Scientific Machine Learning: How to integrate structure and models into learning, what can go wrong, and what to do about it.

Chris Rackauckas

VP of Modeling and Simulation,
Julia Computing

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

Director of Scientific Research,
Pumas-AI

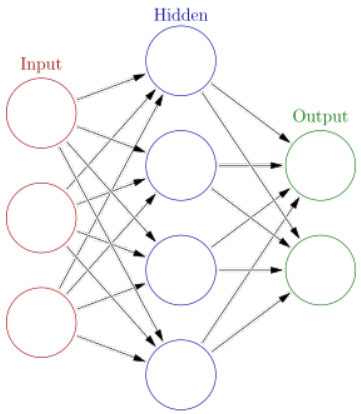
Outline: SciML requires more than just sticking automatic differentiation on a simulator.

Part 1: Understanding derivatives and their potential issues.
Part 2: How simulators must be modified to improve the fitting process.
Part 3: Alternatives to direct simulation fitting which may be more robust in some contexts
Part 4: How the performance of simulators and deep learning differ

Prologue: Why do Differentiable Simulation with SciML?

Universal (Approximator) Differential Equations

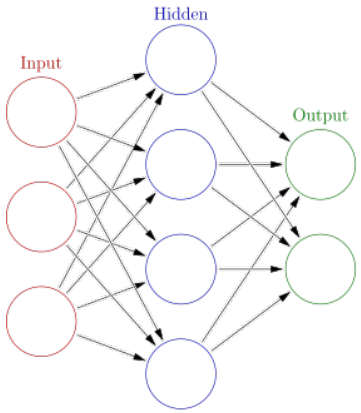
$$u' = f(u,$$



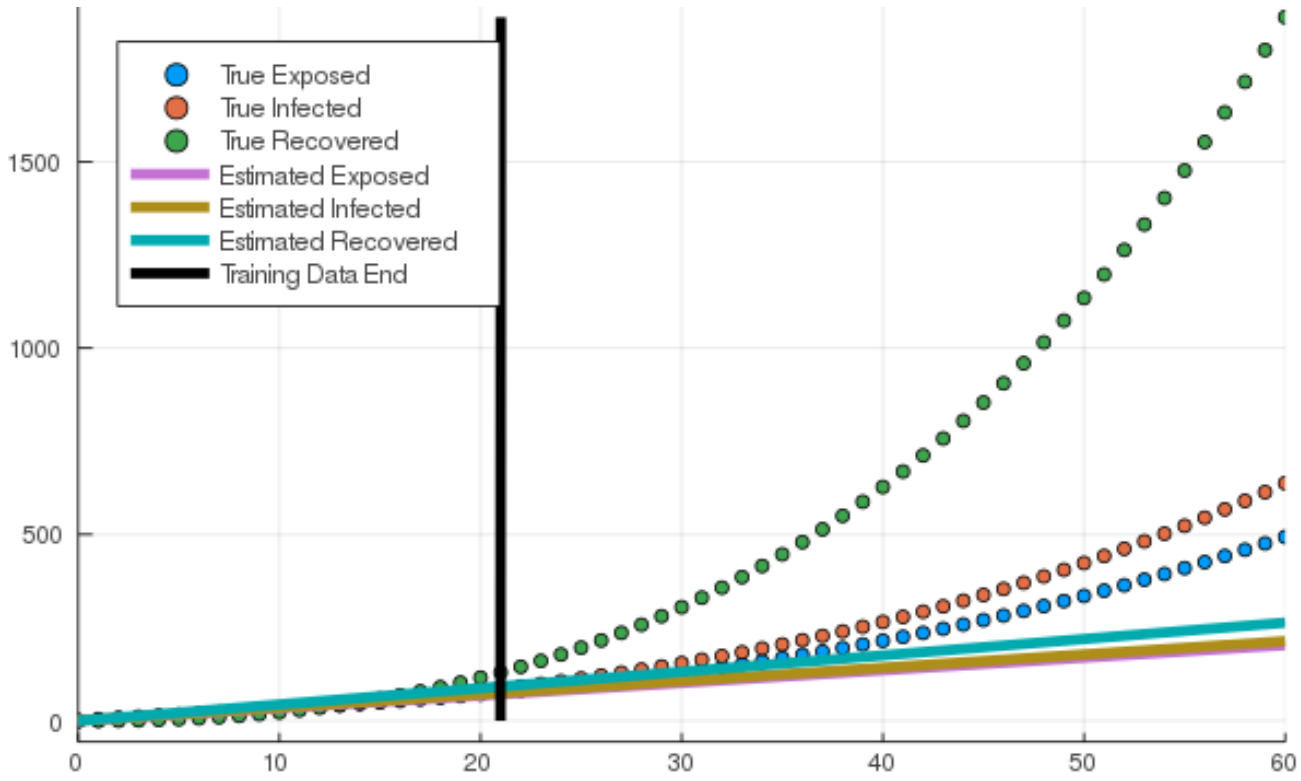
$$, t)$$

$$x' = \alpha x \quad +$$

$$y' = -\beta y +$$



Let’s dive in a bit!
Neural ODE: Learn the whole model

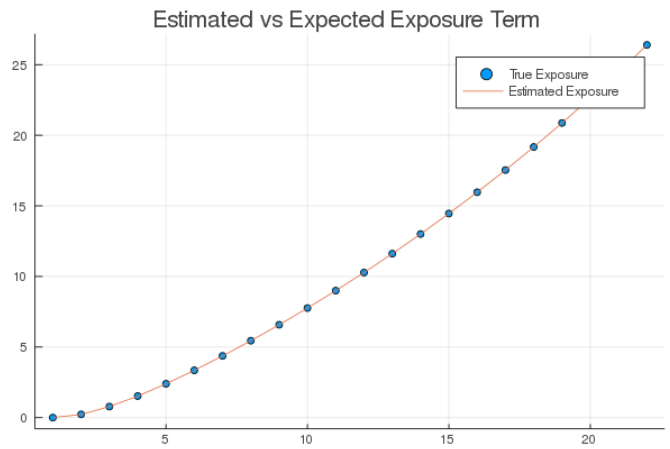


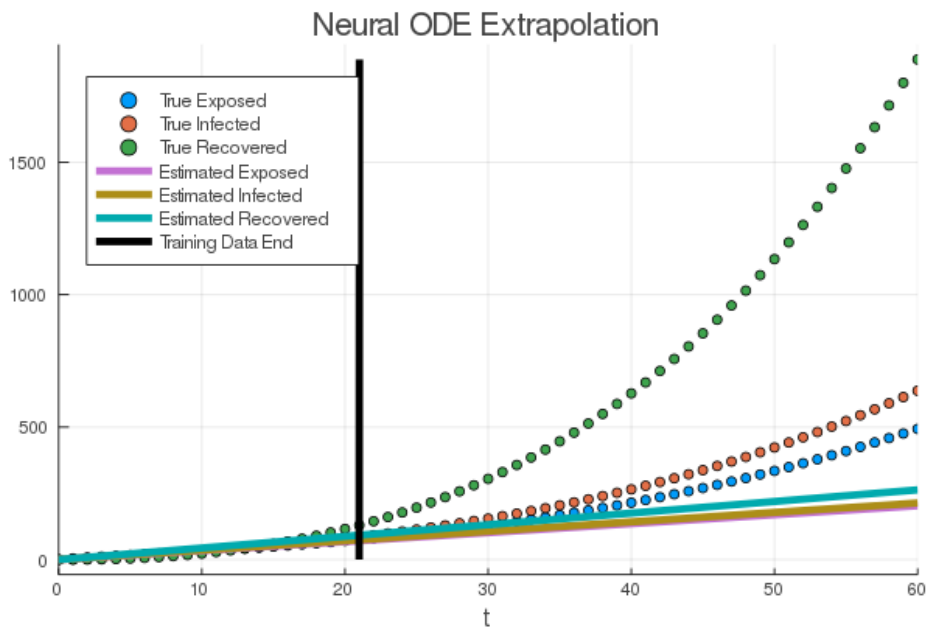
u’=NN(u) trained on 21 days of data

Can fit, but not enough information to accurately extrapolate

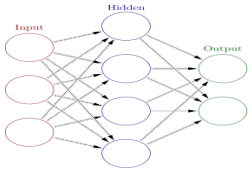
Does not have the correct asymptotic behavior

Universal ODE

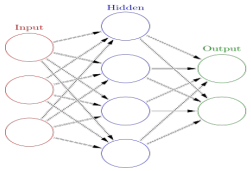




$$
\begin{aligned}
S' &= -\frac{\beta_0 S F}{N} - \\
E' &= \frac{\beta_0 S F}{N} + \\
I' &= \sigma E - (\gamma + \mu) I, \\
R' &= \gamma I - \mu R, \\
N' &= -\mu N, \\
D' &= d\,\gamma I - \lambda D, \quad \text{and} \\
C' &= \sigma E,
\end{aligned}
$$



$- \mu S,$

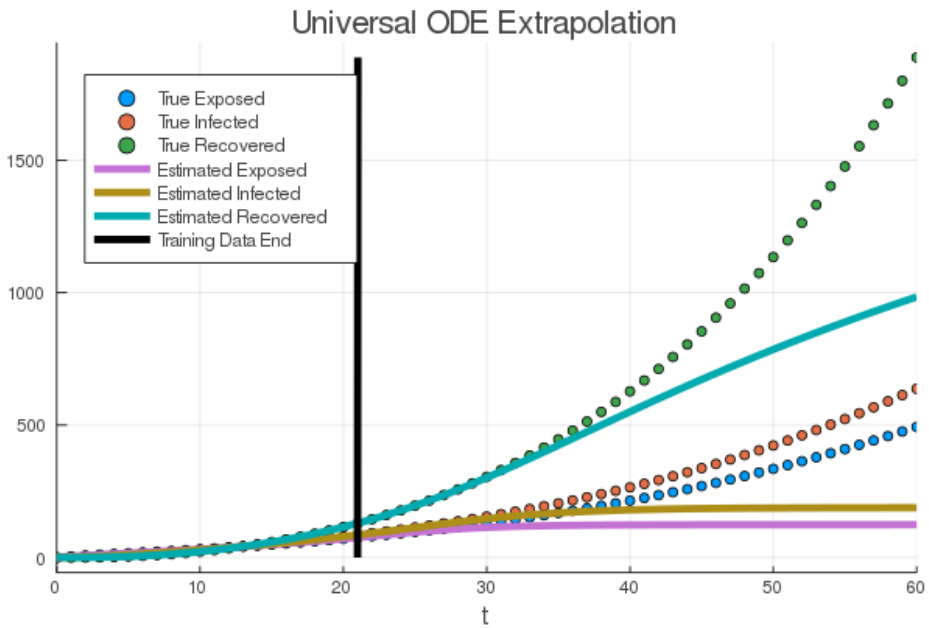


$- \Big(\sigma + \mu\Big) E,$

Exposure:
Unknown

Infection rates: known
From disease quantities

Percentage of cases known to be severe, can be estimated



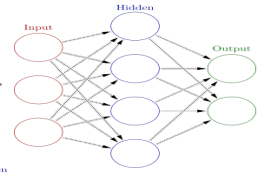
Universal ODE -> Internal Sparse Regression

Sparse Identification on only the missing term:

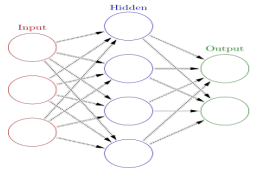
I * 0.10234428543435758 + S/N * I * 0.11371750552005416 + (S/N) ^ 2 * I * 0.12635459799855597

Sparsity improves generalizability!

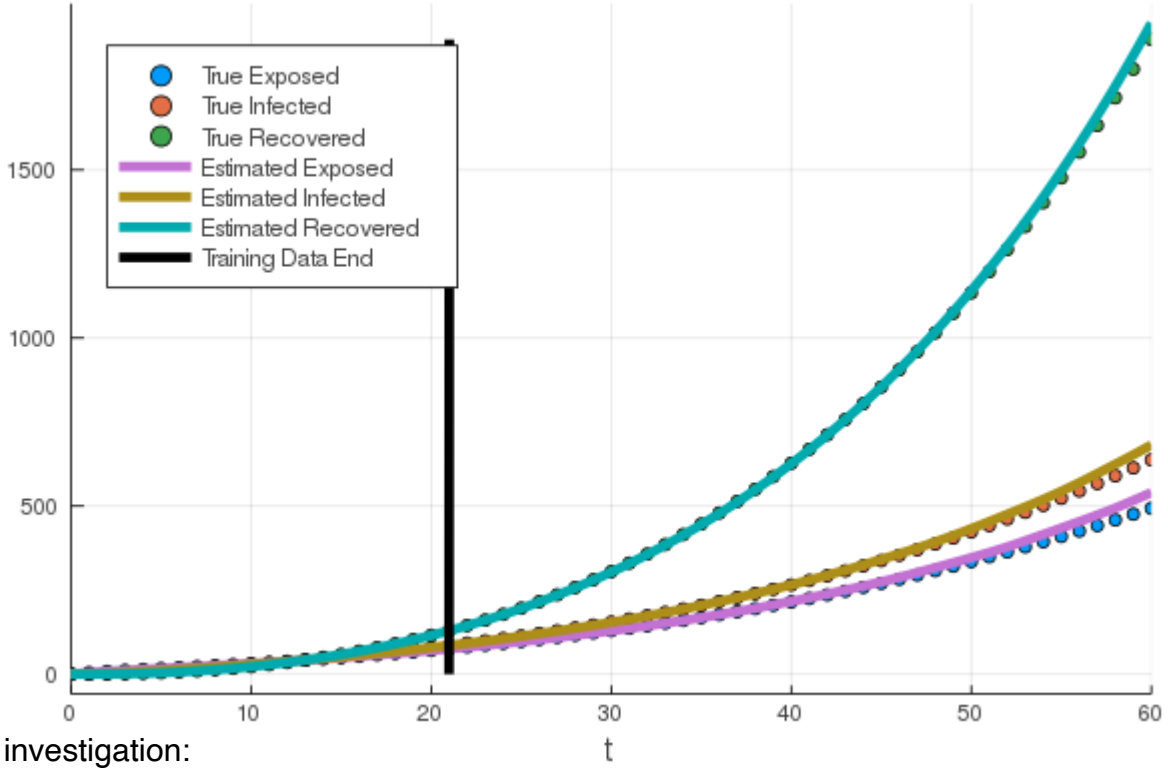
$$
\begin{aligned}
S' &= -\frac{\beta_0 S F}{N} - \\
E' &= \frac{\beta_0 S F}{N} + \\
I' &= \sigma E - (\gamma + \mu) I, \\
R' &= \gamma I - \mu R, \\
N' &= -\mu N, \\
D' &= d\,\gamma I - \lambda D, \quad \text{and} \\
C' &= \sigma E,
\end{aligned}
$$



