

A Comparison Between Differential Equation Solver Suites In MATLAB, R, Julia, Python, C, Mathematica, Maple, and Fortran

CHRISTOPHER RACKAUCKAS

READ REVIEWS

WRITE A REVIEW

CORRESPONDENCE:

me@chrisrackauckas.com

DATE RECEIVED:

August 18, 2018

DOI:

10.15200/winn.153459.98975

ARCHIVED:

August 18, 2018

CITATION:

Christopher Rackauckas, A Comparison Between Differential Equation Solver Suites In MATLAB, R, Julia, Python, C, Mathematica, Maple, and Fortran, *The Winnower* 5:e153459.98975, 2018, DOI:

[10.15200/winn.153459.98975](https://doi.org/10.15200/winn.153459.98975)

© Rackauckas This article is distributed under the terms of the [Creative Commons Attribution 4.0 International License](#), which permits unrestricted use, distribution, and redistribution in any medium, provided that the original author and source are credited.

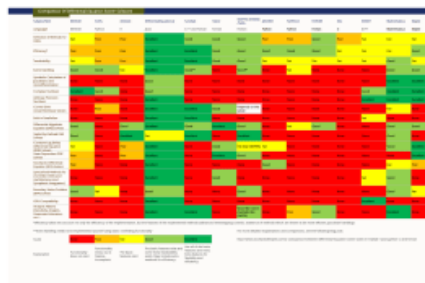


Many times a scientist is choosing a programming language or a software for a specific purpose. For the field of scientific computing, the methods for solving differential equations are one of the important areas. What I would like to do is take the time to compare and contrast between the most popular offerings. This is a good way to reflect upon what's available and find out where there is room for improvement. I hope that by giving you the details for how each suite was put together (and the "why", as gathered from software publications) you can come to your own conclusion as to which suites are right for you.

(Full disclosure, I am the lead developer of DifferentialEquations.jl. You will see at the end that DifferentialEquations.jl does offer pretty much everything from the other suite combined, but that's no accident: our software organization came last and we used these suites as a guiding hand for how to design ours.)

QUICK SUMMARY TABLE

If you just want a quick summary, I created a table which has all of this information. You can find it here ([click for PDF](#)):



MATLAB'S BUILT-IN METHODS

Due to its popularity, let's start with MATLAB's built in differential equation solvers. MATLAB's differential equation solver suite [was described in a research paper by its creator Lawerance Shampine](#), and this paper is one of the most highly cited SIAM Scientific Computing publications.

Shampine also had [a few other papers](#) at this time developing the idea of a "methods for a problem solving environment" or a PSE. The idea is pretty simple: users of a problem solving environment (the examples from his papers are MATLAB and Maple) do not have the same requirements as more general users of scientific computing. Instead of focusing on efficiency, the key for this group is to have a clear and neatly defined (universal) interface which has a lot of flexibility.

The MATLAB ODE Suite does extremely well at hitting these goals. MATLAB documents its ODE solvers very well, there's a similar interface for using each of the different methods, and it tells you in a table in which cases you should use the different methods.

But the modifications to the methods goes even further. Let's take for example the classic ode45. That method just works and creates good plots, right? Well, Shampine added a little trick to it. When you solve an equation using ode45, the Runge-Kutta method uses a "free" interpolation to fill in some extra points. So between any two steps that the solver takes, it automatically adds in 4 extra points using a 4th order interpolation. This is because high order ODE solvers are good enough at achieving "standard user error tolerances" that they actually achieve quite large timesteps, and in doing so step too infrequently to make a good plot. Shampine's scheme is a good quick fix to this problem which most people probably never knew was occurring under the hood!

There's quite a bit of flexibility here. The methods allow you to use complex numbers. You're given access to the "dense output function" (this is the function which computes the interpolations). There's a few options you can tweak. Every one of these methods is setup with event handling, and there are methods which can handle differential-algebraic equations. There are also dde23 and ddesd for delay differential equations, and in the financial toolbox there's an Euler-Maruyama method for SDEs.

While MATLAB does an excellent job at giving a large amount of easily available functionality, where it lacks is performance. There's a few reasons for this. For one thing, these modifications like adding extra points to the solution array can really increase the amount of memory consumed if the ODE system is large! This actually has an impact in other ways. There's a very good example of this in ode45. ode45 is based on the Dormand-Prince 5(4) pair. However, in 1999, the same year the MATLAB ODE Suite was published, [Shampine released a paper with a new 5\(4\) pair which was more efficient than the Dormand-Prince method](#). So that begs the question, why wasn't this used in the MATLAB ODE Suite (it's clear Shampine knew about it!)? (I actually asked him in an email) The reason is because its interpolation requires calculating some extra steps, so it's less efficient if you are ALWAYS interpolating. But since ode45 is always interpolating in order to make the plots look nicer, this would get in the way. Essentially, it can be more efficient, but MATLAB sets things up for nice plotting and not pure efficiency.

But there are other areas where more efficient methods were passed up during the development phase of the ODE suite. For example, Hairer's benchmarks in his book [Solving Ordinary Differential Equations I and II \(the second is for stiff problems\)](#), along with the [benchmarks from the Julia DifferentialEquations.jl suite](#), consistently show that high order Runge-Kutta methods are usually the most efficient methods for high accuracy solving of nonstiff ODEs. These benchmarks both consistently show that, for the same error, high order Runge-Kutta methods (like order >6) can solve the equation much faster than methods like Adams methods. But MATLAB does not offer high order Runge-Kutta methods and only offers ode113 (an Adams method) for high-accuracy solving.

