Confederated modular differential equation APIs for accelerated algorithm development and benchmarking

Christopher Rackauckas*,a,b, Qing Nie b

a Department of Mathematics, Massachusetts Institute of Technology, MA 02139, USA
b Department of Mathematics, University of California, Irvine, CA 92697, USA

A B S T R A C T

Performant numerical solving of differential equations is required for large-scale scientific modeling. In this manuscript we focus on two questions: (1) how can researchers empirically verify theoretical advances and consistently compare methods in production software settings and (2) how can users (scientific domain experts) keep up with the state-of-the-art methods to select those which are most appropriate? Here we describe how the confederated modular API of DifferentialEquations.jl addresses these concerns. We detail the package-free API which allows numerical methods researchers to readily utilize and benchmark any compatible method directly in full-scale scientific applications. In addition, we describe how the complexity of the method choices is abstracted via a polyalgorithm. We show how scientific tooling built on top of DifferentialEquations.jl, such as packages for dynamical systems quantification and quantum optics simulation, both benefit from this structure and provide themselves as convenient benchmarking tools.

1. Introduction

Differential equations are a ubiquitous modeling tool across the biological to physical sciences. In most occasions, these equations have no analytical solution and thus must be solved numerically. However, each of the many families of integration methods utilize different structures of the underlying equations to achieve differing problem-specific efficiencies. Since performant handling of these equations is required for large-scale scientific modeling, an increasingly pressing question is two-fold: (1) how can researchers empirically verify theoretical advances and consistently compare many different methods in production-software settings and (2) how can users (scientific domain experts) keep up with the state-of-the-art methods to select the most appropriate integrator for their problem?

The DifferentialEquations.jl [18] software has rapidly developed into one of the most feature rich ecosystems for solving differential equations, including many unique features like adaptive stochastic differential equation integrators and arbitrary precision event handling [15]. In this manuscript we will describe the confederated modular API of the DifferentialEquations.jl metapackage and show how it has accelerated scientific progress by addressing both the concerns of methods researchers and users. We will begin by describing how the modular package-generic API with automatic composability through multiple dispatch mechanisms allows numerical methods researchers to readily substitute any compatible method directly into full-scale scientific applications for practical use, and also for testing and benchmarking. We will discuss how the needs of three distinct groups of users, methods developers, scientific package developers, and domain modelers, led to the evolution of this mechanism and the problems it has solved.

On the other hand, the common API has led to a rapid expansion in the available algorithm choices and this complexity has led to new problems for end-users who tend to be less familiar with all of the nuances of the field. To handle the growing complexity, we show how the method choices have been abstracted by automatic algorithm specialization via a type-hierarchy classification of differential equation structure and an associated polyalgorithm. We show how the software ecosystem built around this tooling, such as packages for quantum optics and real-time robotics simulation, both benefit from this structure and provide themselves as convenient benchmarking tools. We end by discussing how further automation is being added to the framework and how this software architecture can be replicated in other scientific domains.

2. The confederated modular API

The common API is the API across all of the different solver packages in DifferentialEquations.jl and is diagrammed in Fig. 1.

* Corresponding author.
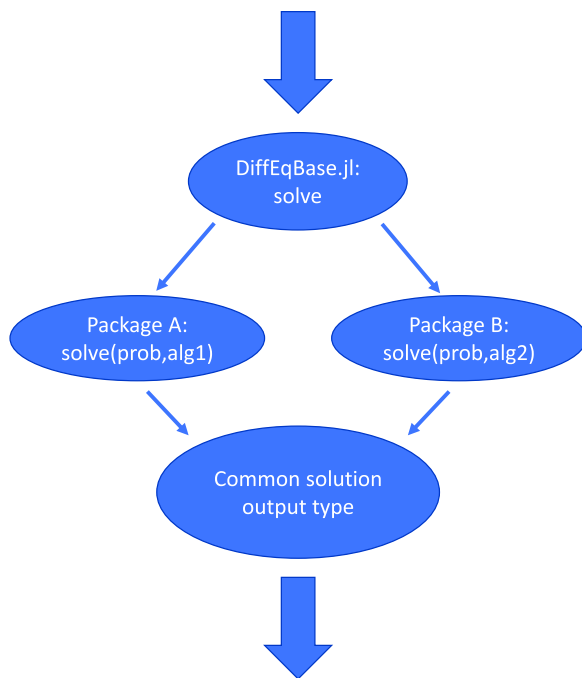 E-mail addresses: accounts@chrisrackauckas.com, crackauc@mit.edu (C. Rackauckas).

