DelayDiffEq: Generating Delay Differential Equation Solvers via Recursive Embedding of Ordinary Differential Equation Solvers

David Widmann [1] and Chris Rackauckas [2,3,4]

[1]Uppsala University, Sweden
[2]Massachusetts Institute of Technology, USA
[3]Julia Computing
[4]Pumas-AI

ABSTRACT

Traditional solvers for delay differential equations (DDEs) are designed around only a single method and do not effectively use the infrastructure of their more-developed ordinary differential equation (ODE) counterparts. In this work we present `DelayDiffEq`, a Julia package for numerically solving delay differential equations (DDEs) which leverages the multitude of numerical algorithms in `OrdinaryDiffEq` for solving both stiff and non-stiff ODEs, and manages to solve challenging stiff DDEs. We describe how compiling the ODE integrator within itself, and accounting for discontinuity propagation, leads to a design that is effective for DDEs while using all of the ODE internals. We highlight some difficulties that a numerical DDE solver has to address, and explain how `DelayDiffEq` deals with these problems. We show how `DelayDiffEq` is able to solve difficult equations, how its stiff DDE solvers give efficiency on problems with time-scale separation, and how the design allows for generality and flexibility in usage such as being repurposed for generating solvers for stochastic delay differential equations.

Keywords

Julia, Delay Differential Equation, Scientific Computing

1. Introduction

In nature many changes do not occur instantaneously, as prominent examples such as gestation times and incubation periods indicate. This lead to the rise of delay differential equations (DDEs) in many areas of mathematical modeling, such as population dynamics, pharmacokinetics, and control problems [2]. In contrast to ordinary differential equations (ODEs), DDEs allow the rate of change to depend on past conditions of the considered system.
However, while DDEs are a small conceptual change to ODEs, there is a larger mathematical difference that arises. Specifically, many DDEs have a discontinuity in the initial condition which, unlike ODEs, will continue to be in the system and propagate as we describe further in Section 2.3. Handling these propagations is crucial to performance and accuracy, and thus the software of this field, such as the MATLAB `dde23` [14] and the classic `RADAR5` [9], are completely separate software from those of ODE integrators. This gives rise to many development and maintenance issues: while in theory an advance in linear solvers (Newton-Krylov methods, preconditioners, etc.) can be used in DDE solvers, in practice none of these solvers support these features while their more-developed ODE counterparts do. This raises the question as to whether DDE software could be designed to more effectively make use of the ODE infrastructure so that all developments of the ODE solvers are inherited by the DDE solvers.
To answer this challenge we developed `DelayDiffEq`, a DDE solver package in Julia [4] which uses the ODE solvers of `OrdinaryDiffEq` [13] to develop the DDE solvers. We describe a form of "compilation trick", compiling an ODE integrator inside of an ODE integrator, which is used to generate the DDE solvers from the ODE solvers. This alone however does not guarantee the accurate handling of discontinuities, and thus `DelayDiffEq` then modifies the stepping to account for these issues. This gives a DDE solver where all methods of the ODE solver, ones for non-stiff and stiff equations, are directly inherited by the DDE solver with all of the available options. We show that this generated solver is capable of handling known difficult equations to verify its accuracy.[1]

2. Background

2.1 Definition

Loosely speaking, a delay differential equation (DDE) is an ordinary differential equation (ODE) in which the derivative depends on past values of the state. More concretely, let $t \in \mathbb{R}$ be the independent variable, which usually denotes time, and $x(t) \in \mathbb{R}^n$ be the dependent state at time t. Then the differential equation

$$x'(t) = f(t, x(t), x(t-\tau)), \tag{1}$$

where $f \colon \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n$ and $\tau > 0$, is called a DDE with constant delay τ. In general, a delay τ may also depend on t or even on both t and $x(t)$. In this case we call the DDE time-dependent or state-dependent, respectively. Additionally, DDEs may contain multiple delays. A further generalization is possible by allowing the state to depend also on its derivative at previous time points, yielding so-called neutral DDEs.

[1]The source code of the paper and its examples are available on Github.

