Journal of
open research software

Rackauckas, C and Nie, Q 2017 DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. *Journal of Open Research Software,* 5: 15, DOI: https://doi.org/10.5334/jors.151

SOFTWARE METAPAPER

DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia

Christopher Rackauckas and Qing Nie

Department of Mathematics, University of California-Irvine, Irvine, CA, 92697, US

Corresponding author: Christopher Rackauckas (accounts@chrisrackauckas.com)

DifferentialEquations.jl is a package for solving differential equations in Julia. It covers discrete equations (function maps, discrete stochastic (Gillespie/Markov) simulations), ordinary differential equations, stochastic differential equations, algebraic differential equations, delay differential equations, hybrid differential equations, jump diffusions, and (stochastic) partial differential equations. Through extensive use of multiple dispatch, metaprogramming, plot recipes, foreign function interfaces (FFI), and call-overloading, DifferentialEquations.jl offers a unified user interface to solve and analyze various forms of differential equations while not sacrificing features or performance. Many modern features are integrated into the solvers, such as allowing arbitrary user-defined number systems for high-precision and arithmetic with physical units, built-in multithreading and parallelism, and symbolic calculation of Jacobians. Integrated into the package is an algorithm testing and benchmarking suite to both ensure accuracy and serve as an easy way for researchers to develop and distribute their own methods. Together, these features build a highly extendable suite which is feature-rich and highly performant.

Keywords: Julia; ordinary differential equations; stochastic differential equations; partial differential equations; multiple dispatch; metaprogramming; high-precision; multithreading

Funding statement: This work was partially supported by NIH grants P50GM76516 and R01GM107264 and NSF grants DMS1562176 and DMS1161621. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1321846, the National Academies of Science, Engineering, and Medicine via the Ford Foundation, and the National Institutes of Health Award T32 EB009418. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the NIH.

(1) Overview

1 Introduction

Differential equations are fundamental components of many scientific models; they are used to describe large-scale physical phenomena like planetary systems [10] and the Earth's climate [12, 18], all the way to smaller scale biological phenomena like biochemical reactions [30] and developmental processes [27, 7]. Because of the ubiquity of these equations, standard sets of solvers have been developed, including Shampine's ODE suite for MATLAB [25], Hairer's Fortran codes [8], and the Sundials CVODE solvers [11].

However, these software packages contain many limitations which stem from their implementation and the time when they were developed. Since the time of their inception, many other forms of differential equations have become commonplace tools not only for mathematicians, but throughout the sciences. Stochastic differential equations (SDEs), have become more commonplace not only in mathematical finance [23, 5], but also in biochemical [4, 13] and ecological models. Delay differential equations have become a ubiquitous tool for modeling phenomena with natural delays as seen in Neuroscience [3, 22] and control theory [24]. However, a user who is familiar with standard ODE tools has to "leave the box" to find a new specialized package to handle these kinds of differential equations, or write their own solver scripts [9]. Also, when many of these methods were implemented the standard computer was limited by the speed of the processor. These days, most processors are multi-core and many computers contain GPGPU [1] or Xeon Phi [17, 6] acceleration cards and thus taking advantage of the ever-present parallelism is key to achieving good performance.

Other design limitations stem from the programming languages used in the implementation. Many of these algorithms, being developed in early C/Fortran, do not have abstractions for generalized array formats. In order to use these algorithms, one must provide the solver with a vector. In cases where a matrix or a higher dimensional tensor are the natural representation of the

differential equation, the user is required to transform their equation into a vector equation for use in these solvers. Also, these solvers are limited to using 64-bit floating point calculations. The numerical precision limits their use in high-precision applications, requiring specialized codes when precision lower than 10^{-16} is required. Lastly, many times these programs are interfaced via a scripting language where looping is not optimized and where "vectorized" codes provide the most efficient solution. However, vectorized coding in the style of MATLAB or NumPy results in temporary allocations and can lack compiler optimizations which require type inference. This increases the computational burden of the user-defined functions which degrades the efficiency of the solver.

The goal of DifferentialEquations.jl is build off of the foundation created by these previous differential equation libraries and modernize them using Julia. Julia is a scripting language, used in-place of languages like R, Python, MATLAB, but offers the performance one would associate with low-level compiled languages. This allows users to start prototypes in Julia, but also solve their large-scale models within the same language, instead of resorting to two language solutions when performance is needed. The language achieves this goal by extensive utilization of multiple dispatch and metaprogramming to design a language that is both easy for a compiler to understand and easy for a programmer to use [2]. DifferentialEquations.jl builds off of these design principles to arrive at a fast, feature-rich, and highly extendable differential equations suite which is easy to use.

We start by describing the innovations in usability. In Section 1.1 we show how multiple dispatch is used to consolidate the functions the user needs to into simple descriptive commands like `solve` and `plot`. Since these commands are used for all forms of differential equations, the user interface is unified in a manner that makes it easy for a user to explore other types of models. Then in Section 1.2 we show how metaprogramming is used to further simplify the user API, allowing the user to define a function in a "mathematical format" which is automatically converted into the computationally-efficient encoding. After that, we describe how the internals were designed in order to be both feature-filled and highly performant. In Section 1.3 we describe the package structure of DifferentialEquations.jl and how the Base libraries, component solvers, and add-on packages come together to provide the full functionality of DifferentialEquations.jl. In Section 1.4 we describe how multiple dispatch is used to write a single generic method which compiles into specialized functions dependent on the number types given to the solver. We show how this allows for the solvers to both achieve high performance while being compatible with any Julia-defined number system which implements a few basic mathematical operations, including fast high and intermediate precision numbers and arithmetic with physical units. In Section 1.5 we describe the experimental within-method multi-threading which is being used to further enhance the performance of the methods, and the multi-node parallelism which is included for performing Monte Carlo simulations of stochastic models. We then discuss some of the tools which allows DifferentialEquations.jl to be a good test suite for the fast development and deployment of new solver algorithms, and the tools provided for performing benchmarks. Lastly, we describe the current limitations and future development plans.

1.1 A Unified API Through Multiple Dispatch

DifferentialEquations.jl uses multiple dispatch on specialized types to arrive at a unified user-API for the different types of equations. To use the package, one follows the steps:

1. Define a problem.
2. Solve the problem.
3. Plot the solution.

This standardization of the API makes complicated solvers accessible to less programming-inclined individuals, giving a good framework for future development and allows for the latest research in numerical differential equations to be utilized without complications.