$- \mu S,$



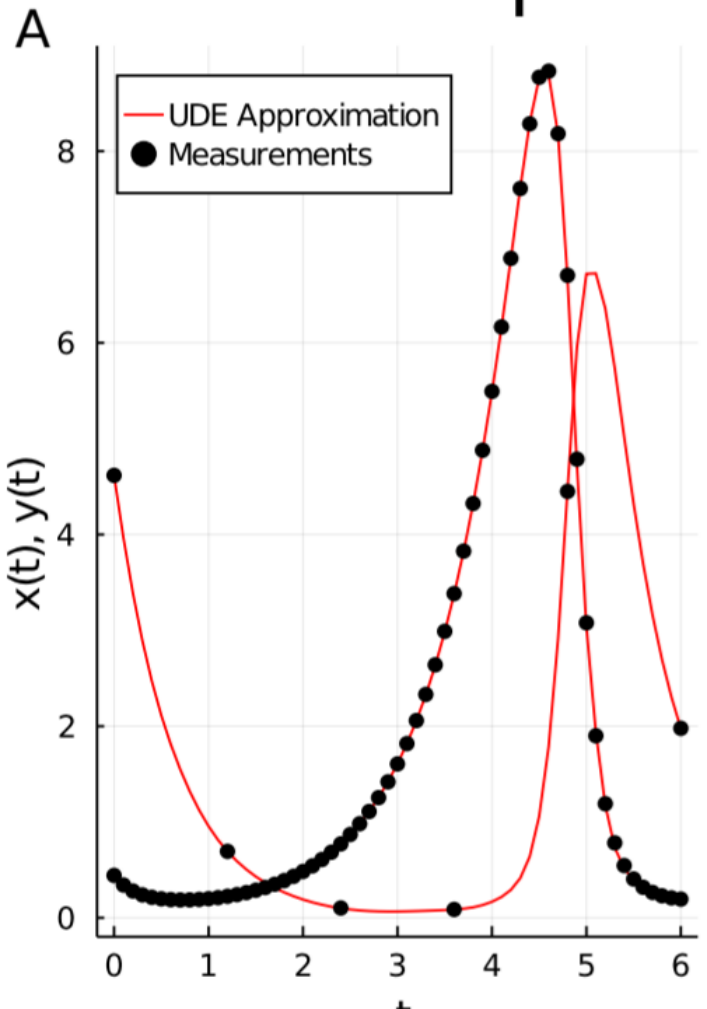
$- \left(\sigma + \mu\right) E,$

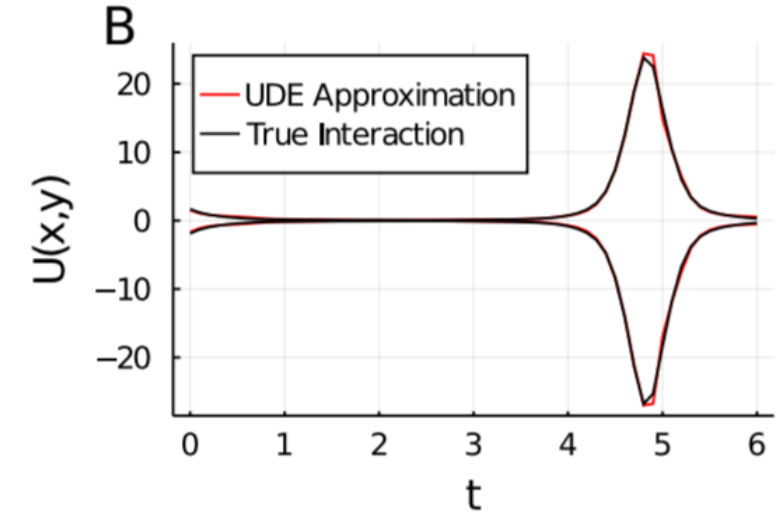


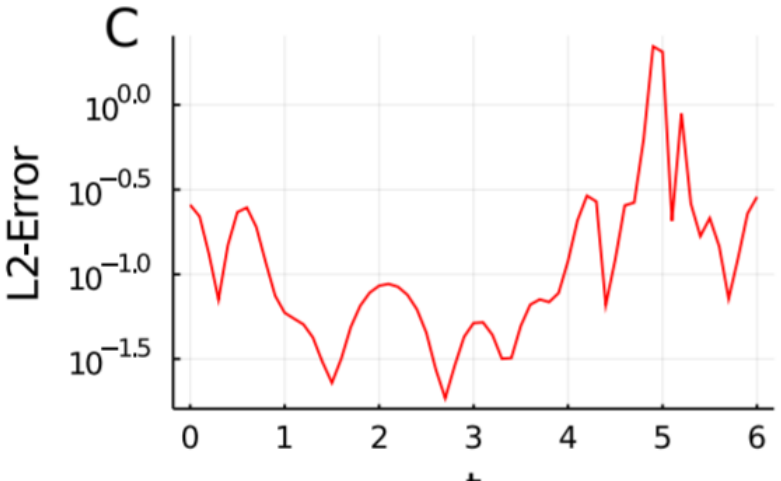
For further investigation:

Acquesta, Erin, Teresa Portone, Raj Dandekar, Chris Rackauckas, Rileigh Bandy, and Jose Huerta. Model-Form Epistemic Uncertainty Quantification for Modeling with Differential Equations: Application to Epidemiology. No. SAND2022-12823. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2022.

Universal (Approximator) Differential Equations







Measurement Data

Known and Unknown Relations

$$\dot{x} = \alpha x + U_1(\theta, x, y)$$
$$\dot{y} = -\theta_1 y + U_2(\theta, x, y)$$

Train Model $\hat{\theta} = \min_\theta \mathcal{L}(\theta)$

Recover Unknowns

$$\hat{\Xi} = \min_{\Xi} \|U(\hat{\theta}, X) - \Theta\Xi\|_2 + \lambda\|\Xi\|_1$$

$$\Theta\hat{\Xi} \mapsto \begin{bmatrix} \xi_1 xy \\ \xi_2 xy \end{bmatrix}$$

Build fully symbolic Model

$$\dot{x} = \alpha x + \xi_1 xy$$
$$\dot{y} = -\theta_1 y + \xi_2 xy$$

UODEs show accurate extrapolation and generalization

Run the code yourself!

https://github.com/Astroinformatics/ScientificMachineLearning/blob/main/neuralode_gw.ipynb

Upon denoting $\mathbf{x} = (\phi, \chi, p, e)$, we propose the following family of UDEs to describe the two-body relativistic dynamics:

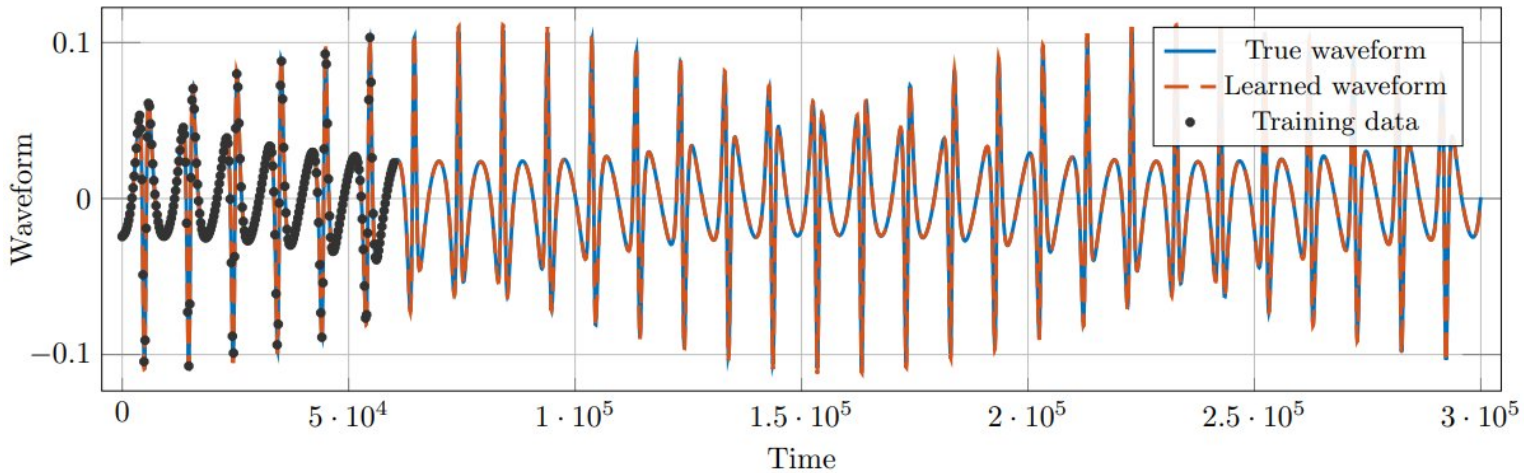
$$\dot{\phi} = \frac{(1 + e\cos(\chi))^2}{Mp^{3/2}}\big(1 + \mathcal{F}_1(\cos(\chi), p, e)\big), \tag{5a}$$

$$\dot{\chi} = \frac{(1 + e\cos(\chi))^2}{Mp^{3/2}}\big(1 + \mathcal{F}_2(\cos(\chi), p, e)\big), \tag{5b}$$

$$\dot{p} = \mathcal{F}_3(p, e), \tag{5c}$$

$$\dot{e} = \mathcal{F}_4(p, e), \tag{5d}$$

Example using binary black hold dynamics with LIGO gravitational wave data



Keith, Brendan, Akshay Khadse, and Scott E. Field. "Learning orbital dynamics of binary black hole systems from gravitational wave measurements." Physical Review Research 3, no. 4 (2021): 043101.

Universal Differential Equations Predict Chemical Processes

$$\frac{\partial c}{\partial t^*} = -\frac{1-\varepsilon}{\varepsilon}\mathrm{ANN}(q, q^*, \theta) - \frac{\partial c}{\partial x^*} + \frac{1}{Pe}\frac{\partial c^2}{\partial x^{*2}},$$

$$\frac{\partial q}{\partial t^*} = \mathrm{ANN}(q, q^*, \theta),$$

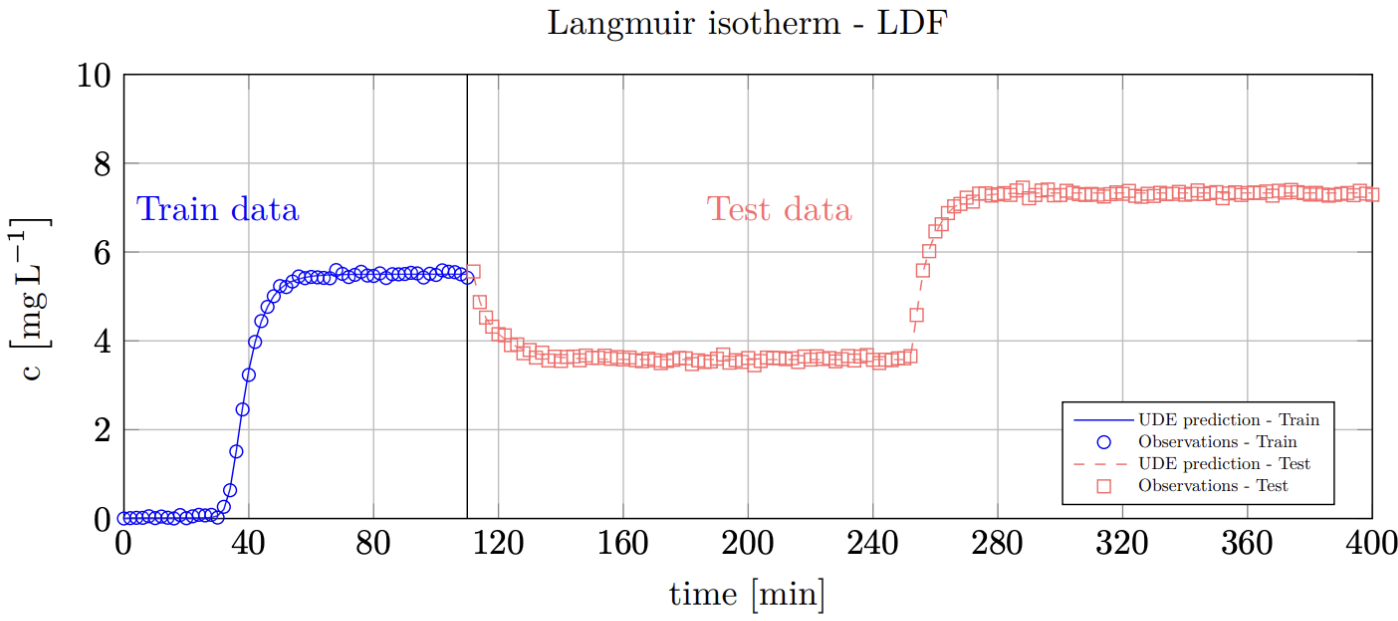
$$\frac{\partial c(x^* = 1, \forall t)}{\partial x^*} = 0,$$

$$\frac{\partial c(x^* = 0, \forall t)}{\partial x^*} = Pe(c - c_{inlet}),$$

$$c(x^* \in (0,1), t^* = 0) = c_0,$$

$$q(x^* \in (0,1), t^* = 0) = q^*(c_0),$$

$$q^* = f(c, p),$$



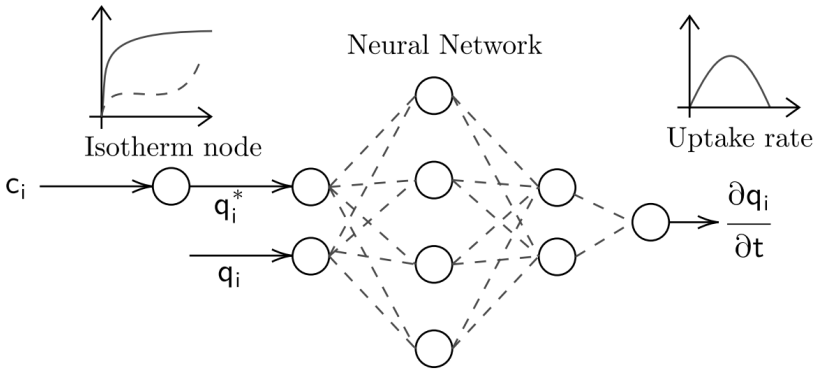


Figure 2: Schematic representation of the proposed hybrid model.

Santana, V. V., Costa, E., Rebello, C. M., Ribeiro, A. M., Rackauckas, C., & Nogueira, I. B. (2023). Efficient hybrid modeling and sorption model discovery for non-linear advection-diffusion-sorption systems: A systematic scientific machine learning approach. *arXiv preprint arXiv:2303.13555*.

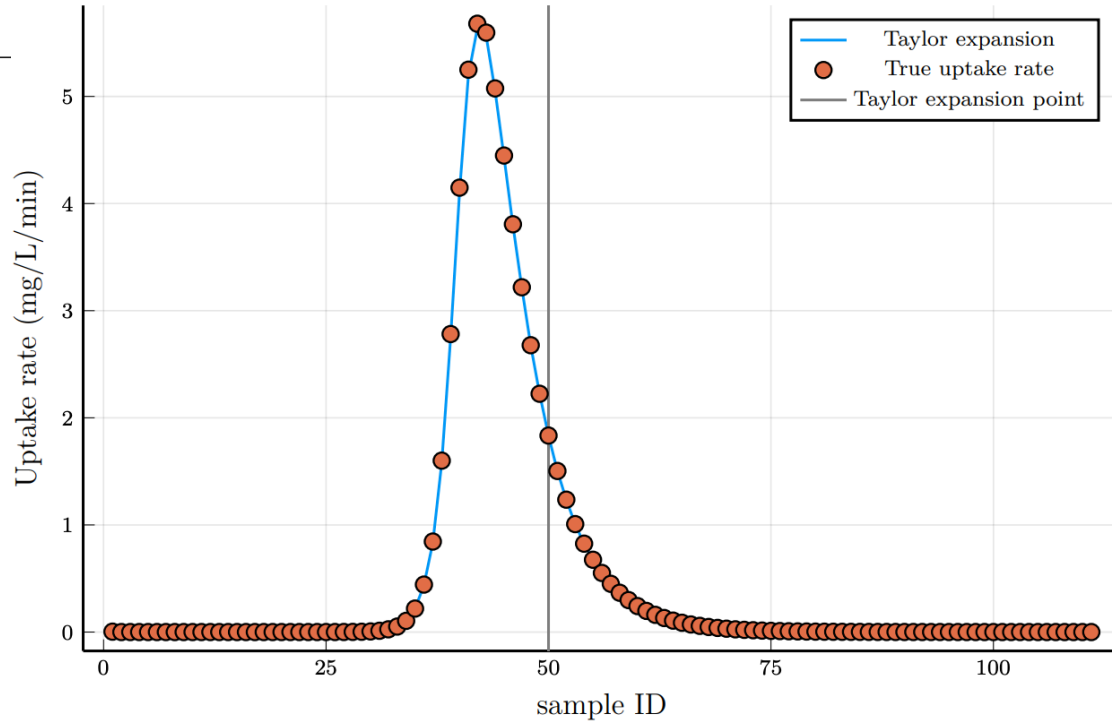
Universal Differential Equations Predict Chemical Processes

Table 5: Symbolic regression learned polynomials.