Some of this is due to a limitation within MATLAB itself. MATLAB's ODE solver requires taking in a user-defined function, and since this function is defined in MATLAB its function calls are very inefficient and expensive. Thus MATLAB's ODE solver suite can become more efficient by using methods which reduce the number of function calls (which multistep methods do). But this isn't the only case where efficient methods are missing. Both Hairer and the JuliaDiffEq benchmarks show that high-order Rosenbrock methods are the most efficient for low-medium accuracy stiff ODEs, but MATLAB doesn't offer these methods. It does offer ode23s, a low-order Rosenbrock method, and ode15s which is a multistep method. ode15s is supposed to be the "meat and potatoes of the MATLAB Suite", but it

cannot handle all equations of since its higher order methods (it's adaptive order) are not L-stable (and not even A-stable). For this reason there's a few other low order SDIRK methods (ode23tb, an ESDIRK method for highly stiff problems) which are recommended to fill in the gaps, but none of the higher order variants which are known to be more efficient for many equations.

This pattern goes beyond the ODE solvers. The DDE solvers are all low order, and in the case of ddesd, it's a low accuracy method which is fast for getting plots correct but not something which converges to many decimal places all too well since it doesn't explicitly track discontinuities. This is even seen in the paper on the method [which shows the convergence only matches dde23 to graphical accuracy on a constant-delay problem](#). Again, this fits in with the mantra of the suite, but may not hit all demographics. [Shampine specifically made a separate version of ddesd for Fortran](#) for people who are interested in efficiency, which is another way of noting that the key of ddesd is features and automatic usage, and not hardcore scientific computing efficiency. The mentioned SDE solver from the financial toolbox [is only order 0.5](#) and thus requires quite a small dt to be accurate.

And I can keep going, but I think you get the moral of the story. This suite was created with one purpose in mind: to make it very easy to solve a wide array of differential equations and get a nice plot out. It does a very good job at doing so. But it wasn't made with efficiency in mind, and so it's missing a lot of methods that may be useful if you want high efficiency or high accuracy. Adding these wouldn't make sense to the MATLAB suite anyways since it would clutter the offering. MATLAB is about ease of use, and not efficiency, and it does extremely well at what it set out to do. For software that was first made before the Y2K crisis with just a few methods added later, it still holds up very well.

What would make sense to add to MATLAB are some "addon routines" though. Estimation of parameters in ODEs/DDEs, sensitivity analysis, etc. These are the kinds of things which users "can" do, but might not necessarily do "correctly" or in full (for example, for estimation of parameters you really want to use sensitivity analysis or autodifferentiation for the gradients, and in many cases you'll want to be using global or semi-global optimization methods. This is in contrast to simple Levenberg-Marquardt that I see code suggestions for in online threads). [AMIGO2](#) is a good MATLAB toolbox to add to the standard library which adds these kinds of functionality, but MATLAB still gets a "None" in this category since I am only considered the built in features there. But while I'm at it, [MATLAB does have a good Runge-Kutta Nystrom implementation](#) in the FileExchange and it really should be cleaned up and added to the standard library. I think this kind of focus on expanding features would be a good fit for MATLAB.

HAIRER'S SOLVERS (FORTRAN)

Next I want to bring up some Fortran solvers because they will come up later. [Hairer's Fortran solvers](#) are a set methods with similar interfaces that were designed with efficiency in mind. Many of these methods are classics: dopri5, dop853, radau, and rodas will specifically show up in many of the suites which are discussed later. These methods are not too flexible: they don't allow event handling (though with enough gusto you can use the dense output to write your own), or numbers that aren't double-precision floating point numbers (it's Fortran). They have a good set of options for tweaking parameters to make the adaptive timestepping more efficient, though you may have to read a few textbooks to know exactly what they do. And that's the key to them: they will solve an ODE, stiff or non-stiff, and they will do so pretty efficiently, but nothing more. But even then, they show some age which don't make them "perfectly efficient". These solvers include their own linear algebra routines which don't multithread like standard BLAS and LAPACK implementations, meaning that they won't make full use of modern CPU architectures. The computations don't necessarily SIMD or use FMA. But most of all, to use it directly you need to use Fortran which would be turn off for many people.

There is some flexibility in the offering. There are symplectic solvers for second order ODEs, the stiff solvers allow for solving DAEs in mass matrix form, there's a constant-lag nonstiff delay differential equation solver (RETARD), there is a fantastic generalization of radau to stiff state-dependent delay differential equations (RADAR5), and there's some solvers specifically for some "mechanical ODEs"

commonly found in physical problems. Of course, to get this all working you'll need to be pretty familiar with Fortran, but this is a good suite to look at if you are.

ODEPACK AND NETLIB ODE SOLVERS (FORTRAN)

ODEPACK is an old set of ODE solvers which accumulated over time. You can learn a bit about its history by [reading this interview with Alan Hindenmarsh](#). I also bundle [the Netlib suite together with it](#). There's a wide variety of methods in there. There's old Runge-Kutta methods due to Shampine, some of Shampine's old multistep methods `ddebd` and `ddeabm`, etc. The reason why this pack is really important though is because of the Lawrance Livermore set of methods, specifically LSODE and its related methods (LSODA, LSODR, VODE, etc.). It includes methods for implicit ODEs (DAEs) as well. These are a group of multistep methods which are descendent from GEAR, the original BDF code. They are pretty low level and thus allow you to control the solver step by step, and some of them have "rootfinding capabilities". This means that you can use these to add event handling, though an event handling interface will take some coding. There's lots of options and these are generally pretty performant for a large array of problems, but they do show their age in the same way that the Hairer codes do. Their linear algebra setups do not make use of modern BLAS and LAPACK, which in practical terms means they don't make full use of modern computer architectures to speed things up. They don't always make use of SIMD and other modern CPU acceleration features. They are the vanilla ODE solvers which have existed through time. In particular, LSODA is popular because this is the most widely distributed method which automatically detects stiffness and switches between integration methods, though it should be pointed out that there is a performance penalty from this feature.

