LaTeX on the road

Piet van Oostrum

1 The context

In July 2018, my wife Cary and I were travelling in South America to visit friends in Brazil and Bolivia, and additionally to have some vacation. We wanted to travel light, so I had decided not to take my MacBook with me, saving a little bit more than 2 kgs. of weight. We both had our iPhones and iPads (mine is an iPad mini), and we hoped that would do. They were mainly to be used for reading email, interactions on social media, searching for city and transport information, and the like.

I did not expect to do any TeX work, maybe some light programming, for which I had a Python system (Pythonista[1]) on my iPad.

While we were travelling in Brazil, on our way to Bolivia, I got an email from a user of the `multirow` package about a possible bug. It came with a solution which was a very simple substitution, and back home on the laptop, it would have been a few minutes to make the change, check it into the version control system, do some tests, generate a new version of the documentation, and upload the new version to CTAN.

Because this person had already made a local change, and the problem was not urgent anyway, my first reaction was: I will correct it when I am back home, which, by the way, would be some two months later. However, when we arrived in Bolivia, where we were staying a couple of weeks, the temptation to solve the problem right there became too large.

First published in *MAPS* 49 (2019.1), pp. 58–70. Reprinted with permission.

[1] `http://omz-software.com/pythonista/`

Figure 1: On our way to Brazil

Figure 2: Our trip

But what would have taken at most 10 minutes at home became a major effort without having a computer with a TeX system. In the end it took me more than two days of struggling, but with victory in the end.

If I distributed the package just as a collection of `.sty` files (there are three included), with separate documentation, the task would have been simple. I could have downloaded the package from CTAN, changed the `.sty` files with a text editor in my iPad, and uploaded them back to CTAN. It might have caused some frowning from the CTAN maintainers if the version number in the documentation would have been different from the one in the `.sty` files, but that would have been temporary anyway.

However, the package is distributed as a `.dtx` file, with a corresponding `.ins` file, and a separate PDF file containing the documentation which is generated from the `.dtx` file. The `.sty` files are also generated from the `.dtx` file with the aid of the `.ins` file. This is the standard setup for most CTAN packages. But this requires the `.dtx` and `.ins` files to be processed by LaTeX (or TeX in case of the `.ins` file). And I did not have a LaTeX distribution on my iPad.

2 What were the options?

There were in practice two solutions:

- Install a LaTeX system on my iPad.
- Use an online (cloud-based) LaTeX system.

2.1 LaTeX apps on the iPad

I found two LaTeX apps in the iOS App Store: Texpad and TeX Writer (see figure 3). Both are offline apps, i.e. you don't need an Internet connection to compile your LaTeX documents. But, on the other hand, to limit the size of the application, they don't

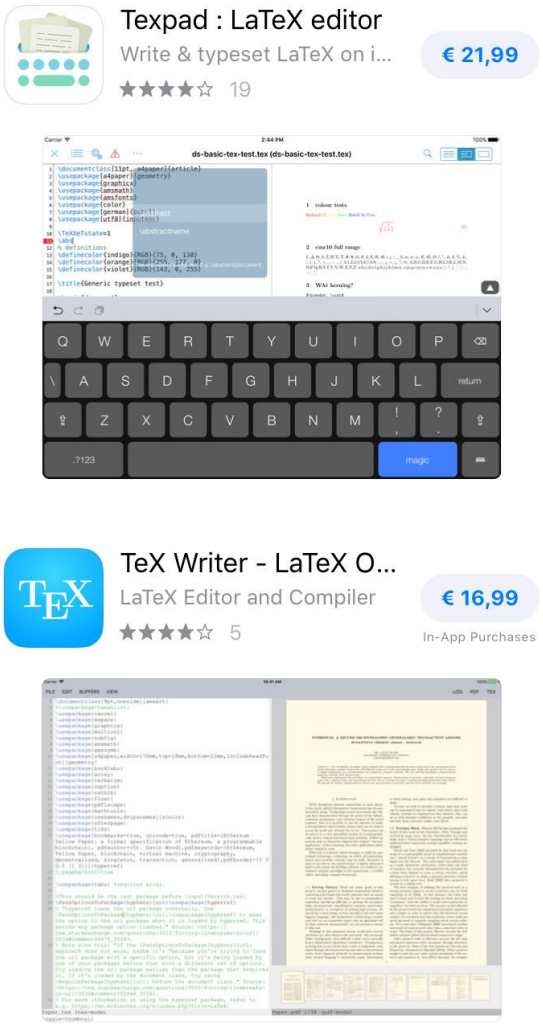

Figure 3: Texpad and TeX Writer in iOS App Store

have every package from CTAN installed. You can install additional packages, but as iOS is quite a closed operating system, you are dependent on the developers to supply these packages. Of course you can always add the required files to your project directory, but there might be some cases (e.g. if you need additional fonts) where this is not sufficient.

Also it isn't clear from the documentation of these packages if they can process something like `.dtx` and `.ins` files to extract the `.sty` files and the documentation for the package, which was essential in my case. I got the impression that they were mainly meant for the 'normal' user to write articles and reports.

They are also not particularly cheap. At the time of writing Texpad costs €21.99 and TeX Writer €16.99. If I remember correctly they were a little bit cheaper at the time I was travelling. In itself that is not a very steep price, but I did not expect to use it very often, and for just this single case I thought it was too much. And they don't have a trial version to see if it suits you, so if you buy one of these, and you don't like it, you have effectively lost your money. And then there is this nagging choice: which of the two is better? All in all, I decided not to go that way.

