Journal of open research software

`Plots.jl` – A User Extendable Plotting API for the Julia Programming Language

SOFTWARE METAPAPER

SIMON CHRIST
DANIEL SCHWABENEDER
CHRISTOPHER RACKAUCKAS
MICHAEL KRABBE BORREGAARD
THOMAS BRELOFF

*Author affiliations can be found in the back matter of this article

ubiquity press

ABSTRACT

There are many excellent plotting libraries. Each excels at a specific use case: one is particularly suited for creating printable 2D figures for publication, another for generating interactive 3D graphics, while a third may have excellent LaTeX integration or be ideal for creating dashboards on the web. The aim of `Plots.jl` is to enable the user to use the same syntax to interact with a range of different plotting libraries, making it possible to change the library that does the actual plotting (the *backend*) without needing to touch the code that creates the content – and without having to learn multiple application programming interfaces (API). This is achieved by separating the specification of the plot from the implementation of the graphical backend. This plot specification is extendable by a *recipe* system that allows package authors and users to create new types of plots, as well as to specify how to plot any type of object (e.g. a statistical model, a map, a phylogenetic tree or the solution to a system of differential equations) without depending on the `Plots.jl` package. This design supports a modular ecosystem structure for plotting and yields a high code reuse potential across the entire Julia package ecosystem. `Plots.jl` is publicly available at https://github.com/JuliaPlots/Plots.jl.

CORRESPONDING AUTHOR:
Simon Christ
Leibniz Universität Hannover, DE
christ@cell.uni-hannover.de

KEYWORDS:
visualization; julia; plotting; julia-language; user-extendable

TO CITE THIS ARTICLE:
Christ S, Schwabeneder D, Rackauckas C, Borregaard MK, Breloff T 2023 `Plots.jl` – A User Extendable Plotting API for the Julia Programming Language. *Journal of Open Research Software*, 11: 5. DOI: https://doi.org/10.5334/jors.431

(1) OVERVIEW

INTRODUCTION

Julia [5] is a programming language that achieves high performance, stellar modularity and easy composability by making use of multiple dispatch and just-in-time compilation. This comes at the cost of increased latency for new function calls, as the language compiles new machine-code the first time any function is called on new types of arguments. This first call compilation time is a notorious issue for packages that call a large part of their codebase in the first call, such as plotting packages. It even coined the term *time to first plot (TTFP)* as a phrase for Julia's start-up latency. Indeed, the Julia language survey 2020 [34] identified *"it takes too long to generate the first plot"* as the biggest problem faced by Julia users.

Package authors try to minimize loading time by reducing the number of dependencies, in particular those that themselves have long loading times. Because depending on a plotting package drastically increases startup time, authors are faced with a challenge if they want to define new plotting functionality for their packages, e.g. if a package for differential equations wishes to define a specialized plotting method for differential equation solutions, to make it easy for users to investigate them visually. Furthermore, depending on a plotting package limits users to plotting with that particular package. If a project imports multiple packages that depend on different plotting packages, this may lead to conflicts and namespace clashes. As a consequence, depending on a plotting package is rarely seen in the Julia ecosystem. `Plots.jl` has solved this problem by introducing the concept of plot *recipes*, which allows package authors to define new types of plots while only depending on a very lightweight package `RecipesBase.jl`. For package developers this makes it easy and unproblematic to define `Plots.jl` recipes for any new types of objects they define in their package. This recipe will automatically support any `Plots.jl` backend without running the risk of package conflicts. From the users point of view, the support for multiple backends considerably lowers the threshold for use, as learning syntax and concepts of a new plotting framework is a significant time investment that many users are reluctant to make. With `Plots.jl`, the backend plotting package can be switched with a single function call, while the syntax and usage stays the same. Thus `Plots.jl` offers a unified and powerful API with a convenient way for package authors to support visualizations with multiple plotting packages, without increasing the loading time of their package, while at the same time offering users more choice and less cognitive load. An example of the convenient composability of `Plots.jl` and third party packages is given in Listing 9.

DEVELOPMENT

`Plots.jl` was created by Tom Breloff between September 2015 and 2017, with the stated goal of creating a plotting API for the Julia [5] language that was powerful, intuitive, concise, flexible, consistent, lightweight and smart. In particular the recipe system helped the package gain traction within the community, as the latency of loading large dependencies was generally recognized as one of the major factors limiting the uptake of Julia.

With time Tom moved on, and the development of `Plots.jl` was continued by Michael K. Borregaard, Daniel Schwabeneder and Simon Christ (cf. Figures 6, 7 and Table 1). The maintenance of the project is now a joint effort of the Julia community. The package has reached a very high uptake in the ecosystem. In the Julia Language Survey of both 2019 [35] and 2020 [34], `Plots.jl` was identified as the Julia community's favorite package across the entire ecosystem with 47 percent of all participants listing it among their favorite packages.

USAGE

`Plots.jl` is used for visualizations in scientific publications from different fields, such as numerics [32, 4, 9, 11, 15, 24], mathematics [14], biology [3, 6], ecology [13] and geology [10, 23] as well as for teaching purposes [8, 22].

Many packages in the Julia ecosystem as well as non-packaged code (e.g. for scientific projects and publications) contain `Plots.jl` recipes. According to recent download statistics [27] `Plots.jl` has between 500 and 2000 downloads per day, and over 300 published packages in the general package registry of Julia currently have recipes for `Plots.jl` defined.

COMPARISON

`Plots.jl` achieves its functionality by leveraging the multiple dispatch paradigm of julia, which allows the user to define multiple methods for the same function, with the compiler selecting the appropriate method based on the types of the input arguments. Because of the close connection to Julia's multiple dispatch paradigm its approach to plotting is fairly unique.

In python, the library unified-plotting [39] shares the aim of providing a unified API for multiple packages, in this case matplotlib [21], pyplot and javascript libraries including d3.js [7]. However, unified-plotting is still in the beta phase and not widely used.

The authors are not aware of other package ecosystems that have a recipe system akin to that of `Plots.jl`, though a recipe system inspired by that of `Plots.jl` is presently being implemented for the Julia library `Makie.jl` [12].

IMPLEMENTATION AND ARCHITECTURE

One-function API

A central design goal of `Plots.jl` is that the user should rarely have to consult the documentation while plotting. This is achieved by having a tightly unified syntax that has few function names to remember, and by having a great deal of flexibility in the inputs of those functions through aliases and redundancy.

`Plots.jl`'s main interface is simply the `plot` function,[1] which creates a new plot object. Additionally, there is a `plot!` function that modifies an existing plot object, e.g. by changing axes limits or adding new elements. Any type of predefined plot (e.g. a histogram, a bar plot, a scatter plot, a heatmap, an image, a geographical map, etc.) may be created by a call to `plot`. The exact type is defined by the keyword argument `seriestype` and the input arguments (type and number). New seriestypes can be created with recipes (see below).

For convenience, `Plots.jl` also exports *shorthand* functions named after the seriestypes (see examples in Listing 1).

All aspects of the plot are controlled by a set of plot *attributes*, that are controlled by keyword arguments [26]. `Plots.jl` distinguishes four hierarchical levels of attributes: *plot attributes*, *subplot attributes*, *axis attributes* and *series attributes* (cf. Figure 1). There is one additional special type of attributes called *magic attributes*. They allow to set multiple attributes that belong to a common element in a single keyword. E.g. `plot(1:2; line = (5, :dash))` is equivalent to `plot(1:2; linewidth = 5, linestyle = :dash)`.

A *series* in the context of `Plots.jl` syntax is an individual plot element, such as a continuous line or a set of scatter points. For example, the left plot in Figure 1 has three series, distinguished by different types of markers. A plot may contain multiple series, e.g. when adding a trend line to a scatter plot. Multiple series may be added in the same `plot` call by concatenating the data as columns in a row matrix (see below), or added sequentially with the `plot!` function.

```
boxplot(args...; kwargs...) = plot(args...; seriestype = :boxplot, kwargs...)
scatter(args...; kwargs...) = plot(args...; seriestype = :scatter, kwargs...)
```

Listing 1 Examples of shorthands. Full list available at https://docs.juliaplots/stable/api/#Plot-specification.

