

Overview

With computational modelling, it is important for not only the computer but also for the human to understand the model specification. Latexify.jl [3] is an open-source package which facilitates automatic transcription of computational objects in the Julia programming language [1] into renderable mathematical notation. It has built-in rules for how to transcribe a large range of types, including different numbers, symbols, arrays, dictionaries and even full expressions. These rules are composable such that composite types only need support for its individual parts in order for the whole to be transcribable. Furthermore, these rules are extendable via a recipe system that allows users and developers of other packages to add support to their own types. The combination of extensibility and composability has enabled the development of a full ecosystem of support allowing even complex NLME models to be automatically transcribed.

Immediate transcription and visualisation of mathematical objects facilitates not only report (and poster) generation but it also helps verify their validity by providing alternative and digestible representations. For example:

```
using Latexify
@Latexify f(x, y, z) = (x/z)^2 + y^2 < 1 ? (x/z)^2 + y^2 : 1
```

$$f(x, y, z) = \begin{cases} \left(\frac{x}{z}\right)^2 + y^2 & \text{if } \left(\frac{x}{z}\right)^2 + y^2 < 1 \\ 1 & \text{otherwise} \end{cases}$$

How it works

Latexify.jl defines a set of rules for how convert different Julia types – e.g. real and complex numbers, symbols, arrays and dictionaries – into latex-formatted code. These rules are recursive and composable such that this matrix of basic Julia types is neatly transcribed:

```
latexify([1 2, 3//4 5+6im
          π Inf true :y])
```

$$\begin{bmatrix} 1 & 2.0 & \frac{3}{4} & 5 + 6i \\ \pi & \infty & \text{true} & \gamma \end{bmatrix}$$

Since Julia parses all code into an expressions type, `Expr`, the rule for recursion through such expressions is particularly powerful. An `Expr` has a tree-like structure where every section of it describes what function or operator to apply to what arguments. Latexify.jl can transcribe such expressions by recursively applying simple rules for how each operation should be treated.

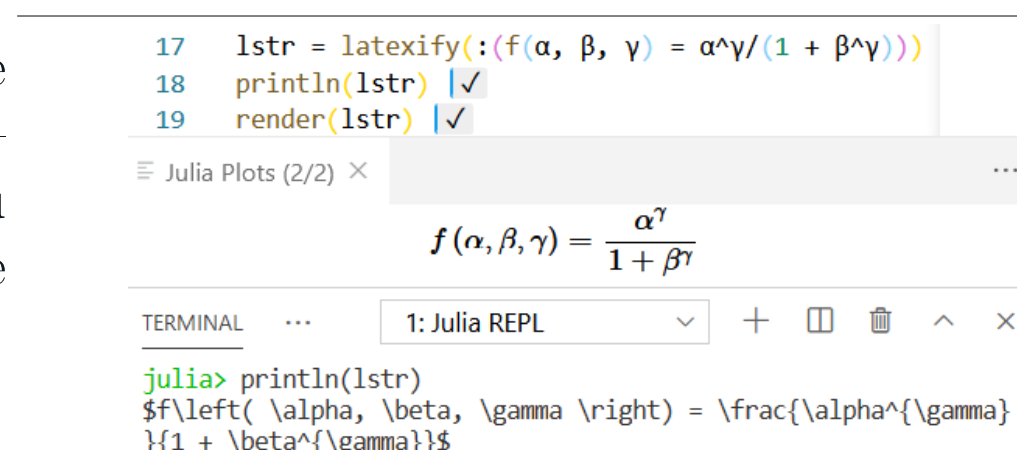
```
julia> ex = :(x / (y + z^2))
julia> ex.args
3-element Vector{Any}:
 :/
 :x
 :(y + z ^ 2)
```

```
latexify(ex)
```

$$\frac{x}{y + z^2}$$

Latexify returns a `LaTeXString` which can be printed, copied directly to your clipboard, or – in many environments – directly rendered as you code. An example using VSCode is shown on the right.

```
17 istr = latexify(:f(α, β, γ) = α^γ/(1 + β^γ))
18 println(istr) ✓
19 render(istr) ✓
```



