we will discuss SVN commands for an already existing repository to get to

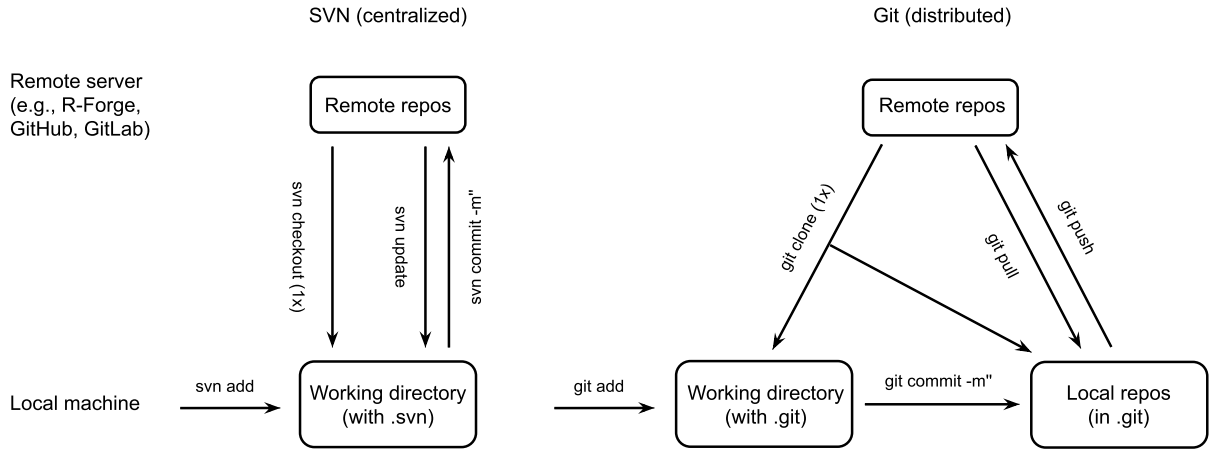

Figure 1: Fundamental operations for the centralized VCS SVN (left) and the distributed VCS Git (right).

know the basics of interacting with a VCS. Section 3 then focuses on the fundamentals of Git, where we cover both the setup of a repository on GitHub and the interaction with Git in more detail. Section 4 then closes with general tips when working with VCSes.

Throughout this article, we follow the KISS principle ("Keep it Simple, Stupid"; (Raymond, 2003, Chapter 1)). As both SVN and Git are straightforward to install (for example with `brew install svn` and `brew install git` on macOS), we omit further installation instructions and assume both VCSes are installed on your computer and a command line interface is available. Finally, note that any account names mentioned throughout this work are understood as synthetic (distinguishing "-local" for local machines, "-rf" for R-Forge accounts and "-gh" for GitHub accounts) and do not represent real accounts that may already exist.

2 SVN

Suppose Alice has already set up the repository on an SVN server. As an example, we use the R package `crop` for cropping figures, which is hosted on R-Forge; Hofert (2024). Alice asks Bob to register for an account on R-Forge and to send her his user name. She can then add Bob to the repository; for this operation, R-Forge also has a "Request to join" link on each repository's website. Once added, Bob can download the repository, which is known as *checkout* operation by SVN. In a terminal, Bob can thus checkout `crop` in his project folder, say `~/pro`, with the following command (typically provided on the hosting server's project website).

```
bob-local:~/pro$ svn checkout svn+ssh://alice-rf@r-forge.r-project.org/svnroot/crop
```

```
A    crop/pkg
A    crop/pkg/DESCRIPTION
A    crop/pkg/man
A    crop/pkg/man/dev_off_crop.Rd
A    crop/pkg/R
A    crop/pkg/R/dev_off_crop.R
A    crop/pkg/R/tools.R
A    crop/pkg/NAMESPACE
A    crop/pkg/TODO
A    crop/www
A    crop/www/index.php
A    crop/README
Checked out revision 8.
```

To download the repository under the directory name `myrepos`, Bob could have instead just provided this name via `svn checkout svn+ssh://alice-rf@r-forge.r-project.org/svnroot/crop myrepos`.

Bob can now navigate to any subdirectory of `crop`, add files and make changes to the repository using his favorite software on his local computer (text editor, file manager, etc.). Say he modified `./crop/pkg/TODO` and added `myfile.R` to `./crop/pkg/R`. After having done so, he can display the status of all files in the repository via `svn st` in order to find out which ones have changed.

```
bob-local:~/pro$ cd crop
bob-local:~/pro/crop$ svn st
```

```
?       pkg/R/myfile.R
M       pkg/TODO
```

In the first column, the letter `M` shows that the file `TODO` was modified. The question mark indicates that SVN does not know of this file that Bob created in `./pkg/R`. This is because the file is in the repository's directory structure, but Bob did not inform SVN to put it *under version control*, so to register it with SVN in order for SVN to keep track of its changes. This can be done as follows.

```
bob-local:~/pro/crop$ svn add pkg/R/myfile.R # alternative: cd pkg/R; svn add myfile.R
```

Now `svn st` indicates the added file by the letter `A` in the first column.

```
bob-local:~/pro/crop$ svn st
```

```
A       pkg/R/myfile.R
M       pkg/TODO
```