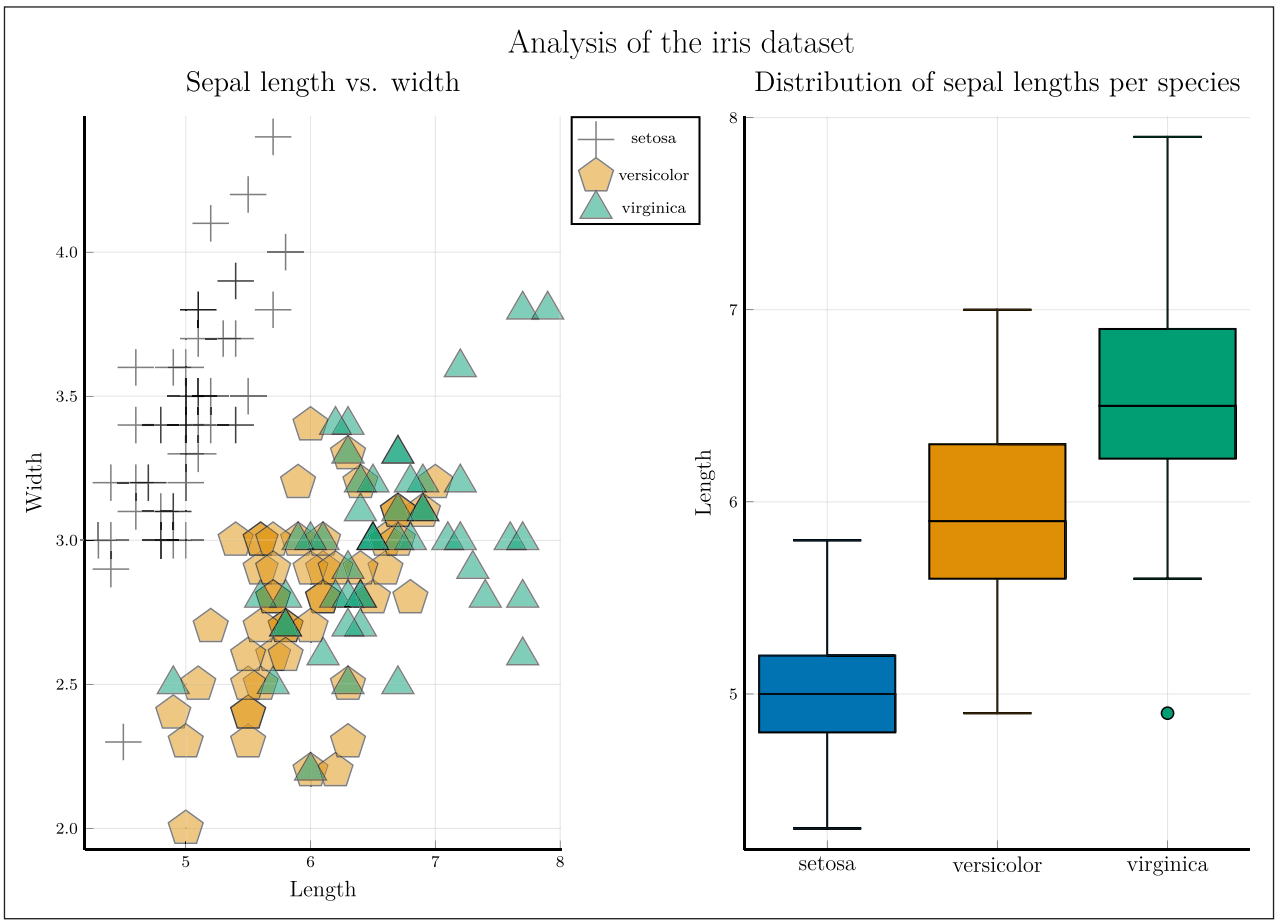

Figure 1 Example plot of the iris dataset [37] to illustrate the use of different attribute types (cf. Listing 2).

Input arguments can have many different forms, such as:

```
plot()                                  # empty Plot with axes
plot(4)                                 # initialize a Plot with 4 empty series
plot(rand(10))                          # 1 series... x = 1:10
plot(rand(10,5), rand(10))              # 5 series... y is the same for all
plot(sin, rand(10))                     # y = sin.(x)
plot([sin,cos], 0, pi)                  # sin and cos lines on the range [0, pi]
                                        # using an automatic adaptive grid
plot(1:10, Any[rand(10), sin])          # 2 series, y is rand(10) and sin.(x)
plot( plot(rand(10)), plot(rand(10)) )  # a layout with two equally sized
                                        # subplots
@df dataset("Ecdat", "Airline") plot(:Cost)  # the :Cost column from a DataFrame
                                        # @df is currently in StatsPlots.jl
```

```
using StatsPlots
import RDatasets
pgfplotsx()                                 # switching to the PGFPlotsX backend

iris = RDatasets.dataset("datasets", "iris") # loading iris data as a DataFrame
default(palette=:seaborn_colorblind)        # change the default color palette

plot(                                       # plot two Plot objects for
                                            # a 2x1 layout
    scatter(                        # plot SepalWidth against
        iris.SepalLength,                   # SepalLength in a scatter plot
        iris.SepalWidth,
        group=iris.Species,                 # series attribute
        title="Sepal length vs. width",     # subplot attribute
        xlabel="Length",                    # axis attribute
        ylabel="Width",                     # axis attribute
        marker=(0.5, [:cross :pentagon :utriangle], 12),
        # marker is a magic attribute to set many marker properties at once
    ),
    boxplot(                        # create a boxplot of the distribution
        iris.Species,                       # of the SepalLengths per Species
        iris.SepalLength,
        group=iris.Species,
        ylabel="Length",
        title = "Distribution of sepal lengths per species",
        xtickfontsize = 11,
        legend = false,
    ),
    plot_title="Visualization of the iris dataset", # plot attribute
    right_margin=1Plots.cm,             # subplot attribute
    size=(900, 600)                     # plot attribute
)

Plots.pdf("attributes.pdf")             # save the last current Plot object
                                        # as a pdf-file
```

Listing 2 Code corresponding to Figure 1.

Calling the `plot` function returns a `Plot` object. The `Plot` object is essentially a big nested dictionary holding the plot attributes for the layout, subplots, series, segments, etc. and their associated data values. The `Plot` object is automatically rendered in the surrounding context[2] when returned to an interactive session, or can be displayed explicitly by calling the `display` function on the object. This delayed rendering means that `plot` calls can be combined without unnecessary intermediate rendering.

Pipeline

The plotting pipeline has two main stages (cf. Figure 2): construction of the plot using `plot/plot!` calls and creation of the output via `savefig/display/gui` calls. These calls are often called implicitly in environments like the Julia REPL, notebooks or IDEs.

The very first step upon construction is to convert all inputs to form the list of plot attributes that constitute the plot specification. As shown in Listing 3 `Plots.jl` is