2.2 History function

For ODEs, so-called initial value problems (IVPs) can be formulated by specifying an initial condition $x(t) = x_0$ at an initial time point t_0. For DDEs, usually declaring an initial value only at time point t_0 is not sufficient. Instead typically a so-called history function, or initial function, has to be provided for some times $t \leq t_0$. Thus commonly an IVP formulation for the DDE in eq. (1) is given by

$$\begin{aligned} x'(t) &= f(t, x(t), x(t - \tau)), & t &\geq t_0, \\ x(t) &= x_0(t), & t &\leq t_0. \end{aligned} \tag{2}$$

2.3 Propagated discontinuities

Even if x_0, f, and $\tau(t, x(t))$ are smooth, typically the solution x is non-smooth since it has a derivative jump discontinuity at the initial time point t_0, i.e.,

$$\lim_{t \to t_0} x_0'(t) \neq \lim_{t \to t_0} x'(t).$$

If there exist time points $t > t_0$ such that the deviated argument $t - \tau(t, x(t)) < t_0$, then the discontinuity at time point t_0 will propagate in time and induce discontinuities at subsequent time points. For non-neutral DDEs these propagated discontinuities are usually discontinuities of higher-order derivatives, leading to a so-called smoothing of the solution, whereas in general for neutral DDEs smoothing does not occur [3, 11].

2.4 Example: Dynamical structure

Mackey and Glass' model of circulating blood cells [8] is a scalar DDE system but still for certain parameter values we can observe a chaotic behavior. The model is given by

$$\begin{aligned} x'(t) &= \frac{\lambda \theta^n x(t - \tau)}{\theta^n + x(t - \tau)^n} - \gamma x(t), & t &\geq 0, \\ x(t) &= x_0, & t &\leq 0. \end{aligned} \tag{3}$$

For the parameter values $\lambda = 2$, $\theta = 1$, $n = 9.65$, $\tau = 2$, $\gamma = 1$, and $x_0 = 0.5$ the system exhibits a chaotic solution [8, Figure 6(e)]. In Figure 1 the chaotic solution computed with `DelayDiffEq` and its time delay embedding are shown for time $300 \leq t \leq 600$.

2.5 Method of steps

Consider an IVP for a DDE with one constant delay, given by eq. (2). Assuming that a solution to this problem exists, a solution can be found by computing solutions of an iterative sequence of IVPs for different ODEs. More concretely, for $k = 1, 2, \ldots$ one computes a solution $x_k \colon [t_{k-1}, t_{k-1} + \tau] \to \mathbb{R}^n$ of the IVP

$$\begin{aligned} x_k'(t) &= f(t, x_k(t), x_{k-1}(t - \tau)), \quad t \in [t_{k-1}, t_k], \\ x_k(t_{k-1}) &= x_{k-1}(t_{k-1}), \end{aligned} \tag{4}$$

where $t_k := t_0 + k\tau$. Then the function x defined by $x(t) := x_k(t) \iff t \in [t_{k-1}, t_k)$ is a solution to the original IVP in eq. (2). This algorithm for solving DDEs is called method of steps. It can be applied more generally also to IVPs of DDEs with multiple state-dependent delays, as long as the delays are uniformly strictly greater than 0. The same iterative approach can be used to transfer results from the theory of ODEs such as existence, uniqueness, and boundedness of solutions to DDEs without vanishing delays.

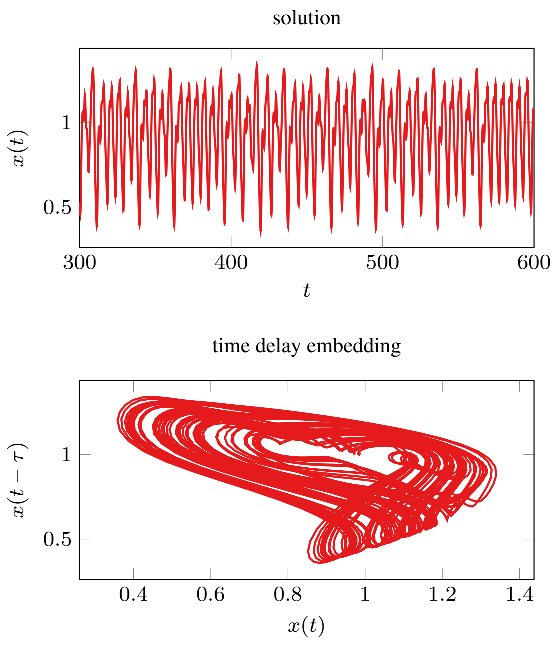

Fig. 1. Solution of problem (3) and its time delay embedding.