Fig. 1. The confederated modular API. The confederated modular API, known as the common API, is called by the user with the solve command. This function is defined in DiffEqBase, but problem and algorithm types are used to dispatch to separate packages. These packages interpret the differential equation problem type (ODE, SDE, etc.) and solve it using the documented algorithm. Each of these packages return a common solution type with a defined API. The user then continues their work using this common output. Note that if not algorithm is given by the user, the algorithm for dispatch is decided automatically.

DifferenitalEquations.jl's structure is defined in the base package DiffEqBase.jl. Its main purpose is to hold the function definitions which are utilized at all levels of the API. The Julia programming language is built around a multiple dispatch system which allows for specializing function calls via methods which are dependent on the inputs [2]. The multiple dispatch mechanism is usually discussed in terms of its ability to allow for code specialization for high performance [1]. For example, the function call `+(2,5)` lowers to a different method than `+(2.0,5.0)` which allows a Julia programmer and the compiler to generate efficient native code that is efficient for each case. The Julia compiler will utilize known type information at compilation time (while making use of type inference) to allow for this structure to have no runtime overhead which leads to performance comparable to statically-typed languages like C or Fortran.

However, the confederated modular API utilizes the multiple dispatch mechanism as a means for allowing input arguments to choose underlying integration methods. The core function is the `solve` function. This is defined as a "stub function" in DiffEqBase.jl, i.e. a function without a method which thus has no functionality. The documented API is that `solve` is called via `solve(prob,alg)` where `prob` is some problem type which describes a differential equation in a chosen family and `alg` is the algorithm type. For example, the `ODEProblem` type has fields for `f`, `u0`, `tspan`, and `p` which define the ODE system:

$$u' = f(u, p, t), \quad u(t_0) = u0.$$

The algorithm type is used simply to dispatch to the solver algorithm to return a solution. An example of user code for solving the Lorenz ODE system by the 5th order explicit Runge–Kutta method [22] is shown in Algorithm 1.

The key feature of this design is that any Julia code can extend the solve function by defining a method dispatch on their algorithm type,

allowing for their code to be used to generate the solution. An example of defining the Euler method for an ODEProblem is shown in Algorithm 2.

This design means that the choices of methods available to the user are not built into the function itself. Rather, every differential equation solver method is defined through a common extension system, and a large set of packages defining integrator methods comprises the JuliaDiffEq organization. However, this extension system does not have a core central authority as any author in any package can extend the function like is shown in Algorithm 2. This is why we describe this system as a confederated modular API (it is colloquially referred to as the common API). Additionally, the common API has documented options (abstol, reltol, etc.) and event-handling choices to allow for all algorithms to act the same on the user's input and thus abstract the package-free nature away from the user.

3. Direct benefits of the API design

The confederated modular API has greatly accelerated integrator methods development in the Julia differential equations community in a few distinct ways. One major way is that it allows researchers to develop implementations which are not part of a central repository. Easy-to-use software helps to facilitate impact of research. Yet, in the highly competitive academic atmosphere, many individuals simply cannot justify donating time to larger open-source projects. The confederated modular API allows these individuals to build methods in their own repositories under their own name, but be utilized by end users just like any other JuliaDiffEq solver. One prominent case of this is LSODA.jl [23] which wraps a thread-safe C++ re-write of the classic LSODA algorithm [5,14], but is not encompassed in the JuliaDiffEq organization since its utilization and maintenance is tracked as part of the individual's academic achievements.

