Version control for reproducible science

Brad Duthie

17 November 2022

Contents

These notes are provided as a guided tutorial to accompany a workshop focused on the use of version control to improve workflow in scientific research, but they can also be used as a standalone introduction to the subject. These notes will help the reader begin using git version control software and the GitHub hosting service. An introduction to git is provided using GitKraken software. For an introduction that includes the command line interface (CLI), an alternative set of notes also exists as part of a tutorial in the Stirling Coding Club. After finishing this tutorial, the reader should be able to use version control in their own scientific workflow.

Introduction: What is version control?

Version control is any system that records changes made within a set of files over time so that different versions of files can be managed and, if necessary, recovered. **Put more intuitively**, version control is a way of taking a snapshot in time (called a 'commit') of all the files in one of your folders (called 'repositories'). As you make changes to the files within your folder, you can always come back to previous snapshots that you've taken (if, e.g., you make a change that you regret, or need information from a previous point in time). You can even have multiple different versions of the same folder existing in parallel (called branches). You can think of it as an extra step on top of 'saving' a file – a step that solidifies a key point in time for your work, records how it changed from previous and subsequent points in time, and records who made the change, when, and why.

Version control is indispensable for large coding projects with multiple developers collaborating on the same code, but it's also a very useful tool for the workflow of scientific research. Using version control can allow you to better manage data files, analysis files (e.g., R code), manuscript files, and more in a way that keep things clean and removes the anxiety of losing track of which file is the 'right' one.

Version control is also an excellent tool for doing open science. By keeping a record of how your data, analysis, and manuscripts change over time, the process of doing science becomes more transparent. By uploading your progress to GitHub, you can make the whole process of doing science accessible to others, and have evidence of priority and accuracy in your conclusions (you can also keep repositories private).

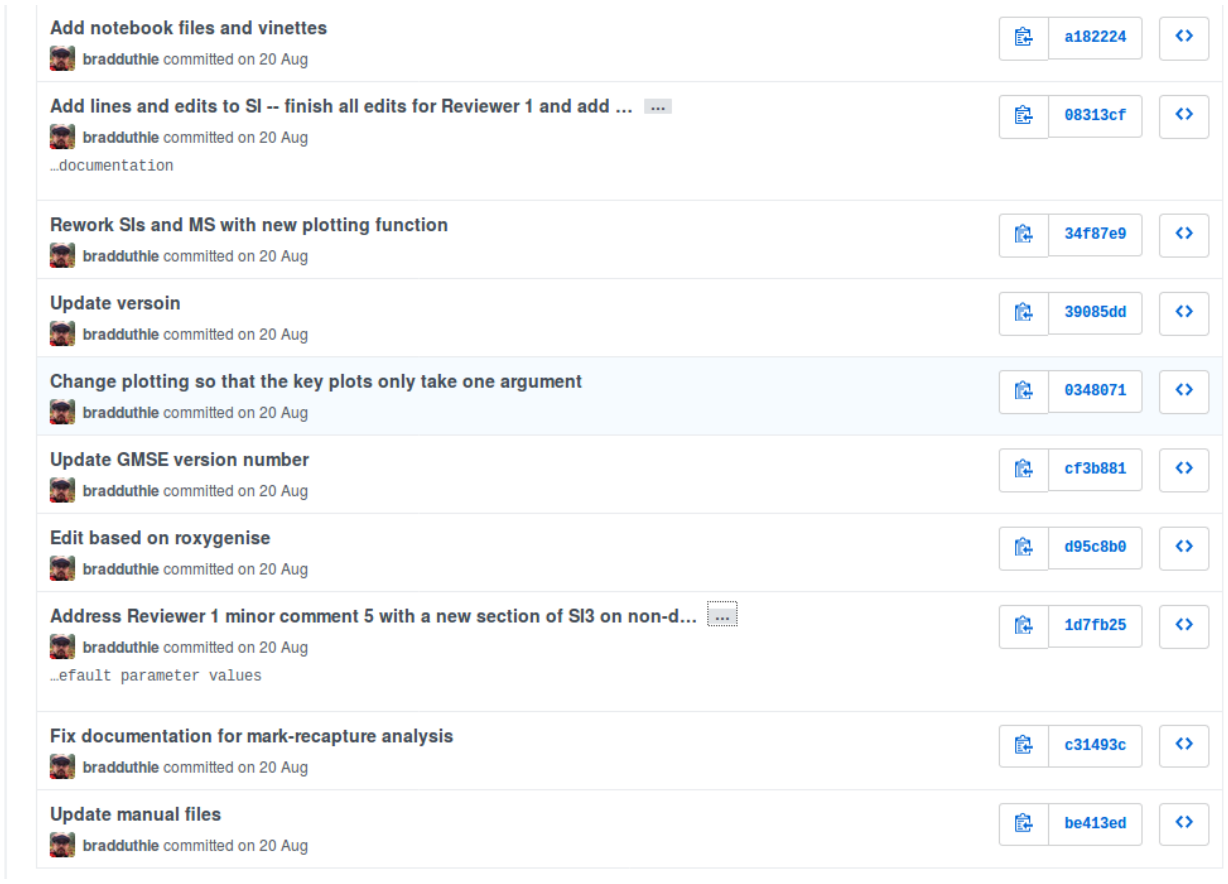
There are many different types of version control available. Here, I am going to focus only on git version control software, which has the advantage of being free, open source, available on all platforms (Linux, Mac, and Windows), and the most popular software among research scientists. The software was invented by Linus Torvalds, the same developer who created the Linux kernel.

In this introduction to using version control, I am going to focus heavily on using two software tools that work with git, GitHub and GitKraken. Like git, both GitHub and GitKraken are free for basic use, though more advanced options can come with a small cost. These two tools make using git much easier, especially if you don't like the idea of working within the command line. GitHub offers a massive online platform where you can store your git repositories, discover and download new repositories, and collaborate with other GitHub users (e.g., in organisations such as the Stirling Coding Club). GitKraken provides a nice graphical user interface for using git, visualising your repository, and linking to GitHub. As you become proficient with git, you might find yourself start thinking less in terms of individual files and file versions, and more in terms of commits and branches with inter-related files.

Things to do before getting started

The software git, by itself, does not have any user interface outside of the command line (i.e., without any other software, you would use it by typing instructions into a computer terminal). Below is an example of what the command line interface (CLI) looks like in use.

```
brad@duthie-pc:~/Dropbox/teaching/workshops/version_control$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   vc_notes.html
    modified:   vc_slides.Rmd
```


Add notebook files and vinettes
braddthie committed on 20 Aug
a182224

Add lines and edits to SI -- finish all edits for Reviewer 1 and add ...
bradduthie committed on 20 Aug
...documentation
08313cf

Rework SIs and MS with new plotting function
bradduthie committed on 20 Aug
34f87e9

Update versoin
bradduthie committed on 20 Aug
39085dd

Change plotting so that the key plots only take one argument
bradduthie committed on 20 Aug
0348071

Update GMSE version number
bradduthie committed on 20 Aug
cf3b881

Edit based on roxygenise
bradduthie committed on 20 Aug
d95c8b0

Address Reviewer 1 minor comment 5 with a new section of SI3 on non-d...
bradduthie committed on 20 Aug
...efault parameter values
1d7fb25

Fix documentation for mark-recapture analysis
bradduthie committed on 20 Aug
c31493c

Update manual files
bradduthie committed on 20 Aug
be413ed

Figure 1: *An example of the timeline of commits for a recent project. Bold titles on top show more recent changes committed to the repository; the bold messages are written at the time of committing, and make it easier to see important changes added over time. In GitHub, you can click on these bold titles to see what changes were made since the last commit; from here, you can also see the whole repository as it was during the time of commit (you can also do this by clickling on the* < > *buttons on the right).*

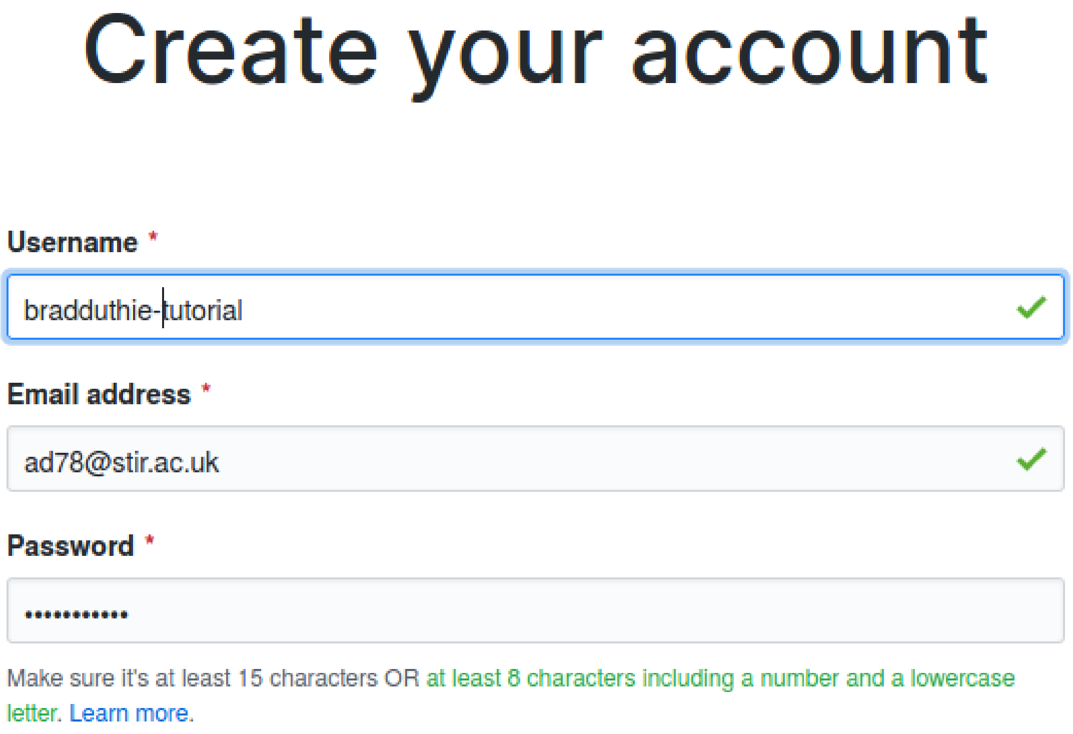
```
    modified:   vc_slides.pdf

no changes added to commit (use "git add" and/or "git commit -a")
brad@duthie-pc:~/Dropbox/teaching/workshops/version_control$ git add *
brad@duthie-pc:~/Dropbox/teaching/workshops/version_control$ git commit -m  "An example commit to demon
[master cf544ae] An example commit to demonstrate what the command line environment looks like
 3 files changed, 765 insertions(+), 585 deletions(-)
 rewrite vc_notes.html (65%)
```

If you do not regularly work with the terminal, this kind of interface can be challenging. Here we will avoid it entirely by using GitHub and GitKraken, both of which have user interfaces that make it much easier to work with git (no command line necessary).

Work flow with git will be much easier if you have everything associated with a particular project organised into a single folder on your computer (including data, analysis files, notes, manuscript drafts, etc.). Lastly, there are some key details about how files are stored and differentiated that are useful to know.

1. Signing up with a GitHub account

If you are just starting out with git, I recommend signing up with a GitHub account. GitHub is a free site where you can host your own projects and collaborate with other researchers. It is mostly known as a host for code and software development, but any projects that are version controlled with git can also be hosted here. Chances are, if there is an R package that you reglularly use (e.g., ggplot2, vegan, lme4), then the code and entire development history will be publicly available, along with a issues board containing known bugs, questions, and requests for new features.


Create your account

Username *

braddduthie-tutorial ✓

Email address *

ad78@stir.ac.uk ✓

Password *

••••••••••••

Make sure it's at least 15 characters OR at least 8 characters including a number and a lowercase letter. Learn more.

Select a username and password to sign up, then click on the button to select a plan. A free plan is likely to be sufficient for all of your needs, but if you are a student, then you can also apply for a student developer pack. In either case, you will be able to create an unlimited number of public and private (i.e., only visible to you) repositories and upload them to GitHub.

> **Exercise:** *Create a GitHub profile. If you have already created a profile, then add some information to personalise your profile, or apply for a student developer pack.*

2. Download GitKraken software

I recommend downloading GitKraken for getting started with git. By downloading GitKraken, you will also simultaneously be downloading git. GitKraken itself is a Graphical User Interface (GUI) to make git much easier to use; no command line is necessary, so all of your version control needs are taken care of in a graphically rich point and click interface. For those who prefer to use the command line interface, see these notes from Stirling Coding Club.

GitKraken is also free, and it easily integrates with GitHub and your GitHub account. And if you are a student and apply for the GitHub student developer pack as suggested above, the pack will come with a free version of GitKraken Pro.

> **Exercise:** *Download GitKraken, then open it.* ***When prompted, choose the option to sign into GitKraken with your GitHub account****. Next, familiarise yourself with the GitKraken environment.*

3. Understanding folder structure

It is important to know where your project folder and all relevant files are stored on your computer. This might sound obvious, but it is actually easy to lose track of this information with a lot of software. Some programs can make it easy to not pay attention to where something is being saved or opened from, e.g., by picking a reasonable default location to save or having a File pulldown menu that lists 'Recent files' to open. Nevertheless, version control aside, it is good practice to always know where files are located on your computer, and it is generally more efficient to keep all of the files within a particular project in the same folder.

The location of your files (i.e., the path to your directory) will look a bit different depending on whether you are using Linux, Mac, or Windows. For example, here is where the file that you are reading is stored on my computer (Linux).

```r
getwd(); # In R, returns the working directory
```

```
## [1] "/home/brad/Dropbox/teaching/workshops/version_control"
```

If you run Mac, a path to directory might look something like the below.

```
/Users/brad/Dropbox/teaching/workshops/version_control
```

And if you run Windows, your path to directory might look something like this.

```
C:/Users/brad/Dropbox/teaching/workshops/version_control
```

Note that in all of these examples, the words between forward slashes represent nested folders that lead to a folder of interest (in this case, 'version_control'). Your path might look different from any of the above, but the idea is the same. The folder or 'repository' that I am working with is named 'version_control', which itself is inside a folder called 'workshops', which is inside a folder called 'teaching', etc. Knowing this path to your project folder will make it easier to work with git, and easier to keep your work flow more organised. Note that 'version_control' itself can have files and sub-folders inside it.

```
list.files(path = ".", include.dirs = TRUE); # In R, shows files in working dir
```

```
##  [1] "Bumpus_data.csv"                      "git_cheat_sheet.pdf"
##  [3] "GitKraken_GitHub_cheet_sheet.pdf"     "GitKraken_interface_cheat_sheet.pdf"
##  [5] "ignored_folder"                       "images"
##  [7] "list.md"                              "Rscript_example.R"
##  [9] "RStudio_and_git.html"                 "RStudio_and_git.Rmd"
## [11] "vc_notes.html"                        "vc_notes.Rmd"
## [13] "vc_slides.pdf"                        "vc_slides.Rmd"
```

The above is a list of all of the files in the current working directory, so, e.g., the path to this file is as below.

```
/home/brad/Dropbox/teaching/workshops/version_control/vc_notes.html
```

Note that there is a sub-folder (which has no extension such as .pdf or .Rmd) named 'images' (and also one named 'ignored_folder'), which is where I have saved all of the images for these notes and accompanying slides.