Before `myfile.R` was under version control, Bob could have simply moved the file, for example to rename it, with no effect (it would still show up with a `?` and can be added to the repository after the move). However, now that it is under version control, renaming it from `myfile.R` to `mynewfile.R` creates the following output, which shows `myfile.R` as missing (with exclamation mark) and `mynewfile.R` as not under version control (with question mark).

```
bob-local:~/pro/crop$ svn st
```

```
!       pkg/R/myfile.R
?       pkg/R/mynewfile.R
M       pkg/TODO
```

If Bob had used `svn mv myfile.R mynewfile.R` to rename the file, SVN would have registered this operation properly and would have continued to track `mynewfile.R` thereafter. We see that `svn st` is helpful in showing each file's status; files deleted from the repository via `svn rm` would appear with letter `D`.

When Bob is done with his work, he wants to push his changes to the server so that Alice can pull them from the server once she continues to work on the project. Bob can push all changes made in any of the files under version control to the server via the below `svn ci -m' mymessage'` command, where `ci` is a shorthand for `svn commit` and where `mymessage` should be a meaningful, short message (the *commit message*) about what has been done (typically a couple of words).

```
bob-local:~/pro/crop$ svn st
```

```
A       pkg/R/myfile.R
M       pkg/TODO
```

```
bob-local:~/pro/crop$ svn ci -m'added more functionality and adapted TODO'
```

```
Adding         R/myfile.R
Sending        TODO
Transmitting file data ..done
Committing transaction...
Committed revision 9.
```

When Alice wants to continue to work on the project, she should *first* do a pull operation to update her local repository with the potential changes Bob previously pushed to the server. If Bob did not work on the project since Alice last pushed her changes, no files need updating

(which SVN realizes of course). If Alice has checked out the repository in her local directory ~/package_crop, the pull operation in SVN can be done as follows; note that `up` is a shorthand for `update`.

```
alice-local:~/package_crop$ svn up
```

```
Updating '.':
U    pkg/R/myfile.R
U    pkg/TODO
Updated to revision 9.
```

We see that Alice's files are now updated with the changes Bob had previously uploaded to the server. From now on, Bob should also remember to do a pull operation via `svn up` every time *before* he continues to work on the project again, to minimize the chances that he modifies a file that Alice had already modified and pushed to the server, so to minimize the chances of a conflict appearing once Bob tries to push his changes to the server. Also, Bob should push his changes to the server fairly frequently *during* his work and soon *after* finishing to work on the project for a longer time period to again minimize the chances of a conflict. Likewise for Alice, of course.

To get an overview over changes in the repository over time, the command `svn log`, executed after `svn up`, can be useful. Here we display the top 12 lines in the log file, containing the most recent three changes.

```
bob-local:~/pro/crop$ svn log | head -12
```

```
------------------------------------------------------------------------
r9 | bob-rf | 2024-05-28 09:44:52 +0800 (Tue, 28 May 2024) | 1 line

added more functionality and adapted TODO
------------------------------------------------------------------------
r8 | alice-rf | 2024-05-27 17:21:20 +0800 (Mon, 04 Mar 2024) | 1 line

minor fixes
------------------------------------------------------------------------
r7 | alice-rf | 2024-05-23 17:17:28 +0800 (Mon, 04 Mar 2024) | 1 line

added new crop feature
```

To also include the information about which files changed, Bob could have used the verbose version `svn log -v | head -20`, for example.

There is much more functionality available for SVN, but these typical operations should give you a basic understanding of how to interact with a VCS. Similar commands are available for Git; see Section 3. In case you wonder how SVN or Git 'magically' keep track of your local files, there is a hidden directory (called `.svn` for SVN and `.git` for Git) in the top-level directory of each repository that does this 'magic' for you.

3 Git

We now turn to the (nowadays more popular) distributed VCS Git and its convenient hosting server GitHub for Git repositories. To this end, suppose both Alice and Bob have accounts (username and password) on github.com; see the end of Section 1 concerning usernames used here. As a general reference for Git, consider Chacon and Straub (2014).

3.1 Setup of a Repository on GitHub

Alice decides to set up a new repository on GitHub. First, she asks Bob to share his username with her, so that she can add him to the new repository. This then enables Bob to download the new repository, so to *clone* it in Git parlance. Second, Alice logs in to github.com, presses the green button "New" in the top left corner of the webpage, provides a repository name, say `myrepos`, and chooses to make this a *private repository* (only Alice and collaborators can see the repository); the default choice is a *public repository* (anyone on the internet can see it). She hits the green button "Create repository" at the bottom to create the repository. Next, GitHub displays information about the newly created repository from which Alice gets the repository's address (here: `git@github.com:alice-gh/myrepos.git`). She saves this information to be able to clone the repository later. Third, Alice navigates to "Settings" at the top of the page to take her to the settings of the repository `myrepos`. After hitting "Collaborators" and then "Add people", Alice can search for Bob by his username and select "Add <Bob's username> to this repository". Bob then receives an email invitation for the project to the email address his GitHub account is registered under. Once he accepts this invitation, he can also see the project's address `git@github.com:alice-gh/myrepos.git` and can clone the repository; see Section 3.3 for the command.