Another reason the confederated API has been a success is that it allows for solver methods to be developed in isolation. This means that a user can develop a useful addition to the available solver methods without having to understand the organizational policies of a larger open-source project. For example, DASSL.jl [3] is a re-write of the classic DASSL algorithm [13] into Julia which doesn't depend on any JuliaDiffEq resources other than DiffEqBase.jl to define the extension. This has been successful in recruiting earlier-stage academics, such as undergraduates in the Google Summer of Code program, to be method contributors since all that is required is the ability to understand the mathematical method and write Julia code. In addition, since changing the solver package choice only requires a change of input type, this allows for package-free methods development tooling, such as convergence testing and benchmarking, to be developed and systematically applied across the whole range of dispatch choices (detailed in a later section).

Lastly, a recent addition to the JuliaDiffEq organization are the diffeqr [16] and diffeqpy [17] packages for R and Python respectively that give multi-language access to the common API. Thus any developer who implements their differential equation solver method as a dispatch on the solve common automatically gets access to scientific users in the Jupyter (Julia, Python, and R) languages. This increases the possible scientific impact of each individual's work, allowing for accelerated scientific discovery as both users and methods developers do not need to worry about developing cross-language interoperability for their method to be impactful to the greater scientific community.

However, the confederation of solver packages has not just been advantageous to methods developers. Downstream scientific modelers and their domain-specific packages have benefitted since allowing the user to choose methods then allows for user choices of packages without any extra support required. An example implementation of a function which does parameter optimization (via an L^2 loss function of the ODE timeseries against a dataset) which works with any common interface ordinary differential equation, delay differential equation (DDE), differential-algebraic equation (DAE), or hybrid differential

```
function lorenz(du,u,p,t)
du[1] = 10.0*(u[2]-u[1])
du[2] = u[1]*(28.0-u[3]) - u[2]
du[3] = u[1]*u[2] - (8/3)*u[3]
end
u0 = [1.0;0.0;0.0]; tspan = (0.0,100.0)
prob = ODEProblem(lorenz,u0,tspan)
sol = solve(prob,Tsit5())
```

Algorithm 1. Lorenz Equation Example. Example user code solving Lorenz ordinary differential equation using DifferentialEquations.jl. The algorithm choice is given by the argument `Tsit5()` which causes `solve` to dispatch to the solver defined in the OrdinaryDiffEq.jl package.

```
struct Euler <: DEAlgorithm end
function DiffEqBase.solve(prob::ODEProblem,alg::Euler,args...;
             dt  =  error(''dt required for Euler''))
  tspan = prob.tspan; p = prob.p
  n = Int((tspan[2] - tspan[1])/dt) + 1
  u = [prob.u0 for i in 1:n]
  t = [tspan[1] + i*dt for i in 0:n-1]
  for i in 2:n
      uprev = u[i-1]
      tprev = t[i-1]
      u[i] = uprev + dt*prob.f(uprev,p,tprev)
  end
  build_solution(prob,alg,t,u)
end
```

Algorithm 2. Dispatch Example. An example code which defines a dispatch for the `solve` command and utilizes the Euler method to solve the given ODE problem.

```
function parameter_l2loss(prob,alg,t,data;kwargs...)
  cost_function = function (p) # Cost function to return
    tmp_prob = remake(prob,p=p) # Same problem, new parameters
    # Get solution at t's
    sol = solve(tmp_prob,alg;saveat=t,save_everystep=false,
                    dense=false,kwargs)
    sum(abs2,data - sol) # Get L2 loss
  end
end
```

Algorithm 3. Generic L^2 Loss Function Generation. This is an example of a package code for parameter optimization. The function `parameter_l2loss` returns a function that internally evaluates the ODE with given parameters and calculates the L^2 loss against a data set. The returned function can be given to a nonlinear optimization package to find the optimal parameters. This function is agnostic to the input problem and algorithm choice: `prob` is the problem to solve (`ODEProblem`, `DAEProblem`, `DDEProblem`, etc.) and `alg` can be an algorithm provided by any package which has a dispatch for `solve` on the given problem type.

equation (including event handling) is shown in Algorithm 3. We want to stress that this compatibility with a wide array of packages and differential equation types is not directly coded into our parameter estimation function. Rather, this functionality comes just by allowing the user to give the input problem type and the solver algorithm type and then all package (and sometimes language) interop is handled in the solve dispatches.

