Books.jl

Create books with Julia

Rik Huijzer

Contents

1 About

Similar to Bookdown this package is, basically, a wrapper around Pandoc. For websites, this package allows for:

- Building a website spanning multiple pages.
- Live reloading the website to see changes quickly; thanks to Pandoc and LiveServer.jl.
- Cross-references from one web page to a section on another page.
- Embedding dynamic output, while still allowing normal Julia package utilities, such as unit testing and live reloading (Revise.jl).
- Showing code blocks as well as output.

If you don't need PDFs or EPUBs, then Franklin.jl is probably a better choice. To create single pages and PDFs containing code blocks, see Weave.jl.

One of the main differences with Franklin.jl, Weave.jl and knitr (Bookdown) is that this package completely decouples the computations from the building of the output. The benefit of this is that you can spawn two separate processes, namely the one to serve your webpages:

```
$ julia --project -e 'using Books; serve()'
Watching ./pandoc/favicon.png
Watching ./src/plots.jl
[...]✓
  LiveServer listening on http://localhost:8001/ ...
  (use CTRL+C to shut down)
```

and the one where you do the computations for your package:

```
$ julia --project -ie 'using Books'

julia> gen()
[...]
Updating html
```

This way, the website remains responsive when the computations are running. Thanks to LiveServer.jl and Pandoc, updating the page after changing text or code takes less than a second. Also, because the `serve` process does relatively few things, it almost never crashes.

As another benefit, the decoupling allows you to have more flexiblity in when you want to run what code. In combination with Revise.jl, you can quickly update your code and see the updated output.

Finally, a big difference with this package and other packages is that you decide yourself what you want to show for a code block. For example, in R

shows the code and not the output. Instead, in Books, you would write which is displayed as

```
print("Hello, world!")
```

Here, `sc` is one of the convenience methods exported by Books.jl. Although this approach is more verbose in some cases, it is also much more flexible. In essence, you can come up with your own pre- or post-processing logic. For example, lets write

which shows the code and output (`sco`) 4 times:

```
df = DataFrame(a=[1, 2], b=[3, 4])
Options(df, caption="A table", label=nothing)
```

Table 1.1: A table

a	b
1	3
2	4

```
df = DataFrame(a=[1, 2], b=[3, 4])
Options(df, caption="A table", label=nothing)
```

Table 1.2: A table

a	b
1	3
2	4

```
df = DataFrame(a=[1, 2], b=[3, 4])
Options(df, caption="A table", label=nothing)
```

Table 1.3: A table

a	b
1	3
2	4

```
df = DataFrame(a=[1, 2], b=[3, 4])
Options(df, caption="A table", label=nothing)
```

Table 1.4: A table

a	b
1	3
2	4

2 Getting started

The easiest way to get started is to

1. copy over the files in docs/
2. step inside that directory and
3. serve your book via:

```
$ julia --project -e 'using Books; serve()'
Watching ./pandoc/favicon.png
Watching ./src/plots.jl
[...]✓
  LiveServer listening on http://localhost:8001/ ...
  (use CTRL+C to shut down)
```

To generate all the Julia output (see Section 3.1 for more information) use

```
$ julia --project -e  'using Books; using MyPackage; M = MyPackage'

julia> gen()
[...]
Updating html
```

To avoid code duplication between projects, this package tries to have good defaults for many settings. For your project, you can override the default settings by creating `config.toml` and `metadata.yml` files. In summary, the `metadata.yml` file is read by Pandoc while generating the outputs. This file contains settings for the output appearance, author and more, see Section 2.1. The `config.toml` file is read by Books.jl before calling Pandoc, so contains settings which are essentially passed to Pandoc, see Section 2.2. Still, these defaults can be overwritten. If you also want to override the templates, then see Section 2.3.

2.1 metadata.yml

The `metadata.yml` file is read by Pandoc. Settings in this file affect the behaviour of Pandoc and get inserted in the templates. For more info on templates, see Section 2.3. You can override settings by placing a `metadata.yml` file at the root directory of your project. For example, the metadata for this project contains:

```
---
title: Books.jl
subtitle: Create books with Julia
book: false
```

```
author: Rik Huijzer
pdf-footer: https://github.com/rikhuijzer/Books.jl

# An example additional header include for html.
# Note that the url will be updated by \`Books.fix_links\`.
header-includes:
- |
  \`\`\`{=html}
  <link rel="stylesheet" href="/files/style.css"/>
  \`\`\`
mousetrap: true
---
```

The following defaults are set by Books.jl.

```
---
title: My book
subtitle: My book subtitle
author: John Doe

# This affects things as whether to include a white page before each chapter in the PDF.
# See the `.tex` template for more information.
# For reports, set `book: false`.
book: true

# Licenses; can be empty.
html-license: <a href="http://creativecommons.org/licenses/by-sa/4.0/">CC BY-SA 4.0</a>
tex-license: Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

pdf-footer: "\\url{https://github.com/johndoe/Book.jl}"

# Make margins a bit smaller. LaTeX has huge default margins.
geometry:
  - top=20mm
  - left=24mm
  - right=24mm
  - bottom=28mm

# A setting for the PDF. I don't know whether it is important.
lang: en-US

tags: [pandoc, Books.jl, JuliaLang]
number-sections: true

code-block-font-size: \scriptsize

titlepage: true
linkReferences: true
bibliography: bibliography.bib
link-citations: true

# These table of contents settings only affect the PDF.
toc: true
```

```
toc-depth: 2

# Cross-reference prefixes.
eqnPrefix: Equation
figPrefix: Figure
tblPrefix: Table
secPrefix: Section

# Keyboard shortcuts.
mousetrap: true
---
```

2.2 config.toml

The `config.toml` file is used by Books.jl. Settings in this file affect how Pandoc is called. In `config.toml` , you can define multiple projects; at least define `projects.default`. The settings of `projects.default` are used when you call `pdf()` or `serve()`. To use other settings, for example the settings for `dev`, use `pdf(project="dev")` or `serve(project="dev")`.