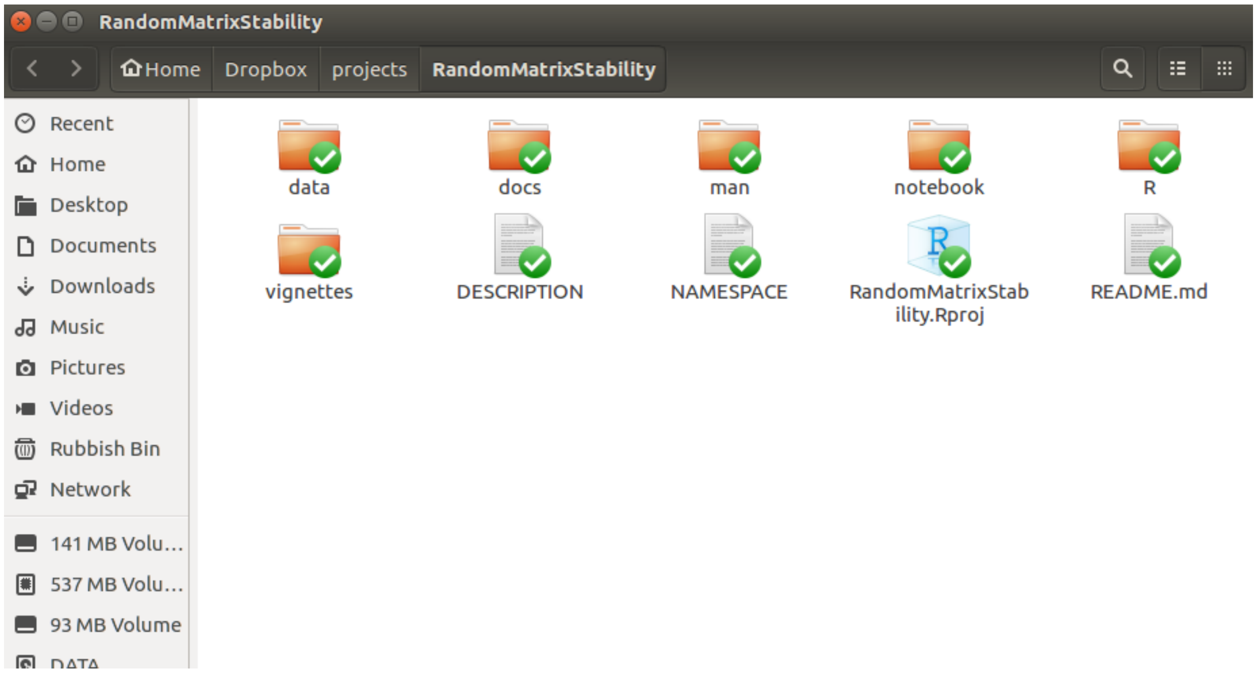
```
list.files(path = "images", include.dirs = TRUE);
```

```
##  [1] "CLI_merge_conflict.png"          "CLI_merge_necessary.png"
##  [3] "duthiefalcy.png"                 "Git_hidden_inside.png"
##  [5] "Git_hidden.png"                  "git.png"
##  [7] "GitHub_branch_2.png"             "GitHub_branching.png"
##  [9] "GitHub_commit.png"               "github_create_account.png"
## [11] "GitHub_create_file.png"          "GitHub_create_new_branch.png"
## [13] "GitHub_create_rep.png"           "GitHub_edit_list_bottom.png"
## [15] "GitHub_edit_list_top.png"        "GitHub_file.png"
## [17] "GitHub_fork.png"                 "github_logo.png"
## [19] "GitHub_new.png"                  "GitHub_pull_request.png"
## [21] "GitHub_URL.png"                  "GitHub_view_list.png"
## [23] "GitKraken_1.png"                 "GitKraken_2.png"
## [25] "GitKraken_3.png"                 "GitKraken_4.png"
## [27] "GitKraken_Abort.png"             "GitKraken_add_remote.png"
## [29] "GitKraken_branch_2.png"          "GitKraken_branch.png"
## [31] "GitKraken_branching.png"         "GitKraken_clone.png"
## [33] "GitKraken_collaborate.png"       "GitKraken_filechange.png"
## [35] "GitKraken_fixed_merge.png"       "GitKraken_forked.png"
## [37] "GitKraken_init.png"              "GitKraken_list_branches.png"
## [39] "GitKraken_merge.png"             "GitKraken_multibranch.png"
## [41] "GitKraken_push.png"              "GitKraken_repo_manage.png"
## [43] "GitKraken_resolve_merge.png"     "GitKraken_toolbar.png"
## [45] "GitKraken_top.png"               "GitKraken_two_list_branches.png"
## [47] "GitKraken_unstaged.png"          "GitKraken_vc_timeline.png"
## [49] "gitkraken.png"                   "gitlab.png"
## [51] "init_GitKraken.png"              "RMS_rep_eg.png"
## [53] "SCC_Logo_slide_sm.png"           "VC_intro.png"
## [55] "vc_timeline_intro.png"
```

To better understand the notation and lists above, you can browse 'version_control' on GitHub to see the files and image folder.

4. Putting project files in a common folder

There are a lot of ways that you can organise your workflow, but I strongly recommend giving any unique project on which you are working its own folder. This could be a folder for each unique manuscript in preparation or dissertation chapter, for each module to which you participate in as a student or teaching role, or for each administrative task that you perform. This makes version control easier by allowing you to track the entire history of an individual project (comprised of multiple inter-related files within a folder). An example for a research project might look like the below.



Separate folders in this project include data, analysis code, a manuscript in preparation, and project notes. The entire history is tracked on git and available on GitHub.

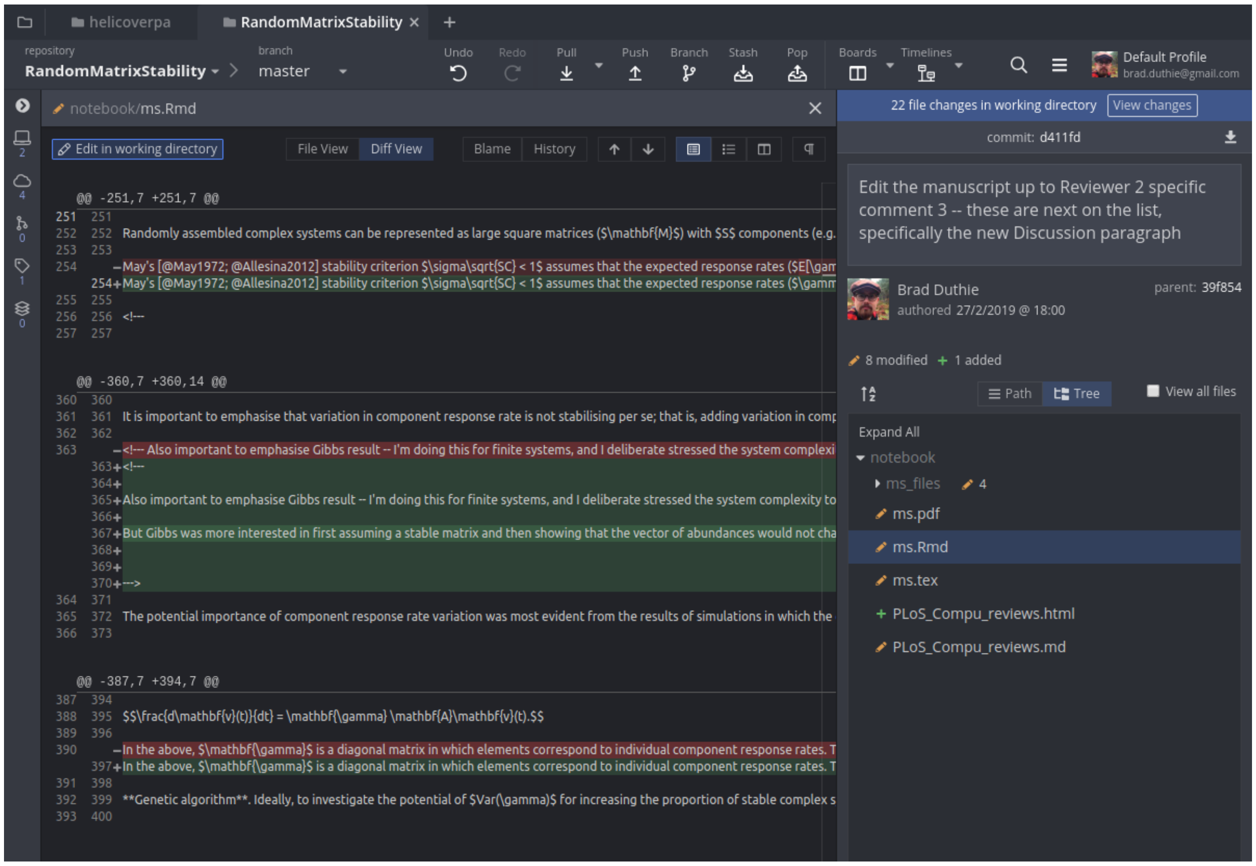
> **Exercise:** *Identify a location on your computer where you would like to create a new folder to be later initialised as a git repository. Do not actually create this folder yet; we will do this later from GitKraken.*

5. Understanding text versus binary files

When working with git, it is important to understand the difference between text files and binary files. These are two general categories of file types; both work with git, but you can do a bit with text files that you cannot do with binary files. For the purpose of working with git, all that is important to know is the following:

- **Text files** can be viewed as plain text in any sort of text editor (e.g., Notepad, TextEdit, gedit) or Integrated Development Environment (e.g., Rstudio, VIM, Emacs). They include files with extensions such as .txt, .R, .csv, .html, .md, .tex (among many others). If you are unsure if something is a text file, try opening it up in a plain text editor such as Notepad or TextEdit. If what you open is readable, then you have probably opened a text file. If what open looks incoherent, then you have probably opened a binary file.
- **Binary files** need to be opened with special software to be readable (e.g., MSWord, Adobe Acrobat, Photo Viewer). They include files with extensions such as .pdf, .docx, .xlsx, .pptx, .jpg, .gif (among many others). If you try to open any of the aforementioned file types in a text editor such as Notepad, you will see a mess of characters that is impossible to interpret.

This distinction matters because while both text and binary files can be used with git, the only way to see changes made to a binary file is to open up old and new versions of it and manually compare them side by side. In contrast, git can point out changes made to text files directly; below is an example showing the changes between different versions of a text file (ms.Rmd) in the GitKraken interface.



```
@@ -251,7 +251,7 @@

Randomly assembled complex systems can be represented as large square matrices ($\mathbf{M}$) with $S$ components (e.g.

-May's [@May1972; @Allesina2012] stability criterion $\sigma\sqrt{SC} < 1$ assumes that the expected response rates ($E[\gam
+May's [@May1972; @Allesina2012] stability criterion $\sigma\sqrt{SC} < 1$ assumes that the expected response rates ($\gamm

<!--


@@ -360,7 +360,14 @@

It is important to emphasise that variation in component response rate is not stabilising per se; that is, adding variation in comp

-<!-- Also important to emphasise Gibbs result -- I'm doing this for finite systems, and I deliberate stressed the system complexi
+<!--
+
+Also important to emphasise Gibbs result -- I'm doing this for finite systems, and I deliberate stressed the system complexity to
+
+But Gibbs was more interested in first assuming a stable matrix and then showing that the vector of abundances would not cha
+
+
+-->

The potential importance of component response rate variation was most evident from the results of simulations in which the


@@ -387,7 +394,7 @@

$$\frac{d\mathbf{v}(t)}{dt} = \mathbf{\gamma} \mathbf{A}\mathbf{v}(t).$$

-In the above, $\mathbf{\gamma}$ is a diagonal matrix in which elements correspond to individual component response rates. T
+In the above, $\mathbf{\gamma}$ is a diagonal matrix in which elements correspond to individual component response rates. T

**Genetic algorithm**. Ideally, to investigate the potential of $Var(\gamma)$ for increasing the proportion of stable complex s
```

commit: d411fd

Edit the manuscript up to Reviewer 2 specific comment 3 -- these are next on the list, specifically the new Discussion paragraph

Brad Duthie
authored 27/2/2019 @ 18:00
parent: 39f854

8 modified + 1 added

- notebook
  - ms_files 4
  - ms.pdf
  - ms.Rmd
  - ms.tex
  - + PLoS_Compu_reviews.html
  - PLoS_Compu_reviews.md

Gitkraken shows lines that have been deleted (red) and added (green) after a file has been updated. This might not be important for some files that you work with, but it's good to keep in mind when using version control. **In the next section, we will get started by making a git repository. You should now have a specified location on your computer where you can get started.**

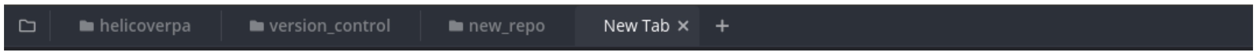
> **Exercise:** *Try to open some different some* ***small*** *files on your computer in a text editor such as NotePad or TextEdit to see what happens. Try it with an R or CSV file, then again with a DOCX or XLSX file.*

Using GitKraken

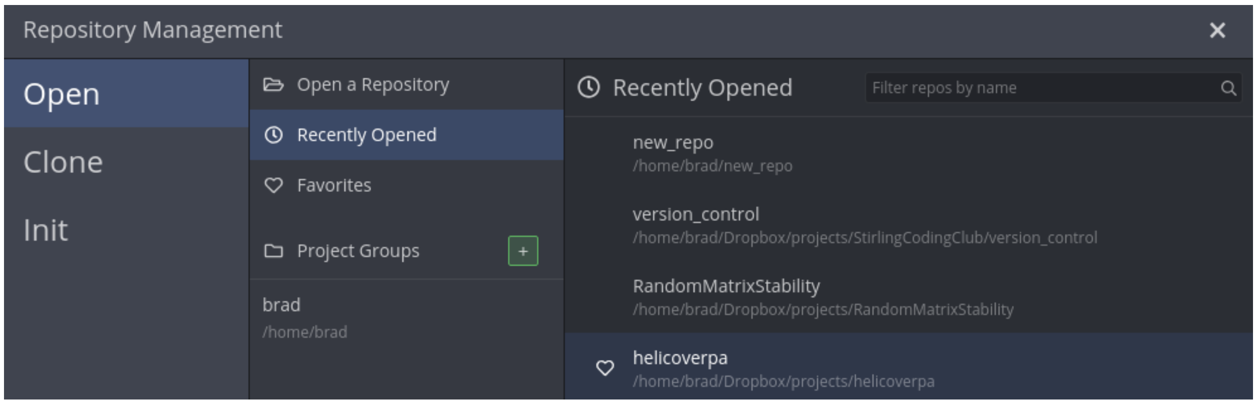
This set of instructions will get you started working with git and GitHub. Throughout this guide, I will explain how to use the basic functions of git and link git with GitHub using the free software GitKraken. GitKraken can run alongside whatever program you're using to actually edit your text files (e.g., Rstudio, MSWord, MSExcel, etc.), but you don't typically make edits from within GitKraken itself – GitKraken is more like a file manager in this way; you can keep it open to stage, commit, push, pull, etc. (more on these later), while working on files in your git repository. GitKraken will monitor any changes to the files inside your repository while you work.

