Books.jl

Create books with Julia

Rik Huijzer

and contributors

Table of Contents

# About

Similar to [Bookdown](https://bookdown.org), this package wraps around [Pandoc](https://pandoc.org/). 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](https://github.com/tlienart/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 to generate PDFs, then [Franklin.jl](https://github.com/tlienart/Franklin.jl) is probably a better choice. To create single pages and PDFs containing code blocks, see [Weave.jl](https://github.com/JunoLab/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

```{r, results='hide'}  
print("Hello, world!")  
```

shows the code and not the output. Instead, in Books, you would write

```jl  
s = """print("Hello, world!")"""  
sc(s)  
```

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

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

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

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

A table.

|  |  |
| --- | --- |
| a | b |
| 1 | 3 |
| 2 | 4 |

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

A table.

|  |  |
| --- | --- |
| a | b |
| 1 | 3 |
| 2 | 4 |

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

A table.

|  |  |
| --- | --- |
| a | b |
| 1 | 3 |
| 2 | 4 |

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

A table.

|  |  |
| --- | --- |
| a | b |
| 1 | 3 |
| 2 | 4 |

# Getting started

The easiest way to get started is to use the template repository at <https://github.com/JuliaBooks/BookTemplate>.

From this repository, you can 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](#sec:embedding-output) for more information) use

$ julia --project -e 'using Books; using MyPackage; M = MyPackage'  
  
julia> gen()  
[...]  
Updating html

where MyPackage is the name of your package. 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](#sec:metadata). 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](#sec:config). Still, these defaults can be overwritten. If you also want to override the templates, then see Section [2.3](#sec:templates).

To generate the PDF, use

julia> pdf()

## 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](#sec:templates). 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  
author:  
 - Rik Huijzer  
 - and contributors  
  
# 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  
  
#  
# PDF only settings.  
#  
pdf-footer: ""  
  
# Avoid adding a blank page before each chapter.  
disable-cleardoublepage: true  
  
bibliography: bibliography.bib  
  
titlepage-top: >  
 \begin{tabular}{l}  
 Rik Huijzer\\  
 University of Groningen\\  
 the Netherlands\\  
 t.h.huijzer@rug.nl\\  
 \end{tabular}  
  
titlepage-bottom: >  
 \url{https://books.huijzer.xyz}  
---

And, the following defaults are set by Books.jl.

---  
title: My book  
subtitle: My book subtitle  
author:  
 - John Doe  
  
# Licenses; can be empty.  
html-license: <a href="http://creativecommons.org/licenses/by-nc-sa/4.0/">CC BY-NC-SA 4.0</a>  
tex-license: Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International  
  
pdf-footer: "\\url{https://github.com/johndoe/Book.jl}"  
  
geometry:  
 # Defaulting to large pages to be able to show 92 chars in code blocks.  
 - paperheight=10in # 25.4 cm.  
 - paperwidth=7in # 17.78 cm.  
 - marginparwidth=30mm  
 - marginparsep=2mm  
 - bindingoffset=10mm  
 - top=10mm  
 - inner=8mm  
 - outer=8mm  
 - bottom=16mm  
 - includehead  
 - includemp  
  
# A setting for the PDF. I don't know whether it is important.  
lang: en-US  
  
# Uncomment the following to enable a Chinese/Japanese/Korean (CJK) font.  
# CJKmainfont: Noto Serif CJK SC  
  
links-as-notes: true  
  
tags: [pandoc, Books.jl, JuliaLang]  
number-sections: true  
  
code-block-font-size: \scriptsize  
  
titlepage: true  
linkReferences: true  
link-citations: true  
  
# These table of contents settings only affect the PDF.  
toc: true  
tocdepth: 1  
  
# Cross-reference prefixes.  
eqnPrefix: Equation  
figPrefix: Figure  
tblPrefix: Table  
secPrefix: Section  
  
# Keyboard shortcuts.  
mousetrap: true  
  
header-left: foobar  
---

Note that Pandoc has a great templating system. For example, the [Pandoc LaTeX template “default.latex”](https://github.com/jgm/pandoc) contains hundreds of options which can all be set via “metadata.yml.” In the template of this project, I tried to get the basics right and haven’t spend hours on making it configurable. However, if you want to add options to the templates, feel free to open a pull request. If you want to configure the template completely by yourself, you can place “template.tex” in “pandoc/” at the root of your project[[1]](#footnote-28). This same holds for the HTML, CSS, and DOCX template.