While its ODE solvers are not particularly interesting anymore (pretty much every ODE solver since was created to improve upon the designs here), there are many non-standard pieces that you still cannot find elsewhere. For example, the BVP solver COLDAE is probably the best BVP solver for DAEs still. It has a multiple shooting BVP solver as well which allows you to give it integration schemes (MUS). It has Runge-Kutta Chevyshev (RKC) methods for first and second order ODEs which are highly-stable methods for semi-stiff equations (typically noted as good for parabolic PDEs). It also has an implicit RKC method as well. It has a MIRK-based adaptive BVP solver, similar to MATLAB's `bvp4c` but with a Fortran implementation. And the implicit ODE (DAE) solvers DASSL, DASKR, and DASPK were the precursors to Sundials' IDA and have different characteristics which can be good/bad depending on the problem. Altogether, there's some interesting stuff here, but the interfaces are not too uniform.

NAG ODE SOLVERS (FORTRAN)

The **NAG ODE solvers** are another classic set of Fortran solvers. If you take a look through them, you'll notice they not only kept ODEPACK and Netlib in their mind... they are very similar. It has a few Runge-Kutta methods (directly derived from RKSUITE), and a multistep solver with Adams and BDF versions. Like the other Fortran suite, it has "event handling" in terms of "you can control it step-by-step and have a rootfinder, so you can write your own event handling". They do hold your hand a little bit more [by providing driver functions which do some cool stuff](#), but this is still definitely the old-school way of having a feature. It also has a good set of BVP solvers covering a wide range of algorithms, similar to ODEPACK and Netlib. While this is one solver I haven't used myself, I am sure that its efficient like the other Fortran methods, but it hasn't updated with implicit Runge-Kutta or Rosenbrock methods which are known to be more efficient in many cases (in their modern form). Thus for the chart I am just going to clump them with ODEPACK and Netlib and think about them as an additional set of methods to add to that group.

SUNDIALS AND ARKCODE (C++ AND FORTRAN)

Sundials' CVODE is a re-write of VODE to C which is a descendent of LSODE which is a descendent itself of the original GEAR multistep code. Yes, this has a long history. But you should think of it as "LSODE upgraded": it makes use of modern BLAS/LAPACK, and also a bunch of other efficient

C/Fortran linear solvers, to give a very efficient Adams and BDF method. Its solver IDA is like CVODE but handles implicit ODEs (DAEs). The interface for these is very similar to the ODEPACK interface, which means you can control it step by step and use the rootfinding capabilities to write your own event handling interface. Since the Adams methods handle nonstiff ODEs and the BDF methods handle stiff ODEs, this performance plus flexibility makes it the "one-stop-shop" for ODE solving. Many different scientific computing software internally are making use of Sundials because it can handle just about anything. Well, it does have many limitations. For one, this only works with standard C++ numbers and arrays. There's no stochasticity or delays allowed. But more importantly, Hairer and DifferentialEquations.jl's show that these multistep methods are usually not the most efficient methods since high order Rosenbrock, (E)SDIRK, and fully implicit RK methods are usually faster for the same accuracy for stiff problems, and high order Runge-Kutta are faster than the Adams methods for the same accuracy. Multistep methods are also not very stable at their higher orders, and so at higher tolerances (lower accuracy) these methods may fail to have their steps converge on standard test problems (see this note in the ROBER tests), meaning that you may have to increase the accuracy (and thus computational cost) due to stiffness issues. But, since multistep methods only require a single function evaluation per step (or are implicit in only single steps), they are the most efficient for asymptotically hard problems (i.e. when the derivative calculation is very expensive or the ODE is a large 10,000+ system). For this reason, these methods excel at solving large discretizations of PDEs. To top it off, there are parallel (MPI) versions for using CVODE/IDA for HPC applications.

I also want to note that recently Sundials added **ARKCODE**, a set of Runge-Kutta methods. These include explicit Runge-Kutta methods and SDIRK methods, including additive Runge-Kutta methods for IMEX methods (i.e. you can split out a portion to solve explicitly so that the implicit portion is cheaper when you know your problem is only partly or semi stiff). This means that it covers the methods which I mentioned earlier are more efficient in many of the cases (though it is a bit lacking on the explicit tableaus and thus could be more efficient, but that's just details). However, benchmarks show that ARKODE is not implemented with the same quality as CVODE, with its default settings usually diverging on stiff problems and tweaked parameters not performing well in benchmarks (see the edits 4/21/2018 for details). Thus it should be considered separate from the rest of the "Sundials brand", giving a form of completeness but not with the same level of performance as you would expect.

If you are using C++ or Fortran and want to write to only one interface, the Sundials suite is a great Swiss Army knife. And if you have an asymptotically large problem or very expensive function evaluations, this will be the most efficient as well. Plus the parallel versions are cool! You do have to live with the limitations of low-level software forcing specific number and array types, along with the fact that you need to write your own event handling, but if you're "hardcore" and writing in a compiled language this suite is a good bet.