1. Initialising a git repository

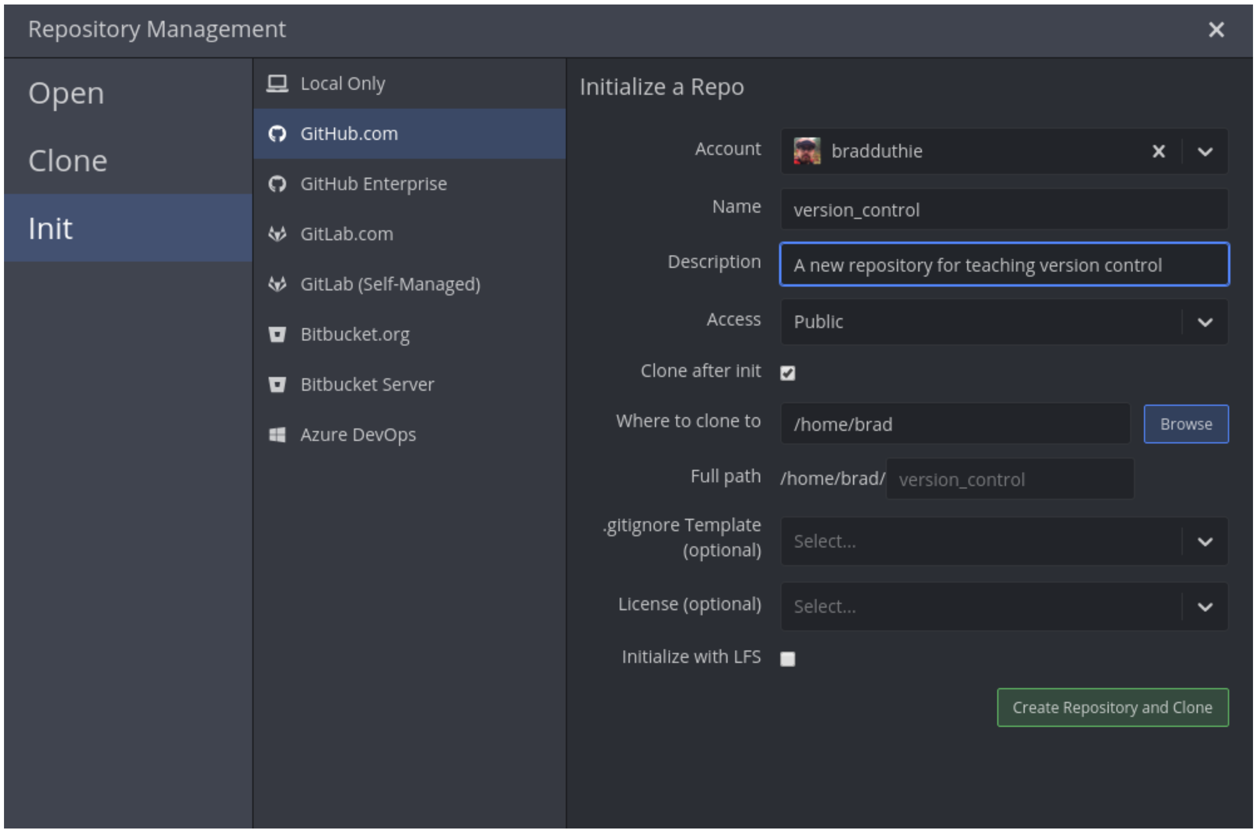
When you open GitKraken, you should see a toolbar at the top of the program that looks something like the below (ignore the specific names, 'helicoverpa', 'version_control', etc.).



Click on the folder outline icon on the very left. This should open the Repository Management window, which will look something like the below.



To initialise a new repository, click on the option 'Init' on the left. This will open up a window that looks something like the below.


Repository Management

Open	Local Only	Initialize a Repo
Clone	GitHub.com	
Init	GitHub Enterprise	
	GitLab.com	
	GitLab (Self-Managed)	
	Bitbucket.org	
	Bitbucket Server	
	Azure DevOps	

Account	braddutie
Name	version_control
Description	A new repository for teaching version control
Access	Public
Clone after init	✓
Where to clone to	/home/brad
Full path	/home/brad/ version_control
.gitignore Template (optional)	Select...
License (optional)	Select...
Initialize with LFS	

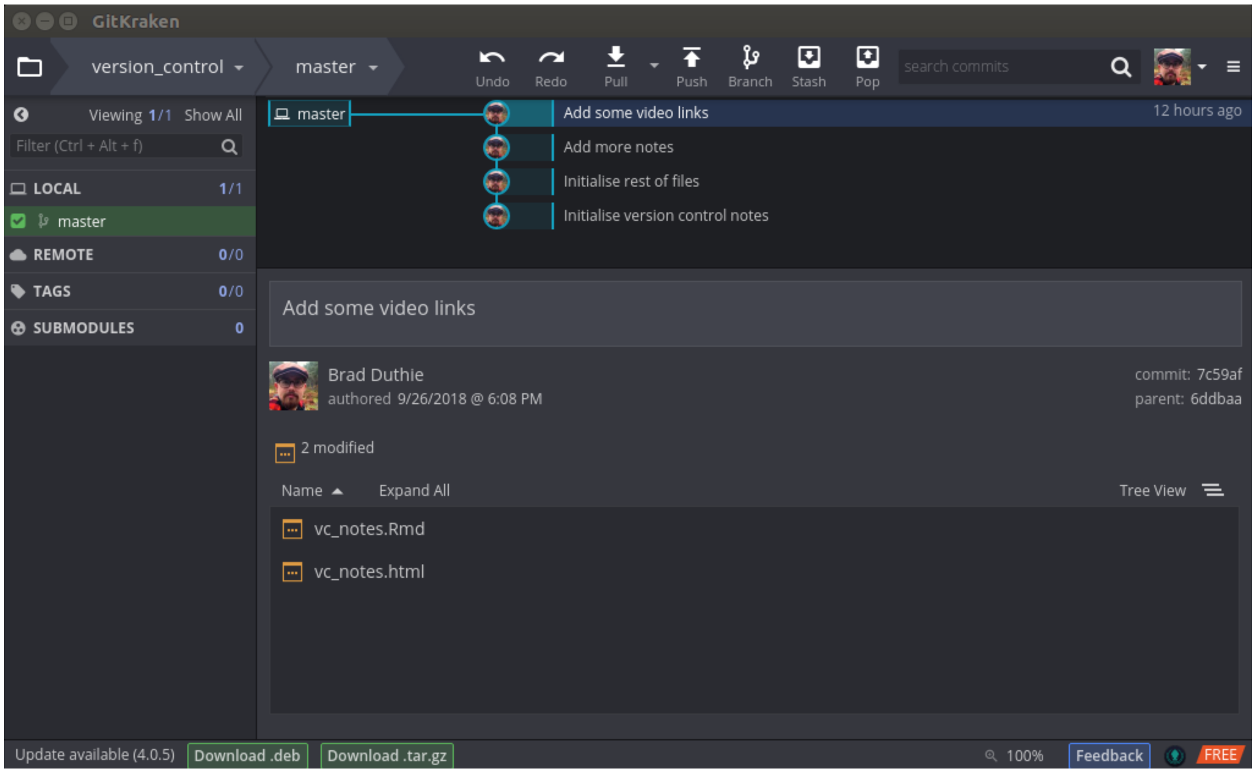
Notice that there are several options in the middle tab. You can initialise a git repository that is 'Local Only', which means that the history of the project will only be saved on your own computer, and not hosted online (at least not right away). Alternatively, you can choose from multiple online hosts where you will later 'push' your files. Since you should have already signed up with a GitHub account, choose the tab 'GitHub.com' as selected above. You should see your account listed in the right tab; if it's not there, then you need to link GitKraken to your GitHub account before continuing. You can also choose the name of the repository (avoid using spaces in the name) and provide a brief description, and choose if you prefer to have the repository public or private on GitHub. Check 'Clone after init' as indicated above, and locate where you decided that your repository will be stored by typing or finding (using 'Browse') the location on your computer. Click the green 'Create Repository and Clone'.

If you open a file browser on your computer (e.g., 'File Explorer' on Windows, 'Finder' on Mac), you should now see your repository in the location that you created it with GitKraken. If you look inside, then you will see a file called 'README.md', which GitKraken has created to get you started. Your repository should now be ready to go and linked to your GitHub account (go to GitHub to see it).

The image below shows the version_control repository during an early stage of development (but several commits after initialisation). Each of the four cyan circles shows an individual commit with a message about what the commit is doing. You can think of each commit as a separate 'Save As' of one or more files, or as a snapshot of what the 'version_control' folder looked like at a particular time. At any time, I can go back to see what the directory looked like in a previous commit by right clicking one of these circles and selecting 'Checkout this commit'.

Note that the bottom area of the GitKraken interface shows the changes made during the last commit.

> **Exercise:** *Find the repository that you just initialised in GitKraken on a file browser on your computer. Look inside the repository and open the 'README.md' file. Make a change to this file, then save. Then, shift back to GitKraken; what has changed? Lastly, copy a (small) file from*



master — Add some video links — 12 hours ago
Add more notes
Initialise rest of files
Initialise version control notes

Add some video links

Brad Duthie
authored 9/26/2018 @ 6:08 PM
commit: 7c59af
parent: 6ddbaa

2 modified

vc_notes.Rmd
vc_notes.html

Figure 2: *The GitKraken interface*

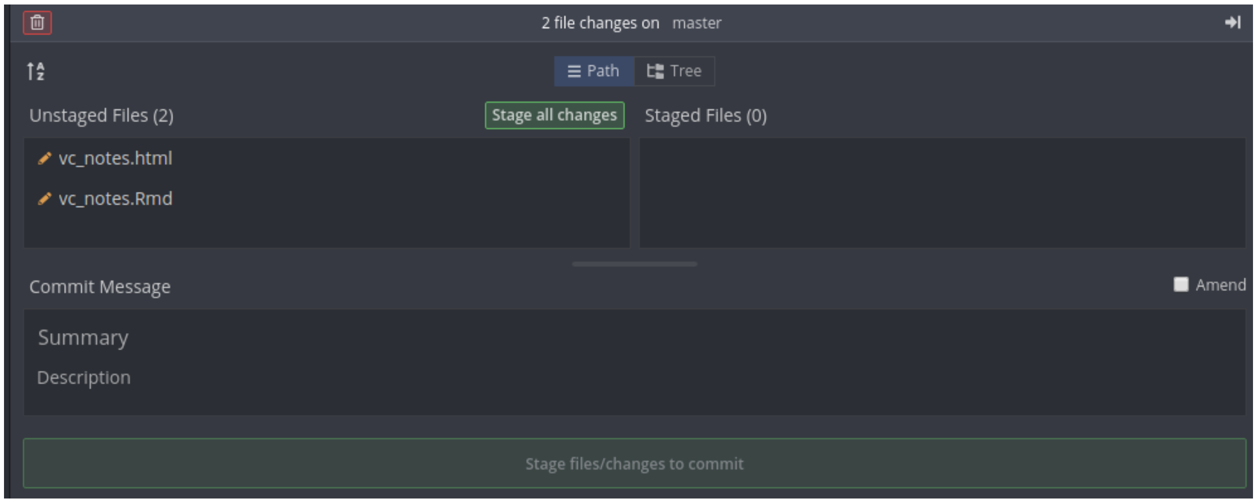
> *somewhere else on your computer and paste it into the new repository. Shift back to GitKraken again; what has changed now?*

Since you have linked GitKraken to your GitHub account, you should be able to see the newly created repository on GitHub.

> **Exercise:** *Open a browser and navigate to GitHub. Find your new repository. Explore the features associated with your repository on GitHub (e.g., Issues, Projects, Wiki), and think about how these might or might not be incorporated into your workflow.*

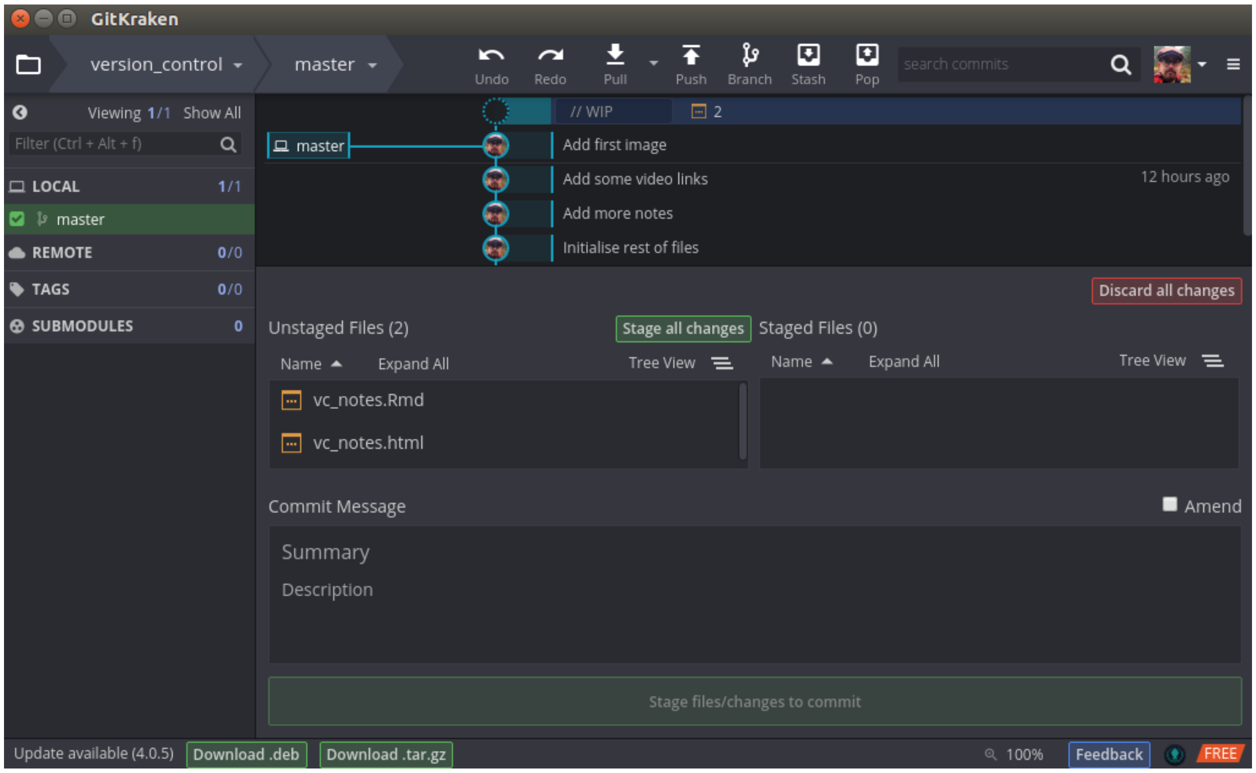
2. Add and commit files

The importance of saving and backing up saved files is obvious to anyone who has ever had a program crash on them. Version control does not replace saving files, but is more like an additional layer of security on top of saving to avoid losing your important work. After a file is saved, it can then be 'staged' in git, which means that it is ready to be committed to the history of the repository. GitKraken will automatically recognise when files in your repository have been saved and list them as unstaged.



In the above, ‘vc_notes.html’ and ‘vc_notes.Rmd’ have been saved outside of GitKraken. GitKraken recognises this and lists them as being available to stage. After a file staged, it can be committed, which creates a reference point for the entire repository to which you can always refer back.