## 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](#sec:about_contents). 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"  
  
 # Full URL, required for the sitemap and robots.txt.  
 online\_url = "https://books.huijzer.xyz"  
  
 # Extra directories to be copied.  
 extra\_directories = [  
 "images",  
 "files"  
 ]  
  
 port = 8012  
  
 [projects.notes]  
 contents = [  
 "demo",  
 "notes",  
 "references"  
 ]  
  
 # This project is only used when testing Books.jl.  
 [projects.test]  
 contents = [  
 "test"  
 ]  
  
 online\_url = "https://example.com"  
 online\_url\_prefix = "/Example.jl"

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"  
  
 # Full URL, required for the sitemap.  
 online\_url = "https://example.com"  
  
 # 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  
  
 # Syntax highlighting.  
 highlight = true  
  
 # 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 [1](#fig:store):

![Book store.](images/book-store.jpg){#fig:book\_store}

shows as



Figure 1: Book store.

### 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.

### Website landing page

By default, Books.jl assumes that you will want a separate landing page for your book when you host it. This page is not added to the generated ouputs, like PDF or DOCX, so it’s a nice place to put links to the generated outputs. You will need to create a index.md file in the contents directory. Then, using an top-level header from Markdown (e.g. “# Title”), give the file a title. Immediately after the title, you need to write {-} to avoid this chapter showing up in your HTML menu.

Here is an example of how an example index.md file looks like:

# My Book's Awesome Title! {-}  
  
Welcome to the landing page for my awesome book!

## 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 [1](#tbl:templates).

Table 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](#ref-orwell1945animal) ([1945](#ref-orwell1945animal))
* parenthetical: ([Orwell, 1945](#ref-orwell1945animal))

For other citation styles from the [citation-style-language](https://github.com/citation-style-language/styles), users have to manually specify the author in the in-text citations.

# Demo

We can refer to a section with the normal [pandoc-crossref](https://lierdakil.github.io/pandoc-crossref/) syntax. For example,

See @sec:getting-started.

See Section [2](#sec:getting-started).

We can refer to citations such as [Orwell](#ref-orwell1945animal) ([1945](#ref-orwell1945animal)) and ([Orwell, 1949](#ref-orwell1949nineteen)) or to equations such as Equation [1](#eq:example).

## 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

```jl  
M.julia\_version()  
```

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()`  
Updating html

Calling gen will place the text

This book is built with Julia 1.6.6.

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. For example,

This book is built with Julia `jl VERSION`.

This book is built with Julia 1.6.6.

While doing this, it is expected that you also have the browser open and a server running, see Section [2](#sec:getting-started). 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](https://github.com/timholy/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. To only run gen for one file, such as “contents/my\_text.md,” use:

mygen() = gen("my\_text")  
entr(mygen, ["contents"], [M])

With this, mygen will be called every time something changes in one of the files in the contents folder or when something changes in your module M. Note that you have to run this while serve is running in another terminal in the background. Then, your Julia code is executed and the website is automatically updated every time you change something in “content” or your 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

```jl  
M.my\_table()  
```

Then, it will show as

Table 2: 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 [2](#tbl:my_table) with

@tbl:my\_table

Table [2](#tbl:my_table)

To show multiple objects, pass a Vector:

function multiple\_df\_vector()  
 [DataFrame(Z = [3]), DataFrame(U = [4, 5], V = [6, 7])]  
end  
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: A.

|  |  |
| --- | --- |
| X | Y |
| 3 | 5 |
| 4 | 6 |

Table 4: B.

|  |  |
| --- | --- |
| U | V |
| 7 | 9 |
| 8 | 10 |

To define the labels and/or captions manually, see Section [3.2](#sec:labels-captions). For showing multiple plots, see Section [3.4](#sec:plots).

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:

```jl  
s = """  
 struct Point  
 x  
 y  
 end  
 """  
sc(s)  
```

which shows as

struct Point  
 x  
 y  
end

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

```jl  
sco("p = Point(1, 2)")  
```

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

```{r, results='hide'}  
x = rnorm(100)  
```

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().

## Labels and captions

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

function options\_example()  
 df = DataFrame(A=[1], B=[2], C=[3])  
 caption = "My DataFrame."  
 label = "foo"  
 return Options(df; caption, label)  
end  
options\_example()

Table 5: My DataFrame.

|  |  |  |
| --- | --- | --- |
| A | B | C |
| 1 | 2 | 3 |

which can be referred to with

@tbl:foo

Table [5](#tbl:foo)

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)

## 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.

```jl  
@sco M.my\_data()  
```

This gives

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

Table 6: My data.

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

To only show the source code, use @sc:

```jl  
@sc M.my\_data()  
```

resulting in

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

To override options for your output, use the pre keyword argument of @sco:

```jl  
let  
 caption = "This caption is set via the pre keyword."  
 pre(out) = Options(out; caption)  
 @sco pre=pre my\_data()  
end  
```

which appears to the reader as:

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

This caption is set via the pre keyword.

