Introduction

Over the past decade of my academic and professional career I spent a fair bit of time documenting the things I was working on. While in graduate school this was primarily in Latex, documenting thousands of pages, some of which are available in my grad school notes. For the past few years my work has been less academic, less mathematical, and involved more programming. It was also not often distributed or published. As such my documentation lately has been primarily in Markdown. I found myself initially using Atom with the markdown-pdf plugin to periodically generate PDFs when necessary. This approach was fine, but it provided no real support for equations, and Atom was not my preferred editor.

I was looking for a better solution than this that provided the following:

- **Short syntax** that is easily human readable (e.g. markdown headings as `#` instead of Latex `\section{}` or HTML `<h1></h1>`). This was to make my writing quicker and more efficient and the document source easier to read. From this perspective, markdown was an attractive option. For example, bullets in markdown source look like bullets, versus those in Latex or HTML.
  - This was could be facilitated further by good syntax highlighting in an editor, e.g. bolding headings `#` and `**bold text**`.
- **Latex equation** support. While it is expected that equations be infrequent and generally simple, they needed to be supported.
- **Cross platform** should allow the source document to be easily deployed across many platforms. For example:
  - Github wikis
  - Web via a static site generator like Hugo, used to create this site
  - PDFs. It is important to note that I don't anticipate the need for documents to be duplicated across these mediums, rather I wanted to be able to adopt an efficient and standard way of writing without thinking what the destination format would be when writing.
- **Flexible styling** across the above platforms. A solution which facilitated uniform styling across mediums would be ideal, whereby a single `.css` or `.sty` file could be used to consistently format both a generated PDF and content for the web.
- **Minimal tooling** required.
- **No dedicated app** or IDE required to effectively note-take.
- **Fast to compile or view**. However, the need for this was inversely proportional to the complexity of the document syntax. That is, if I could adopt a solution with a sufficiently simple syntax that allowed it to be easily read, then it would reduce the necessity of compiling to view the generated output while working.
- **Offered citation support** for referencing entries from a `.bib.` file.

Solution: Markdown and Pandoc

After considering the above requirements for a simple, easy-to-read syntax, markdown was a good choice. Furthermore, it is widely supported on and offline, with easy tooling to generate PDFs including Pandoc and good support online at Github, Gitlab, and with Hugo, Jekyll, and more. Markdown supports embedded HTML and Latex with tools like KaTex and MathJax.

For the generation of PDFs Pandoc is easy to use and supports Latex. Latex is not supported in Github, but seems to be in Gitlab. Markdown is easy-to-read meaning I can efficiently view, edit, and write the source document, reducing the need for real-time rendering or to frequently and quickly compile. References can be included from a `.bib` file. Using the Sublime Text markdown syntax highlighting works well for markdown, although it does not do anything for Latex. But as the primary goal was for a simple, easy syntax that allowed for the inclusion of equations rather than a focus on highly mathematical documents, this was not a big deal - equations should be relatively simple and infrequent. And admittedly I've made no attempt yet to see if there are any options for syntax highlighting that may work better for markdown with Latex.

Latex was an alternate source format considered, with Pandoc providing tools for conversion to markdown or HTML for use on the web, and easy generation of PDFs. However, even with templates, Latex source is more verbose and and cumbersome to use for relatively simple note taking and documentation, most of what I do now. Furthermore, Latex is not a format well supported on the web. Jupyter was considered as well but seemed much more heavyweight than desired, required more tooling, and is not as widely or easily supported as markdown (although it is supported in Github, for example.)

Styling flexibility with this solution is nearly unlimited, although consistency between the web (e.g. via Hugo) and generated PDFs (e.g. via Pandoc and Latex) may not be easily maintained though. CSS used for the web could not necessarily be applied to PDFs and vice versa, but of the above requirements styling is not the most important. Once the desired styling is set for each of these outputs, it will likely not often be changed. There may even be options to use CSS with Pandoc and Latex, or tools to generate a `.sty` file from CSS, although this is another thing I've not yet looked into at the time of this writing.

Using Pandoc

The Basic Pandoc command for generating `doc.pdf` from `doc.md` is:

```
$ pandoc doc.md -o doc.pdf
```

For more about Pandoc check out the Pandoc User's Guide.