SCIPY'S SOLVERS (PYTHON)

Now we come to **SciPy's suite**. SciPy 1.0 includes some of its own Runge-Kutta methods, and has tableaus for Dormand-Prince 4/5 and Bogacki-Shampine 2/3. However, its basic Runge-Kutta integrator is written directly in Python with loops, utilizes **an old school timestepping method instead of newer more efficient ones** (this makes it less stable and more likely to diverge than most implementations), it doesn't have very many options, etc. So this is a definite step backwards in terms of "hardcore efficiency" and the features for optimizing an RK method to a given problem. However, this is a big step forward because it allows the methods to solve on matrices and tensors instead of just arrays, allows for a few of the methods to have dense output, and more flexibility. These same changes were made to its BDF method as well. The developer chats **seem to say it's around a 10x performance loss** for problems with around 200 ODEs, smaller problems getting more overhead and larger ones getting less (due to the use of vectorization when possible).

Using a Python interface with classes and the like, it does offer a step-by-step interface. It now has some basic event handling, but for example **it only checks the endpoints so if your function isn't monotonic it can easily miss it** along with other things like **floating point issues which come from not**

doing a pullback, so it's still at the point where it's a decent undergraduate homework problem to ask them to find a way to break it (there are several quick ways). This isn't just theoretical, users have posted more than [issue with easy ways to break its event handling](#), so even for simple cases use caution. It still offers wrappers to LSODA and radau (Hairer's and ODEPACK's methods). It has dop853 and dopri5 hidden in an old interface. In the developer's tests these are more efficient, but they don't have these nice extra features like post-solution interpolation. It does have a big options list, so if you were planning on swapping out the linear solver for PETSc you're out of luck, but it does expose some of the internal options of LSODA for banded Jacobians.

It only has ODE solvers, no differential-algebraic, delay, or stochastic solvers. It does include a single BVP solver though. The tableaux are all stored at double precision so even if higher precision numbers are accepted it wouldn't give a higher precision result. As with the methods analysis, its only higher order Runge-Kutta method for efficient solving of nonstiff equations is dop853 which is now buried in the legacy interface without extra features and it's missing Rosenbrock and SDIRK methods entirely, opting to only provide the multistep methods.

I should note here that it has the same limitation as MATLAB though, namely that the user's function is Python code. Since the derivative function is where the ODE solver spends most of its time (for sufficiently difficult problems), this means that even though you are calling Fortran code, you will lose out on a lot of efficiency. Still, if efficiency isn't a big deal and you don't need bells and whistles, this suite will do the basics. While before it was far behind MATLAB, SciPy 1.0's big update put it more mildly behind Python. It handles a much smaller domain and is missing a bunch of methods, but if your problem is simple enough it'll do it and can handle some basic events as well. It's nice to teach a class with and make some plots, and then you can go grab a Sundials wrapper when you need it.

DESOLVE PACKAGE (R)

There's not much to say other than that [deSolve](#) is very much like SciPy's suite. It wraps almost the same solvers, has pretty much the same limitations, and has the same efficiency problem since in this case it's calling the user-provided R function most of the time. One advantage is that it does have event handling. Less vanilla with a little bit more features, but generally the same as SciPy.

I do want to make a note about its delay differential equation solvers dede. The documentation specifically says:

dede does not deal explicitly with propagated derivative discontinuities, but relies on the integrator to control the stepsize in the region of a discontinuity.

dede does not include methods to deal with delays that are smaller than the stepsize, although in some cases it may be possible to solve such models.

For these reasons, it can only solve rather simple delay differential equations.

As a methods researcher, I am horrified by this. Here's a few things wrong with it:

1. The error estimator is only at the end of the interval, meaning it's a very bad check for discontinuities. This is why ddesd used a special residual. None of their linked methods do this.
2. Not propagating discontinuities means order loss to 1.
3. They allow using a multistep method over a discontinuity. This might converge to order 1? There's no theory for this.
4. When delays are smaller than the step size, the method is actually implicit, even if the ODE solver it's using is explicit. What this means is that this algorithm in this case is essentially a zero iteration Picard solver for the fixed point problem. This is probably the worst part.

They end with saying it's only for "simple" delay equations without mentioning these as the reasons. I think that's going too far: there is no good guarantee that the error is low or the method converges well. I will mark it as "Poor" in the table, but I would advise against using this method (and would recommend

the developers to remove this from the docs because this is definitely an easy place for users to be misled by a numerical result).

PYDSTOOL (PYTHON)

PyDSTool is an odd little beast. Part of the software is for analytic continuation (i.e. bifurcation plotting). But another part of it is for ODE solvers. It contains one ODE solver which is written in Python itself and it recommends against actually using this for efficiency reasons. Instead, it wraps some of the Hairer methods, specifically **dopri5** and **radau**, and recommends these. But it's different than SciPy in that it takes in the specification of the ODE as a string, and compiles it to a C function, and uses this inside the ODE solver. By doing so, it's much more efficient. We still note that its array of available methods is small and it offers **radau** which is great for high accuracy ODEs and DAEs, but is not the most efficient at lower accuracy so it would've been nice to see Rosenbrock and ESDIRK methods. It has some basic event handling and methods for DDEs (again as wrappers to a Hairer method). This is a pretty good suite if you can get it working, though I do caution that getting the extra (non-Python) methods setup and compiled is nontrivial. One big point to note is that I find the documentation spectacularly difficult to parse. Together, it's pretty efficient and has a good set of methods which will do basic event handling and solve problems at double precision.