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

See ?sco for more information. 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:

```jl  
@sco M.my\_data\_mean(my\_data())  
```

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,

```jl  
@sc M.hello("" )  
```

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](https://discourse.julialang.org/t/exportall/4970/16).

## Plots

An AlgebraOfGraphics plot is shown below in Figure [2](#fig:example_plot). For Plots.jl and Makie.jl see, respectively section Section [3.4.1](#sec:jlplots) and Section [3.4.2](#sec:makie). 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()

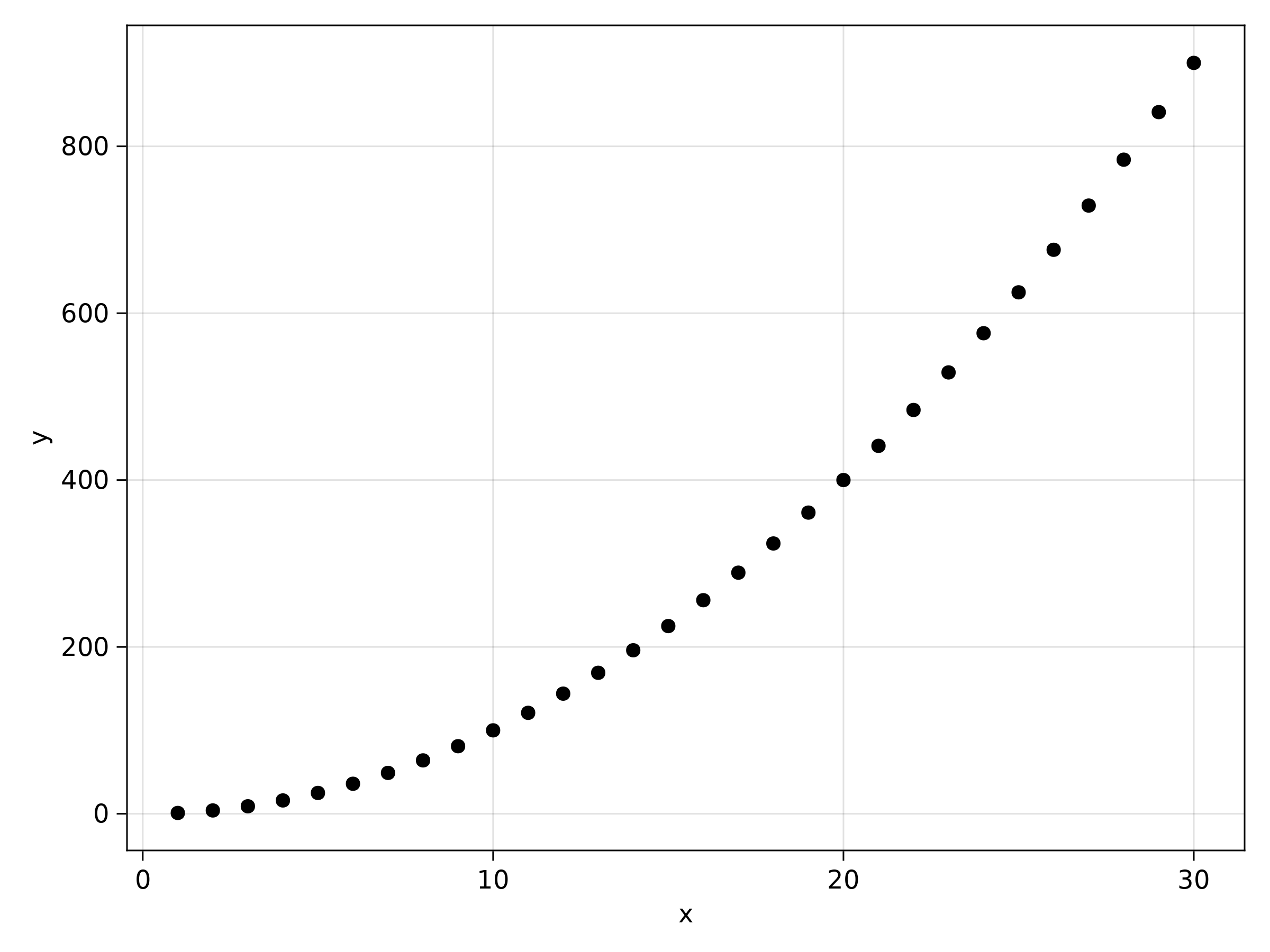


Figure 2: Example plot.