For the cloud-based systems, I had heard about Overleaf (formerly called WriteLatex) and ShareLaTeX, so I decided to investigate these. It appeared that at that time, these two systems were in the process of being merged. The result was Overleaf version 2 which had the ShareLaTeX interface, but was still in beta phase. For the simple task that I had, a free account would be sufficient, so I started to try that. However, the merging process introduced some teething troubles. In fact it made editing the files from the iPad browser almost impossible. It wasn't clear if this was a specific problem on the iPad, or if the browser interface in general was not yet mature enough. In effect it wasn't usable at all, because its behaviour was very erratic.

I also tried the Overleaf version 1 interface, but I could not get that working either. I have no idea whether these problems were iPad specific, but anyway I could not use it. By the way, the Overleaf editor is now functioning also on the iPad. However, some functionality is not available without an external keyboard, because they are invoked with control keys. For example the search function is invoked by Control-F on Windows and Linux, and by Command-F on MacOS. On an iPad you can't give these with the virtual keyboard. With an external keyboard it is possible. The current Overleaf editor is reasonable. It has some TEX-specific functionality. For example, if you type `\begin{enumerate}` the editor adds `\item` and `\end{enumerate}` and positions the cursor after the `\item` (see figure 4).

```
712 \begin{enumerate}
713     \item |
714 \end{enumerate}
```

Figure 4: Overleaf editor supplies useful parts

2.2 Cloud-based LATEX systems

Despite the problems that the editor gave at that time, it seemed to me that this was the best way to go forward. Figure 5 shows the screen from the current version of Overleaf on my MacBook. The default screen has an edit window with the LATEX source text and a preview window with the resulting PDF. The preview is not live, you have to hit the Recompile button to update it. There is also a file

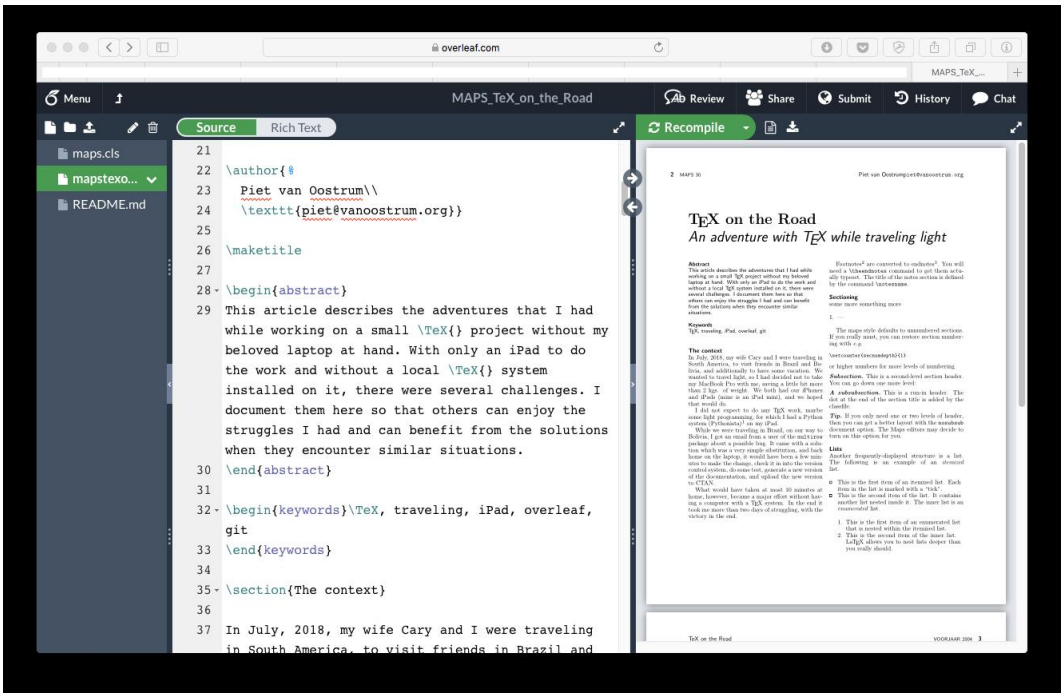

Figure 5: An early version of this article in Overleaf, with some of the `maps.cls` documentation still in place