Note that if Alice selects her avatar on the top-right and chooses "Your repositories", she obtains an overview of all her repositories. Clicking a particular one and going to its settings allows her to delete a repository via "Delete this repository". In private repositories such as `myrepos`, only the owner Alice is allowed to delete it, not her collaborator Bob.

Finally, note that GitHub accounts also come with limitations, for example, concerning file sizes. Added via the command line (browser), files cannot exceed 100 MiB (25 MiB). For larger files, the extension Git Large File Storage (Git LFS; git-lfs.com) can be used, which replaces large binary files by pointers. Git LFS can be installed via `git lfs install` and putting a large file under Git LFS version control can be done via `git lfs track`. Afterwards, just follow the normal Git workflow detailed later. Another tool for version control of large datasets, images, audio or video files is Data Version Control (dvc.org).

3.2 Password-Free Interaction with GitHub

Whenever Alice tries to pull/push files from/to the remote repository or otherwise interact with GitHub, she will be prompted for a password, which makes interacting with GitHub cumbersome. Fortunately, GitHub accepts authentication by public-key cryptography.

To set this up, Alice first navigates to her home directory `~/` (on Windows typically `C:\Users\alice-local`) to check if she already has an existing directory `~/.ssh`. It should look something like the following.

```
cd ~/.ssh
ls -lh # list directories in long format and with more readable file size units
```

```
total 60K
-rw------- 1 alice-local staff 2.8K 2023-11-07 18:34 config
-rw------- 1 alice-local staff 1.8K 2022-11-24 12:38 id_ed25519
-rw------- 1 alice-local staff  396 2022-11-24 12:38 id_ed25519.pub
-rw------- 1 alice-local staff  24K 2024-05-16 13:07 known_hosts
```

If the *private key* `id_ed25519` and the *public key* `id_ed25519.pub` do not exist yet, Alice can generate them via the following command, with her correct email address specified.

```
ssh-keygen -t ed25519 -C "alice@example.com"
```

When executing this command, Alice will be prompted for a password for the key pair at some point. Here it is important that she just hits the return key and does *not* provide a password. If she provides a password, she will be prompted for this password on every interaction with GitHub again. However, if she does not provide one, the connection to GitHub will still be safe due to the key pair authentication, but she will not be prompted for a password anymore when interacting with GitHub.

Now that Alice has the *private key* `~/.ssh/id_ed25519` and the *public key* `~/.ssh/id_ed25519.pub` (either previously set up or newly generated as just described), she needs to take the public key (and only that; never share the private key with anyone) and upload it to GitHub so that GitHub can use it for authentication. The public key should look something like `ssh-ed25519 [...] alice@example.com`, where `[...]` is used as a placeholder for a longer string here. Alice copies this key (without additional whitespace), navigates to github.com, clicks her avatar, then "Settings" and then "SSH and GPG keys". There she clicks "New SSH key", pastes the copied public key into the textbox "Key", sets the "Key type" to "Authentication Key" (the default) and then chooses "Add SSH key"; it is not necessary to provide a title for the key. After these steps are completed, she should be able to run Git commands from the terminal without being prompted for a password on every interaction with the server. She tells Bob to also set up a key pair for authentication.

Besides the above-used algorithm Ed25519 (Edwards-curve digital signature algorithm) for generating a key pair, a still widely used algorithm is that of RSA (Rivest–Shamir–Adleman), which can be generated via `ssh-keygen -t rsa -C "alice@example.com" -b 4096`, where the last argument specifies the number of bits in the key (which should be sufficiently large). The private and public keys in `~/.ssh` would then be named `id_rsa` and `id_rsa.pub`, respectively.

3.3 Basic Git Commands

3.3.1 Cloning a Repository

Bob now can download (or, as already mentioned, clone) the repository into his project folder `~/pro` as follows; the same applies to Alice, of course.

```
bob-local:~/pro$ git clone git@github.com:alice-gh/myrepos.git
```

```
Cloning into 'myrepos'...
warning: You appear to have cloned an empty repository.
```

This will generate the directory `~/pro/myrepos`; had Bob used `git clone git@github.com:alice-gh/myrepos.git work_with_alice`, the directory containing the repository would have been `~/pro/work_with_alice`. Currently, Git reports that the repository is empty as it contains no files under version control. However, it is not truly empty.

```
bob-local:~/pro$ cd myrepos
bob-local:~/pro/myrepos$ ls -al
```

```
total 0
drwxr-xr-x   3 bob-local staff  96 2024-05-28 13:23 .
drwx------+ 11 bob-local staff 352 2024-05-28 13:23 ..
drwxr-xr-x   9 bob-local staff 288 2024-05-28 13:23 .git
```

We see that the repository contains the already mentioned directory `.git` which is similar to the `.svn` directory for SVN in that it does the version control 'magic' for you in the background; inside `.git` also resides the local Git repository (but one never has to touch or access `.git` or its content explicitly).