1.1.1 Solving ODEs

To define a problem, a user must call the constructor for the appropriate problem object. Since ordinary differential equations (ODEs) are represented in the general form as

$$\frac{du}{dt} = f(t,u), \quad u(0) = u_0, \tag{1}$$

the `ODEProblem` is defined by specifying a function f and an initial condition u_0. For example, we can define the linear ODE using the commands:

```
using DifferentialEquations
f(t,y) = 0.5y
u0 = 1.5
timespan = (0.0,1.0) # Solve from time = 0 to
time = 1
prob = ODEProblem(f,u0,timespan)
```

Many other examples are provided in the documentation[1] and the Jupyter notebook tutorials in DiffEqTutorials.jl[2] (for use with Julia, see IJulia.jl[3]). To solve the ODE, the user can simply call the solve command on the problem:

```
sol = solve(prob) # Solves the ODE
```

By using a dispatch architecture on `AbstractArrays` and using the array-defined indexing functionality provided by Julia (such as `eachindex(A)`), DifferentialEquations.jl accepts problems defined on arrays of any size. For example, one can define and solve a system of equations where the dependent variable u is a matrix as follows:

```
A = [1. 0 0 -5
     4 -2 4 -3
    -4  0 0  1
     5 -2 2  3]
u0 = rand (4,2)
f(t,u) = A*u
prob = ODEProblem(f,u0,timespan)
sol = solve(prob)
```

For most other packages, one would normally have to define u as a vector and rewrite the system of equations in the vector form. However, by allowing arbitrary problem

sizes, DifferentialEquations.jl allows the user to specify problems in the natural format and solve directly on any array of numbers. This can be helpful for problems like discretizations of partial differential equations (PDEs) where the matrix format matches some underlying structure, and could result in a denser formulation.

The solver returns a solution object which holds all of the information about the solution. Dispatches to array functions are provided on the `sol` object, allowing for the solution object act like a timeseries array. In addition, high-order efficient interpolations are lazily constructed throughout the solution (by default, a feature which can be turned off) and the `sol` object's call is overloaded with the interpolating function. Thus the solution object can both be used as an array of the solution values, and as a continuous approximation given by the numerical solution. The syntax is as follows:

```
sol[i] # ith solution value
sol.t[i] # ith timepoint
sol(t) # Interpolated solution at time t
```

The solution can be plotted using the provided plot recipes through Plots.jl[4]. The plot recipes use the solver object to build a default plot which is customizable using any of the commands from the Plots.jl package, and can be plotted to any plotting backend provided by Plots.jl. For example, we can by default plot to the PyPlot.jl[5] backend (a Julia wrapper for matplotlib[6]) via the command:

```
plot(sol)
```

These defaults are deliberately made so that a standard user does not need to dig further into the manual and understand the differences between all of the algorithms. However, an extensive set of functionality is available if the user wishes. All of these functions can be modified via additional arguments. For example, to change the solver algorithm to a highly efficient Order 7 method due to Verner [29], set the line width in the plot to 3 pixels, and add some labels to the plot, one could instead use the commands:

```
sol = solve(prob, Vern7()) # Unrolled Verner 7th
Order Method
plot(sol,linewidth=3,xlabel="t",ylabel="u(t)")
```

The output of this command is shown in **Figure 1**. Note that the output is automatically smoothed using `10*length(sol)` equally spaced interpolated values through the timespan.

Lastly, these solvers tie into Julia integrated development environments (IDEs) to further enhance the ease of use. Users of the Juno IDE [16] are equipped with a progressbar and time estimates to monitor the progress of the solver. Additionally, all of the DifferentialEquations.jl functions are thoroughly tested and documented with the Jupyter notebook system [19], allowing for reproducible exploration.

1.1.2 Solving SDEs

By using multiple-dispatch, the same user API is offered for other types of equations. For example, if one wishes to solve a stochastic differential equation (SDE):

$$dX_t = f(t,X_t)dt + g(t,X_t)dW_t, \tag{2}$$

then one builds an SDEProblem object by specifying the initial condition and now the two functions, f and g. However, the rest of the usage is the same: simply use the solve and plot functions. To extend the previous example to have multiplicative noise, the code would be:

```
g(t,u) = 0.3u
prob = SDEProblem(f,g,u0,timespan)
sol = solve(prob)
plot(sol)
```

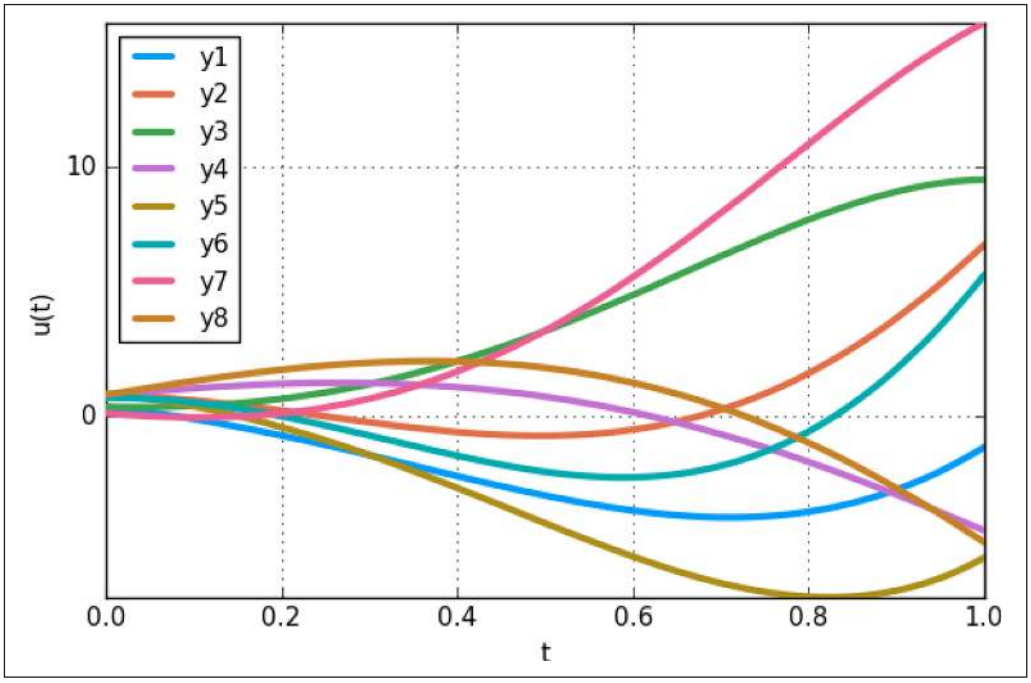

Figure 1: Example of the ODE plot recipe. This plot was created using the PyPlot backend through Plots.jl. Shown is the solution to the 4 × 2 ODE with $f(t,u) = Au$ where A is given in the code. Each line corresponds to one component of the matrix over time.