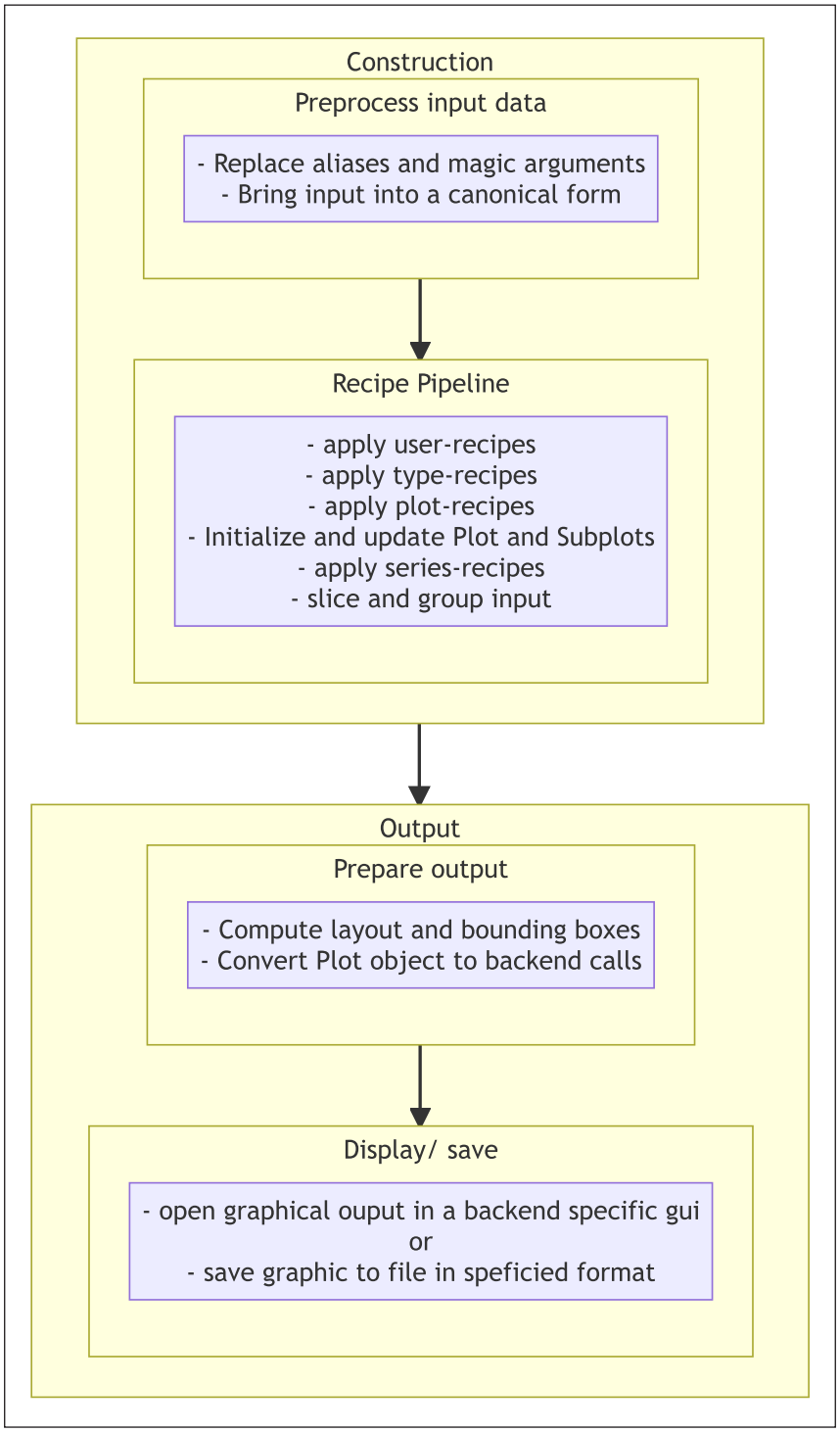

Figure 2 Plotting pipeline in `Plots.jl`. The separation of construction and output production enables the flexible use of different backends in the same session and helps to avoid unnecessary intermediate calculation. Created using `mermaid` [1].

```
plot(2:4, c = :steelblue)                # c is the shortest alias for seriescolor
plot([2,3,4], color = 1)                 # :steelblue is the first color
                                         # of the default palette
plot(1:3, [2,3,4], colour = :auto)       # the recipe for a single input
                                         # will use 1:3 as x-values
plot(1:3, [2,3,4], seriescolors = 1)     # you can use singular
                                         # or plural version of attributes
plot([1,2,3], [2,3,4], seriescolor = RGBA{Float64}(0.275,0.51,0.706,1.0))
# this is the fully expanded call
```

Listing 3 Examples of input preprocessing steps in `Plots.jl`. All these calls are equivalent.

very flexible about possible input values. The conversion step involves defining values for every attribute based on the provided keyword arguments. This includes replacing *aliases* of attributes (which are multiple alternatively spelled keywords, such as 'c' or 'color', encoding the same attribute), handling of missing and nothing values in the input data and attribute values, and determining the final values based on the set of defaults. The default values are organized in a hierarchical framework, based on the values of other attributes; e.g. `linecolor`, `fillcolor` and `markercolor` will default to `seriescolor` under most seriestypes. But, for instance, under the `bar` seriestype, `linecolor` will default to `:black`, giving bars a black border by default. This allows the construction of appropriate plots with a minimum of user specification, an input paradigm that contrasts with that of e.g. matplotlib, where every aspect of the plot is usually defined manually by the user. This paradigm allows for quick and simple visualisation as part of e.g. data analysis.

After input and attribute conversion, recipes are applied recursively and the `Plot` and `Subplot` objects are initialized. Recipes will be explained in detail in the next section.

When an output is to be produced the layout will be computed and the backend-specific code will be executed to produce the result.

Recipes

As mentioned in the introduction, recipes are the key mechanism in the `Plots.jl` pipeline to allow composable definition of visualizations across Julia packages. A recipe works as a template that allows the users and package developers to define custom plotting routines while only depending on the lightweight package `RecipesBase.jl` (instead of on `Plots.jl`). Recipes are applied recursively, which makes it easy to compose or combine multiple recipes. This means that any recipe may call other recipes until the plot consists of a *seriestype* defined internally in `Plots.jl`. This composability, is a major improvement to ecosystem support, as it gives a combinatory reduction in the amount of code required for downstream libraries to add native plotting support for their types.

`Plots.jl` distinguishes four types of recipes: *user recipes*, *type recipes*, *plot recipes* and *series recipes* [20] (cf. Listing 4). By far the most commonly used types are *User recipes*, which define how to plot objects of a certain type, and *series recipes*, which define a new seriestype. All four types can be constructed with the `@recipe` macro which acts on a function definition and creates a new method for the `RecipesBase.apply_recipe` function. The recipe type is determined by the signature of the function definition, utilizing the multiple dispatch capabilities of the Julia programming language.

It is sufficient to depend on the `RecipesBase.jl` package, a small and lightweight dependency to define a recipe. The aim of `RecipesBase.jl` is to make specialized syntax available for the code author to define visualizations; it has no effect until the package end user loads `Plots.jl` directly.

How is this an improvement over other approaches to defining new visualizations? In most plotting libraries such as `matplotlib` [21], a downstream ODE solver library can add a new function `plotsolution` that will plot an ODE solution. However, the primary technological advance of the `Plots.jl` recipe system is that the application of recipes is recursive and extendable via multiple dispatch. This solves a combinatory problem for downstream support: it is possible to combine and chain recipes to support plotting on new combinations of input types without ever defining a recipe for that specific combination.

To illustrate this, consider the example of combining recipes defined by the Julia packages `DifferentialEquations.jl` [33] and `Measurements.jl` [16] (cf. Figure 3 and Listing 9). In this example, a user solves a differential equation with uncertain initial conditions specified by `Measurements.Measurement` objects. The uncertainty encoded in the `Measurement` objects are automatically propagated through the ODE solver, as multiple methods for this type have been defined for arithmetic functions. The resulting ODE solution `Sol` will then also be specified in terms of `Measurements.Measurement`s. When calling `plot(sol)`, the recipe for ODE solvers will transform the `ODESolution` object into an array of arrays, each representing a time series to plot, using techniques like dense output to produce a continuous looking solution. This array of arrays contains number types matching the state of the solution, in this case `Measurements.Measurement`s. Successive applications of the user recipe defined in `Measurements.jl` then take each state value and assign the `uncertainty` part of the state to the `yerror` attribute

```
using RecipesBase
struct CustomStruct end
@recipe function f(arg::CustomStruct; custom_kw = 1)            # user recipe
end
@recipe function f(::Type{CustomStruct}, val::CustomStruct)     # type recipe
end
@recipe function f(::Type{Val{:recipename}}, plt::AbstractPlot) # plot recipe
end
@recipe function f(::Type{Val{:recipename}}, x, y, z)           # series recipe
end
```

Listing 4 Recipe signatures.

and pass the `value` part of the state to the next recipe. When used with the initial seriestype `:scatter` this results in a scatter plot with proper error bars as seen in Figure 3.

Notably, the two packages have not been developed to work together and are not aware of each other. Yet, multiple dispatch allows to efficiently combine functionality from both packages, and the `Plots.jl` recipe system allows the combined visualization to work automatically.

The recipe of `Measurements.jl` is an example of a particularly short recipe (cf. Listing 5). A `Measurements.Measurement` is represented as a type with two fields: `value` and `uncertainty`. It can be conveniently constructed with the Unicode infix operator ±. Thus, the object $a \pm b$ has a as the `value` and b as the `uncertainty`. An array of measurement values can be converted into an array of floating point values to plot, along with having the uncertainties as error bars, via the recipe defined in Listing 5:

Structure and interfaces

The code for `Plots.jl` is not located in one repository, but split into a few packages, to enhance reuse of more general parts of the code by other packages (cf. Figure 4). In the following the different packages and their use cases will be described.