3.3.2 Adding Files, Checking the Status

Bob now adds the files `report.tex` and `TODO` to the directory `myrepos`. With `git status -s` he can check the status of the repository, similarly to the SVN command `svn st` already seen.

```
bob-local:~/pro/myrepos$ git status -s
```

```
?? TODO
?? report.tex
```

As for SVN before, Bob first needs to tell Git to put these files under version control.

```
bob-local:~/pro/myrepos$ git add TODO report.tex
bob-local:~/pro/myrepos$ git status -s
```

```
A  TODO
A  report.tex
```

Now the two files have been added. Moving files under version control with Git (for example for renaming them) can be done via `git mv <oldfile> <newfile>`. For deleting files, there is `git rm <myfile>`, however (and in contrast to SVN), just deleting a file normally will also correctly set its status to `D` to schedule it for deletion from the repository on the next push operation.

3.3.3 Committing and Pushing

When Bob is done with his work, he wants Alice to consider his changes. In SVN we simply used `svn ci -m''` to this end. However, we already mentioned that Git is distributed, so has a local repository and a remote one. Hence Bob's workflow for sharing his changes with Alice consists of two steps, the *commit* operation, where Bob's changes are registered with his local repository and the actual *push* operation, where Bob's changes are uploaded from the local repository to the remote one so that Alice can receive Bob's changes on her next pull operation. If Bob ignores the commit and tries to push directly, he runs into an error.

```
bob-local:~/pro/myrepos$ git push
```

```
error: failed to push some refs to 'github.com:alice-gh/myrepos.git'
```

First committing and then pushing works, of course. To this end, note that a commit message is required; if you need quotes, use back quotes inside the message (as quotation marks are already used as string delimiters).

```
bob-local:~/pro/myrepos$ git commit -m'added files `TODO`, `report.tex`; started
    report' -a
```

```
[main (root-commit) 871bae7] added files `TODO`, `report.tex`; started report
 2 files changed, 1001 insertions(+)
 create mode 100644 TODO
 create mode 100644 report.tex
```

The option `-a` indicates that all files that have changed will be committed, so will be included in the current commit; if only selected files shall be committed, replace `-a` by their file names, separated by a blank, as usual.

At this point, Bob has committed his changes to the local repository, but, as already indicated, Alice would not see them on a `git pull` from the server yet. Bob first needs to also do the second step and push the changes to the remote repository before Alice can see them on her `git pull`.

```
bob-local:~/pro/myrepos$ git push
```

```
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 10 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 15.31 KiB | 7.65 MiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:alice-gh/myrepos.git
 * [new branch]      main -> main
```

If Alice now runs `git pull` from anywhere in her local repository, she will get Bob's updates. Similarly as for SVN, both Alice and Bob should always remember to do a `git pull` *before* they start to work on the repository's files and to do a `git commit -m''` followed by a `git push` *after* they finished their work on the repository. The more often this is done, the less likely a conflict will appear when both Alice and Bob modify the same lines in the same files.

3.3.4 Reviewing the Commit History

Let us assume that Alice updated the `TODO` file, committed and pushed it, and Bob pulled it. To get an overview of the changes in the repository over time, the command `git log` can be used (similar to the SVN command `svn log` already seen).

```
bob-local:~/pro/myrepos$ git log | head -12
```

```
commit a6d88b7393fc905a8db331abb8cb2330ad07f852
Author: Alice <alice@example.com>
Date:   Tue May 28 18:11:23 2024 +0800

    updated TODO

commit 26dc1f9a57f394348506e1e3a54dd7112b566e2d
Author: Bob <bob@example.com>
Date:   Tue May 28 18:06:02 2024 +0800

    added files `TODO`, `report.tex`; started report
```

In the first line of each commit, we see the commit hash which uniquely identifies this commit.

3.4 More Advanced Git Commands

The presented operations are already the most fundamental ones one should know when working with VCSes. As for SVN, there is much more functionality available for Git, some features occasionally of interest are described next; another one, yet again more advanced, is described in Ap-

pendix A. For even more advanced features, see the online Git documentation (git-scm.com/doc) or Chacon and Straub (2014).

3.4.1 Commit Differences

To see file differences between two commits, one can use the following command; note that one can abbreviate the commit hashes.

```
bob-local:~/pro/myrepos$ git diff 26dc1f9 a6d88b7
```

```
diff --git a/TODO b/TODO
index c907db8..1438a8e 100644
--- a/TODO
+++ b/TODO
@@ -1 +1,2 @@
 - Bob: Add more of the literature review
+- Alice: Start coding next week
```

When executing `git pull`, one often wants to see the differences between the previous version and the obtained latest version of all files, which can be viewed via `git diff HEAD^ HEAD`.

3.4.2 Revert to a Previous Version

Although this should rarely be needed, based on the commit hashes, one can also revert a repository to a previous stage. Executing

```
git push
git revert <hash_to_revert_to>..<latest_hash_to_revert>
```

one can revert all commits from `<latest_hash_to_revert>` (included) to `<hash_to_revert_to>` (not included), so revert the repository to its commit hash `<hash_to_revert_to>`. To also include the commit related to the first provided hash in the reversion, one can use `^`, so the reversion command would then be of the form `git revert <earliest_hash_to_revert>^..<latest_hash_to_revert>`. And if only an individual commit (but no others) should be reverted, one can simply use `git revert <hash_to_revert>`.