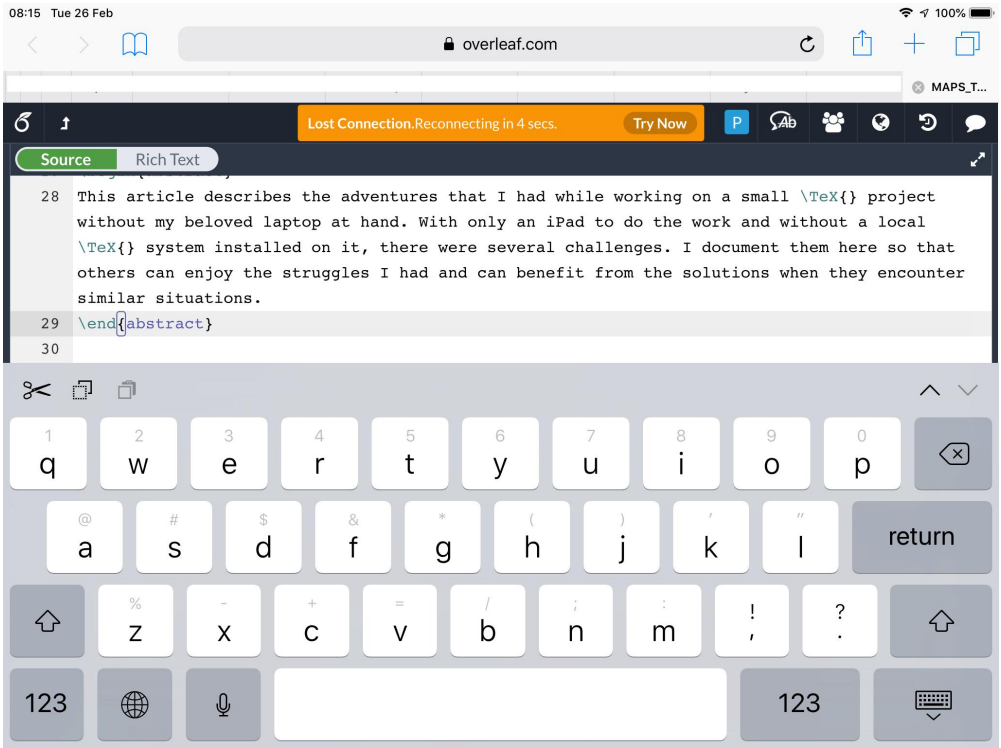

Figure 6: Overleaf screen with virtual keyboard on an iPad

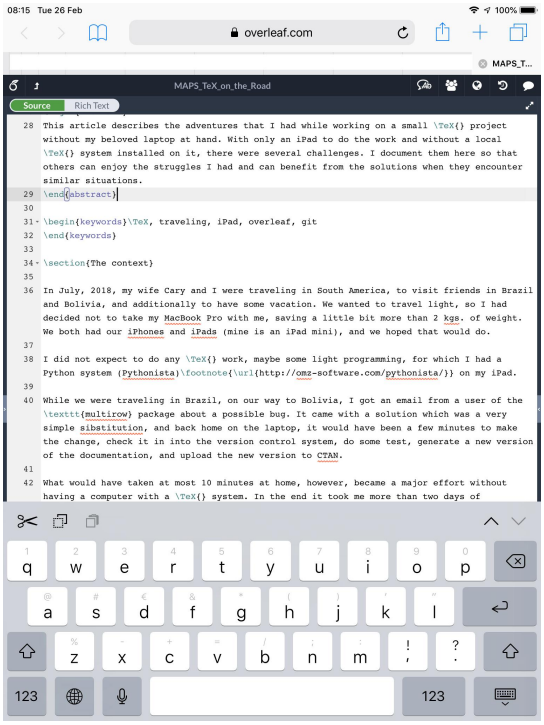

Figure 7: Overleaf screen with virtual keyboard on an iPad in portrait mode

Figure 8: iPad mini with external keyboard

list on the left and it has the capability to hide or show each of these parts and to adjust the sizes of each part. Especially on the smaller iPad screen it is advisable to have only the source code part showing while editing. But even then, the virtual iPad keyboard takes so much space that hardly any source code is visible (see figure 6). Also in this case, the file list at the left would make the edit window even smaller, but the file list can be hidden, as shown in the image.

It helps to put the iPad in portrait mode, as shown in figure 7. But then the keyboard is rather small. For a setup like this to be workable, it would be better to use an external keyboard. There are several keyboards on the market that can be used. They are generally connected through Bluetooth. They are light-weight and don't take much space, so are ideal for travelling light (see figure 8). I did not have one at that moment, however.

3 Setting up the project

Setting up the project is easy. You can create a new project in the Overleaf in the Web interface. You can upload each file individually, or a zip file with everything included. Overleaf will unpack the zip file in your project.

Immediately, it became apparent that there was a problem with my project. Overleaf wants you to designate one of your files as the main TEX file, which for me would have been `multirow.dtx`, but it doesn't accept this. It wants to have a `.tex` file. It does not recognise the `.dtx` file as a valid LATEX file. Nor does it want to edit the `.dtx` file, but as the editor was unusable, this was of a minor concern. I would have to edit the files locally on my iPad anyway.

So I had to give it a `.tex` file extension to make it (and myself) happy. I tried two ways

- Copy `multirow.dtx` to `multirow.tex`
- Make a file `multirow.tex` that just contains `\include{multirow.dtx}`

I had expected that each of these would compile the `.dtx` file when the Compile button would be pressed. However, it didn't. It took some time to find out why. My `multirow.dtx` contains a line

```
\DocInput{\jobname.dtx}
```

which is quite usual in `.dtx` files. After some searching I found out that `\jobname` wasn't `multirow` as was to be expected, but `output`. It appears that Overleaf runs the job in a kind of *sandbox* where the jobname of the main file is `output`.

After some googling I found that Overleaf uses `Latexmk`[2] to process the job. It provides a standard, but invisible, `latexmkrc` file that controls the compilation process. However, you can also supply your `latexmkrc` file. This file, and the handling of the `output` name, is described in section 'Latexmk' on page 242.