Plots.jl: The main user facing package; Defines all default values and holds the code for layouting, conversion of input arguments, output generation, all backend code and the default recipes. This is the repository with the highest rate of change.

StatsPlots.jl: A drop-in replacement for `Plots.jl`, meaning it loads and reexports all of `Plots.jl` and adds recipes that are specially targeted at visualisation of statistical data; It aims to be integrated with Julia's statistical package ecosystem under the JuliaStats organisation. Therefore it has more dependencies than `Plots.jl`, which increases the loading time. Since not all `Plots.jl` users need this functionality it is separated in its own repository.

PlotUtils.jl: Provides general utility routines, such as handling colors, optimizing ticks or function sampling; This package is also used by e.g. the newer plotting package `Makie.jl`.

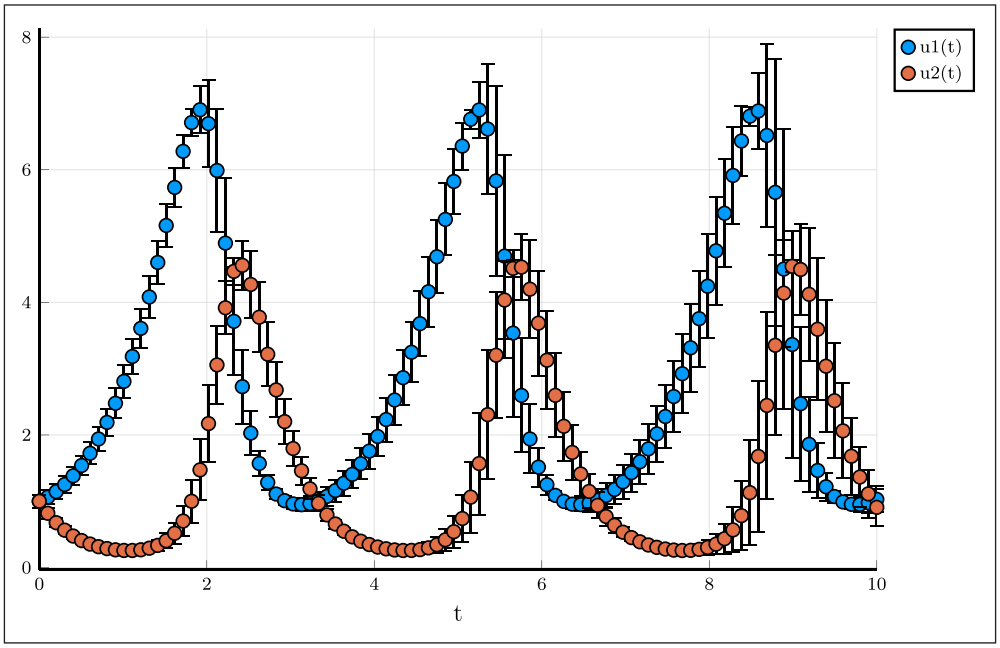

Figure 3 Showcase of composing recipes. Plotting a `ODESolution` object from `DifferentialEquations.jl` containing `Measurements` from `Measurements.jl` will apply the recipe of `DifferentialEquations.jl` which will return vectors of `Measurements`, which will apply the recipe from `Measurements.jl`; yielding the solutions of the Lotka-Volterra system [2] with correct error bounds without the user having to change the callsite. Neither of these packages has code in their recipes for handling types of the other package. Full code available in Listing 9.

```
@recipe function f(x::AbstractArray, y::AbstractArray{<:Measurement})
    yerror := uncertainty.(y)       # := is special syntax in the @recipe block which
                                    # sets an attribute overriding
                                    # any present value. The alternative syntax
                                    # is --> to give call-site values precedence.
    x, value.(y)
end
```

Listing 5 `Measurements.jl` recipe.

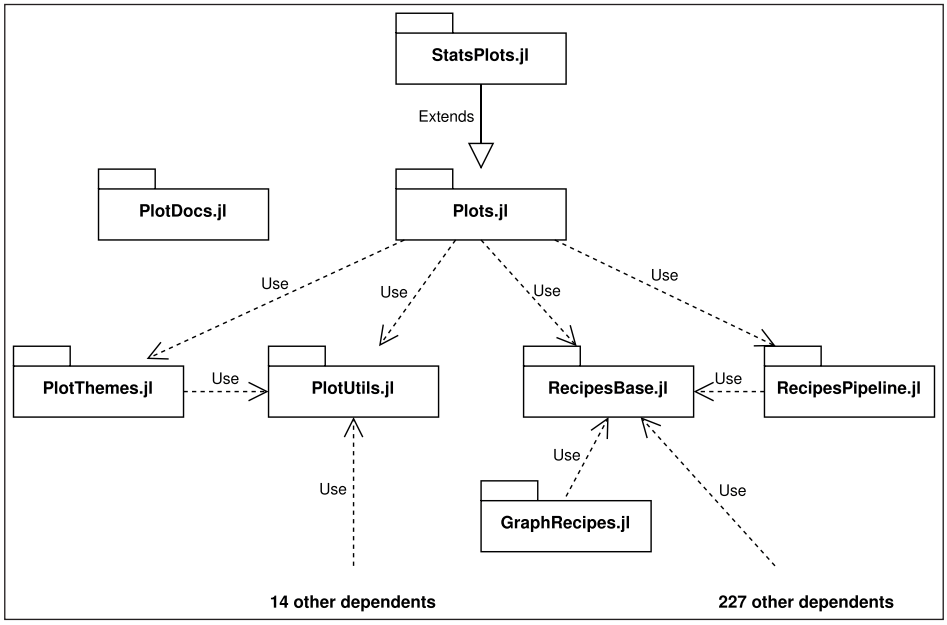

Figure 4 Overview of the `Plots.jl` ecosystem and its interfaces with other Julia packages. The numbers of dependents are taken from juliahub [30].

RecipesBase.jl: A package with zero 3rd-party dependencies, that can be used by other packages to define recipes for their own types without needing to depend on `Plots.jl`.

RecipesPipeline.jl: Another lightweight package that defines an API such that other plotting packages can consume recipes from `RecipesBase.jl` without needing to become a backend of `Plots.jl`.

GraphRecipes.jl: A package that provides recipes for visualisation of graphs in the sense of graph theory; These are also split out because they have some heavy dependencies.

PlotThemes.jl: Provides different themes for `Plots.jl`.

PlotDocs.jl: Hosts the documentation of `Plots.jl`.

Backends

`Plots.jl` currently supports seven plotting frameworks as backends. These backends are the libraries that do the actual rendering, either on the screen or to a file. The code in `Plots.jl` translates the user code to backend code (cf. Listings 6 to 8 and Figure 5).

The backend packages are independently developed by different people and organizations. Typically these plotting frameworks themselves have different graphic libraries as

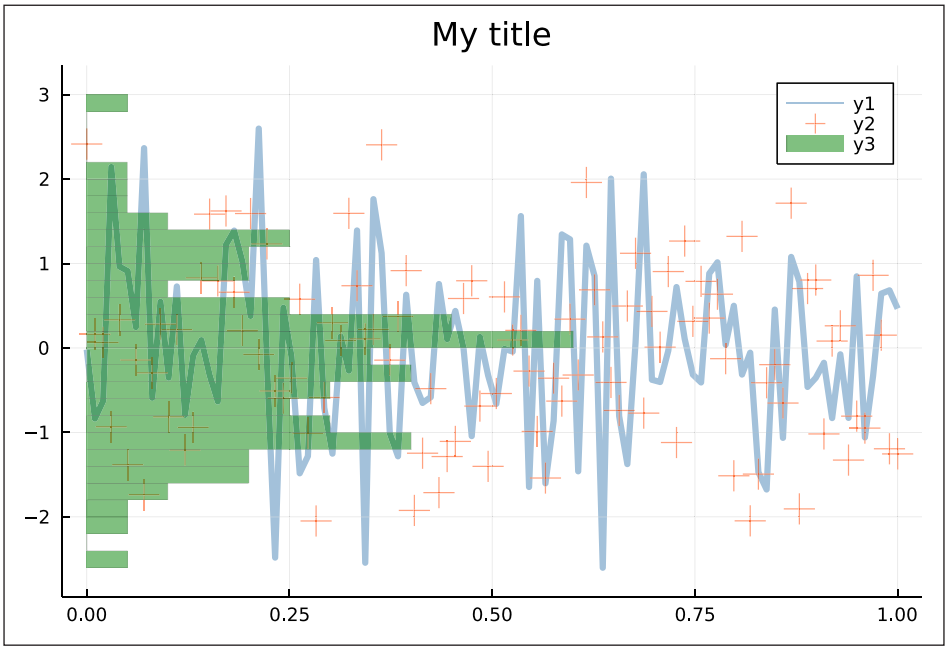

Figure 5 An example figure generated from the code shown in Listing 6. Listings 7 and 8 illustrate how the generated backend code could look like for different backends. The actual backend code is more verbose and likely uses more low-level functions. These listings also show how `Plots.jl` provides a unified API for its backend packages, since the translation between Listings 7 and 8 is not straightforward.

backends to support different output types. They differ in their area of expertise and have different trade-offs.