Isotherm	Kinetic	True kinetics	Learned kinetics
Langmuir	LDF	$0.22q^* - 0.22q$	$-0.535 - 0.225q + 0.234(q^*)$
Langmuir	improved LDF	$0.22(q^* + 0.2789q^* e^{\frac{-q}{2q^*}} - q)$	$-0.554 - 0.234q + 0.281(q^*)$
Langmuir	Vermeulen's	$0.22\frac{q^{*2}-q^2}{2.0q}$	$-0.6098 + 0.0122q + 0.263q^*$ $-0.00526qq^*$
Sips	LDF	$0.22q^* - 0.22q$	$0.198q^* - 0.200q$
Sips	improved LDF	$0.22(q^* + 0.2789q^* e^{\frac{-q}{2q^*}} - q)$	$0.277q^* - 0.241q$
Sips	Vermeulen's	$0.22\frac{q^{*2}-q^2}{2.0q}$	$-0.003557q^{*2} - 0.216q + 0.395q^*$

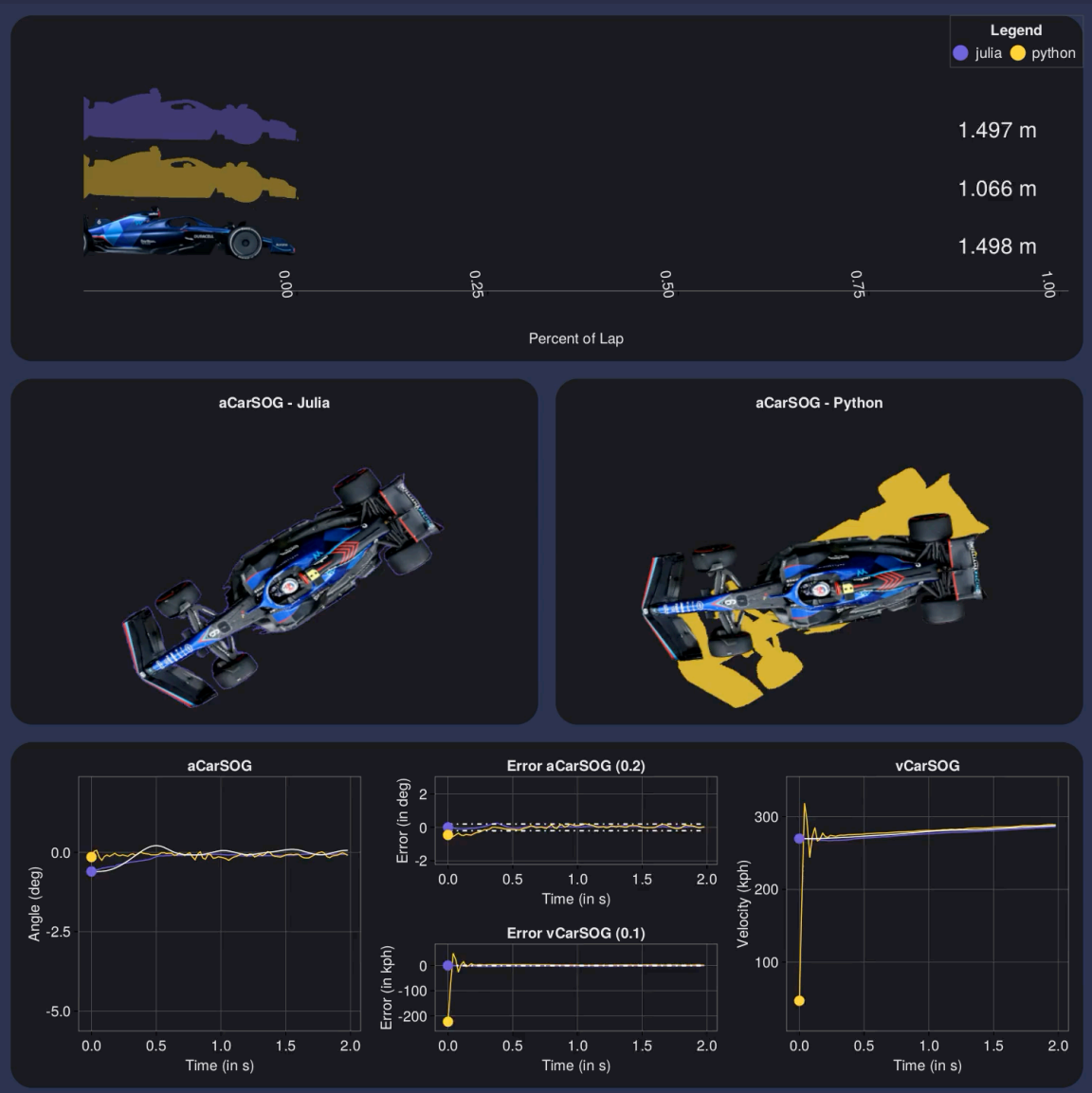
$$0.22(q^* + 0.2789q^* e^{\frac{-q}{2q^*}} - q)(49.23, 49.22) \approx 1.834 + 0.275q^* - 0.238q + \mathcal{O}(\|x^2\|)$$

Recovers equations with the same 2nd order Taylor expansion



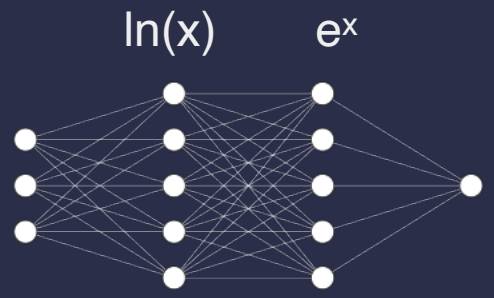
Scientific Machine Learning Digital Twins: More Realistic Results than Pure ML

WILLIAMS RACING



Physically-Informed Machine Learning

$$\dot{\beta} \approx \frac{a_y}{u} - \frac{a_x}{u} - r$$



Using knowledge of the physical forms as part of the design of the neural networks.

Smoother, more accurate results

For more information, see the case study on the JuliaHub website

SciML Shows how to build Earthquake-Safe Buildings

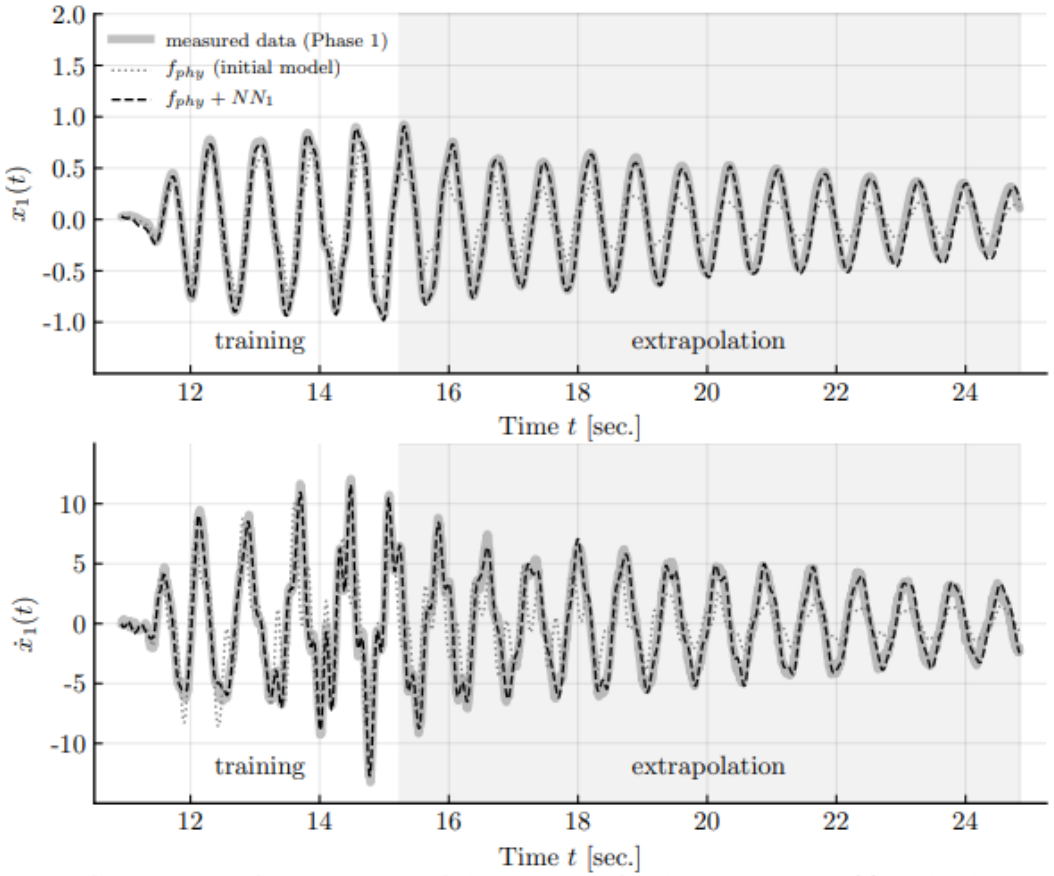


Figure 12: Comparison of time history of the response for displacement $x_1(t)$ and velocity $\dot{x}_1(t)$ for the NSD experiment (Phase 1).

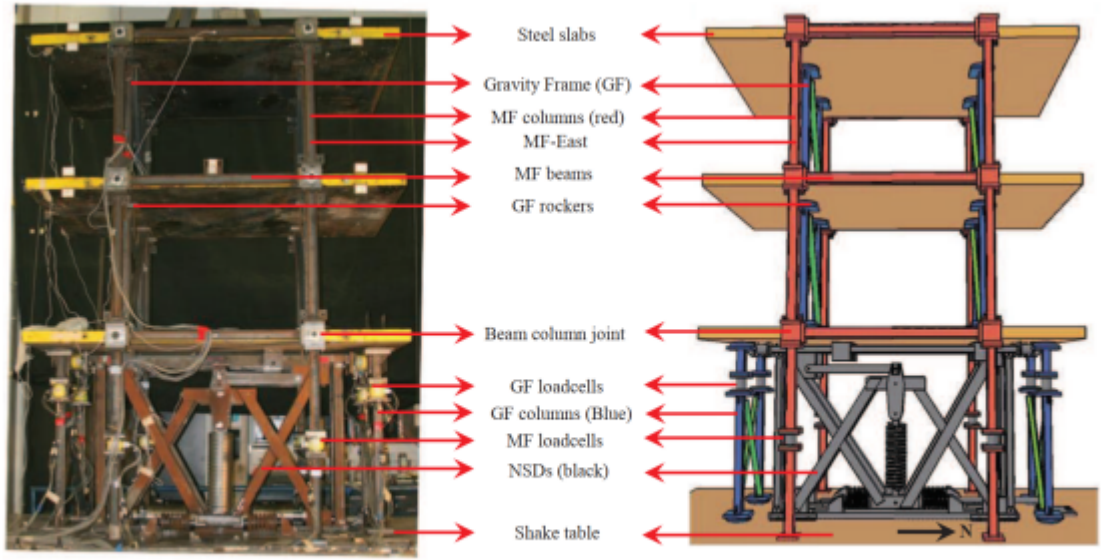


Figure 10: The structural system equipped with a negative stiffness device in between the first floor and the shake table.

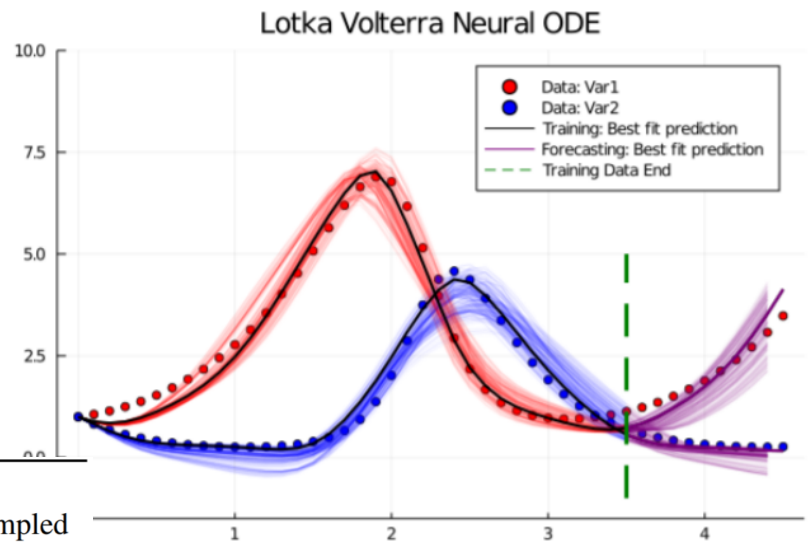
Structural identification with physics-informed neural ordinary differential equations
Lai, Zhilu, Mylonas, Charilaos, Nagarajaiah, Satish, Chatzi, Eleni

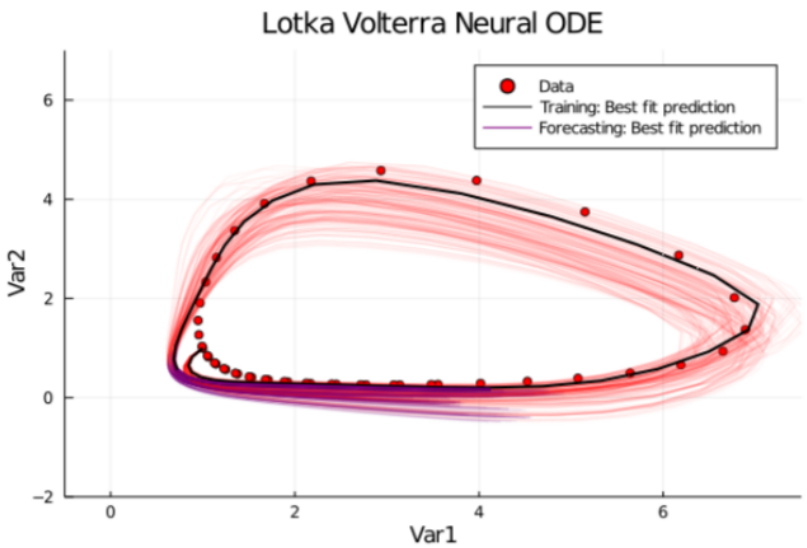
For a detailed walkthrough of UDEs and applications watch on Youtube:

Chris Rackauckas: Accurate and Efficient Physics-Informed Learning Through Differentiable Simulation

Bayesian UODEs: Knowledge-Enhanced Model Discovery with UQ

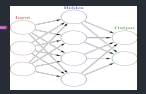
Result: Probability of Missing Mechanisms





λ	Number of Active terms	Dominant terms	Error	Mean AIC score	% sampled
0.01	9	u_1^2, u_2^2, u_1u_2 $u_1^2u_2^2, u_1^2u_2, u_2^2u_1$ u_1u_2, const	0.765	40.4	100
0.1	9	u_1^2, u_2^2, u_1u_2 $u_1^2u_2^2, u_1^2u_2, u_2^2u_1$ u_1u_2, const	0.764	35	100
1	5	u_1^2, u_2^2, u_2 $u_1^2u_2, u_1u_2$	0.764	21.6	100
2	2	$u_1^2u_2, u_1u_2$	0.634	7.2	100
3	**1**	u_1u_2	**0.7**	**4.1**	**100**
5	1	$u_1^2u_2$	2.49	-1	100

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



Dandekar, R., Dixit, V., Tarek, M., Garcia-Valadez, A., & Rackauckas, C. (2020). Bayesian Neural Ordinary Differential Equations. Languages for Inference (LAFI) 2021 - POPL 2021