JITCODE AND JITCSDE (PYTHON)

JITCODE is another Python library which makes things efficient by compiling the function that the user provides. It uses the SciPy integrators and does something similar to PyDSTool in order to get efficiency. I haven't tried it out myself but I'll assume this will get you as efficient as though you used it from Fortran. However, it is lacking in the features department, not offering advanced things like arbitrary number types, event handling, etc. But if you have a vanilla ODE to solve and you want to easily do it efficiently in Python, this is a good option to look at.

Additionally, **JITCDDDE** is a version for constant-lag DDEs similar to **dde23**. **JITCSDE** is a version for stochastic differential equations. It uses the high order (strong order 1.5) adaptive Runge-Kutta method for diagonal noise SDEs developed by Rackauckas (that's me) and Nie which has been demonstrated as much more efficient than the low order and fixed timestep methods found in the other offerings. It employs the same compilation setup as **JitCODE** so it should create efficient code as well. I haven't used this myself but it would probably be a very efficient ODE/DDE/SDE solver if you want to use Python and don't need events and other sugar.

BOOST ODEINT SOLVER LIBRARY (C++)

The **Boost ODEINT solver library** has some efficient implementations of some basic explicit Runge-Kutta methods (including high order RK methods) and some basic Rosenbrock methods (including a high order one). Thus it can be pretty efficient for solving the most standard stiff and nonstiff ODEs. However, its implementations do not make use of advanced timestepping techniques with its explicit RK method (PI-controllers) which makes it require more steps to achieve the same accuracy as some of the more advanced software, making it not really any more efficient in practice (though it does do Gustafsson acceleration in its Rosenbrock scheme to prevent "the Hump" behavior). It doesn't have event handling, but it is flexible with the number and array types you can put in there via C++ templates. It and **DifferentialEquations.jl** are the only two suites that are mentioned that allow for solving the differential equations on the GPU. Thus if you are familiar with templates and really want to make use of them, this might be the library to look at, otherwise you're probably better off looking elsewhere like **Sundials**.

GSL ODE SOLVERS (C)

To say it lightly, the **GSL ODE solvers** are kind of a tragedy. Actually, they are just kind of weird. When comparing their choices against what is known to be efficient according to the ODE research and benchmarks, the methods that they choose to implement are pretty bizarre like extrapolation methods which have repeatedly been shown to not be very efficient, while not included other more important

methods. But they do have some of the basics like multistep Adams and BDF methods. This, like Boost, doesn't do all of the fancy PI-controlled adaptivity and everything though, so YMMV. **This benchmark**, while not measuring runtime and only uses function evaluations (which can be very very different according to more sophisticated benchmarks like the Hairer and DifferentialEquations.jl ones!), clearly shows that the GSL solvers can take way too many function evaluations because of this and thus, since it's using methods similar to LSODA/LSODE/dopri5, probably have much higher runtimes than they should.

MATHEMATICA'S ODE SOLVERS

Mathematica's ODE solvers are very sophisticated. It has a lot of explicit Runge-Kutta methods, including newer high order efficient methods due to Verner and Shampine's efficient method mentioned in the MATLAB section. These methods can be almost 5x faster than the older high order explicit RK methods which themselves are the most efficient class of methods for many nonstiff ODEs, and thus these do quite well. It includes interpolations to make their solutions continuous functions that plot nicely. Its native methods are able to make full use of Mathematica and its arbitrary precision, but sadly most of the methods it uses are wrappers for the classic solvers. Its stiff solvers mainly call into Sundials or LSODA. By using LSODA, it tends to be able to do automatic stiffness detection by default. It also wraps Sundials' IDA for DAEs. It uses a method of steps implementation over its explicit Runge-Kutta methods for solving nonstiff DDEs efficiently, and includes high order Runge-Kutta methods for stochastic differential equations (though it doesn't do adaptive timestepping in this case). One nice feature is that all solutions come with an interpolation to make them continuous. Additionally, it can use the symbolic form of the user-provided equation in order to create a function for the Jacobian, and use that (instead of a numerical differentiation fallback like all of the previous methods) in the stiff solvers for more accuracy and efficiency. It also has symplectic methods for solving second order ODEs, and its event handling is very expressive. It's very impressive, but since it's using a lot of wrapped methods you cannot always make use of Mathematica's arbitrary precision inside of these numerical methods. Additionally, its interface doesn't give you control over all of the fine details of the solvers that it wraps.

MAPLE'S ODE SOLVERS

Maple's ODE solvers has the basics. It for some reason has Fehlberg and Cash-Karp order 5/4 methods instead of the Dormand-Prince, Bogacki-Shampine, or Tsitorious ones. It has a classic higher order explicit RK method due to Verner (not the newer more efficient ones though), and for stiff problems it uses a high order Rosenbrock method. It also wraps LSODE like everything else. Its native methods can make use of arbitrary arithmetic and complex numbers. It has a BVP solver, but it explicitly says that it's not for stiff equations or ones with singularities (it relies on deferred correction or extrapolation). It can solve state-dependent delay differential equations with its explicit RK and Rosenbrock methods. It can solve SDEs using Euler-Maruyama. It has event handling which doesn't seem to be well documented. As another symbolic programming language, it computes Jacobians analytically to pass to the stiff solvers like Mathematica, helping it out in that respect. Thus while some of its basics are a little odd (why a Fehlberg method? The Dormand-Prince paper is one of the most famous papers about increasing efficiency in explicit RK tableaux) and not as fleshed out, it does have a good variety with quite a bit of capabilities. I would actually go as far as to say that, if I had to create a suite with a small set of solvers I'd probably make something similar to what Maple has. It matches what seems to benchmark best "for most problems in normal tolerances", which is what most people are probably solving.