While this user interface is simple, the default methods these algorithms can call are efficient high-order solvers with adaptive timestepping [21]. These methods tie into the plotting functionality and IDEs in the same manner as the ODE solvers, making it easy for users to explore stochastic modeling without having to learn learn a new interface.

1.1.3 Solving (Stochastic) PDEs

Again, the same user API is offered for the available stochastic PDE solvers. Instead, one builds a `HeatProblem` object which dispatches to algorithms for solving (Stochastic) PDEs. An example using the previously defined functions is:

```
T = 5
dx = 1/2^(1)
dt = 1/2^(7)
fem_mesh = parabolic_squaremesh([0 1 0 1],dx,
dt,T,:neumann)
prob = HeatProblem(f,mesh,σ=σ)
sol = solve(prob)
```

Additional keyword arguments can be supplied to `HeatProblem` to specify boundary data and initial condtions. Notice that the main difference is now we must specify a space-time mesh (and boundary conditions as optional keyword arguments). Again, the same plotting and analysis commands apply to the solution object `sol` (where now the `plot` dispatch is to a `trisurf` plot).

1.2 Enhanced Performance and Readability Through Macros

1.2.1 A Macro-Based Interface

Most differential equations packages require that the user understands some details about the implementation of the library. However, the DifferentialEquations.jl ecosystem implements various Domain-Specific Languages (DSLs) via macros in order to give more natural options for defining mathematical constructs. In this section we will demonstrate the DSL for defining ODEs. For demonstrations related to other types of equations, please see the documentation.

The famous Lorenz system is mathematically defined as

$$\frac{dx}{dt} = \sigma(y - x) \tag{3}$$

$$\frac{dy}{dt} = x(\rho - z) - y \tag{4}$$

$$\frac{dz}{dt} = xy - \beta z \tag{5}$$

A user must re-write this function in a "computer friendly format", defining `u=[x;y;z]` as a vector and writing the equation in terms of this vector. The format for ODE.jl, which is similar to other scripting languages like SciPy or MATLAB, is as follows:

```
f = (t,u,du) -> begin
du[1] = 10*(u[2]-u[1])
du[2] = u[1]*(28-u[3]) - u[2]
du[3] = u[1]*u[2] - 8/3*u[3]
end
```

While this format is accepted by DifferentialEquations.jl, additional usability macros are provided which will automatically translate user input from a more mathematical format. For ODEs, `@ode_def` is provided which allows the user to define the same ODE as follows:

```
f = @ode_def Lorenz begin
dx = σ*(y-x)
dy = x*(ρ-z) - y
dz = x*y - β*z
end σ=>10. ρ=>28. β=(8/3)
```

Since Julia allows for the use of Unicode within code, this format matches the style one would expect to see in a TeX'd publication. The macro takes in this definition, finds the values for the left-hand side of the form "d__", and uses a dictionary in order to find/replace these values to write a function which is in the format of the other scripting language libraries. Thus the translation to a vector system can be done by DifferentialEquations.jl, allowing the users to have more readable scripts while not sacrificing performance. In addition, the macro produces a function which updates an input `du` in-place as the output. This detail can be hard for non-programmers to understand but is required for achieving fast solutions since otherwise every function call requires an array allocation.

1.2.2 Explicit Parameters

A unique feature from this form of function definition is that the parameters are built into the function type itself. The actual implementation involves creating a type `Lorenz` with fields for the parameters (inlining parameters defined with = instead of => during compilation). Then the type is set to have its call overloaded by the standard `f(t,u,du)` function signature, effectively acting like the appropriate function. However, the parameters are still accessible via the type fields, for example `f.a` or the overloaded `f[:a]`. This allows for sensitivity analysis, bifurcation diagrams, and parameter estimations to be computed using the same function, allowing for this infrastructure to extend far beyond the domain of differential equations solvers.

1.2.3 Enhanced Performance Through Symbolic Calculations

Also, since the code is analyzed by the program at the expression level, silent optimizations are able to be performed. For example, during the construction of the function, the code is transformed into a symbolic form for use in the high-performance CAS SymEngine.jl[7] [28], where the Jacobian is calculated and an in-place function for its computation is created. In addition, the symbolic expression is inverted, allowing for stiff solvers which require inverting Jacobians to be written as directly computed matrices and matrix multiplications.

1.3 The Distributed Structure of DifferentialEquations.jl

The full functionality of DifferentialEquations.jl is defined in the more than 40 packages in the JuliaDiffEq Github organization[8]. It splits the main packages into three parts: the Base libraries, the component solvers, and the add-on packages. DifferentialEquations.jl is a metapackage which utilizes all of these packages

together and adds default behavior to give a cohesive ecosystem.