Universal Differential-Algebraic Equations: Encoding Physical Constraints

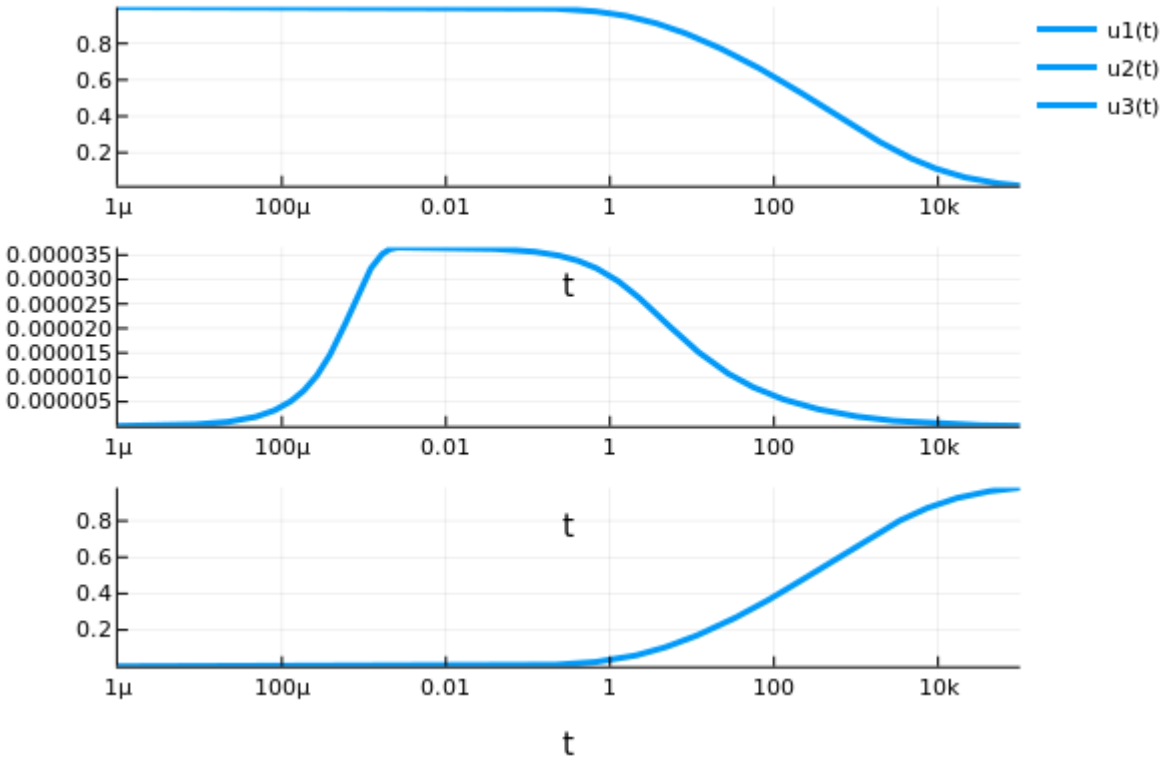
Utilize known chemical kinetics

$$y_1' = -0.04y_1 + \mathrm{NN}_1(y_1, y_2, y_3)$$
$$y_2' = 0.04y_1 + \mathrm{NN}_2(y_1, y_2, y_3)$$
$$1 = y_1 + y_2 + y_3$$

With known conservation laws

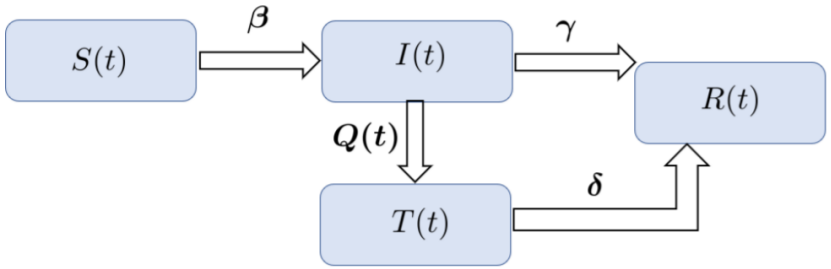
$$Mu' = f(u) + NN(u)$$

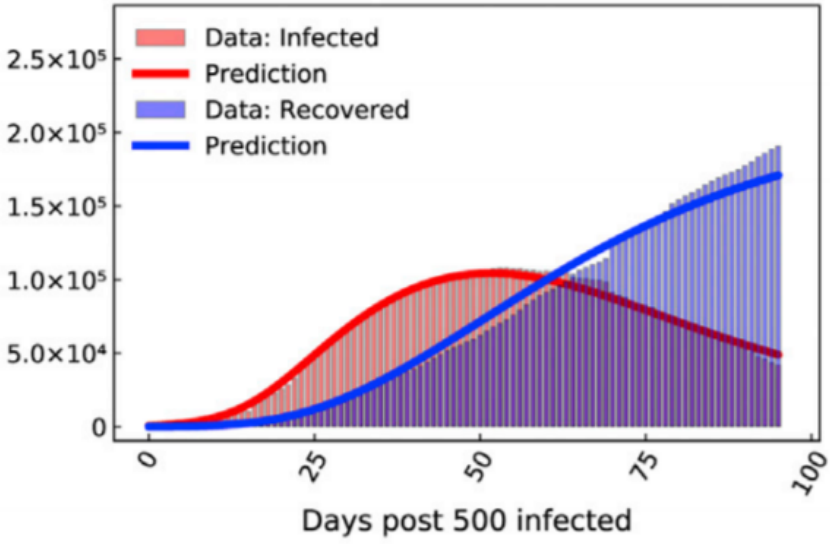
Convert to a mass-matrix DAE (singular mass matrix) and fit



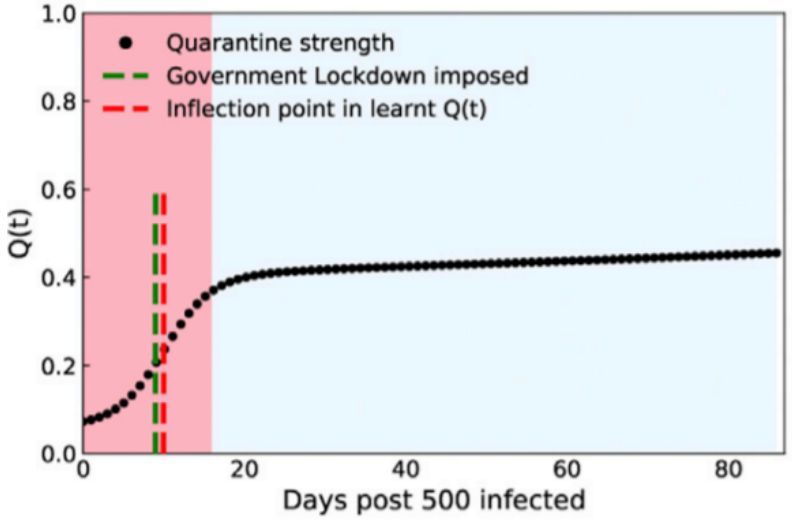
Learn highly stiff equations: **Hessian condition number** 10^{13}

QSIR Predicts Quarantine Measure Evolution

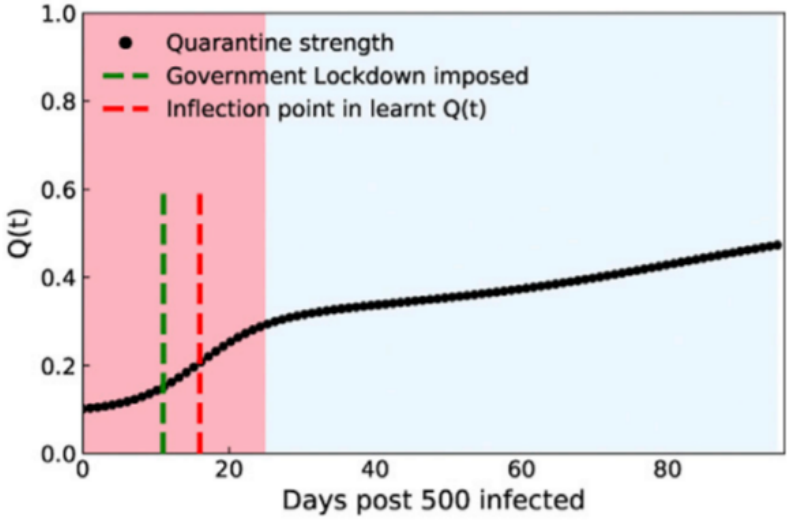




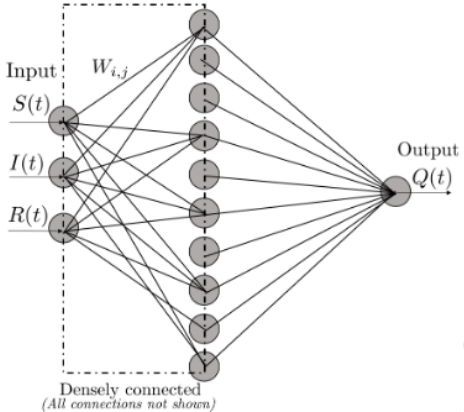
Italy



Spain



Italy



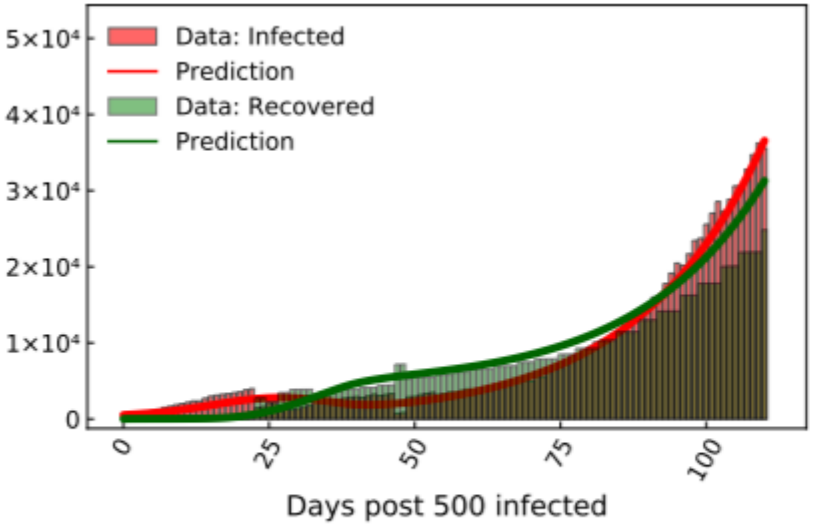
The QSIR Learns A Simplified SIR With Quarantine, and Quarantine Predictions are Within Days of Reported Changes

QSIR Counterfactuals: How Many Unnecessary Deaths in the Southern US?

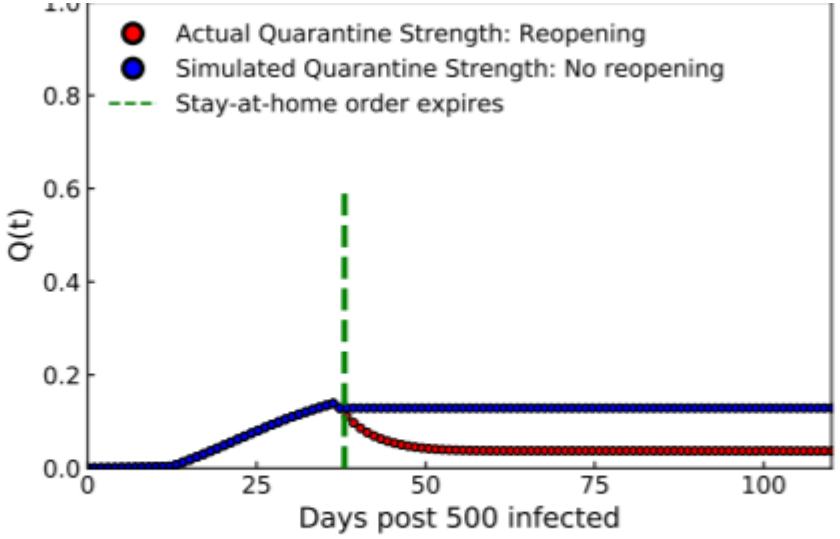
Verified QSIR now let's us ask questions: what if Q(t) didn't change?

Table 2. Infected count reduction by 14 July, 2020, if states had not reopened early, as estimated by our model.

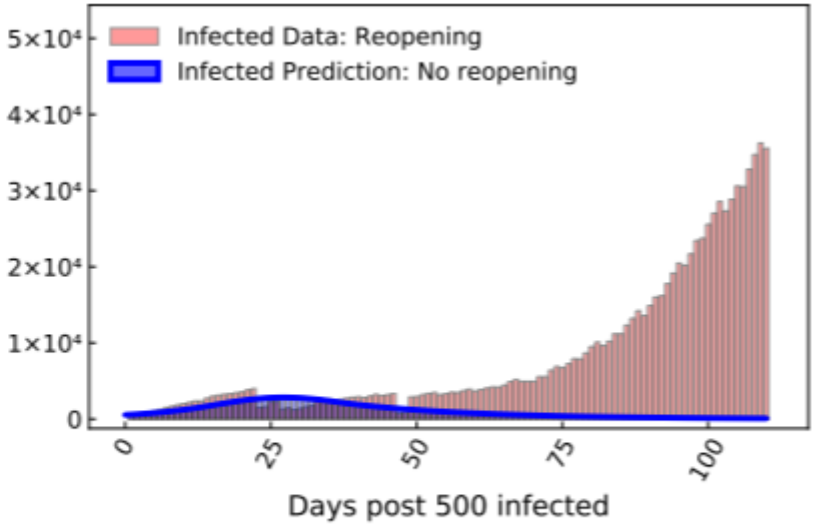
State	% decrease range (5% - 95% quantiles)	Mean % decrease	Case reduction range	Mean case reduction
1. Arizona	35 – 62	49	44000 – 79000	63000
2. Florida	20 – 75	49	57000 – 218000	144000
3. Louisiana	37 – 50	44	31000 – 41000	36000
4. Nevada	32 – 68	51	10000 – 20000	15000
5. Oklahoma	46 – 69	58	10000 – 15000	13000
6. South Carolina	83 – 86	84	50000 – 52000	51000
7. Tennessee	41 – 53	47	27000 – 36000	31000
8. Texas	41 – 51	46	115000 – 143000	129000
9. Utah	35 – 47	41	11000 – 14000	12000



(g) South Carolina



(h)



(i)

The UDE formulation fairly generally allows for imposing prior known structure

Discretized PDE Operators are Convolutions

$$\frac{u(x+\Delta x, y) - 2u(x,y) + u(x-\Delta x, y)}{\Delta x^2} + \frac{u(x, y+\Delta y) - 2u(x,y) + u(x-x, y-\Delta y)}{\Delta y^2}$$

$1_{\times 1}$	$1_{\times 0}$	$1_{\times 1}$	0	0
$0_{\times 0}$	$1_{\times 1}$	$1_{\times 0}$	1	0
$0_{\times 1}$	$0_{\times 0}$	$1_{\times 1}$	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved Feature

