Latexify.jl

Latexify.jl is a package which supplies functions for producing $\LaTeX$ formatted strings from Julia objects. The package allows for latexification of a many different kinds of Julia object and it can output several different $\LaTeX$ or Markdown environments.

A small teaser:

```
using Latexify
Latexify.set_default(; starred=true)
m = [2//3 "e^(-c*t)" 1+3im; :(x/(x+k_1)) "gamma(n)" :(log10(x))]
latexify(m)
```

$$\begin{bmatrix} \frac{2}{3} & e^{(-c) \cdot t} & 1 + 3i \\ \frac{x}{x + k_{1}} & \Gamma\left(n \right) & \log_{10}\left(x \right) \end{bmatrix}$$

Supported input

This package supplies functionality for latexifying objects of the following types:

- Expressions,
- Strings,
- Numbers (including rational and complex),
- Missing,
- Symbols,
- Symbolic expressions from SymEngine.jl,
- DataFrame from DataFrames.jl,
- Any shape of array containing a mix of any of the above types,
- ParameterizedFunctions from DifferentialEquations.jl,
- ReactionNetworks from DifferentialEquations.jl

Example:

```
str = "x/(2*k_1+x^2)"
latexify(str)
```

$$\frac{x}{2 \cdot k_{1} + x^{2}}$$

Supported output

Latexify has support for generating a range of different $\LaTeX$ environments. The main function of the package, `latexify()`, automatically picks a suitable output environment based on the type(s) of the input. However, you can override this by passing the keyword argument `env =`. The following environments are available:

environment	`env=`	description
no env	`:raw`	Latexifies an object and returns a $\LaTeX$ formatted string. If the input is an array it will be recursed and all its elements latexified. This function does not surround the resulting string in any $\LaTeX$ environments.
Inline	`:inline`	latexify the input and surround it with $ for inline rendering.
Align	`:align`	Latexifies input and surrounds it with an align environment. Useful for systems of equations and such fun stuff.
Equation	`:equation` or `:eq`	Latexifies input and surrounds it with an equation environment.
Array	`:array`	Latexify the elements of an Array or a Dict and output them in a $\LaTeX$ array.
Tabular	`:table` or `:tabular`	Latexify the elements of an array and output a tabular environment. Note that tabular is not supported by MathJax and will therefore not be rendered in Jupyter, etc.
Markdown Table	`:mdtable`	Output a Markdown table. This will be rendered nicely by Jupyter, etc.
Markdown Text	`:mdtext`	Output and render any string which can be parsed into Markdown. This is really nothing but a call to `Base.Markdown.parse()`, but it does the trick. Useful for rendering bullet lists and such things.
Chemical arrow notation	`:chem`, `:chemical`, `:arrow` or `:arrows`	Latexify an AbstractReactionNetwork to $\LaTeX$ formatted chemical arrow notation using mhchem.

Modifying the output

Some of the different outputs can be modified using keyword arguments. You can for example transpose an array with `transpose=true` or specify a header of a table or mdtable with `header=[]`.

For more options, see the List of possible arguments.

Printing vs displaying

`latexify()` returns a LaTeXString. Using `display()` on such a string will try to render it.

```
latexify("x/y") |> display
```

$$\frac{x}{y}$$

Using `print()` will output text which is formatted for latex.

```
latexify("x/y") |> print
```

```
$\frac{x}{y}$
```

Number formatting

You can control the formatting of numbers by passing any of the following to the `fmt` keyword:

- a printf-style formatting string, for example `fmt = "%.2e"`.
- a single argument function, for example `fmt = x -> round(x, sigdigits=2)`.
- a formatter supplied by Latexify.jl, for example `fmt = FancyNumberFormatter(2)` (thanks to @simeonschaub). You can pass any of these formatters an integer argument which specifies how many significant digits you want.
  - `FancyNumberFormatter()` replaces the exponent notation, from `1.2e+3` to `1.2 \cdot 10^3`.
  - `StyledNumberFormatter()` replaces the exponent notation, from `1.2e+3` to `1.2 \mathrm{e} 3`.