So the challenge was now to upload a correct `latexmkrc` file, and to update the `multirow.dtx` file. This could be done by uploading these files after

[2] `https://mg.readthedocs.io/latexmk.html`

each modification, but this might be an error-prone process, and you don't have a record of what has been done. Enter *version management*.

4 Distributed version management

In any project where you have to make changes more or less regularly, it is important to keep track of what you have done. Also, in general it is useful to have access to previous versions of your project, for example if you want to go back to a previous situation. Some people do this by making copies of their files at regular moments. Sometimes they put the date and the time in the file names, to keep a kind of history. But this soon becomes unwieldy. This is the problem that version management systems (also called version control systems) offer a solution for. Any serious developer, whether of software or text, should consider using a version management system.

For those readers that are unfamiliar with version management, here follows a brief description. You have a *working copy* or *working directory*, which is the collection of files that you work upon in your project. This is just like when you do not use version management. Additionally you have a *repository*, which is a kind of database containing the history of your project. It will contain the state of your *working copy* at certain moments in the past, together with information about who made the changes, and a description of what has changed.

If, at a certain moment, you have a state of your project that you want to keep, you *commit*, which means a copy is stored in the *repository*, together with a description that you enter. The opposite operation (i.e. making a copy from your repository to your working directory) is called *checkout*. You usually have a separate repository for each project. The repository can be on your local computer, or on a server. In the latter case it is possible that different people working on the same project use the same repository. They would then each have their own *working copy*. As they are working independently, these could be different. A version management system usually has provisions to resolve conflicting working copies.

Although these systems can store any type of file, they work best with plain text files. As our TEX sources are plain text, they are ideal candidates for using a version management system.

There are several version management systems available. One older, well-known system is *subversion* (SVN[3]). It usually has the repositories on a central server, but you can also have the repository on your local computer, if you are working alone. As SVN has only one repository per project it is called a centralised version management system.

Centralised version management systems have some big disadvantages for cooperation in teams:

- If you work together the repository must be on a central server, which means you cannot use it when you are offline.
- If you want to keep your changes registered often in the repository, then this can be confusing for the other team members. On the other hand, if you want to keep the repository relatively clean, that is, only commit major updates, then you lose the possibility to keep your own history detailed.

One solution is to have both a central repository for the team, and a local repository for your own work, but then synchronising these repositories could become tedious. However, this is where *distributed version management systems* have their strength.

In a *distributed version management system* you can have both a local repository on your computer and a central repository on a server. Or even more than one of each. Furthermore, these can be easily synchronised. The usual way to work in a team is to have a central repository for the team, and a local repository on each team member's computer. Each team member keeps a history in the local repository. This can be done often, and also offline. When changes are good enough to be put in the central repository, a team member *pushes* the local changes to the central repository, often after making one set of changes that do not reflect all the details of the work done locally. Another team member can then *fetch* these changes from the central repository when they want to be up to date. It is then probable that the newly-fetched changes are not consistent with other changes that they have made themselves in the meantime. The two sets of changes must then be *merged*. This is the basic scenario. Much more complicated workflows are also possible.

Both centralised and distributed version management systems support the concept of *branches*. A *branch* is a separate line of development in your project. For example you have a project that you publicly release from time to time. The development of this release version would for example take place on the main branch in your repository. Now after a release you want to start working on some very new experimental features for a future release. If you just continue your development, then when a bug in your release is detected, your project would be in an unstable state. So you cannot just apply a bug-fix

[3] `http://subversion.apache.org`

to the current state of your project, but you would have to go back to the state just after the release. As the repository has kept the history of your project, this is easy, but you want also to keep the current state, so that you can go back there after making the bug-fix.

Here branches come to the rescue. After your release you create a new branch for your experimental work, and continue working there. When you want to make the bug-fix, you switch back to the main branch. The repository will remember your experimental branch, and after releasing the bug-fix you can switch back to the experimental branch. If you wish you can then also *merge* the fix in your experimental branch. Later when your experiment is successful and you want to release it, you can merge it back to the main branch. You can have as many branches as you want. For example if your bug-fix is expected to be complicated, you can first try it out on a separate branch.

A very popular site for central repositories is Github.[4] This site is based on the distributed version management system Git. Git is probably the most popular version management system in use today. For my own projects I use Git exclusively nowadays, often only locally, but sometimes in combination with Github.

4.1 Use with Overleaf

To come back to the TEX project I am currently describing, it appeared that Overleaf also had Git capabilities. Although these were in beta phase at that moment, it could be used for my project. Nowadays you need a paid account on Overleaf to use the Git facilities, but because I had started using them during the beta testing, I have access to them in my free account.

Git can be used in two ways on Overleaf.

- your Overleaf project can function as a Git repository;
- your Overleaf project can be synchronised with a Github repository.

I decided to take the Github route, mainly because I have experience with Github and I could not get the direct Git repository on Overleaf working from the iPad. At this moment it is working, but its functionality is very limited compared to Github.

In order to use Git on the iPad you need a Git app. I found Git2Go,[5] which is said to be the first app to use Git on iOS. It worked well for my needs, but later I tried two others that I found: Working Copy[6] and TIG.[7] In the appendix I give a comparison of these apps.