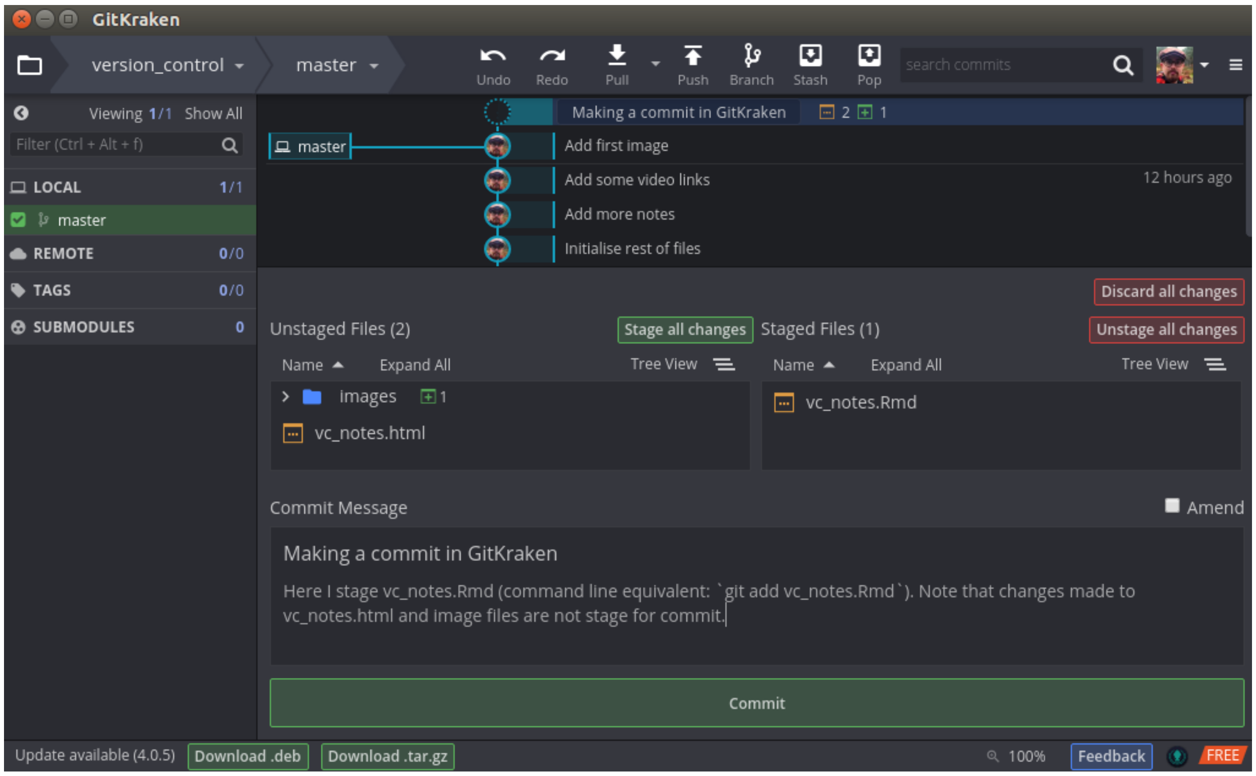
Work flow incorporating git typically proceeds as follows: (1) saving files outside of GitKraken, (2) adding (or ‘staging’) files in GitKraken, (3) committing files in GitKraken, and (4) pushing files to GitHub in GitKraken. Rather than saving the most recent changes that you’ve made to the files in your repository, the idea here is to save a snapshot of the whole repository using ‘commit’. You can then always return to this commit to view what your repository looked like at a previous time.



In the above image, the history of commits is shown with in the cyan circles, along with a commit message written to explain the changes made in each commit (by convention, these are typically written in the present tense). So you can see that my most recent commits were to ‘Add first image’, and before that to ‘Add some video links’, etc. You can see what the directory looked like at these points by right clicking on these commits and selecting 'Checkout this commit' (viewing a commit is also possible, and often easier, in GitHub where you can see the history of this repository). The very top shows WIP (work in progress), which is the directory as it looks at the moment.

As you work on your files, continue to save as you normally would; you always need to save before files can be added or committed. In the above, you can see the files 'vc_notes.Rmd' and 'vc_notes.html' (yellow boxes). These have been saved recently, which is why they are listed as 'Unstaged files' (shown with yellow icons).

Now assume that we want to commit the changes we have saved, establishing a snapshot of the directory that we can return to in the future. The common wisdom in using version control is **"commit early, commit often"**, meaning that it is generally better to err on the side of commiting changes that you make to your repository more often than you think is necessary. In practice, I try to commit frequently, almost as often as I save file changes; even small changes can be useful to have their own commit because the changes are easier to read in small chunks rather than massive changes to files. To commit our changes, we need to 'stage' the files. You can stage all of the changed files at once, if desired, or you can stage only a subset of them (if, e.g., you wanted to record a change in one file, but wait to commit changes in another file). In GitKraken, staging is done simply by hovering over the file of interest in the 'Unstaged files' field, then clicking on the green 'Stage file' box that appears to the right of it. If you instead want to stage all files, then you can click on the 'Stage all changes' box in green.



Making a commit in GitKraken

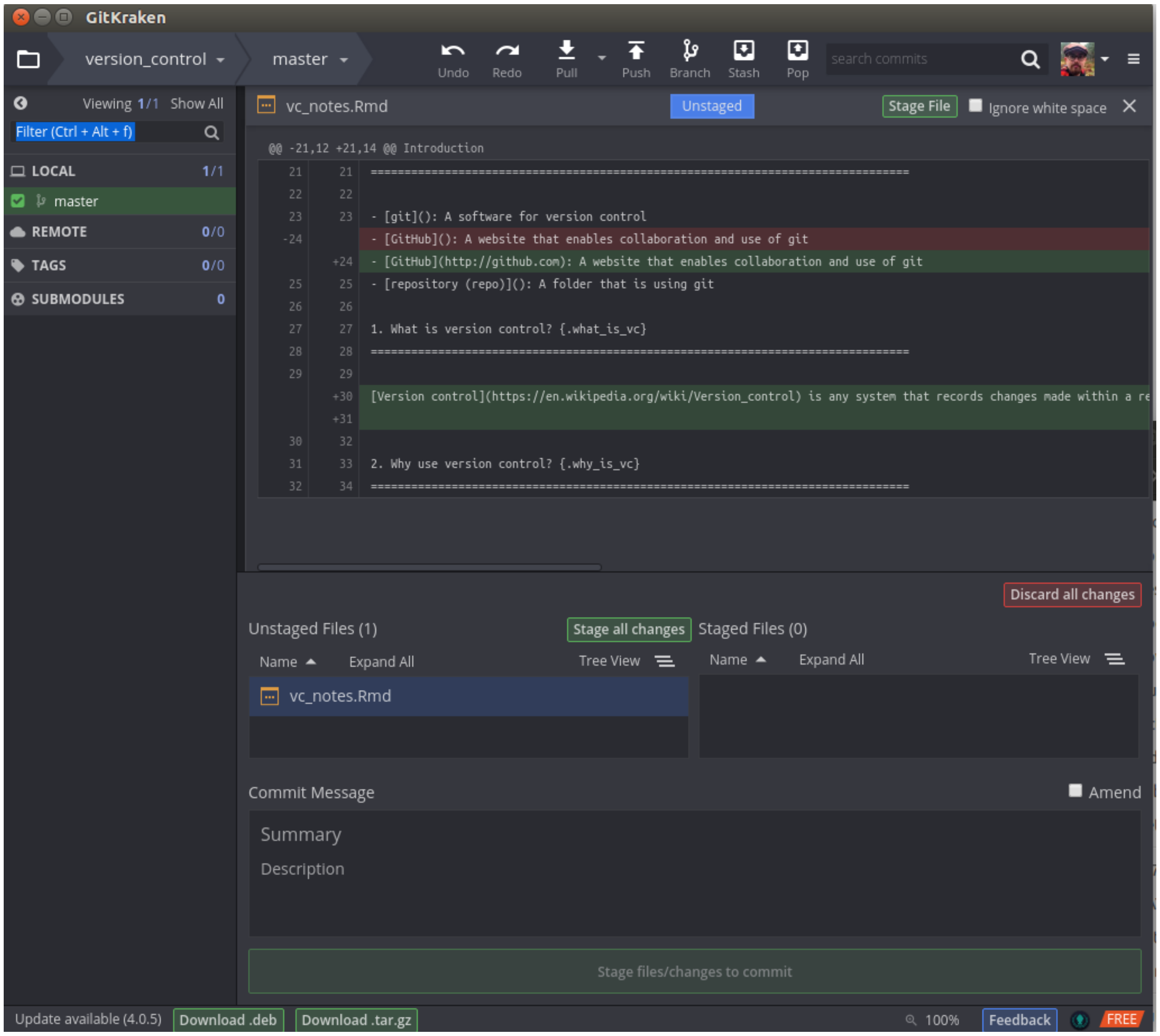
Here I stage vc_notes.Rmd (command line equivalent: `git add vc_notes.Rmd`). Note that changes made to vc_notes.html and image files are not stage for commit.

In the above screenshot, the 'vc_notes.Rmd' file has been staged, while the the changed file 'vc_notes.html' and image files in the subdirectory 'images' have not been staged.

Committing the staged file is easy in GitKraken. Simply add a commit message. In the above example, this message is 'Making a commit in Gitkraken', with further (optional) description in the message below. In general, it is a good idea to make commit messages meaningful, so that you can understand the history of changes in a repository and easily refer back to key points in development (e.g., "Add new linear model analysis" or "Make edits to first paragraph of manuscript"). In practice, it's easy to get lazy with this. To finalise the commit in GitKraken, just click the big green 'Commit' button. You will then see another addition to your commit history in the GitKraken interface (another cyan circle added to the chain).

Note that you can always undo or redo commits (or most other actions) in GitKraken using the buttons on the toolbar.

One last thing with respect to staging and committing files. One very useful feature of git is the ability to see changes that are made to files from one commit to the next. You can do this in GitKraken simply by clicking the staged, unstaged, or already committed file. What you'll see is a record of the changes that have been made since the previous commit.



```
@@ -21,12 +21,14 @@ Introduction
 21   21  ==============================================================================
 22   22
 23   23  - [git](): A software for version control
-24        - [GitHub](): A website that enables collaboration and use of git
     +24  - [GitHub](http://github.com): A website that enables collaboration and use of git
 25   25  - [repository (repo)](): A folder that is using git
 26   26
 27   27  1. What is version control? {.what_is_vc}
 28   28  ==============================================================================
 29   29
     +30  [Version control](https://en.wikipedia.org/wiki/Version_control) is any system that records changes made within a re
     +31
 30   32
 31   33  2. Why use version control? {.why_is_vc}
 32   34  ==============================================================================
```

In the above, the red refers to lines that have been removed, while the green shows lines that have been added. Once everything has been pushed to GitHub (see below), we can also see the record of these changes in the exact same way on the GitHub repository. To interpret the above, we can see that the GitHub URL has been added to line 24, and a new line 30 talking about 'Version control' has been added. **Note that you can only see changes to text files in this way** (e.g., R, Rmd, TXT, CSV, HTML, MKD). If we try to look at the changes to non-text files (e.g., any image files, DOCX, DOC, PDF), it won't work (i.e., if you can open a file and read it in Rstudio, notepad, or some other text editor, then you'll be able to see changes in git; if you can't open it in one of these programs, you can still stage and commit the files in git, but you won't be able to see changes).

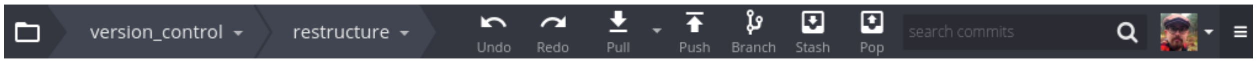
You are now able to stage and commit files. This is most of what you'll do with version control, and all that you need to get started. Next, I'll show you how to push your commits to GitHub, and pull changes from

GitHub to your local branch. Once you're able to do this, you will have all of the necessary skills for using git and GitHub effectively in your own projects; after that, I'll explain some of the features that make git such an effective tool for collaboration.

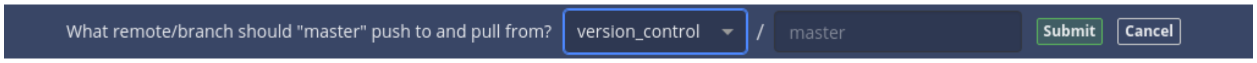
> **Exercise:** *Make some changes to the README.md file, save the changes, then stage and commit your changes. Continue to do this for a while, getting used to the process saving, staging, and committing. If you want to, add other files to your repository (but avoid adding especially large files, over 100 MB, for now).*

3. Pushing and pulling with GitKraken

Assuming that you have linked your local repository with GitHub, uploading the changes that you commit to GitHub (pushing), and downloading changes on GitHub and incorporating them into your local repository (pulling) is easy with GitKraken. You should see a top bar in GitKraken that looks like the below.



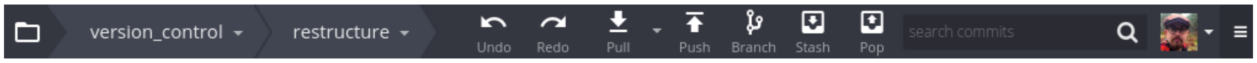
To push all of your commits to your GitHub account online, simply hit the 'Push' button. You might be prompted to specify where to push (if not, skip down to the next **Exercise**).


What remote/branch should "master" push to and pull from? version_control / master

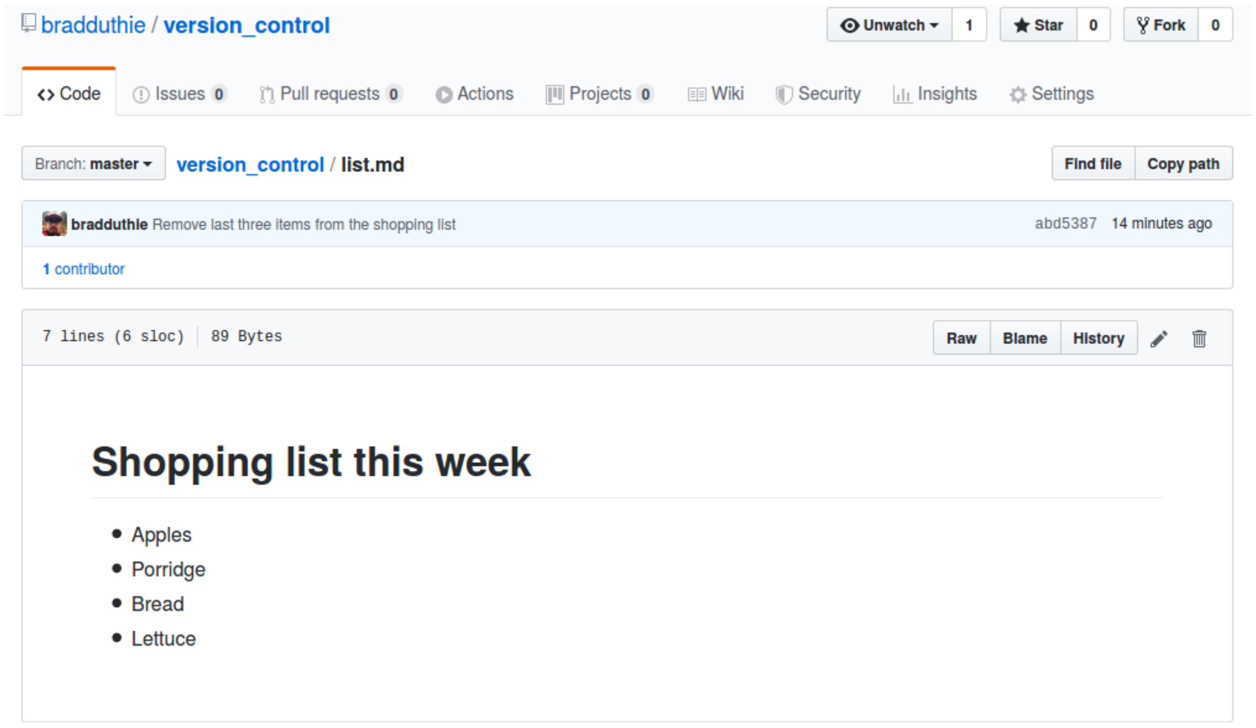
If so, make sure that the choice from the pull down (in this case, 'version_control') matches the repository name on GitHub (see it here), and keep the branch 'master' for now. Click on the green 'Submit' button, and you should succcessfully have pushed to GitHub. To see the changes made, you can go to the commit history in your GitHub repository.

> **Exercise:** *You now know how to push changes to GitHub using GitKraken. Make some more commits to the file(s) in your repository, then push them to GitHub to see what happens. Explore the repository and the commit history on GitHub.*

Pulling changes from GitHub works in roughly the same way. To pull your changes in GitKraken, just click on the 'Pull' icon. Your commits from GitHub should be incorporated automatically, assuming that there is no conflict in merging (more on that later).