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))  
 ]  
 return Options.(objects, filenames)  
end

Resulting in Figure [3](#fig:example_plot_2) and Figure [4](#fig:example_plot_3):

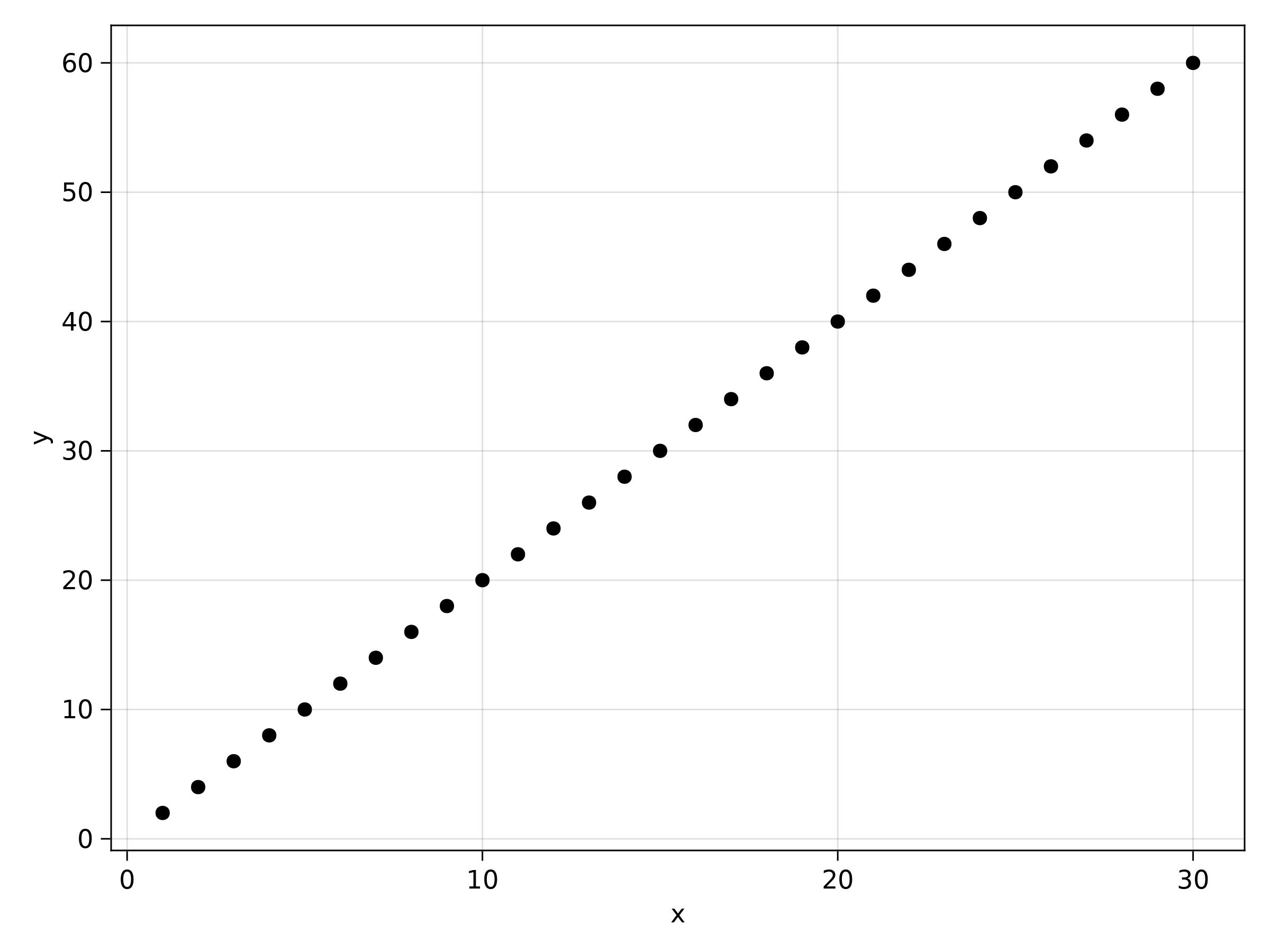


Figure 3: Example plot 2.