Examples:

```
latexify(12345.678; fmt="%.1e")
```

$1.2e+04$

$$1.2e+04$$

```
latexify([12893.1 1.328e2; "x/y" 7832//2378]; fmt=FancyNumberFormatter(3))
```

$$\begin{bmatrix} 1.29 \cdot 10^{4} & 133 \\ \frac{x}{y} & \frac{3916}{1189} \end{bmatrix}$$

$$\begin{bmatrix} 1.29 \cdot 10^{4} & 133 \\ \frac{x}{y} & \frac{3916}{1189} \end{bmatrix} \tag{1}$$

```
using Formatting
latexify([12893.1 1.328e2]; fmt=x->format(round(x, sigdigits=2), autoscale=:metric))
```

$$\begin{bmatrix} 13k & 130 \end{bmatrix}$$

Automatic copying to clipboard

The strings that you would see when using print on any of the above functions can be automatically copied to the clipboard if you so specify. Since I do not wish to mess with your clipboard without you knowing it, this feature must be activated by you.

To do so, run

```
copy_to_clipboard(true)
```

To once again disable the feature, pass `false` to the same function.

The copying to the clipboard will now occur at every call to a Latexify.jl function, regardless of how you chose to display the output.

Automatic displaying of result

You can toggle whether the result should be automatically displayed. Instead of

```
latexify("x/y") |> display
## or
display( latexify("x/y") )
```

one can toggle automatic display by:

```
auto_display(true)
```

after which all calls to `latexify` will automatically be displayed. This can be rather convenient, but it can also cause a lot of unwanted printouts if you are using `latexify` in any form of loop. You can turn off this behaviour again by passing `false` to the same function.

Setting your own defaults

If you get tired of specifying the same keyword argument over and over in a session, you can just reset its default:

```
set_default(fmt = "%.2f", convert_unicode = false)
```

Note that this changes Latexify.jl from within and should therefore only be used in your own Julia sessions (do not call this from within your packages).

The calls are additive so that a new call with

```
set_default(cdot = false)
```

will not cancel out the changes we just made to `fmt` and `convert_unicode`.

To view your changes, use

```
get_default()
```

and to reset your changes, use

```
reset_default()
```

Macros

Three macros are exported.

- `@latexify` simply latexifies the expression that you provide to it, similar to `latexify(:(...))`.
- `@latexrun` both executes and latexifies the given expression.
- `@latexdefine` executes the expression, and latexifies the expression together with the its value

They can for example be useful for latexifying simple mathsy functions like

```
julia> lstr = @latexrun f(x; y=2) = x/y
L"$f\left( x; y = 2 \right) = \frac{x}{y}$"

julia> f(1)
0.5
```

```
julia> @latexdefine x = 1/2
L"$x = \frac{1}{2} = 0.5
```

```
julia> x
0.5
```

The arguments to the macro can be interpolated with `$` to use the actual value, instead of the representation:

```
julia> @latexify x = abs2(-3)
L"$x = \left|-3\right|^{2}$"

julia> @latexify x = $(abs2(-3))
L"$x = 9$"
```

Keyword arguments can be supplied after these macros:

```
julia> @latexdefine x env=:equation
L"\begin{equation}
x = 0.5
\end{equation}
"
```

A special keyword `post` can be supplied to `@latexdefine`, which is a function that will be called on the final right hand sign before latexification. This is merely formatting and will not affect any assignments.

```
julia> @latexdefine x=π  post=round
L"$x = \pi = 3.0$"

julia> @latexdefine x
L"$x = \pi$"
```

External rendering

While LaTeXStrings already render nicely in many IDEs or in Jupyter, they do not render in the REPL. Therefore, we provide a function `render(str)` which generates a standalone PDF using LuaLaTeX and opens that file in your default PDF viewer.

I have found the following syntax pretty useful:

```
latexify(:(x/y)) |> render
```

Alternatively, `render(str, mime)` can also be used to generate and display DVI, PNG and SVG files, which might be useful for other purposes:

```
latexify(:(x/y)) |> s -> render(s, MIME("image/png"))
```

PNG and SVG outputs rely on dvipng and dvisvgm, respectively.

If your code requires specific packages or document classes to render correctly, you can supply those as keyword arguments:

```
L"\qty{1.25}{nm}" |> render(s, MIME("image/png"); documentclass="article", packages=(
```

The arguments to these are either strings, or tuples of strings where the first one is the name of the package or class, and any further are optional arguments.

Legacy support

Latexify.jl has stopped supporting Julia versions older than 0.7. This does not mean that you cannot use Latexify with earlier versions, just that these will not get new features. Latexify.jl's release v0.4.1 was the last which supported Julia 0.6. Choose that release in the dropdown menu if you want to see that documentation.

Recipes »