Below, the default configuration is shown. When not defining a `config.toml` file or omitting any of the settings, such as `port`, these defaults will be used. You don't have to copy all these defaults, only *override* the settings that you want to change. The benefit of multiple projects is, for example, that you can run a `dev` project locally which contains more information than the `default` project. One example could be where you write a paper, book or report and have a page with some notes.

The meaning of `contents` is discussed in Section 2.2.1. The `pdf_filename` is used by `pdf()` and the `port` setting is used by `serve()`. For this documentation, the following config is used

```
[projects]

  [projects.default]
  contents = [
    "about",
    "getting-started",
    "demo",
    "references",
  ]
  output_filename = "books"

  # Prefix for GitHub Pages.
  online_url_prefix = "/Books.jl"

  # Extra directories to be copied.
  extra_directories = [
    "images",
    "files"
  ]

  port = 8012
  [projects.notes]
  contents = [
    "demo",
    "notes",
    "references"
  ]
```

Which overrides some settings from the following default settings

```
[projects]

  # Default project, used when calling serve() or pdf().
  [projects.default]
  homepage_contents = "index"

  metadata_path = "metadata.yml"

  contents = [
    "introduction",
    "analysis",
    "references"
  ]

  # Output pdf or docx filename.
  output_filename = "analysis"

  # Prefix for GitHub or GitLab Pages.
  online_url_prefix = ""

  # Port used by serve().
  port = 8010

  # Extra directories to be copied from the project root into `_build/`.
  extra_directories = []

  # For large books, it can be nice to show some information on the homepage
  # which is only visible to online visitors and hidden from offline users (PDF).
  include_homepage_outside_html = false



  # Alternative project, used when calling, for example, serve(project="dev").
  [projects.dev]
  homepage_contents = "index"

  metadata_path = "metadata.yml"

  contents = [
    "introduction",
    "analysis",
    "notes",
    "references"
  ]
```

```
output_filename = "analysis-with-notes"

port = 8011

extra_directories = []

include_homepage_outside_html = false
```

Here, the `extra_directories` allows you to specify directories which need to be moved into `_build`, which makes them available for the local server and online. This is, for instance, useful for images like Figure 2.1:

Figure 2.1: Book store.

2.2.1 About contents

The files listed in `contents` are read from the `contents/` directory and passed to Pandoc in the order specified by this list. It doesn't matter whether the files contain headings or at what levels the heading are. Pandoc will just place the texts behind each other.

This list doesn't mention the homepage for the website. That one is specified on a per project basis with `homepage_contents`, which defaults to `index`. The homepage typically contains the link to the generated PDF. Note that the homepage is only added to the html output and not to pdf or other outputs.

2.3 Templates

Unlike `metadata.yml` and `config.toml`, the default templates should be good for most users. To override these, create one or more of the files listed in Table 2.1.

Table 2.1: Default templates.

File	Description	Affects
`pandoc/style.csl`	citation style	all outputs
`pandoc/style.css`	style sheet	website
`pandoc/template.html`	HTML template	website
`pandoc/template.tex`	PDF template	PDF

Here, the citation style defaults to APA, because it is the only style that I could find that correctly supports parenthetical and in-text citations. For example,

- in-text: Orwell (1945)
- parenthetical: (Orwell, 1945)

For other citation styles from the citation-style-language, users have to manually specify the author in the in-text citations.

3 Demo

We can refer to a section with the normal pandoc-crossref syntax. For example,

> See Section 2.

We can refer to citations such as Orwell (1945) and (Orwell, 1949) or to equations such as Equation 3.1.

$$y = \frac{\sin x}{\cos x} \tag{3.1}$$

3.1 Embedding output

For embedding code, you can use the `jl` inline code or code block. For example, to show the Julia version, define a code block like

in a Markdown file. Then, in your package, define the method `julia_version()`:

```
M.julia_version() = "This book is built with Julia $VERSION."
```

Next, ensure that you call `using Books; gen(; M)`, where `M = YourModule`. Alternatively, if you work on a large project and want to only generate the output for one or more Markdown files in `contents/`, such as `index.md`, use

```
gen("index")
```

```
Writing output of `M.homepage_intro()` to _gen/M_homepage_intro_.md
Updating html
```

Calling `gen` will place the text

```
This book is built with Julia 1.6.1.
```

at the right path so that it can be included by Pandoc. You can also embed output inline with single backticks like

```
`jl julia_version()`
```

or just call Julia's constant `VERSION` directly from within the Markdown file:

```
This book is built with Julia `jl string(VERSION)`.
```

This book is built with Julia 1.6.1.

While doing this, it is expected that you also have the browser open and a server running, see Section 2. That way, the page is immediately updated when you run `gen`.