Is equivalent to the stencil

$$\begin{matrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$$

$$\frac{u(x+\Delta x) - 2u(x) + u(x-\Delta x)}{\Delta x^2} = u''(x) + \mathcal{O}\left(\Delta x^2\right)$$

$$\Delta u = u_{xx} + u_{yy}$$

Automatically Learning PDEs from Data: Universal PDEs for Fisher-KPP

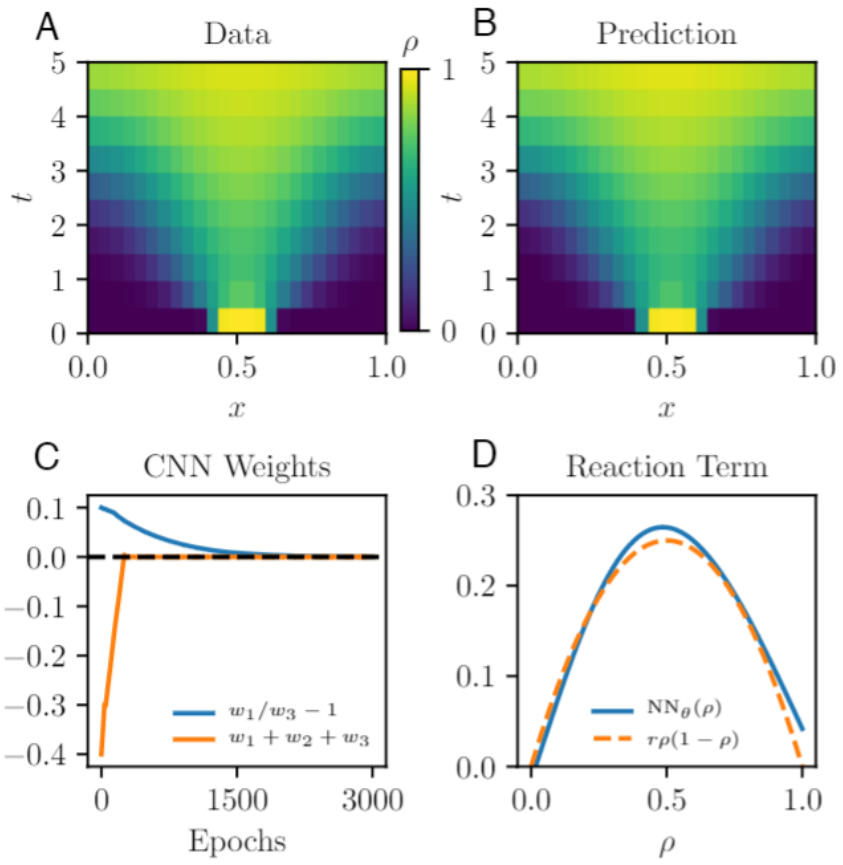
Truth: Fisher-KPP Equations

$$\rho_t = r\rho(1-\rho) + D\rho_{xx},$$

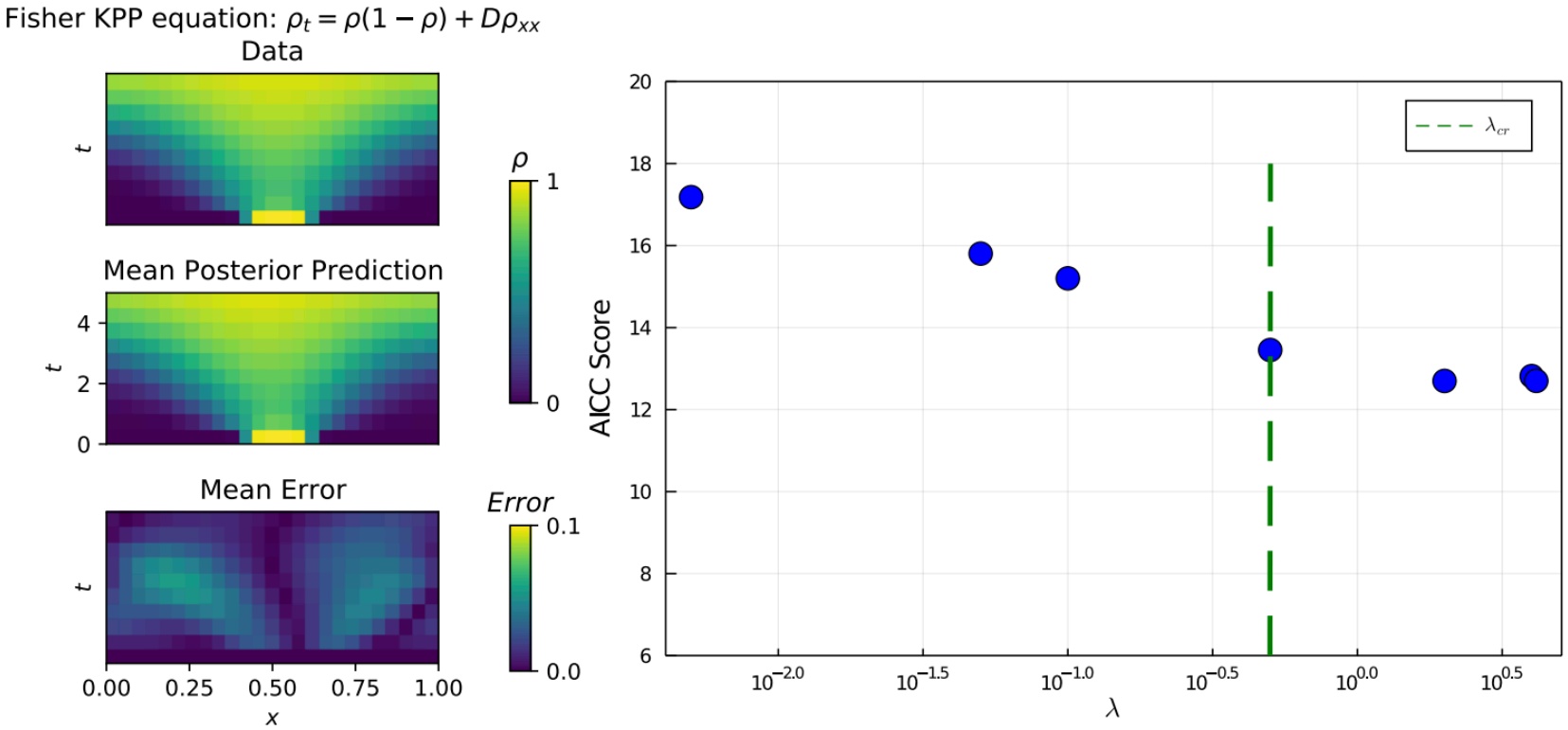
Truth: Universal Differential Equation

$$\rho_t = \mathrm{NN}_\theta(\rho) + D\,\mathrm{CNN}(\rho),$$

Automatically recover that the dynamical system has a diffusion operator and a quadratic reaction term!



Bayesian Universal Differential Equations for PDEs



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

UDEs Effectively Recover Nonlinearities of Epidemic Models

The baseline case:

$$\frac{\mathrm{d}S(t)}{\mathrm{d}t} = -\frac{\tau_{SI}\, S(t)\, I(t)}{N}$$
$$\frac{\mathrm{d}I(t)}{\mathrm{d}t} = \frac{\tau_{SI}\, S(t)\, I(t)}{N} - \tau_{IR} I(t) - \tau_{ID} I(t)$$
$$\frac{\mathrm{d}R(t)}{\mathrm{d}t} = \tau_{IR} I(t)$$
$$\frac{\mathrm{d}D(t)}{\mathrm{d}t} = \tau_{ID} I(t).$$

Replacement of all terms with neural networks:

$$\frac{\mathrm{d}S(t)}{\mathrm{d}t} = -NN_{SI}$$
$$\frac{\mathrm{d}I(t)}{\mathrm{d}t} = NN_{SI} - NN_{IR} - NN_{ID}$$
$$\frac{\mathrm{d}R(t)}{\mathrm{d}t} = NN_{IR}$$
$$\frac{\mathrm{d}D(t)}{\mathrm{d}t} = NN_{ID}$$

Use SciML knowledge to constrain the interaction graph, but learn the nonlinearities!

	Actual Equations	SINDY Active terms	SINDY Equations	Minimum AICC
NN_{SI}	0.85 S I	1: SI	0.74 S I	14
NN_{IR}	0.1 I	1: I	0.097 I	19
NN_{ID}	0.05 I	1: I	0.049 I	21

Table 4: SIRD: SINDY Recovered terms

Bayesian Chemical Reaction Neural Network

B-CRNN learns reaction networks from time course data and quantifies uncertainty in learned network

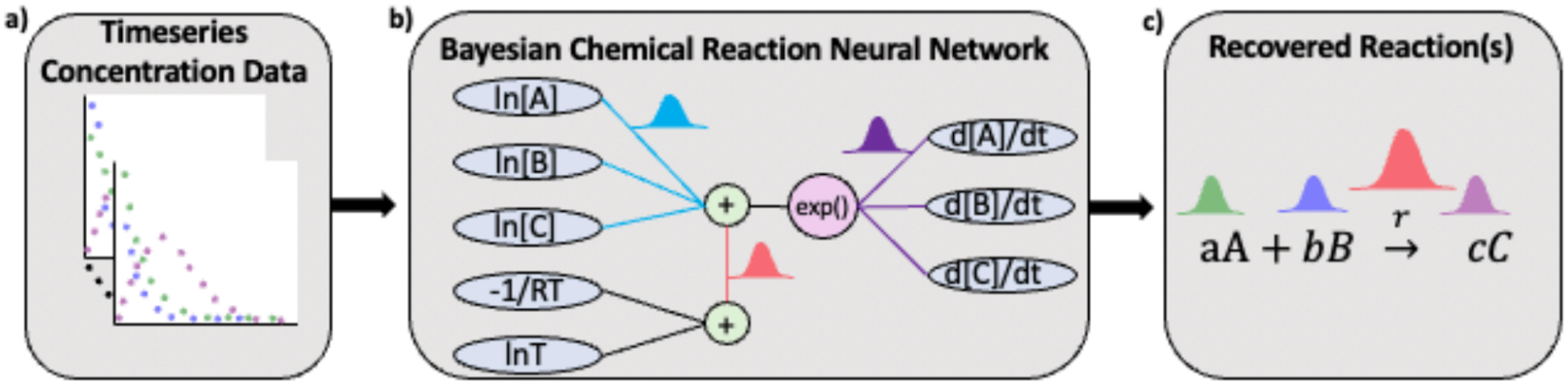


Figure 2. Overview of the B-CRNN which uses time course concentration data (a) to train a constrained neural network (b) that uses a preconditioned SGLD optimizer to reconstruct the reaction network and estimate the uncertainty in the learned stoichiometry and reaction rates (c).

Bayesian Chemical Reaction Neural Network

B-CRNN describes uncertainty in learned reaction rates

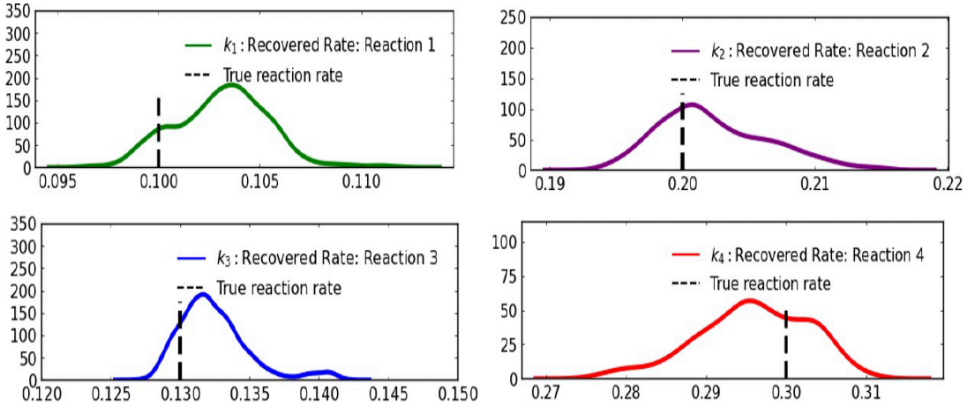
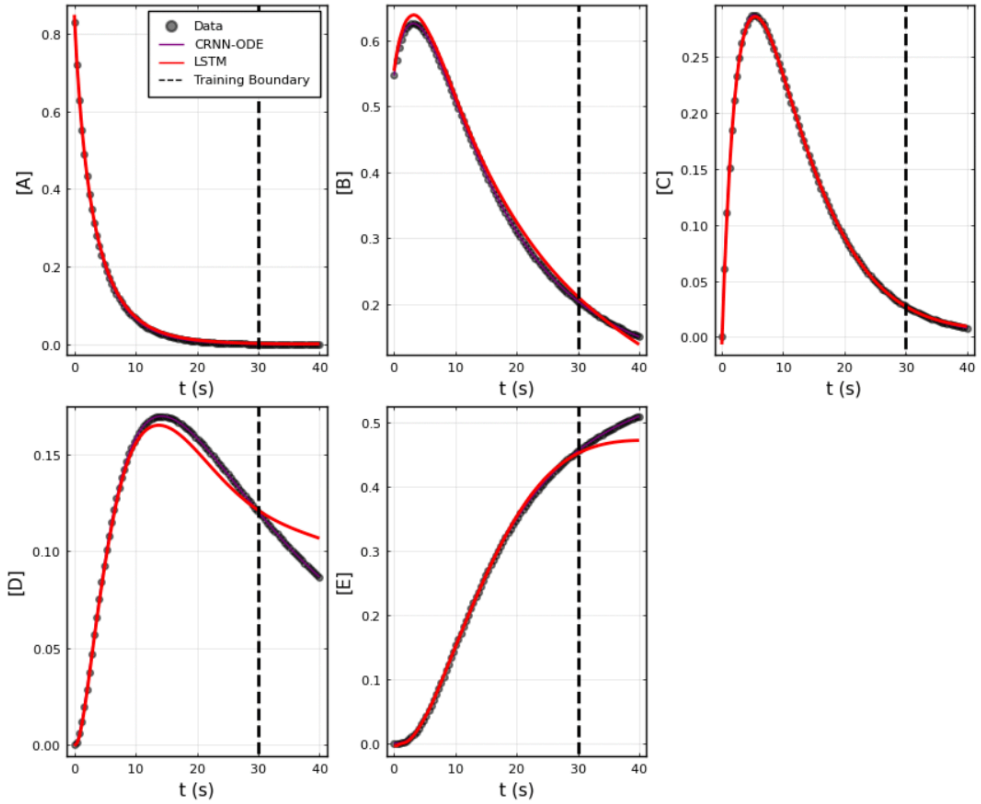


Figure 3. Posterior distribution of learned reaction rates for the four reactions included in table 1. Vertical dashed lines are true rates.

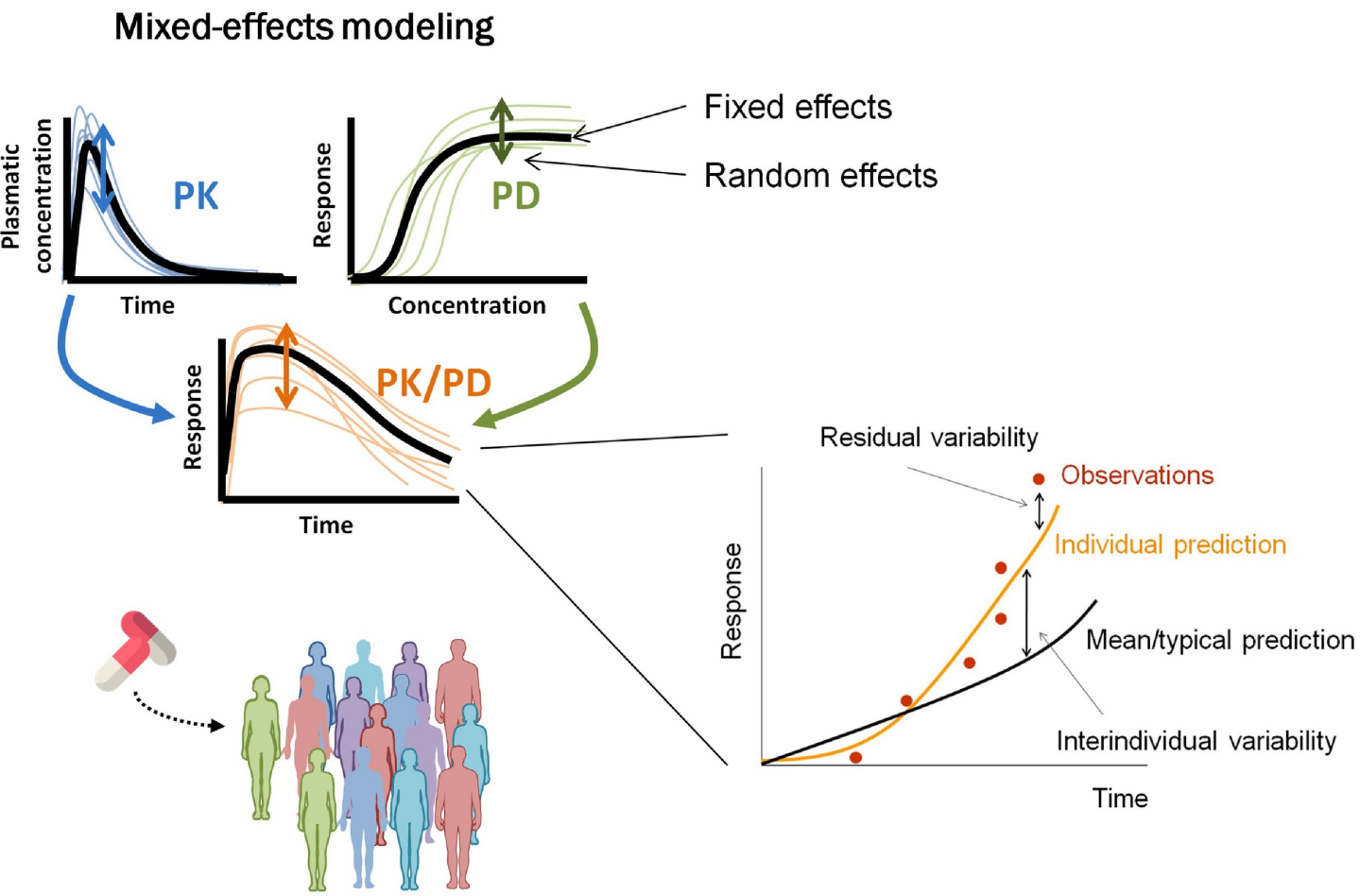
B-CRNN can extrapolate beyond training region, purely data-driven ML cannot



What’s something that’s not quite “just an ODE” where the UDE technique can give an equation discovery method?