FATODE (FORTRAN)

FATODE is a set of methods written in Fortran. It includes explicit Runge-Kutta methods, SDIRK methods, Rosenbrock methods and fully implicit RK methods. Thus it has something that's pretty efficient for pretty much every case. What makes it special is that it includes the ability to do sensitivity analysis calculations. It can't do anything but double-precision numbers and doesn't have event handling, but the sensitivity calculations makes it quite special. If you're a FORTRAN programmer, this

is worth a look, especially if you want to do sensitivity analysis.

DIFFERENTIALEQUATIONS.JL (JULIA)

Okay, time for [DifferentialEquations.jl](#). I left it for last because it is by far the most complex of the solver suites, and pulls ideas from many of them. While most of the other suite offer no more than about 15 methods on the high end (with most offering about 8 or less), DifferentialEquations.jl offers 200+ methods and is continually growing. Like the standard Python and R suites, it offers wrappers to Sundials, ODEPACK, and Hairer methods. However, since Julia code is always JIT compiled, its wrappers are more akin to PyDSTool or JiTCODE in terms of efficiency. Thus all of the standard methods mentioned before are available in this suite.

But then there are the native Julia methods. For ODEs, these include explicit Runge-Kutta methods, (E)SDIRK methods, and Rosenbrock methods. In each of these categories it has a huge amount of variety, offering pretty much every method from the other suites along with some unique methods. Some unique methods to point out are that it has the only 5th order Rosenbrock method, it has the efficient Verner methods discussed in the Mathematica part, it has newer 5th order methods (i.e. it includes the Bogacki-Shampine method discussed as an alternative to ode45's tableau, along with an even newer tableau due to Tsitorious which is even more efficient). It has methods specialized to reduce interpolation error (the OwrenZen methods), and methods which are strong-stability preserving (for hyperbolic PDEs). It by default the solutions as continuous functions via a high order interpolation (though this can be turned off to make the saving more efficient). Each of these implementations employ a lot of extra tricks for efficiency. For example, the interpolation is "lazy", meaning that if the method requires extra function evaluations for the continuous form, it will only do those extra calculations when the continuous function is used (so when you directly ask for it or when you plot). This is just a peek at the special things the library does to gain an edge.

The native Julia methods benchmark very well as well, and all of the benchmarks are openly available. Essentially, these methods make use of the native multithreading of modern BLAS/LAPACK, FMA, SIMD, and all of the extra little compiler goodies that allows code to be efficient, along with newer solver methods which theoretically reduce the amount of work that's required to get the same error. They even allow you to tweak a lot of the internals and swap out the linear algebra routines to use parallel solvers like PETSc. The result is that these methods usually outperform the classic C/Fortran methods which are wrapped. Additionally, it has ways to symbolically calculate Jacobians like Mathematica/Maple, and instead of defaulting to numerical differentiation the stiff solvers fall back to automatic differentiation which is more efficient and has much increased accuracy. There is built-in parallelism for solving DEs in Monte Carlo which works with Julia's native distributed parallelism (and has been tested on SGE and Slurm clusters thanks to [XSEDE](#)).

Its event handling is the most advanced out of all of the offerings. You can change just about anything. You can make your ODE do things like change its size during the solving if you want, and you can make the event handling adjust and adapt internal solver parameters. It's not a hyperbole to say that the user is given "full control" since the differential equation solvers themselves are written as essentially a method on the event handling interface, meaning that anything it can do internally you can do.

The variety of methods is also much more diverse than the other offerings. It has symplectic integrators like Hairer's suite, but has more high and low order methods. It has a range of Runge-Kutta Nystrom methods for efficiently solving second order ODEs. It has the same high order adaptive method for diagonal noise SDEs as JiTCSDE, but also includes high order adaptive methods specifically for additive noise SDEs. It also has methods for stiff SDEs in Ito and Stratanovich interpretations, and allows for event handling in the SDE case (with the full flexibility). It has DDE solvers for constant-lag and state-dependent delays, and it has stiff solvers for each of these cases. The stiff solvers also all allow for solving DAEs in mass matrix form (though fully implicit ODEs are possible, but can only be solved using a few methods like a wrapper to Sundials' IDA and doesn't include event handling here

quite yet).

It allows arbitrary numbers and arrays like Boost. This means you can use arbitrary precision numbers in the native Julia methods, or you can use "array-like" objects to model multiscale biological organisms instead of always having to use simple contiguous arrays. It has addons for things like sensitivity analysis and parameter estimation. Also like Boost, it can solve equations on the GPU by using a GPUArray.

It hits the strong points of each of the previously mentioned suites because it was designed from the get-go to do so. And it benchmarks extremely well. The only downside is that, because it is so feature and performance focused, its documentation is heavy. The beginning tutorial will give you the basics (making it hopefully as easy as MATLAB to pick up), but pretty much every control knob is available, making the extended portion of the documentation a long read.

CONCLUSION

Let's take a step back and summarize this information. DifferentialEquations.jl objectively has the largest feature-set, swamping most of the others while wrapping all of the common solvers. Since it also features solvers for the non-ordinary differential equations and its unique Julia methods also benchmarks well, I think it's clear that DifferentialEquations.jl is by far the best choice for "power-users" who are looking for a comprehensive suite.