This is a good time to mention that you can make changes to text files directly from GitHub, if you wish. To do this, you can open a file within GitHub by clicking on it. Below, I have opened up the file list.md, which is a shopping list written in markdown.


bradduthie / version_control
Unwatch 1 Star 0 Fork 0
Code Issues 0 Pull requests 0 Actions Projects 0 Wiki Security Insights Settings
Branch: master
Find file Copy path
Raw Blame History

version_control / list.md

bradduthie Remove last three items from the shopping list abd5387 14 minutes ago

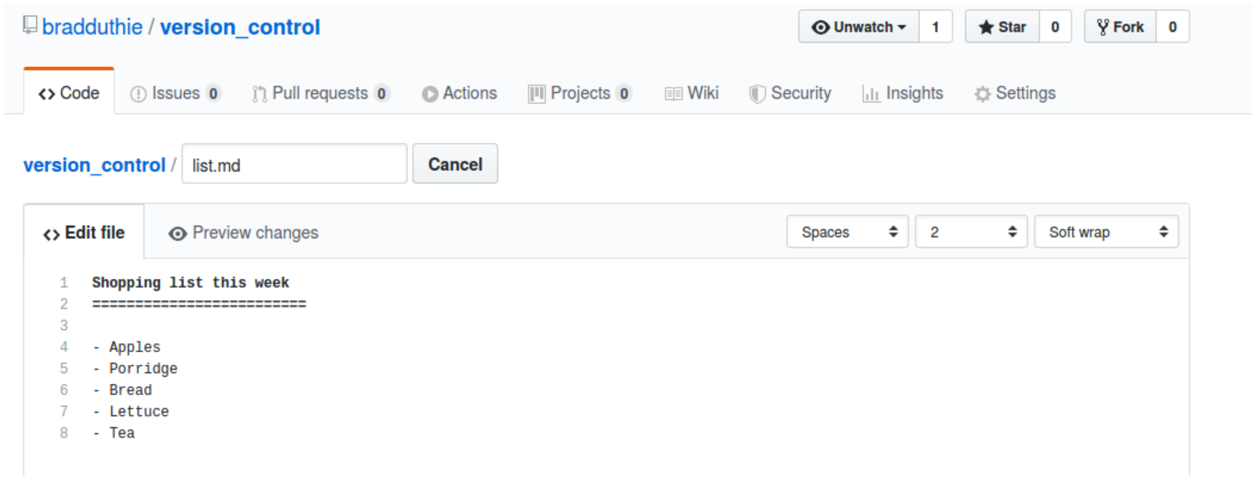
1 contributor

7 lines (6 sloc) | 89 Bytes

Shopping list this week

- Apples
- Porridge
- Bread
- Lettuce

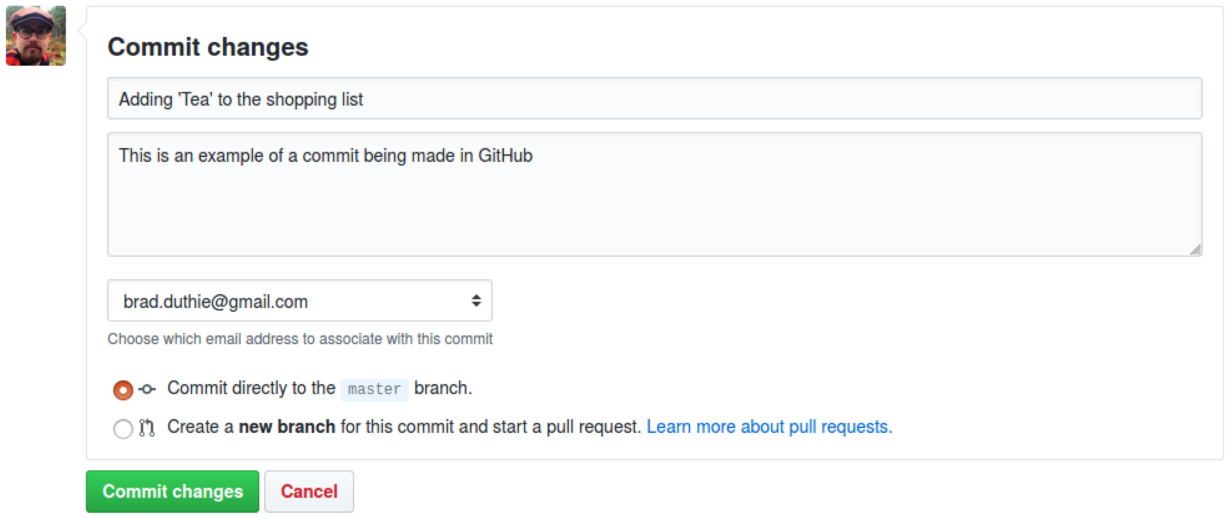
To edit this file, you can click on the pencil icon above the body of the file (between the box 'History' and the bin icon). This will take you to a text editor within GitHub.


bradduthie / version_control
Unwatch 1 Star 0 Fork 0
Code Issues 0 Pull requests 0 Actions Projects 0 Wiki Security Insights Settings
Cancel
Edit file
Preview changes
Spaces
2
Soft wrap

version_control / list.md

```
Shopping list this week
=========================

- Apples
- Porridge
- Bread
- Lettuce
- Tea
```

I have edited the file to now include 'Tea' to the shopping list. There is no need to save or stage the file when editing directly from GitHub; we can just commit it directly to the master branch. The option to commit is at the bottom of the page.


Commit changes

Adding 'Tea' to the shopping list

This is an example of a commit being made in GitHub

brad.duthie@gmail.com

Choose which email address to associate with this commit

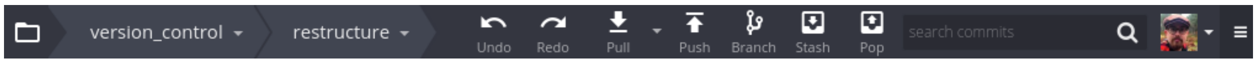
[✓ Commit directly to the `master` branch.]

Create a **new branch** for this commit and start a pull request. Learn more about pull requests.

Remember to add a meaningful message for the commit change. Above I have committed directly to the master branch. It is also possible to create a new branch for the commit – we will look at branches in the next section. In the mean time, try editing your README.md file directly in GitHub (note – make sure that you do not have any unpushed changes to README.md on your local computer).

Exercise: *Edit your README.md file directly from GitHub, then commit the change.*

When you are finished committing the change that you made to README.md on GitHub, go back to GitKraken. The 'Pull' button on your tool bar will take the updated commit history from GitHub and incorporate it back into GitKraken, and the updated README.md should automatically show up on your computer when you open it.



Exercise: *Pull the changes you made to README.md on GitHub to GitKraken.*

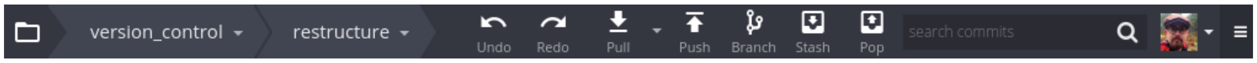
In practice, you will not be making changes directly from GitHub very often (I don't believe I have ever done it, except for demonstration purposes). But even if you are working by yourself on projects, you still might need to pull from GitHub. If, for example, you work on more than one computer (e.g., a Desktop in the office and laptop during travel), then it is important to know how to push from one computer to GitHub so that changes to your repository can be pulled from GitHub to another computer.

You now have all of the tools that you need to save changes to a project repository, commit them to git, and push to and pull from GitHub. This is all that you need for working independently to safely back up and record all of the changes in a project that you are working on. A good work flow is to save the files that you work on frequently, and keep GitKraken open in the background. Then, whenever you feel like you have made some important changes to your files (e.g., a completed paragraph, some new code that works), commit the changes and push to GitHub. **Pull from GitHub every time you start working on the same project from a different computer.** In addition to backing up the history if all of your project development, it is also nice to have a record of your progress over time. If you have any difficulty with these instructions, you can submit a question to the 'version_control' issues. I will read through these and try to help you troubleshoot. Next, I will explain how to use branching in git.

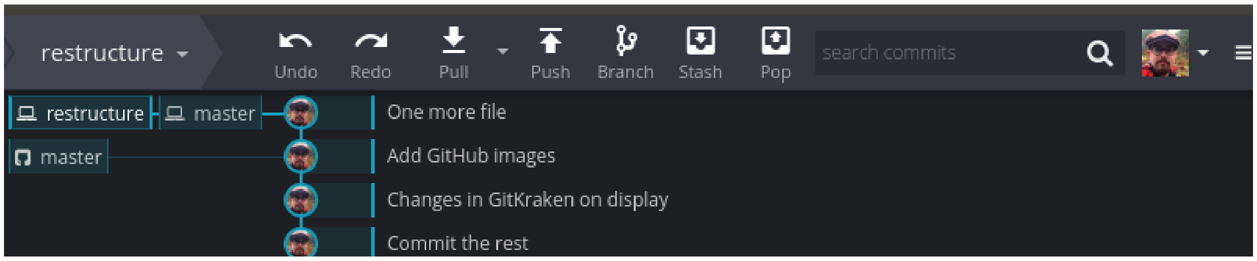
4. Branches in git

Once you have a clear understanding of how commits work, you can start thinking in terms of different branches of commit histories. Up until now, we have looked at the history of commits on GitHub and

GitKraken as a single linear timeline. With branching, we can effectively create alternative timelines of commits comprised of different snapshots existing in parallel. It is then possible to quickly switch between different versions of the same repository. For example, you might want to have a branch that contains code that has been tested and demonstrated to work, but also a branch in which the same code is being further developed (and is therefore untested). For large projects, I will typically reserve the 'master' branch as a sort of stable version in which I am confident that everything works as intended, and do all new work on separate branches, only merging to the master branch when I am satisfied that what I have done is sufficient. Then, if I am unsatisfied, I can just abandon my working branch, or delete it. In GitKraken, branching is easy. Just click the 'Branch' button on the top tool bar.



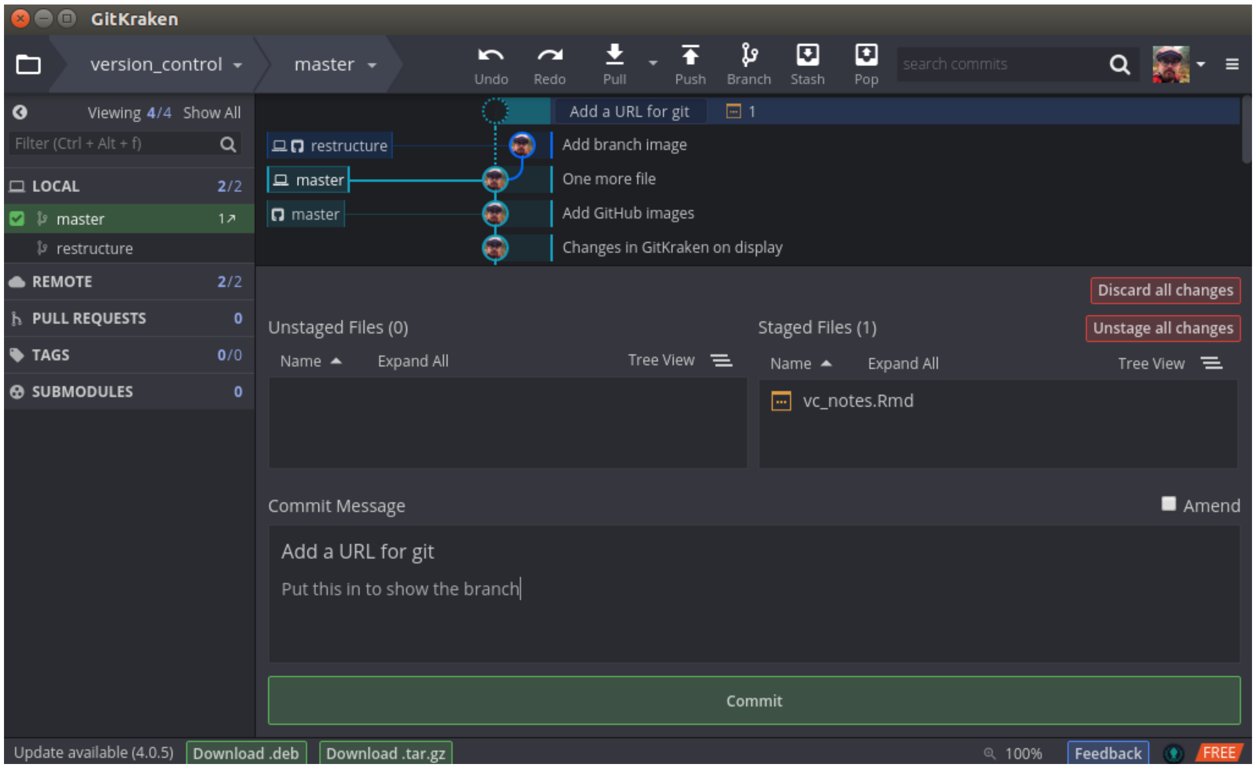
An empty box will then appear in your commit history, and GitKraken will prompt you to write a new name for a branch on your timeline. In the case of the below, I have titled the new branch 'restructure'.



One more file
Add GitHub images
Changes in GitKraken on display
Commit the rest

The new branch 'restructure' starts off in the exact same place as the old 'master' branch did. The files have not changed at all – branching simply tells git to keep everything in the 'master' branch as is, but also start a new branch ('restructure', in this case) to which new commits are recorded (but do not affect the master branch). Hence, any new commits that we make will start a whole new path of commit history while keeping the master branch fixed.