What first-time Git users are often afraid of (unsubstantiated) is that they accidentally delete files in a repository. Suppose Bob worked on the repository, made changes to `report.tex`, but accidentally deleted `TODO`. Besides reverting to a previous version, a more brute-force but perhaps easier to implement approach is the following. Bob could simply keep a local copy of his changes in `report.tex` outside the repository `myrepos`, then delete his local version of `myrepos` entirely, clone the repository again to get the last working version and then insert his changes to `report.tex` manually again (for example, using a more fail-safe text editor or file browser that is capable of displaying file differences). He can then follow the normal procedure of `git commit` and `git push`. This should really not be required, but is, in principle, a valid option that is also easy to understand and occasionally used by inexperienced users.

3.4.3 Solving a Conflict

Suppose Alice and Bob both edited the same line in `report.tex`. Alice committed and pushed her changes. Afterwards, Bob is just done committing his changes and tries to push them to the remote repository. He then sees the following.

```
bob-local:~/pro/myrepos$ git push
```

```
To github.com:alice-gh/myrepos.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'github.com:alice-gh/myrepos.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

As instructed, Bob first executes a pull operation to obtain the changes from Alice he was not aware of.

```
bob-local:~/pro/myrepos$ git pull
```

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 970 bytes | 194.00 KiB/s, done.
From github.com:alice-gh/myrepos
   e410dbb..3fc8090  main       -> origin/main
hint: You have divergent branches and need to specify how to reconcile them.
hint: You can do so by running one of the following commands sometime before
hint: your next pull:
hint:
hint:   git config pull.rebase false  # merge
hint:   git config pull.rebase true   # rebase
hint:   git config pull.ff only       # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a default
hint: preference for all repositories. You can also pass --rebase, --no-rebase,
hint: or --ff-only on the command line to override the configured default per
hint: invocation.
fatal: Need to specify how to reconcile divergent branches.
```

We see that the pull operation failed as Git does not know how to combine the changes of both Alice and Bob when encountering this situation for the first time in a repository. As typically the correct choice, Bob chooses `git config pull.rebase false` to make Git try to merge his and Alice's changes automatically on future pulls. The `true` option would temporarily remove Bob's commit and reapply his changes in comparison to Alice's latest ones (so Bob's latest changes are based on Alice's latest changes now, which explains the term "rebase"). And the `git config pull.ff only` first checks if Bob has unique commits not present remotely, if not then pull will succeed ("fast-forward merge" possible) and otherwise Git will not perform a merge but instead allow Bob to manually incorporate his changes and thus resolve this divergence.

If there are no changes by Alice and Bob in the same lines, the following `git pull` would merge their changes and would thus be successful. However, as Alice and Bob made changes in the same line of the same file, Bob's pull operation creates a conflict as Git does not know how to merge the two changes in the sense of whether to accept Alice's or Bob's version.

```
bob-local:~/pro/myrepos$ git config pull.rebase false
bob-local:~/pro/myrepos$ git pull
```

```
Auto-merging report.tex
CONFLICT (content): Merge conflict in report.tex
Automatic merge failed; fix conflicts and then commit the result.
```

Bob now has to solve this conflict before he can commit and push his changes. To this end, he looks into `report.tex` which created the conflict and finds the following change that Git made in this file.

```
<<<<<<< HEAD
In this joint paper, Alice and Bob try to analyze\dots
=======
In this joint study, Alice and Bob try to analyze\dots
>>>>>>> 3fc8090b5b2c1f7fa6a97cdf38387f3cec9c7d32
```

The first part, between the markers `<<<<<<<` and `=======`, contains Bob's version that he wanted to push, the second part, between the markers `=======` and `>>>>>>>` is Alice's version that she already pushed. A simple way for Bob to solve this conflict is now to decide which version to keep and to remove the other version including all markers. Bob decides to keep Alice's version, so he corrects this part of `report.tex` as follows and removes all markers.

```
In this joint study, Alice and Bob try to analyze\dots
```

After that, Bob can commit and then push the changes. Note that his push contains both his original changes that Git could have already merged with Alice's changes and the changes Bob just made to fix the conflict due to the overlap with Alice's version.

```
bob-local:~/pro/myrepos$ git commit -m'updated report and fixed conflict' -a
bob-local:~/pro/myrepos$ git push
```

Both commands now work flawlessly. Note that if Git detects several lines that create a conflict, several markers as indicated before are found in the respective files. Bob would need to solve all of such instances in all affected files before he can commit and push.

Another simple but brute-force way to solve a conflict for Bob is to do a "force pull" and just overwrite all his changes by Alice's, trusting her that she uniformly did a better job than he did. Bob's changes will then all be lost, of course, so also those that did not create a conflict with Alice's version. As already mentioned, Bob could keep local copies of the files he changed outside the repository and manually include his changes after this "force pull" operation. As before, this is not a fail-safe solution but easier to apply in case of many conflicts (think of Bob forgetting to push changes to the server over a longer time period and Alice driving the project forward in the meanwhile, with lots of changes Bob agrees on to be incorporated). To this end, Bob could simply use `git fetch --all; git reset --hard origin/main` to overwrite all his changes by Alice's. Also here, this should not be required very often but is good to know that it exists. It also demonstrates the power of Git, for virtually all situations there is a (sequence of) command(s).

