

Integrating equation solvers with probabilistic programming through differentiable programming

Chris Rackauckas

Director of Modeling and Simulation,
Julia Computing

Research Affiliate, Co-PI of Julia Lab,
Massachusetts Institute of Technology, CSAIL

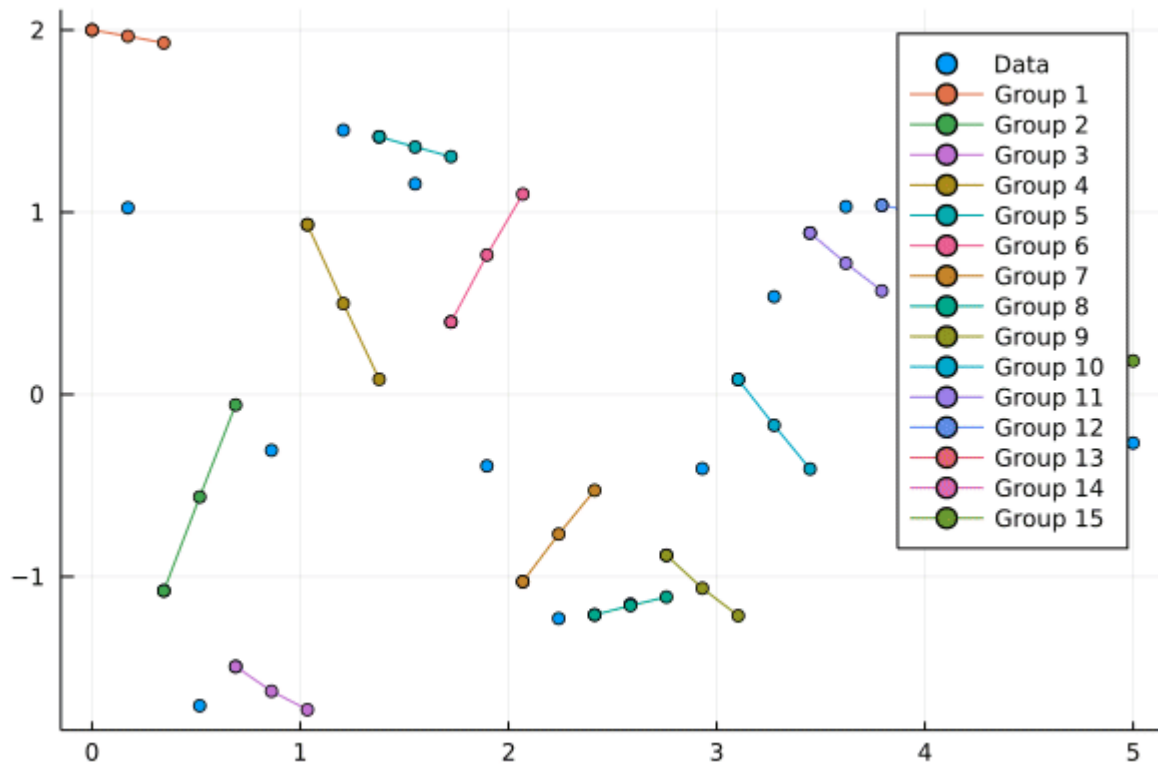
Director of Scientific Research,
Pumas-AI

Turing.jl arguably has both the most extensive differential equation solving support, and no support for differential equations at all.

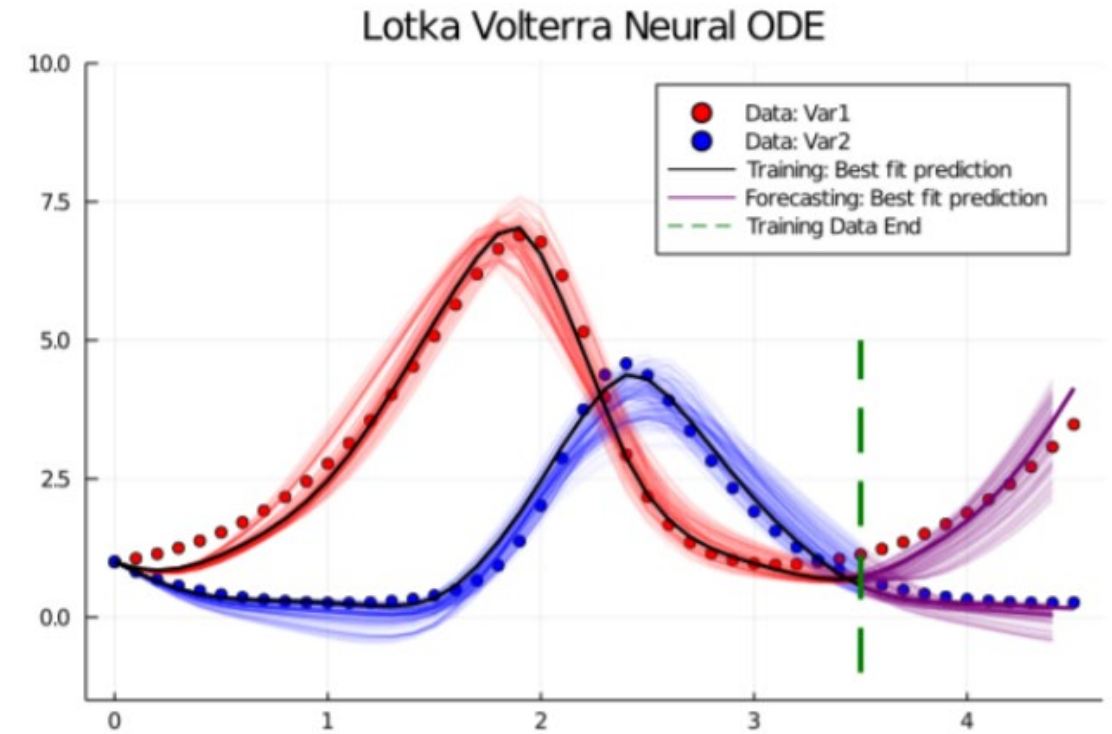
Let me explain.

Outline

Mixing equation discovery into epidemic modeling workflows will revolutionize the field



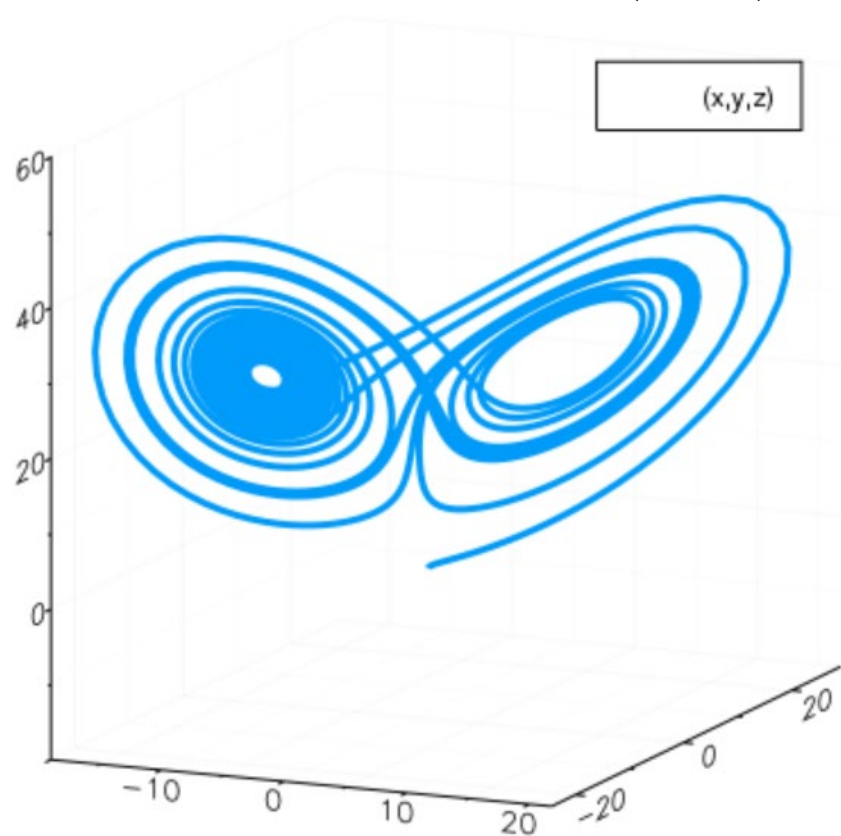
1. What could “extensive” differential equation support even mean?



2. Does Turing.jl have it? And what does that mean about its developer community?

ODEs are Simple! Just call an ODE Solver!

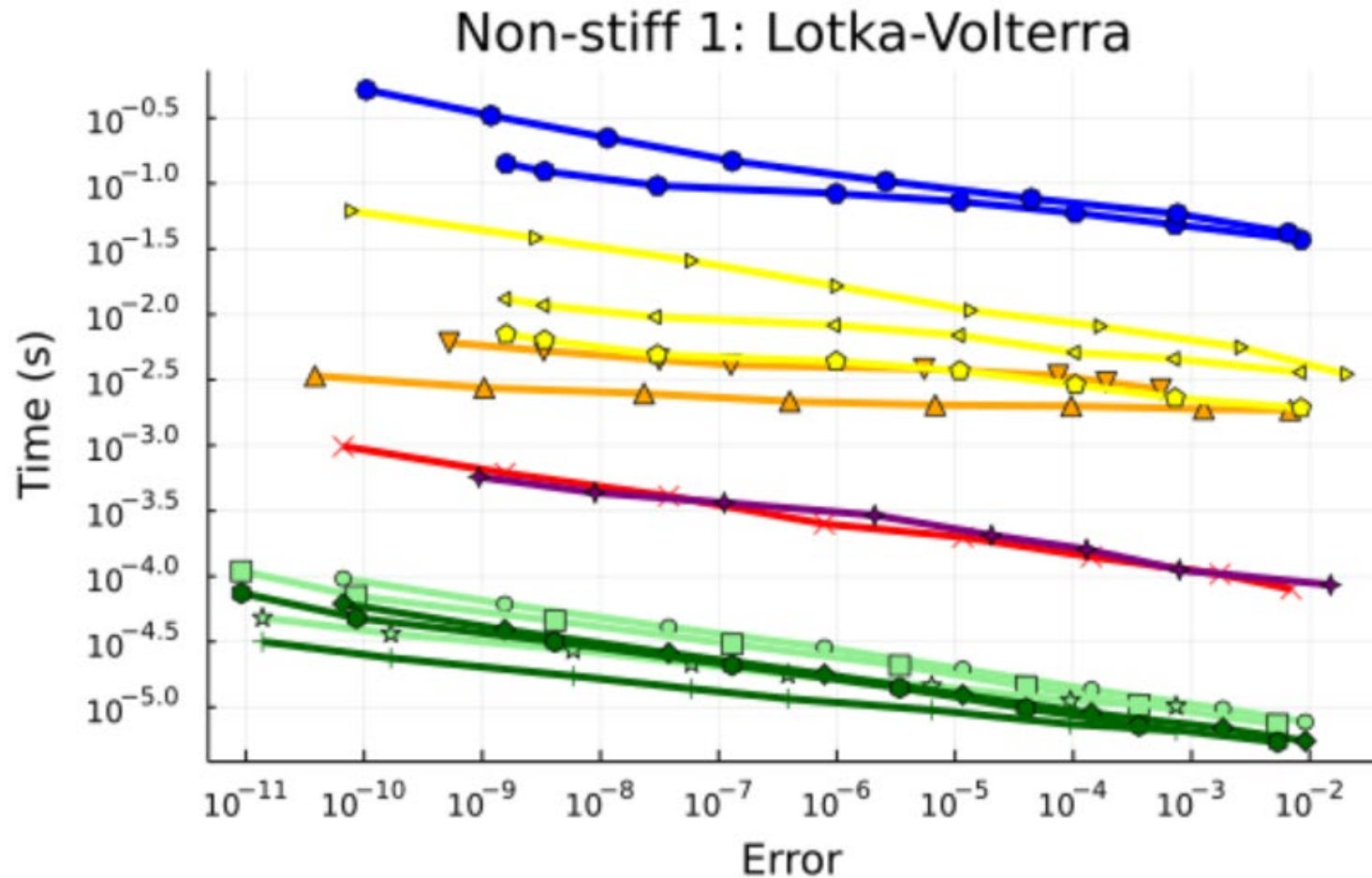
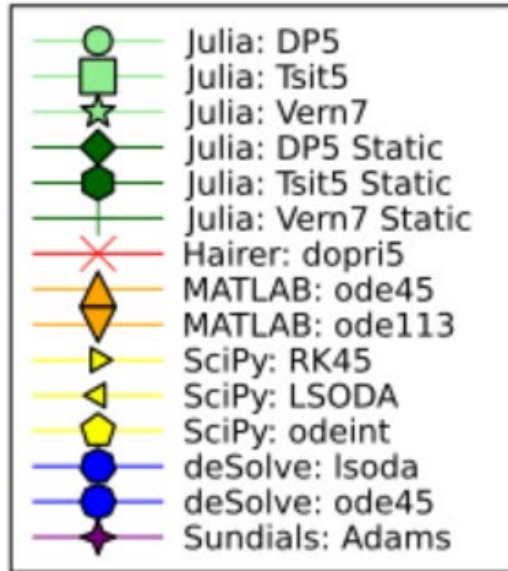
Problem: Lorenz equation on t in $(0,100)$



```
using OrdinaryDiffEq
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())
using Plots;
plot(sol, vars=(1, 2, 3));
```

But there are lots of little optimizations that can be done

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



Foundation: Fast Differential Equation Solvers

1. Speed
2. Stability
3. Stochasticity
4. Adjoint and Inference
5. Parallelism

DifferentialEquations.jl is generally:

- 50x faster than SciPy
- 50x faster than MATLAB
- 100x faster than R's deSolve

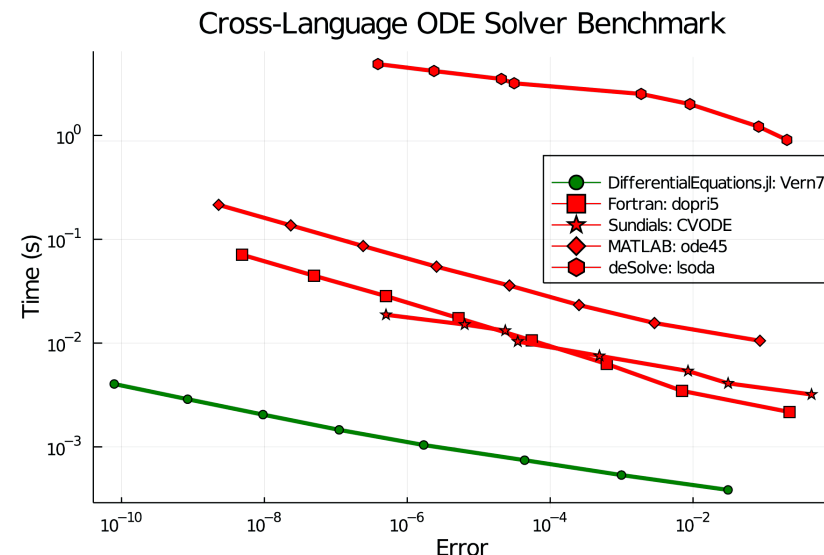
When optimally JIT compiling Py/Mat/R

<https://github.com/SciML/SciMLBenchmarks.jl>

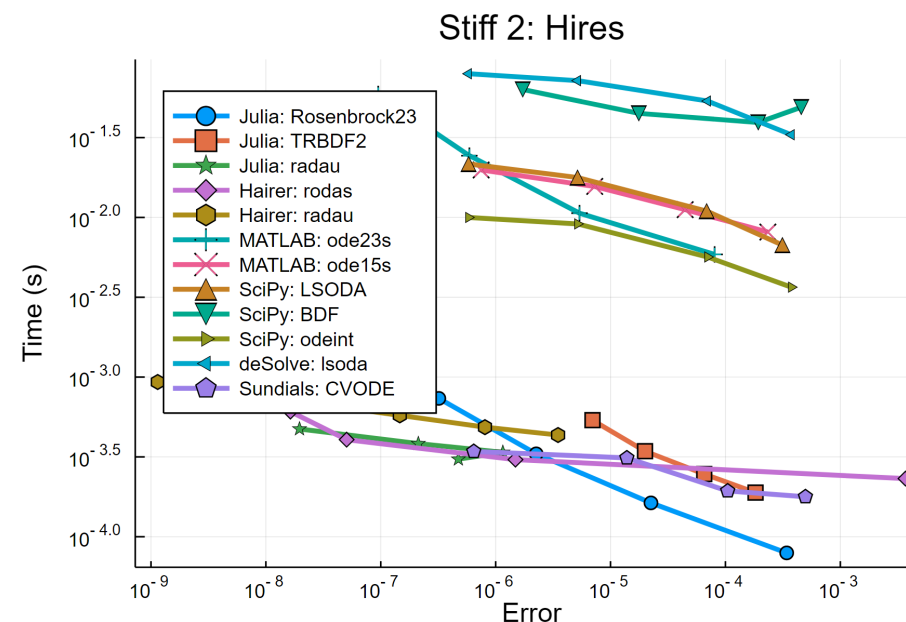
Rackauckas, Christopher, and Qing Nie. "Differenialequations.jl—a performant and feature-rich ecosystem for solving differential equations in julia." Journal of Open Research Software 5.1 (2017).