Note that it doesn't matter where you define the function `julia_version`, as long as it is in your module. To save yourself some typing, and to allow yourself to get some coffee while Julia gets up to speed, you can start Julia for your package with

```
$ julia --project -ie 'using Books; using MyPackage; M = MyPackage'
```

which allows you to re-generate all the content by calling

```
julia> gen()
```

To run this method automatically when you make a change in your package, ensure that you loaded Revise.jl before loading your package and run

```
entr(gen, ["contents"], [M])
```

where M is the name of your module. Which will automatically run `gen()` whenever one of the files in `contents/` changes or any code in the module `M`. In the background, `gen` passes the methods through `convert_output(expr::String, path, out::T)` where `T` can, for example, be a DataFrame or a plot. To show that a DataFrame is converted to a Markdown table, we define a method

```
my_table() = DataFrame(U = [1, 2], V = [:a, :b], W = [3, 4])
```

and add its output to the Markdown file with

Then, it will show as

Table 3.1: My table.

U	V	W
1	a	3
2	b	4

where the caption and the label are inferred from the `path`. Refer to Table 3.1 with

```
@tbl:my_table
```

> Table 3.1

To show multiple objects, pass a `Vector`:

```
multiple_df_vector() =
    [DataFrame(Z = [3]), DataFrame(U = [4, 5], V = [6, 7])]
multiple_df_vector()
```

Z
3

U	V
4	6
5	7

When you want to control where the various objects are saved, use `Options`. This way, you can pass a informative path with plots for which informative captions, cross-reference labels and image names can be determined.

```
function multiple_df_example()
    objects = [
        DataFrame(X = [3, 4], Y = [5, 6]),
        DataFrame(U = [7, 8], V = [9, 10])
    ]
    filenames = ["a", "b"]
    Options.(objects, filenames)
end
multiple_df_example()
```

Table 3.4: A.

X	Y
3	5
4	6

Table 3.5: B.

U	V
7	9
8	10

To define the labels and/or captions manually, see Section 3.2. For showing multiple plots, see Section 3.4.

Most things can be done via functions. However, defining a struct is not possible, because `@sco` cannot locate the struct definition inside the module. Therefore, it is also possible to pass code and specify that you want to evaluate and show code (sc) without showing the output:

```
struct Point
    x
    y
end
```

and show code and output (sco). For example,

shows as

```
p = Point(1, 2)
```

```
Point(1, 2)
```

Note that this is starting to look a lot like R Markdown where the syntax would be something like

I guess that there is no perfect way here. The benefit of evaluating the user input directly, as Books.jl is doing, seems to be that it is more extensible if I'm not mistaken. Possibly, the reasoning is that R Markdown needs to convert the output directly, whereas Julia's better type system allows for converting in much later stages, but I'm not sure.

> **Tip**: When using `sco`, the code is evaluated in the `Main` module. This means that the objects, such as the `Point` struct defined above, are available in your REPL after running `gen()`.

3.2 Labels and captions

To set labels and captions, wrap your object in `Options`:

```
options_example() = Options(DataFrame(A = [1], B = [2], C = [3]);
                        caption="My DataFrame.", label="foo")
options_example()
```

Table 3.6: My DataFrame.

A	B	C
1	2	3

which can be referred to with

```
@tbl:foo
```

> Table 3.6

It is also possible to pass only a caption or a label. This package will attempt to infer missing information from the `path`, `caption` or `label` when possible:

```
julia> Books.caption_label("foo_bar()", missing, missing)
```

```
(caption = "Foo bar.", label = "foo_bar")

julia> Books.caption_label("foo_bar()", "My caption.", missing)
(caption = "My caption.", label = "foo_bar")

julia> Books.caption_label("foo_bar()", "My caption.", nothing)
(caption = "My caption.", label = nothing)

julia> Books.caption_label(missing, "My caption.", missing)
(caption = "My caption.", label = nothing)

julia> Books.caption_label(missing, missing, "my_label")
(caption = "My label.", label = "my_label")

julia> Books.caption_label(missing, missing, missing)
(caption = nothing, label = nothing)
```

3.3 Obtaining function definitions

So, instead of passing a string which `Books.jl` will evaluate, `Books.jl` can also obtain the code for a method directly. (Thanks to `CodeTracking.@code_string`.) For example, inside our package, we can define the following method:

```
function my_data()
    DataFrame(A = [1, 2], B = [3, 4], C = [5, 6], D = [7, 8])
end
```

To show code and output (sco) for this method, use the `@sco` macro. This macro is exported by Books, so ensure that you have `using Books` in your package.

This gives

```
function my_data()
    DataFrame(A = [1, 2], B = [3, 4], C = [5, 6], D = [7, 8])
end
my_data()
```

Table 3.7: My data.

A	B	C	D
1	3	5	7
2	4	6	8

To only show the source code, use `@sc`:

resulting in

```
function my_data()
    DataFrame(A = [1, 2], B = [3, 4], C = [5, 6], D = [7, 8])
end
```

Since we're using methods as code blocks, we can use the code shown in one code block in another. For example, to determine the mean of column A:

shows as

```
function my_data_mean(df::DataFrame)
    Statistics.mean(df.A)
end
my_data_mean(my_data())
```

1.5

Or, we can show the output inline, namely 1.5, by using

```
`jl M.my_data_mean(my_data())`
```

It is also possible to show methods with parameters. For example,

shows

```
hello(name) = "Hello, $name"
```

Now, we can show

```
M.hello("World")
```

```
Hello, World
```

Here, the `M` can be a bit confusing for readers. If this is a problem, you can export the method `hello` to avoid it. If you are really sure, you can export all symbols in your module with something like this.