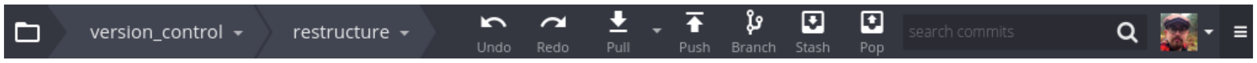
> **Exercise:** *In GitKraken, create a new branch in your repository. Then, make some more changes to README.md and commit. Notice that the new commits are only to the new branch, while the master branch remains intact.*



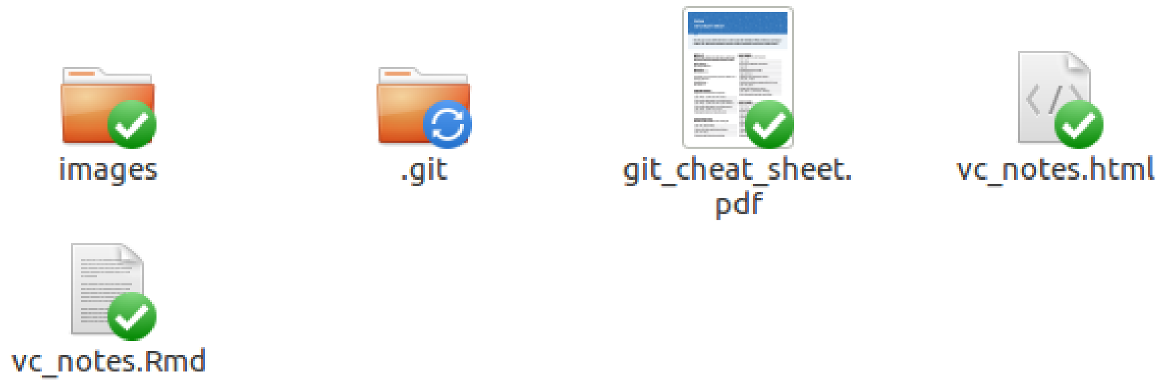
Add a URL for git

Put this in to show the branch

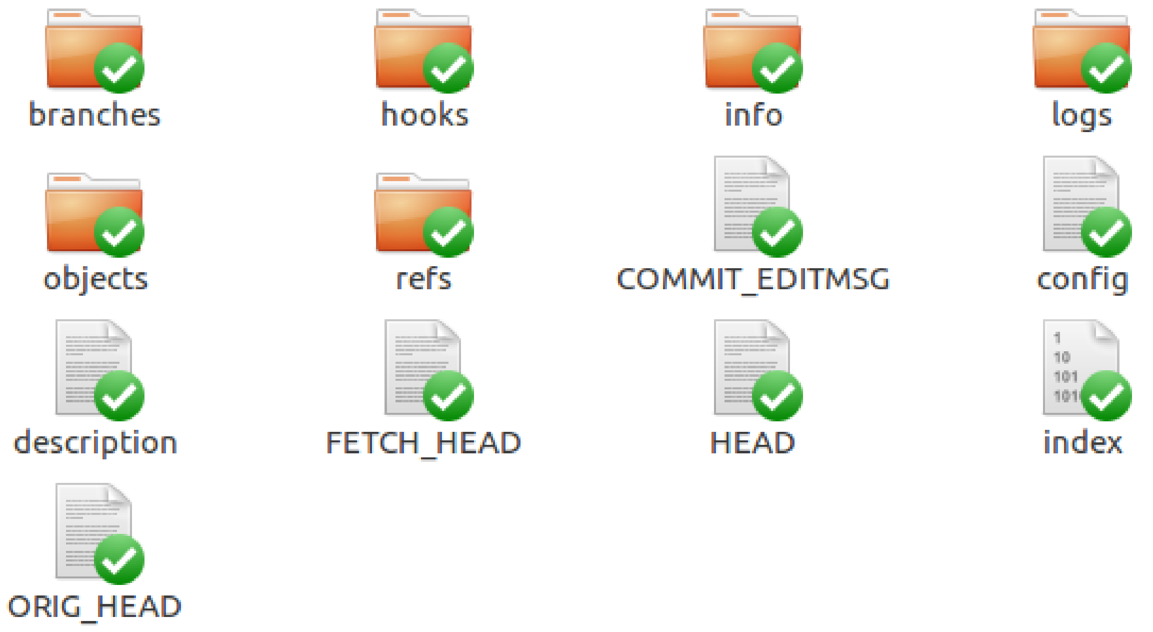
Switching between branches is easy. The toolbar below shows that we are on the branch "restructure". To switch to a different branch, simply click on the pulldown and select the one that you want. Before you do this, **make sure that you do not have any uncommitted changes on the branch from which you are changing** (GitKraken will warn you if you try).



Note that when you switch between branches, your repository will automatically revert back to what it looked like at this point in time. **Don't panic if things change**; if you have added files since starting a new branch, they won't be here when you go back to master, but still exist in the new branch. The way this works behind the scenes is that git stores a hidden folder in your git repository called `.git` (I've told my operating system to show hidden files, so you can see this below).



Within this hidden folder, all of the history and branches are stored (**don't delete this**).

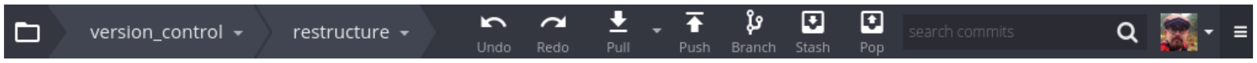


Back in GitKraken, you can shift between branches 'master' and 'restructure' as you wish by clicking on them.

> **Exercise:** *In a text editor, make some more changes to README.md on your new branch, then commit them. Keep README.md open, but go back to GitKraken and checkout the master branch. Look at README.md again to see what has changed, then checkout your new branch again.*

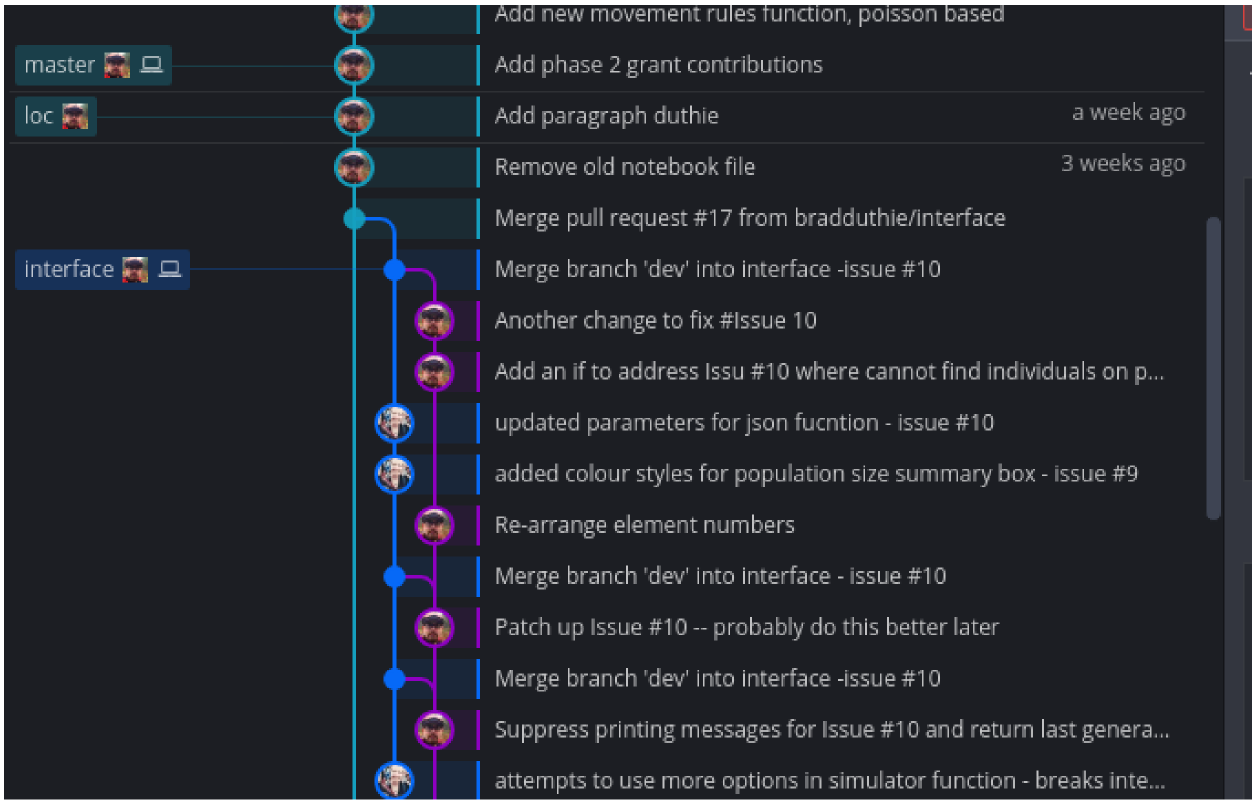
For now, I will only make changes to 'restructure', using it as a place to experiment with new changes to the repository. I'll leave 'master' as it is, but it is entirely possible to have multiple branches with changes being made independently on each. For large projects (e.g., an R package with code, documentation, and a manuscript), I'll typically have a master branch of publication quality (`master`), a development branch where everything appears to be working (`dev`), a branch where revisions are taking place (`rev`), and several temporary branches for experimenting with new writing or code.

After adding some commits to the branch 'restructure', I might be satisfied with the changes to the repository and therefore want to merge the 'restructure' branch back into the 'master' branch. Merging means incorporating the changes made in one branch into another branch; this works seamlessly when the branch being merged into has not changed (i.e., if after creating and committing on the new branch 'restructure', the original 'master' branch remains untouched). Merging gets slightly more tricky if commits have been made to both branches since the time of branching, but more on this in a moment. To merge in GitKraken, make sure that you are working on the branch that you want to merge *to* (i.e., the branch where you have *not* made the changes that you are satisfied with, but the one that you want to send them to). This is shown in the top toolbar.



The top toolbar above shows that I am on the branch 'restructure', so I want to click the down arrow on the 'restructure' pull down and switch to 'master'. Next, find the 'restructure' branch in the commit timeline (rectangle box to the left of the cyan cicles), right click, and select 'merge restructure into master'. You should now see that the changes have been moved into 'master', and the branches 'master' and 'restructure' are now identical; you can continue to make commits on either branch, or, if there is no longer any need for the 'restructure' branch, it can be deleted by right clicking on it and selecting 'delete restructure' (note, there is nothing special about the branch named 'master'; you could equally well use it as a working branch and merge changes that you like into a branch named 'stable' or 'final', or something similar).

Branching allows you to quickly start new experimental pathways in a project and make big changes without fear of disrupting anything. When you collaborate, if different people are working on different aspects of a project, each can work on a separate branch to be merged later. You can view a tree of branch history on GitHub in the 'insights' then 'networks' tab in a repository (e.g., in the vegan R package). If you're working with multiple collaborators who are all pulling from and pushing to GitHub, making commits, and creating and merging branches (or if you're doing all of these things yourself with multiple branches), things can start to look messy (for another example, see the Rmarkdown repository in Rstudio). In GitKraken, this is what it can look like with only two people and three branches.



Add new movement rules function, poisson based
Add phase 2 grant contributions
Add paragraph duthie
Remove old notebook file
Merge pull request #17 from braddthie/interface
Merge branch 'dev' into interface -issue #10
Another change to fix #Issue 10
Add an if to address Issu #10 where cannot find individuals on p...
updated parameters for json fucntion - issue #10
added colour styles for population size summary box - issue #9
Re-arrange element numbers
Merge branch 'dev' into interface - issue #10
Patch up Issue #10 -- probably do this better later
Merge branch 'dev' into interface -issue #10
Suppress printing messages for Issue #10 and return last genera...
attempts to use more options in simulator function - breaks inte...

Sometimes, merging between branches can create minor (but never disastrous!) issues that need to be resolved, which I will explain in the next step on dealing with merge conflicts (see below).

5. Dealing with merge conflicts

Merge conflicts occur when there are differences between two files that git cannot resolve on its own during a merge. This tends to happen when you (or you and your collaborators) have been working on two different branches separately, and changes have been made to the same file that are in conflict. If there is any doubt, then git will not asume that changes from one branch take priority over another. Hence, where merge conflicts occur, you will need to inspect the cause of conflict and decide how to resolve it by choosing what the merged file contents should be.

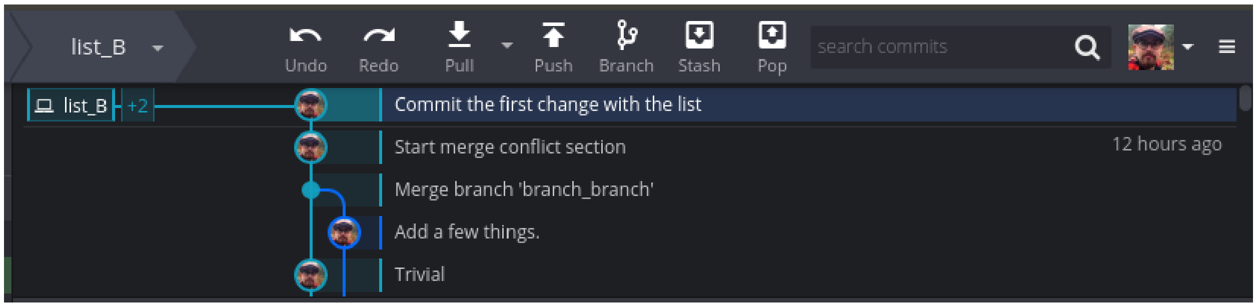
Generally, git is actually very good about merging branches. It can recognise when new files have been added to each branch, and should preserve both of them during the merge; it can even recognise when different lines of the same text file have changed on each branch, and preserve these changes in a merged file. Sometimes, however, different branches will attempt to change the same lines of a file. When this happens, merging one branch into another will generate a merge conflict. The first thing to keep in mind if this happens is that there is no reason to panic. You haven't broken anything, and the worst thing that can happen, realistically, is that you have to either undo the merge attempt (easy in GitKraken) or revert to your most recent commits on each branch. But usually what happens instead is that git will helpfully show you the cause of the conflict in the text file (by changing the text file to show you the conflict – i.e., it writes what each branch wants the line to read as, so you can compare and decide). You will then be able to figure out what you want the change to actually be. Cleverly, the way to deal with the merge conflict is therefore to change the text in the conflict file and *then* commit the change you've made (i.e., 'fixing' the merge conflict is resolved with a new commit, which then becomes part of your commit history just like any other commit).

I'll intentionally generate a merge conflict so you can see how this is dealt with in git in GitKraken and the command line. First, I'll initialise a new file called `list.md` (this is an example of a markdown file, the same kind used for GitHub README files), which will have the text below (see the file here).

```
Shopping list this week
=========================

- Apples
- Porridge
- Bread
- Lettuce
- Tomatoes
```

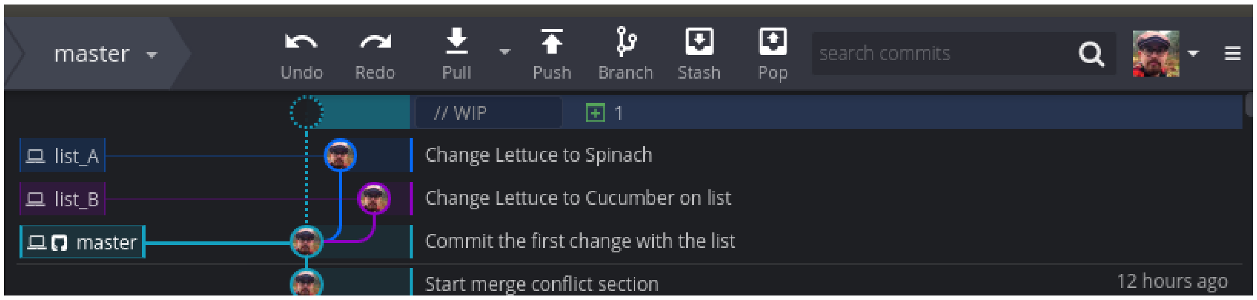
> **Exercise** *Create your own text file (e.g., .txt, .md, .mkd) and your own list as above. Save it to your repository and commit the initalised file.*

I will now make two new branches named 'list_A' and 'list_B'. After creating these branches, each will be identical to the original branch in 'master', but I will edit them differently to generate a merge conflict. First I will create the two branches in GitKraken as explained in the above section. This is what it looks like in GitKraken.



Commit the first change with the list
Start merge conflict section 12 hours ago
Merge branch 'branch_branch'
Add a few things.
Trivial

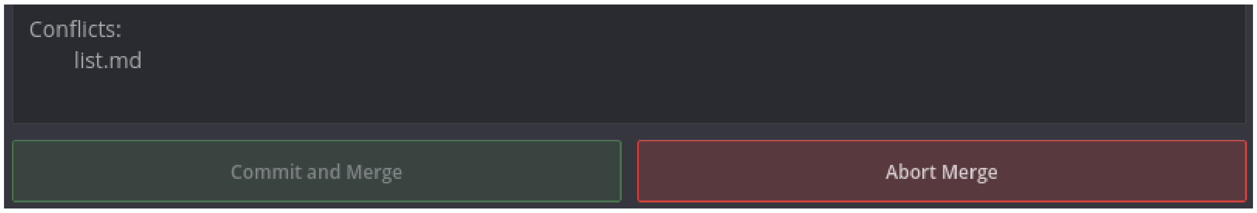
You can see that I am now on the 'list_B' branch from the upper left corner (grey sub-box, not the cyan one), and there are three branches at the very tip of the tree ('list_B', 'list_A', and 'master'). Next, I'm going to make two different changes to the list in branch 'list_A' versus 'list_B'. In the file 'list.md' on branch 'list_A', I will change 'Lettuce' in the list above to 'Spinach', while in the same 'list.md' file on branch 'list_B', I will change 'Lettuce' in the list above to 'Cucumber'. Hence, on two independent branches, I have changed the list in conflicting ways. I then switched back to the master branch to write this paragraph. Here's what GitKraken looks like now.