GR: The default backend; It uses the GR framework [18] and is among the fastest backends with a good coverage of functionality.

Plotly/PlotlyJS: The backend with the most interactivity and best web support using the `plotly` javascript library [29]; One use case is to create interactive plots in documentation [31] or notebooks. The `Plotly` backend is a version with minimal dependencies, which doesn't require the user to load any other Julia package and displays its graphics in the browser, while `PlotlyJS` requires the user to load `PlotlyJS.jl`, but offers display of plots in a standalone window.

PyPlot: `PyPlot.jl` is the Julia wrapper of `matplotlib` [21] and covers a lot of functionality at moderate speed.

PGFPlotsX: Uses the `pgfplots` LaTeXpackage [28]; Thus, it is the slowest of the backends, but integrates very good with LaTeXdocuments.

InspectDR: Fast backend with GUI and some interactivity; It does good for 2D and handles large datasets and high refresh rates [25].

```
n = 100
x, y = range(0, 1, length=n), randn(n, 3)
##
using Plots
plot(
    x, y,
    line=(0.5, [4 1 0], [:path :scatter :histogram]),
    normalize=true,
    bins=30,
    marker=(10, 0.5, [:none :+ :none]),
    color=[:steelblue :orangered :green],
    fill=0.5,
    orientation=[:v :v :h],
    title="My title",
)
```

Listing 6 `Plots.jl` code corresponding to Figure 5.

```
n = 100
x, y = range(0, 1, length=n), randn(n, 3)
##
import PyPlot
fig = PyPlot.gcf()
fig.set_size_inches(4, 3, forward=true)
fig.set_dpi(100)
PyPlot.clf()

PyPlot.plot(x, y[:, 1], alpha=0.5, "steelblue", linewidth=4)
PyPlot.scatter(x, y[:, 2], alpha=0.5, marker="+", s=100, c="orangered")
PyPlot.plt.hist(
    y[:, 3],
    orientation="horizontal",
    alpha=0.5,
    density=true,
    bins=30,
    color="green",
    linewidth=0
)

ax = PyPlot.gca()
ax.xaxis.grid(true)
ax.yaxis.grid(true)
PyPlot.title("My title")
PyPlot.legend(["y1", "y2", "y3"])
```

Listing 7 `PyPlot.jl` code roughly corresponding to `using Plots; pyplot()` in line 4 of Listing 6.

```
n = 100
x, y = range(0, 1, length=n), randn(n, 3)
##
using PGFPlotsX
using Colors
using StatsBase: Histogram, fit, normalize

@pgf Axis(
    {
        axis_x_line = "bottom",
        axis_y_line = "left",
        title = "My title",
        width = "400px",
        height = "300px",
        ymajorgrids = true,
        xmajorgrids = true
    },
    Plot(
        {solid, color = colorant"steelblue", line_width = 4, draw_opacity = 0.5},
        Coordinates(x, y[:, 1])
    ),
    LegendEntry("y1"),
    Plot(
        {only_marks, mark = "+", color = colorant"orangered", mark_size = "10px",
            draw_opacity = 0.5, line_width = 1},
        Coordinates(x, y[:, 2])
    ),
    LegendEntry("y2"),
    Plot(
        {xbar, fill = colorant"green", fill_opacity = 0.5, line_width = 0},
        # pgfplots only has a dedicated method for vertical histograms
        # so we have to transform the data on the julia side
        Table({y_index = 0, x_index = 1},
            normalize(fit(Histogram, y[:, 3], nbins=30), mode=:pdf)
        )
    ),
    LegendEntry("y3"),
)
```

Listing 8 `PGFPlotsX.jl` code roughly corresponding to `using Plots; pgfplotsx()` in line 4 of Listing 6.

UnicodePlots: A backend that allows plotting in the terminal with unicode characters; It can be used in a terminal also on headless machines [38]. Therefore it lacks a lot of functionality compared to the other backends.

HDF5: A backend that can be used to save the `Plot` object along with the data in a hdf5-file using `HDF5.jl` [19], such that it can be recovered with any backend; It potentially allows interfacing with `Plots.jl` from other programming languages.

Furthermore, there are six deprecated backends that were used in the earlier stages of `Plots.jl`, but which are no longer maintained as well as the `Gaston.jl` backend which is in an early experimental stage. `Gaston.jl` is a Julia interface for `gnuplot` [17]. This shows that `Plots.jl` can be sustained even if a maintainer of backend code leaves. Either the backend will be maintained by the community or it will be replaced by another backend.

The backend code of these backends, that is the glue code that translates `Plots.jl` objects and attributes into calls and objects of the backend library, is located in the `src/backends` folder and, apart from the default backend, are only loaded when the backend gets activated.

A shortcoming of the backend design is that, in terms of maintenance, this part of the codebase is the biggest challenge, since it requires knowledge of `Plots.jl` as well as of the target backend library. Maintainers of those libraries are usually working to capacity on their library, while users of `Plots.jl` are often using `Plots.jl` because they don't want to learn the usage of one or even several different backends. That is why feature coverage between backends typically varies, though a

good amount of these holes can be covered by recipes. Sometimes these even provide features that the backend library is missing or gives at least easier access in terms of syntax. On the other hand there are some features that are present in one backend library, but not in others. Some examples are layouts, information on hover, shared or split legends, hexagonal bins and more. In these cases one either has to find workarounds leveraging other aspects of the backend code, not support that feature at all, or only support it partially.

QUALITY CONTROL

`Plots.jl` runs unit tests of all backends, as well as visual regression tests of the default backend, against the latest version of macOS, Ubuntu and Windows, using the current stable version of Julia, the long term support version and the nightly version, on every pull request and pushes to the default branch of `Plots.jl`. Furthermore, benchmarks are run to detect performance regressions. Lastly, building the documentation creates a suite of example plots for every backend, which also sometimes highlight hard-to-detect errors.

However, the size and flexibility of this project also creates a large surface area that is hard to cover by tests in its entirety. And while it is continually worked on to increase the coverage, there is probably always something missing.

(2) AVAILABILITY

OPERATING SYSTEM

`Plots.jl` is tested on Windows, Linux and macOS.

PROGRAMMING LANGUAGE

`Plots.jl` v1.13.2 runs on Julia 1.5 and later.

ADDITIONAL SYSTEM REQUIREMENTS

Dependencies

`Plots.jl` has the following direct dependencies:

Contour.jl v0.5
FFMPEG.jl v0.2 – v0.4
FixedPointNumbers v0.6 – v0.8
GR.jl v0.46 – v0.55, v0.57
GeometryBasics.jl v0.2, v0.3.1 – v0.3
JSON.jl v0.21, v1
Latexify.jl v0.14 – v0.15
Measures.jl v0.3
NaNMath.jl v0.3
PlotThemes.jl v2
PlotUtils.jl v1
RecipesBase.jl v1
RecipesPipeline.jl v0.3
Reexport.jl v0.2, v1
Requires.jl v1
Scratch.jl v1
Showoff.jl v0.3.1 – v0.3, v1
StatsBase.jl v0.32 – v0.33

In addition, `Plots.jl` has 125 indirect dependencies all of which can be seen at JuliaHub [30]. Thanks to Julia's excellent package manager `Pkg.jl`, `BinaryBuilder.jl`, semantic versioning and required upper bounds for the general registry, handling of dependencies is relatively painless for users.