3.4 Plots

An AlgebraOfGraphics plot is shown below in Figure 3.1. For Plots.jl and Makie.jl see, respectively section Section 3.4.1 and Section 3.4.2. This is actually a bit tricky, because we want to show vector graphics (SVG) on the web, but these are not supported (well) by LaTeX. Therefore, portable network graphics (PNG) images are also created and passed to LaTeX.

```
function example_plot()
    I = 1:30
    df = (x=I, y=I.^2)
    xy = data(df) * mapping(:x, :y)
    fg = draw(xy)
end
example_plot()
```

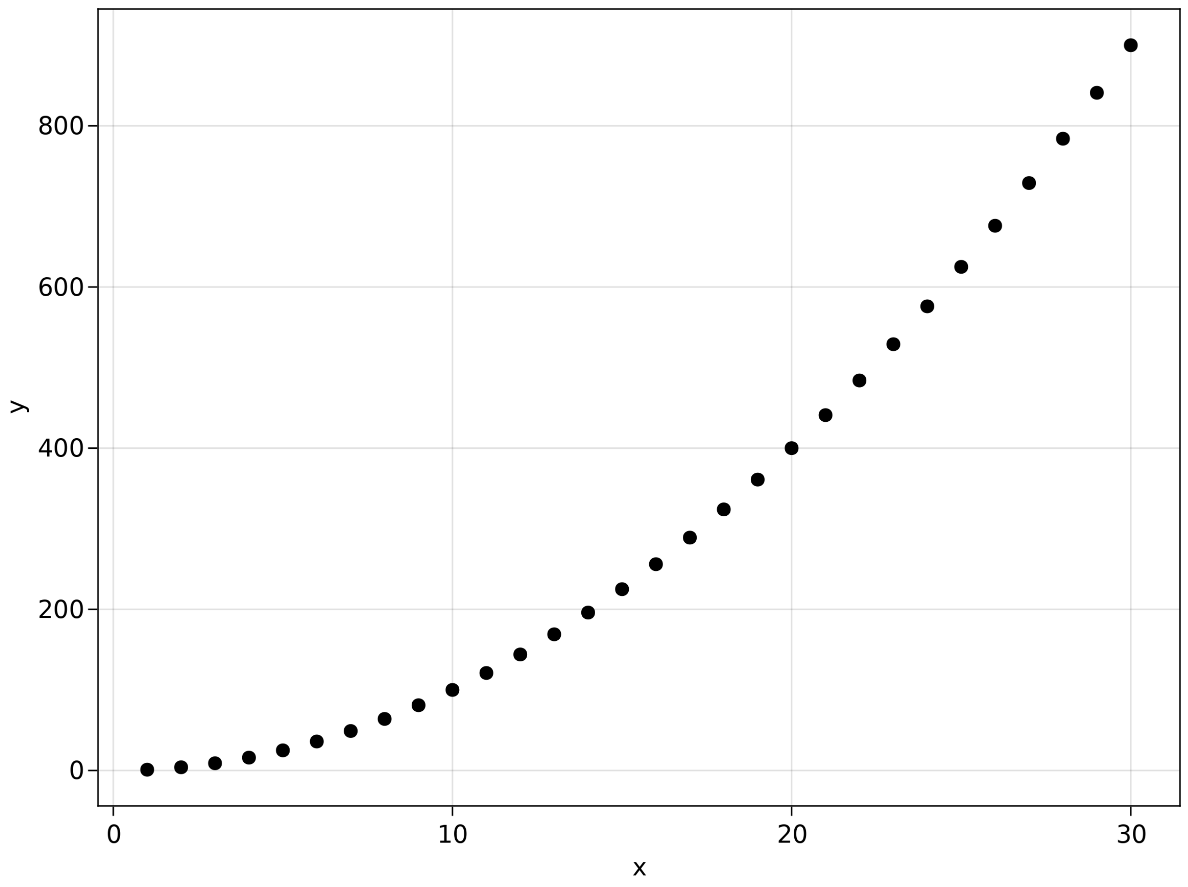

Figure 3.1: Example plot.

If the output is a string instead of the output you expected, then check whether you load the related packages in time. For example, for this plot, you need to load AlgebraOfGraphics.jl together with Books.jl so that Requires.jl will load the code for handling AlgebraOfGraphics objects.

For multiple images, use `Options.(objects, paths)`:

```
function multiple_example_plots()
    filenames = ["example_plot_$i" for i in 2:3]
    I = 1:30
    df = (x=I, y=I.*2, z=I.^3)
    objects = [
        draw(data(df) * mapping(:x, :y))
        draw(data(df) * mapping(:x, :z))
    ]
    Options.(objects, filenames)
end
```

Resulting in Figure 3.2 and Figure 3.3:

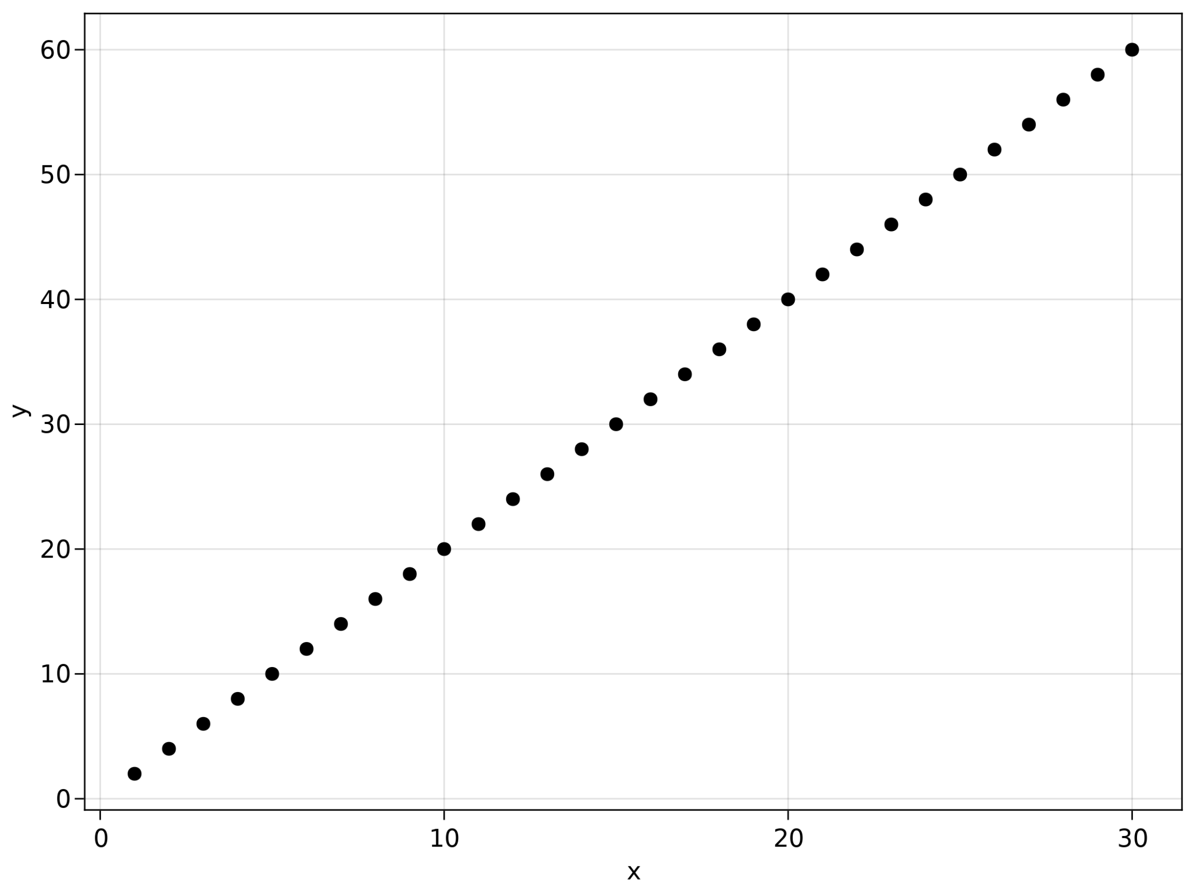

Figure 3.2: Example plot 2.

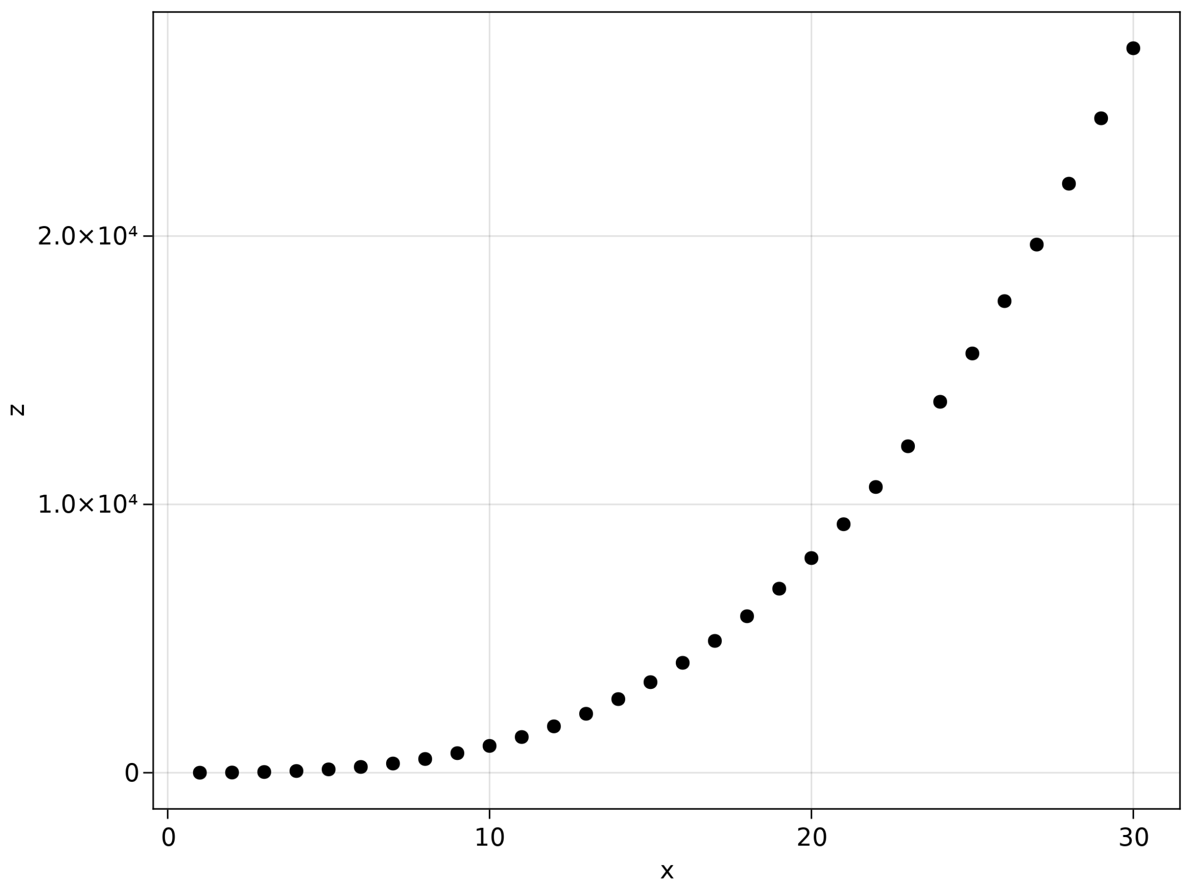

Figure 3.3: Example plot 3.