1.3.1 Building the solve function from DiffEqBase.jl and Component Solvers

DifferentialEquations.jl is designed to use multiple-dispatch in order to allow for the different solver methods to be defined in separate packages. In Julia, the command:

```
solve(prob,Algorithm())
```

calls a different "method" depending on the type of `Algorithm`. This function is called the "common solve interface". The actual method that it calls does not need to be in the same package where the original function is defined. Using this design, the central package to the DifferentialEquations.jl ecosystem is DiffEqBase.jl[9]. DiffEqBase.jl defines the abstract type hierarchy, along with the problem and solution types, and the shared components. All of the component solver packages have a dependency on DiffEqBase.jl and add a method to this solve function. For example, for ODEs, the current list of component solvers is:

- OrdinaryDiffEq.jl[10]
- Sundials.jl[11]
- ODE.jl[12]
- ODEInterface.jl[13]
- LSODA.jl[14]

Some of the packages, like OrdinaryDiffEq.jl and ODE.jl, are native Julia libraries, whereas Sundials.jl, ODEInterface.jl, and LSODA.jl are all interfaces to popular C and Fortran codes. Through this interface, users can switch between libraries by switching only the algorithm choice, and new packages can extend what's available as needed. Note that the information in this structure is unidirectional: anyone can add a new package of solvers to the `solve` function by adding a dependency on DiffEqBase.jl without DiffEqBase.jl needing to be changed. This means that private projects, such as those which contain private research or proprietary methods, can extend this interface without being forced to edit the public DiffEqBase.jl repository. This same setup is then applied to each of the other types of equations. For full details on the current list of solvers methods that are available, along with the available methods for the other types of equations, please see the documentation. Note that from this modular structure, developers of other packages can use the functionality of DifferentialEquations.jl without having to depend on the entirety of the differential equations suite. For example, one can build an add-on package which uses the native ODE solvers and only include DiffEqBase.jl and OrdinaryDiffEq.jl as dependencies. This reduces the complexity associated with using the functionality, and helps developers keep dependencies as lean as possible.

1.3.2 The Add-on Packages

The add-on packages are a set of functionality which use the common solve interface. For example, parameter estimation functionality is provided by defining algorithms which only use the abstraction of the `solve` function, and allows the user to pass in the algorithm. Therefore these algorithms are generic, supporting many different equation types and internally able to use many different solver packages. Other such add-on functionality includes parameter sensitivity analysis, Monte Carlo parallelism functions, and uncertainty quantification. For more information on the current add-on packages, please consult the documentation.

1.4 Multiple Dispatch as a tool for Arbitrary Numerics

Julia's base library defines its standard numeric types, `Float64`, `Int64`, etc., as concrete subtypes of the abstract type `Number`. The implementation is contained within Julia: using a concrete `primative type` as a way to store numbers, and defining the operations such as +,–, etc. for each pair of numbers using dispatch on subtypes of `Number`. The result is that each number type receives its own compiled function for each operation, resulting in performance which can be 1x with C (as can be investigated via the `@code_llvm` and `@code_native` macros). This design allows for users to develop pure Julia packages which implement new number systems. DifferentialEquations.jl utilizes Julia's multiple dispatch architecture to allow for fast performance over these arbitrary numerical types. The design of the integration schemes includes a wrapper over the integration loops which matches types (to ensure type-stability), choosing the types for the problem by the user defined u_0 (the initial condition) and timespan (i.e. separate types are allowed for the dependent and independent variables). It then calls a type-dependent integration function which is optimized via JIT compilation for the numeric types given to the function. Different dispatches are given for subtypes of Number and AbstractArray since arrays are mutable and heap allocated, meaning that when numbers are treated directly instead of as arrays of size 1 a large speedup can occur. This allows for the internal integration algorithms to achieve C/Fortran speeds, while allowing for the generic numerical types and the readability of being in Julia itself. The following subsections highlight two important examples.

1.4.1 Case 1: Arbitrary Precision Numerics

One advantage of this design beyond speed is that it allows the user of DifferentialEquations.jl to use any type which is a subclass of Number as the number type for the equations. This includes not only the basic types like `Float64` and `Int64`, but also Rational and arbitrary precision `BigFloats` (based off of GNU MPFR). However, even numeric types defined in packages (which implement +,–,/, and for optionally for adaptive timestepping, `sqrt`) can be used within DifferentialEquations.jl. Some examples which have been shown to work are ArbFloats.jl[15] (a library for faster high-precision numbers than MPFR floats between 64 and 512 bits based on the Arb library of Fredrik Johansson [14]) and DecFP.jl[16] (an implementation of IEEE 754–2008 Decimal Floating-Point Arithmetic).

The combination of high-performance number systems with high order Runge-Kutta methods such as the Order

14 methods due to Feagin allows for fast solving with high accuracy. For an example showing this combination, see the "Feagin's Order 10, 12, and 14 methods" notebook in the examples folder[17].

1.4.2 Case 2: Unitful Numbers

This design also allows DifferentialEquations.jl to be compatible with number systems which have physical units. SIUnits.jl[18] and Unitful.jl[19] are packages which have developed number implementations which have units. Numbers defined by these packages automatically constrain the equations to satisfy dimensional constraints. For example, if one tries to add a quantity with units of seconds with a quantity with units of Newtons, it will throw an error. This is useful in fields like physics where these dimensional analysis tools are used to check for correctness in equations. DifferentialEquations.jl was developed such that the internal solvers satisfy dimensional constraints. Thus one can use unitful numbers like other arbitrary number systems. The "Unit Checked Arithmetic via Unitful" notebook in the examples folder[20] describes the usage of this feature. For example, we can solve an ODE where the dependent variable is in terms of seconds and the independent variable is in terms of Newtons via the following equation:

```
using DifferentialEquations, Unitful
f = (t,y) -> 0.5*y
u = 1.5u"N"
prob = ODEProblem(f,u,(0.0u"s",1.0u"s"))
sol = solve(prob,dt=(1/2^4)u"s")
```

The attentive reader should realize that this will correctly throw an error: the output of the function in an ODE must be a rate, and therefore must have units of N/s in this example. Unitful.jl will thus return an error notifying the user that the dimensions are off by a unit of seconds. Instead, the pleased physicists would modify the previous code by a rate constant and use the following code instead:

```
f = (t,y) -> 0.5*y/3.0u"s"
u = 1.5u"N"
prob = ODEProblem(f,u,(0.0u"s",1.0u"s"))
sol = solve(prob,dt=(1/2^4)u"s")
```

This will produce an output whose units are in terms of Newtons, and with time in terms of seconds.

1.5 Integrated Parallelism

1.5.1 Within-Method Multithreading

DifferentialEquations.jl also includes parallelism whenever possible. One area where parallelism is currently being employed is via "within-method" parallelism for Runge-Kutta methods. Using Julia's experimental multithreading, DifferentialEquations.jl provides a multithreaded version of the `DP5` solver. Benchmarks using the tools from Section 1.6 show that this can give a 30% non-multithreaded algorithm for problem sizes ranging from 75×75 matrices to 200×200 matrices. For larger problems this trails off as more time is spent within the function evaluations, thus reducing the difference between the methods. See the "Multithreaded Runge-Kutta Methods" notebook[21] in DiffEqBenchmarks.jl for the most up-to-date results as this may change rapidly along with Julia's threading implementation.