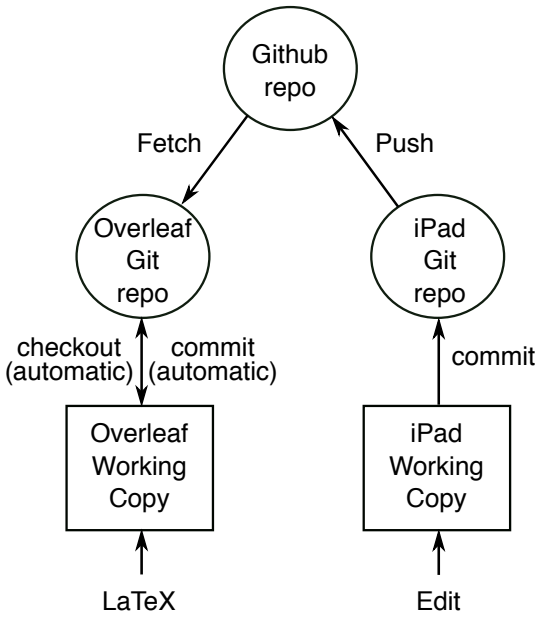

Figure 9: Git workflow

My workflow can be seen in figure 9. My editing took place in the lower right corner, on the working copy (managed by Git2Go). I could have used the editor that Git2Go provides, but it is not very sophisticated. It does not have syntax highlighting for LATEX files, and it gives no editing support beyond the standard iPad keyboard. I also had a much better text editing program called Textastic.[8] It has syntax highlighting for LATEX, good search facilities and an extended keyboard (see figure 10) that makes it easier to enter non-alphanumeric symbols. Also it has a special provision for easy cursor movement. Git2Go, and the other Git apps mentioned above, function as a kind of file system, which means that Textastic can directly edit their files without copying between the two apps. So the only extra operation to edit in Textastic rather than in Git2Go itself is switching between the apps. This extra effort I deemed worthwhile for the added comfort of using a good text editor.

After editing the file(s), I switch to Git2Go, commit the change, and immediately push it to the Github repository. Then I switch to Overleaf in the browser, fetch the changes from Github to Overleaf in the Overleaf synchronisation menu, and process the files, hopefully producing a new PDF file. Many

[4] https://www.github.com
[5] https://git2go.com
[6] https://workingcopyapp.com
[7] https://itunes.apple.com/us/app/tig-git-client/id1161732225
[8] https://www.textasticapp.com

Figure 10: Textastic extended keyboard

times it did not yet work correctly, so I had to go back to Textastic and start a new cycle.

The problem wasn't so much in the LaTeX code, as the changes there were very simple. The main problem was getting the `latexmkrc` file correct. One difficulty was that Overleaf did not have good documentation about the context in which the Latexmk program was running. Also, running it on their server did not give as much feedback as running on your own computer. Several times I had to write extra information to a text file, and then download that to the iPad to see what happened. For example, I had to make directory listings, and write them to a text file, just to see what files were generated and what their names were. And the process was a bit tedious because I had to synchronise the files as described above before each try. But after some 50 tries, everything worked perfectly. I will spare you all the attempts that I made, but in the next section I will give you the resulting `latexmkrc` file, and explain what it does.

5 Latexmk

Latexmk is a program (a Perl script) to process a LaTeX file with all the necessary `bibtex`, `makeindex` and similar calls. It will run LaTeX and these other programs as many times as is necessary to get a completely processed and stable output.

For the run-of-the-mill LaTeX file, Latexmk has enough knowledge to know what to do. However, when there are additional requirements, like a non-standard index, glossaries, etc., you must give Latexmk a recipe of how to process the various stages. The recipe is given in the `latexmkrc` file, which in fact is also a Perl script. Latexmk has an enormous number of possibilities, and its manual[9] contains 48 pages. So it took some time to get everything right.

Overleaf provides a standard `latexmkrc` file for its jobs, but as we have seen above, this is not adequate for processing the `.ins` and `.dtx` files. To make Overleaf happy, we must provide a main `.tex` file, but with our `latexmkrc` file we don't use it, so its content is unimportant.

In figure 11 the resulting `latexmkrc` for this process is given, annotated with line numbers. In the remainder of this section I explain what it does.

line 1. This sets the timezone to your local time. This is so that messages with date and time will get your local time, and not the time of Overleaf's servers, which would be useless in most cases. As I was in Bolivia at the time, the timezone was 'America/La Paz'. Now at home it would be 'Europe/Amsterdam'.

line 3-6. In a `.dtx` file the extension `.glo`, which is normally used for glossaries, is used for the list of changes. And the sorted version, to be created by `makeindex`, will be `.gls`. These lines give a recipe how to create the `.gls` file from the `.glo` file using `makeindex`.

line 8. For processing the normal index in a `.dtx` file `makeindex` needs the additional argument `-s gind.ist`.

line 10. This defines which extra file extensions we need in the process. Besides the already mentioned `.glo` and `.gls`, there is also `.glg` which is the log output of the `makeindex` command from line 5. And the `.txt` extension is used for debugging.

line 12. Here comes the trick to let Overleaf do our work. Normally it will run `pdflatex` on the main TeX file in the project, which in our case is `multirow.tex`. But you can define the `$pdflatex` variable to let it use another command. In our case we let it run the internal function `mylatex` that follows. In this function we do all the preparatory work before we run the actual `pdflatex` command.