As for the other choices from scripting languages, MATLAB wasn't designed to have all of the most efficient methods, but it'll handle basic equations with delays and events and output good plots. R's deSolve is similar in most respects to MATLAB. SciPy's offering is lacking in comparison to MATLAB and R's due to the lack of event handling. But MATLAB/Python/R all have efficiency problems due to the fact that the user's function is written in the scripting language. JiTCODE and PyDSTool are two Python offerings make the interface to the Fortran solvers more efficient than straight SciPy. Mathematica and Maple will do symbolic pre-calculations to speed things up and can JiT compile functions, along with offering pretty good event handling, and thus their wrappers are more like DifferentialEquations.jl in terms of flexibility and efficiency (and Mathematica had a few non-wrapper goodies mentioned as well). So in a pinch when not all of the bells and whistles are necessary, each of these scripting language suites will get you by. Behind DifferentialEquations.jl, I would definitely put Mathematica's or Maple's suite second for scripting languages with everything else much further behind. I was actually kind of surprised: normally people think of a CAS as for symbolic computing and think of MATLAB/R/Python as more "numerical" languages, but at least in the case of differential equation solvers the CASs (Mathematica and Maple) seem to be much more developed and complete.

If you're already hardcore and writing in C++/Fortran, Sundials is a pretty good "one-stop-shop" to get everything you need, especially when you add in ARKCODE. Still, you're going to have to write a lot of stuff yourself to make the rootfinding into an event handling interface, but if you put the work in this will do you well. Hairer's codes are a great set of classics which cover a wide variety of equations, and FATODE is the only non-DifferentialEquations.jl suite which offers a way to calculate sensitivity equations (and its sensitivity equations are more advanced). Any of these will do you well if you want to really get down-and-dirty in a compiled language and write a lot of the interfaces yourself, but they will be a sacrifice in productivity with no clear performance gain over the scripting language methods which also include some form of JIT compilation. With these in mind, I don't really see a purpose for the GSL or Boost suites, and the ODEPACK methods are generally outdated.

I hope this review helps you make a choice which is right for you.

EDITS: 9/26/2017

Shortly after being posted this has received quite a bit of buzz. I am thankful so many people are interested! A few things were pointed out to me that I have since corrected:

1. I didn't have a row for implicitly defined DAEs in the chart. This has been added. The classic

softwares were placed at excellent because they do give you step-by-step control of BDF methods to in theory allow you to build whatever interface you want on top of it. MATLAB and Mathematica are Good because you can interface with them and use their respective event handling interfaces. DifferentialEquations.jl is only fair since while we do wrap more of these than other software ecosystems, we don't have a way of using events or anything fancy here. In a few days I will be posting about the release of 3.0 and this is actually one of the big bullet points on our roadmap.

2. I upgraded the efficiency of Sundials to "Excellent" when ARKODE is considered. While I still think that they need newer/better explicit RK tableaus to really handle nonstiff equations to the best of their ability (the high order Verner methods for sure would be a great improvement), their offering does hit most of the other points (except for missing Rosenbrock methods) and it is very well-tuned. In addition, they make full use of additive RK methods for IMEX which can be very helpful in semi-stiff problems.
3. For now I am ignoring PDEs. The reason is because it's too hard to judge what matters. MATLAB and Mathematica both have diffusion-advection equation solvers, so does that mean that they are good for solving PDEs? Not necessarily to someone who wants to solve the Hamilton-Jacobi Bellman equation. The sensible thing to do instead would be to either do it at a very fine level and also compare the available toolboxes. But going through the details of the different finite element method toolboxes is a huge task. PDEs will likely be their own blog post in the future.

There are probably a few other corrections needed. For one, I may have put PyDSTool too high in some areas like state-dependent delay solvers (I had notes that say it can't do it, but I can't find it in the docs anymore...), so please feel free to send corrections.

EDITS 2: 9/26/2017

More edits today. This time for Maple. I missed some things that Maple has and updated the discussion about Maple to include its BVP/DDE solvers and its ability to handle arbitrary precision and complex numbers.

EDITS: 9/28/2017

Some more refinements. Mostly adding extra capabilities in the Netlib part of the chart (there's a lot of interesting stuff in there). Also, I wanted to note that **PyODESys** was pointed out to me which will take your ODE in Python and calculate Jacobians automatically and send them along to the ODE solver. While this is not part of any of the mentioned libraries and thus doesn't change the chart, this is something SciPy users might want to know about.

EDITS: 9/29/2017

This has become quite a living document. I got some questions about development goals, which was actually the purpose of this in the first place, so let me share some concise recommendations. These are recommendations for next developments which would be the most beneficial for each. I'll omit the tools which are essentially "done" like Netlib.

1. MATLAB. I would really like to see a higher-order explicit RK method (one of the Verner efficient ones) and an implementation of Radau. That would really round out the offering. Tie ins with symbolic differentiation and autodifferentiation tools would be very helpful as well. A much larger goal would be to add discontinuity tracking to their state-dependent delay equation solver.
2. SciPy. SciPy needs a lot of work. But two very simple steps they can take are to wrap Radau and Rodas. Since they already wrap two of Hairer's methods, it can't be that bad. A larger goal would be to expose dense output and add an event handling interface. I would like it if they picked a JiT setup to go along with their offering since that would go a long way, but it's okay if those remain extra tools.
3. R deSolve. Add a Rodas wrapper. Add a higher order explicit RK method. Once again, a way to compile user-functions would be great but is a larger project
4. DifferentialEquations.jl. Implicitly-defined ODEs can use some help since the solvers in this domain are just simple wrappers to IDA and DASKR, meaning that they don't get the same features (types,

event handling, etc.) as the rest of the suite. The BVP solvers can do a lot of interesting stuff but have a weak core which should be improved.