DeepNLME: Integrate neural networks into traditional NLME modeling
DeepNLME is SciML-enhanced modeling for clinical trials

DeepNLME is SciML-enhanced modeling for clinical trials



- Automate the discovery of predictive covariates and their relationship to dynamics
- Automatically discover dynamical models and assess the fit
- Incorporate big data sources, such as genomics and images, as predictive covariates

Trends in Pharmacological Sciences

From Dynamics to Nonlinear Mixed Effects (NLME) Modeling

Goal: Learn to predict patient behavior (dynamics) from simple data (covariates)

$$Z_i = \begin{bmatrix} wt_i, \\ sex_i, \end{bmatrix}$$

Covariates

Math: Find (θ, η) such that $\mathrm{E}\left[\eta\right] = 0$

Requires special fitting procedures (Pumas)

$$g_i = \begin{bmatrix} Ka \\ CL \\ V \end{bmatrix} = \begin{bmatrix} \theta_1 e^{\eta_{i,1}\kappa_{i,k,1}}, \\ \theta_2 (\frac{wt_i}{70})^{0.75} \theta_4^{sex_i} e^{\eta_{i,2}}, \\ \theta_3 e^{\eta_{i,3}}, \end{bmatrix}$$

Structural Model (pre)

$$\frac{d[\mathrm{Depot}]}{dt} = -Ka[\mathrm{Depot}],$$
$$\frac{d[\mathrm{Central}]}{dt} = Ka[\mathrm{Depot}] - \frac{CL}{V}[\mathrm{Central}].$$

Dynamics

Intution: η (the random effects) are a fudge factor

Find θ (the fixed effect, or average effect) such that you can predict new patient dynamics as good as possible

The Impact of Pumas (PharmacUtical Modeling And Simulation)

“We have been using Pumas software for our pharmacometric needs to support our development decisions and regulatory submissions.

Pumas software has surpassed our expectations on its accuracy and ease of use. We are encouraged by its capability of supporting different types of pharmacometric analyses within one software. **Pumas has emerged as our "go-to" tool for most of our analyses in recent months.** We also work with Pumas-AI on drug development consulting. We are impressed by the quality and breadth of the experience of Pumas-AI scientists in collaborating with us on modeling and simulation projects across our pipeline spanning investigational therapeutics and vaccines at various stages of clinical development

Husain A. PhD (2020)
Director, Head of Clinical Pharmacology and Pharmacometrics,
Moderna Therapeutics, Inc

moderna™

messenger therapeutics

Built on SciML

From Dynamics to Nonlinear Mixed Effects (NLME) Modeling

Goal: Learn to predict patient behavior (dynamics) from simple data (covariates)

$$Z_i = \begin{bmatrix} wt_i, \\ sex_i, \end{bmatrix}$$

Covariates

Math: Find (θ, η) such that $\mathrm{E}[\eta] = 0$

$$g_i = \begin{bmatrix} Ka \\ CL \\ V \end{bmatrix} = \begin{bmatrix} \theta_1 e^{\eta_{i,1}\kappa_{i,k,1}}, \\ \theta_2 (\frac{wt_i}{70})^{0.75} \theta_4^{sex_i} e^{\eta_{i,2}}, \\ \theta_3 e^{\eta_{i,3}}, \end{bmatrix}$$

Structural Model (pre)

How can we find these models?

$$\frac{d[\text{Depot}]}{dt} = -Ka[\text{Depot}],$$
$$\frac{d[\text{Central}]}{dt} = Ka[\text{Depot}] - \frac{CL}{V}[\text{Central}].$$

Dynamics

Intution: η (the random effects) are a fudge factor

Find θ (the fixed effect, or average effect) such that you can predict new patient dynamics as good as possible

From Dynamics to Nonlinear Mixed Effects (NLME) Modeling

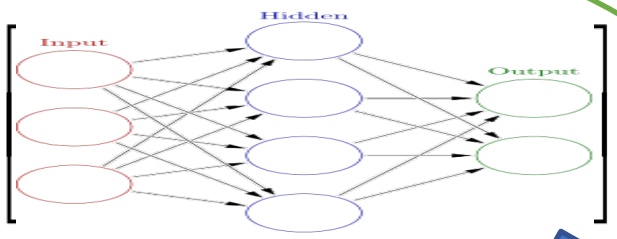
Goal: Learn to predict patient behavior (dynamics) from simple data (covariates)

$$Z_i = \begin{bmatrix} wt_i, \\ sex_i, \end{bmatrix}$$

Covariates

Math: Find (θ, η) such that $\mathrm{E}\left[\eta\right] = 0$

$$g_i = \begin{bmatrix} Ka \\ CL \\ V \end{bmatrix} =$$

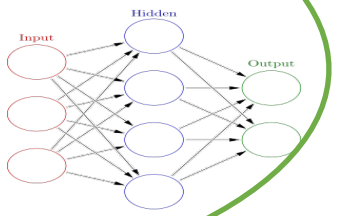


Structural Model (pre)

How can we find these models?

$$\frac{d[\text{Depot}]}{dt} = -Ka[\text{Depot}],$$

$$\frac{d[\text{Central}]}{dt} = Ka[\text{Depot}] -$$

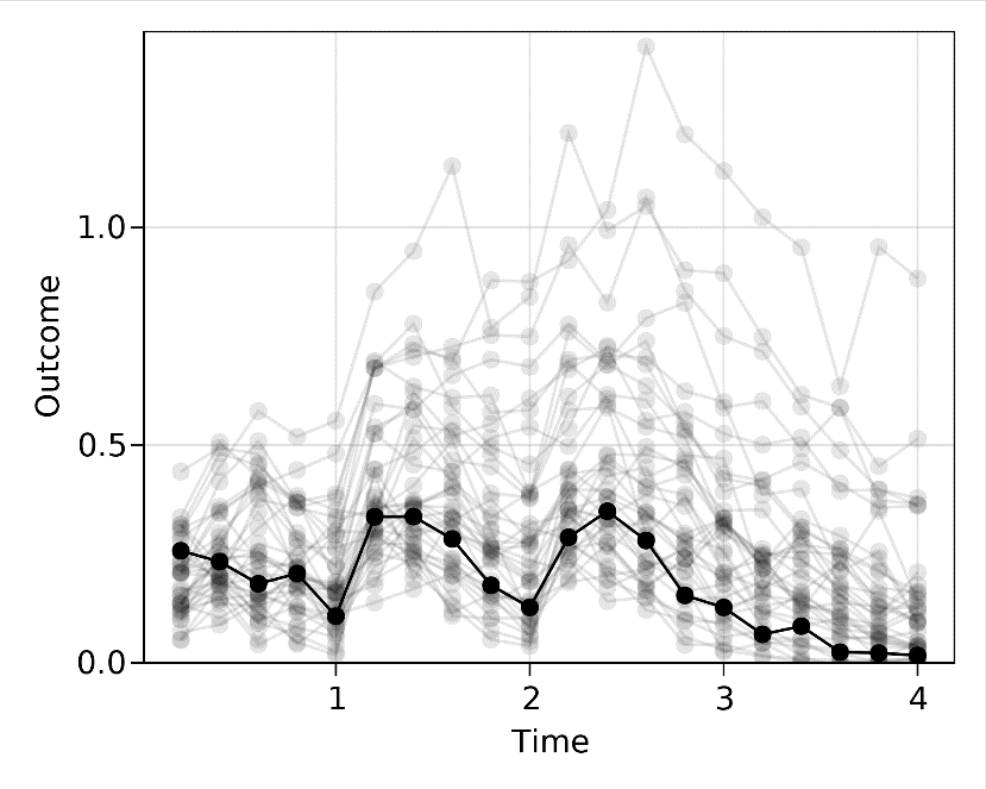


Dynamics

Idea: Parameterize the model such that the models can be neural networks, where the weights of the neural networks are fixed effects!

Indirect learning of unknown functions!

From Dynamics to Nonlinear Mixed Effects (NLME) Modeling



Typical values

$$\theta \in \mathbb{R}_+^3$$
$$\Omega \in \mathbb{R}_+^3$$

Patient data

Age
Weight

Random effects

$$\eta \sim \text{MvNormal}\left(\Omega\right)$$

Individual parameters

$$\begin{aligned} Ka_i &= \theta_1 \cdot e^{\eta_{i,1}} + c_1 \cdot Age_i \quad + \\ CL_i &= \theta_2 \cdot e^{\eta_{i,2}} \\ V_i &= \theta_3 \cdot e^{\eta_{i,3}} + c_2 \cdot Weight_1^{c_3} \quad + \end{aligned}$$

Dynamics

$$\begin{aligned} \frac{d[\text{Depot}]}{dt} &= -Ka[\text{Depot}], \\ \frac{d[\text{Central}]}{dt} &= Ka[\text{Depot}] - \end{aligned}$$

Error model

$$Outcome \sim \text{Normal}\left(Central, \sqrt{Central} \cdot \sigma\right)$$

Therefore, any solver you can differentiate can do UDE things

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

$$u' = f(u, p, t)$$

$$du = f(u, p, t)dt + g(u, p, t)dW_t$$

$$\vdots$$

Optimization.jl: Unified Optimization Interface

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

Integrals.jl: Unified Quadrature Interface

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

Unified Partial Differential Equation Interface

$$u_t = u_{xx} + f(u)$$
$$u_{tt} = u_{xx} + f(u)$$
$$\vdots$$

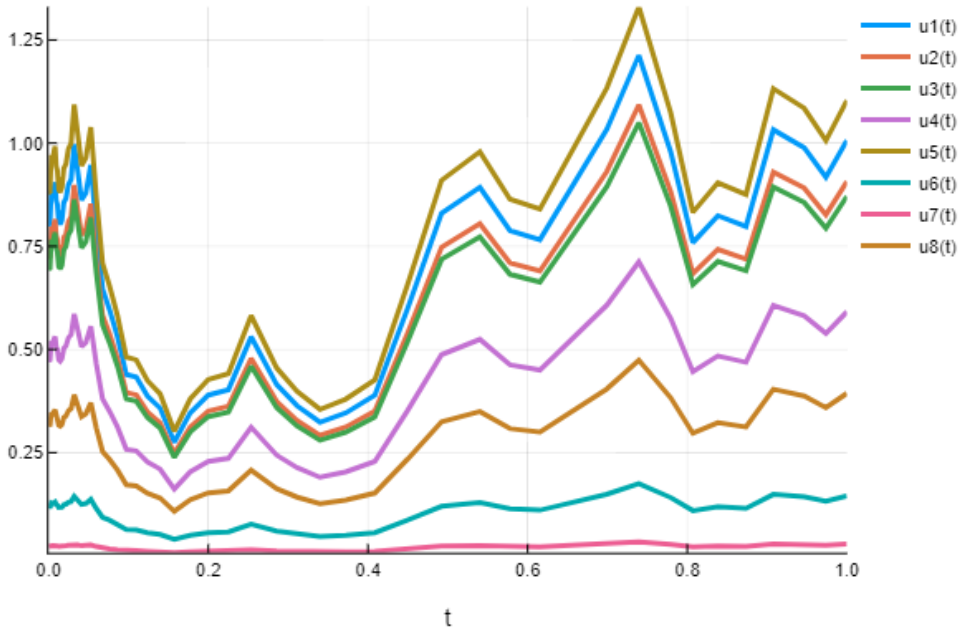
The SciML Common Interface for Julia Equation Solvers

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

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?

Unified Interfaces to Partial Differential Tooling

```julia
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])
```

Lots of Auto-Discretizers:

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

New SciML Docs: Comprehensive Documentation of Differentiable Simulation

HOME | MODELING | SOLVERS | ANALYSIS | MACHINE LEARNING | DEVELOPER TOOLS | Search...

EQUATION SOLVERS
- LinearSolve
- NonlinearSolve
- DifferentialEquations
- Integrals
- Optimization
- JumpProcesses

INVERSE PROBLEMS / ESTIMATION
- SciMLSensitivity
- DiffEqParamEstim
- DiffEqBayes

PDE SOLVERS
- MethodOfLines
- NeuralPDE
- NeuralOperators
- FEniCS
- HighDimPDE
- DiffEqOperators

THIRD-PARTY PDE SOLVERS
- Trixi
- Gridap
- ApproxFun
- VoronoiFVM

of the highest performance and parallel implementations one can find.

Scientific Machine Learning (SciML) = Scientific Computing + Machine Learning

Where to Start?

- Want to get started running some code? Check out the Getting Started tutorials.
- What is SciML? Check out our Overview.
- Want to see some cool end-to-end examples? Check out the Extended Tutorials.
- Curious about our performance claims? Check out the SciML Open Benchmarks.

Version v0.2

SciML Interface Coverage is Growing: Bringing AD to All Solvers by Default

LinearSolve.jl

1. SuiteSparse.jl (KLU, UMFPACK)
2. RecursiveFactorization.jl
3. Base.LinearAlgebra
4. FastLapackInterface.jl
5. Pardiso.jl
6. CUDA.jl (automated GPU offloading)
7. IterativeSolvers.jl
8. Krylov.jl
9. KrylovKit.jl

…
More keep being added (PETSc, Magma, HSL, Hypre, Elemental, CuSolverRF, …)

NonlinearSolve.jl

1. NLsolve.jl (KLU, UMFPACK)
2. SteadyStateDiffEq.jl
3. MINPACK
4. SUNDIALS (KINSOL)
5. New methods

…
More keep being added (PETSc, SpeedMapping.jl, etc.)