For changing the size, use `axis` from AlgebraOfGraphics:

```
function image_options_plot()
    I = 1:0.1:30
    df = (x=I, y=sin.(I))
    xy = data(df) * visual(Lines) * mapping(:x, :y)
    axis = (width = 600, height = 140)
    draw(xy; axis)
end
image_options_plot()
```

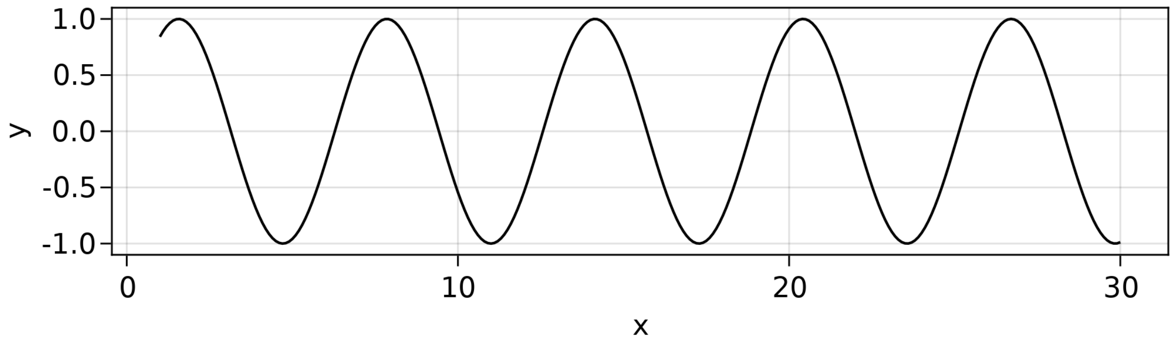

Figure 3.4: Image options plot.

And, for adjusting the caption, use `Options`:

```
function combined_options_plot()
    fg = image_options_plot()
    Options(fg; caption="Sine function.")
end
combined_options_plot()
```

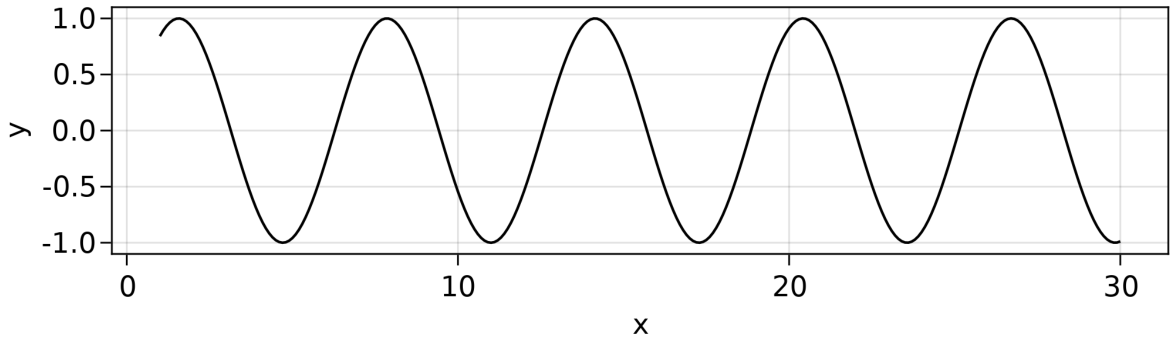

Figure 3.5: Sine function.

or the caption can be specified in the Markdown file:

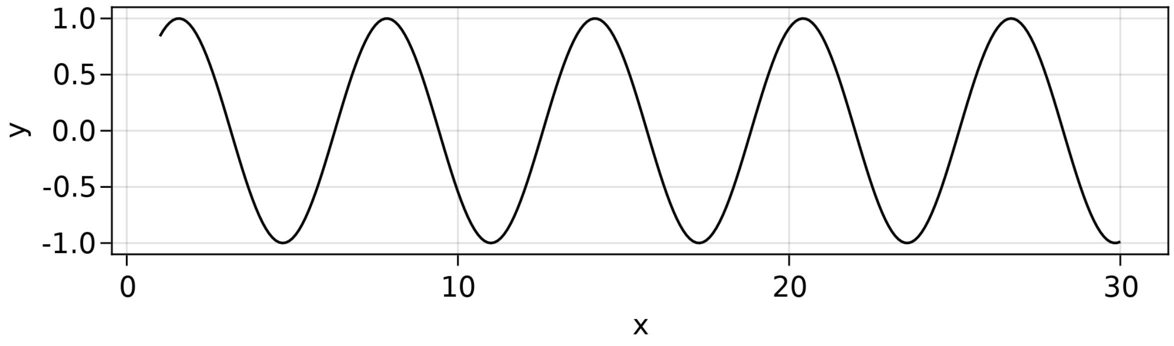

Figure 3.6: Label specified in Markdown.