1.5.2 Multi-Node Monte Carlo Simulations

Also, DifferentialEquations.jl provides methods for performing parallel Monte Carlo simulations. Using Julia's pmap construct, one is able to specify for a problem to be solved N times, and DifferentialEquations.jl will distribute this automatically across multiple nodes of a cluster. A vector of results along with summary statistics is returned for the solution. This functionality has been tested on the local UC Irvine cluster (using SGE) and the XSEDE Comet cluster (using Slurm).

1.6 Development, Testing, and Benchmarking

DifferentialEquations.jl includes a suite specifically designed for researchers interested in developing new methods for differential equations (like the authors themselves). This includes functionality for easy integration of new methods, extensive testing, and a benchmarking suite.

1.6.1 Development

The design of DifferentialEquations.jl allows for users to add new integration methods by adding new dispatches. One way to add new methods is to simply create a new package which extends the `solve` function of DiffEqBase.jl as described in 1.3. Another way is to extend the current open-source native Julia solvers on the common interface. Let's take as an example the ODE solver suite OrdinaryDiffEq.jl[22], which contains the native Julia methods developed in tandem with DifferentialEquations.jl. The ODE solver works by setting up options and fixing types, and then builds the `integrator` type and enters the internal loop. All dispatchs on loop functions are done by the algorithm/cache type. Thus to define a new algorithm, one defines a new algorithm type which subtypes the `OrdinaryDiffEqAlgorithm` type. This will make `solve` plug into OrdinaryDiffEq.jl's version of `solve`. From there, a new dispatch for `perform_step` needs to be added which performs the algorithm's update from u_n to u_{n+1}. For example, the Midpoint Method steps as follows:

```
@inline function perform_step!(integrator,
                  cache::MidpointConstantCache,
                  f=integrator.f)
   @unpack t,dt,uprev,u,k = integrator
   halfdt = dt/2
   k = integrator.fsalfirst
   k = f(t+halfdt,uprev+halfdt*k)
   u = uprev + dt*k
   integrator.fsallast = f(t+dt,u) # For
             interpolation, then FSAL'd
   integrator.k[1] = integrator.fsalfirst
   integrator.k[2] = integrator.fsallast
   @pack integrator = t,dt,u
end
```

Additionally a new `cache` type must be developed for holding the cache variables. By doing this, all of the functionality of OrdinaryDiffEq.jl will be automatically applied to the algorithm. Thus interpolated (dense) output, FSAL optimizations (first-same-as-last, skipping an extra function evaluation), progress monitoring, event handling, and integrator interfaces will be available. Additionally, if an error estimator is given, the PI-contolled adaptive timestepping will be enabled for the algorithm. For more details, see the Developer Documentation[23].

1.6.2 Testing

The DifferentialEquations.jl suite includes a large number of testing functions to ensure correctness of all of the algorithms. Many premade problems with analytical solutions are provided and convergence testing functionality is included to be able to test the order of accuracy and plot the results. All of the DifferentialEquations.jl algorithms are tested using the Travis and AppVoyer Continuous Integration (CI) testing services to ensure correctness.

1.6.3 Benchmarking

Lastly, a benchmarking suite is included to test the efficiency of different algorithms. Two forms of benchmarking are included: the `Shootout` and the `WorkPrecision`. A `Shootout` solves using all of the algorithms in a given setup and calculates an average time (over a user-chosen number of runs) and error for each algorithm. The `WorkPrecision` and `WorkPrecisionSet` additionally take in vectors of tolerances and draw work-precision diagrams to compare algorithms. Up to date benchmarks can be found in the repository's benchmarks folder[24]. These notebooks can be opened to be run locally via the commands

```
using IJulia
notebook(dir=Pkg.dir("DiffEqBenchmarks")*"/
benchmarks")
```

As of this publication, the benchmarks show that native methods from OrdinaryDiffEq.jl (which were developed in tandem with DifferentialEquations.jl) achieve an order of magnitude speedup on nonstiff problems when achieving the same error over the classic Hairer Runge-Kutta implementations and the ODE.jl implementations.

1.7 Limitations and Future Development Plans

While DifferentialEquations.jl already offers many new features and high performance, the package is still under heavy development and will be for the foreseeable future. Currently, most of the methods for stiff equations are wrapped methods (only the Rosenbrock with/without local extrapolation, and the order 1/2 BDF methods exist in native forms). While these methods, such as CVODE (provided by Sundials.jl[25]) and radau (provided by ODEInterface.jl[26]), are widely regarded standards for stiff ODEs, by not being native Julia functions these algorithms choices do not allow for the extra functionality such as arbitrary precision and arithmetic with physical units (these features require a pure-Julia implementation). Instead, since these are the standard algorithms wrapped in packages such as SciPy [15] and R's deSolve [26], the limitations of these wrapped solvers match the limitations of other common libraries.

The native Julia methods have far less limitations because they work on the general abstract types `AbstractArray` and `Number`. For example, while other packages are limited to non-distributed arrays and thus must be able to fit the problem in the memory of one computer or node, any user can define the input equation using a `DistrbutedArray` from DistributedArrays.jl[27], Julia will automatically compile a new dispatch for the solver commands to make use of the distributed structure. In addition, the native Julia solvers of OrdinaryDiffEq.jl allow one to swap out the linear and nonlinear solver methods, allowing the users to parallel methods from PETSc.jl[28] and and GPU methods from packages like CUSOLVER.jl[29].

However, while the genericness of the implementation makes it very flexible, one limitation of the design is that the full extent of the compatibility is not able to be easily documented or known. The practice of "duck typing" means that the generic functions are left open ended, and functionality will work on available types depending on whether certain traits or operations are defined. For example, Julia-defined numbers systems are compatible with the if certain operations (+,–,*,/) are defined. However, different solver algorithms can have slightly different compatibility requirements. Adaptive timestepping also requires that the number system has a well-defined `sqrt` function. Thus `Rational` numbers are compatible with explicit methods, but not when adaptive timestepping is enabled. Some of the stiff solvers require the ability to be used in autodifferentiation via ForwardDiff.jl[30] if the user does not provide a function for calculating the Jocobian. However, ForwardDiff.jl currently does not include compatibility with complex numbers. Errors for these issues are only thrown at runtime. This leads to a combinatorial explosion in the amount of details required to describe the compatibility of each useful type with each method. Finding a way to better document the compatibilities and incompatibilities, as well as continuing to extend the compatibility, for this extended range of useful types is a long-term goal.