Rackauckas, Christopher, and Qing Nie. "Confederated modular differential equation APIs for accelerated algorithm development and benchmarking." Advances in Engineering Software 132 (2019): 1-6.

Non-Stiff ODE: Rigid Body System



8 Stiff ODEs: HIRES Chemical Reaction Network



Foundation: Fast Differential Equation Solvers

DifferentialEquations.jl is:

- Faster than C codes like CVODE and Fortran codes like LSODE/LSODA on stiff equations
- Has symbolic compilers to automatically improve numerical stability and performance of user code

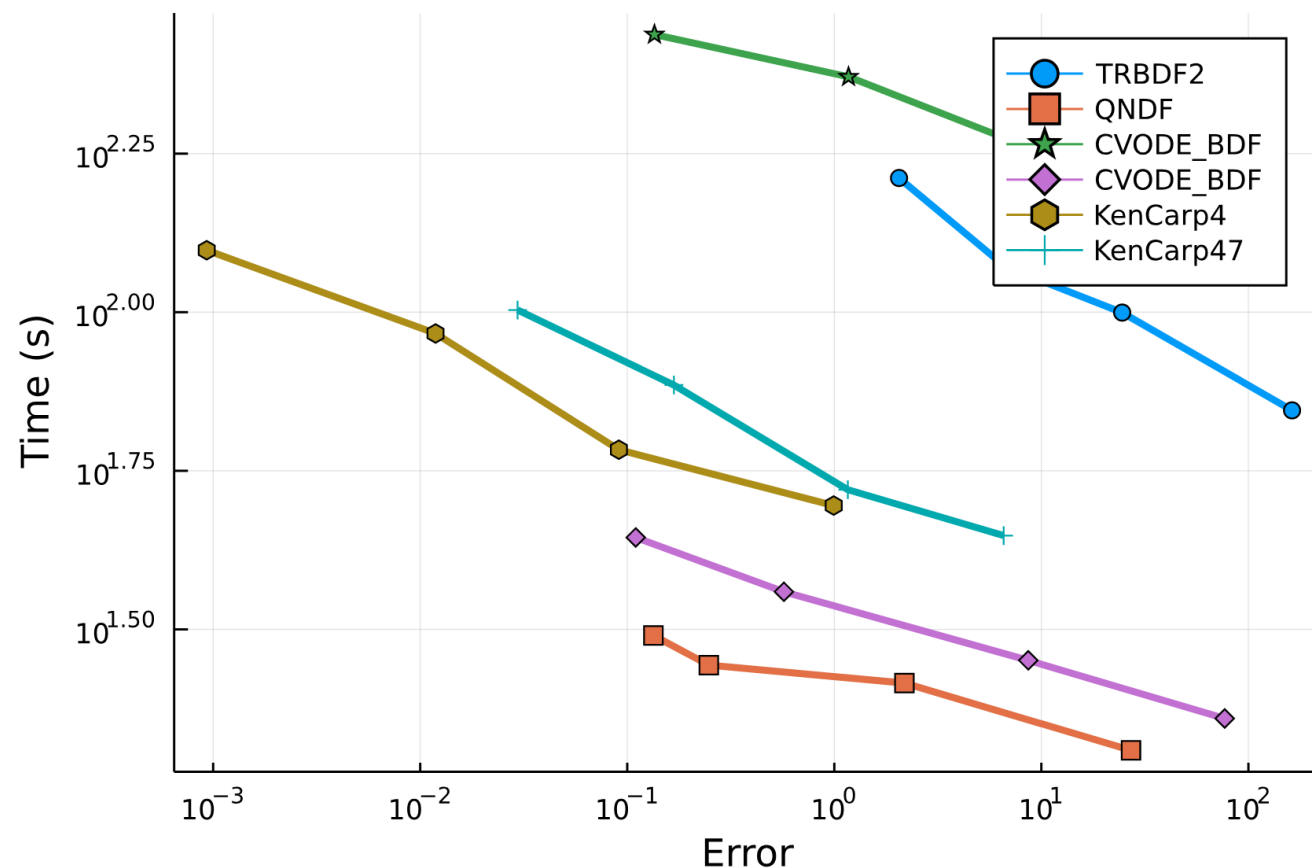
This excludes the extra 2x from symbolics and 2x from sparse parallel compilation!

<https://github.com/SciML/SciMLBenchmarks.jl>

Gowda, Shashi, Yingbo Ma, Alessandro Cheli, Maja Gwozdz, Viral B. Shah, Alan Edelman, and Christopher Rackauckas. "High-performance symbolic-numerics via multiple dispatch." To appear in ACM Communications in Computer Algebra (2021).

Ma, Yingbo, Shashi Gowda, Ranjan Anantharaman, Chris Laughman, Viral Shah, and Chris Rackauckas. "ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling." Submitted (2021).

1122 Stiff ODEs: BCR Chemical Reaction Network



Speed alone does not give good robust differential equation solving.

There's a lot more to the algorithms.

What is stiffness?

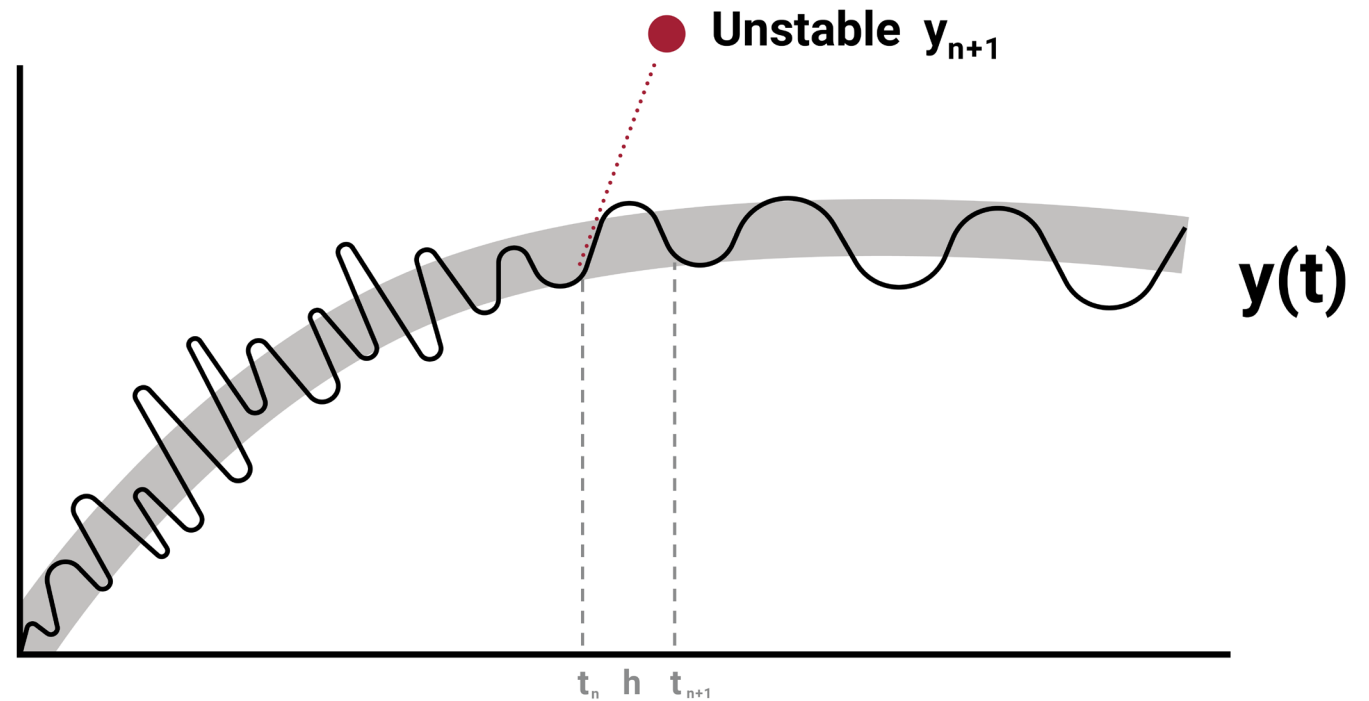
Multiscale Behavior

$$S = \frac{\max(|\operatorname{Re}(\lambda)|)}{\min(|\operatorname{Re}(\lambda)|)} (t_{\text{final}} - t_0)$$

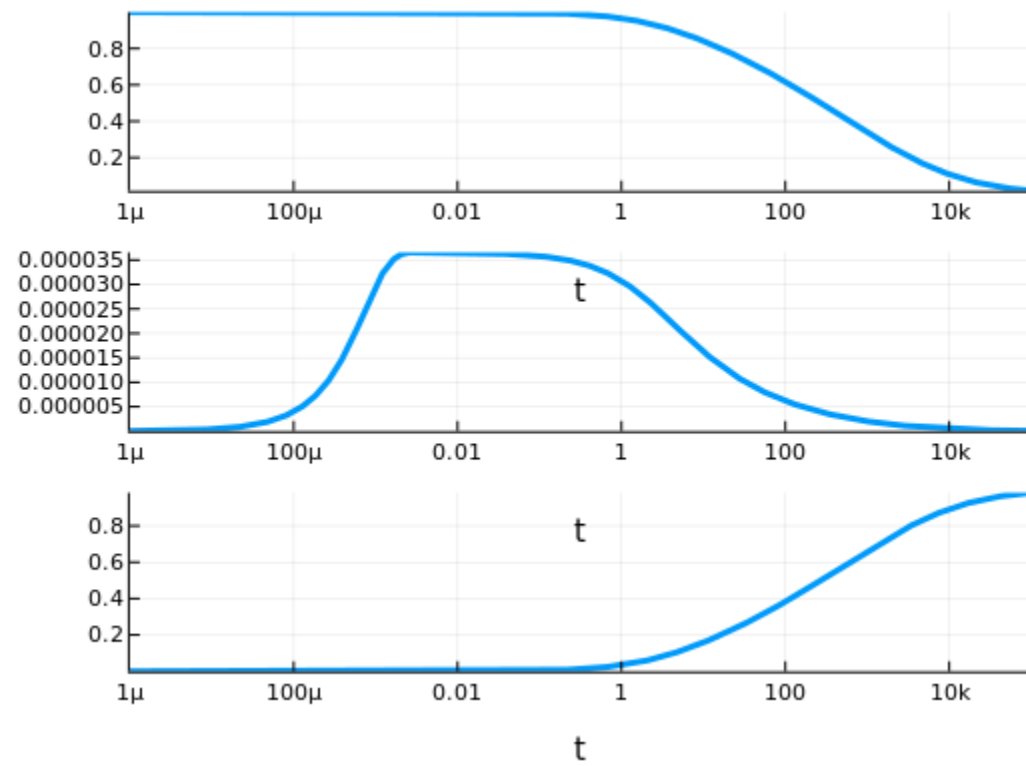
“

Stiff equations are problems for which explicit methods don't work.

- Ernst Hairer



So you just use a stiff ODE solver? 2 Solvers and we're done?



```
using DifferentialEquations
function rober(du, u, p, t)
    y1, y2, y3 = u
    k1, k2, k3 = p
    du[1] = -k1 * y1 + k3 * y2 * y3
    du[2] = k1 * y1 - k2 * y2^2 - k3 * y2 * y3
    du[3] = k2 * y2^2
    nothing
end
prob = ODEProblem(rober, [1.0, 0.0, 0.0], (0.0, 1e5), [0.04, 3e7, 1e4])
sol = solve(prob, Rodas5())
plot(sol, tspan=(1e-2, 1e5), xscale=:log10)
```

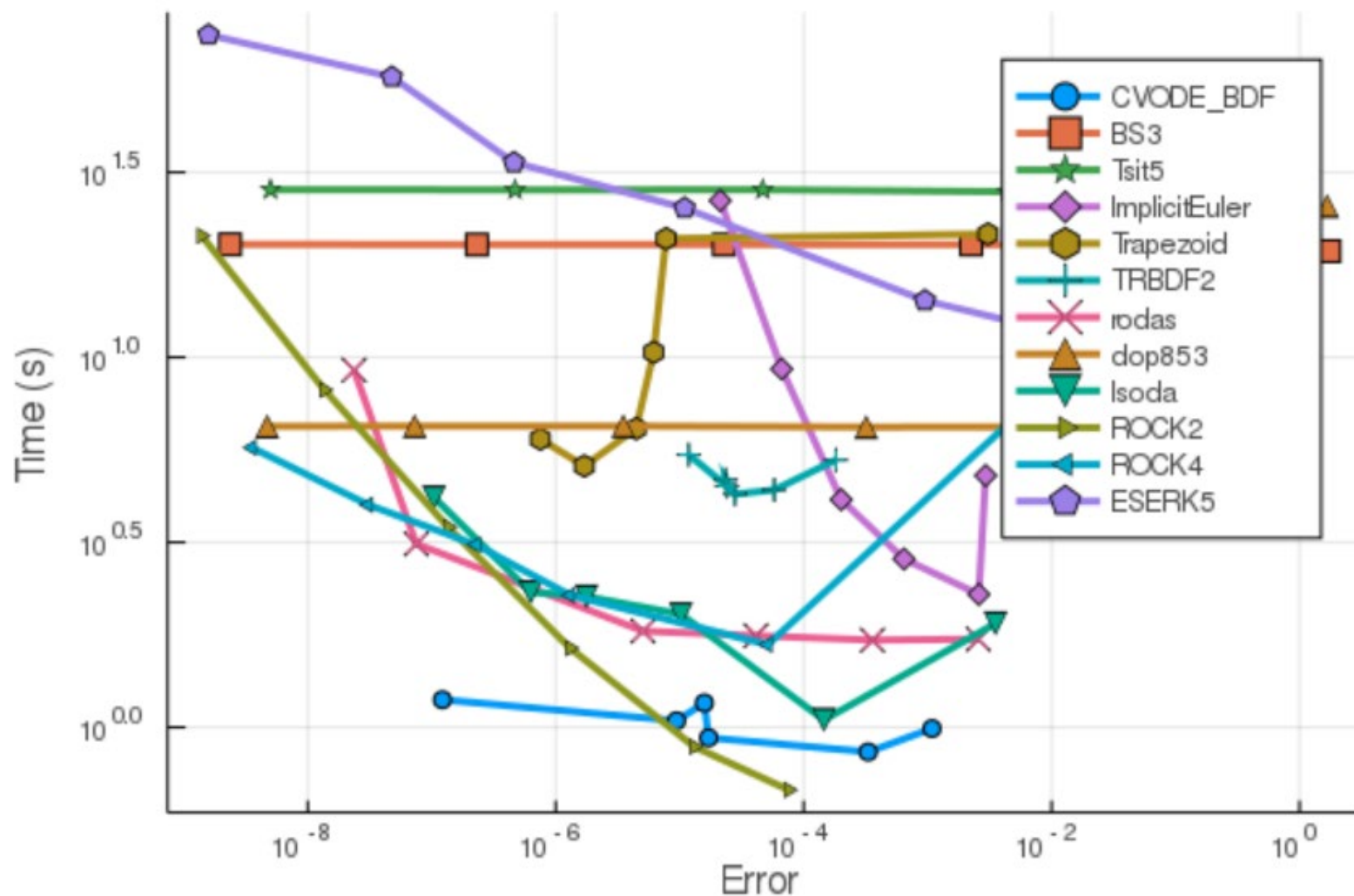
Okay, maybe more than two just so you can optimize the performance?