3. Numerical challenges

For DDEs, the computation of numerical solutions of IVPs can be challenging. In theory the method of steps allows to reduce the problem to the computation of a sequence of numerical solutions to IVPs of ODEs. A crucial requirement for this approach to work are so-called continuous ODE methods that provide a numerical solution not only for a grid of discrete time points but for all time points in the integration interval.

A disadvantage of the method of steps that it can be inefficient. The algorithm requires that the step size is upper bounded by the minimally occurring delay, and hence the number of time steps can become large if the maximally allowed step size is small relative to the integration time interval. Additionally, problems with vanishing delays cannot be solved by the method of steps.

Another difficulty is posed by the fact that usually numerical methods for solving ODEs expect the solution to be sufficiently smooth in each step. To ensure this, propagated discontinuities have to be considered and included in the grid of time points that the numerical method steps to [3]. For constant delays the relevant discontinuities can be computed a priori but for state-dependent delays this is not possible. Even for constant delays numerical difficulties can arise, as Shampine and Thompson [14] illustrate: two constant delays of $1/3$ and 1, albeit both inducing propagated discontinuities at time point $t_0 + 1$, might seem to cause discontinuities at different time points numerically since $1/3$ cannot be represented exactly by the floating point arithmetic of computers.

4. How DelayDiffEq works

So far one of the core design principles of `DelayDiffEq` has been to exploit the large amount of numerical ODE algorithms readily available in the Julia package `OrdinaryDiffEq`. The ODE solvers in `OrdinaryDiffEq` support both stiff and non-stiff ODEs[2]. Moreover, the numerical methods are continuous and come with specialized interpolations.

The design of `DelayDiffEq` is as follows. In order for a user to describe a DDE, we need to have a history function which the user can access. This history function is a continuous function of the states of the DDE solution, which, if we only had an ODE, would be given by the dense output (continuous interpolation) of the ODE solver. Thus what we do is the following:

(1) Create an ODE integrator from `OrdinaryDiffEq` with state size matching the DDE we wish to solve. The integrator comes with the property that $\text{integ}(t)$ computes the interpolation at time point t given the current solution of the ODE.

(2) Generate an ODE by enclosing this interpolation object into the user-defined DDE: $f_{\text{ODE}}(t, x(t)) = f(t, x(t), \text{integ})$.

(3) Create a new ODE integrator to solve the f_{ODE} system. If the interpolation integ is correct, then the solution of this new ODE will give the solution of the DDE.

The second ODE integrator will have the correct state values $x(t)$ and stage calculation if the function integ is correct. On the other hand, integ is only a correct interpolation if it has the correct state values $x(t)$ and stage calculations. Thus we effectively alias the caches between the two versions of the ODE integrators, so that a step of the second ODE integrator updates its history function, which then allows for an accurate step of the integrator.

Notably, this scheme does not require implementing any new numerical integrators and simply relies on the ODE integration itself. However, as described this scheme is correct only if a new step uses only the history of previous steps, i.e., only when the step size is smaller than the minimum delay time. Thus while this generates an ODE solver, more work must be done to make this routine efficient.

4.1 Unconstrained time stepping

As discussed above, it is desirable to not upper bound the step size that the numerical solver is allowed to take. However, unconstrained time stepping comes at a cost. Allowing unconstrained steps implies that the numerical solution of the DDE is only available as the solution to an implicit problem, even for explicit ODE methods. Essentially, one must know that integ is correct to know that the next step is correct, but if the time step is larger than the minimum delay then the integrator will use an inaccurate extrapolation from the previous time step. In order to correct for this behavior, we transform this problem into an implicit problem which can be solved via fixed-point iteration.

Formally, in the kth integration step we want to compute an approximation $\hat{x}_k \colon [t_{k-1}, t_k] \to \mathbb{R}^n$ of the solution to the DDE in eq. (1) with approximate solution $\hat{x}(t)$ for $t \leq t_{k-1}$ as history function. In `DelayDiffEq`, we perform a fixed-point iteration to compute $\hat{x}_k$ if it is only available implicitly. Hereby we proceed in the following way:

(1) Extend the dense solution $\hat{x}$ up to time t_{k-1} to the time interval $[t_{k-1}, t_k]$ by extrapolating the specialized collocation polynomials available in `OrdinaryDiffEq`.