[9] `http://mirrors.ctan.org/support/latexmk/latexmk.pdf`

Latexmkrc file:

```
1  $ENV{'TZ'} = 'America/La Paz';
2
3  add_cus_dep('glo', 'gls', 0, 'makeglo2gls');
4  sub makeglo2gls {
5      system("makeindex -s gglo.ist -o \"$_[0].gls\" \"$_[0].glo\"");
6  }
7
8  $makeindex = 'makeindex -s gind.ist -o %D %S';
9
10 push @generated_exts, 'glo', 'gls', 'glg', 'sty', 'txt';
11
12 $pdflatex = 'internal mylatex';
13 sub mylatex {
14     my @args = @_;
15     (my $base = $$Psource) =~ s/\.[^.]+$//;
16     system("tex $base.ins");
17     # backslashes are interpreted by (1) perl string (2) shell (3) sed regexp
18     # therefore we need 8 backslashes to match a single one
19     system("sed -e s/\\\\\\\\\jobname/$base/g $base.dtx > $base.tex");
20     return system("pdflatex @args");
21 }
```

Figure 11: The final latexmkrc file. The line numbers are not part of the file.

line 14. Pick up the arguments from the call to `mylatex` in the variable `@args`. This is standard Perl prose.

line 15. Latexmk puts the name of the main TEX file in `$$Psource` (see page 45 in the Latexmk manual). This line is actually a shorthand for two statements:

```
my $base = $$Psource;
$base =~ s/\.[^.]+$//;
```

The first line copies `$$Psource` to a local variable `$base`. The second line strips off anything after (and including) the last dot. So the string '`multirow.tex`' will be transformed to just '`multirow`'. I use `$$Psource` rather than just using `multirow` so that now the `latexmkrc` file is also usable for other `.dtx` files.

line 16. First we run `tex` on our `.ins` file, which would be `multirow.ins` in our case. This generates the required `.sty` files. This is to ensure that we use the new versions of our `.sty` files, rather than an outdated version in Overleaf's TEX system.

line 19. From our `.dtx` file we generate a `.tex` file where the text `\jobname` is replaced by the actual base name of our file (in our case `multirow`). This is necessary as Overleaf defines a `\jobname` of `output`. So in this case we generate `multirow.tex` from `multirow.dtx`. This `.tex` file will input `multirow.dtx` during its processing.

We do the replacement by calling the Unix program `sed`. The `\jobname` is inside a regular expression in `sed`, therefore the backslash must be doubled. But then, this command is processed by the Unix shell, which also interprets backslashes. Therefore we must double all the backslashes again. And then this command is inside a Perl string where backslashes are also interpreted. So we must double them again, and we end up with 8 backslashes to represent a single one.

line 20. Finally we run the real `pdflatex` command with the original arguments. Note that we process the new `multirow.tex` file, because that is what Overleaf expects to do. Also, because this is run in a *sandbox* (i.e. on a copy of the original files in a separate directory), this does not affect our original file.

Finally, we also show a modified `latexmkrc` file with debugging statements included in figure 12, and the corresponding output in figure 13. You see the values of `$$Psource` and `$base`, the arguments to the `pdflatex` call, and the directory listing at the end of the process. Note that in the directory listing there is a file `multirow.log`; this is the result of the call `tex multirow.ins`. Note also the generated

Latexmkrc with debugging:

```
$ENV{'TZ'} = 'America/La Paz';

add_cus_dep('glo', 'gls', 0, 'makeglo2gls');
sub makeglo2gls {
    system("makeindex -s gglo.ist -o \"$_[0].gls\" \"$_[0].glo\"");
}
$makeindex = 'makeindex -s gind.ist -o %D %S';

push @generated_exts, 'glo', 'gls', 'glg', 'sty', 'txt';

$pdflatex = 'internal mylatex';
sub mylatex {
    my @args = @_;
    Run_subst("echo \"%%B=%B %%R=%R %%S=%S %%T=%T\" > debugout.txt"); ## DEBUG ##
    system("echo '\@args' = \"@args\"               >> debugout.txt"); ## DEBUG ##
    system("echo '\$\$Psource' = \"$$Psource\"     >> debugout.txt"); ## DEBUG ##
    (my $base = $$Psource) =~ s/\.[^.]+$//;
    system("echo '\$base' = \"$base\"               >> debugout.txt"); ## DEBUG ##
    system("tex $base.ins");
    # backslashes are interpreted by (1) perl string (2) shell (3) sed regexp
    # therefore we need 8 backslashes to match a single one
    system("sed -e s/\\\\\\\\\jobname/$base/g $base.dtx > $base.tex");
    $status = system("pdflatex @args");
    system("ls -l                                   >> debugout.txt"); ## DEBUG ##
    return $status;
}
```