3 Solvers?

Magnetic Dipole PDE

Neither stiff nor non-stiff

Performance winner: ROCK2



There are lots of special properties

Some problems need special integrators

- Symplectic integrators for long time Hamiltonian systems
- Magnus for $u' = A(t) * u$
- Munthe-Kaas methods for $u' = A(u) * u$
- Nystrom specializations for 2nd order
- Exponential integrators for semilinear ODEs $u' = Au + f(u)$
- Implicit-Explicit (IMEX) methods for partly stiff equations
- Runge-Kutta-Chebyshev methods for semi-stiff equations

...

StackExchange

Search on Computational Science...

11,765 ● 1 ● 38 ● 66

Computational Science

Home

PUBLIC

Questions

What does "symplectic" mean in reference to numerical integrators, and does SciPy's odeint use them?

Asked 4 years, 3 months ago Modified 5 months ago Viewed 8k times

75

Let me start off with corrections. No, `odeint` doesn't have any symplectic integrators. No, symplectic integration doesn't mean conservation of energy.

What does symplectic mean and when should you use it?

First of all, what does symplectic mean? Symplectic means that the solution exists on a symplectic manifold. A symplectic manifold is a solution set which is defined by a 2-form. The details of symplectic manifolds probably sound like mathematical nonsense, so instead the gist of it is there is a direct relation between two sets of variables on such a manifold. The reason why this is important for physics is because Hamiltonian's equations naturally have that the solutions reside on a symplectic manifold in phase space, with the natural splitting being the position and momentum components. For the true Hamiltonian solution, that phase space path is constant energy.

A symplectic integrator is an integrator whose solution resides on a symplectic manifold. Because of discretization error, when it is solving a Hamiltonian system it doesn't get exactly the correct trajectory on the manifold. Instead, that trajectory itself is perturbed $\mathcal{O}(\Delta t^n)$ for the order n from the true trajectory. Then there's a linear drift due to numerical error of this trajectory over time. Normal integrators tend to have a quadratic (or more) drift, and do not have any good global guarantees about this phase space path (just local).

What this tends to mean is that symplectic integrators tend to capture the long-time patterns better than normal integrators because of this lack of drift and this almost guarantee of periodicity. [This notebook displays those properties well on the Kepler problem.](#) The first image shows what I'm talking about with the periodic nature of the solution.

Kepler Problem Solution

First Integrals

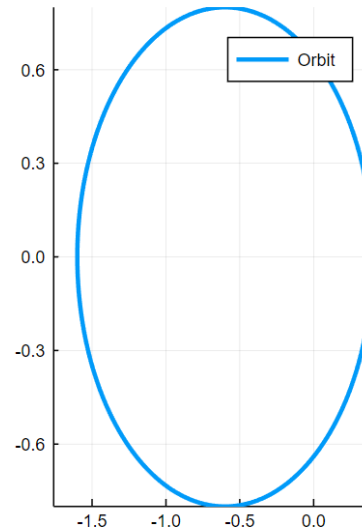
There are lots of special properties

Some problems need special integrators

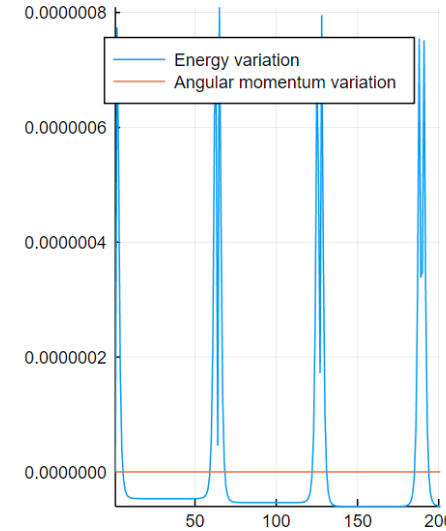
- Symplectic integrators for long time Hamiltonian systems
- Magnus for $u' = A(t)*u$
- Munthe-Kaas methods for $u' = A(u)*u$
- Nystrom specializations for 2nd order
- Exponential integrators for semilinear ODEs $u' = Au + f(u)$
- Implicit-Explicit (IMEX) methods for partly stiff equations
- Runge-Kutta-Chebyshev methods for semi-stiff equations

...

Kepler Problem Solution

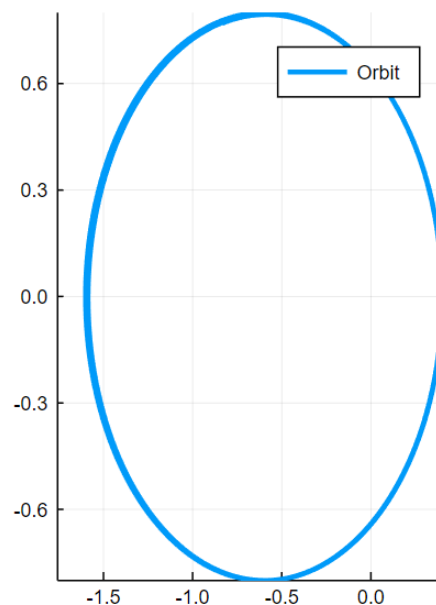


First Integrals

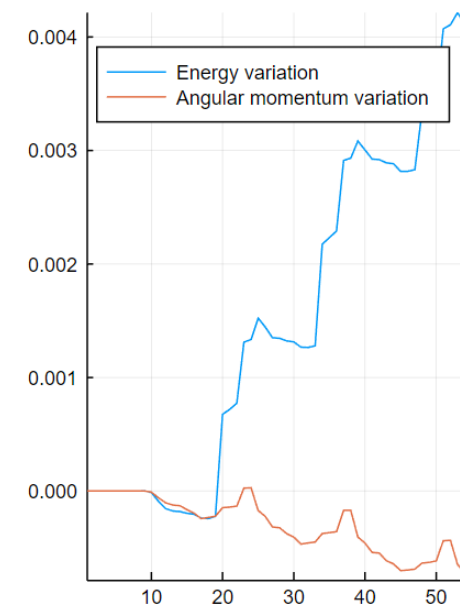


Symplectic

Kepler Problem Solution



First Integrals



Standard Integrator (Tsit5)

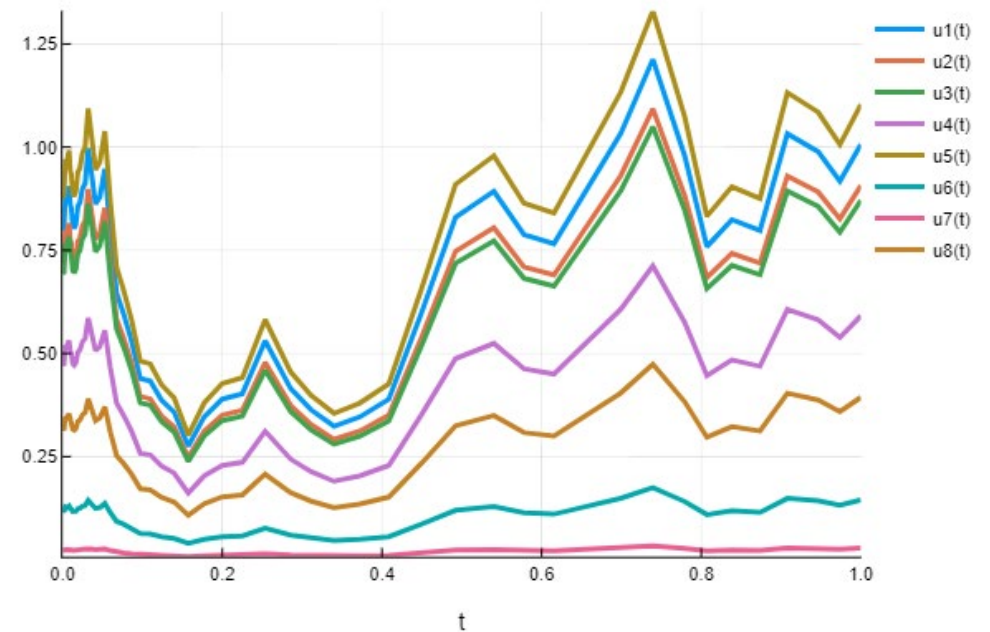
Julia's DifferentialEquations.jl is at over 300 integrators, and there's still more to do.

But that's just adding more solvers, right?

Differential Equations Go Beyond ODEs

- Discrete equations (function maps, discrete stochastic (Gillespie/Markov) simulations)
- Ordinary differential equations (ODEs)
- Split and Partitioned ODEs (Symplectic integrators, IMEX Methods)
- Stochastic ordinary differential equations (SODEs or SDEs)
- Stochastic differential-algebraic equations (SDAEs)
- Random differential equations (RODEs or RDEs)
- Differential algebraic equations (DAEs)
- Delay differential equations (DDEs)
- Neutral, retarded, and algebraic delay differential equations (NDDEs, RDDEs, and DDAEs)
- Stochastic delay differential equations (SDDEs)
- Experimental support for stochastic neutral, retarded, and algebraic delay differential equations (SNDDEs, SRDDEs, and SDDAEs)
- Mixed discrete and continuous equations (Hybrid Equations, Jump Diffusions)
- (Stochastic) partial differential equations ((S)PDEs) (with both finite difference and finite element methods)

...



**But if you keep adding
solver choices,
then you're okay?**

Equations that Cannot be Solved: DAE Index Reduction

Eye-balling Index Issues: any algebraic equation should be dependent on some algebraic variable

$$\begin{aligned}x' &= v_x \\ v'_x &= Tx\end{aligned}$$

Not solvable by standard numerical solvers!

$$\begin{aligned}y' &= v_y \\ v'_y &= Ty - g \\ 0 &= x^2 + y^2 - L^2\end{aligned}$$



Differentiate the last equation twice, do a few substitutions...

$$\begin{aligned}x' &= v_x \\ v'_x &= xT\end{aligned}$$

Easy to solve!

$$\begin{aligned}y' &= v_y \\ v'_y &= yT - g \\ 0 &= 2(v_x^2 + v_y^2 + y(yT - g) + Tx^2)\end{aligned}$$

States: x , v_x , y , v_y , and T . **Algebraic equation states:** x and y (no T).

If you don't know the details about why this makes a better numerical simulation, then you should be using ModelingToolkit.jl

DAE Index Reduction is Automatic with ModelingToolkit.jl

```
using DifferentialEquations, ModelingToolkit
using LinearAlgebra, Plots

function pendulum!(du, u, p, t)
    x, dx, y, dy, T = u
    g, L = p
    du[1] = dx
    du[2] = T*x
    du[3] = dy
    du[4] = T*y - g
    du[5] = x^2 + y^2 - L^2
    return nothing
end

pendulum_fun! = ODEFunction(pendulum!,
                             mass_matrix=Diagonal([1,1,1,1,0]))

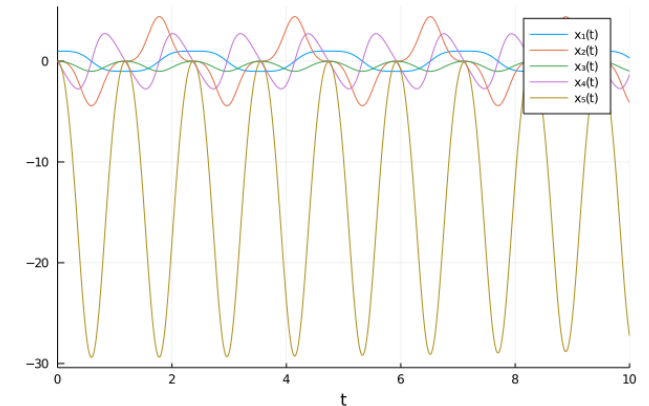
u0 = [1.0, 0, 0, 0, 0]
p = [9.8, 1]
tspan = (0, 10.0)
pendulum_prob = ODEProblem(pendulum_fun!, u0, tspan, p)
sol = solve(pendulum_prob)
```

```
[ Warning: dt <= dtmin. Aborting. There is either an error in your model specification or the true solution is unstable.
@ SciMLBase C:\Users\accou\.julia\packages\SciMLBase\A0oIW\src\integrator_interface.jl:345
```

Let me fix that for you...

```
@named traced_sys = modelingtoolkitize(pendulum_prob)
pendulum_sys = structural_simplify(traced_sys)
prob = ODAEProblem(pendulum_sys, [], tspan)
sol = solve(prob, Tsit5(), abstol=1e-8, reltol=1e-8)
plot(sol, vars=states(traced_sys))
```

structural_simplify:
The God of Transforms



Standard equation solvers (IDA, DASKR, all of the Julia DAE solvers, etc.) cannot solve even a Cartesian pendulum without symbolic modification!

And when you finally get there, you have to keep customizing

```
using DifferentialEquations, LinearAlgebra, SparseArrays

const N = 32
const xyd_brusselator = range(0, stop=1, length=N)
brusselator_f(x, y, t) = (((x-0.3)^2 + (y-0.6)^2) <= 0.1^2) * (t >= 1.1) * 5.
limit(a, N) = a == N+1 ? 1 : a == 0 ? N : a
function brusselator_2d_loop(du, u, p, t)
    A, B, alpha, dx = p
    alpha = alpha/dx^2
    @inbounds for I in CartesianIndices((N, N))
        i, j = Tuple(I)
        x, y = xyd_brusselator[I[1]], xyd_brusselator[I[2]]
        ip1, im1, jp1, jm1 = limit(i+1, N), limit(i-1, N), limit(j+1, N), limit(j-1, N)
        du[i,j,1] = alpha*(u[im1,j,1] + u[ip1,j,1] + u[i,jp1,1] + u[i,jm1,1] - 4u[i,j,1]) +
            B + u[i,j,1]^2*u[i,j,2] - (A + 1)*u[i,j,1] + brusselator_f(x, y, t)
        du[i,j,2] = alpha*(u[im1,j,2] + u[ip1,j,2] + u[i,jp1,2] + u[i,jm1,2] - 4u[i,j,2]) +
            A*u[i,j,1] - u[i,j,1]^2*u[i,j,2]
    end
end
p = (3.4, 1., 10., step(xyd_brusselator))

function init_brusselator_2d(xyd)
    N = length(xyd)
    u = zeros(N, N, 2)
    for I in CartesianIndices((N, N))
        x = xyd[I[1]]
        y = xyd[I[2]]
        u[I,1] = 22*(y*(1-y))^(3/2)
        u[I,2] = 27*(x*(1-x))^(3/2)
    end
end
u0 = init_brusselator_2d(xyd_brusselator)
prob_ode_brusselator_2d = ODEProblem(brusselator_2d_loop, u0, (0., 11.5), p)
```

Some giant ODE



https://diffeq.sciml.ai/stable/tutorials/advanced_ode_example/

DifferentialEquations.jl uses LinearSolve.jl internally, so all options are available!