(2) Based on the approximation up to time t_k, compute a discrete approximation of $x(t_k)$ and additional collocation stages in the time interval $[t_{k-1}, t_k]$ by using an ODE method from `OrdinaryDiffEq`.

(3) Obtain a new dense approximation of the solution on time interval $[t_{k-1}, t_k]$ by completing this discrete approximation using the specialized collocation polynomials in `OrdinaryDiffEq`.

(4) If in steps 2 and 3 the "old" dense approximation was only evaluated at time points $\leq t_{k-1}$, we stop. Otherwise we repeat steps 2 and 3 iteratively based on the newly calculated approximations, until some stopping criterion is met.

The fixed-point iteration can be performed efficiently since `OrdinaryDiffEq` allows us to specify that the Jacobian and its factorization should only be evaluated once. Optionally, Anderson acceleration [1, 16] can be used to accelerate the convergence of the fixed-point iteration. We have observed that, e.g., for the one-dimensional IVP

$$\begin{aligned} x'(t) &= -x(t - 1/3) - x(t - 1/5), && t \in [0, 100], \\ x(t) &= \delta_0(t), && t \leq 0, \end{aligned}$$

Anderson acceleration reduces the number of evaluations of the model, fixed-point iterations, and convergence failures from 5517, 720, and 58 to 2829, 308, and 30, respectively, when the explicit ODE method `Tsit5`, relative tolerance 10^{-3}, and absolute tolerance 10^{-6} are used.

4.2 Discontinuities

User-provided discontinuities and discontinuities arising from constant time delays up to the order of the ODE method are added as grid points. Existing functionality in `OrdinaryDiffEq` for handling discontinuities ensures that the numerical solver steps to these discontinuities exactly and integration is restarted appropriately at these time points.

Discontinuities due to state-dependent delays are handled by locating time points ξ that satisfy

$$\zeta = \xi - \tau(\tilde{x}(\xi), \xi), \tag{5}$$

where ζ is a known discontinuity and $\tilde{x}$ is the current approximation of the solution x, and adding them as grid points. We search for such discontinuities only if a step is rejected and rely on the error estimator otherwise. If a step is rejected, we check for sign changes of $\xi_i - \tau(\tilde{x}(\xi_i), \xi_i) - \zeta_j$ at pre-defined number of equally spaced time points ξ_i in the current time interval $[t_{k-1}, t_k]$, where $\zeta_j \leq t_{k-1}$ are known discontinuities. If a sign change is detected, a solution ξ of eq. (5) is calculated by a root finding algorithm and added as grid point, and the step size is reduced accordingly. We do not try to find the earliest time point that satisfies eq. (5) but stop as soon as one discontinuity is found in the time interval $[t_{k-1}, t_k]$.

A very similar algorithm is used by the DDE solver `RADAR5` by Guglielmi and Hairer [9], however, they additionally locate discontinuities if the error estimates increases above a certain threshold in subsequent time steps.

Users may also specify no time delays at all. In that case, the choice of a method with residual error estimator such as `RK4` or `OwrenZen5` is strongly recommended because the residual error estimator attempts to ensure accuracy over the entire solution interval instead of simply at step points. This may be required for the history function (i.e., the ODE integrator's interpolation) to have the relevant accuracy.

[2]Although it "is extremely difficult to classify stiffness in a completely rigorous manner" [6], it is commonly characterized by the existence of processes with significantly different time scales [6, 7].