SciML Interface Coverage is Growing: Bringing AD to All Solvers by Default

Integrals.jl

1. QuadGK.jl
2. Cuba.jl
3. Cubature.jl
4. Hcubature.jl
5. MonteCarloIntegration.jl

Optimization.jl

Overview of the Optimizers

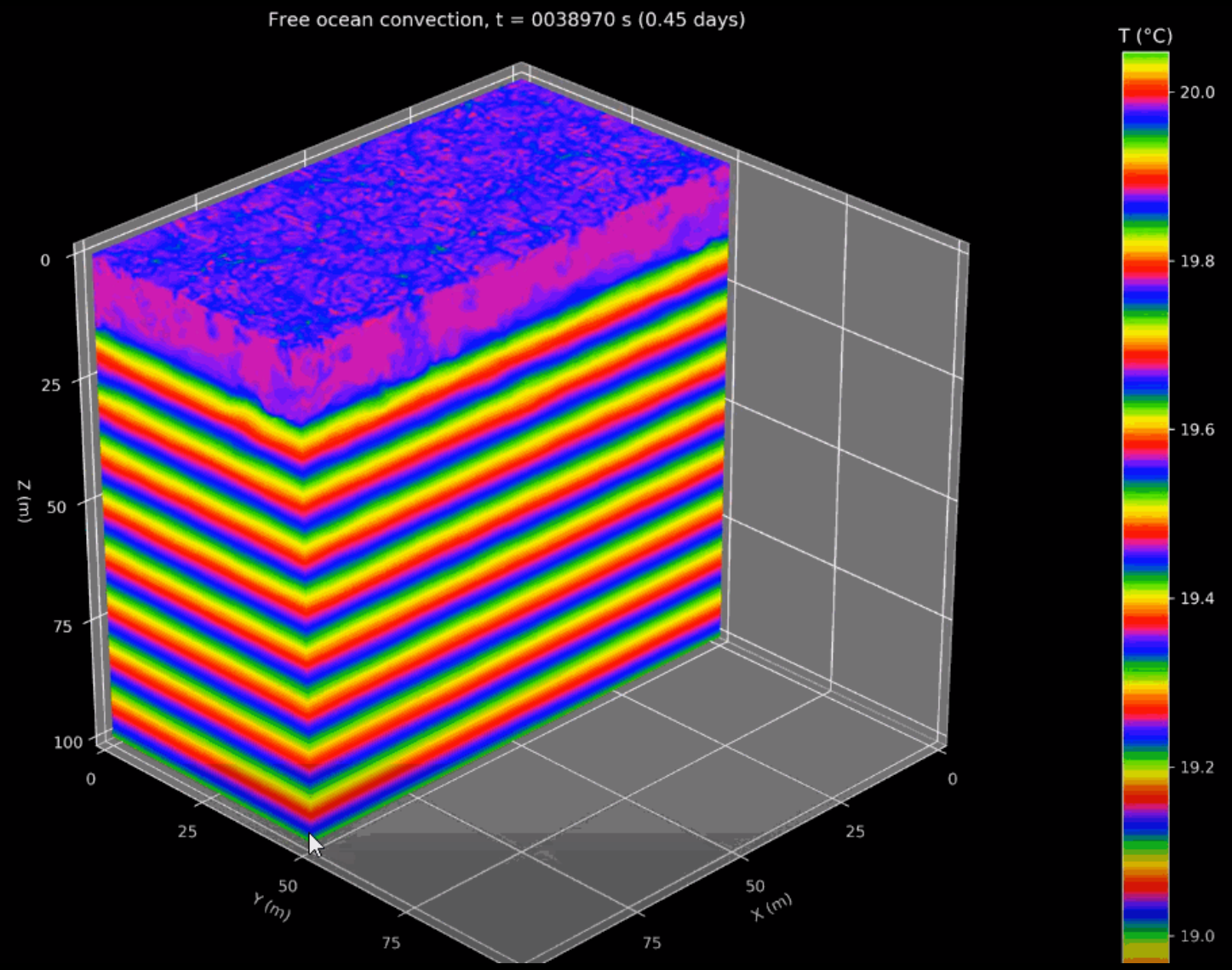
Package	Local Gradient-Based	Local Hessian-Based	Local Derivative-Free	Local Constrained	Global Unconstrained	Global Constrained
BlackBoxOptim	✗	✗	✗	✗	✓	✗
CMAEvolutionaryStrategy	✗	✗	✗	✗	✓	✗
Evolutionary	✗	✗	✗	✗	✓	●
Flux	✓	✗	✗	✗	✗	✗
GCMAES	✗	✗	✗	✗	✓	✗
MathOptInterface	✓	✓	✓	✓	✓	●
MultistartOptimization	✗	✗	✗	✗	✓	✗
Metaheuristics	✗	✗	✗	✗	✓	●
NOMAD	✗	✗	✗	✗	✓	●
NLopt	✓	✗	✓	●	✓	●
Nonconvex	✓	✓	✓	●	✓	●
Optim	✓	✓	✓	✓	✓	✓
QuadDIRECT	✗	✗	✗	✗	✓	✗

Does doing such methods require differentiation of the simulator?

High fidelity surrogates of ocean columns for climate models

3D simulations are high resolution but too expensive.

Can we learn faster models?



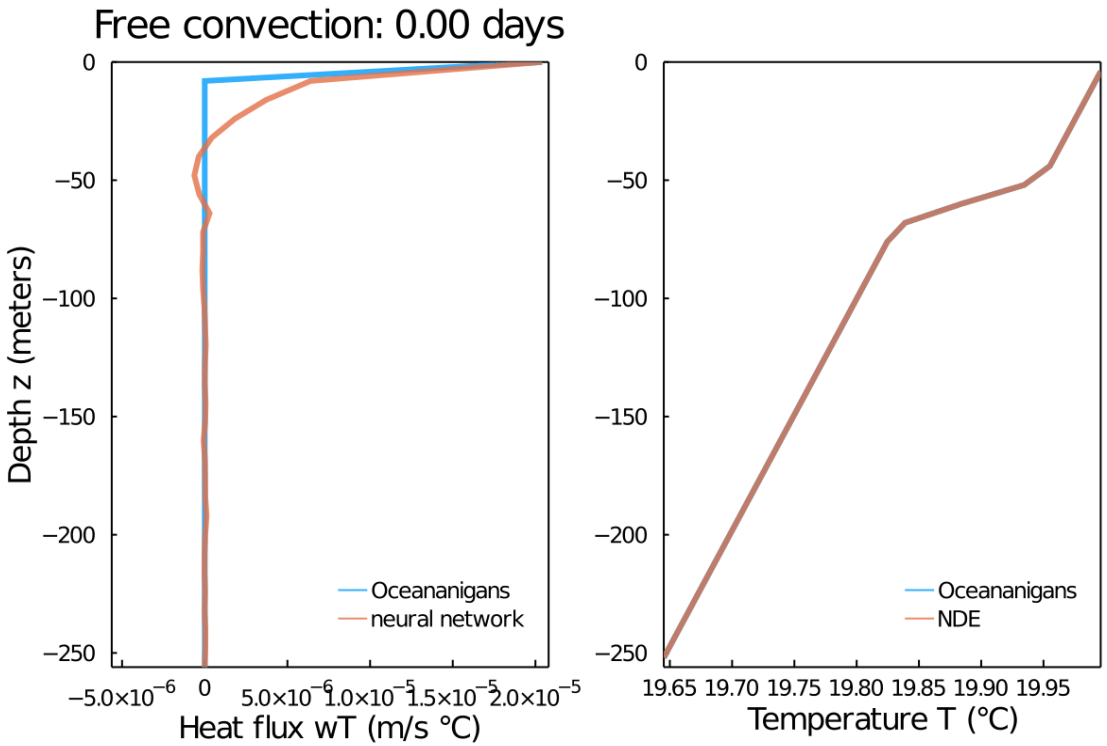
Neural Networks Infused into Known Partial Differential Equations

Derive a 1D approximation to the 3D model

$$\frac{\partial T}{\partial t} = -\frac{\partial}{\partial z}\underbrace{\left(\text{NN} - K\frac{\partial T}{\partial z}\right)}_{\overline{w'T'}}$$

Incorporate the "convective adjustment"

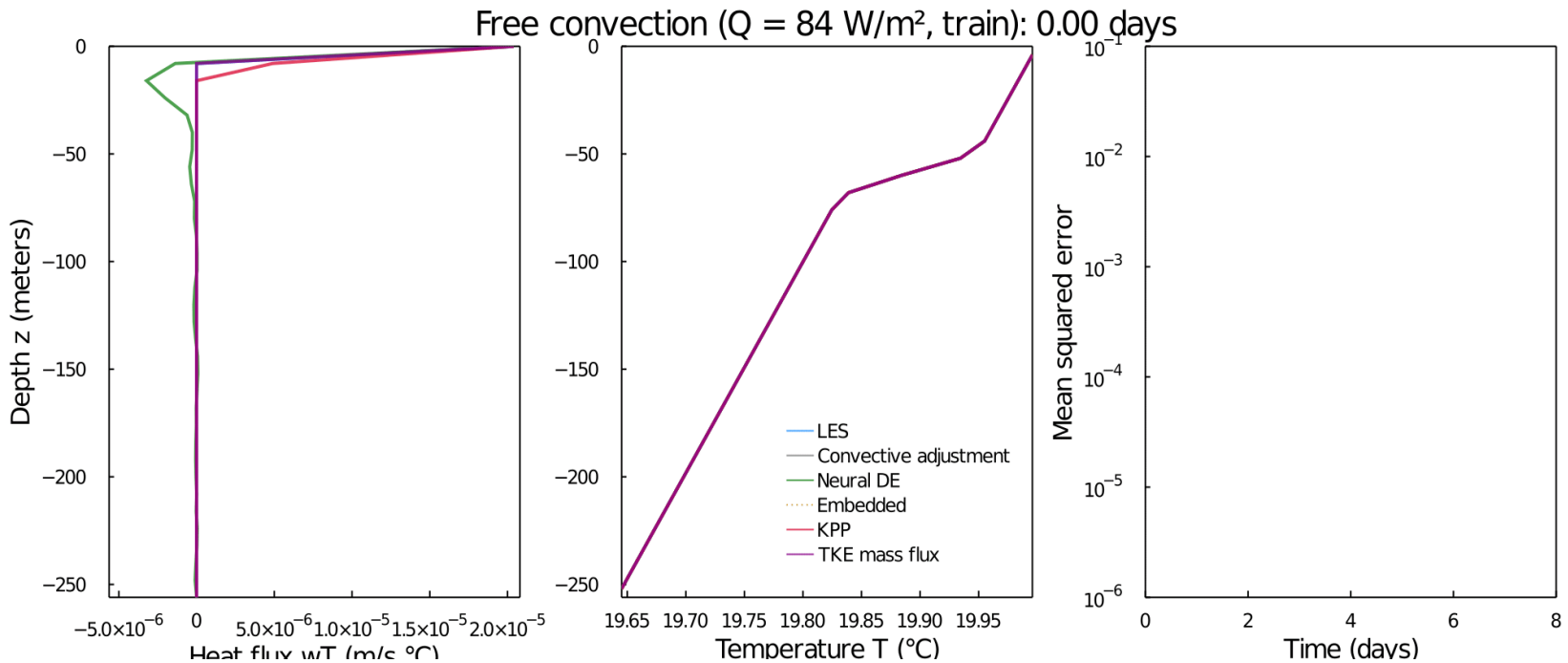
$$K = \begin{cases} 0 & \text{if } \partial_z T > 0 \\ 100\ \mathrm{m}^2/\mathrm{s} & \text{if } \partial_z T < 0 \end{cases}$$



$$loss(T, wT) = |NN(T) - wT|^2$$

Only okay, **but why?**

Good Engineering Principles: Integral Control!



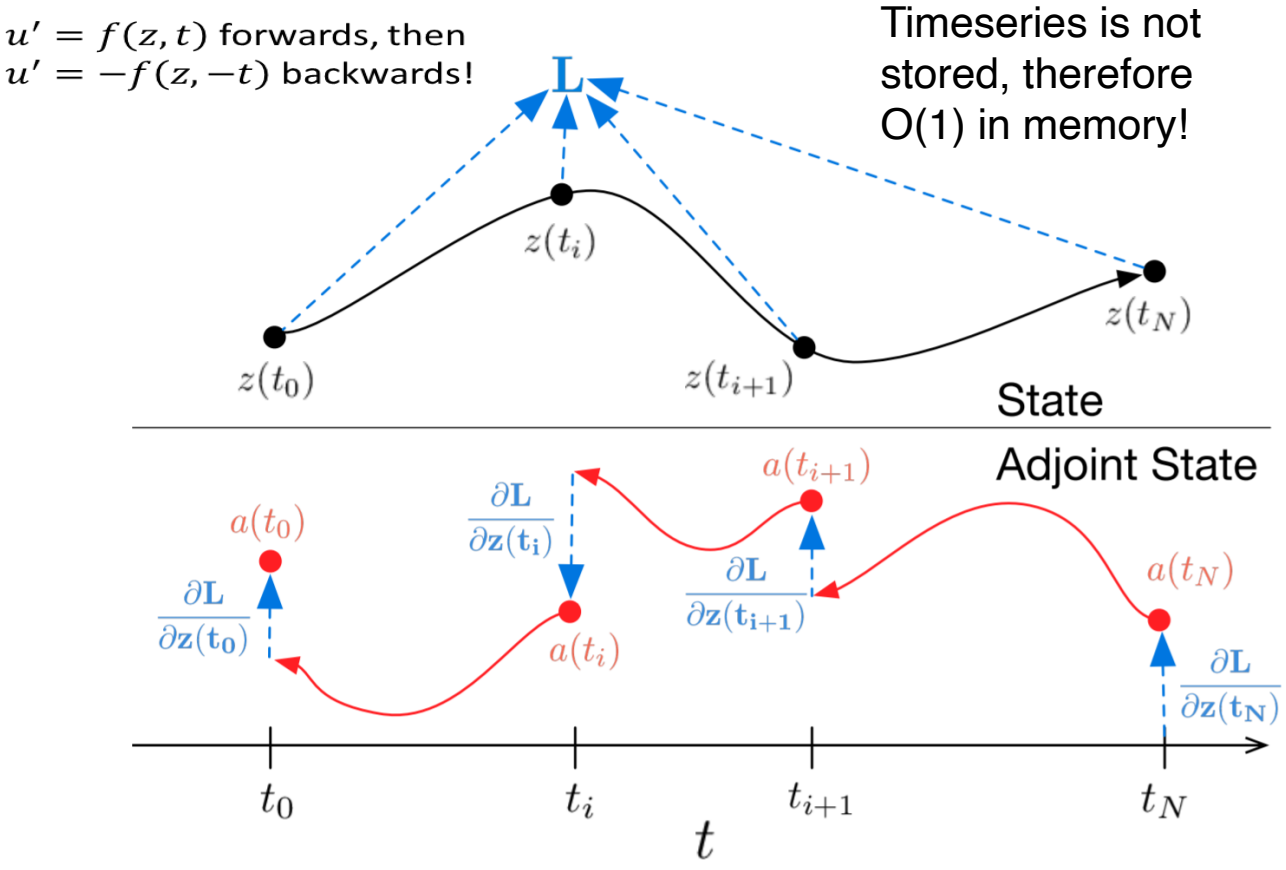
$$\frac{\partial T}{\partial t} = -\frac{\partial}{\partial z}\underbrace{\left(\text{[neural network]} - K\frac{\partial T}{\partial z}\right)}_{\overline{w'T'}}$$

$$loss(T_{NN}, T) = |T_{NN}(z,t) - T(z,t)|^2$$

But how do you fit a neural network inside of a simulator?

Part 1: Differentiation of Solvers

Machine Learning Neural Ordinary Differential Equations



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

The adjoint equation is an ODE!

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

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

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

Chen, Ricky TQ, et al. "Neural ordinary differential equations." Advances in neural information processing systems. 2018.

But… really?