And when you finally get there, you have to keep customizing

```
using BenchmarkTools # for @btime
@btime solve(prob_ode_brusselator_2d_sparse, TRBDF2(), save_everystep=false)
# 680.612 ms (37905 allocations: 359.34 MiB)
```

And when you finally get there, you have to keep customizing

```
using BenchmarkTools # for @btime
@btime solve(prob_ode_brusselator_2d_sparse, TRBDF2(), save_everystep=false)
# 680.612 ms (37905 allocations: 359.34 MiB)
```

Baseline with sparse Jacobian, pretty decent

```
@btime solve(prob_ode_brusselator_2d_sparse, KenCarp47(linsolve=KLUFactorization()), save_everystep=false)
# 342.017 ms (65150 allocations: 158.99 MiB)
@btime solve(prob_ode_brusselator_2d, KenCarp47(linsolve=KrylovJL_GMRES()), save_everystep=false)
# 707.439 ms (173868 allocations: 31.07 MiB)
```

**Just pass different LinearSolve.jl algorithms to try different internal solvers.
Non-trivial differences!**

https://diffeq.sciml.ai/stable/tutorials/advanced_ode_example/

DifferentialEquations.jl uses LinearSolve.jl internally, so all options are available!

And when you finally get there, you have to keep customizing

```
using BenchmarkTools # for @btime
@btime solve(prob_ode_brusselator_2d_sparse, TRBDF2(), save_everystep=false)
# 680.612 ms (37905 allocations: 359.34 MiB)
```

```
using IncompleteLU
function incompletelu(W, du, u, p, t, newW, Plprev, Prprev, solverdata)
    if newW === nothing || newW
        Pl = ilu(convert(AbstractMatrix, W), τ=50.0)
    else
        Pl = Plprev
    end
    Pl, nothing
end

@time solve(prob_ode_brusselator_2d_sparse, KenCarp47(linsolve=KrylovJL_GMRES(),
    prec=incompletelu, concrete_jac=true), save_everystep=false);
# 174.386 ms (61756 allocations: 61.38 MiB)
```

Use the preconditioner interface for iLU with GMRES, chunked it down a few notches

https://diffeq.sciml.ai/stable/tutorials/advanced_ode_example/

DifferentialEquations.jl uses LinearSolve.jl internally, so all options are available!

What About Partial Differential Equations (and Beyond?)

```
using ModelingToolkit
import ModelingToolkit: Interval, infimum, supremum

@parameters x y
@variables u(..)
Dxx = Differential(x)^2
Dyy = Differential(y)^2

# 2D PDE
eq = Dxx(u(x,y)) + Dyy(u(x,y)) ~ -sin(pi*x)*sin(pi*y)

# Boundary conditions
bcs = [u(0,y) ~ 0.f0, u(1,y) ~ -sin(pi*1)*sin(pi*y),
       u(x,0) ~ 0.f0, u(x,1) ~ -sin(pi*x)*sin(pi*1)]

# Space and time domains
domains = [x ∈ Interval(0.0,1.0),
           y ∈ Interval(0.0,1.0)]
pde_system = PDESystem(eq,bcs,domains,[x,y],[u])
```

PDEs need lots of discretizers

- Physics-Informed NNs: NeuralPDE.jl
- Finite Difference: MethodOfLines.jl
- Neural Operators: NeuralOperators.jl
- Finite Volume: Trixi.jl
- Finite Element: Gridap.jl
- Pseudospectral: ApproxFun.jl
- High Dimension: HighDimPDE.jl

Etc. a bunch more issues to address...

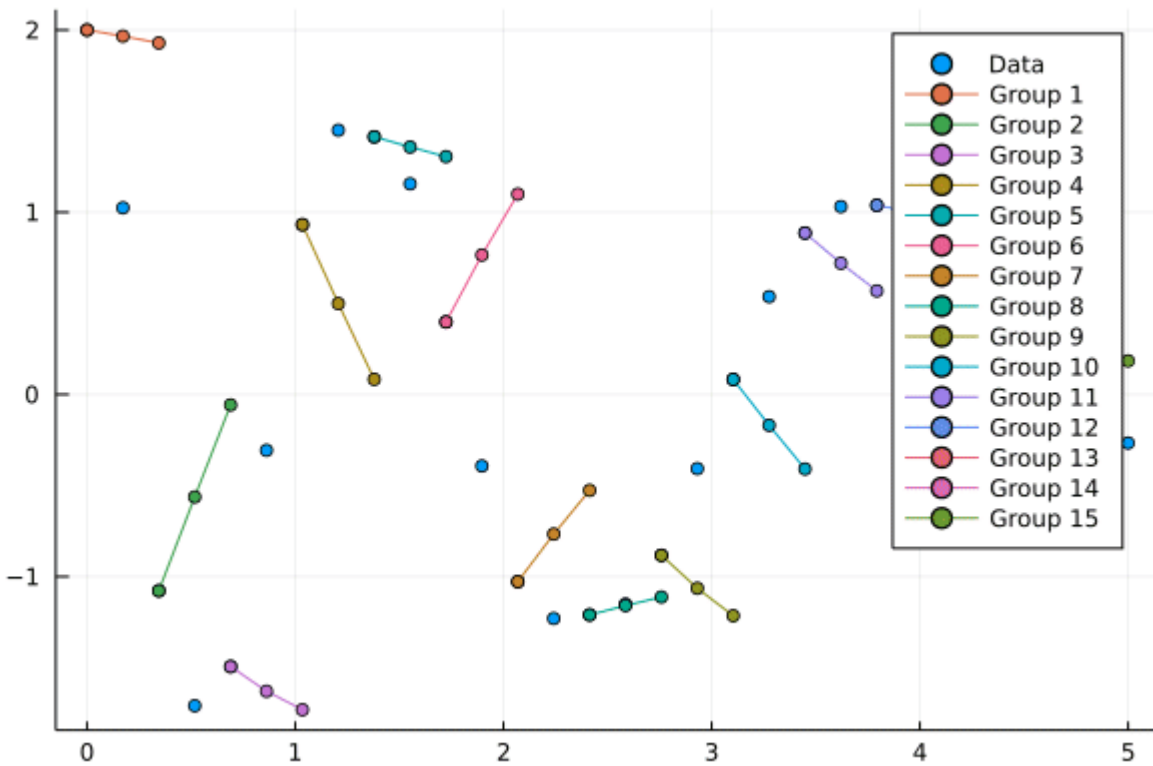
Okay, now I'm just ranting.

But the point is, equation solving is a huge topic. As big as probabilistic programming.

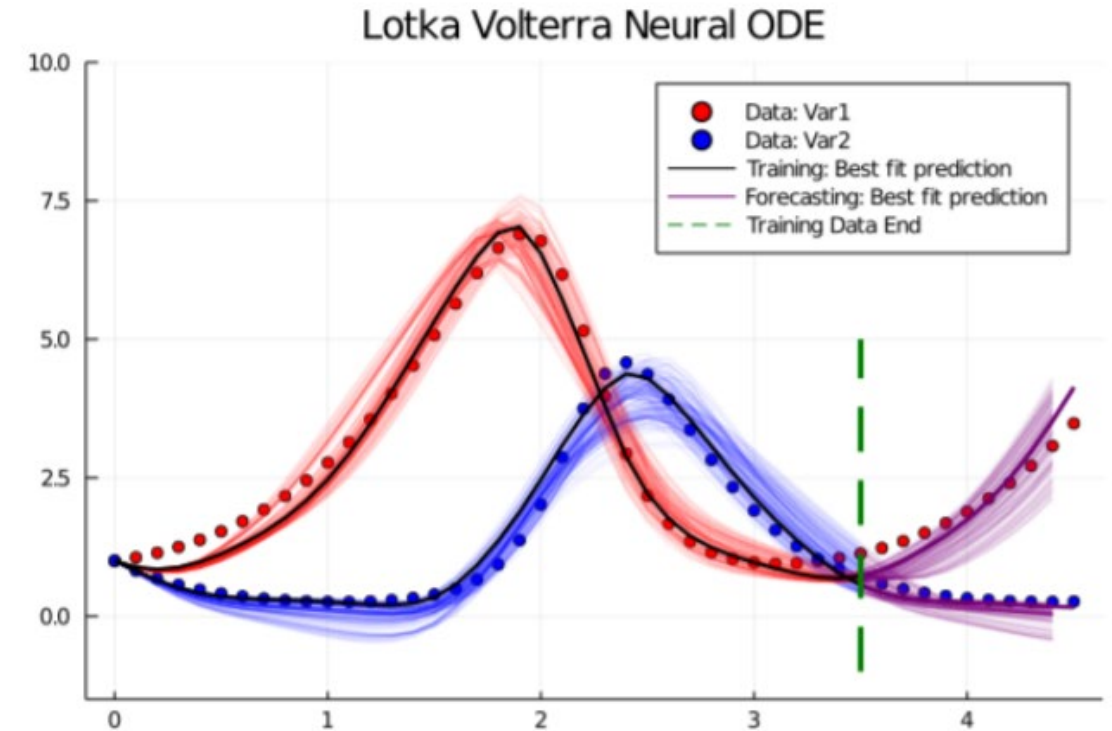
So how do you integrate that with PPLs?

Outline

Mixing equation discovery into epidemic modeling workflows will revolutionize the field



1. What could “extensive” differential equation support even mean?



2. Does Turing.jl have it? And what does that mean about its developer community?

Stan(dard) PPL DSLs: The Top-Down Approach

Stan User's Guide

Version 2.30

Stan Development Team

13. Ordinary Differential Equations 186

- 13.1 Notation 188
- 13.2 Example: simple harmonic oscillator 188
- 13.3 Coding the ODE system function 188
- 13.4 Measurement error models 190
- 13.5 Stiff ODEs 194
- 13.6 Control parameters for ODE solving 194
- 13.7 Adjoint ODE solver 197
- 13.8 Solving a system of linear ODEs using a matrix exponential 198

16. Differential-Algebraic Equations 210

- 16.1 Notation 210
- 16.2 Example: chemical kinetics 211
- 16.3 Index of DAEs 211
- 16.4 Coding the DAE system function 211
- 16.5 Solving DAEs 213
- 16.6 Control parameters for DAE solving 214

Stan(dard) PPL DSLs: The Top-Down Approach

- `rk45`: a fourth and fifth order Runge-Kutta method for non-stiff systems (Dormand and Prince 1980; Ahnert and Mulansky 2011). `rk45` is the most generic solver and should be tried first.
- `bdf`: a variable-step, variable-order, backward-differentiation formula implementation for stiff systems (Cohen and Hindmarsh 1996; Serban and Hindmarsh 2005). `bdf` is often useful for ODEs modeling chemical reactions.
- `adams`: a variable-step, variable-order, Adams-Moulton formula implementation for non-stiff systems (Cohen and Hindmarsh 1996; Serban and Hindmarsh 2005). The method has order up to 12, hence is commonly used when high-accuracy is desired for a very smooth solution, such as in modeling celestial mechanics and orbital dynamics (Montenbruck and Gill 2000).
- `ckrk`: a fourth and fifth order explicit Runge-Kutta method for non-stiff and semi-stiff systems (Cash and Karp 1990; Mazzia, Cash, and Soetaert 2012).

Top-Down => Limited Developer Support => Our Issues Are Here

16.3. Index of DAEs

The index along a DAE solution $y(t)$ is the minimum number of differentiations of some of the components of the system required to solve for y' uniquely in terms of y and t , so that the DAE is converted into an ODE for y . Thus an ODE system is of index 0. The above chemical kinetics DAE is of index 1, as we can perform differentiation of the third equation followed by introducing the first two equations in order to obtain the ODE for y_3 .

Most DAE solvers, including the one in Stan, support only index-1 DAEs. For a high index DAE problem the user must first convert it to a lower index system. This often can be done by carrying out differentiations analytically (Ascher and Petzold 1998).

No DAE automation

No SDEs, DDEs, PDEs, ...

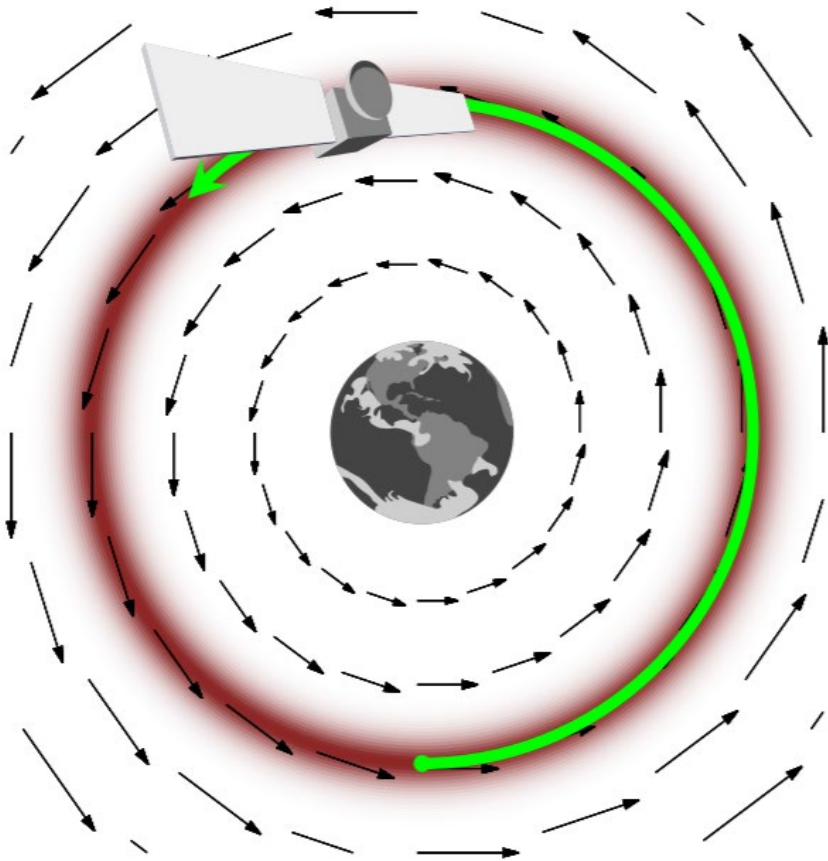
Symplectic, Munthe-Kaas, ...

No KLU, iLU preconditioning, algebraic multigrid...

Okay, I can poke fun at Stan.

But the “why” is more important.