Also, DifferentialEquations.jl is currently limited on the types of PDEs it natively supports, and the mesh generation tools are still in their infancy. To address these issues and more, planned functionality includes (but is not limited to):

- Finite Difference Methods for common elliptic, parabolic, and hyperbolic PDEs, including high order methods for SPDEs
- Highly parallel accelerated solvers using GPGPUs and Xeon Phi cards (prototypes have already been developed [20])
- High order methods for stiff SDEs

Check the repository issues for the most up to date roadmap.

1.8 Quality Control

Continuous Integration testing with the latest versions of Julia on Mac, Linux, and Windows are provided via Travis and AppVoyer. These tests check most of the features of DifferentialEquations.jl, including the convergence of each algorithm, the ability to plot, the number types used in the computations, and more. Coveralls and Coverage badges are provided on the repository for test coverage analysis. As with other Julia packages, a user can check to see if these functionalities are working on their local machine via the command Pkg.test ("DifferentialEquations"). Benchmarks in Jupyter notebooks are provided to test the differences between the integrator implementations. Additionally, each Base library, component solver, and add-on package contains

its own set of tests for the functionality that it implements. All have the same continuous integration setup.

Another method for quality control is user feedback. DifferentialEquations.jl receives bug reports and feature requests through the Julialang Discourse[31], the Github issues page[32], and the JuliaDiffEq Gitter channel[33]. These are tracked and as releases occur, are broadcasted to the community using the JuliaDiffEq blog[34].

(2) Availability

2.1 Operating system

DifferentialEquations.jl is CI tested on MacOSX and Linux via Travis CI, and Windows via AppVoyer.

2.2 Programming language

Julia v0.5+

2.3 Dependencies

Dependencies are split into two groups. The direct dependencies of DifferentialEquations.jl are the packages of JuliaDiffEq which are built around the common interface and developed in tandem with DifferentialEquations.jl. The indirect dependencies are the dependencies of the direct dependencies, which are packages which are not actively developed as part of JuliaDiffEq activity.

The direct dependencies of DifferentialEquations.jl are the packages of JuliaDiffEq. These are:

- DiffEqBase.jl[35]
- StochasticDiffEq.jl[36]
- FiniteElementDiffEq.jl[37]
- DiffEqDevTools.jl[38]
- OrdinaryDiffEq.jl[39]
- AlgebraicDiffEq.jl[40]
- StokesDiffEq.jl[41]
- DiffEqParamEstim.jl[42]
- DiffEqSensitivity.jl[43]
- Sundials.jl[44]
- ODEInterfaceDiffEq.jl[45]
- ParameterizedFunctions.jl[46]
- DiffEqPDEBase.jl[47]
- DelayDiffEq.jl[48]
- DiffEqCallbacks.jl[49]
- DiffEqMonteCarlo.jl[50]
- DiffEqJump.jl[51]
- DiffEqFinancial.jl[52]
- DiffEqBiological.jl[53]
- MultiScaleArrays.jl[54]

Indirect dependencies include:

- RecipesBase.jl[55]
- Optim.jl[56]
- Parameters.jl[57]
- ForwardDiff.jl[58]
- IterativeSolvers.jl[59]
- GenericSVD.jl[60]
- Compat.jl[61]
- InplaceOps.jl[62]
- SymEngine.jl[63]

All of these dependencies will automatically install upon `Pkg.add("DifferentialEquations")`. See the REQUIRE files for the Julia packages for more information on their specific dependencies.

Optional dependencies of DifferentialEquations.jl include the additional solver packages:

- ODEInterface.jl[64]
- ODE.jl[65]
- LSODA.jl[66]

For information on how to install these libraries, see their respective repositories.

2.4 List of contributors

- Christopher Rackauckas, Lead Developer of JuliaDiffEq
- Mauro Werder, contributions to DiffEqBase.jl
- Scott P. Jones, contributions to DiffEqBase.jl
- Virgile Andreani, contributed the enhanced plotting functionality
- Ethan Levien, contributions to DiffEqMonteCarlo.jl and DiffEqJump.jl
- Michael Fiano, contributions to the documentation
- David Barton, contributions to the dense output in OrdinaryDiffEq.jl

2.5 Software location

***Archive*:** Zenodo
Name: JuliaDiffEq/DifferentialEquations.jl
Persistent identifier: DOI: 10.5281/zenodo.283869
Licence: MIT
Publisher: Christopher Rackauckas
Version published: v1.8.0
Date published: 2/9/2017
***Code repository*:** Github
Name: JuliaDiffEq/DifferentialEquations.jl
Persistent identifier: github.com/JuliaDiffEq/DifferentialEquations.jl
Licence: MIT
Date published: 09/21/2016

2.6 Language

English

(3) Reuse Potential

Differential equations form the bedrock of many scientific fields. Therefore, there is no question as to whether numerical differential equation solvers will be used, rather the question is which ones will be used. Julia is a relatively young language which is seeing rapid adoption in the fields of data science and scientific computing due to the performance and productivity that it offers. Because of this, many scientists using Julia will need these tools either as a means to analyze models themselves, or as intermediate tools in more complex methods. With its applicability to many classes of differential equations, its included analysis tools for performing parameter estimation and sensitivity analysis, and the rapid pace at which this software is being developed, DifferentialEquations.jl looks to be a viable choice for many Julians looking for a differential equations library. Lastly, as an open-source software with a modular

structure, it is easily extendable. For information on how to extend the functionality of DifferentialEquations.jl please see the Contributor's Guide at DiffEqDevDocs.jl.