5. Mathematica. Add some Rosenbrock methods. These methods do extremely well when Jacobians are given analytically, so Mathematica is really missing an opportunity more than others by not having them. Also a state-dependent delay equation solver. As noted in the comments, I think a good simple step for Mathematica would be adding some parameter inference routines as well which put together their sensitivity analysis and optimization into some easier setups.
6. Maple. Document what you are doing. Documentation is what I would like to see more of than anything else here. Adding a BVP solver which can handle stiff systems along with implicitly-defined DAEs would go a long way.
7. Boost. Add PI-controllers for the RK methods. It's not a hard change but can make a big difference. Also, it needs more than just a single 4th order Rosenbrock method for stiff equations. Probably add an adaptive order/timestep BDF for large equations, and a Radau.
8. PyDSTool. It has parameter estimation tools but just uses Levenberg-Marquardt and Bounded Minimization. At least add L-BFGS and some global optimizer and then this would be a really interesting part of the suite. Additionally, this is another package which could add Rodas pretty easily and should.

EDITS: 11/11/2017

SciPy 1.0 was released and redid the ODE solvers quite a bit. Their goal was more flexibility, and they did well. Event handling now exists and it handles more than just vectors now. However, the developer chat does mention that this degrade the efficiency quite a bit (links in the discussion above) over just wrapping the Fortran solvers for the Runge-Kutta and BDF methods, so that is something to keep in mind. But it's much closer to MATLAB now. And it has a Radau wrapper. It actually hit most of the developer goals I hit above, so I think I'd recommend they expose a ton of options now, write compiled versions of their solvers, and make them more type-independent. If if they are just going for features, some analysis addons like uncertainty quantification and parameter estimation would be nice for their user-base.

I also now mention the NAG solvers by request. They were grouped with the Netlib solvers because, well, look at them side by side (or read the explanation above) and you'll know what I mean.

EDITS: 4/21/2018

Since the last edits, the biggest changes have been the Sundials 3.0/3.1 release (well, adoption, it came out days before the last edit), along with DifferentialEquations.jl v3.0 and v4.0 up to v4.4. Sundials is mostly unchanged except for "power uses", with its new SUNLINSOL and SUNMATRIX modules being very flexible ways to define and use your own linear solver routines for their implicit solvers. DifferentialEquations.jl's releases have included a lot more, including adaptive time stepping for non-diagonal SDEs, sensitivity analysis, automatic stiffness detection and switching, etc. DifferentialEquations.jl is also branching out to other languages with bindings to R via diffeqr.

DifferentialEquations.jl's latest releases also now wrap Sundials v3.1 along with ARKODE, adding these to the benchmarking suite. When ARKODE was ran through the comprehensive set of benchmarks its results were less exciting than one would've hoped. From [this benchmark you can see that they perform about an order of magnitude slower than the DifferentialEquations.jl counterparts](#), but it also highlights the fact that the ARKODE methods failed (diverged) on most of the stiff benchmark problems when using the default parameters. There is a fix to heavily decrease the nonlinear convergence coefficient, but this adversely effect effects time quite drastically. This fix is not my own hack, it's what's recommended in the Sundials examples, specifically the simplest Robertson problem. From some other tests it seems that this doesn't adversely effect the behavior on some semi-stiff diffusion-advection problems which may be the reason for these parameter choices, but it's interesting to note that this divergence occurs on pretty much every highly stiff problem. Part of the reason is due to odd defaults. Almost no algorithm uses a time stepping safety factor that's not 0.9, but ARKODE defaults to 0.96 which in tern puts it closer to the error/stability limits. That's just one of many

parameter defaults that is odd. In addition, if they implemented the algorithm like [they described at this site](#), then the implicit equations would be prone to divergence when there's stiffness. [Shampine noted that an altered form is more stable](#), and this altered form is used by DifferentialEquations.jl which can also explain the difference.

DifferentialEquations.jl hit an issue in one of the stiff DDE benchmarks. This makes the current ranking of stiff DDE solvers go RADAR5 (Hairer) > Maple > DifferentialEquations.jl. There is some active development to fix this.

EDITS: 4/30/2018

DifferentialEquations.jl released [bindings for R and Python](#). In this post there is an example showing calling the Julia suite from Python speeds up code by about 10x over SciPy+Numba, and calling it from R speeds up code 12x over deSolve. If you factor in that [MATLAB was found to be almost 100x slower than DifferentialEquations.jl on similar problems](#), that means R does well in comparison to MATLAB and so does SciPy if you JIT compile the ODE function. However, just JIT compilation this one function is still about an order of magnitude way from pure C++/Fortran/Julia codes.

EDITS: 7/14/2018

I made some updates to the chart. After a lot of testing of Sundials v3, I upped its tweakability to Excellent since the new SunMatrix and SunLinSolve interfaces are absolutely amazing for getting every little detail right, along with some of the new NVector types. Control over nonlinear solvers is coming in v4 too, and this will be something to really play with in order to get the stiff solvers really optimized to a problem. And after more testing I bumped them up to Excellent in the addons portion. While they do not do things like (Bayesian) parameter estimation, their sensitivity analysis portion is best-in-class. Tests repeatedly shows it as a very efficient implementation and the checkpointing scheme for adjoint sensitivities works really well for calculating gradients of PDE-constrained models. Thus even though it's only implemented a portion of the possible addons, what it does it does well so it deserves the Excellent.