4.3 Interface

We do not present all features and options of `DelayDiffEq` but only provide one example for how DDEs can be declared and solved with `DelayDiffEq`. For a more thorough explanation we refer the reader to the extensive documentation of the `DifferentialEquations` framework.[3]
The interface of `DelayDiffEq` is similar to the one for ODEs in `OrdinaryDiffEq` or stochastic differential equations (SDEs) in `StochasticDiffEq` [12]. The code snippet below shows how a version of the so-called Hutchinson's equation [10], a population growth model also known as delay logistic equation, can be formulated with `DelayDiffEq`. The model is given by

$$\begin{aligned} x'(t) &= -x(t-1), && t \geq 0, \\ x(t) &= 1, && t \leq 0. \end{aligned} \tag{6}$$

In `DelayDiffEq`, a DDE problem consists of the differential equation, the history function, the integration time span, and the time delays. Optionally a value at the initial time point and model parameters can be provided.

```
using DelayDiffEq

# Hutchinson's equation
f(x, h, p, t) = - h(p, t - 1)

# History function
h(p, t) = 1.0

# Integration time span
tspan = (0.0, 10.0)

# Complete DDE problem formulation
prob = DDEProblem(f, h, tspan;
                  constant_lags = (1.0,))
```

Although not useful in this case, for illustrative purposes we demonstrate how a problem with state-dependent delay can be formulated.

```
# Complete DDE problem formulation
# with state-dependent delay
lag(p, t) = 1.0
prob = DDEProblem(f, h, tspan;
                  dependent_lags = (lag,))
```

In both cases presented above,

```
solve(prob, MethodOfSteps(Tsit5()))
```

computes a solution to the problem with the ODE method `Tsit5`, using the default solver options. `DelayDiffEq` supports DDEs with mass matrices and neutral DDEs as well. Similar to the ODE case, singular mass matrix DDEs generate differential-algebraic equations (DAEs) which are then solved via the `OrdinaryDiffEq` DAE solvers.
From the structure of the design, `DelayDiffEq` includes all of the features of the `OrdinaryDiffEq` ODE solvers, including (but not limited to):

—Explicit Runge-Kutta methods from 1st order to 9th order, and arbitrary order extrapolation methods
—Semi-implicit Rosenbrock methods (allows for singular mass matrices)
—Implicit methods via Singularly Diagonally Implicit Runge-Kutta methods (SDIRK) and Backwards Differentiation Formulae (BDF)
—Fully-Implicit Runge-Kutta (FIRK) methods
—Stabilized explicit methods via Runge-Kutta Chebyshev methods (ROCK schemes)
—Dense output
—Support for sensitivity analysis via forward and reverse mode automatic differentiation
—Event handling for hybrid equations
—Jacobians for implicit methods defined by automatic differentiation, along with the ability to define analytical Jacobians
—Support for a wide range of linear solvers, such as preconditioned GMRES

We note in passing that the quasi-Newton schemes used in the ODE solvers only require the Jacobian and its factorization at the beginning of the step for the non-FIRK implicit methods. This means that the entire handling of the unconstrained time stepping of Section 4.1 can be done without recalculating the Jacobian or refactorizing the Jacobian. As these steps are commonly the most costly aspect for solving large stiff equations, this greatly improves the efficiency of the implementation over using a naive Newton solve.

5. Examples

5.1 Stiff problem: Antibody production

Next we consider a DDE model of antigen stimulated antibody production by Waltman [17]. The model is given by

$$\begin{aligned} x_1'(t) &= -rx_1(t)x_2(t) - sx_1(t)x_4(t), \\ x_2'(t) &= -rx_1(t)x_2(t) + \alpha r x_1(x_5(t))x_2(x_5(t))H(t-t_1), \\ x_3'(t) &= rx_1(t)x_2(t), \\ x_4'(t) &= -sx_1(t)x_4(t) - \gamma x_4(t) \\ &\quad + \beta r x_1(x_6(t))x_2(x_6(t))H(t-t_2), \\ x_5'(t) &= H(t-t_1)\frac{x_1(t)x_2(t)+x_3(t)}{x_1(x_5(t))x_2(x_5(t))+x_3(x_5(t))}, \\ x_6'(t) &= H(t-t_2)\frac{10^{-12}+x_2(t)+x_3(t)}{10^{-12}+x_2(x_6(t))+x_3(x_6(t))}, \end{aligned} \tag{7}$$