Figure 12: `latexmkrc` file with debug statements

Debug output:

```
%B=output %R=output %S=multirow.tex %T=multirow.tex
@args = -synctex=1 -interaction=batchmode -recorder
        -output-directory=/compile --jobname=output
        multirow.tex
$$Psource = multirow.tex
$base = multirow
total 1176
-rw-r--r-- 1 tex tex   3871 Mar  4 14:12 README
-rw-r--r-- 1 tex tex     49 Mar  4 14:12 README.md
-rw-r--r-- 1 tex tex   1417 Mar  4 14:12 bigdelim.sty
-rw-r--r-- 1 tex tex   1234 Mar  4 14:12 bigstrut.sty
-rw-r--r-- 1 tex tex    203 Mar  4 14:12 debugout.txt
-rw-r--r-- 1 tex tex   1054 Mar  4 14:12 latexmkrc
-rw-r--r-- 1 tex tex  80398 Mar  4 14:12 multirow.dtx
-rw-r--r-- 1 tex tex   2182 Mar  4 14:12 multirow.ins
-rw-r--r-- 1 tex tex   3719 Mar  4 14:12 multirow.log
-rw-r--r-- 1 tex tex   5022 Mar  4 14:12 multirow.sty
-rw-r--r-- 1 tex tex  80398 Mar  4 14:12 multirow.tex
-rw-r--r-- 1 tex tex   3487 Mar  4 14:12 output.aux
-rw-r--r-- 1 tex tex      0 Mar  4 14:12 output.chktex
-rw-r--r-- 1 tex tex  25207 Mar  4 13:10 output.fdb_latexmk
-rw-r--r-- 1 tex tex  20593 Mar  4 14:12 output.fls
-rw-r--r-- 1 tex tex   3281 Mar  4 14:12 output.glo
-rw-r--r-- 1 tex tex   3578 Mar  4 08:13 output.gls
-rw-r--r-- 1 tex tex   3270 Mar  4 14:12 output.idx
-rw-r--r-- 1 tex tex    891 Mar  4 08:13 output.ilg
-rw-r--r-- 1 tex tex   2655 Mar  4 08:13 output.ind
-rw-r--r-- 1 tex tex  34086 Mar  4 14:12 output.log
-rw-r--r-- 1 tex tex 610336 Mar  4 14:12 output.pdf
-rw-r--r-- 1 tex tex 262970 Mar  4 14:12 output.synctex.gz
-rw-r--r-- 1 tex tex   1467 Mar  4 14:12 output.toc
```

Figure 13: `latexmkrc` debug output (the `@args` line has been broken into several lines for print)

`.sty` files. The files resulting from the `pdflatex` call on `multirow.tex/dtx` are all called `output.*`. So `makeindex` must also act on these files. In figure 11, line 5, this is accomplished because the file name is given as an argument to the function `makeglo2gls`. In line 8 it is accomplished because the patterns `%S` and `%D` are replaced by the *source* and `destination` of the command, respectively, i.e. `output.idx` and `output.ind`.

6 Conclusion

Although working at home on my MacBook is much more comfortable, it is possible to do some serious LaTeX work on your iPad while you are travelling. It takes some effort to find the proper way to do it, however. I hope this article helps you to get started if you need this work flow.

A Appendix — iOS Git apps compared

In this section I compare the three Git apps on iOS that I tried. I did all the production work in Git2Go, but after it was finished I also tried Working Copy and TIG.

Git2Go has a limitation that it only cannot work with Git repositories on all servers. It works with a limited number of services, namely Github, Bitbucket[10] and Gitlab[11]. Other remote repositories can be used if they offer access by the SSH protocol. SSH is one of the two main protocols used to connect to Git servers. The other is HTTPS. Overleaf only offers HTTPS, which Git2Go does not support.

To create a repository on your iPad you must *clone* (i.e. copy) an existing repository on one of the supported servers. You cannot create a local-only repository on the iPad. Once you have the repository on your iPad, you can edit the files in the repository, commit the changes, create new branches. It can fetch from and push to the remote repository, but these are not separate operations. It always does a fetch (which may be empty), followed by a push. It can also merge different branches. It is a limited set of operations compared to the full Git functionality, but it is sufficient for a normal workflow as described above. Also cooperation with other people would be possible as long as the more esoteric Git functionality is not required. Git2Go's editor has syntax highlighting for a limited number of programming languages.

Git2Go is free, as long as you only access *public* repositories (i.e. repositories that everybody can see). To access private repositories you would have to buy an upgrade.

For the push operation you will have to login, and Git2Go will remember your username and password, until you explicitly logout.

And occasionally it crashes.

Last minute note: I later tried to re-install Git2Go on my iPhone, and got the message that it was no longer available on the App Store. Also a search in the App Store did not come up with Git2Go. I have no idea if this is a permanent situation, or if it might be in the process of updating.

Working Copy is the nicest of the three apps. It has a very elaborate set of functions. It can connect to all kinds of servers, including Overleaf. However, to use the `push` functionality you have to pay. The price is quite steep (€17.99 at the time of writing), but you can get a free 10 day trial. I used this for writing this article to see how it worked.

Working Copy can clone from existing repositories, including through SSH and HTTPS, and also create local repositories. It can also create a local repository from a `.zip` file. Once you have a repository it can connect your repository to more than one remote repository, which sometimes can be quite handy. For example in the current example, the repository on the iPad could have been connected both to the Overleaf repository and to the Github repository. Of course you will have to be careful not to mess up your workflow.

If your iPad is connected to a Mac or PC with iTunes, you can drag and drop a repository on your computer through iTunes, and it will be copied to the iPad.

Working Copy's editor has syntax highlighting for more than 50 different languages. It can show nice graphical representations of your branches and your commit history (see figure 14). Besides the `merge` functionality it also has the `rebase` functionality, which is an alternative for merge. For cooperating in large projects this functionality is sometimes necessary.

There is more than fits in this limited space, but Working Copy is by far the best of the three apps. It is expensive, but if you do a lot of work with Git on your iPad, it is worth the price. Working Copy operates in a small market, so the price is understandable.