Bibliography

A bibliography, in the form of a `.bib` can be easily be included with Pandoc. To format the references, the Citation Style Language can be specified. Thousands of CSL files can be found here. The following Pandoc options can be used to include the bibliography.

```
--filter pandoc-citeproc \
--bibliography=test.bib \
--csl ieee.csl \
```

Citing the bib entry `my-citation` are accomplished in markdown by `[@my-citation]` .

Styling

To style the Pandoc generated output, several options for templates were available that can be used with the `--template` option. The Eisvogel Pandoc Latex template was one of the simplest and easiest. Just download it, put it in the default pandoc template location `~/.pandoc/templates/` , and used with `--template eisvogel` . The result out-of-the-box is quite good, additional styling options will be described below.

Docs on Pandoc's different flavors of markdown described in the docs: Pandoc's Markdown It says in the post Customizing pandoc to generate beautiful pdfs from markdown:

> GitHub style markdown is recommended if you wish to use the same source (or with minor changes) in multiple places.

I chose to use `markdown` instead, as `yaml_metadata_block` not supported by `gfm` , nor does not work with Eisvogel template.

Bash Script

With the Pandoc options above (using filters, including a bibliography, specifying CSL file, and applying a template) the command to run Pandoc was becoming quite long. As was well described in the blog post Customizing pandoc to generate beautiful pdfs from markdown using a simple script to call Pandoc was an obvious solution. Calling Pandoc to convert a markdown to PDF required the following command:

```
$ ~/.pandoc/md2pdf.sh doc.md ~/Desktop/doc.pdf
```

The contents of the script `md2pdf.sh` with all of the final options will be listed below.

Markdown Processing in Pandoc and Hugo

With the above, the first problem that was encountered when attempting to generate a PDF from the source from this site was in the differences between the markdown processor of Hugo versus that of Pandoc. With the markdown processor in Hugo extra arguments can be passed to a code block, for example, specifying that line numbers should be turned off as shown below. However, when code such as this is used in the markdown file and Pandoc is called, it does not know how to interpret these extra arguments and ends up

rendering the code weirdly. Again, it was not necessarily a primary use case that I generate PDFs from the posts on this site, but I wanted to have the flexibility to do so.

````
```js {linenos=false}
var your = "code here"
```
````

Here is what it looks like:

```
js {linenos=false} var your = "code here"
```

It looks just fine in Hugo - a nicely formatted code block without line numbers. This is not the case when generating a PDF with Pandoc as Pandoc cannot intepret the `{lineos=false}`. In the PDF the code is formatted as in-line code rather than in a block, all of the statements are on a single line, and `{linenos=false}` appears in the output as if it were inside the code block. To address this problem this additional code needed to be interpreted by Pandoc or simply ignored, with the desired effects of such code when used on the web to be achieved via Latex styling.

Solution Option 1: Don't use Pandoc

This is the obvious solution. For markdown source that is used to generate content only for the web a separate way to generate a PDF document may or will not be necessary. In rare cases when converting web content to PDF is necessary, printing from a browser to PDF is definitely a primitive but rather effective option. This was again important to acknowledge but not a viable solution to the underlying problem of how to handle flavors or features of markdown and the different processors that may not be supported by Pandoc.

Solution Option 2: HTML/CSS Tricks and Pandoc Arguments

Another solution is to segregate markdown that is to be processed by Pandoc versus that that is to be processed by Hugo and its corresponding markdown engine. This could be accomplished using the following code, and using the Pandoc option `--from markdown-markdown_in_html_blocks-native_spans`. This tells Pandoc to process the HTML `span` with `class="hide-me"` thus showing its contents in the PDF generated by Pandoc. At the same time, when processed by Hugo for the web, the HTML `span` with `class="hide-me"` will be hidden using CSS `display: none;`. And the `p` element will naturally show in Hugo, but is hidden from Pandoc.

```
<!-- this shows in Pandoc -->
<span class="hide-me">
```js
var your = "code here"
```


<!-- this shows in Hugo -->
<p>
```js {linenos=false}
var your = "code here"
```
</p>
```

The results of this solution are as desired. With some cumbersome HTML and CSS, parts of the source can be defined that show up either in the Pandoc PDF output or on the web, and thus each of these parts can be written differently depending on how each markdown processor will use them.