This extensibility by allowing type inputs has been utilized in downstream domain-modeling packages. For example, for solving the quantum dynamical equations such as the Schrodinger partial differential equations (PDEs) in QuantumOptics.jl [11], for systems with smaller numbers of states one may want to make use of the implicit BDF method available from the SUNDIALS C++ package [9] and wrapped via Sundials.jl, while to conserve memory for larger PDEs one may want to resort to the low-storage Runge-Kutta implementations in OrdinaryDiffEq.jl. QuantumOptics.jl allows the user to pass in any common interface algorithm type, and thus the user can change integrators (and thus the underlying solver package) by changing this input value,

allowing QuantumOptics.jl to be package-generic without having to directly support any of the possible choices. In addition, this allows domain-specific packages to add new methods as required to accelerate problems in their domain. For example, DynamicalSystems.jl [4] is a package downstream of DifferentialEquations.jl which utilizes the discrete evolution problems (functional iteration), but added their own algorithm type and dispatch which can more efficiently handle the event-free case. By implementing via extension, DynamicalSystems.jl and its users can still easily switch to any of the available DifferentialEquations.jl methods when required. The result is that developers and users have not had to worry about learning or wrapping many package APIs since the confederated API is sufficient for calling out to the many choices in the Julia package ecosystem.

4. Polyalgorithms for automatic algorithm specialization

The confederated formulation of the API has allowed for an explosion in the available solver methods. As of the writing of this
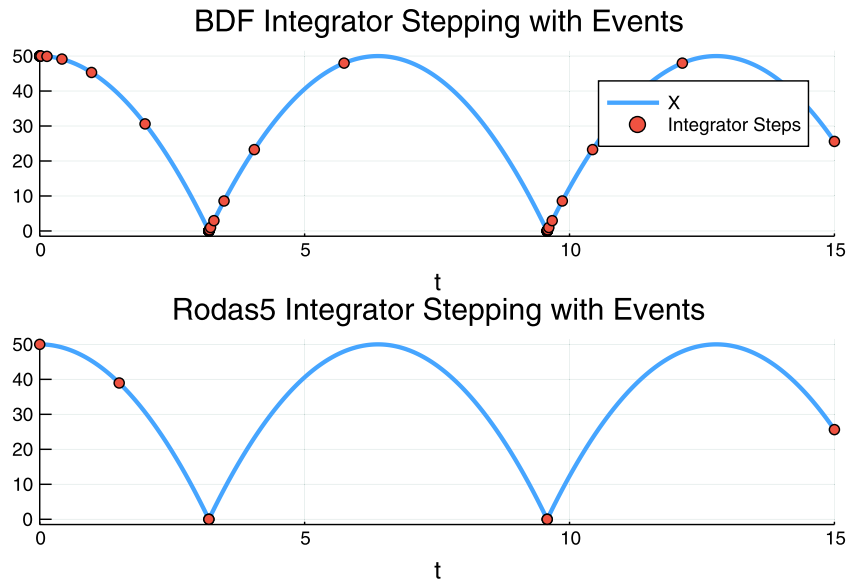
Fig. 2. BDF integrator time stepping with events. Shown is the solution of the bouncing ball hybrid ODE system $v' = -g$, $x' = v$ where g is the gravitational constant 9.8 and the sign of v flips whenever $x = 0$. In the case of the BDF integrator (SUNDIALS 3.1), events cause a reset of the order of the integrator to 1, causing 39 time steps to be taken (shown in orange). Meanwhile, the Rodas5 integrator (OrdinaryDiffEq.jl) solves the equation in 5 time steps, jumping from event to event and utilizing an interpolation to fill in the intermediate region.

manuscript, there exist over 200 methods simply for ODEs, and many more when including the other forms of differential equations offered by DifferentialEquations.jl (SDEs, DDEs, DAEs, jump equations like Gillespie SSA, etc.). Thus while ODE solver APIs like those offered by previous packages like MATLAB [20], SciPy [10], or R's deSolve [21] all have around 10 methods and document method choosing schemes for users to follow, we have noticed that our ecosystem's size is far too large to assume that any user will willingly learn all of the algorithm differences to make the most effective choice.