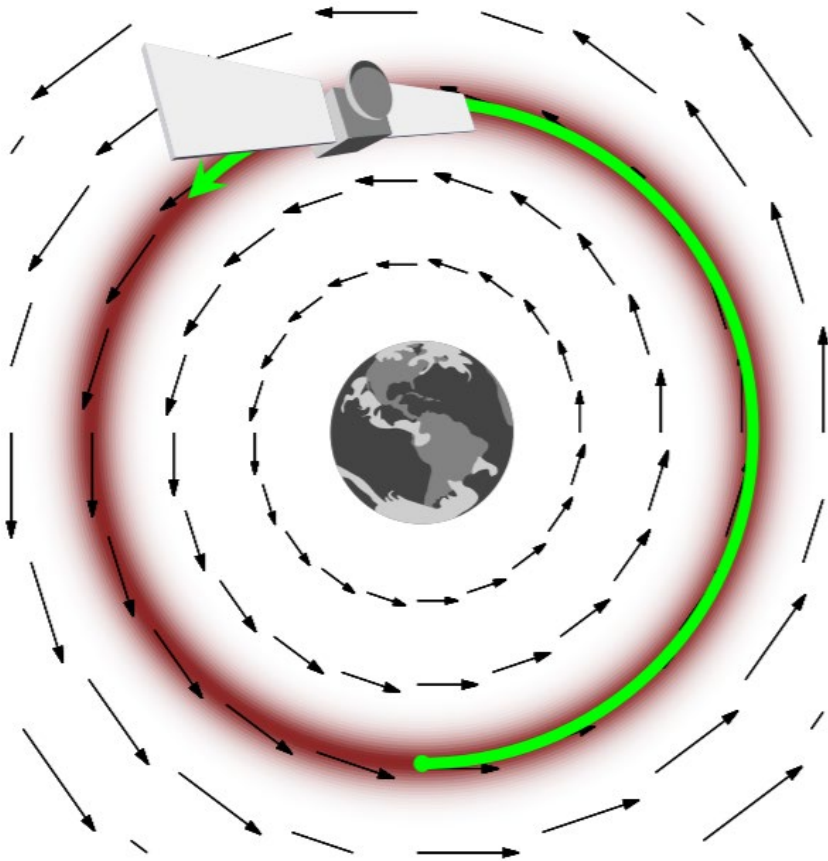
PPLs Need Derivatives



$$\begin{aligned}\frac{dx}{dt} &= \frac{dH}{dp} \\ &= -\frac{dH}{dx}\end{aligned}$$

Good PPL methods (Hamiltonian Monte Carlo, ADVI, etc.) Requires Good Derivatives of Every Operation

PPLs Need Derivatives

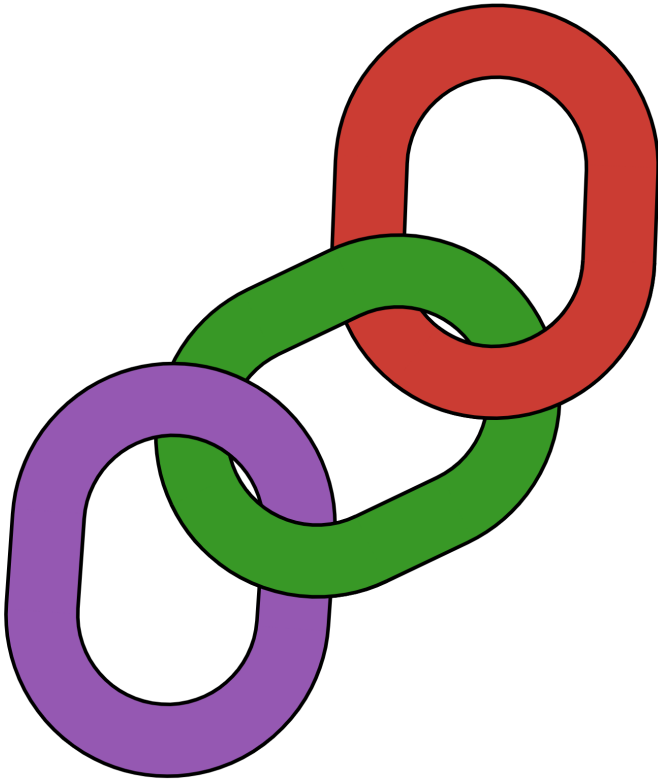


$$\begin{aligned}\frac{dx}{dt} &= \frac{dH}{dp} \\ &= -\frac{dH}{dx}\end{aligned}$$

You cannot just stick an ODE solver into a PPL and expect it to work!

Good PPL methods (Hamiltonian Monte Carlo, ADVI, etc.) Requires Good Derivatives of Every Operation

Julia's Pervasive Differentiable Programming



ChainRules.jl

Julia has a pervasive language-wide system for differentiable programming

No DSL required: directly support Julia code in any code that requires differentiation!

Just do it!

```
@model function fitlv(data, prob)
  # Prior distributions.
   $\sigma \sim \text{InverseGamma}(2, 3)$ 
   $\alpha \sim \text{truncated}(\text{Normal}(1.5, 0.5), 0.5, 2.5)$ 
   $\beta \sim \text{truncated}(\text{Normal}(1.2, 0.5), 0, 2)$ 
   $\gamma \sim \text{truncated}(\text{Normal}(3.0, 0.5), 1, 4)$ 
   $\delta \sim \text{truncated}(\text{Normal}(1.0, 0.5), 0, 2)$ 

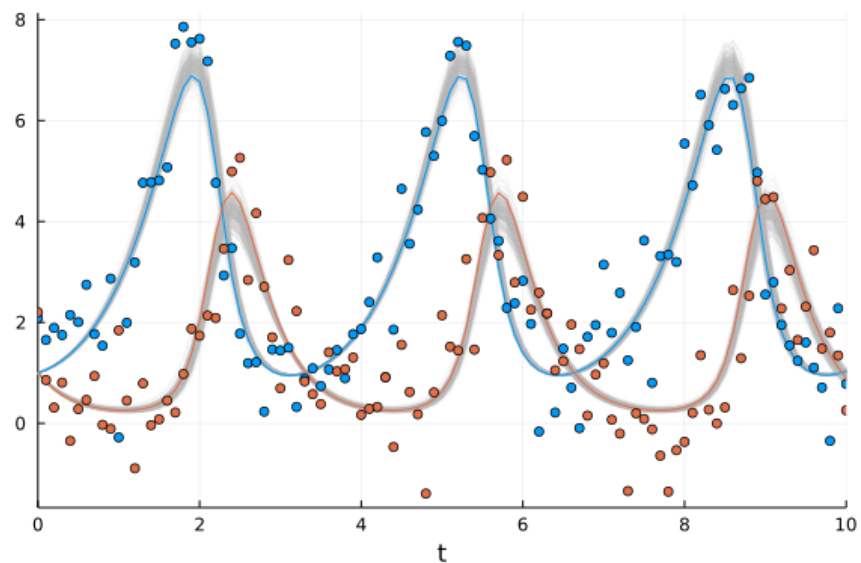
  # Simulate Lotka-Volterra model.
  p = [ $\alpha$ ,  $\beta$ ,  $\gamma$ ,  $\delta$ ]
  predicted = solve(prob, Tsit5(); p=p, saveat=0.1)

  # Observations.
  for i in 1:length(predicted)
    data[:, i]  $\sim \text{MvNormal}(\text{predicted}[i], \sigma^2 * \text{I})$ 
  end

  return nothing
end

model = fitlv(odedata, prob)
```

Turing.jl + DifferentialEquations.jl:
Just use the ODE solver inside of
Turing.



```
# Sample 3 independent chains with forward-mode automatic differentiation (the default)
chain = sample(model, NUTS(0.65), MCMCSerial(), 1000, 3; progress=false)
```


Improving Coverage of Automatic Differentiation over Solvers

LinearSolve.jl: Unified Linear Solver Interface

$$A(p)x = b$$

NonlinearSolve.jl: Unified Nonlinear Solver Interface

$$f(u, p) = 0$$

DifferentialEquations.jl: Unified Interface for all
Differential Equations

$$\begin{aligned} u' &= f(u, p, t) \\ du &= f(u, p, t)dt + g(u, p, t)dW_t \\ &\vdots \end{aligned}$$

Optimization.jl: Unified Optimization Interface

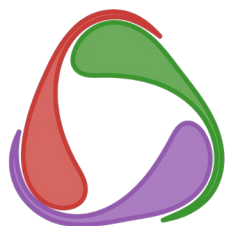
$$\begin{aligned} &\text{minimize } f(u, p) \\ &\text{subject to } g(u, p) \leq 0, h(u, p) = 0 \end{aligned}$$

Integrals.jl: Unified Quadrature Interface

$$\int_{lb}^{ub} f(t, p)dt$$

Unified Partial Differential Equation Interface

$$\begin{aligned} u_t &= u_{xx} + f(u) \\ u_{tt} &= u_{xx} + f(u) \\ &\vdots \end{aligned}$$

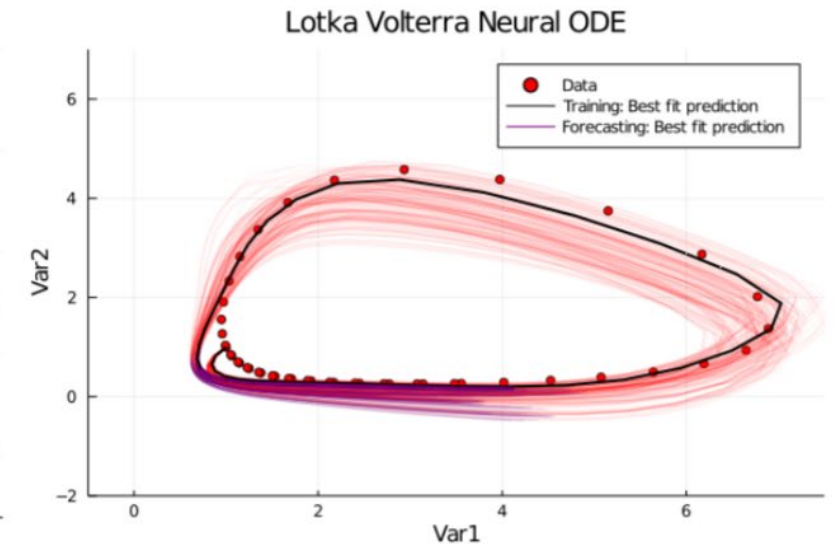
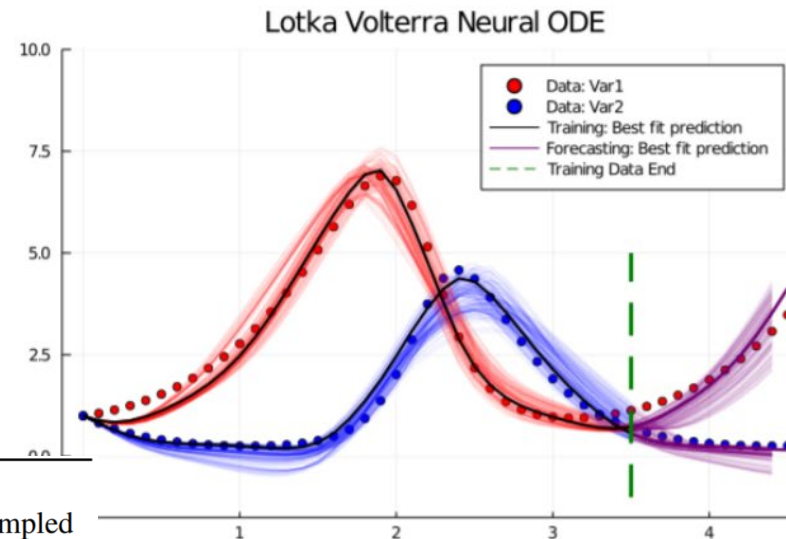


The SciML Common Interface for Julia Equation Solvers

<https://scimlbase.sciml.ai/dev/>

Bayesian UODEs: Knowledge-Enhanced Model Discovery with UQ

Probabilistic Model Discovery with Turing.jl



```
function lotka_volterra!(du, u, p, t)
    x, y = u
    α, β, δ, γ = p
    du[1] = dx = α*x - β*x*y
    du[2] = dy = -δ*y + γ*x*y
end
```

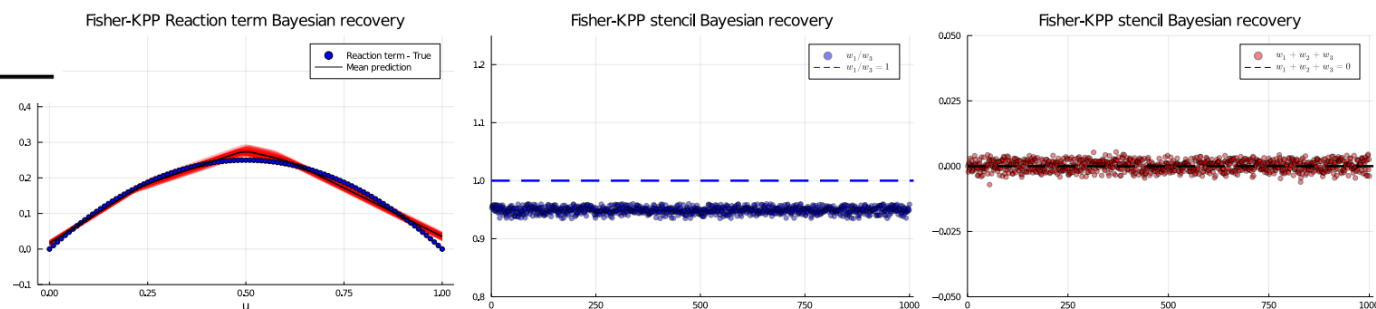
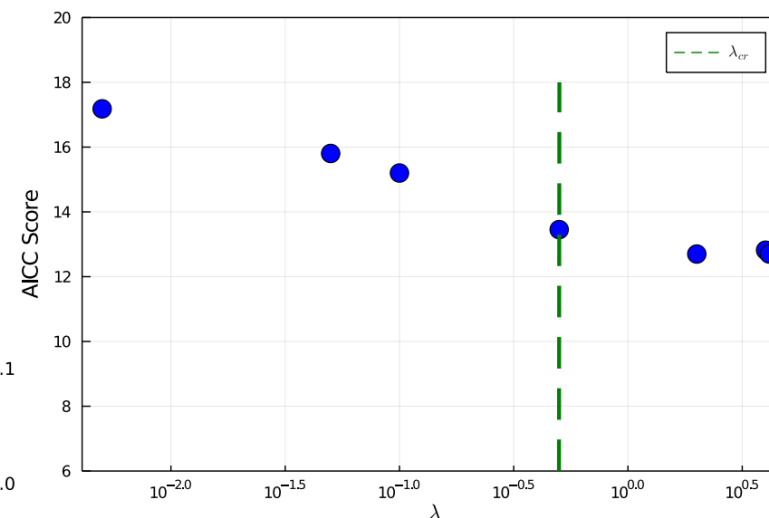
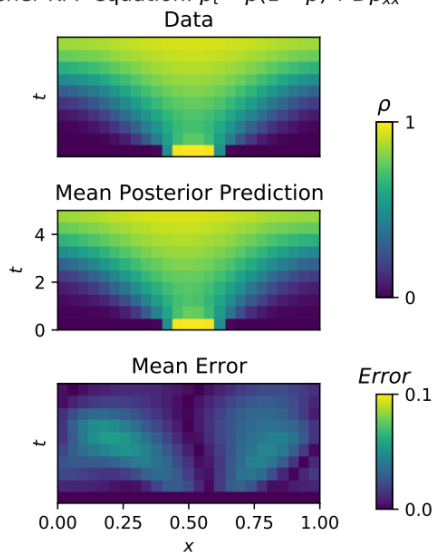


Bayesian UODEs: Knowledge-Enhanced Model Discovery with UQ

Probabilistic PDE Discovery with Turing.jl

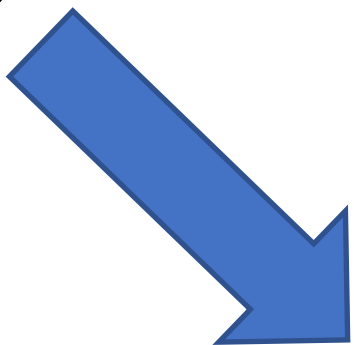
λ_{cr}	Number of Active terms	Dominant terms	% of samples
0.5	2	ρ, ρ^2	73
0.5	3	ρ, ρ^2, ρ^3	27

Fisher KPP equation: $\rho_t = \rho(1 - \rho) + D\rho_{xx}$



Downside: Documentation

The only mention of differential equations is in a tutorial! No API docs!



USING TURING ~

Getting Started

Quick Start

Guide

Advanced Usage

Automatic Differentiation

Performance Tips

Using DynamicHMC

Sampler Visualization

FOR DEVELOPERS ~

TUTORIALS ~

Home

Introduction to Turing

Gaussian Mixture Models

Bayesian Logistic Regression

Bayesian Neural Networks

Hidden Markov Models

Linear Regression

Infinite Mixture Models

Bayesian Poisson Regression

Multinomial Logistic Regression

Variational Inference

Bayesian Differential Equations

Probabilistic PCA

Gaussian Processes

API ~

Turing Documentation

Welcome to the documentation for Turing.