LIST OF CONTRIBUTORS

The `Plots.jl` project lives from the many contsibutions of its community. Table 1 lists all contributors of `Plots.jl` and Figures 6 and 7 illustrate the distribution of code over the different contributors. The code for creating Table 1 and figures is publicly available at https://gitlab.uni-hannover.de/comp-bio/manuscripts/plots-paper.

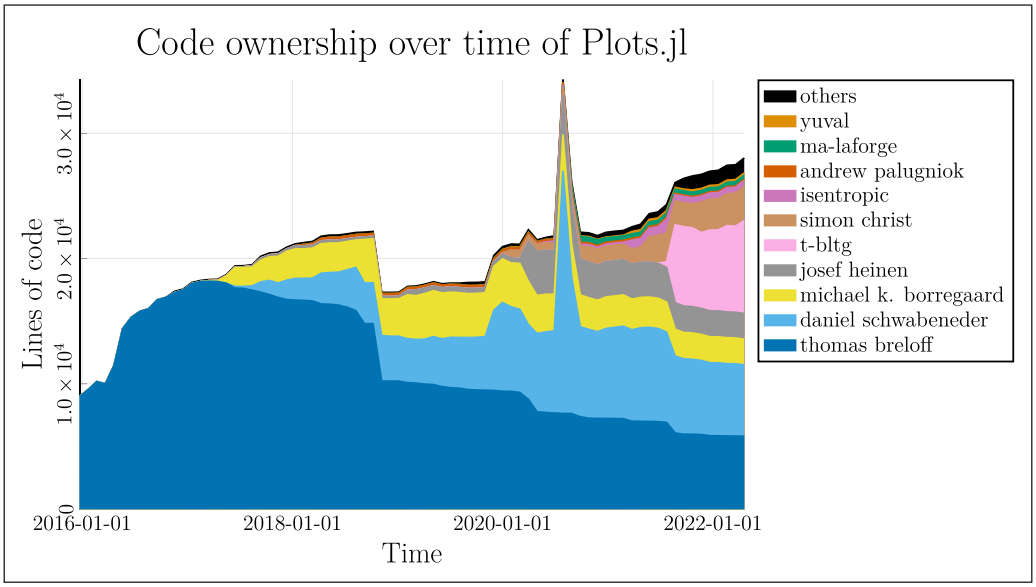

Figure 6 Lines of code alive of the top ten contributors of the `Plots.jl` repository over time. Data created with `hercules` [36].

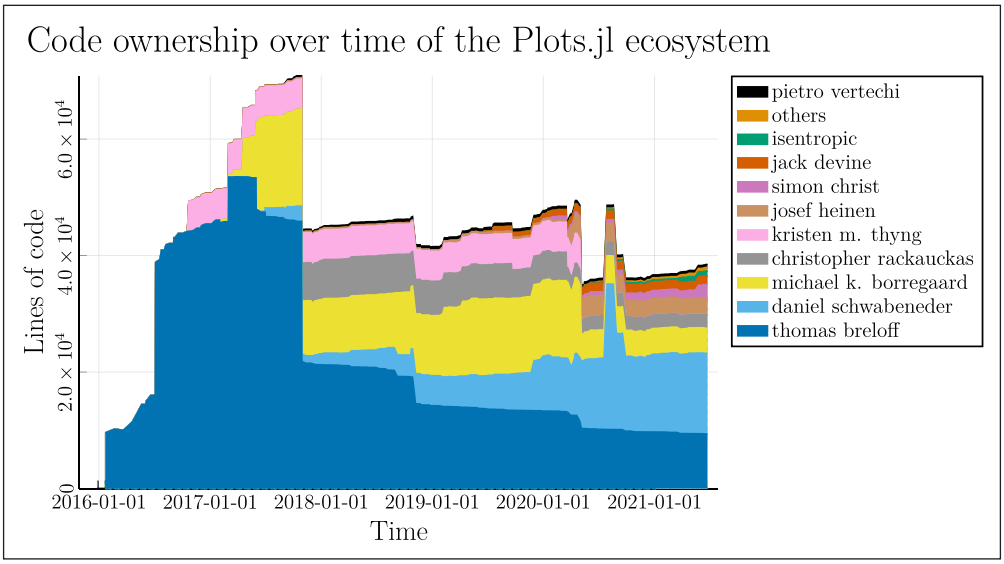

Figure 7 Lines of code alive of the top ten contributors of the `Plots.jl` ecosystem (Figure 4) over time. Data created with `hercules` [36].

NAME	AFFILIATION	ROLE	ORCID
Tom Breloff	Headlands Technologies	Creator	missing
Daniel Schwabeneder	TU Wien	ProjectLeader	0000-0002-0412-0777
Michael Krabbe Borregaard	GLOBE Institute, University of Copenhagen	ProjectLeader	0000-0002-8146-8435
Simon Christ	Leibniz Universität Hannover	ProjectLeader	0000-0002-5866-1472
Josef Heinen	Forschungszentrum Jülich	ProjectMember	0000-0001-6509-1925
Yuval	missing	Other	missing
Andrew Palugniok	missing	ProjectMember	missing
Simon Danisch	@beacon-biosignals	Other	missing
Pietro Vertechi	Veos Digital (https://veos.digital/)	ProjectMember	missing
Zhanibek Omarov	Korea Advanced Inst. of Science and Technology (KAIST)	ProjectMember	0000-0002-8783-8791
Thatcher Chamberlin	missing	Other	missing
@ma-laforge	missing	ProjectMember	missing
Christopher Rackauckas	Massachusetts Institute of Technology	Other	0000-0001-5850-0663
Oliver Schulz	Max Planck Institute for Physics	Other	missing
Sebastian Pfitzner	@JuliaComputing	Other	missing
Takafumi Arakaki	missing	Other	missing
Amin Yahyaabadi	University of Manitoba	Other	missing
Jack Devine	missing	Other	missing
Sebastian Pech	missing	Other	missing
Patrick Kofod Mogensen	@JuliaComputing	Other	0000-0002-4910-1932
Samuel S. Watson	missing	Other	missing
Naoki Saito	UC Davis	Other	0000-0001-5234-4719
Benoit Pasquier	University of Southern California (USC)	Other	0000-0002-3838-5976
Ronny Bergmann	NTNU Trondheim	Other	0000-0001-8342-7218
Andy Nowacki	University of Leeds	Other	0000-0001-7669-7383
Ian Butterworth	missing	Other	missing
David Gustavsson	Lund University	Other	0000-0002-0195-475X
Anshul Singhvi	Columbia University	Other	0000-0001-6055-1291

(Contd.)

NAME	AFFILIATION	ROLE	ORCID
david-macmahon	missing	Other	missing
Fredrik Ekre	missing	Other	missing
Maaz Bin Tahir Saeed	missing	Other	missing
Kristoffer Carlsson	missing	Other	missing
Will Kearney	missing	Other	missing
Niklas Korsbo	missing	Other	missing
Miles Lucas	missing	Other	missing
@Godisemo	missing	Other	missing
Florian Oswald	missing	Other	missing
Diego Javier Zea	missing	Other	missing
@WillRam	missing	Other	missing
Fedor Bezrukov	missing	Other	missing
Spencer Lyon	missing	Other	missing
Darwin Darakananda	missing	Other	missing
Lukas Hauertmann	missing	Other	missing
Huckleberry Febbo	missing	Other	missing
@H-M-H	missing	Other	missing
Josh Day	missing	Other	missing
@wfgra	missing	Other	missing
Sheehan Olver	missing	Other	missing
Jerry Ling	missing	Other	missing
Jks Liu	missing	Other	missing
Seth Axen	missing	Other	missing
@o01eg	missing	Other	missing
Sebastian Micluța-Câmpeanu	missing	Other	missing
Tim Holy	missing	Other	missing
Tony Kelman	missing	Other	missing
Antoine Levitt	missing	Other	missing
Iblis Lin	missing	Other	missing
Harry Scholes	missing	Other	missing
@djsegal	missing	Other	missing
Goran Nakerst	missing	Other	missing
Felix Hagemann	missing	Other	missing
Matthieu Gomez	missing	Other	missing
@biggsbiggsby	missing	Other	missing
Jonathan Anderson	missing	Other	missing
Michael Kraus	missing	Other	missing
Carlo Lucibello	missing	Other	missing
Robin Deits	missing	Other	missing
Misha Mkhasenko	missing	Other	missing
Benoît Legat	missing	Other	missing
Steven G. Johnson	missing	Other	missing
John Verzani	missing	Other	missing

(Contd.)