3.4.4 .gitignore

Bob works on his `TODO` file and adds `publication.pdf` to the directory `./lib`. He then also writes about it in the report file `report.tex`.

```
bob-local:~/pro/myrepos$ mkdir ./lib # then Bob copies publication.pdf into ./lib
bob-local:~/pro/myrepos$ git add ./lib/publication.pdf
```

```
bob-local:~/pro/myrepos$ git commit -m'added a reference and updated report.tex' -a
bob-local:~/pro/myrepos$ git push
bob-local:~/pro/myrepos$ git pull # Bob checks whether there are changes from Alice
```

To make sure he only pushes files to Alice that actually compile (so that Alice can keep working on them uninterruptedly and not having to first find out why the files do not compile), he compiles `report.tex`. Afterwards, he checks the status of the repository's files again and sees the following.

```
bob-local:~/pro/myrepos$ git status -s
```

```
?? report.aux
?? report.bcf
?? report.log
?? report.out
?? report.pdf
?? report.run.xml
```

As indicated by Git, these temporary files are not under version control but always show up on `git status -s`. We do not want to put these files under version control as they frequently change and can thus unnecessarily cause conflicts, as they can sometimes be large in size and thus prevent efficient version control, and as they simply clutter the output of `git status -s`, for example. How can Bob tell Git to simply ignore them? With a file `.gitignore`, typically residing in the top-level directory of the repository where also the hidden folder `.git` is located, so in `./myrepos` in this case. Bob puts the following in `.gitignore`, which also includes some more file endings of temporary files that should be ignored by Git.

```
*-blx.bib
*.aux
*.bbl
*.bcf
*.blg
*.idx
*.ilg
*.ind
*.log
*.out
*.ptc
*.rel
*.run.xml
*.synctex
*.synctex.gz
*.toc
auto/
report.pdf
```

Note that Bob specifically also includes `report.pdf` but not `*.pdf` as the latter would lead to Git ignoring all future additions of PDF files in `./lib` and Bob wants them to be under version control to share them with Alice (which is fine for such PDF files as they rarely change so it is unlikely that they produce a conflict). Now temporary files do not appear anymore when checking the status, only `.gitignore` itself, as Bob newly created this file in the repository.

```
bob-local:~/pro/myrepos$ git status -s
```

```
?? .gitignore
```

Bob therefore adds, commits and pushes `.gitignore` to be part of the repository so that also Alice benefits from it after her next `git pull`.

```
bob-local:~/pro/myrepos$ git add .gitignore
bob-local:~/pro/myrepos$ git commit -m'added .gitignore' .gitignore
bob-local:~/pro/myrepos$ git push
```

Note that we also demonstrated here how a single file can be committed by providing it after the commit message; this is equivalent to `git commit -m'added .gitignore' -a` in this scenario here as `.gitignore` was the only file that changed since the last commit.

4 General Tips

We close with some general tips for working with VCSes.

1) Do not be discouraged by the (perceived) more involved setup steps or "learning curve" to get a VCS running for you. Yes, this is a bit more involved than simply using Overleaf (for collaborative LaTeX projects; overleaf.com) or Dropbox (for sharing files; dropbox.com). However, once in place, VCSes are much more powerful and flexible. Also, as they work in the background (instead of the foreground such as Overleaf and Dropbox, partly forcing you to work in a browser), they allow you to work completely under your preferred workflow, with your preferred software (text editor, file manager, PDF viewer, etc.). You can use whatever tools you have used so far locally to work on the files in a repository and thus do not disrupt your workflow, whereas using multiple online software such as Overleaf and Dropbox to (only) get a minor fraction of the functionality (for example Overleaf only hosts LaTeX projects) requires to partially or fully alter your workflow. Furthermore, don't be discouraged to use VCSes only because most of your collaborators prefer to work with other tools, as integration is often available. For example, Overleaf for LaTeX projects also provides Git integration, so that you can work with Git while other collaborators can edit files on overleaf.com (you receive changes they made in the browser GUI via `git pull` then, as usual).
2) Follow the KISS principle, ask your collaborators to set up the repository for you if that part of the process is unclear to you, start with the very basic commands to execute the pull and commit/push operations, learn how to add/move files under version control, how to check file differences and the status of your local repository. You most likely for a very long time will not need more advanced features. And once you do, at that point you have sufficient experience to utilize more advanced features.
3) As long as you follow the main rule to first do a pull operation before making changes (for example, in the morning) and a commit/push operation after you made changes (for example, when going for lunch), the chances of producing a conflict should be fairly minimal for reasonable collaboration team sizes, and can be further reduced by frequent intermediate commit/push operations. If a conflict is unavoidable, a careful read of the respective messages is advisable. For example, the output message of Git on the conflict we encountered in Section 3.4.3 provided all steps we needed in order to solve the conflict.
4) Write short, but meaningful commit messages; for more professional commit messages than those we used in this article, see, for example, conventionalcommits.org.