TIG is the third app I tried. It takes more or less a middle ground between Git2Go and Working Copy. Like Working Copy it can connect to all kinds of repositories, including Overleaf, and it has *push* functionality. And it is free. It can clone existing repositories, and create local ones. It can also connect repositories to more than one remote repository.

Its editor has syntax highlighting support for 166 languages.

However, although the functionality is great for a free app, I found its user interface sometimes confusing. And to fetch/push to your remote repositories you have to enter your username and password every time. I did not find a way in which it could remember these. This is very annoying. And it crashed quite often.

As I mentioned above, all these apps have the facility that you can open their files in an external editor. Figure 15 shows how to open files from Git2Go in Textastic. This is done inside Textastic with the

[10] `https://bitbucket.org`
[11] `https://gitlab.com`

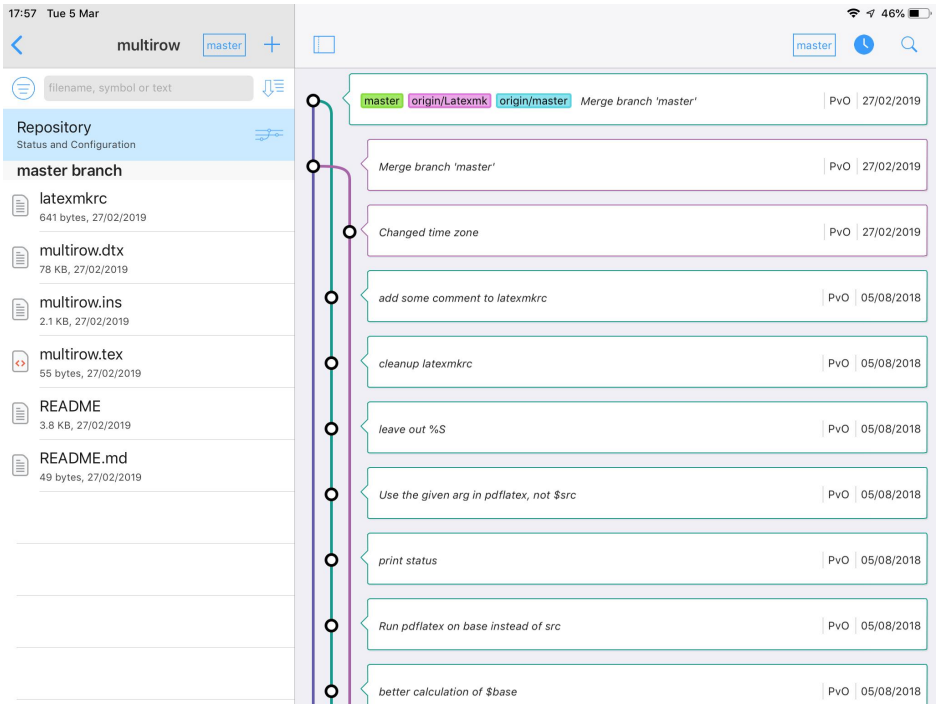

Figure 14: Graphical commit history in Working Copy

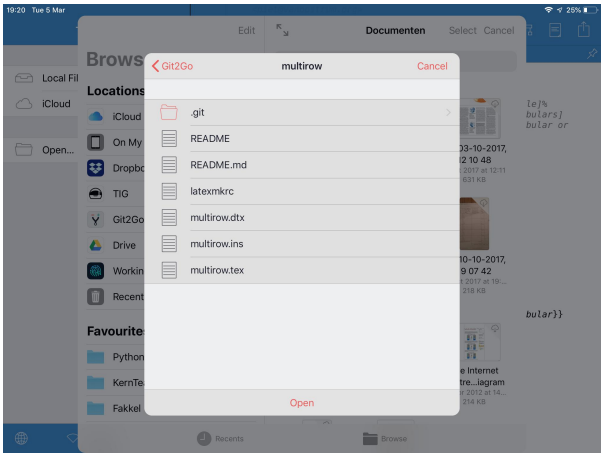

Figure 15: Opening a Git2Go file or repository in Textastic

"Open. . . " button, then selecting "Git2Go". It is then possible to choose "Open" down in the pop-up, which will open the whole directory in Textastic, or select one filename, which will open that file.

Summary:

- If you only need access to repositories hosted by Github, Bitbucket or Gitlab, or repositories that can be accessed by the SSH protocol, and your requirements are modest, you can choose Git2Go (if still available).
- If you need access to repositories that do not fall in the previous categories (such as Overleaf), and you can live with a not so optimal user interface, and your requirements are modest, you can choose TIG. It may be a good choice when you want to connect to a repository that Git2Go does not support, and when you find Working Copy too expensive.
- If you want the top Git app on your iPad (or iPhone) and are willing to pay the price, I would recommend Working Copy. If you want to do serious work with Git, this is the choice and it would be worth the price.

There are nowadays some other Git apps available, but it seems that they are roughly comparable to one of the above. Some of them only support just Github, Bitbucket or Gitlab. I have not found any free app that comes with the functionality of Git2Go or TIG. Other paid apps may be slightly cheaper than Working Copy, but they also have less functionality.

◇ Piet van Oostrum
http://piet.vanoostrum.org
piet (at) vanoostrum dot org