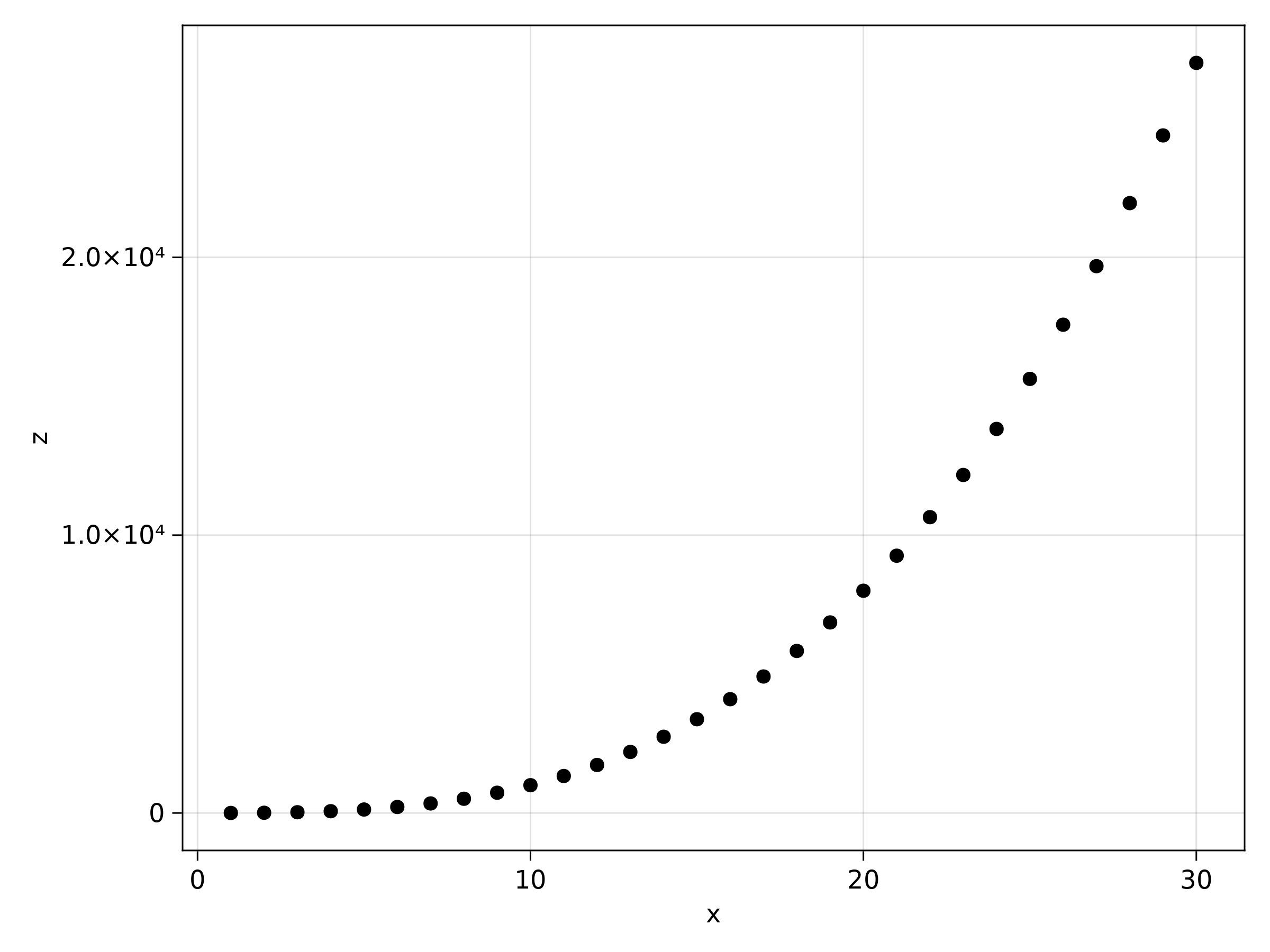


Figure 4: Example plot 3.

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()