5) Provide file names according to a reasonable convention. For example, in a project folder with one source file and 100 figures, it makes sense to name the figures with the same prefix (such as `fig_*`), so that they are all listed together when viewing the project folder in the file manager or terminal (if they all start with different prefixes, it may be hard to even find the actual important source file in the folder). Or name files in lower case and without blanks to avoid additional key presses when typing them in the terminal (locally or remotely). For file content, although a bit archaic by now, keep the maximum number of characters per column limited to 80, which allows for a more readable output when displaying file differences. For more such tips, see Hofert and Schepsmeier (2016), Hofert and Schepsmeier (2017a) and Hofert and Schepsmeier (2017b).
6) Work with the command line interface, because in most bigger projects, you need to anyways. For example, all 500 of the top 500 most powerful non-distributed computers in the world are Linux based and a vast majority (if not all) require command line interface access. Also, as already mentioned, on GitHub the maximal allowed file size when uploaded via command line interface is four times as large as when uploaded via the web interface.
7) If you are not a fan of the command line, then GitHub comes with the GUI GitHub Desktop (desktop.github.com). It contains buttons for the various operations that follow the same terminology we used before and often (as the command line interface does) assists with instructions what operations to execute next. Also other pieces of software come with Git integration, for example editors (such as Visual Studio Code) or integrated development environments (IDEs) (such as RStudio). Note, however, it is still good to know the shell commands as they are useful when working remotely (for example, to clone software hosted on GitHub to install in your home directory on a remote cluster).
8) As indicated when discussing the meaning of `.gitignore`, when certain source files are frequently used to produce binaries, only the source files (not: their binaries) should be put under version control in order to avoid conflicts produced by (frequently changing) binaries as much as possible. Also, binaries can be rather large in size and thus prevent efficient version control (for example, GitHub does not even allow for files exceeding 100 MiB to be under version control, which binaries such as PDF files can easily reach).
9) As already mentioned, using a VCS can also be advisable if you do not collaborate. In bigger personal projects, important files under version control can be reverted back if necessary and a private repository on a server can act as some sort of backup in case you forget to do your proper backups regularly. We write "some sort of backup" as Git is not recommended to be used for backups, for example Git ignores file ownership information. Nevertheless, the server provides you with a remote copy of the most important files in your project in case your local files get damaged.
10) You can download (checkout/clone) a repository in multiple locations on multiple of your devices. As long as you follow the main rule to first do a pull operation before making changes and a commit/push operation after you made changes, you can work from either of them.

A Git Branches

If Alice and Bob collaborate on the same source file in a private repository and the main direction of implementation is clear, they typically both commit and push their changes to the same line

of development within the repository, the *main branch*. However, if they work on a piece of software in a public repository, whose latest version needs to remain stable, they would not want to risk pushing changes that are still experimental and not sufficiently tested yet to the stable code base. Say Bob wants to implement a new feature, but is not sure whether this should be included in the main branch. He could, conceptually, copy all files from the repository to another folder and continue his development in parallel there. But this would either clutter the current repository (if the folder is located within the repository) or would lead to various additional project folders (if located outside the repository). Either way, a lot of the code would be overlapping with the main branch but would not get updated if there are updates to the main branch. To avoid such problems when following different lines of development in parallel within the same repository, Git provides Bob with the option of opening additional *branches*. If Bob is done with the implementation of the feature in a different branch (the *feature branch*), Alice can check out his implementation in the feature branch, try it, see if she finds it useful and ready to be included into the stable code base in the main branch.

As a first task, Bob would like to know if there are already different branches available, so he lists all branches as follows.

```
bob-local:~/pro/myrepos$ git branch -a
```

```
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
```

Bob sees that he still works on the main branch, indicated by an asterisk. After a `git pull` to update the main branch, he decides to create the feature branch (which can be done with `git branch feature`) and change to it (which can be done via `git switch feature`, more classically via `git checkout feature`). Both operations can be done together as follows.

```
bob-local:~/pro/myrepos$ git switch -c feature
```

```
Switched to a new branch 'feature'
```

Now Bob lists all branches again and sees that the feature branch was created and that he is now working on that branch.

```
bob-local:~/pro/myrepos$ git branch -a
```

```
* feature
  main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
```

Bob can now make changes to the feature branch, perhaps execute a series of `git add` and `git commit` commands. Assuming he is done with his work, Bob is still in the feature branch and would now like to `git push` from there, so let us see what happens.

```
bob-local:~/pro/myrepos$ git push
```

```
fatal: The current branch feature has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin feature
```

```
To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'.
```

Git doesn't know which branch on the remote server Bob's local feature branch should be connected (and thus pushed) to. A shorter solution than the one suggested in the error message is the following, where `-u` abbreviates `--set-upstream`.

```
bob-local:~/pro/myrepos$ git push -u origin feature
```

Now a tracking relationship between Bob's local feature branch and the remote (tracking) branch `origin/feature` is established. Subsequent `git push` or `git pull` commands implicitly refer to this connection and require no explicit specification of the branch names anymore. As Bob's implementation of the feature is done now, he switches back to the main branch; note that switching to the previous branch can also be done with `git switch -`.

```
bob-local:~/pro/myrepos$ git switch main
```