Introduction

Turing is a general-purpose probabilistic programming language for robust, efficient Bayesian inference and decision making. Current features include:

- **General-purpose** probabilistic programming with an intuitive modelling interface;
- Robust, efficient [Hamiltonian Monte Carlo \(HMC\)](#) sampling for differentiable posterior distributions;
- **Particle MCMC** sampling for complex posterior distributions involving discrete variables and stochastic control flow; and
- Compositional inference via **Gibbs sampling** that combines particle MCMC, HMC and random-walk MH (RWMH).

Upside: Lots of Features!

```
@model function fitlv_dde(data, prob)
  # Prior distributions.
   $\sigma$  ~ InverseGamma(2, 3)
   $\alpha$  ~ Truncated(Normal(1.5, 0.5), 0.5, 2.5)
   $\beta$  ~ Truncated(Normal(1.2, 0.5), 0, 2)
   $\gamma$  ~ Truncated(Normal(3.0, 0.5), 1, 4)
   $\delta$  ~ Truncated(Normal(1.0, 0.5), 0, 2)

  # Simulate Lotka-Volterra model.
  p = [ $\alpha$ ,  $\beta$ ,  $\gamma$ ,  $\delta$ ]
  predicted = solve(prob, MethodOfSteps(Tsit5())); p=p, saveat=0.1)

  # Observations.
  for i in 1:length(predicted)
    data[:, i] ~ MvNormal(predicted[i],  $\sigma^2$  * I)
  end
end

model_dde = fitlv_dde(ddedata, prob_dde)

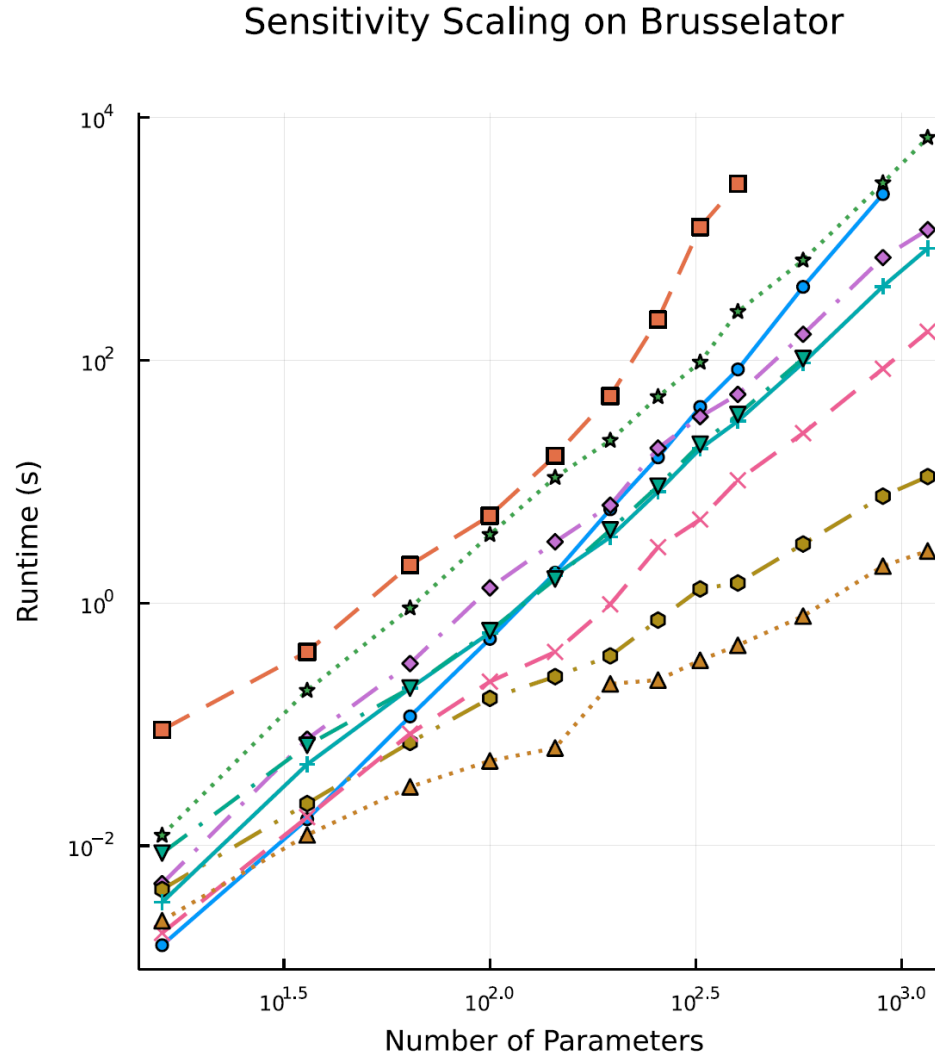
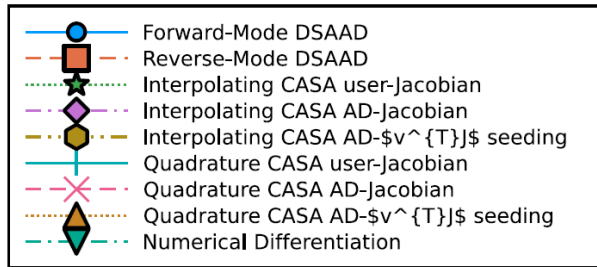
# Sample 3 independent chains.
chain_dde = sample(model_dde, NUTS(0.65), MCMCSerial(), 300, 3; progress=false)
```

Also includes stochastic differential equations, delay differential equations, etc. all just from slapping DifferentialEquations.jl inside!

How the adjoint is calculated also matters!

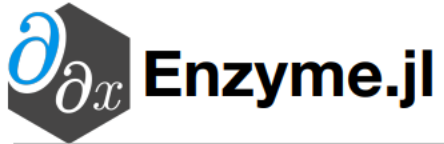
Gradient calculations on a stiff PDE, varying Δt

Rackauckas, Christopher, et al. "A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions." *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 1-8.



For more details on the performance of the adjoint methods, see *Accurate and Efficient Physics-Informed Learning Through Differentiable Simulation*

Enzyme is fast!!!



- Static analysis & optimization => very, very fast scalar AD

```
function taylor(x, N)
    sum = 0 * x
    for i = 1:N
        sum += x^i / i
    end
    return sum
end
```

```
def taylor_jax(x, N):
    sum = 0 * x
    for i in range(1,N):
        sum += x**i / i
    return sum
```

```
def taylor_lax(x, N):
    return jax.lax.fori_loop(
        1,
        N,
        lambda i, cur:
            cur + x**i / i,
        0)
```

```
@btime Enzyme.autodiff(Forward, taylor, Duplicated(0.5, 1.0), 10^6)
#      30 ms (0 bytes)

@btime Enzyme.autodiff(Reverse, taylor, Active(0.5), 10^6).
#      30 ms (0 bytes)

@btime ForwardDiff.derivative(x -> taylor(x, 10^6), 0.5)
#      60 ms (0 bytes)

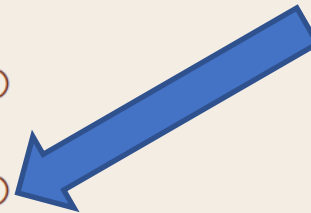
@btime Zygote.gradient(taylor, 0.5, 10^6)
#     993 ms (663.56 MiB)

@btime Difffractor.gradient(taylor, 0.5, 10^6)
#   96665 ms (96.37 GiB)

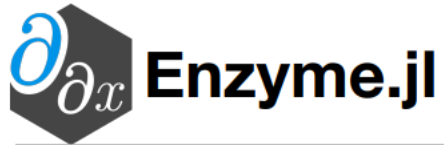
@pytime jax.grad(taylor_jax)(0.5, 10^5)
# >183993 ms

@pytime jax.grad(taylor_lax)(0.5, 10^6)
#      95 ms
```

Is Jax close enough?



Enzyme is fast!!!



- Static analysis & optimization => very, very fast scalar AD

```
function taylor(x, N)
    sum = 0 * x
    for i = 1:N
        sum += x^i / i
    end
    return sum
end
```

```
def taylor_jax(x, N):
    sum = 0 * x
    for i in range(1,N):
        sum += x**i / i
    return sum
```

```
def taylor_lax(x, N):
    return jax.lax.fori_loop(
        1,
        N,
        lambda i, cur:
            cur + x**i / i,
        0)
```

```
@btime Enzyme.autodiff(Forward, taylor, Duplicated(0.5, 1.0), 10^6)
#      30 ms (0 bytes)

@btime Enzyme.autodiff(Reverse, taylor, Active(0.5), 10^6).
#      30 ms (0 bytes)

@btime ForwardDiff.derivative(x -> taylor(x, 10^6), 0.5)
#      60 ms (0 bytes)

@btime Zygote.gradient(taylor, 0.5, 10^6)
#     993 ms (663.56 MiB)

@btime Difffractor.gradient(taylor, 0.5, 10^6)
#   96665 ms (96.37 GiB)

@pytime jax.grad(taylor_jax)(0.5, 10^5)
# >183993 ms

@pytime jax.grad(taylor_lax)(0.5, 10^6)
#      95 ms
```

construct	jit	grad
if	✗	✓
for	✓*	✓
while	✓*	✓
lax.cond	✓	✓
lax.while_loop	✓	fwd
lax.fori_loop	✓	fwd
lax.scan	✓	✓

If forward mode only.

Enzyme is fast!!!

Differentiating after optimization can create *asymptotically faster* gradients!

$O(n^2)$

```
for i = 1:n  
    out[i] /= mag(in)  
end
```

Optimize

$O(n)$

```
res = mag(in)  
for i = 1:n  
    out[i] /= res  
end
```

AD

$O(n)$

```
d_res = 0.0  
for i = n:1  
    d_res += d_out[i]...  
end  
∇mag(d_in, d_res)
```

$O(n^2)$

```
for i = 1:n  
    out[i] /= mag(in)  
end
```

AD

$O(n^2)$

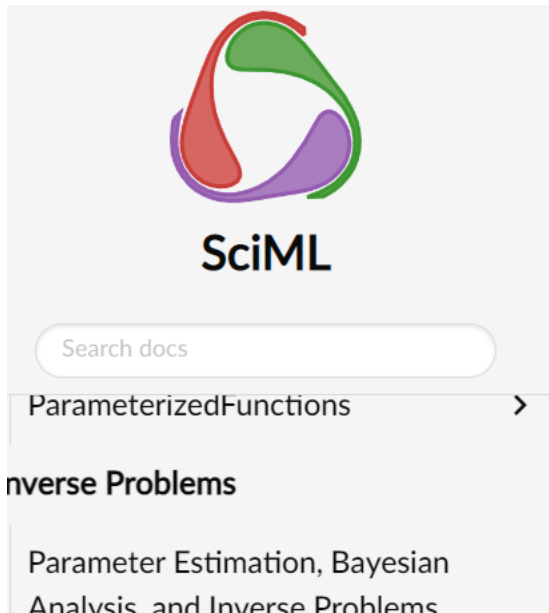
```
for i = n:1  
    d_res = d_out[i]...  
    ∇mag(d_in, d_res)  
end
```

Optimize

$O(n^2)$

```
for i = n:1  
    d_res = d_out[i]...  
    ∇mag(d_in, d_res)  
end
```

Downside: Documentation



[Inverse Problems](#) / [SciMLSensitivity](#) / [Tutorials](#) / [Bayesian Estimation Tutorials](#)
/ [Bayesian Estimation of Differential Equations with Probabilistic Programming](#)

[Edit on GitHub](#) 

Bayesian Estimation of Differential Equations with Probabilistic Programming

For a good overview of how to use the tools of SciML in conjunction with the Turing.jl probabilistic programming language, see the [Bayesian Differential Equation Tutorial](#).

« [Bouncing Ball Hybrid ODE Optimization](#) [Solving Optimal Control Problems with Universal Differential Equations](#) »

Powered by [Documenter.jl](#) and the [Julia Programming Language](#).

**If the composition of two
packages automatically
constructs the functionality,
who documents it?**

Pervasive Differentiable Programming greatly enlarges the developer pool and accelerates development.

But relying on package composability creates a documentation and “ownership” problem.

We are looking for nice solutions to the latter issues with our automatically constructed feature sets.

SciML Open Source Software Organization sciml.ai

- DifferentialEquations.jl: 2x-10x Sundials, Hairer, ...
- DiffEqFlux.jl: adjoints outperforming Sundials and PETSc-TS
- ModelingToolkit.jl: 15,000x Simulink
- Catalyst.jl: >100x SimBiology, gillespy, Copasi
- DataDrivenDiffEq.jl: >10x pySindy
- NeuralPDE.jl: ~2x DeepXDE* (more optimizations to be done)
- NeuralOperators.jl: ~3x original papers (more optimizations required)
- ReservoirComputing.jl: 2x-10x pytorch-esn, ReservoirPy, PyRCN
- SimpleChains.jl: 5x PyTorch GPU with CPU, 10x Jax (small only!)
- DiffEqGPU.jl: Some wild GPU ODE solve speedups coming soon

And 100 more libraries to mention...

If you work in SciML and think optimized and maintained implementations of your method would be valuable, please let us know and we can add it to the queue.

**Democratizing SciML via pedantic code optimization
Because we believe full-scale open benchmarks matter**



There are many different ways, all with engineering trade-offs

Method	Stability	Stiff Performance Scaling	Memory Usage
BacksolveAdjoint	Poor	$O((s + p)^3)$	Low. $O(1)$
InterpolatingAdjoint	Good	$O((s + p)^3)$	High. Requires full continuous solution of forward
QuadratureAdjoint	Good	$O(s^3 + p)$	Higher. Requires full continuous solution of forward and Lagrange multiplier
BacksolveAdjoint (Checkpointed)	Okay	$O((s + p)^3) + C$	Medium. $O(c)$ where c is the number of checkpoints
InterpolatingAdjoint (Checkpointed)	Good	$O((s + p)^3) + C$	Medium. $O(c)$ where c is the number of checkpoints
ReverseDiffAdjoint	Best	$O(s^3 + p) + C$	Highest. Requires full forward and reverse AD of solve
TrackerAdjoint	Best	$O(s^3 + p) + C$	Highest. Requires full forward and reverse AD of solve
ForwardLSS/AdjointLSS/N ILSS	Chaos	Not even comparable: expensive.	Super duper high OMG.