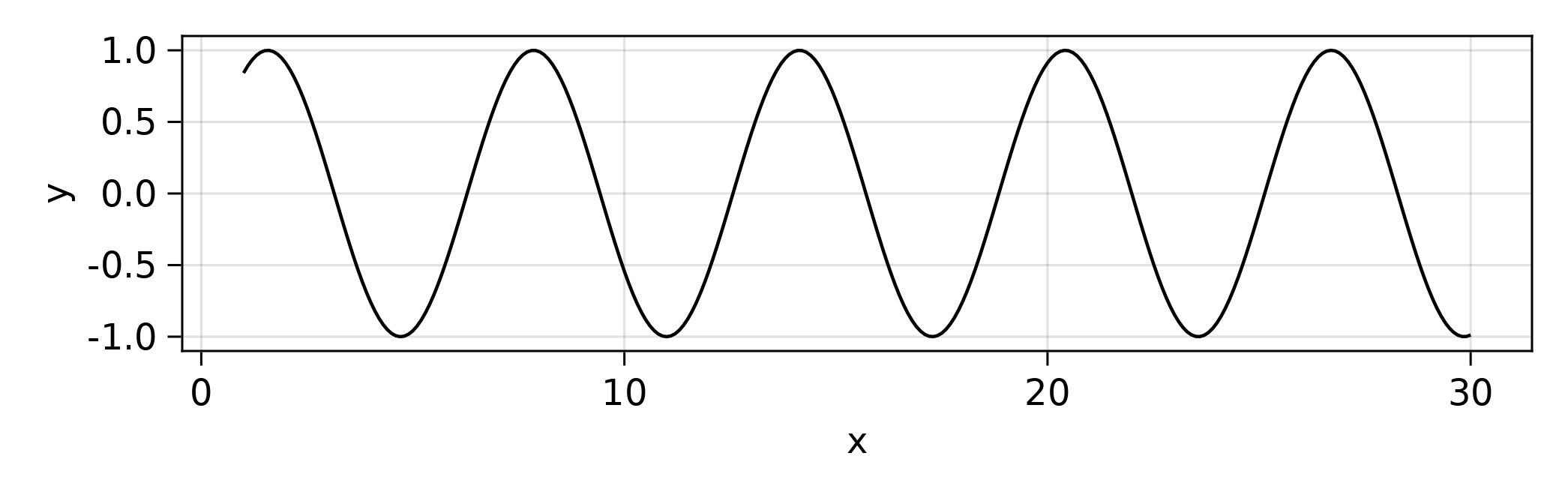
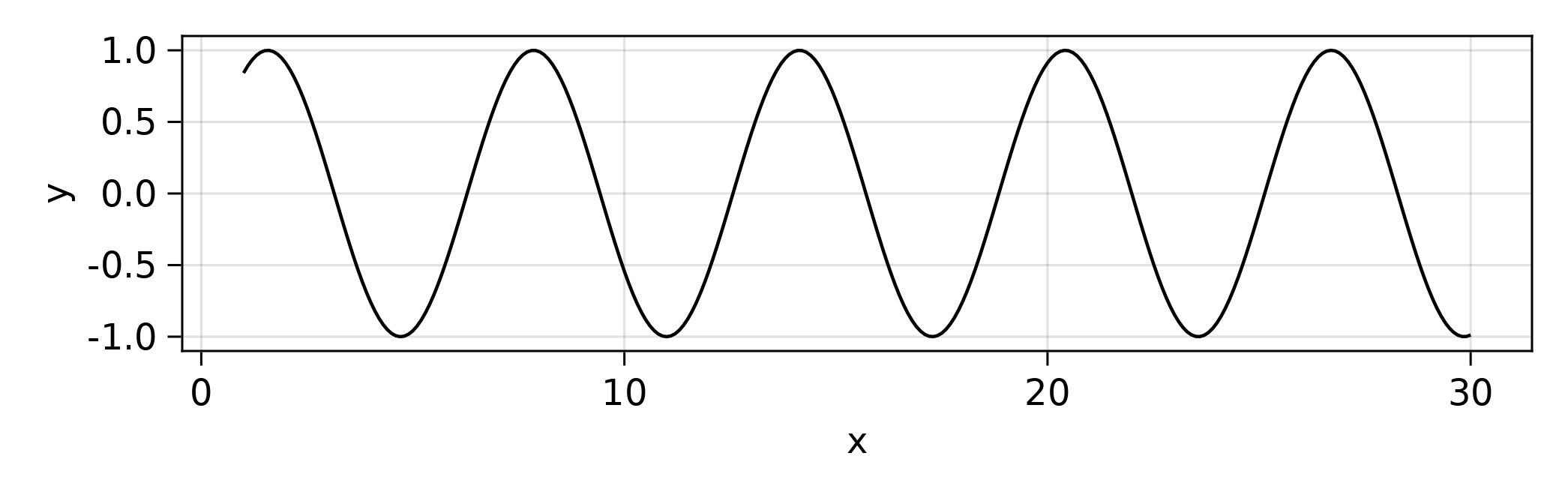


Figure 5: Image options plot.

And, for adjusting the caption, use Options:

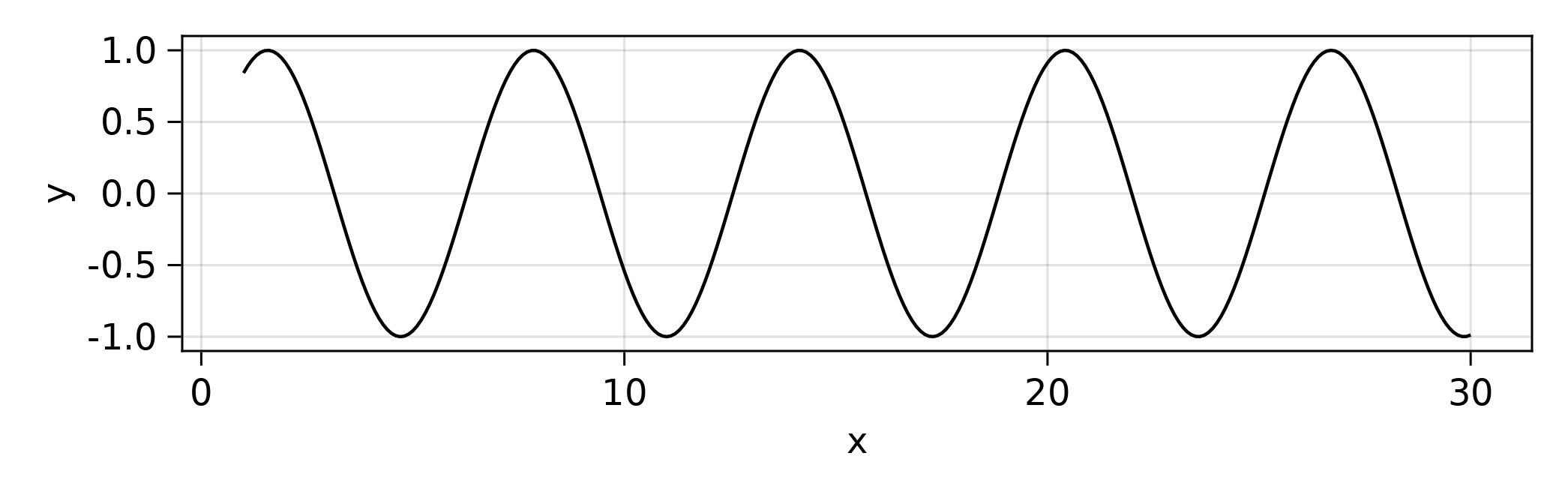
function combined\_options\_plot()  
 fg = image\_options\_plot()  
 Options(fg; caption="Sine function.")  
end  
combined\_options\_plot()



Sine function.

or the caption can be specified in the Markdown file:

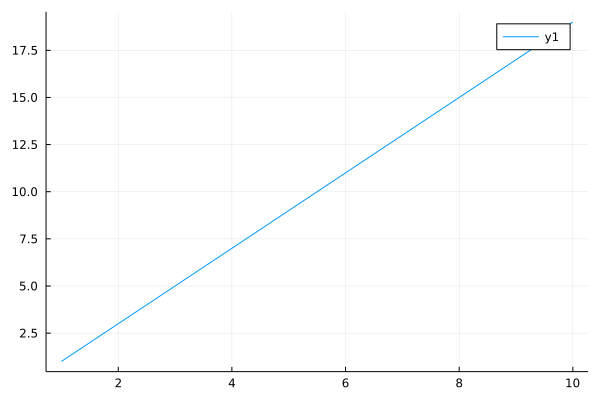
```jl  
p = M.image\_options\_plot()  
Options(p; caption="Label specified in Markdown.")  
```



Label specified in Markdown.

### Plots.jl

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



An example plot with Plots.jl.

### Makie

This time, we also pass link\_attributes to Pandoc (Figure [6](#fig:makie)):

function makiejl()  
 x = range(0, 10, length=100)  
 y = sin.(x)  
 p = lines(x, y)  
 caption = "An example plot with Makie.jl."  
 label = "makie"  
 link\_attributes = "width=70%"  
 Options(p; caption, label, link\_attributes)  
end  
makiejl()

