Modular Algorithms for Scientific Computing in Julia

Christopher Rackauckas[1]

[1]Affiliation not available

April 17, 2023

the WINNOWER

COMPUTER SCIENCES

Modular Algorithms for Scientific Computing in Julia

CHRISTOPHER RACKAUCKAS

CORRESPONDENCE:
me@chrisrackauckas.com

DATE RECEIVED:
August 18, 2018

DOI:
10.15200/winn.153459.98996

ARCHIVED:
August 18, 2018

CITATION:
Christopher Rackauckas, Modular Algorithms for Scientific Computing in Julia, *The Winnower* 5:e153459.98996 , 2018 , DOI: 10.15200/winn.153459.98996

When most people think of the Julia programming language, they usually think about its speed. Sure, Julia is fast, but what I emphasized in a previous blog post is that what makes Julia tick is not just-in-time compilation, rather it's specialization. In that post, I talked a lot about how multiple dispatch allows for Julia to automatically specialize functions on the types of the arguments. That micro-specialization is what allows Julia to compile functions which are as fast as those from C/Fortran.

However, there is a form of "macro specialization" that Julia's design philosophy and multiple dispatch allows one to build libraries for. It allows you to design an algorithm in a very generic form, essentially writing your full package with inputs saying "insert scientific computing package here", allowing users to specialize the entire overarching algorithm to the specific problem. JuliaDiffEq, JuliaML, JuliaOpt, and JuliaPlots have all be making use of this style, and it has been leading to some really nice results. What I would like to do is introduce how to program in this style, and the advantages that occur simply by using this design. I want to show that not only is this style very natural, but it solves many extendability problems that libraries in other languages did not have a good answer for. The end result is a unique Julia design which produces both more flexible and more performant code.

A CASE STUDY: PARAMETER ESTIMATION IN DIFFERENTIAL EQUATIONS

I think the easiest way to introduce what's going on and why it's so powerful is to give an example. Let's look at parameter estimation in differential equations. The setup is as follows. You have a differential equation $y'_\theta = f(t,y_\theta,\theta)$ which explains how $y_\theta(t)$ evolves over time with θ being the coefficients of the model. For example, θ could be the reproduction rates of different species, and this ODE describes how the populations of the ecosystem (rabbits, wolves, etc.) evolve over time (with $y(t)$ being a vector at each time which is indicative of the amount of each species). So if we just have rabbits and wolves, $y(1) = [1.5;.75]$ would mean there's twice as many rabbits as there are wolves at time $t = 1$, and this could be with reproduction rates $\theta = [0.25,0.5]$ births per time unit for each species.

Assume that we have some measurements $\overline{y}(t)$ which are real measurements for the number of rabbits and wolves. What we would like to do is come up with a good number for the reproduction rates θ such that our model's outputs match reality. The way that one does this is you set up a optimization problem. The easiest problem is to simply say you want to minimize the (Euclidian) difference between the model's output and the data. In mathematical notation, this is written as:

$$\text{argmin}_\theta L(\theta) = [\Vert ?]\overline{y}(t) - y_\theta(t)[\Vert ?]$$

or more simply, find the θ such that the loss function is minimized (the norm notation $[\Vert ?]x[\Vert ?]$ just means sum of the squares, so this is the sum of squared differences between the model and reality. This is the same loss measure which is used in regressions). Intuitive, right?

THE STANDARD APPROACH TO DEVELOPING AN ALGORITHMS FOR ESTIMATION

That's math. To use math, we need to pull back on the abstraction a little bit. In reality, $\overline{y}(t)$ is not a function at all times, we have discrete times where we measured data points. And also, we don't know $y(t)$! We know the differential equation, but most differential equations cannot be solved analytically. So what we must do to solve this is:

1. Numerically solve the differential equation.
2. Use the numerical solution in some optimization algorithm.

At this point, the simple mathematical problem no longer sounds so simple: we have to develop a software for solving differential equations, and software for doing optimization!

If you check out how packages normally will do this, you will see that they tend to re-invent the wheel. There are issues with this approach. For one, you probably won't develop the fastest most complex

numerical differential equation algorithms or optimization algorithms because this might not be what you do! So you'll opt for simpler implementations of these parts, and then the package won't get optimal performance, and you'll have more to do/maintain.