Differentiating Ordinary Differential Equations: The Trick

We wish to solve for some cost function $G(u, p)$ evaluated throughout the differential equation, i.e.:

$$G(u, p) = G(u(p)) = \int_{t_0}^T g(u(t, p)) dt$$

To derive this adjoint, introduce the Lagrange multiplier λ to form:

$$I(p) = G(p) - \int_{t_0}^T \lambda^* (u' - f(u, p, t)) dt$$

Since $u' = f(u, p, t)$, this is the mathematician's trick of adding zero, so then we have that

$$s = \frac{du}{dp} \quad \frac{dG}{dp} = \frac{dI}{dp} = \int_{t_0}^T (g_p + g_u s) dt - \int_{t_0}^T \lambda^* (s' - f_u s - f_p) dt$$

Differentiating Ordinary Differential Equations: Integration By Parts

for s being the sensitivity, $s = \frac{du}{dp}$. After applying integration by parts to $\lambda^* s'$, we get that:

$$\begin{aligned}\int_{t_0}^T \lambda^* (s' - f_u s - f_p) dt &= \int_{t_0}^T \lambda^* s' dt - \int_{t_0}^T \lambda^* (f_u s - f_p) dt \\ &= |\lambda^*(t)s(t)|_{t_0}^T - \int_{t_0}^T \lambda^{*'} s dt - \int_{t_0}^T \lambda^* (f_u s - f_p) dt\end{aligned}$$

To see where we ended up, let's re-arrange the full expression now:

$$\begin{aligned}\frac{dG}{dp} &= \int_{t_0}^T (g_p + g_u s) dt + |\lambda^*(t)s(t)|_{t_0}^T - \int_{t_0}^T \lambda^{*'} s dt - \int_{t_0}^T \lambda^* (f_u s - f_p) dt \\ &= \int_{t_0}^T (g_p + \lambda^* f_p) dt + |\lambda^*(t)s(t)|_{t_0}^T - \int_{t_0}^T (\lambda^{*'} + \lambda^* f_u - g_u) s dt\end{aligned}$$

Differentiating Ordinary Differential Equations: The Final Form

$$\frac{dG}{dp} = \int_{t_0}^T (g_p + \lambda^* f_p) dt + [\lambda^*(t)s(t)]_{t_0}^T - \int_{t_0}^T (\lambda^{*'} + \lambda^* f_u - g_u) s dt$$

That was just a re-arrangement. Now, let's require that

$$\lambda' = -\frac{df^*}{du} \lambda - \left(\frac{dg}{du} \right)^*$$

$$\lambda(T) = 0$$

This means that the boundary term of the integration by parts is zero, and also one of those integral terms are perfectly zero. Thus, if λ satisfies that equation, then we get:

$$\frac{dG}{dp} = \lambda^*(t_0) \frac{dG}{du}(t_0) + \int_{t_0}^T (g_p + \lambda^* f_p) dt$$

Differentiating Ordinary Differential Equations: Summary

Summary:

1. Solve $u' = f(u, p, t)$

2. Solve $\lambda' = -\frac{df^*}{du} \lambda - \left(\frac{dg}{du}\right)^*$

$$\lambda(T) = 0$$

3. Solve $\frac{dG}{dp} = \lambda^*(t_0) \frac{dG}{du}(t_0) + \int_{t_0}^T (g_p + \lambda^* f_p) dt$

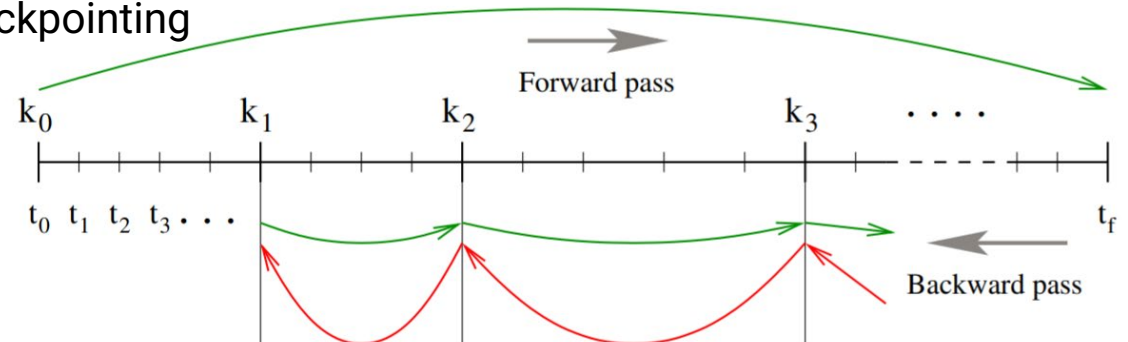
Differentiating Ordinary Differential Equations: Step 2 Details

2. Solve $\lambda'_{(t)} = -\frac{df^*}{du_{(t)}} \lambda^{(t)} - \left(\frac{dg}{du_{(t)}}\right)^*$

$\lambda(T) = 0$


How do you get $u(t)$ while solving backwards?
3 options!

1. $u' = f(z, t)$ forwards, then
 $u' = -f(z, -t)$ backwards!
2. Store $u(t)$ while solving forwards (dense output)
3. Checkpointing



How the gradient (adjoint) is calculated also matters!

This term is traditionally computed via differentiation and then multiplied to lambda
Reverse-mode embedded implementation: push-forward $f(u)$ pullback lambda
Computational cost $O(n) \rightarrow O(1)$ f evaluations and automatically uses optimized backpropagation!


$$M^* \lambda' = - \boxed{\frac{df^*}{du} \lambda} - \left(\frac{dg}{du} \right)^*$$
$$\lambda(T) = 0,$$


Adjoint Differential Equation

Six choices for this computation:

- Numerical
- Forward-mode
- Reverse-mode traced compiled graph (ReverseDiffVJP(true))
 - Fast method for scalarized nonlinear equations
 - Requires CPU and no branching (generally used in SciML)
- Reverse-mode static
 - Fastest method when applicable
- Reverse-mode traced
 - Fast but not GPU compatible
- Reverse-mode vector source-to-source
 - Best for embedded neural networks

Differentiating Ordinary Differential Equations: Step 3 Details

3. Solve $\frac{dG}{dp} = \lambda^*(t_0) \frac{dG}{du}(t_0) + \int_{t_0}^T (g_p + \lambda^*_{(t)} f_p) dt$



How do you calculate the integral?

1. Store $\lambda(t)$ while solving backwards (dense output)
2. $\mu' = -\lambda^* f_p + g_p$ where $\mu(T) = 0$

What's the trade-off between these ideas?

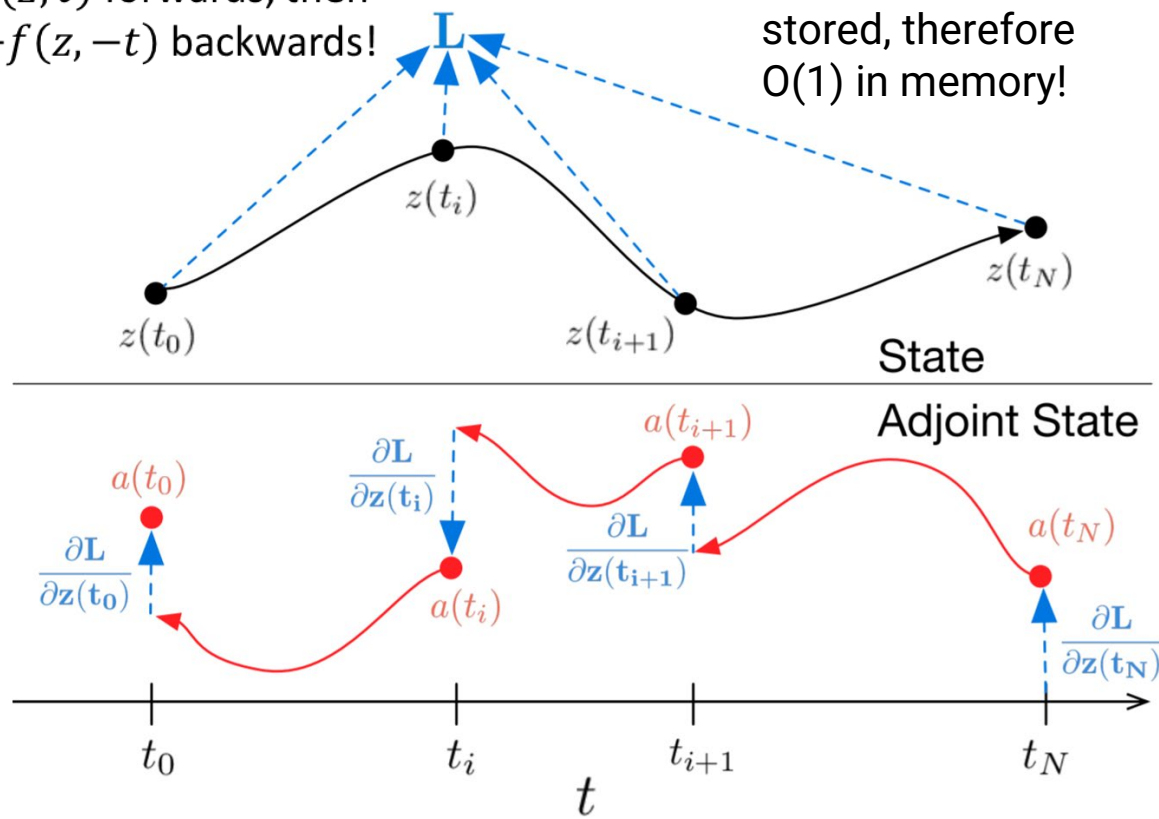
Some methods are “mathematically correct”, but “numerically incorrect”

SciML is a software problem.

Machine Learning Neural Ordinary Differential Equations

$u' = f(z, t)$ forwards, then
 $u' = -f(z, -t)$ backwards!

Timeseries is not
 stored, therefore
 $O(1)$ in memory!



The adjoint equation is an ODE!

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^\top \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}}$$

How do you get $z(t)$? One suggestion:
 Reverse the ODE

$$\frac{d\mathbf{a}_{aug}(t)}{dt} = -[\mathbf{a}(t) \quad \mathbf{a}_\theta(t) \quad \mathbf{a}_t(t)] \frac{\partial f_{aug}}{\partial [\mathbf{z}, \theta, t]}(t)$$

“Adjoint by reversing” also is unconditionally unstable on some problems!

Advection Equation:

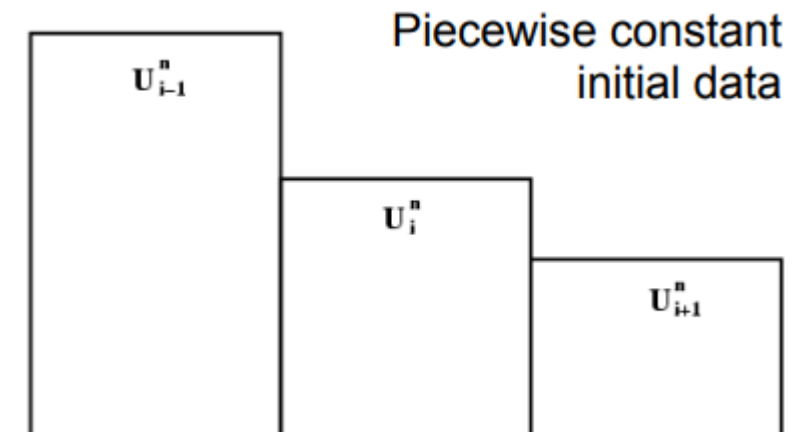
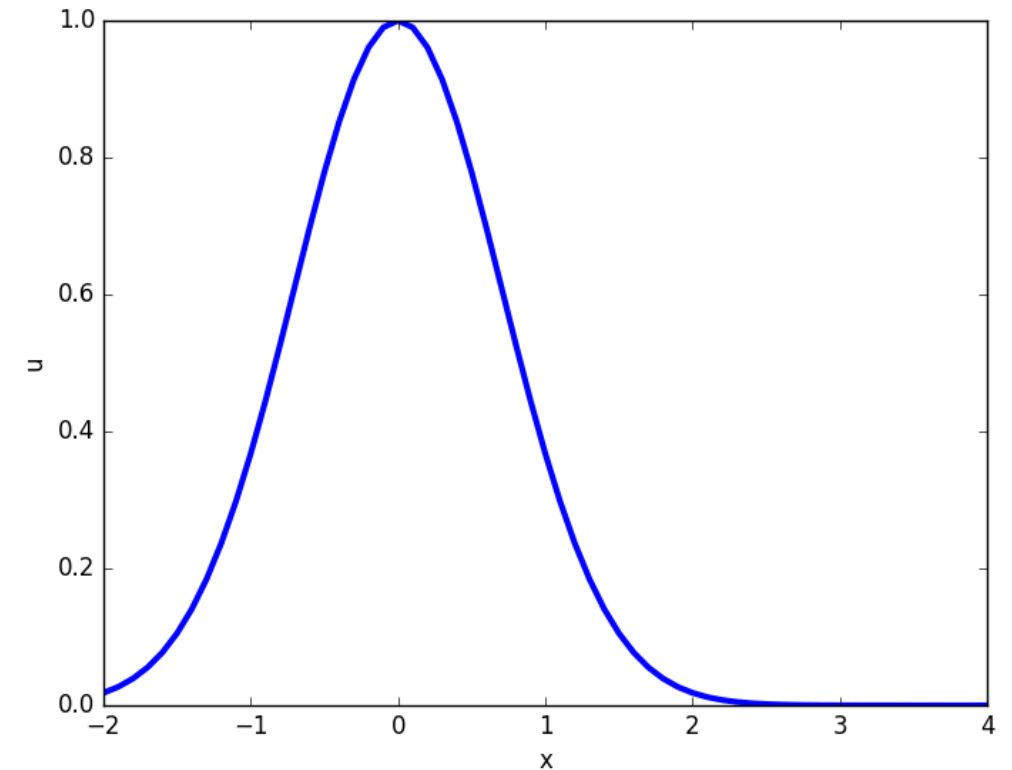
$$\frac{\partial u}{\partial t} + a \frac{\partial u}{\partial x} = 0$$

Approximating the derivative in x has two choices: forwards or backwards

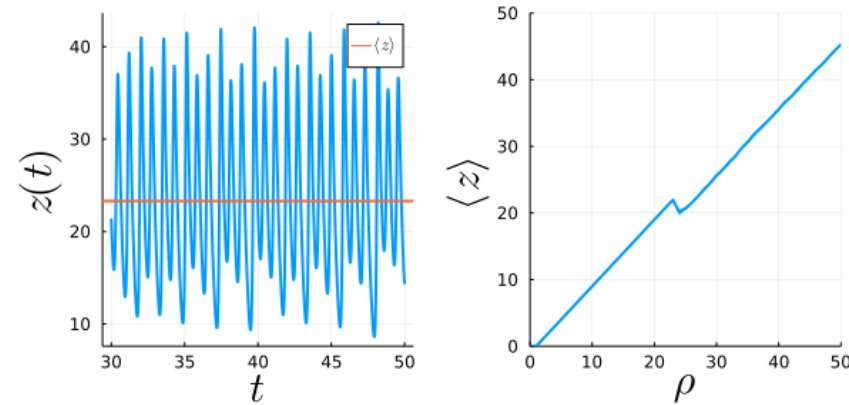
$$u'_i = -\frac{a(u_i - u_{i-1})}{\Delta x} \text{ or } u'_i = -\frac{a(u_{i+1} - u_i)}{\Delta x}?$$

If you discretize in the wrong direction you get **unconditional instability**

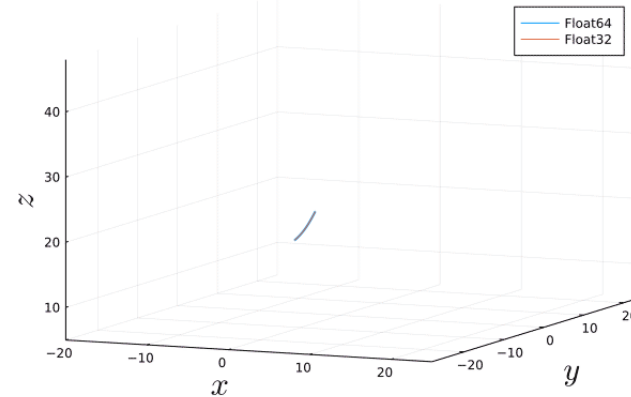
You need to understand the engineering principles and the numerical simulation properties of domain to make ML stable on it.