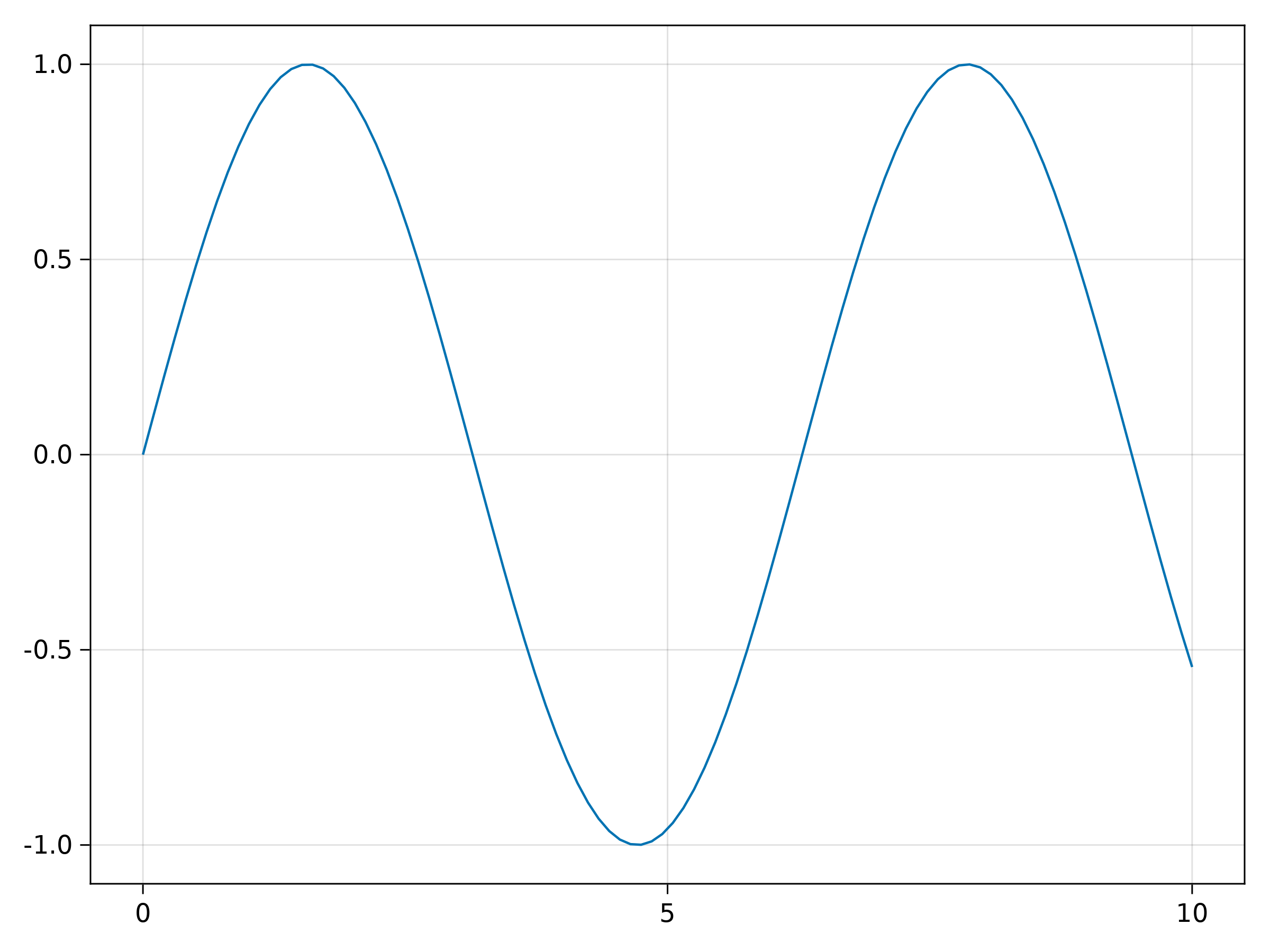


Figure 6: An example plot with Makie.jl.

## Other notes

### Multilingual books

For an example of a multilingual book setup, say English and Chinese, see the book by [Jun Tian](https://github.com/LearnJuliaTheFunWay/LearnJuliaTheFunWay.jl).

### Footnotes

Footnotes can be added via regular Markdown syntax:

Some sentence[^foot].  
  
[^foot]: Footnote text.

Some sentence[[2]](#footnote-69).

### 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

### 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.

### 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 pre, process and post for sco. The idea of these arguments is that they allow you to pass a function to alter the processing that Books.jl does. pre is applied **before** Books.convert\_output, process is applied **instead** of Books.convert\_output and post is applied **after** Books.convert\_output. For example, to force books to convert a DataFrame to a string instead of a Markdown table, use:

```jl  
s = "df = DataFrame(A = [1], B = [Date(2018)])"  
sco(s; process=string, post=output\_block)  
```

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, missing)

This also works for @sco. For example, for my\_data we can use:

```jl  
@sco process=string post=output\_block my\_data()  
```

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

### Fonts

The code blocks default to JuliaMono in HTML and PDF. For the HTML, this package automatically handles JuliaMono. However, for the PDF, this just doesn’t work out (see, e.g., [257](https://github.com/JuliaBooks/Books.jl/pull/257)). To get JuliaMono to work with the PDF build, install it globally. See the instructions at the [JuliaMono site](https://juliamono.netlify.app/download/#installation). On Linux, you can use Books.install\_extra\_fonts(), but beware that it might override user settings.

Ligatures from JuliaMono are disabled. For example, none of these symbols are combined into a single glyph.

|> => and <=

### Long lines in code blocks

When code or output is getting too long, a horizontal scrollbar is visible on the website to scroll horizontally.

### Code blocks in lists

To embed code blocks inside lists, indent by 3 spaces and place an empty line before and after the code block. For example, this will show as:

1. This is a list item with some code and output:

* x = 2 + 1
* 3

1. And the list continues
   * with an example on the third level:
   * x = 3 + 1
   * 4
   * another third level item

# References

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

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

1. With LaTeX, be ready for hours of fiddling to get things right though. The current LaTeX template is adjusted to work with Tectonic, Julia syntax highlighting and the Tufte LaTeX class. [↑](#footnote-ref-28)
2. Footnote text. [↑](#footnote-ref-69)