For example, for large stiff ODE systems a BDF method is recommended since the multistep method require fewer function evaluations than methods like ESDIRK. But too large of a system means that a user's Jacobian will not fit into memory, and thus if no sparse Jacobian form is given then the user will need to use a low-storage Runge–Kutta method (or the implicit methods will error due to running out of memory). Meanwhile, BDF integrators are not as efficient on event-heavy codes due to the fact that the order must reset to 1 when a derivative discontinuity is encountered, which results in a time step decrease and increased computational cost as shown in Fig. 2. Thus in this case one may want to use an ESDIRK method. If the user wants to solve the equation at low tolerances, the benchmarks frequently show that the high (and adaptive) order fully implicit Runge-Kutta integrator Radau [8] is the most efficient, while not being as efficient at higher tolerances (lower accuracy). Thus, while users have successfully been walked through these explanations in the JuliaDiffEq and Julia programming chatrooms, this attention to solver nuances is not something we have found most users are interested in.

Therefore we have developed polyalgorithms which auto-specialize on the problem and solver structures to allow for an automatic effective choice to be made if the user calls the solve function without an algorithm (i.e. `solve(prob)` or with options like `solve(prob,reltol=1e-8)`). Polyalgorithms have been applied in other numerical domains such as numerical linear algebra where MATLAB and Julia's backslash (\) check properties of the matrix before choosing a factorization method. However, polyalgorithms like this have been nonexistent in differential equation solver software sans a few automated stiffness detection algorithms which switch between two fixed choices during the time course of the integration [14]. Unlike these previous methods, our algorithm utilizes structural properties of the solver to choose integration methods, and we also utilize the user options and

allow for hints.

The algorithm starts by dispatching to a choice tree based on the input problem type, i.e. whether the problem is an ODE, SDE, DAE, DDE, etc. The dispatch is not only dependent on the type of differential equation but also its structure. DifferentialEquations.jl for example allows users to define ODEs in forms such as partitioned ODEs for second order differential equations and dynamical equations resulting from Hamiltonian equations, split ODEs ($u' = f_1(u, p, t) + f_2(u, p, t)$), and semilinear ODEs ($u' = Au + f(u, p, t)$). These special forms have an equivalent form as a first order ODE, but this special structure can be exploited by integrators to give more accurate results, such as symplectic or Runge-Kutta Nystrom integrators for partitioned ODEs, implict-explicit (IMEX) integrators for split ODEs, and exponential integrators for semilinear ODEs. When applicable, the polyalgorithm will pick an appropriate algorithm from these sub-classes specific to the problem's structure, or if no structure applies it will fall back to the first order ODE integrator methods.

When choosing a method from a class, features of the problem and solution are then used to make the final choice. As an example, the current first order ODE choice tree is shown in Fig. 3 and makes use of the size of the differential equation system, the user-chosen tolerance, algorithm hints about stiffness (along with automatic stiffness detection when not hint is given), and the presence of events. These choices are heuristics derived from the extensive benchmarks to give a choice which is likely to be efficient without having to rely on users having extensive knowledge of the solver details. We have found that most of the codes from package users utilize the automatic algorithm choices, whereas code from package developers and methods researchers (who generally take the time to benchmark methods on their specific problems) make use of direct algorithm choices. This shows that the algorithm has been successful in achieving its goal of abstracting away the numerical nature of differential equation problem solving while not impeding the flexibility required for developers.