Change Lettuce to Spinach
Change Lettuce to Cucumber on list
Commit the first change with the list
Start merge conflict section

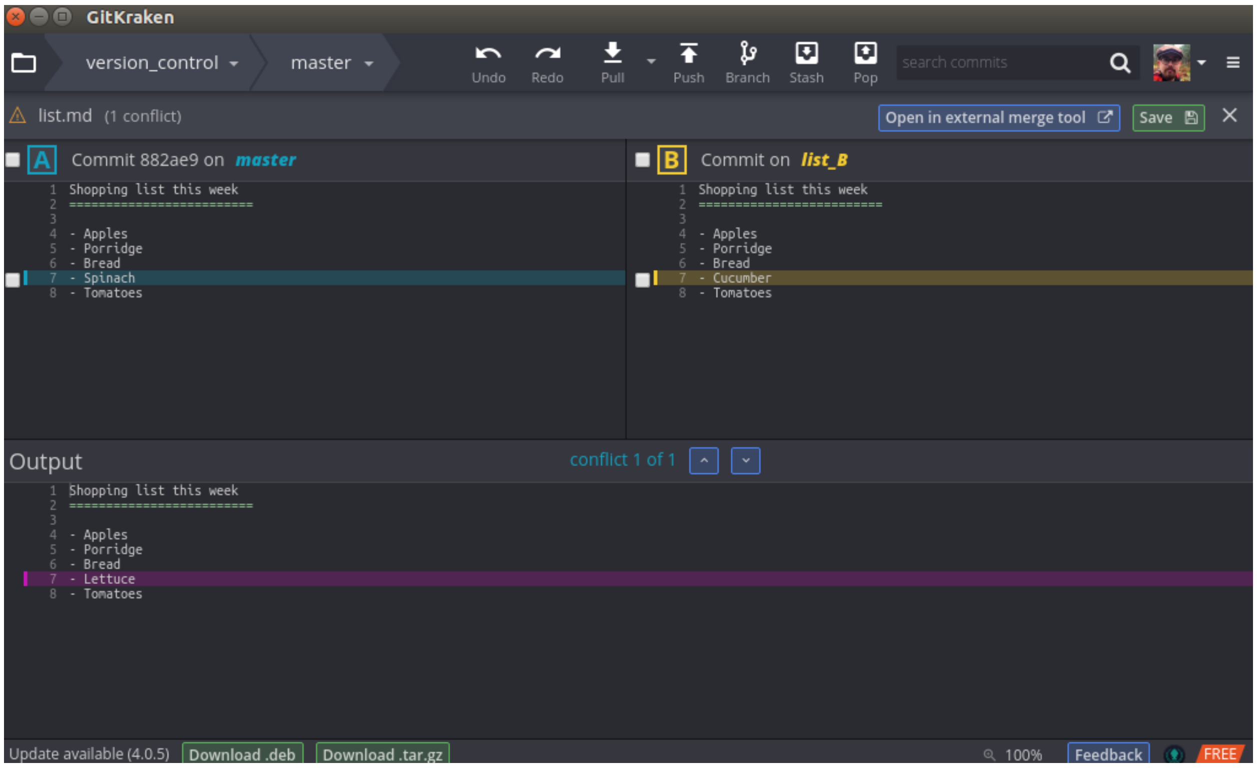
Exercise: *Create your own two branches in GitKraken. Checkout the first branch, make several edits to your list, save, and commit. Then, checkout the second branch, make several edits, save and commit. You now should have two branches with a file that will cause a merge conflict.*

You can see the two different branches coming off of the master branch, each with a commit message that explains the change. I will now attempt to merge list_A, then list_B back into the master branch. Merging the branch list_A produces no conflicts; the branch merges fine as normal. But when I then try to merge branch list_B into master, GitKraken warns me that there is a merge conflict. Again, it's worth emphasising that there is no need to panic here; nothing is going to be lost, and GitKraken immediately presents me with the option to abort the merge (note the big red button below).



Conflicts:
list.md

Aborting the merge will immediately stop it from happening if you realise that it's not what you wanted. Alternatively, however, you can also take a look at what is causing the conflict, then potentially make a decision about how to resolve it within GitKraken. If I click on the triangle with an exclamation mark on it to the left of the conflict file 'list.md', then the conflicting line(s) is shown; in my case, I know in advance this is a conflict because line seven has the word 'Spinach' on the branch 'master' (after 'list_A' merged with master), while line seven has the word 'Cucumber' on the list_B branch that we're attempting to merge into 'master'. This is shown in the GUI below after clicking the yellow triangle.



A Commit 882ae9 on *master*

```
1  Shopping list this week
2  ========================
3
4  - Apples
5  - Porridge
6  - Bread
7  - Spinach
8  - Tomatoes
```

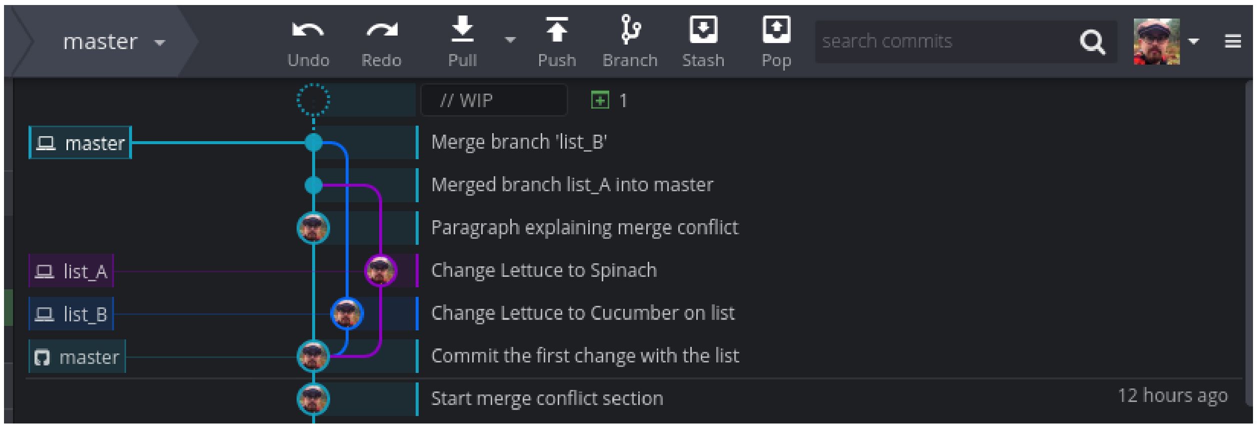
B Commit on *list_B*

```
1  Shopping list this week
2  ========================
3
4  - Apples
5  - Porridge
6  - Bread
7  - Cucumber
8  - Tomatoes
```

Output

```
1  Shopping list this week
2  ========================
3
4  - Apples
5  - Porridge
6  - Bread
7  - Lettuce
8  - Tomatoes
```

The GitKraken environment therefore shows you, side-by-side, the branch you're merging into (left), the branch you're attempting to merge (right), and the resulting output that will happen when you're done (bottom; note that it currently has reverted back to the old commit with 'Lettus' on line 7). If you prefer one line over the other, you can simply click the left or the right (Spinach or Cucumber), and that choice will be what's placed on line 7 when the merge conflict is resolved (GitKraken also explains this in a short YouTube video). The resolution of the merge conflict – i.e., the changes you make to the file to fix the conflict – is *itself* a commit, and is treated just like any other commit that you can come back to later. Here's what that looks like in GitKraken, with the 'list_B' branch successfully merged after the conflict has been resolved.



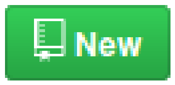
Merge branch 'list_B'
Merged branch list_A into master
Paragraph explaining merge conflict
Change Lettuce to Spinach
Change Lettuce to Cucumber on list
Commit the first change with the list
Start merge conflict section

That's it! Note that merge conflicts might also occur when collaborating with others who are working in the same repository and on the same files. While git is generally good about figuring out where everything should go, if the same line is changed in a different way on a file, a merge conflict will inevitably arise.

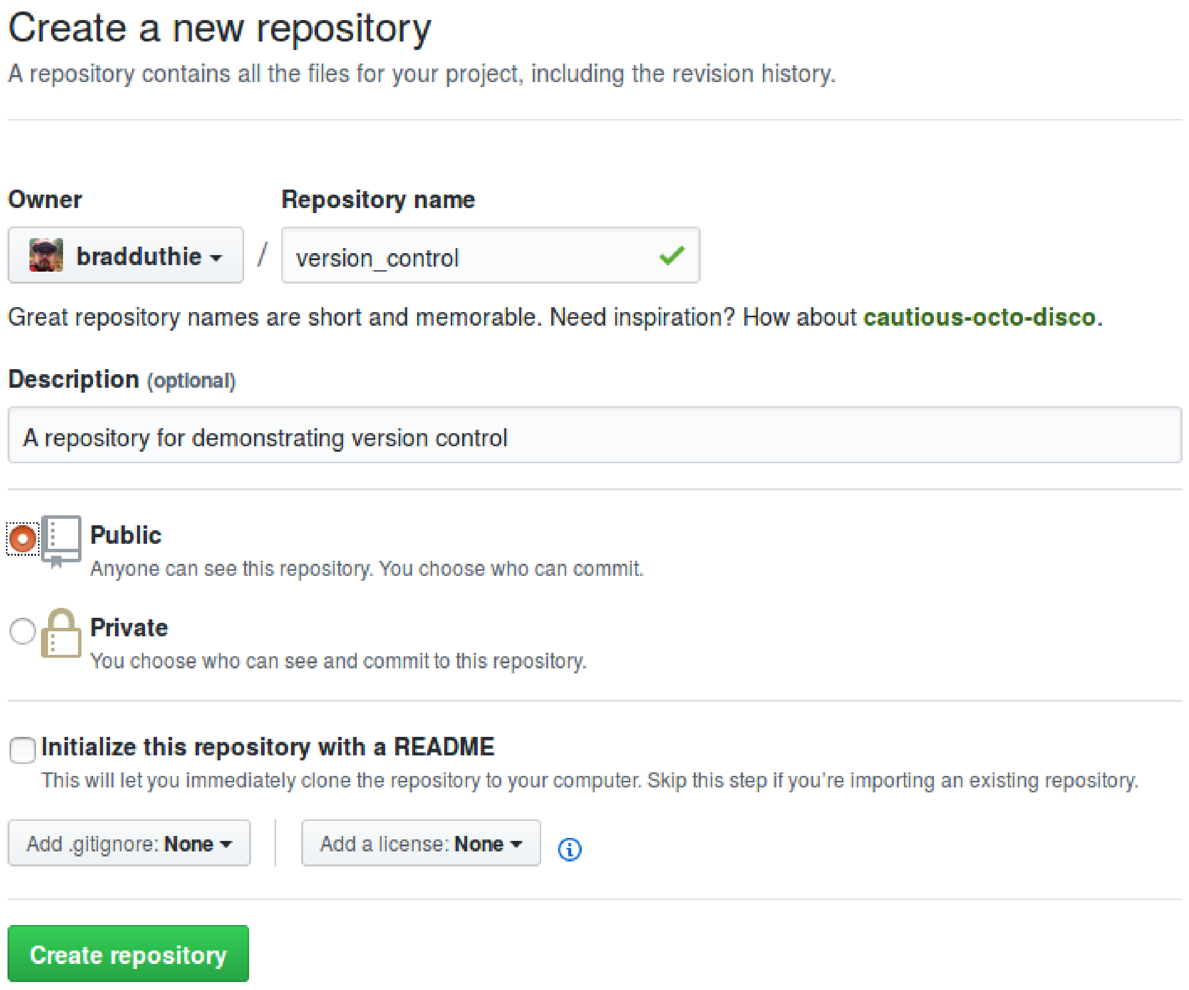
Additional topics

1. Linking an existing repository with GitHub

To link an existing repository to your GitHub account within GitKraken, first go to GitHub and login. Next find and click the button that looks like the one below.



This will take you to the page that allows you to set up a new repository. **Be sure to use the same repository name on GitHub that you do on your local computer**, and avoid the use of spaces.



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: braddtuthie / Repository name: version_control

Great repository names are short and memorable. Need inspiration? How about cautious-octo-disco.

Description (optional): A repository for demonstrating version control

[✓ Public] Anyone can see this repository. You choose who can commit.

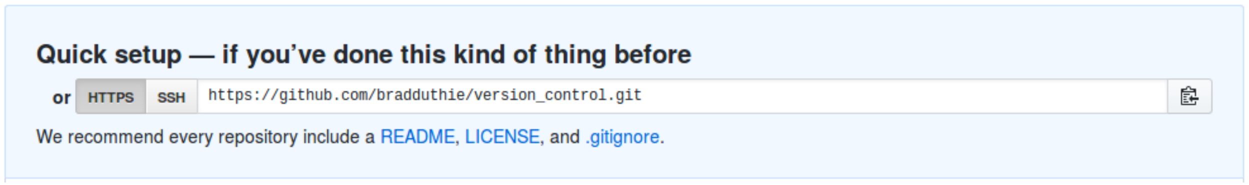
Private: You choose who can see and commit to this repository.

Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None

Add a license: None

After you click the green button 'Create repository', GitHub will take you to a place that gives you a URL for a quick setup. The URL for the version_control repository that I just set up is below.

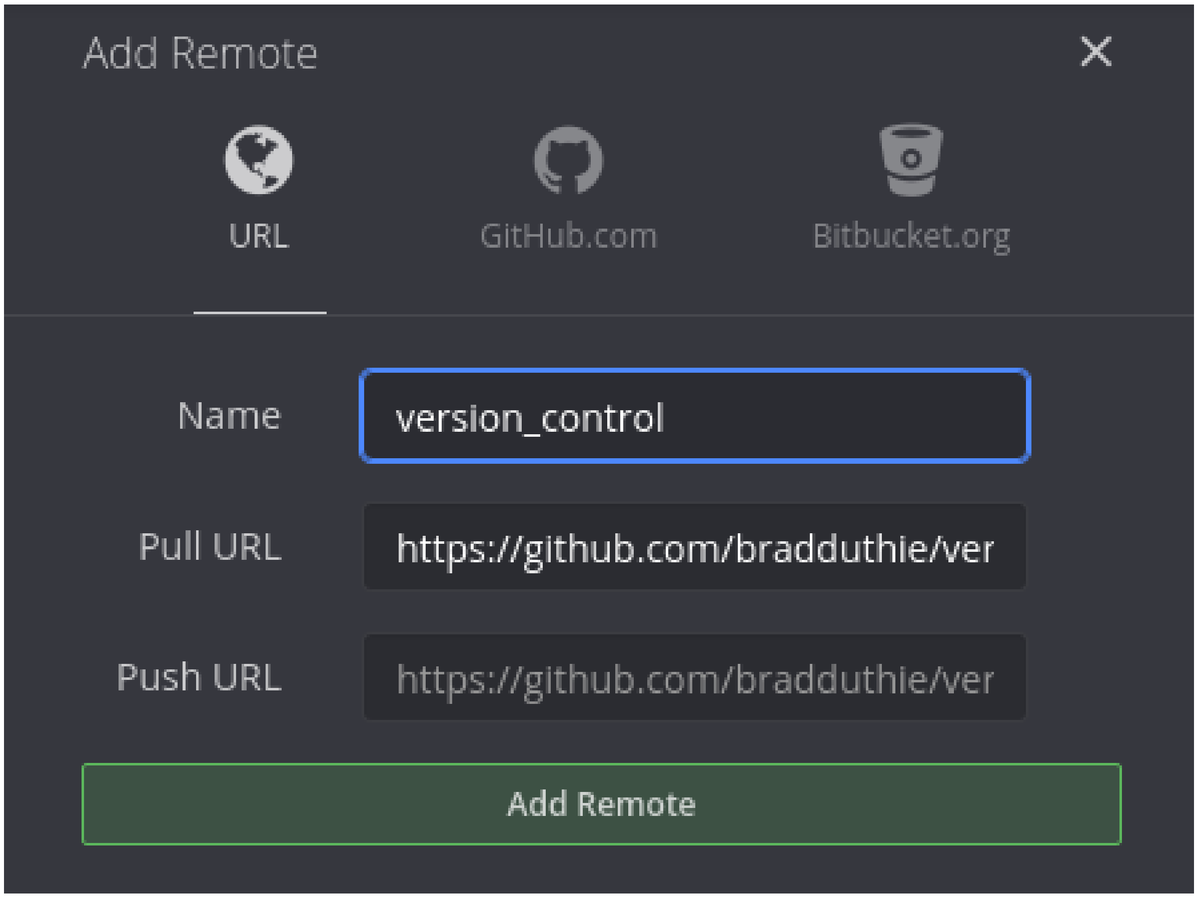


Quick setup — if you've done this kind of thing before

or HTTPS SSH https://github.com/bradduthie/version_control.git

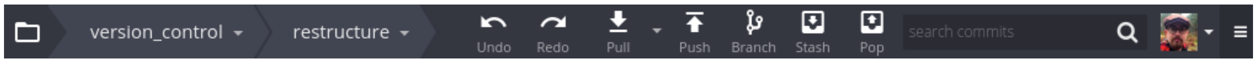
We recommend every repository include a README, LICENSE, and .gitignore.