Recipes for composability

Users and developers of other packages can easily extend Latexify.jl's functionality through a recipe system that enables complete control of how their own types should be transcribed.

```
struct PolarCoord
    r
    θ
end
@Latexrecipe function f(p::PolarCoord)
    cdot --> false
    r, θ = p.r, p.θ
    return :(r * exp(sθ * i))
end
M = [PolarCoord(1, 1/2) PolarCoord(2, 2)
     PolarCoord(3, 3/4) PolarCoord(4, π)]
latexify(M)
```

For example, the newly defined polar coordinate type to the left can be placed in a matrix and latexified to yield:

$$\begin{bmatrix} 1e^{0.5i} & 2e^{2i} \\ 3e^{0.75i} & 4e^{\pi i} \end{bmatrix}$$

Here, we see composition in two ways. First, the The composability of rules ensured that we did not need to define what to do with `exp` or the different number types we used. Nor did we have to define how to treat a matrix of `PolarCoords`. Furthermore, this composability spans across different packages.

An example of the ecosystem

The symbolical calculation package Symbolics.jl [2] defines a set of rules for latexifying its types.

```
using Latexify, Symbolics
@variables x, y
latexify((x+x)/(y*y))
```

$$\frac{2x}{y^2}$$

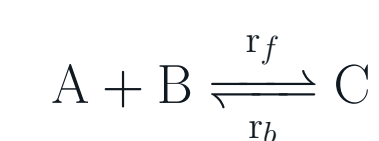
These Symbolics.jl types are then used in, for example, ModelingToolkit.jl [5] to generate efficient computational models. Since ModelingToolkit.jl can inherit the rules created by Symbolics.jl, it only needed one line of additional code to enable Latexify.jl support for its own types, enabling transcription of entire, runnable, models:

```
using ModelingToolkit
@variables t, x(t), y(t), z(t)
@parameters σ ρ β
D = Differential(t)
eqs = [D(D(x)) ~ σ*(y-x),
       D(y) ~ x*(ρ-z)-y,
       D(z) ~ x*y - β*z]
latexify(eqs)
```

$$\begin{aligned} \frac{d}{dt} \left(\frac{dx(t)}{dt} \right) &= \sigma (y(t) - x(t)) \\ \frac{dy(t)}{dt} &= x(t) (\rho - z(t)) - y(t) \\ \frac{dz(t)}{dt} &= x(t) y(t) - \beta z(t) \end{aligned}$$

The chain continues since Catalyst.jl [4] utilises ModelingToolkit.jl when it defines a framework for modelling chemical reaction networks. Much of the rules for transcribing the mathematics of a model is thus inherited and only some rules that are specific to chemical arrow notation was added to enable:

```
using Catalyst
rn = @reaction_network begin
    (r_f, r_b), A + B ↔ C
end r_f r_b
latexify(rn)
```



$$\begin{aligned} \frac{dA(t)}{dt} &= r_b C(t) - r_f A(t) B(t) \\ \frac{dB(t)}{dt} &= r_b C(t) - r_f A(t) B(t) \\ \frac{dC(t)}{dt} &= -r_b C(t) + r_f A(t) B(t) \end{aligned}$$

Use with Pumas.jl

Latexify.jl is especially useful for pharmacometricians through it's support of Pumas.jl. All parts of non-linear mixed effect models defined with the Pumas.jl `@model` macro are latexifiable.

```
using Pumas
model = @model begin
    ... # skipping some model blocks so that this fits on the poster.
    @pre begin
        CL = tvcl * (1 + (isPM == "yes" ? ispmocl : 0)) * (wt/70)^0.75 * exp(η[1])
        Vc = tvvc * (wt/70) * exp(η[2])
        Ka = isfed == "yes" ? tvkafed * exp(η[3]) : tvkafasted * exp(η[3])
        Q = tvq * (wt/70)^0.75 * exp(η[4])
        Vp = tvvp * (wt/70) * exp(η[5])
    end
    @dynamics Depots1Central1Periph1
    ...
end
```

```
latexify(model, :pre)
```