3.4.1 Plots

```
function plotsjl()
    p = plot(1:10, 1:2:20)
    caption = "An example plot with Plots.jl."
    # Label default to `nothing`, which will not create a cross-reference.
    label = missing
    Options(p; caption, label)
end
plotsjl()
```

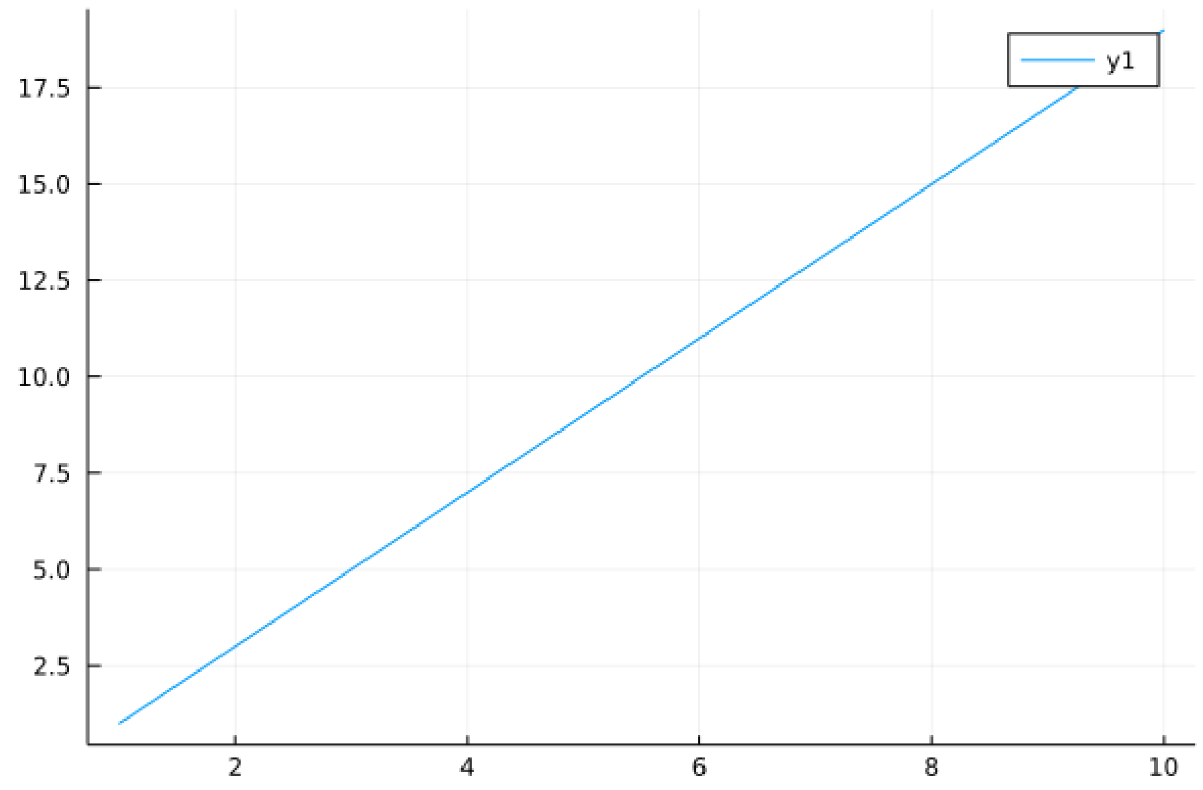

Figure 3.7: An example plot with Plots.jl.

3.4.2 Makie

```
function makiejl()
    x = range(0, 10, length=100)
    y = sin.(x)
    p = lines(x, y)
    caption = "An example plot with Makie.jl."
    label = missing
    Options(p; caption, label)
end
makiejl()
```