Copy that URL, then go back to GitKraken. Look at the left-hand side of the interface which has names that include 'Local', 'Remote', 'Pull Requests', etc. Hover over the 'Remote' tab and a green '+' should appear for you to click. This will open a box like the one below with the options 'URL', 'GitHub.com', and 'Bitbucket.org'.

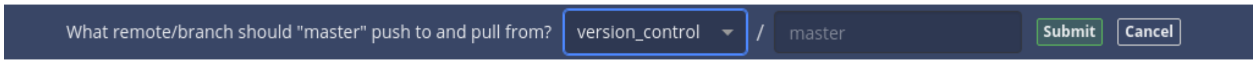


Name: version_control
Pull URL: https://github.com/bradduthie/ver
Push URL: https://github.com/bradduthie/ver

Confusingly, you do not want to select the 'GitHub.com' tab. Select the URL tab instead; type in the repository name (the name on GitHub should be the same as the folder name on your computer), then paste the copied URL above into the 'Pull URL' field. The 'Push URL' field should be identical to the 'Pull URL' field. After you click the green 'Add Remote' button, you should be able to go back to the GitKraken interface and click the 'Push' button.



After you click 'Push', you will be asked what branch to push and pull from. Keep 'master' for now and just click 'Submit'.


What remote/branch should "master" push to and pull from? version_control / master

Now if you go back to GitHub, you should see any files you had in your repository appear.

If you want to link your local repository with GitHub using the command line instead of GitKraken, then after creating a new repository on GitHub, use the command below.

```
brad@duthie-pc:~/Dropbox/projects/StirlingCodingClub/version_control$ git remote add origin https://git
```

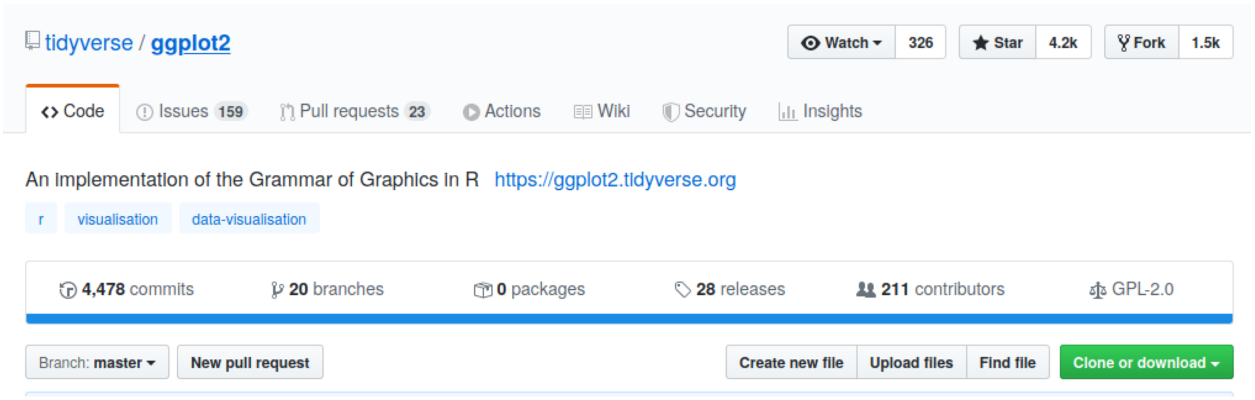
You can then push all of your files to the GitHub repository with the command below.

```
brad@duthie-pc:~/Dropbox/projects/StirlingCodingClub/version_control$ git push -u origin master
```

You should now have a git repository up and running on your local machine, and a linked version on GitHub.

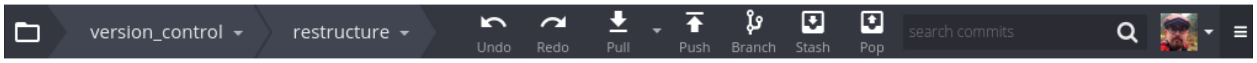
2. Cloning and forking from GitHub

One nice feature of GitHub is that it allows you to make a copy of any public repository and add it to your own GitHub list of repositories. There are two ways to do this; you can fork or clone a repository that you find on GitHub. For example, we might want to have our own copy of the 'ggplot2' repository from Tidyverse.

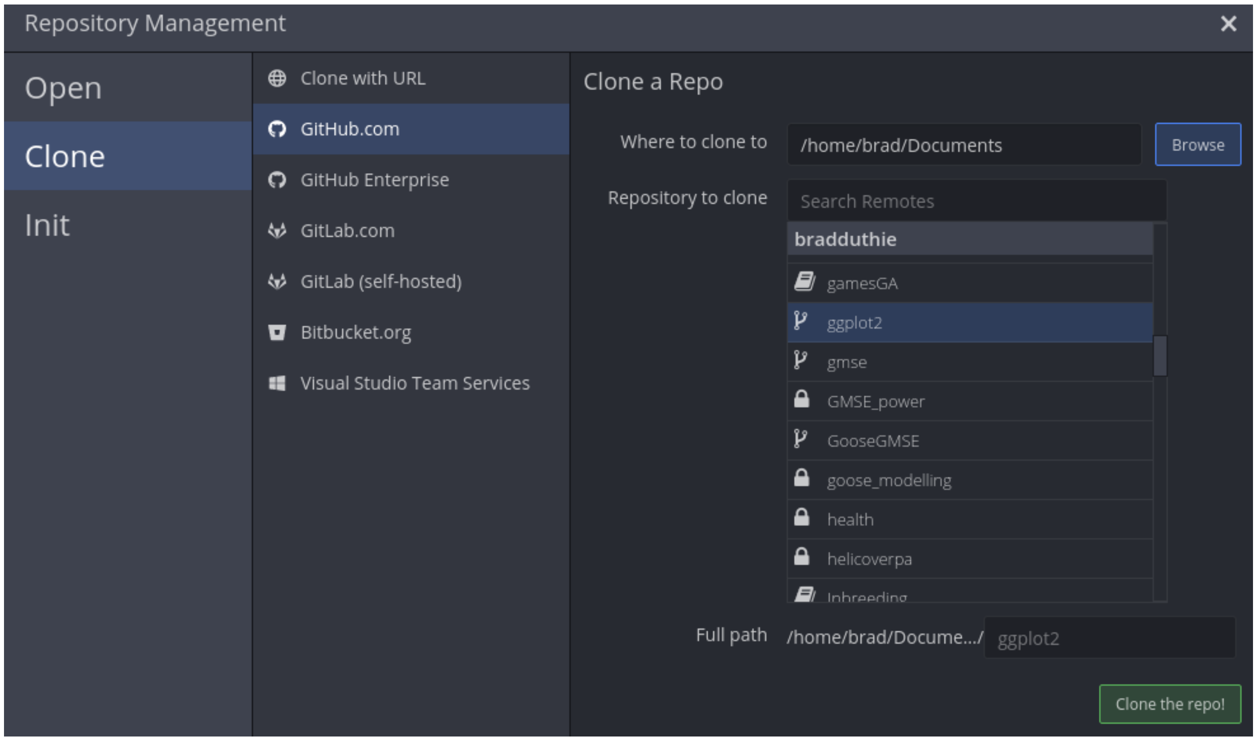


The option to fork the repository is in the upper right corner, while the option 'Clone or download' is shown in the green button on the lower right. Forking and cloning are very similar; both get you the full repository. Both effectively clone someone else's repository to your own GitHub repository list, while still retaining the full commit history. You can make whatever changes you want to the forked repository, but the link to the original version will remain visible, and this can make it easier to suggest changes (i.e., send a 'pull request') to the original author(s).

The repository that you have forked on GitHub will not be available to you on your computer until you also clone it. With GitKraken, it is easy to clone a repository that you have forked. First click on the folder outline in the toolbar, just as you would do when initialising a new repository.



This time, choose the 'Clone' option on the left tab, and 'GitHub.com' in the middle tab. Choose a location on your computer where you want the cloned repository to go, then look through the listed options to the right of 'Repository to Clone'. You should find the repository that you forked in GitHub.



Where to clone to /home/brad/Documents

Repository to clone

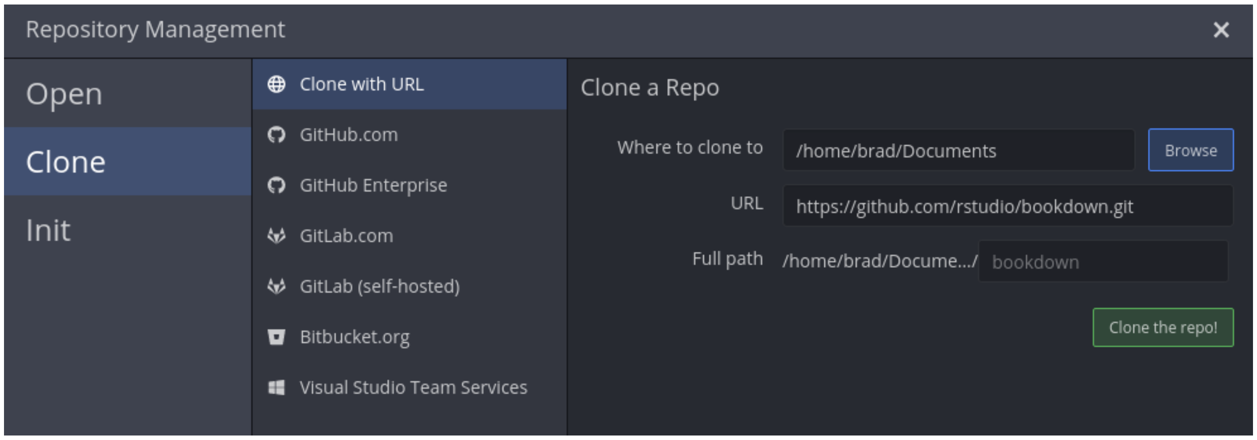
braddhuthie
- gamesGA
- ggplot2
- gmse
- GMSE_power
- GooseGMSE
- goose_modelling
- health
- helicoverpa
- Inbreeding

Full path /home/brad/Docume.../ ggplot2

In the above, the forked ‘ggplot2’ is now available. When you click ‘Clone the repo!’, the whole repository will be downloaded to your computer. If you want to see the whole history of the repository, go to ‘Open’ in GitKraken and find the folder on your computer. You should see the entire project history on GitKraken, and be able to make edits and push them to your forked repository on GitHub.

> **Exercise:** *Find an R package or other repository that interests you on GitHub and fork it. In GitKraken, clone the repository to a location on your own computer. Make some small changes to the repository and commit them in GitKraken, then push the commit to your forked repository on GitHub.*

Cloning a repository directly from GitHub is similar. If, for example, I wanted to clone the bookdown repository, then I could click the green button ‘Clone or download’. I could download the package as a compressed file, but it’s easier to copy the URL and clone through GitKraken.



Where to clone to /home/brad/Documents

URL https://github.com/rstudio/bookdown.git

Full path /home/brad/Docume.../ bookdown

Above, I go back to the Repository Management window, but this time choose to ‘Clone’ and ‘Clone with URL’. On the right, I specify a path on the computer where I want the repository to be located and the

URL taken from GitHub. After clicking ‘Clone the repo!’ GitKraken will clone the entire repository to the specified path on my computer. The downside to this is that you will need to manually connect your repository to GitHub if you want to push and pull. Hence, I recommend forking repositories.

3. Ignoring some git files

Sometimes you might want to have files in your repository that are not tracked with git. For example, if you have one or more especially large files (e.g., over 100 MB), or files containing sensitive information, you might not want to push these files to GitHub. To force git to ignore some files, you need to create a text file called ‘.gitignore’ and save it within your git repository (not inside a subfolder within the repository). The best way to do this is to open a text editor (e.g., NotePad or Textedit, but Rstudio should also work) and type each file that you want git to ignore on its own line. Below is an example of a .gitignore file.

```
.Rproj.user
.Rhistory
.RData
.Ruserdata
src/*.o
src/*.so
src/*.dll
inst/doc
notebook/ms/comments
notebook/ms/docx_template.docx
notebook/ms/ms.docx
notebook/ms/ms.pdf
notebook/ms/.~lock.ms.docx#
notebook/ms/Rd1_comments*
README.md~
index.md~
```

Note that there are some shortcuts that allow you to tell git to ignore multiple files at once. For example, the asterisk on the line showing `notebook/ms/Rd1_comments*` tells git to ignore anything in a file or subfolder that starts with `notebook/ms/Rd1_comments` (in practice, anything in the folder ‘Rd1_comments’, which is inside the folder ‘ms’, which is inside the folder ‘notebook’). If I wanted git to ignore everything in the folder ‘notbook’, then I could add the line `notebook*` (and remove all of the other lines that start with `notebook`). Similarly, the line showing `src/*.o` tells git to ignore any file with the extension ‘.o’ that is located inside the folder ‘src’ (e.g., `src/file1.o` and `src/file2.o` would be ignored).

> **Exercise:** *Create and save a .gitignore file that ignores folder in your repository. Stage and commit the file in GitKraken, then add some files to the ignored folder to confirm that git is ignoring them.*

You can see the .gitignore file for this repository here (though obviously not anything in the folder ‘ignored_folder’).

Note that **GitHub does not allow you to push files larger than 100 MB**, but there are ways of dealing with especially large files aside from just ignoring them. It is also worth mentioning that there are no limits in GitHub on the number of total files or repositories that you have, or the total amount of space you use (as of the time of writing, you just can’t push files larger than 100 MB).

Exercise: *Initialise a new repository somewhere on your computer using GitKraken, then add all of the files relevant to one of your research projects (e.g., all data, analysis, and writing for a thesis chapter). Try working from this repository while using control in GitKraken and pushing and pulling from GitHub (you might want to initialise the repository as private).*

Additional resources

Cheat sheets

- Cheat sheet for GitKraken with GitHub
- Cheat sheet for GitKraken Interface

Introductions to version control (guides)

- British Ecological Society: A guide to reproducible code in ecology and evolution
- An introduction to version control

Introductions to version control (videos)

- Git and GitHub for Poets: Intro (no programming knowledge needed)
- Git and GitHub for Poets: Branches
- Git and GitHub for Poets: Fork and Pull