Differentiation of Chaotic Systems: Shadow Adjoints



chaotic systems: trajectories diverge to $o(1)$ error ... **but shadowing lemma** guarantees that the solution lies on the attractor



$$\frac{d}{d\rho} \langle z \rangle_{\infty} \neq \lim_{T \rightarrow \infty} \frac{\partial}{\partial \rho} \langle z \rangle_T$$

- **AD** and finite differencing fails!

$$\left. \frac{d\langle z \rangle_{\infty}}{d\rho} \right|_{\rho=28} \approx -49899 \text{ (ForwardDiff)}$$

$$\left. \frac{d\langle z \rangle_{\infty}}{d\rho} \right|_{\rho=28} \approx 472 \text{ (Calculus)}$$

- **Shadowing methods** in DiffEqSensitivity.jl

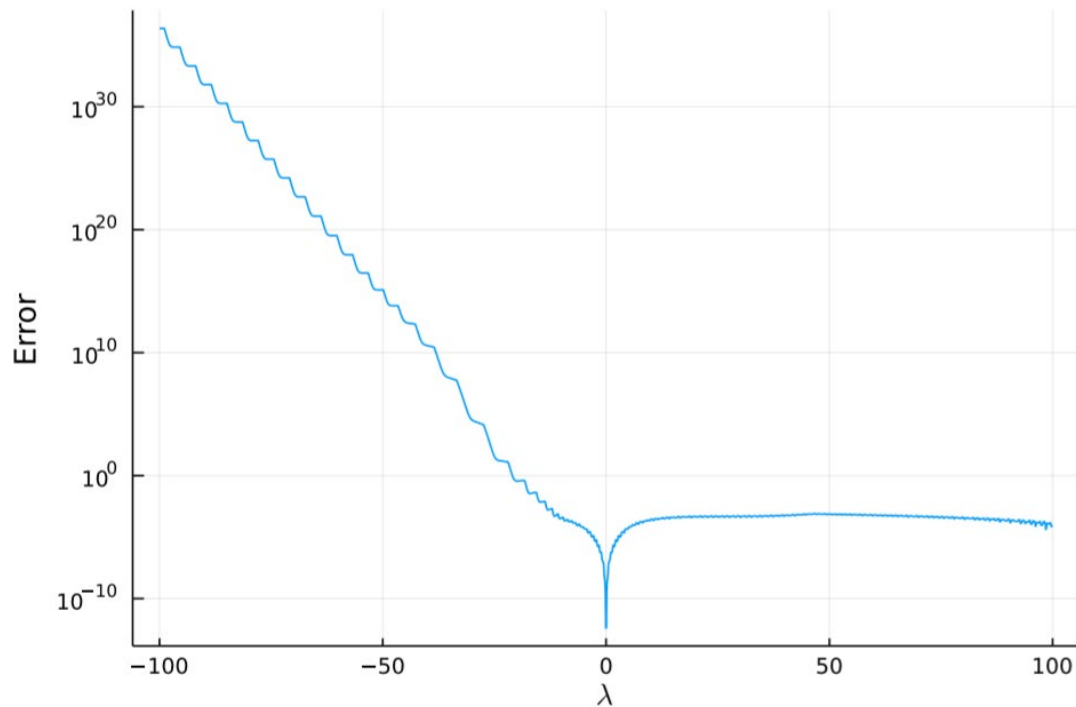
$$\left. \frac{d\langle z \rangle_{\infty}}{d\rho} \right|_{\rho=28} \approx 1.028 \text{ (LSS/AdjointLSS)}$$

$$\left. \frac{d\langle z \rangle_{\infty}}{d\rho} \right|_{\rho=28} \approx 0.997 \text{ (NILSS)}$$

Problems With Naïve Adjoint Approaches On Stiff Equations

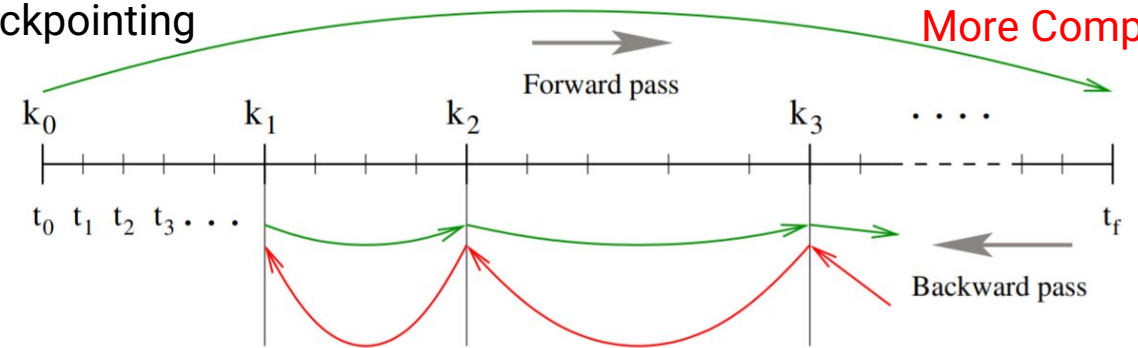
Error grows exponentially...

$u'(t) = \lambda u(t)$, plot the error in the reverse solve:



How do you get $u(t)$ while solving backwards?
3 options!

1. $u' = f(z, t)$ forwards, then
 $u' = -f(z, -t)$ backwards! **Unstable**
2. Store $u(t)$ while solving forwards (dense output) **High memory**
3. Checkpointing **More Compute**



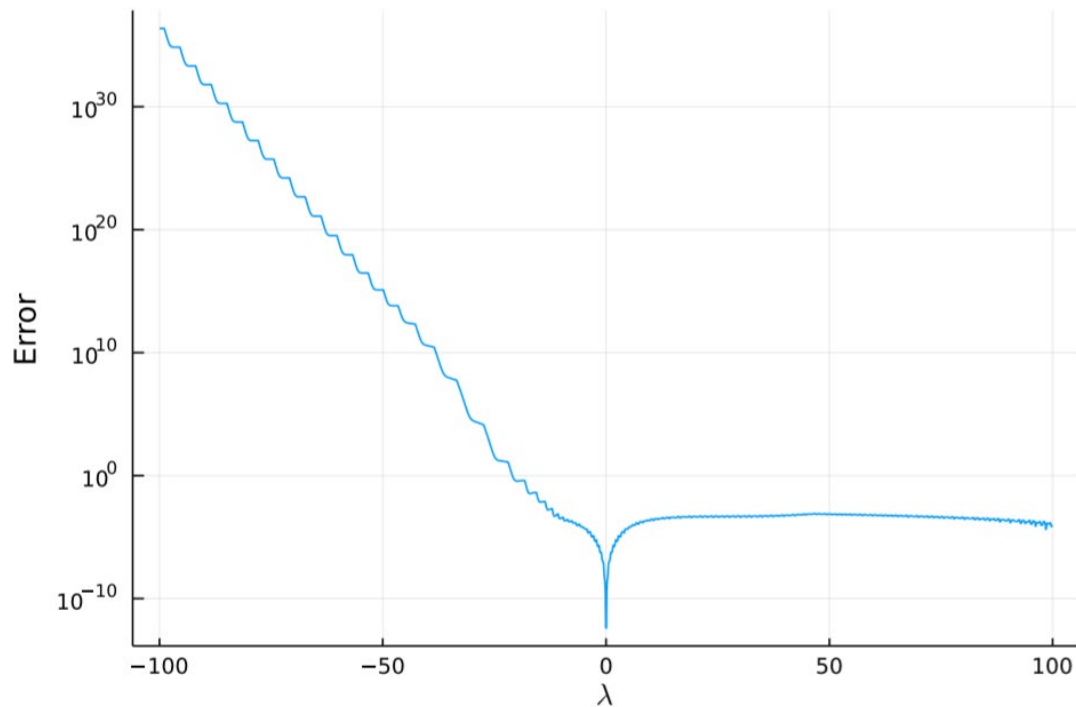
Each choices has an engineering trade-off!

Kim, Suyong, Weiqi Ji, Sili Deng, and Christopher Rackauckas. "Stiff neural ordinary differential equations." *Chaos* (2021).

Problems With Naïve Adjoint Approaches On Stiff Equations

Error grows exponentially...

$u'(t) = \lambda u(t)$, plot the error in the reverse solve:



Kim, Suyong, Weiqi Ji, Sili Deng, and Christopher Rackauckas. "Stiff neural ordinary differential equations." *Chaos* (2021).

Compute cost is cubic with parameter size when stiff

Size of reverse ODE system is:

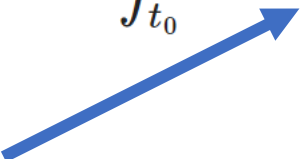
$$2states + parameters$$

Linear solves inside of stiff ODE solvers, \sim cubic

Thus, adjoint cost:

$$O((states + parameters)^3)$$

Problems With Naïve Adjoint Approaches On Stiff Equations

$$\frac{dG}{dp} = \lambda^*(t_0) \frac{dG}{du}(t_0) + \int_{t_0}^T (g_p + \lambda^* f_p) dt$$


How do you calculate the integral?

1. Store $\lambda(t)$ while solving backwards (dense output)

High memory

2. $\mu' = -\lambda^* f_p + g_p$ where $\mu(T) = 0$ Size = Number of Parameters

3. Use an IMEX integrator and solve $\mu' = -\lambda^* f_p + g_p$ explicitly

4. Our paper describes a 4th way!



Compute cost is cubic with parameter size when stiff

Size of reverse ODE system is:

$$2states + parameters$$

Linear solves inside of stiff ODE solvers, ~cubic

Thus, adjoint cost:

$$O((states + parameters)^3)$$

Thus, adjoint cost without extra memory:

$$O(states^3 + parameters)$$

Kim, Suyong, Weiqi Ji, Sili Deng, and Christopher Rackauckas. "Stiff neural ordinary differential equations." *Chaos* (2021).

The math has >20 ways to implement.

Every choice makes engineering trade-offs.

SciML is a software problem.

DiffEqSensitivity.jl: Every adjoint is optimized for a different case

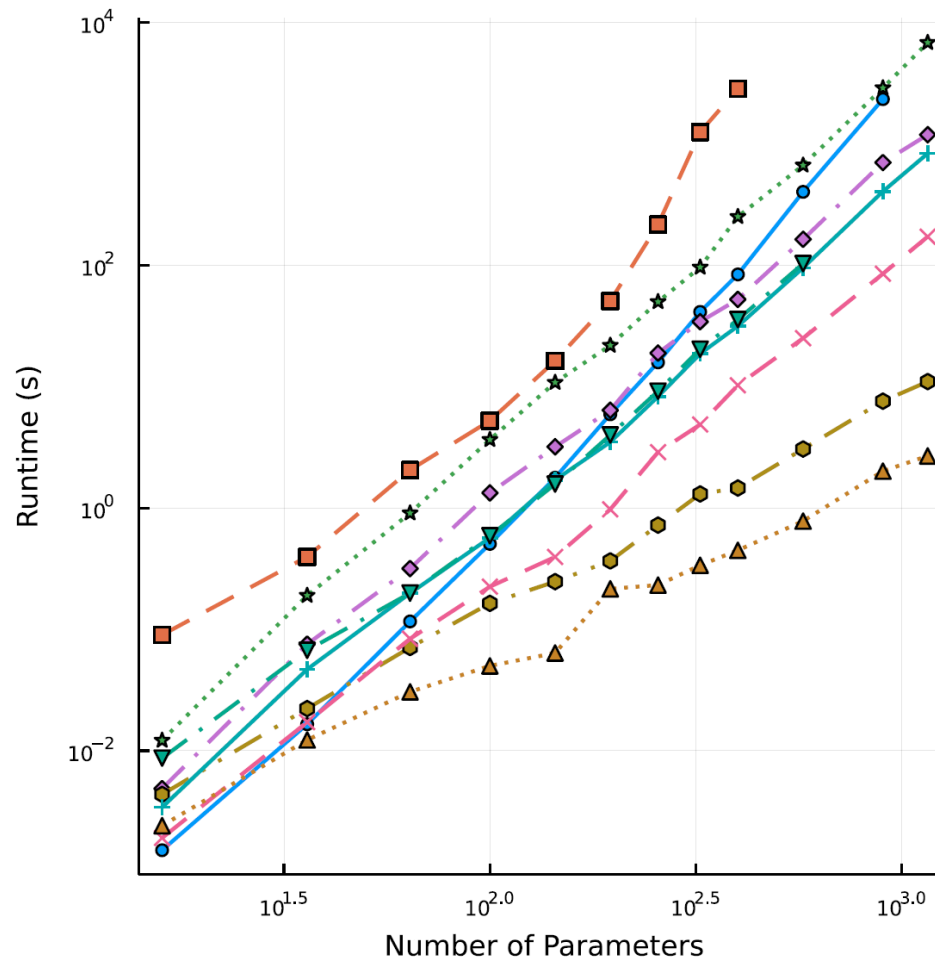
Method	Stability	Stiff Performance Scaling	Memory Usage
BacksolveAdjoint	Poor	$O((s + p)^3)$	Low. $O(1)$
InterpolatingAdjoint	Good	$O((s + p)^3)$	High. Requires full continuous solution of forward
QuadratureAdjoint	Good	$O(s^3 + p)$	Higher. Requires full continuous solution of forward and Lagrange multiplier
BacksolveAdjoint (Checkpointed)	Okay	$O((s + p)^3) + C$	Medium. $O(c)$ where c is the number of checkpoints
InterpolatingAdjoint (Checkpointed)	Good	$O((s + p)^3) + C$	Medium. $O(c)$ where c is the number of checkpoints
ReverseDiffAdjoint	Best	$O(s^3 + p) + C$	Highest. Requires full forward and reverse AD of solve
TrackerAdjoint	Best	$O(s^3 + p) + C$	Highest. Requires full forward and reverse AD of solve
ForwardLSS/AdjointLSS/N ILSS	Chaos	Not even comparable: expensive.	Super duper high OMG.

How the adjoint is calculated also matters!

Gradient calculations on a stiff PDE, varying Δt

Rackauckas, Christopher, et al. "A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions." *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 1-8.

Sensitivity Scaling on Brusselator



Methods with Reverse-mode vjp seeding + new adjoints give 3 orders of magnitude improvement!

The SciML ecosystem is the only one with fully-featured Universal Differential Equations

Feature	SciML (Julia)	Sundials (C++)	PETSc TS (C++)	torchdiffeq	Jax
Stiff ODEs and DAEs	Hundreds of methods tested and tuned on hundreds of problems	Yes (CVODE_BDF and IDA)	Yes (Rosenbrock-W methods, BDFs, etc.)	None	None (one in progress, ~200 times slower than SciPy according to the author!)
Adjoint Methods	11 choices tuned for different scenarios, including stabilized checkpointing, differentiate the solver, reversing adjoint	Stabilized checkpointing, no AD integration, no chaos compatibility	Discrete sensitivity analysis, no AD integration, no chaos compatibility	Requires reversing the ODE or differentiate the solver (tracing)	Requires reversing the ODE
Parallelism	GPU, MPI, multithreading	GPU, MPI, multithreading	GPU, MPI, and multithreading	GPU	GPU
Event handling	Yes	Yes	Yes	None	None
SDEs	Lots of methods, including stabilized, methods for stiff equations, high strong order, high weak order	None	None	torchsde, only diagonal noise (or order 0.5), requires reversing the SDE	None
Delays	All ODE methods	None	None	None	None

The performance difference in UDEs is not small when the right solvers and adjoints are chosen

These ODEs are non-stiff ODEs from astrodynamics, chemical kinetics, numerical weather prediction, etc. and include scalarized operations

Relative time to solve

Number of ODEs	3	28	768	3,072	12,288	49,152	196,608	786,432
DifferentialEquations.jl	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x	1.0x
DifferentialEquations.jl dopri5	1.0x	1.6x	2.8x	2.7x	3.0x	3.0x	3.9x	2.8x
torchdiffeq dopri5	4,900x	190x	840x	220x	82x	31x	24x	17x

Spiral Neural ODE (from original Neural ODE paper)

- DiffEqFlux defaults: 7.4 seconds
- DiffEqFlux optimized: 2.7 seconds
- torchdiffeq: 288.965871299999 seconds

Geometric Brownian Motion of size 4

The SDE is solved 100 times. The summary of the results is as follows:

- torchsde: 1.87 seconds
- DifferentialEquations.jl: 0.00115 seconds

Note: performance is not necessarily indicative of large “pure” neural equations