This solution is at best horribly inelegant, requiring extra HTML elements and significant duplicated source just because the markdown processor of Pandoc is different than that of Hugo.

Solution Option 3: Pandoc Filters

There is lots of information on Pandoc filters written in Python, php, Lua, etc. online, and this solution also seemed to the most elegant and flexible. This solution would require creating a filter called, for example, `filter.lua` that would handle the offending code when using Pandoc using the option `--lua-filter=filter.lua` .

A few minutes were spent looking at the filters, but I realized it might be a bit involved and so this option was set aside to come back to after seeing if there may be more simpler options. The Pandoc Filters docs were a useful reference. As filters were an interesting option, I was particularly interested in Pandoc Lua Filters as they seemed to be frequently used with success and I'd enjoyed working with Lua before. Filters can also be written in Python and used with Panflute, as described in the blog post Technical Writing with Pandoc and Panflute.

Solution Option 4: Replace Offending Argument with sed

As the current problem was limited to one particular case (the occurrence of `{linenos=false}`) the stream editor sed could be easily used to look through the markdown source, replace offending code, and plumb the output into Pandoc. This way was very fast to understand and implement, more elegant than the HTML/CSS hacking above, and seemed somewhat flexible although it was far less elegant than ideal. But

in order to ensure parts of the code could be specified for removal (and not occurrences in the text, like `{linenos=false}`) a ` ` was tacked on. Then sed and Pandoc can be run as follows.

```
$ sed 's///g' documentation.md | pandoc \
  -o documentation.pdf
```

The result is the following, no line numbers for the desired code blocks on the web, and correctly rendered PDF output from Pandoc.

````
```js {linenos=false}  
var test
```
````

Of course the doesn't enable the code block line numbers to be selectively turned off in the Pandoc output.

It's not the most elegant solution as it requires remembering to include a superfluous ` ` after each `{linenos=false}` (or having such commands be removed everywhere) and as such will not scale well depending on how many other commands are relied upon on in the future that are not compatible with Pandoc, and adds a bit of an ugly extra step to the pandoc script. But for now it's a decent solution for this single occurrence with low overhead. Should I revisit this later to come up with a better solution, I can simply `grep` my notes and remove this ` ` or just leave it in there, as it doesn't really hurt anything. And yet again, it is unlikely that markdown from this site will be given to Pandoc anyway.

Styling Pandoc Output

As mentioned in Styling above, using a downloaded template such as with `--template eisvogel` produced a PDF output that looked pretty decent. However further customization was needed.

Styling Fonts

The first was to be able to customize the fonts in the PDF output. The Pandoc options below can be used to easily change fonts.

```
-V mainfont="SFNS Display" \
-V monofont="Menlo Regular" \
```

In addition a Latex header, for example `headings.tex` , can be used for more specific customization.

```
\newfontfamily\sfnsdisplaybold{SFNS Display Bold}
\sectionfont{\fontsize{16pt}{16pt}\selectfont\sfnsdisplaybold}
```

Custom fonts can be added and used as well. On Mac, fonts are in `~/Library/Fonts` . The header above can be used with `--include-in-header ~/.pandoc/tex-headers/headings.tex` To see what fonts are installed, the following command can be used:

```
fc-list | grep "SF-Pro-Text-Regular"
```

The San Francisco font, for example, can be downloaded in `.ttf` here.

Styling Code

Listings

First option is to use the `--listings` option with Pandoc. This made block code look quite nice, but the in-line code I was not happy with. The use of the `--listings` option put inline code in `lstinline` and block code in a `lstlisting` environment. The easiest way to see this was by just outputting `.tex` document from Pandoc. This is also a great way to get visibility into the intermediate step of generating PDF from markdown, and see how to best apply Latex headers and style the intermediate Latex source. This could even then be altered and output generated with whatever Latex engine, in this case I was using XeLatex.

The solution when using `--listings` was to use a latex header to style these environment(s) as desired. For example, made `listings-code.tex` and included with

```
--include-in-header ~/.pandoc/tex-headers/listings-code.tex
```

as with our header for heading fonts.