5. Pervasive benchmarks for empirically-driven development

Given the wide range and available packages which are directly available via the common API, we developed a set of integrator analysis tools in DiffEqDevTools.jl. These tools can directly compute error estimates by using lower tolerance solutions (and in the case of stochastic
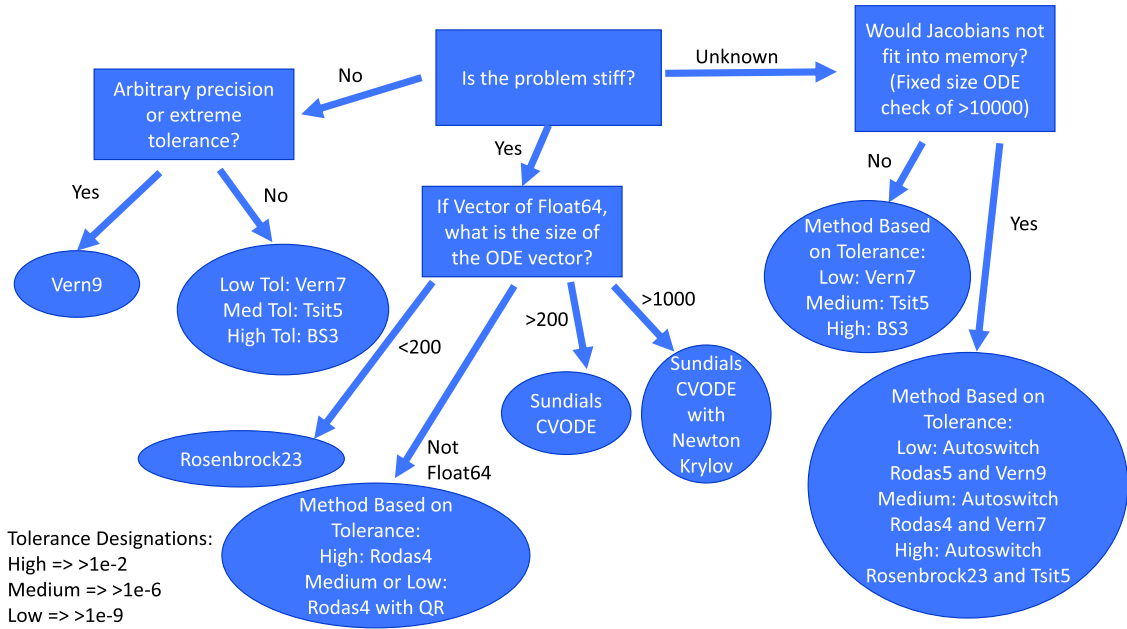
Fig. 3. First order ODE polyalgorithm. Depicted is the decision tree for the DifferentialEquation.jl polyalgorithm for choosing the integrator method based on the user's input tolerances and properties of the underlying ODE. Note that "Autoswitch" denotes an algorithm with stiffness detection that allows for automated switching between a method optimized for stiff equations and a method optimized for non-stiff equations.
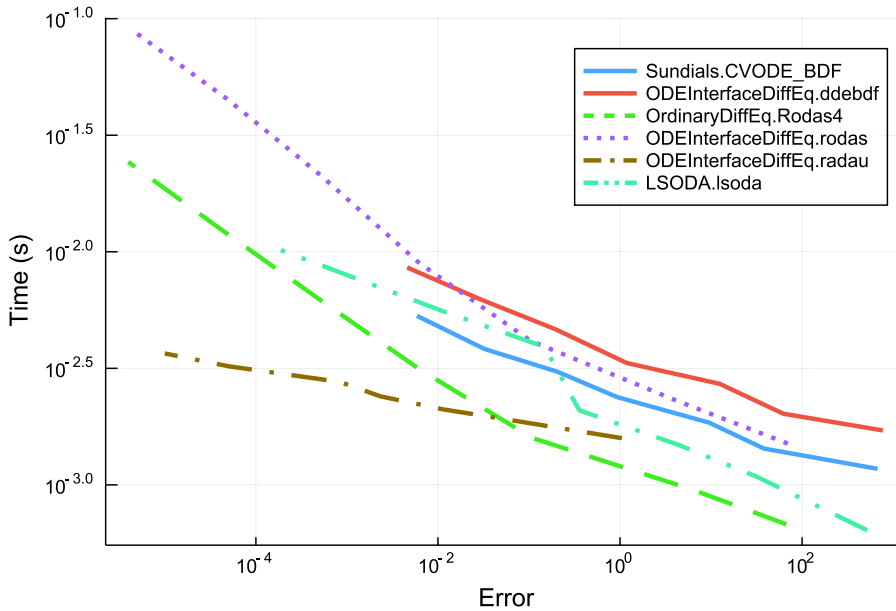