where H is the Heavyside step function, i.e., $H(t) = 0$ if $t < 0$ and $H(t) = 1$ if $t \geq 0$. As Guglielmi and Hairer [9], we consider the model on the time interval $t \in [0, 300]$ with parameter values $\alpha = 1.8$, $\beta = 20$, $\gamma = 0.002$, $r = 5 \times 10^4$, $s = 10^5$, $t_1 = 35$, $t_2 = 197$, and history functions $x_1(t) = 5 \times 10^{-6}$, $x_2(t) = 10^{-15}$, and $x_3(t) = x_4(t) = x_5(t) = x_6(t) = 0$ for all $t \leq 0$.
This problem is very stiff, contains vanishing state-dependent delays, and displays steep increases in components x_2, x_4, and x_6 at time points t_1 and t_2 [9, 17]. Guglielmi and Hairer [9] compared their code RADAR5 with five existing numerical solvers, and only RADAR5 was able to solve problem (7).[4] As shown in Figure 2, `DelayDiffEq` manages to reproduce Figure 3 in [9] using the stiff

[3]The documentation is available online at `https://docs.sciml.ai/DiffEqDocs/stable/`.

[4]A later produced blog post showed that Maple was able to successfully solve the problem, while Mathematica and MATLAB solvers were not.

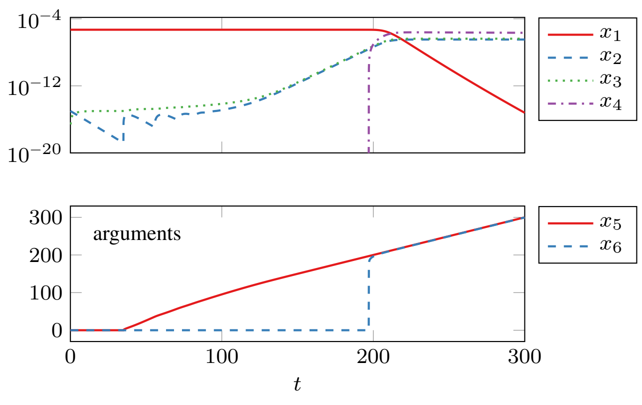

Fig. 2. Solution of problem (7).

ODE solver `KenCarp5`. As Guglielmi and Hairer [9], we used relative tolerances of 10^{-9} and absolute tolerances 10^{-21} and 10^{-9} for the first four and the last two components, respectively. We are pleased to see that `DelayDiffEq` can cope with the discontinuities at the time points $t_1 = 35$ and $t_2 = 197$, even though they are not declared as grid points as in [9].

5.2 Sensitivity analysis via automatic differentiation

The `SciMLSensitivity` tutorials highlight that `DelayDiffEq` is compatible with forward- and reverse-mode automatic differentiation. For this example, we define a Lotka-Volterra model with delay and constant history function:

```
using DelayDiffEq
using SciMLSensitivity, ForwardDiff, Zygote

# Lotka-Volterra model with delay
function lv!(du, u, h, p, t)
    x, y = u
    α, β, δ, γ = p
    xτ = h(p, t - 0.1)[1]
    du[1] = dx = (α - β * y) * xτ
    du[2] = dy = (δ * x - γ) * y
    return nothing
end

# Constant history function
history_lv(p, t) = ones(2)
```

The following function computes the solution of the DDE model on an equally-spaced time grid:

```
function predict_lv(p)
    # Define the DDE problem
    prob = DDEProblem(lv!, history_lv, (0.0, 5.0),
                      p;
                      constant_lags = (0.1,))

    # Solve the problem and save the solution at
    # time points 0, 0.1, 0.2, ..., 5
    sol = solve(prob, MethodOfSteps(Tsit5());
                saveat = 0.1)

    return Array(sol)
end
```

This function includes the DDE solver and is compatible with forward- and reverse mode automatic differentiation:

```
# Parameters
p = [2.2, 1.0, 2.0, 0.4]

# Compute Jacobian
jacFD = ForwardDiff.jacobian(predict_lv, p)
jacZygote = Zygote.jacobian(predict_lv, p)[1]
```