The contents of this header which are included in the linked `.pandoc` repo were overly complex. The details will not be covered here, but some helpful references were:

- A gist with a bunch of options for `lstset` : LaTex settings for embedding Python with Monokai theme.
- StackExchange answer: How to redefine \lstinline to automatically highlight or draw frames around all inline code snippets?
- StackExchange answer: Adding background color to \verb or \lstinline command without \Colorbox
- StackExchange answer: Avoid line breaks after \lstinline
- StackExchange answer: Colored background in inline listings

The result in its current form was acceptable, although not great - the inline code just didn't look quite right. This could be improved with the Latex header, but it was already much more complicated than desired. I chose to consider an approach without using the listings package to style code. When not using the `--listings` option, inline code is in `texttt{}` and a custom `Shaded` and `Highlighted` environment for block code.

Highlight Style: Inline Code

The inline code can be easily styled via the included Latex header by putting it into a `\colorbox` and setting the font size and style as desired. Not much else needs to be done to achieve a satisfactory result for inline code. This can be seen in the linked `.pandoc` repository at the end of this post.

Highlight Style: Block Code

The block code can be styled with a Pandoc style, which caan be exported, modified, and used as:

```
--highlight-style ~/.pandoc/themes/pygments-mod.theme
```

This provides a very basic styling only to the Highlight environment. This is well described in the Docs: Syntax highlighting. This is probably not as good of an option as using listings (when writing in Latex natively, this has been my prefered approach for styling block code) but is the prefered approach for now due to its simplicity. To further style the block code, the `fvextra` package can be used. `fvset` can be used to apply options to verbatim environment used for block code, allowing the addition of line numbers, background color, margins, and more to be set. With this, a simple Latex header can style the inline and block code as desired, without the complications when using the listings package.

Styling Hyperlinks

To specify the hyperlink color in the generated PDF, another Latex header can be used:

```
--include-in-header ~/.pandoc/tex-headers/link-color.tex
```

Another issue when generating a PDF was with HTML links in the markdown source. Remember from setting up this site that HTML links were used to enable opening the link in a new tab. A Lua filter was used to handle HTML links from markdown document and keep the links in the generated PDF. Fortunately, a suitable filter was found in the Stack Overflow answer HTML-formatted hyperlinks not preserved in bookdown PDF, so it did not need to be written from scratch. There were also similar filters using Panflute as described here: How to convert markdown link to html using Pandoc.

Conclusions

Hopefully the above provided a useful discussion of some of the problems encountered when using markdown on the web and for generating PDFs with Pandoc, describing the options and styles used to achieve the desired results in both of these mediums. Below a short list is provided of the items from setting up this site and this document that make up the current solution and workflow:

- Specify all links with HTML using the attribute `target="_blank"` .

- When the link text is a valid hyperlink itself, break it up inside with an empty `<span></span>`. `bash <a href="https://gohugo.io/" target="_blank">htt<span></span>ps://gohugo.io/</a>`
- Use MathJax to render Latex.
- When omitting line numbers in block code in Hugo, follow the argument with a character such as a non-breaking space: `{linenos=false}  ` and use sed before Pandoc to remove this argument that Pandoc doesn't understand.
- Make sure to not use relative links for anything that will end up in a PDF, or the relative link will be broken.
- Use simple Latex headers to style document text, headers, hyperlinks, code, and more.
- Use Lua filters to accommodate HTML in markdown with Pandoc.
- Run Pandoc with desired options via a script, e.g. `~/.pandoc/md2pdf.sh`.

In the future, will need to accommodate (likely via Lua filters) additional HTML in markdown when generating PDFs with Pandoc. For example when including images using the HTML `<img>` tag. The following is included to see how it will look in the generated PDF.

$$\rho\left(\frac{\partial \underline{v}}{\partial t}+\underline{v}\cdot\underline{\nabla v}\right)=-\underline{\nabla} p+\mu\nabla^2\underline{v}+\rho\underline{g}$$

$$\rho\frac{D\underline{v}}{Dt}=-\underline{\nabla} p+\underline{\nabla}\cdot\underline{\underline{\sigma}}+\rho\underline{g}$$

$$\rho\frac{D\underline{v}}{Dt}=-\underline{\nabla} p+\mu\nabla^2\underline{v}+\rho\underline{g}$$

```
```js {linenos=false}
$ var your = "code here"
```
```

Resources

- The PDF generated exactly from the markdown source used to create this page is available: documentation.pdf
- My `.pandoc` directory, including all Pandoc options, Latex headers, and Lua filters is available: https://github.com/dpwiese/.pandoc