Fig. 4. Example work-precision diagram. This is a work-precision diagram of various common interface packages solving the OREGO system of 3 stiff ODEs [7]. Each algorithm is documented as `Package.algorithm`, for example `Sundials.CVODE_BDF` refers to the use of CVODE (with BDF coefficients) from the SUNDIALS C++ package [9]. The error is the absolute average time series error $\sum_i^n \|\hat{x}(t_i) - x(t_i)\|/n$ calculated against a reference solution solved with absolute and relative tolerances at 10^{-14}. The timings are derived from the benchmark computer which is a dual Xeon E5-2697A, but as an interactive notebook users and developers can re-run these benchmarks on their own hardware to generate comparable figures.

problems, utilize the same Brownian process to measure strong/weak error), allowing the use of realistic nonlinear models to be used for benchmarking. A set of benchmarks is shared by the JuliaDiffEq organization at the JuliaDiffEq/DiffEqBenchmarks.jl repository. An example work-precision diagram from the stiff ODE benchmark problem OREGO is shown in Fig. 4.

We have found that these benchmarks have been useful in guiding the development of the field. Since a large number of methods and packages can be pulled from (including classic C++ packages like SUNDIALS [9] and Fortran codes like Hairer's dop853 [6,7]), these benchmarks are able to effectively isolate differences between different implementations of the same method and performance enhancements which are due the integration method itself. While these benchmarks have been useful to the JuliaDiffEq developers for identifying performance issues and regressions, these benchmarks have also had a large

effect on our development goals. Observations around the efficiencies of Rosenbrock and ESDIRK methods for small stiff ODEs and the importance of coefficient optimization in high order explicit ODE Runge-Kutta integrators spawned the motivation to optimize stochastic differential equation integrators of a similar form [19]. Results of benchmarks on LSODA's efficiency vs that of pure BDF methods was the impetus for a Google Summer of Code project on developing the automatic stiffness detection and switching in OrdinaryDiffEq.jl. By having open benchmarks, newcomers to Julia have been able to directly see how the native Julia implementations fare against those of the more developed statically compiled languages (C++ and Fortran), which has spurred other Google Summer of Code students to focus on areas where JuliaDiffEq was previously lacking such as adaptive multistep integrators. Additionally, these benchmarks have been helpful in allowing researchers to assess the necessity of switching their current

tooling by seeing timings on real applications. For example, these benchmarks routinely show that integrators within the same class (for example, stiff ODEs looking at BDF vs (E)SDIRK vs Rosenbrock) generally do not differ in work-precision timing by more than an order of magnitude with the largest differences being between families of methods rather than within. Additionally, we see large differences in performance depending on the language which is used, for example MATLAB ODE's and R's deSolve suite taking routinely 100x longer, while the SciPy integrators with Numba-accelerated derivative functions taking routinely 10x longer than the optimized Julia and C++ codes. We hope this verification and quantification of the Julia performance claims in real-world scenarios can better help other researchers choose the right path for their project and team.

From these benchmarks there are many interesting properties about the families of differential equation solver methods and their performance on different types of problems which can be elaborated. Our team wishes to summarize these results in a publication in the near future.

6. Discussion and conclusion

The confederated modular API of the DifferentialEquations.jl package has had a large impact on the ability for possible methods developers to contribute to the ecosystem. This in turn has led to the availability of hundreds of methods from this interface which are directly available to users of downstream packages and users of the Python and R wrappers. This catalogue has been and is continuously being benchmarked on real-world problems in order to guide future JuliaDiffEq development projects and build an empirically-based polyalgorithm for automatic method selection.