NAME	AFFILIATION	ROLE	ORCID
Mattias Fält	missing	Other	missing
Rashika Karki	missing	Other	missing
Morten Piibeleht	missing	Other	missing
Filippo Vicentini	missing	Other	missing
David Anthoff	missing	Other	missing
Leon Wabeke	missing	Other	missing
Yusuke Kominami	missing	Other	missing
Oscar Dowson	missing	Other	missing
Max G	missing	Other	missing
Fabian Greimel	missing	Other	missing
Jérémy	missing	Other	missing
Pearl Li	missing	Other	missing
David P. Sanders	missing	Other	missing
Asbjørn Nilsen Riseth	missing	Other	missing
Jan Weidner	missing	Other	missing
@jakkor2	missing	Other	missing
Pablo Zubieta	missing	Other	missing
Hamza Yusuf Çakır	missing	Other	missing
John Rinehart	missing	Other	missing
Martin Biel	missing	Other	missing
Moritz Schauer	missing	Other	missing
Mosè Giodano	missing	Other	missing
@olegshtch	missing	Other	missing
Leon Shen	missing	Other	missing
Jeff Fessler	missing	Other	missing
@hustf	missing	Other	missing
Asim H Dar	missing	Other	missing
@8uurg	missing	Other	missing
Abel Siqueira	missing	Other	missing
Adrian Dawid	missing	Other	missing
Alberto Lusiani	missing	Other	missing
Balázs Mezei	missing	Other	missing
Ben Ide	missing	Other	missing
Benjamin Lungwitz	missing	Other	missing
Bernd Riederer	University of Graz	Other	0000-0001-8390-0087
Christina Lee	missing	Other	missing
Christof Stocker	missing	Other	missing
Christoph Finkensiep	missing	Other	missing
@Cornelius-G	missing	Other	missing
Daniel Høegh	missing	Other	missing
Denny Biasiolli	missing	Other	missing
Dieter Castel	missing	Other	missing
Elliot Saba	missing	Other	missing

(Contd.)

NAME	AFFILIATION	ROLE	ORCID
Fengyang Wang	missing	Other	missing
Fons van der Plas	missing	Other	missing
Fredrik Bagge Carlson	missing	Other	missing
Graham Smith	missing	Other	missing
Hayato Ikoma	missing	Other	missing
Hessam Mehr	missing	Other	missing
@InfiniteChai	missing	Other	missing
Jack Dunn	missing	Other	missing
Jeff Bezanson	missing	Other	missing
Jeff Eldredge	missing	Other	missing
Jinay Jain	missing	Other	missing
Johan Blåbäck	missing	Other	missing
@jmert	missing	Other	missing
Lakshya Khatri	missing	Other	missing
Lia Siegelmann	missing	Other	missing
@marekkukan-tw	missing	Other	missing
Mauro Werder	ETH Zurich	Other	0000-0003-0137-9377
Maxim Grechkin	missing	Other	missing
Michael Cawte	missing	Other	missing
@milesfrain	missing	Other	missing
Nicholas Bauer	missing	Other	missing
Nicolau Leal Werneck	missing	Other	missing
@nilshg	missing	Other	missing
Oliver Evans	missing	Other	missing
Peter Gagarinov	missing	Other	missing
Páll Haraldsson	missing	Other	missing
Rik Huijzer	missing	Other	missing
Romain Franconville	missing	Other	missing
Ronan Pigott	missing	Other	missing
Roshan Shariff	missing	Other	missing
Scott Thomas	missing	Other	missing
Sebastian Rollén	missing	Other	missing
Seth Bromberger	missing	Other	missing
Siva Swaminathan	missing	Other	missing
Tim DuBois	missing	Other	missing
Travis DePrato	missing	Other	missing
Will Thompson	missing	Other	missing
Yakir Luc Gagnon	missing	Other	missing
Benjamin Chislett	missing	Other	missing
@hhaensel	missing	Other	missing
@improbable22	missing	Other	missing
Johannes Fleck	missing	Other	missing
Peter Czaban	missing	Other	missing

(Contd.)

NAME	AFFILIATION	ROLE	ORCID
@innerlee	missing	Other	missing
Mats Cronqvist	missing	Other	missing
Shi Pengcheng	missing	Other	missing
@wg030	missing	Other	missing
Will Tebbutt	University of Cambridge	Other	missing
@t-bltg	missing	Other	missing
Fred Callaway	missing	Other	missing
Jan Thorben Schneider	missing	Other	missing
Lee Phillips	Alogus Research Corporation	Other	0000-0003-4102-2460
Tom Gillam	missing	Other	missing

Table 1 Contributors sorted by number of commits.

SOFTWARE LOCATION

Code repository Github

Name: JuliaPlots/Plots.jl

Persistent identifier: https://doi.org/10.5281/zenodo.4725318

Licence: MIT

Version published: 1.13.2

Date published: 28/04/2021

The first version of `Plots.jl` was published on github at 11/09/2015.

LANGUAGE

julia

(3) REUSE POTENTIAL

`Plots.jl` can be used by people working in all fields for data visualization. In particular, it is possible to define backend agnostic recipes for their domain specific data structures with minimal dependencies. These can be shared, reused and extended by peers with ease by including these recipes in their packages or published scripts. Moreover, it is possible for other plotting software with Julia bindings to take advantage of the recipe system either by contributing backend code to `Plots.jl` or by using `RecipesPipeline.jl` to become an independent consumer of `RecipesBase.jl`'s recipes. Plotting software without Julia bindings could potentially use the HDF5 backend to consume fully processed and serialized recipe data.

People interested in modifying, extending or maintaining `Plots.jl` can get in contact either via the github issue tracker, the Julia discourse forum or the Julia slack and zulip spaces. There are quarterly maintenance calls that can be joined on request.

CODE EXAMPLES

```
using Plots, Measurements, OrdinaryDiffEq
pgfplotsx() # change to pgfplotsx backend

# define Lotka-Volterra equations
function f(du,u,p,t)
  du[1] = p[1]*u[1] - p[2]*u[1]*u[2]  #prey
  du[2] = -p[3]*u[2] + p[4]*u[1]*u[2] #predator
end
u0 = [1.0 ± 0.1 ; 1.0 ± 0.1] # define initial conditions with uncertainty
tspan = (0.0,10.0)              # define start and end time
p = [1.5,1.0,3.0,1.0]           # define vector of parameters
prob = ODEProblem(f,u0,tspan,p)     # create a ODEProblem object

sol = solve(prob, Tsit5())          # solve the problem using the Tsit5
                                    # integrator. Returs a ODESolution
pl = scatter(sol, plotdensity = 75) # plotdensity is a keyword of the recipe
                                    # defined in OrdinaryDiffEq
savefig(pl, "DiffEq<3Measurements.pdf") # save plot as pdf-file
pl                                  # return plot to display
```

Listing 9 Recipes showcase.

NOTES

1 Technically the API consists of more than one function, but the vast majority is `plot/plot!` and aliases thereof.

2 That is an external window when using a plain terminal, a plot pane in an IDE or the output area in a notebook environment.

ACKNOWLEDGEMENTS

We like to acknowledge the support of the Julia community and the numerous contributors that keep this project alive.

FUNDING INFORMATION

Michael K. Borregaard was supported by grant number CF19-0695 from the Carlsberg Foundation.

COMPETING INTERESTS

The authors have no competing interests to declare.