$$\begin{aligned} CL &= tvcl \cdot \left(1 + \begin{cases} ispmocl & \text{if } (isPM = yes) \\ 0 & \text{otherwise} \end{cases} \right) \cdot \left(\frac{wt}{70} \right)^{0.75} \cdot e^{\eta_1} \\ Vc &= tvvc \cdot \frac{wt}{70} \cdot e^{\eta_2} \\ Ka &= \begin{cases} tvkafed \cdot e^{\eta_3} & \text{if } (isfed = yes) \\ tvkafasted \cdot e^{\eta_3} & \text{otherwise} \end{cases} \\ Q &= tvq \cdot \left(\frac{wt}{70} \right)^{0.75} \cdot e^{\eta_4} \\ Vp &= tvvp \cdot \frac{wt}{70} \cdot e^{\eta_5} \end{aligned}$$

```
latexify(model, :dynamics)
```

$$\begin{aligned} \frac{dDepot(t)}{dt} &= -Ka \cdot Depot(t) \\ \frac{dCentral(t)}{dt} &= Ka \cdot Depot(t) - \frac{CL + Q}{Vc} \cdot Central(t) + \frac{Q}{Vp} \cdot Peripheral(t) \\ \frac{dPeripheral(t)}{dt} &= \frac{Q}{Vc} \cdot Central(t) - \frac{Q}{Vp} \cdot Peripheral(t) \end{aligned}$$

This automatic transcription of entire models not only provides copy-pastable mathematics for insertion to posters such as this one, it enables fully automatic report generation pipelines - either by your own design or through the report generation supplied by Pumas.jl. Furthermore, immediate visual feedback through rendering in a development environment makes is easier to double-check that model definitions are correct.

Conclusion and Discussion

Latexify.jl is an open-source Julia package that defines a framework for automatic transcription of computational models and other objects. Its functionality is easily extended by users or developers of other packages. The rules for how to transcribe objects compose recursively with one another which has enabled a powerful ecosystem of Latexify.jl support to grow out of simple rules defined by individual packages such as Symbolics.jl, ModelingToolkit.jl, Catalyst.jl and Pumas.jl. Latexify.jl has proven useful beyond its original intent to reduce time spent manually transcribing models for reports, etc. Automatic transcription reduces mistakes in model representations since it traces the very same expressions that will be used by your computer for simulations. If one spots an error in the Latexify.jl representation of a model it is likely that the error also exists in the computational model. Immediate rendering and visualisation of model equations during development thus helps to catch mistakes early on in the modelling process. Furthermore, automatic transcription and rendering has proven useful for developing GUI applications as well as enabling fully automated report generation in, for example, Pumas.jl.

Acknowledgements

I would like to thank my former PhD supervisor, Professor Henrik Jönsson at Cambridge University, in who's lab I initially developed Latexify.jl. I would also like to thank my current employer, Pumas-AI, for encouraging my continued open-source maintenance and development work. Also, thanks to all the people who have contributed to the open-source development of Latexify.jl.

References

- Jeff Bezanson et al. 'Julia: A fresh approach to numerical computing'. In: *SIAM review* 59.1 (2017), pp. 65–98. URL: <https://doi.org/10.1137/141000671>.
- Shashi Gowda et al. 'High-performance symbolic-numeric via multiple dispatch'. In: *arXiv preprint arXiv:2105.03949* (2021).
- Niklas Korsbo. *Latexify.jl*. Version 0.15.5. 2021. URL: <https://github.com/korsbo/Latexify.jl>.
- Torkel Loman et al. *Catalyst.jl*. Version 6.14.0. 2021. URL: <https://github.com/SciML/Catalyst.jl>.
- Yingbo Ma et al. 'ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling'. In: (2021). arXiv: 2103.05244 [cs.MS].