5.3 Algorithm development and benchmarking

To facilitate the research and development of efficient DDE solving, `DelayDiffEq` ties into the SciMLBenchmarks ecosystem to provide an easy way to benchmark its DDE solvers. A demonstration of a work-precision diagram generated by `DiffEqDevTools` on a delay differential equation model of quorum sensing [5] is shown in Figure 3.

```
using DelayDiffEq
using DiffEqDevTools
using DDEProblemLibrary

function benchmark()
    # Compute reference solution of third state
    # with low tolerances
    fpsolve = NLFunctional(; max_iter = 1000)
    alg = MethodOfSteps(Vern9(); fpsolve)
    sol = TestSolution(solve(prob_dde_qs, alg;
                             reltol = 1e-14,
                             abstol = 1e-14,
                             save_idxs = 3))

    # Benchmark different algorithms
    setups = [
        Dict(:alg => MethodOfSteps(RK4())),
        Dict(:alg => MethodOfSteps(DP5())),
        Dict(:alg => MethodOfSteps(OwrenZen3())),
        Dict(:alg => MethodOfSteps(OwrenZen5())),
        Dict(:alg => MethodOfSteps(Vern6())),
        Dict(:alg => MethodOfSteps(Vern7())),
        Dict(:alg => MethodOfSteps(Vern8())),
        Dict(:alg => MethodOfSteps(Vern9())),
        Dict(:alg => MethodOfSteps(Rosenbrock23())),
        Dict(:alg => MethodOfSteps(Rodas4())),
        Dict(:alg => MethodOfSteps(Rodas5P())),
    ]
    abstols = exp10.(-(4:11))
    reltols = exp10.(-(1:8))
    return WorkPrecisionSet(prob_dde_qs,
                            abstols,
                            reltols,
                            setups;
                            save_idxs = 3,
                            appxsol = sol,
                            maxiters = Int(1e5),
                            error_estimate = :l2)
end
```

Notice how with very little code, DDE integrators for non-stiff and stiff DDEs can be compared to help optimize the implementations.

5.4 Stochastic delay differential equations

This same infrastructure of `DelayDiffEq` was used on `StochasticDiffEq` for SDEs to generate convergent solvers for stochastic delay differential equations, as was demonstrated in [15].

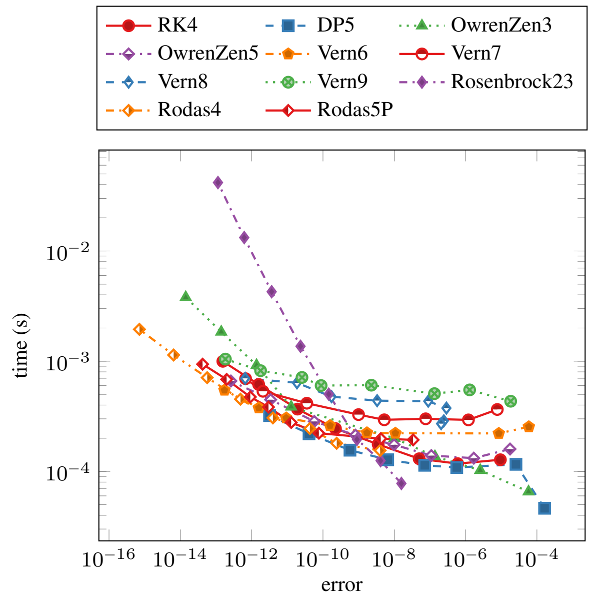

Fig. 3. Benchmarks on a DDE model of quorum sensing [5].

6. Conclusion

In this paper we presented `DelayDiffEq`, a Julia software package for numerically solving DDEs. We explained the employed numerical algorithm and showed that it can successfully solve very stiff systems with state-dependent delays. But most importantly, the design of `DelayDiffEq` allows it to inherit all of the development work from the `OrdinaryDiffEq` solvers of `DifferentialEquations`, a design which in turns leads to rapid development of `DelayDiffEq` even without developers having to focus on DDEs. This is thus an example of how composable software designs can change the efficiency of scientific computing.