AUTHOR AFFILIATIONS

Simon Christ orcid.org/0000-0002-5866-1472
Leibniz Universität Hannover, DE

Daniel Schwabeneder orcid.org/0000-0002-0412-0777
TU Wien, AT

Christopher Rackauckas orcid.org/0000-0001-5850-0663
Massachusetts Institute of Technology, US

Michael Krabbe Borregaard orcid.org/0000-0002-8146-8435
Center for Macroecology, Evolution and Climate, Globe Institute, University of Copenhagen, DK

Thomas Breloff
Headlands Technologies, US

REFERENCES

1. *About Mermaid*. URL: https://mermaid-js.github.io/mermaid/#/README (visited on 03/02/2022).
2. **Lotka AJ.** *Elements of Physical Biology*. In collab. with Indian Institute Of Science, IISc Library, and Jiju. Williams and Wilkins Company. 1925; 495 pp. URL: http://archive.org/details/elementsofphysic017171mbp (visited on 05/31/2022).
3. **Angevaare A, Feng Z, Deardon R.** *Infectious Disease Transmission Network Modelling with Julia*; Feb. 13, 2020.
4. **Carlson FB.** MonteCarloMeasurements.Jl: Propagation of Distributions by Monte-Carlo Sampling: Real Number Types with Uncertainty Represented by Particle Clouds. 2019. URL: http://lup.lub.lu.se/record/8ff6a743-0ad6-4d98-bbb3-5d549c698bc1 (visited on 05/03/2021).
5. **Bezanson J,** et al. Julia: A Fresh Approach to Numerical Computing. In: *SIAM Rev*. Jan. 2017; 59(1): 65–98. ISSN: 0036-1445, 1095-7200. DOI: https://doi.org/10.1137/141000671
6. **Bonham KS,** et al. Microbiome.Jl and BiobakeryUtils.Jl – Julia Packages for Working with Microbial Community Data. In: *Journal of Open Source Software*. Nov. 17, 2021; 6(67): 3876. ISSN: 2475-9066. DOI: https://doi.org/10.21105/joss.03876
7. **Bostock M.** *D3.Js – Data-Driven Documents*. URL: https://d3js.org/ (visited on 03/02/2022).
8. **Boyd S, Vandenberghe L.** *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares*. 1st ed. Cambridge University Press; June 7, 2018. ISBN: 978-1-316-51896-0 978-1-108-58366-4. DOI: https://doi.org/10.1017/9781108583664
9. **Caldwell A,** et al. BAT.Jl Upgrading the Bayesian Analysis Toolkit. In: *EPJ Web Conf*. 2020; 245: 06001. ISSN: 2100-014X. DOI: https://doi.org/10.1051/epjconf/202024506001
10. **Constantinou N,** et al. GeophysicalFlows.Jl: Solvers for Geophysical Fluid Dynamics Problems in Periodic Domains on CPUs GPUs. In: *JOSS*. Apr. 21, 2021; 6(60): 3053. ISSN: 2475-9066. DOI: https://doi.org/10.21105/joss.03053
11. **Čufar M.** Ripserer.Jl: Flexible and Efficient Persistent Homology Computation in Julia. In: *JOSS*. Oct. 19, 2020; 5(54): 2614. ISSN: 2475-9066. DOI: https://doi.org/10.21105/joss.02614
12. **Danisch S, Krumbiegel J.** Makie.Jl: Flexible High-Performance Data Visualization for Julia. In: *Journal of Open Source Software*. Sept. 1, 2021; 6(65): 3349. ISSN: 2475-9066. DOI: https://doi.org/10.21105/joss.03349
13. **Dansereau G, Poisot T.** SimpleSDMLayers.Jl and GBIF.Jl: A Framework for Species Distribution Modeling in Julia. In: *JOSS*. Jan. 27, 2021; 6(57): 2872. ISSN: 2475-9066. DOI: https://doi.org/10.21105/joss.02872
14. **Driscoll T.** ComplexRegions.Jl: A Julia Package for Regions in the Complex Plane. In: *JOSS*. Dec. 2, 2019; 4(44): 1811. ISSN: 2475-9066. DOI: https://doi.org/10.21105/joss.01811
15. **Fairbrother J,** et al. GaussianProcesses.Jl: A Nonparametric Bayes Package for the Julia Language; June 30, 2019. URL: http://arxiv.org/abs/1812.09064 (visited on 05/03/2021).
16. **Giordano M.** Uncertainty Propagation with Functionally Correlated Quantities. In: *ArXiv e-prints*; Oct. 2016. arXiv: 1610.08716 [physics.data-an].
17. *Gnuplot Homepage*. URL: http://www.gnuplot.info/ (visited on 05/31/2022).
18. **Heinen J.** *GR Framework — GR Framework 0.64.0 Documentation*. URL: https://gr-framework.org/index.html (visited on 03/03/2022).
19. *Home · HDF5.Jl*. URL: https://juliaio.github.io/HDF5.jl/stable/ (visited on 03/03/2022).
20. *How Do Recipes Actually Work?* URL: https://daschw.github.io/recipes/#what_are_recipes (visited on 03/02/2022).
21. **Hunter JD.** Matplotlib: A 2D Graphics Environment. In: *Computing in Science & Engineering*. 2007; 9(3): 90–95. DOI: https://doi.org/10.1109/MCSE.2007.55

22. *Introduction to Computational Thinking*. URL: https://computationalthinking.mit.edu/Spring21/newton_method/ (visited on 05/27/2021).
23. **Keller CB, Harrison TM.** Constraining Crustal Silica on Ancient Earth. In: *Proceedings of the National Academy of Sciences*. Sept. 2020; 117(35): 21101–21107. DOI: https://doi.org/10.1073/pnas.2009431117
24. **Lindner M,** et al. NetworkDynamics.Jl – Composing and Simulating Complex Networks in Julia; Mar. 26, 2021. URL: http://arxiv.org/abs/2012.12696 (visited on 05/03/2021).
25. **ma-laforge.** *InspectDR.Jl: Fast, Interactive Plots*; Mar. 3, 2022. URL: https://github.com/ma-laforge/InspectDR.jl (visited on 03/03/2022).
26. *Overview · Plots*. URL: https://docs.juliaplots.org/latest/attributes/ (visited on 05/11/2021).
27. *Package Download Stats for Julia*. URL: https://pkgs.genieframework.com/ (visited on 03/02/2022).
28. *PGFPlots – A LaTeX Package to Create Plots*. URL: http://pgfplots.sourceforge.net/ (visited on 03/03/2022).
29. *Plotly JavaScript Graphing Library*. URL: https://plotly.com/javascript/ (visited on 03/03/2022).
30. *Plots · JuliaHub*. URL: https://juliahub.com/ui/Packages/Plots/ld3vC/1.13.2?t=1 (visited on 05/11/2021).
31. *Plotting · SpectralDistances*. URL: https://baggepinnen.github.io/SpectralDistances.jl/latest/plotting/ (visited on 05/27/2021).
32. **Rackauckas C, Nie Q.** DifferentialEquations.Jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. In: *Journal of Open Research Software*. 1 May 25, 2017; 5(1): 15. ISSN: 2049-9647. DOI: https://doi.org/10.5334/jors.151
33. **Rackauckas C,** et al. SciML/DifferentialEquations.Jl: V7.1.0. *Zenodo*; Jan. 11, 2022. DOI: https://doi.org/10.5281/zenodo.5837925
34. **Shah VB, Claster A.** *2020 Julia User and Developer Survey*. URL: https://julialang.org/blog/2020/08/2020-julia-user-anddeveloper-survey/ (visited on 05/30/2022).
35. **Shah VB, Claster A, Abhijith C.** *Julia User – Developer Survey 2019*. URL: https://julialang.org/blog/2019/08/2019-julia-survey/(visited on 05/30/2022).
36. *Src-d/Hercules*. source{d}, May 19, 2021. URL: https://github.com/srcd/hercules (visited on 05/19/2021).
37. *UCI Machine Learning Repository: Iris Data Set*. URL: https://archive.ics.uci.edu/ml/datasets/Iris/ (visited on 04/07/2022).
38. *UnicodePlots*. JuliaPlots; Mar. 3, 2022. URL: https://github.com/JuliaPlots/UnicodePlots.jl (visited on 03/03/2022).
39. *Unified Plotting — Unified-Plotting 0.5.0rc4 Documentation*. URL: https://robert-haas.github.io/unified-plotting-docs/ (visited on 08/04/2021).

TO CITE THIS ARTICLE:

Christ S, Schwabeneder D, Rackauckas C, Borregaard MK, Breloff T 2023 `Plots.jl` – A User Extendable Plotting API for the Julia Programming Language. *Journal of Open Research Software*, 11: 5. DOI: https://doi.org/10.5334/jors.431

Submitted: 01 June 2022 **Accepted:** 25 January 2023 **Published:** 14 February 2023