Notes

[1] http://docs.juliadiffeq.org/latest/
[2] https://github.com/JuliaDiffEq/DiffEqTutorials.jl
[3] https://github.com/JuliaLang/IJulia.jl
[4] https://github.com/JuliaPlots/Plots.jl
[5] https://github.com/JuliaPy/PyPlot.jl
[6] http://matplotlib.org/
[7] https://github.com/symengine/SymEngine.jl
[8] https://github.com/JuliaDiffEq
[9] https://github.com/JuliaDiffEq/DiffEqBase.jl
[10] https://github.com/JuliaDiffEq/OrdinaryDiffEq.jl
[11] https://github.com/JuliaDiffEq/Sundials.jl
[12] https://github.com/JuliaDiffEq/ODE.jl
[13] https://github.com/luchr/ODEInterface.jl
[14] https://github.com/rveltz/LSODA.jl
[15] https://github.com/JuliaArbTypes/ArbFloats.jl
[16] https://github.com/stevengj/DecFP.jl
[17] https://github.com/JuliaDiffEq/DiffEqTutorials.jl/blob/master/ExtraODEFeatures/Feagin's%20Order%2010%2C%2012%2C%20and%2014%20methods.ipynb
[18] https://github.com/Keno/SIUnits.jl
[19] https://github.com/ajkeller34/Unitful.jl
[20] https://github.com/JuliaDiffEq/DiffEqTutorials.jl/blob/master/ExtraODEFeatures/Unit%20Checked%20Arithmetic%20via%20Unitful.ipynb
[21] https://github.com/JuliaDiffEq/DiffEqBenchmarks.jl/blob/master/Parallelism/Multithreaded%20Runge-Kutta%20Methods.ipynb
[22] https://github.com/JuliaDiffEq/OrdinaryDiffEq.jl
[23] http://devdocs.juliadiffeq.org/latest
[24] https://github.com/JuliaDiffEq/DifferentialEquations.jl/tree/master/benchmarks
[25] https://github.com/JuliaDiffEq/Sundials.jl
[26] https://github.com/luchr/ODEInterface.jl
[27] https://github.com/JuliaParallel/DistributedArrays.jl
[28] https://github.com/JuliaParallel/PETSc.jl
[29] https://github.com/kshyatt/CUSOLVER.jl
[30] https://github.com/JuliaDiff/ForwardDiff.jl
[31] https://discourse.julialang.org/
[32] https://github.com/JuliaDiffEq/DifferentialEquations.jl/issues
[33] https://gitter.im/JuliaDiffEq/Lobby
[34] http://juliadiffeq.org/news
[35] https://github.com/JuliaDiffEq/DiffEqBase.jl
[36] https://github.com/JuliaDiffEq/StochasticDiffEq.jl
[37] https://github.com/JuliaDiffEq/FiniteElementDiffEq.jl
[38] https://github.com/JuliaDiffEq/DiffEqDevTools.jl
[39] https://github.com/JuliaDiffEq/OrdinaryDiffEq.jl
[40] https://github.com/JuliaDiffEq/AlgebraicDiffEq.jl
[41] https://github.com/JuliaDiffEq/StokesDiffEq.jl
[42] https://github.com/JuliaDiffEq/DiffEqParamEstim.jl
[43] https://github.com/JuliaDiffEq/DiffEqSensitivity.jl
[44] https://github.com/JuliaDiffEq/Sundials.jl
[45] https://github.com/JuliaDiffEq/ODEInterfaceDiffEq.jl
[46] https://github.com/JuliaDiffEq/ParameterizedFunctions.jl
[47] https://github.com/JuliaDiffEq/DiffEqPDEBase.jl
[48] https://github.com/JuliaDiffEq/DelayDiffEq.jl
[49] https://github.com/JuliaDiffEq/DiffEqCallbacks.jl
[50] https://github.com/JuliaDiffEq/DiffEqMonteCarlo.jl
[51] https://github.com/JuliaDiffEq/DiffEqJump.jl
[52] https://github.com/JuliaDiffEq/DiffEqFinancial.jl
[53] https://github.com/JuliaDiffEq/DiffEqBiological.jl
[54] https://github.com/JuliaDiffEq/MultiScaleArrays.jl
[55] https://github.com/JuliaPlots/RecipesBase.jl
[56] https://github.com/JuliaNLSolvers/Optim.jl
[57] https://github.com/mauro3/Parameters.jl
[58] https://github.com/JuliaDiff/ForwardDiff.jl
[59] https://github.com/JuliaMath/IterativeSolvers.jl
[60] https://github.com/simonbyrne/GenericSVD.jl
[61] https://github.com/JuliaLang/Compat.jl
[62] https://github.com/simonbyrne/InplaceOps.jl
[63] https://github.com/symengine/SymEngine.jl
[64] https://github.com/luchr/ODEInterface.jl
[65] https://github.com/JuliaDiffEq/ODE.jl
[66] https://github.com/rveltz/LSODA.jl

Acknowledgments

We would like to thank the Julia community for the support they have given me throughout this project. Special thanks goes out to Ismael Venegas Castello (@Ismael-VC), Lyndon White (@oxinabox), Fengyang Wang (@TotalVerb), and Scott P. Jones (@ScottPJones) from whom CR learned many languages tricks in our long Gitter discussions. CR would also like to acknowledge Tom Breloff (@tbreloff) for his work on Plots.jl and the development of the recipe concept which allowed DifferentialEquations.jl to so easily meld with the plot functionalities. Lastly, CR would like to thank @finmod for repeatedly helping to identify documentation which needs updates.

Competing Interests

The authors have no competing interests to declare.

References

1. **Ahnert, K, Demidov, D** and **Mulansky, M** 2014 *Solving Ordinary Differential Equations on GPUs,* pp. 125–157. Springer International Publishing, Cham. DOI: https://doi.org/10.1007/978-3-319-06548-9_7
2. **Bezanson, J, Edelman, A, Karpinski, S** and **Shah, V B** 2017 Julia: A Fresh Approach to Numerical Computing, *SIAM Review*, 59(1), pp. 65–98. DOI: https://doi.org/10.1137/141000671
3. **Campbell, S A** 2007 *Time Delays in Neural Systems*, pp. 65–90, Springer Berlin Heidelberg, Berlin, Heidelberg.
4. **Carletti, M** 2006 Numerical solution of stochastic differential problems in the biosciences, *Journal of Computational and Applied Mathematics*, 185(2), pp. 422–440. DOI: https://doi.org/10.1016/j.cam.2005.03.020
5. **DeSantiago, R, Fouque, J-P** and **Sølna, K** 2007 *Bond Markets with Stochastic Volatility*.
6. **Fang, J, Varbanescu, A L, Sips, J, Zhang, L, Che, Y** and **Xu, C** 2013 An Empirical Study of Intel Xeon Phi, *CoRR*, abs/1310.5842.