We see the possibility for these architectures to be employed in other scientific disciplines for similar effects. While machine learning packages like SciKitLearn [12] allow for calling a large set of methods on the same dataset, a confederated version built around dispatch could have similar effects on the machine learning community as we have seen in JuliaDiffEq.

Acknowledgments

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.advengsoft.2019.03.009

References

[1] Bezanson J, Chen J, Chung B, Karpinski S, Shah VB, Vitek J, Zoubritzky L. Julia: dynamism and performance reconciled by design. PACMPL 2018;2(OOPSLA). https://doi.org/10.1145/3276490. 120:1-120:23.
[2] Bezanson J, Edelman A, Karpinski S, Shah V. Julia: a fresh approach to numerical computing. SIAM Rev 2017;59:65–98. https://doi.org/10.1137/141000671.
[3] Biernat P, Rackauckas C. DASSL.jl. 2016.
[4] Datseris G. DynamicalSystems.jl: a julia software library for chaos and nonlinear dynamics. J Open Source Software 2018;3(23):598. https://doi.org/10.21105/joss.00598.
[5] Forst S. Liblsoda. 2016.
[6] Hairer E, Norsett SP, Wanner G. Solving ordinary differential equations I: nonstiff problems. 2nd rev. ed Springer series in computational mathematics Heidelberg; London: Springer; 2009. OCLC: ocn620251790
[7] Hairer E, Wanner G. Solving ordinary differential equations II - stiff and differential-algebraic problems. Springer; 1991.
[8] Hairer E, Wanner G. Stiff differential equations solved by Radau methods. J Comput Appl Math 1999;111:93–111. https://doi.org/10.1016/S0377-0427(99)00134-X.
[9] Hindmarsh AC, Brown PN, Grant KE, Lee SL, Serban R, Shumaker DE, Woodward CS. SUNDIALS: suite of nonlinear and differential/algebraic equation solvers. ACM Trans Math Softw 2005;31:363–96. https://doi.org/10.1145/1089014.1089020.
[10] Jones E, Oliphant T, Peterson P, others. Scipy: open source scientific tools for Python. 2001.
[11] Kramer S, Plankensteiner D, Ostermann L, Ritsch H. QuantumOptics.jl: a Julia framework for simulating open quantum systems. Comput Phys Commun 2018;227:109–16. https://doi.org/10.1016/j.cpc.2018.02.004.
[12] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D. Scikit-learn: machine learning in python. Mach Learn Python 2011:6.
[13] L. Petzold, Description of dassl: a differential/algebraic system solver (1982).
[14] L. Petzold, Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations, 1983, 10.1137/0904010.
[15] Rackauckas C. A comparison between differential equation solver suites in matlab, r, julia, python, c, mathematica, maple, and fortran. Winnower 2018. https://doi.org/10.15200/winn.153459.98975.
[16] Rackauckas C. Diffeqr. 2018.
[17] Rackauckas C, Arakaki T. Diffeqpy. 2018.
[18] Rackauckas C, Nie Q. DifferentialEquations.jl - a performant and feature-rich ecosystem for solving differential equations in julia. J Open Res Softw 2017;5:15. https://doi.org/10.5334/jors.151.
[19] C. Rackauckas, Q. Nie, Stability-optimized high order methods and stiffness detection for pathwise stiff stochastic differential equations, (2018). arXiv:1804.04344.
[20] Shampine LF, Reichelt MW. The matlab ode suite. SIAM J Sci Comput 1997;18(1):1–22. https://doi.org/10.1137/S1064827594276424.
[21] Soetaert K, Petzoldt T, Setzer RW. Solving differential equations in r: Package desolve. J Stat Softw 2010;33(9):1–25. https://doi.org/10.18637/jss.v033.i09.
[22] Tsitouras C. Runge–Kutta pairs of order 5(4) satisfying only the first column simplifying assumption. Comput Math Appl 2011;62(2):770–5. https://doi.org/10.1016/j.camwa.2011.06.002.
[23] Veltz R, Rackauckas C, Frost S. LSODA.jl. 2017.