Another approach is to use packages. For example, you can say "I will call _____ for solving the differential equations, and _____ for solving the optimization problem". However, this has some issues. First of all, many times in scientific computing, in order to compute the solution more efficiently, the algorithms may have to be tailored to the problem. This is why there's so much research in new multigrid methods, new differential equations methods, etc! So the only way to make this truly useful is to make it expansive enough such that users can pick from many choices. This sounds like a pain.

But what if I told you there's a third way, where "just about any package" can be used? This is the modular programming approach in Julia.

MODULAR PROGRAMMING THROUGH COMMON INTERFACES

The key idea here is that multiple dispatch allows many different packages to implement the same interface, and so if you write code to that interface, you will automatically be "package-independent". Let's take a look at how DiffEqParamEstim.jl does it (note that this package is still under development so there are a few rough edges, but it still illustrates the idea beautifully).

In JuliaDiffEq, there is something called the "Common Interface". This is documented in the DifferentialEquations.jl documentation. What it looks like is this: to solve an ODE $y' = f(t,y)$ with initial condition y_0 over a time interval $tspan$, you make the problem type:

prob = ODEProblem(f,y0,tspan)
and then you call solve with some algorithm. For example, if we want to use the Tsit5 algorithm from OrdinaryDiffEq.jl, we do:

sol = solve(prob,Tsit5())
and we can pass a bunch more common arguments. More details for using this can be found in the tutorial. But the key idea is that you can call the solvers from Sundials.jl, a different package, using

sol = solve(prob,CVODE_BDF())
Therefore, this "solve(prob,alg)" is syntax which is "package-independent" (the restrictions will be explained in a later section). What we can then do is write our algorithm at a very generic level with "put ODE solver here", and the user could then use any ODE solver package which implements the common interface.

Here's what this looks like. We want to take in an ODEProblem like defined above (assume it's the problem which defines our rabbit/wolves example), the timepoints for which we have data at, the array of data values, and the common interface algorithm to solve the ODE. The resulting code is very simple:

```
function build_optim_objective(prob::AbstractODEProblem,t,data,alg;loss_func = L2DistLoss,kwargs...)
  f = prob.f
  cost_function = function (p)
    for i in eachindex(f.params)
      setfield!(f,f.params[i],p[i])
    end

    sol = solve(prob,alg;kwargs...)

    y = vecvec_to_mat(sol(t))
    norm(value(loss_func(),vec(y),vec(data)))
  end
end
```

In takes in what I said and spits out a function which:

1. Sets the model to have the parameters p
2. Solves the differential equation with the chosen algorithm
3. Computes the loss function using LossFunctions.jl

We can then use this as described in the DifferentialEquations.jl manual with any ODE solver on the common interface by plugging it into Optim.jl or whatever optimization package you want, and can change the loss function to any that's provided by LossFunctions.jl. Now any package which follows these interfaces can be directly used as a substitute without users having to re-invent this function (you can even use your own ODE solver).

(Note this algorithm still needs to be improved. For example, if an ODE solver errors, which likely happens if the parameters enter a bad range, it should still give a (high) loss value. Also, this implementation assumes the solver implements the "dense" continuous output, but this can be eased

to something else. Still, this algorithm in its current state is a nice simple example!)

HOW DOES THE COMMON INTERFACE WORK?

I believe I've convinced you by now that this common interface is kind of cool. One question you might have is, "I am not really a programmer but I have a special method for my special differential equation. My special method probably doesn't exist in a package. Can I use this for parameter estimation?". The answer is yes! Let me explain how.

All of these organizations (JuliaDiffEq, JuliaML, JuliaOpt, etc.) have some kind of "Base" library which has the functions which are extended. In JuliaDiffEq, there's DiffEqBase.jl. DiffEqBase defines the method "solve". Each other the component packages (OrdinaryDiffEq.jl, Sundials.jl, ODE.jl, ODEInterface.jl, and recently LSODA.jl) import DiffEqBase:

import DiffEqBase: solve
and add a new dispatch to "solve". For example, in Sundials.jl, itdefines (and exports) the algorithm types (for example, CVODE_BDF) all as subtypes of SundialsODEAlgorithm, like:

Abstract Types
abstract SundialsODEAlgorithm{Method,LinearSolver}
and then it defines a dispatch to solve:

```
function solve{uType,tType,isinplace,F,Method,LinearSolver}(
   prob::AbstractODEProblem{uType,tType,isinplace,F},
   alg::SundialsODEAlgorithm{Method,LinearSolver},args...;kwargs...)

  ...
```

where I left out the details on the extra arguments. What this means is that now

```
sol = solve(prob,CVODE_BDF())
```

which points to the method defined here in the Sundials.jl package. Since Julia is cool and compiles all of the specializations based off of th

So yes, if you import DiffEqBase and put this interface on your ODE solver, parameter estimation from DiffEqParamEstim.jl will "just work".

SUMMARY OF WHAT JUST HAPPENED!

This means that if domains of scientific computing in Julia conform to common interfaces, not only do users need to learn only one API, but

WHERE DO WE GO FROM HERE?

Okay, so I've probably convinced you that these common interfaces are extremely powerful. What is their current state? Well, let's look the

1. Optimization

2. Numerical Differential Equations

3. Machine Learning

4. Plotting

5. Linear Solving (Ax=b)

6. Nonlinear Solving (F(x)=0, find x)

Almost every problem in scientific computing seems to boil down to repeated application of these algorithms, so if one could write all of thes

One use case is as follows: say your research is in numerical linear algebra, and you've developed some new multigrid method with helps s

```
algs = [Rosenbrock23(linear_solver=MyLinearSolver());
```

```
    Rosenbrock23(linear_solver=qrfact());
    Rosenbrock23(linear_solver=lufact());
    ...
   ]

for alg in algs
 sol = solve(prob,alg)
 @time sol = solve(prob,alg)
end
```

and tada you're benchmarking every linear solver on the same ODE using the Rosenbrock23 method (this is what DiffEqBenchmarks.jl doe

Are we there yet? No, but we're getting there. Let's look organization by organization.

JULIAOPT

JuliaOpt provides a common interface for optimization via MathProgBase.jl. This interface is very mature. However, Optim.jl doesn't seem t

JULIADIFFEQ

JuliaDiffEq is rather new, but it has rapidly developed and now it offers a common API which has been shown in this blog post and is docum

DifferentialEquations.jl no longer directly contains any solver packages. Instead, its ODE solvers were moved to OrdinaryDiffEq.jl, its SDE (s

JULIAML

JuliaML is still under heavy development (think of it currently as a "developer preview"), but its structure is created in this same common AP

JULIAPLOTS

Plots.jl (technically not in JuliaPlots, but ignore that) implements a common interface for plotting using many different plotting packages as "l

```
plot(sol)
```

plots the solution, and using Plots.jl's backend controls, this works with a wide variety of packages like GR, PyPlot, Plotly, etc. This is really

LINEAR SOLVING

We are close to having a common API for linear solving. The reason is because Base uses the type system to dispatch on \. For example, t

```
K = lufact(A)
x = K\b
```

and to solve using QR-factorization, one instead would use

```
K = qrfact(A)
x = K\b
```

Thus what one could do is take in a function for a factorization type and use that:

```
function test_solve(linear_solve)
  K = linear_solve(A)
  x = K\b
end
```

Then the user can pass in whatever method they think is best, or implement their own linear_solve(A) which makes a type and has a dispat

Fortunately, this even works with PETSc.jl:

```
K = KSP(A, ksp_type="gmres", ksp_rtol=1e-6)
x = K\b
```

so in test_solve, one could make the user do:

```
test_solve((A)->KSP(A, ksp_type="gmres", ksp_rtol=1e-6))
```

(since linear_solve is supposed to be a function of the matrix, so use an anonymous function to make a closure have all of the right argume

NONLINEAR SOLVING

There is no common interface in Julia for nonlinear solving. You're back to the old way of doing it right now. Currently I know of 3 good pack

1. NLsolve.jl
2. Roots.jl
3. Sundials.jl (KINSOL)

NLsolve.jl and Roots.jl are native Julia methods and support things like higher precision numbers, while Sundials.jl only supports Float64 an

CONCLUSION

These common interfaces which we are seeing develop in many different places in the Julia package ecosystem are not just nice for users,

That is the power of multiple dispatch.