7. **Gierer, A** and **Meinhardt, H** 1972 A theory of biological pattern formation, *Kybernetik*, 12(1), pp. 30–39. DOI: https://doi.org/10.1007/BF00289234
8. **Hairer, E, Nørsett, S P** and **Wanner, G** 2009 *Solving ordinary differential equations I : nonstiff problems*, Springer series in computational mathematics. Springer, Heidelberg; London, 2nd rev. ed.
9. **Higham, D J** An Algorithmic Introduction to Numerical Simulation of Stochastic Differential Equations.
10. **Hill, G W** 1900 On the Extension of Delaunay's Method in the Lunar Theory to the General Problem of Planetary Motion, *Transactions of the American Mathematical Society*, 1(2), pp. 205–242.
11. **Hindmarsh, A C, Brown, P N, Grant, K E, Lee, S L, Serban, R, Shumaker, D E** and **Woodward, C S** 2005 Sundials: Suite of nonlinear and differential/algebraic equation solvers, *ACM Trans. Math. Softw.*, 31(3), pp. 363–396. DOI: https://doi.org/10.1145/1089014.1089020
12. **James, W, Esther, W, Jonathan, H** and **Richard, M** 2016 Periodic orbits for a discontinuous vector field arising from a conceptual model of glacial cycles, *Nonlinearity*, 29(6), 1843. DOI: https://doi.org/10.1088/0951-7715/29/6/1843
13. **Jha, S K** and **Langmead, C J** 2012 Exploring behaviors of stochastic differential equation models of biological systems using change of measures, *BMC Bioinformatics*, 13(Suppl 5), pp. S8–S8. DOI: https://doi.org/10.1186/1471-2105-13-S5-S8
14. **Johansson, F** 2014 Efficient implementation of elementary functions in the medium-precision range, *CoRR*, abs/1410.7176.
15. **Jones, E, Oliphant, T, Peterson, P** et al. 2001 – SciPy: Open source scientific tools for Python, [Online; accessed 2017-03-17]
16. **Juno** 9/21/2016, *v0.2.1*, https://github.com/JunoLab/uber-juno.
17. **Lima, J V F, Maillard, N, Broquedis, F, Gautier, T, Lubin, M** and **Dunning, I** 2013 Performance evaluation of intel xeon phi coprocessor using xkaapi, *Workshop on Parallel and Distributed Processing.*
18. **Manabe, S** and **Bryan, K** 1969 Climate Calculations with a Combined Ocean-Atmosphere Model, *Journal of the Atmospheric Sciences*, 26(4), 786–789. DOI: https://doi.org/10.1175/1520-0469(1969)026<0786:CCWACO>2.0.CO;2
19. **Pérez, F** and **Granger, B E** 2007 May. IPython: a System for Interactive Scientific Computing, *Computing in Science and Engineering*, 9(3), 21–29. DOI: https://doi.org/10.1109/MCSE.2007.53
20. **Rackauckas, C** 2016 Interfacing with a Xeon Phi via Julia, *StochasticLifestyle.com*.
21. **Rackauckas, C** and **Nie, Q** 2016 Adaptive Methods for Stochastic Differential Equations via Natural Embeddings and Rejection Sampling with Memory, *Discrete and Continuous Dynamical Systems – Series B,* 22(7), pp. 2731–2761. DOI: https://doi.org/10.3934/dcdsb.2017133
22. **Saarinen, A, Linne, M-L** and **Yli-Harja, O** 2008 Stochastic Differential Equation Model for Cerebellar Granule Cell Excitability, *PLoS Comput Biol*, 4(2), e1000004. DOI: https://doi.org/10.1371/journal.pcbi.1000004
23. **Safarov, N** and **Atkinson, C** 2015 Natural Gas Storage Valuation and Optimisation Under Time-Inhomogeneous Exponential Lévy Processes, *International Journal of Computer Mathematics*.
24. **Shampine, L F** and **Gahinet, P** 2006 Delay-differential-algebraic equations in control theory, *Appl. Numer. Math.*, 56(3–4), pp. 574–588. DOI: https://doi.org/10.1016/j.apnum.2005.04.025
25. **Shampine, L F** and **Reichelt, M W** The MATLAB ODE Suite.
26. **Soetaert, K, Petzoldt, T** and **Setzer, R W** 2010 Solving Differential Equations in R: Package deSolve, *Journal of Statistical Software*, 33(9), pp. 1–25. DOI: https://doi.org/10.18637/jss.v033.i09
27. **Sosnik, J, Zheng, L, Rackauckas, C V, Digman, M, Gratton, E, Nie, Q** and **Schilling, T F** 2016 Noise modulation in retinoic acid signaling sharpens segmental boundaries of gene expression in the embryonic zebrafish hindbrain, *eLife*, 5, e14034. DOI: https://doi.org/10.7554/eLife.14034
28. **SymEngine** 9/21/2016 *v0.2.0,* https://github.com/symengine/symengine.
29. **Verner, J H** 2013 Explicit Runge Kutta pairs with lower stage-order, *Numerical Algorithms*, 65(3), pp. 555–577. DOI: https://doi.org/10.1007/s11075-013-9783-y
30. **Xiao, L, Cai, Q, Li, Z, Zhao, H** and **Luo, R** 2014 A multi-scale method for dynamics simulation in continuum solvent models. i: Finite-difference algorithm for Navier–Stokes equation, *Chemical Physics Letters*, 616–617, 67–74. DOI: https://doi.org/10.1016/j.cplett.2014.10.033

How to cite this article: Rackauckas, C and Nie, Q 2017 DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. *Journal of Open Research Software,* 5: 15, DOI: https://doi.org/10.5334/jors.151

Published: 29 September 2016 **Accepted:** 15 May 2017 **Published:** 25 May 2017

]u[*Journal of Open Research Software* is a peer-reviewed open access journal published by Ubiquity Press

OPEN ACCESS