ACKNOWLEDGMENTS

Large parts of this work were completed for a Master thesis about delay differential equation models of quorum sensing at the department of Mathematics at TU Munich, Germany, under the supervision of Christina Kuttler. This work was financially supported by the Centre for Interdisciplinary Mathematics (CIM) at Uppsala University, Sweden, and by the Swedish Foundation for Strategic Research via the project *ASSEMBLE* (contract number: RIT15-0012).
This material is based upon work supported by the National Science Foundation under grant no. OAC-1835443, grant no. SII-2029670, grant no. ECCS-2029670, grant no. OAC-2103804, and grant no. PHY-2021825. We also gratefully acknowledge the U.S. Agency for International Development through Penn State for grant no. S002283-USAID. The information, data, or work presented herein was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0001211 and DE-AR0001222. We also gratefully acknowledge the U.S. Agency for International Development through Penn State for grant no. S002283-USAID. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. This material was supported by The Research Council of Norway and Equinor ASA through Research Council project "308817 - Digital wells for optimal production and drainage". Research was sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

7. References

[1] Donald G. Anderson. Iterative procedures for nonlinear integral equations. *Journal of the ACM*, 12(4):547–560, 10 1965. doi:10.1145/321296.321305.

[2] Christopher T. H. Baker, Christopher A. H. Paul, and David R. Willé. Issues in the numerical solution of evolutionary delay differential equations. *Advances in Computational Mathematics*, 3(1):171–196, 1995. doi:10.1007/bf02988625.

[3] Alfredo Bellen and Marino Zennaro. *Numerical Methods for Delay Differential Equations*. Oxford University Press, 3 2003. doi:10.1093/acprof:oso/9780198506546.001.0001.

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[5] Katharina Buddrus-Schiemann, Martin Rieger, Marlene Mühlbauer, Maria Vittoria Barbarossa, Christina Kuttler, Burkhard A. Hense, Michael Rothballer, Jenny Uhl, Juliano R. Fonseca, Philippe Schmitt-Kopplin, Michael Schmid, and Anton Hartmann. Analysis of *N*-acylhomoserine lactone dynamics in continuous cultures of *Pseudomonas putida* IsoF by use of ELISA and UHPLC/qTOF-MS-derived measurements and mathematical models. *Analytical and Bioanalytical Chemistry*, 406(25):6373–6383, August 2014. doi:10.1007/s00216-014-8063-6.

[6] Kevin Burrage. *Parallel and sequential methods for ordinary differential equations*. Clarendon Press, Oxford, 1995.

[7] Germund Dahlquist. Recent work on stiff differential equations. Technical report, CM-P00069399, 1975.

[8] Leon Glass and Michael C. Mackey. Pathological conditions resulting from instabilities in physiological control systems. *Annals of the New York Academy of Sciences*, 316(1):214–235, 1979.

[9] Nicola Guglielmi and Ernst Hairer. Implementing Radau IIA methods for stiff delay differential equations. *Computing*, 67(1):1–12, 07 2001. doi:10.1007/s006070170013.

[10] G. Evelyn Hutchinson. Circular causal systems in ecology. *Annals of the New York Academy of Sciences*, 50(4 Teleological):221–246, 1948. doi:10.1111/j.1749-6632.1948.tb39854.x.

[11] Kenneth W. Neves and Alan Feldstein. Characterization of jump discontinuities for state dependent delay differential equations. *Journal of Mathematical Analysis and Applications*, 56(3):689–707, 12 1976. doi:10.1016/0022-247x(76)90033-0.

[12] Christopher Rackauckas and Qing Nie. Adaptive methods for stochastic differential equations via natural embeddings and rejection sampling with memory. *Discrete and continuous dynamical systems. Series B*, 22(7):2731, 2017.

[13] Christopher Rackauckas and Qing Nie. DifferentialEquations.jl – A performant and feature-rich ecosystem for solving differential equations in julia. *The Journal of Open Research Software*, 5(1), 2017. doi:10.5334/jors.151. Exported from https://app.dimensions.ai on 2019/05/05.

[14] Lawrence F. Shampine and Sylvester Thompson. Solving DDEs in Matlab. *Applied Numerical Mathematics*, 37(4):441–458, 6 2001. doi:10.1016/s0168-9274(00)00055-6.

[15] Henrik T. Sykora, Christopher Rackauckas, David Widmann, and Dániel Bachrathy. StochasticDelayDiffeq.jl - An integrator interface for stochastic delay differential equations in Julia. 2020.

[16] Homer F. Walker and Peng Ni. Anderson acceleration for fixed-point iterations. *SIAM Journal on Numerical Analysis*, 49(4):1715–1735, 1 2011. doi:10.1137/10078356x.

[17] Paul Waltman. A threshold model of antigen-stimulated antibody production. *Theoretical Immunology*, pages 437–453, 1978.