How does this look like on Alice's side? Assuming her local repository resides in `~/myrepos`, she does a `git pull`, followed by a `git branch -a` to see the following.

```
alice-local:~/myrepos$ git branch -a
```

```
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/feature
  remotes/origin/main
```

Similar to Bob before, Alice can then switch to the feature branch to check out Bob's new feature, and perhaps modify it, etc.

```
alice-local:~/myrepos$ git switch feature
```

Suppose Alice decides that the feature is ready to be incorporated into the stable codebase in the main branch. She would thus like to *merge* the feature branch into the main branch. Since she already is in the feature branch, and also to be more fail-safe, Alice instead decides to first do the opposite, namely to merge the main branch into the feature branch, so that she can solve potential conflicts in the feature branch first, making the final merge into the main branch cleaner.

```
alice-local:~/myrepos$ git merge main
```

Now she changes back to the main branch to merge the feature branch into the main. This merge already creates a *merge commit* (unless it is a *fast-forward merge*, where there are no new commits in the main branch not already present in the feature branch's history; the option `--no-ff` would force a merge commit even if the merge is a fast-forward merge), so no additional `git commit` is necessary, but only a `git push`; here it is helpful to just carefully read Git's messages, if a commit is required, Git will request it.

```
alice-local:~/myrepos$ git switch - # change back to the previous branch (main)
alice-local:~/myrepos$ git merge feature # merge feature into main
alice-local:~/myrepos$ git push # push the merge commit
```

Alice may now want to delete the feature branch, which can be done by first deleting the remote feature branch and the local branch tracking the remote feature branch, and then by deleting the local branch, leading to the following two commands, respectively.

```
alice-local:~/myrepos$ git push -d origin feature
alice-local:~/myrepos$ git branch -d feature
```

There are several things to note here that help to understand branches. After the first command, the output of `git branch -a` is the following.

```
  feature
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
```

We see that the local feature branch is still there. Alice could have also executed the first command from the local feature branch, then the output above would show the asterisk in front of `feature` instead of `main`. But the second command she cannot execute from the local feature branch (this would show the error `error: Cannot delete branch 'feature' checked out at '/Users/alice-local/myrepos'`, so she would need to change to the main branch first. After executing the second deletion command (`git branch -d feature`) from the main branch, Alice then sees the following on `git branch -a`, as expected after also deleting the local feature branch.

```
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
```

Suppose Alice forgot to merge the feature branch into the main branch and that she would execute `git branch -d feature`. Then Alice would see `error: The branch 'feature' is not fully merged. If you are sure you want to delete it, run 'git branch -D feature'.` So to force delete the feature branch, she could use `git branch -D feature` as Git suggested.

After the feature branch was merged into the main branch, a merge commit is created that ties the two branches together. This preserves the history as it happened, but may lead to a rather cluttered commit history if various lines of development are opened. An alternative is to base the feature branch on the latest commit in main (thus "replay" the feature branch commits in comparison to the latest commit in main) and to create new commits at the time of this *rebase* operation. This can be done with `git rebase`. If Alice wanted to do that (instead of `git merge`), she could proceed as follows.

```
alice-local:~/myrepos$ git fetch origin # update the main and feature branch
alice-local:~/myrepos$ git switch feature # change to the feature branch
alice-local:~/myrepos$ git rebase origin/main # rebase the feature branch onto the
    remote tracking branch
alice-local:~/myrepos$ git push -f origin feature # force (-f) push the rebase
    operation (-f required here)
alice-local:~/myrepos$ git switch - # change to the previous branch (main)
alice-local:~/myrepos$ git merge feature # fast-forward merge into the main branch
alice-local:~/myrepos$ git push origin main # push the main branch (with the merge) to
     the remote
```

Note that if Alice runs into conflicts during the rebase, she can solve them and then continue the rebase operation with `git rebase --continue` or abort it with `git rebase --abort`.

Acknowledgement

The author would like to thank editor Jun Yan and two anonymous reviewers for valuable feedback on this work.

References

Chacon S, Straub B (2014). *Pro Git*. Apress. 2 edition. https://git-scm.com/book/en/v2.

Hofert M, Schepsmeier U (2016). Guidelines for statistical projects: General aspects (Part i). *International Chinese Statistical Association Bulletin*, 28(2): 110–116.

Hofert M, Schepsmeier U (2017a). Guidelines for statistical projects: Coding and typography (Part ii). *International Chinese Statistical Association Bulletin*, 29(1): 52–58.

Hofert M, Schepsmeier U (2017b). Guidelines for statistical projects: Coding and typography (Part iii). *International Chinese Statistical Association Bulletin*, 29(2): 113–122.

Hofert M (2024). crop: Graphics Cropping Tool. r-forge.r-project.org/projects/crop.

Raymond ES (2003). *The Art of UNIX Programming*. Pearson Education.

Zolkifli NN, Ngah A, Deraman A (2018). Version control system: A review. *Procedia Computer Science*, 135: 408–415. https://doi.org/10.1016/j.procs.2018.08.191