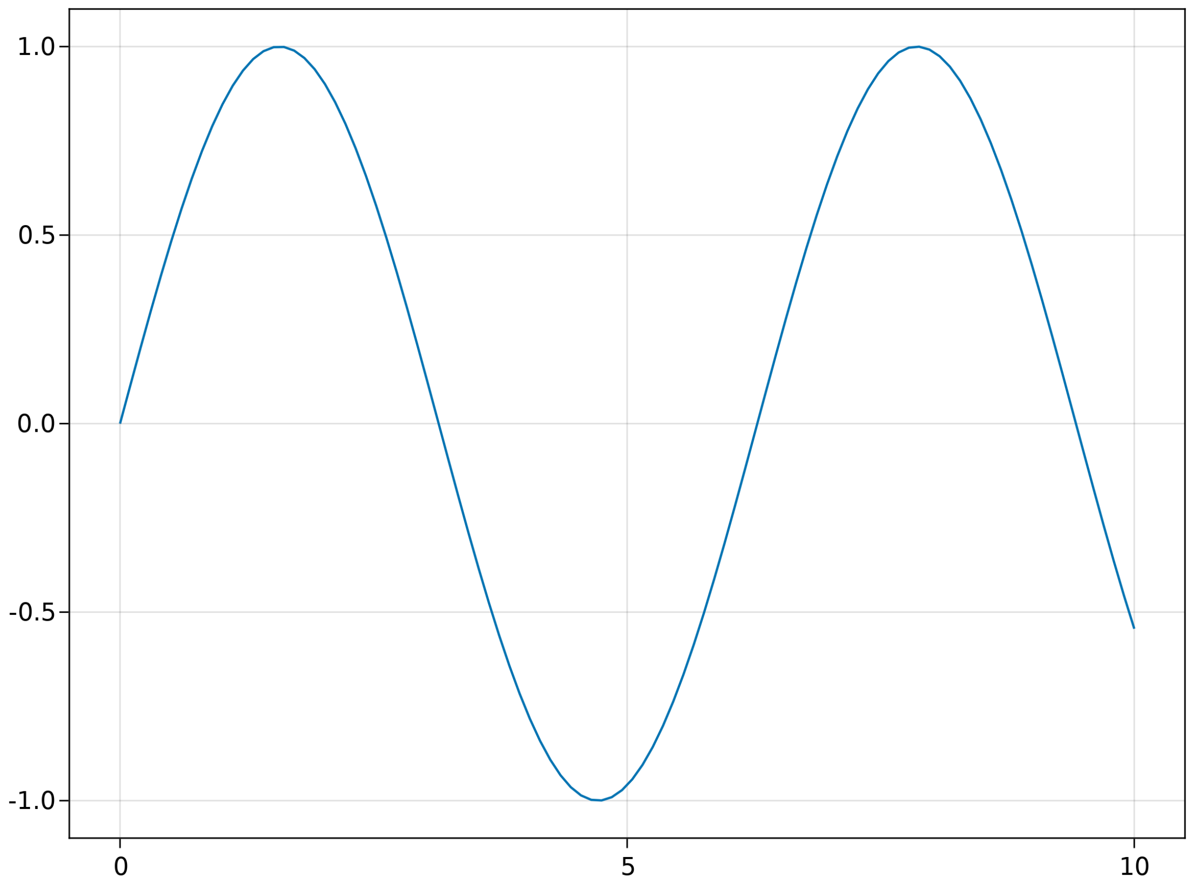

Figure 3.8: An example plot with Makie.jl.

3.5 Other notes

3.5.1 Multilingual books

For an example of a multilingual book setup, say English and Chinese, see the book by Jun Tian.

3.5.2 Show

When your method returns an output type `T` which is unknown to Books.jl, it will be passed through `show(io::IO, ::MIME"text/plain", object::T)`. So, if the package that you're using has defined a new `show` method, this will be used. For example, for `MCMCChains`,

```
chain() = MCMCChains.Chains([1])
chain()
```

```
Chains MCMC chain (1×1×1 Array{Int64, 3}):

Iterations        = 1:1:1
Number of chains  = 1
Samples per chain = 1
```

```
parameters        = param_1

Summary Statistics
  parameters      mean       std   naive_se      mcse       ess      rhat
      Symbol   Float64   Float64    Float64   Missing   Missing   Missing

     param_1    1.0000       NaN        NaN   missing   missing   missing

Quantiles
  parameters      2.5%     25.0%     50.0%     75.0%     97.5%
      Symbol   Float64   Float64   Float64   Float64   Float64

     param_1    1.0000    1.0000    1.0000    1.0000    1.0000
```

3.5.3 Note box

To write note boxes, you can use

```
> **_NOTE:_**  The note content.
```

> ***NOTE:*** The note content.

This way is fully supported by Pandoc, so it will be correctly converted to outputs such as PDF or DOCX.

3.5.4 Advanced `sco` options

To enforce output to be embedded inside a code block, use `scob`. For example,

```
scob("
df = DataFrame(A = [1], B = [Date(2018)])
string(df)
")
```

```
df = DataFrame(A = [1], B = [Date(2018)])
string(df)
```

```
1×2 DataFrame
 Row │ A      B│
      Int64  Date─────┼────────────────────

   1 │      1  2018-01-01
```

or, with a string

```
s = "Hello"
```

```
Hello
```

Another way to change the output is via the keyword arguments `process` and `post` for `sco`.

which shows the following to the reader:

```
df = DataFrame(A = [1], B = [Date(2018)])
```

```
1×2 DataFrame
 Row │ A      B│
      Int64  Date─────┼──────────────────────


   1 │     1  2018-01-01
```

Without `process=string`, the output would automatically be converted to a Markdown table by Books.jl and then wrapped inside a code block, which will cause Pandoc to show the raw output instead of a table.

```
df = DataFrame(A = [1], B = [Date(2018)])
```

```
|   A |          B |
| ---:| ----------:|
|   1 | 2018-01-01 |
```

Without `post=output_block`, the DataFrame would be converted to a string, but not wrapped inside a code block so that Pandoc will treat is as normal Markdown:

```
df = DataFrame(A = [2], B = [Date(2018)])
```

Options(1×2 DataFrame Row ⊠ A B ⊠ Int64 Date ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠ 2 2018-01-01, missing, nothing, nothing)

This also works for `@sco`. For example, for `my_data` we can use:

which will show as:

```
function my_data()
    DataFrame(A = [1, 2], B = [3, 4], C = [5, 6], D = [7, 8])
end
my_data()
```

```
2×4 DataFrame
 Row │ A      B      C      D│
      Int64  Int64  Int64  Int64─────┼─────────────────────────────


   1 │     1      3      5      7
   2 │     2      4      6      8
```

3.5.5 Fonts

The code blocks default to JuliaMono in html and PDF. Ligatures from JuliaMono are disabled. For example, none of these symbols are combined into a single glyph.

```
|> => and <=
```

References

Orwell, G. (1945). *Animal farm: a fairy story*. Houghton Mifflin Harcourt.

Orwell, G. (1949). *Nineteen eighty-four: a novel*. Secker & Warburg.