Differentiating Ordinary Differential Equations: The Trick

We with 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

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

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:

$$\int_{t_0}^{T} \lambda^* \left(s' - f_u s - f_p\right) dt = \int_{t_0}^{T} \lambda^* s' dt - \int_{t_0}^{T} \lambda^* \left(f_u s - f_p\right) dt$$

$$= \left|\lambda^*(t) s(t)\right|_{t_0}^{T} - \int_{t_0}^{T} \lambda^{*\prime} s dt - \int_{t_0}^{T} \lambda^* \left(f_u s - f_p\right) dt$$

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

$$\frac{dG}{dp} = \int_{t_0}^{T} (g_p + g_u s) dt + \left|\lambda^*(t) s(t)\right|_{t_0}^{T} - \int_{t_0}^{T} \lambda^{*\prime} s dt - \int_{t_0}^{T} \lambda^* \left(f_u s - f_p\right) dt$$

$$= \int_{t_0}^{T} (g_p + \lambda^* f_p) dt + \left|\lambda^*(t) s(t)\right|_{t_0}^{T} - \int_{t_0}^{T} \left(\lambda^{*\prime} + \lambda^* f_u - g_u\right) s dt$$

Differentiating Ordinary Differential Equations: The Final Form

$$\frac{dG}{dp} = \int_{t_0}^{T} (g_p + \lambda^* f_p) dt + |\lambda^*(t) s(t)|_{t_0}^{T} - \int_{t_0}^{T} (\lambda^{*\prime} + \lambda^* f_u - g_u)\, s dt$$

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 $\lambda'^{(t)} = -\frac{df}{du^{(t)}}^{*} \lambda^{(t)} - \left(\frac{dg}{du^{(t)}}\right)^{*}$

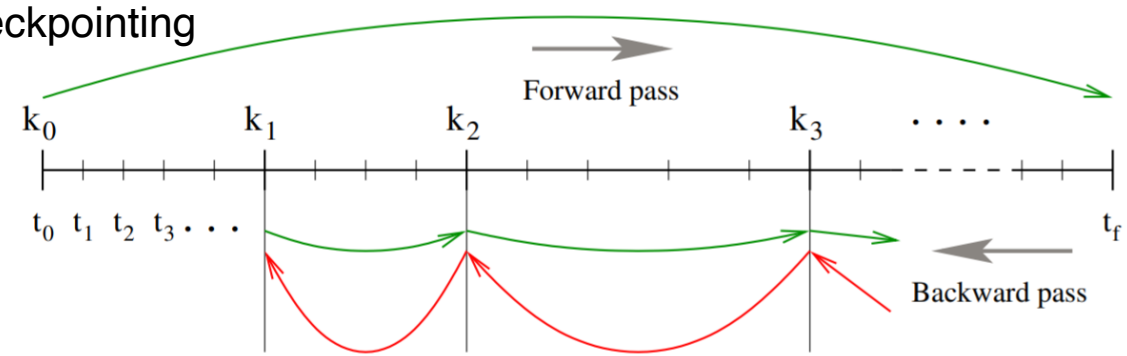
$$\lambda(T) = 0$$

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) -> 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?

Cool. Can this go wrong?

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

Advection Equation:

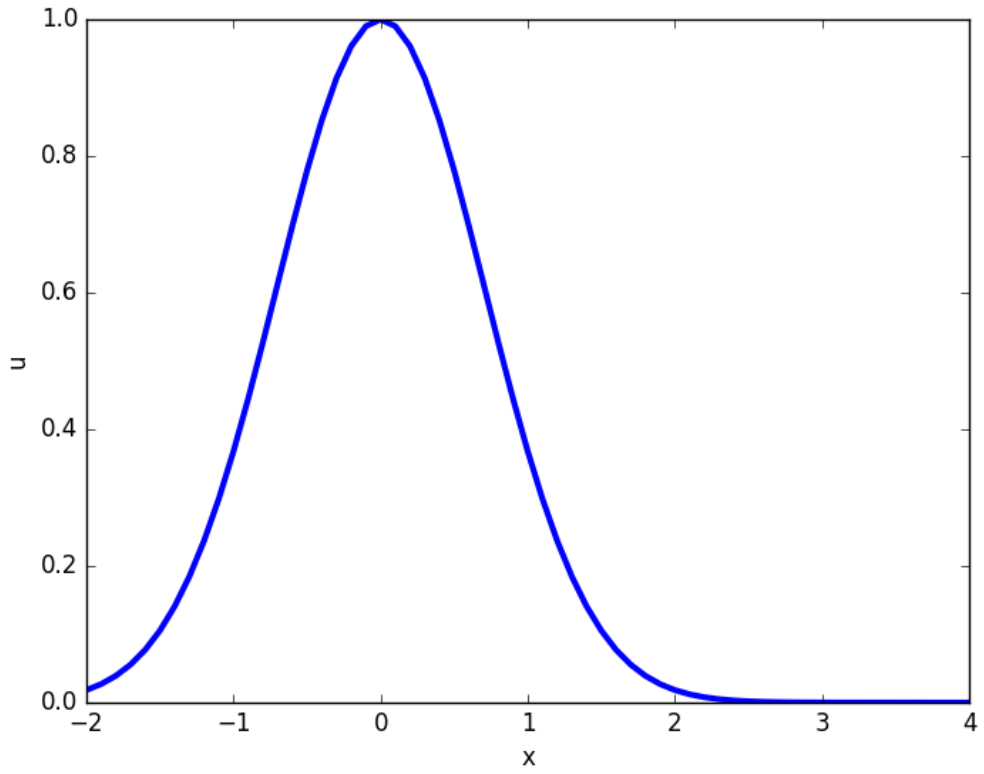
$$\frac{\partial u}{\partial t} + \frac{a(\partial u)}{\partial x} = 0$$

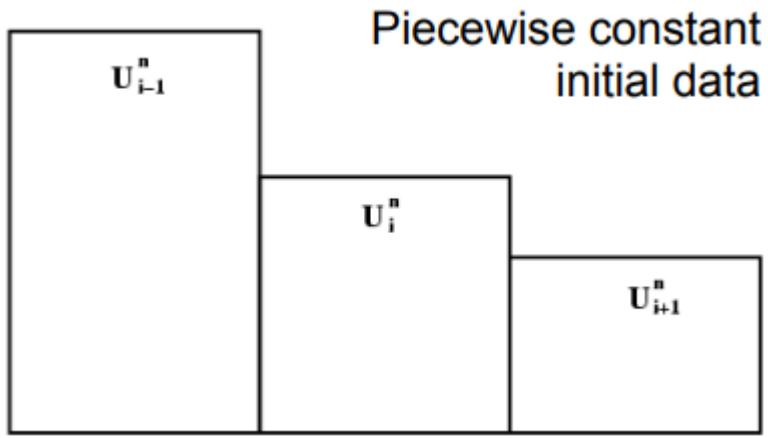
Approximating the derivative in ***X*** has two choices: forwards or backwards

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

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.

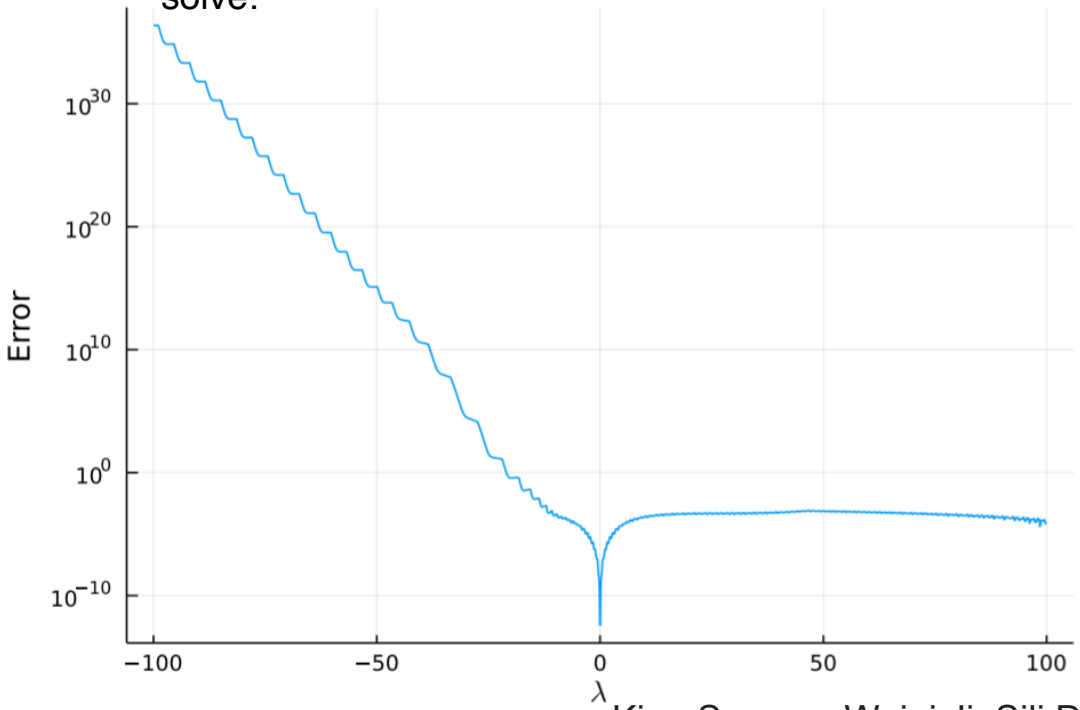




Problems With Naïve Adjoint Approaches On Stiff Equations

Error grows exponentially…

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

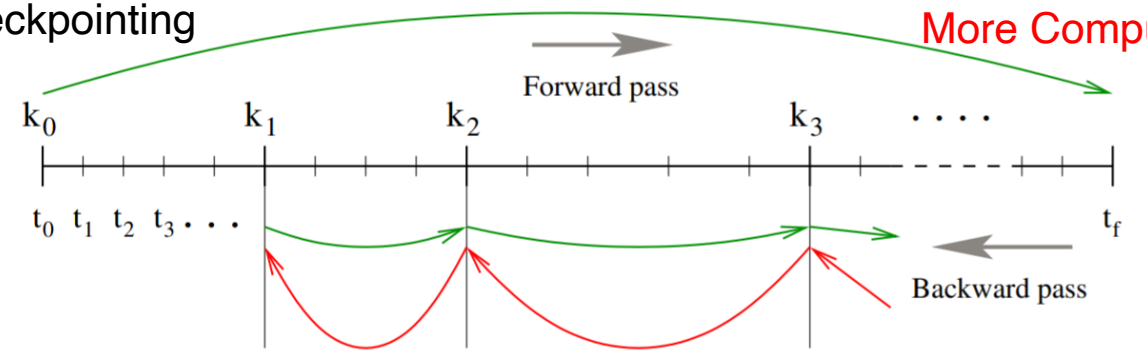


How do you get u(t) while solving backwards?
3 options!

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



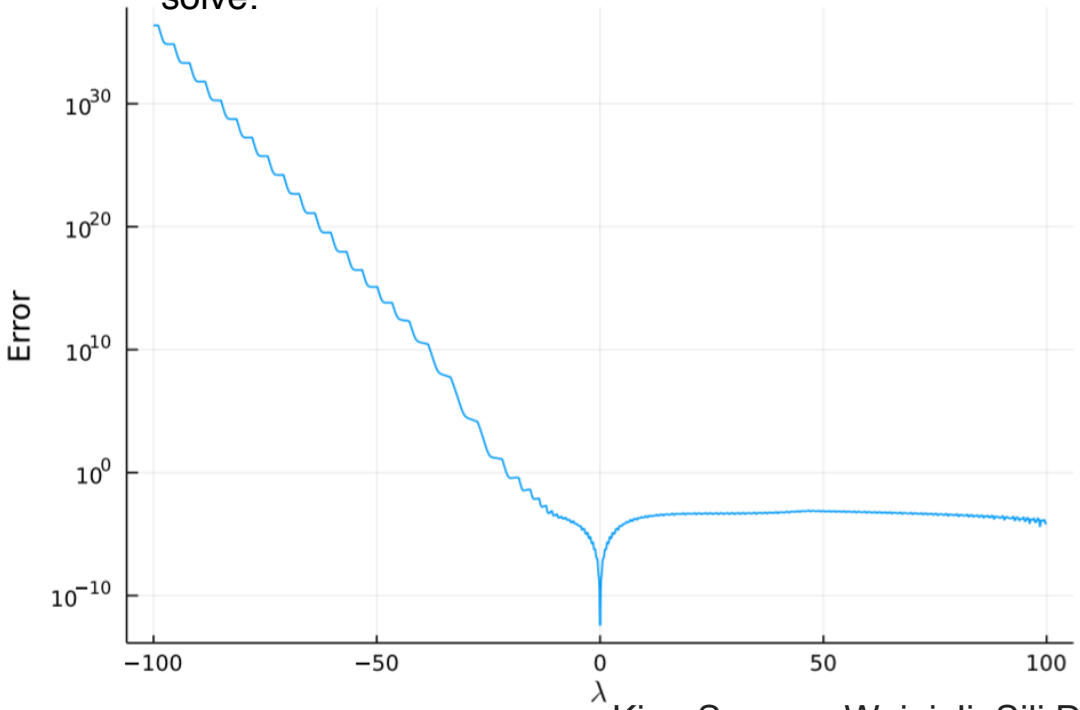
Each choices has an engineering trade-off!

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:



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:

$$O\left(\left(states + parameters\right)^3\right)$$

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

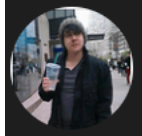
But automatic differentiation

How does it work, and does it fix the problem?

Symbolic Differentiation on Code

Does Zygote differentiate symbolically?

Specific Domains · Machine Learning · question · zygote · differentiation

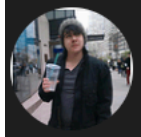


```julia
using Symbolics
@variables x
function f(x)
    out = one(x)
    for i in 1:5
        out *= x^i
    end
    out
end
sin(f(x)) # sin(x^15)
```

Evaluation with symbolic variables completely removes the “non-mathematical” computational expressions, and then we symbolically differentiate in this language:

```julia
Symbolics.derivative(sin(f(x)),x) # 15(x^14)*cos(x^15)
```

Automatic Differentiation as Differentiation in the Language of Code



```
function f(x)
    out = x
    for i in 1:5
        out *= sin(out)
    end
    out
end
sin(f(x)) # sin(x*sin(x)*sin(x*sin(x))*sin(x*sin(x)*sin(x*sin(x)))*sin(x*sin
```

```
Symbolics.derivative(sin(f(x)),x) # (sin(x)*sin(x*sin(x))*sin(x*sin(x)*sin(x
```

Automatic Differentiation as Differentiation in the Language of Code

On that same example, this looks like:

```
function f(x)
    out = x
    for i in 1:5
        # sin(out) => chain rule sin' = cos
        tmp = (sin(out[1]), out[2] * cos(out[1]))
        # out = out * tmp => product rule
        out = (out[1] * tmp[1], out[1] * tmp[2] + out[2] * tmp[1])
    end
    out
end
function outer(x)
    # sin(x) => chain rule sin' = cos
    out1, out2 = f(x)
    sin(out1), out2 * cos(out1)
end
dsinfx(x) = outer((x,1))[2]

f((1,1)) # (0.01753717849708632, 0.36676042682811677)
dsinfx(1) # 0.3667040292067162
```

More Details on the Algorithm, see the SciML Book:

book.sciml.ai

Chapter 10

What does automatic differentiation of an ODE solver give you?

Are there cases where that is mathematically correct but numerically incorrect?

Wrong gradient for some sensealgs #273

Closed anhi opened this issue on Jun 8, 2020 · 3 comments · Fixed by SciML/DiffEqBase.jl#529

anhi commented on Jun 8, 2020

We are currently experimenting with time dependent parameters, but the gradients often seem to come out wrong. For instance, this here is an artificially simple example for clarity:

```
using DiffEqSensitivity, OrdinaryDiffEq, Zygote

function get_param(breakpoints, values, t)
    for (i, tᵢ) in enumerate(breakpoints)
        if t <= tᵢ
            return values[i]
        end
    end

    return values[end]
end

function fiip(du, u, p, t)
    a = get_param([1., 2., 3.], p[1:4],  t)

    du[1] = dx =  a * u[1] - u[1] * u[2]
    du[2] = dy = -a * u[2] + u[1] * u[2]
end

p = [1., 1., 1., 1.]; u0 = [1.0;1.0]
prob = ODEProblem(fiip, u0, (0.0, 4.0), p);

Zygote.gradient(p->sum(concrete_solve(prob, Tsit5(), u0, p, sensealg = ForwardDiffSensitivity(), saveat = 0.1))
Zygote.gradient(p->sum(concrete_solve(prob, Tsit5(), u0, p, sensealg = ForwardSensitivity(), saveat = 0.1)), p
```

Assignees

No one—assig

Labels

None yet

Projects

None yet

Milestone

No milestone

Development

Successfully
issue.

Single sig
PumasAI

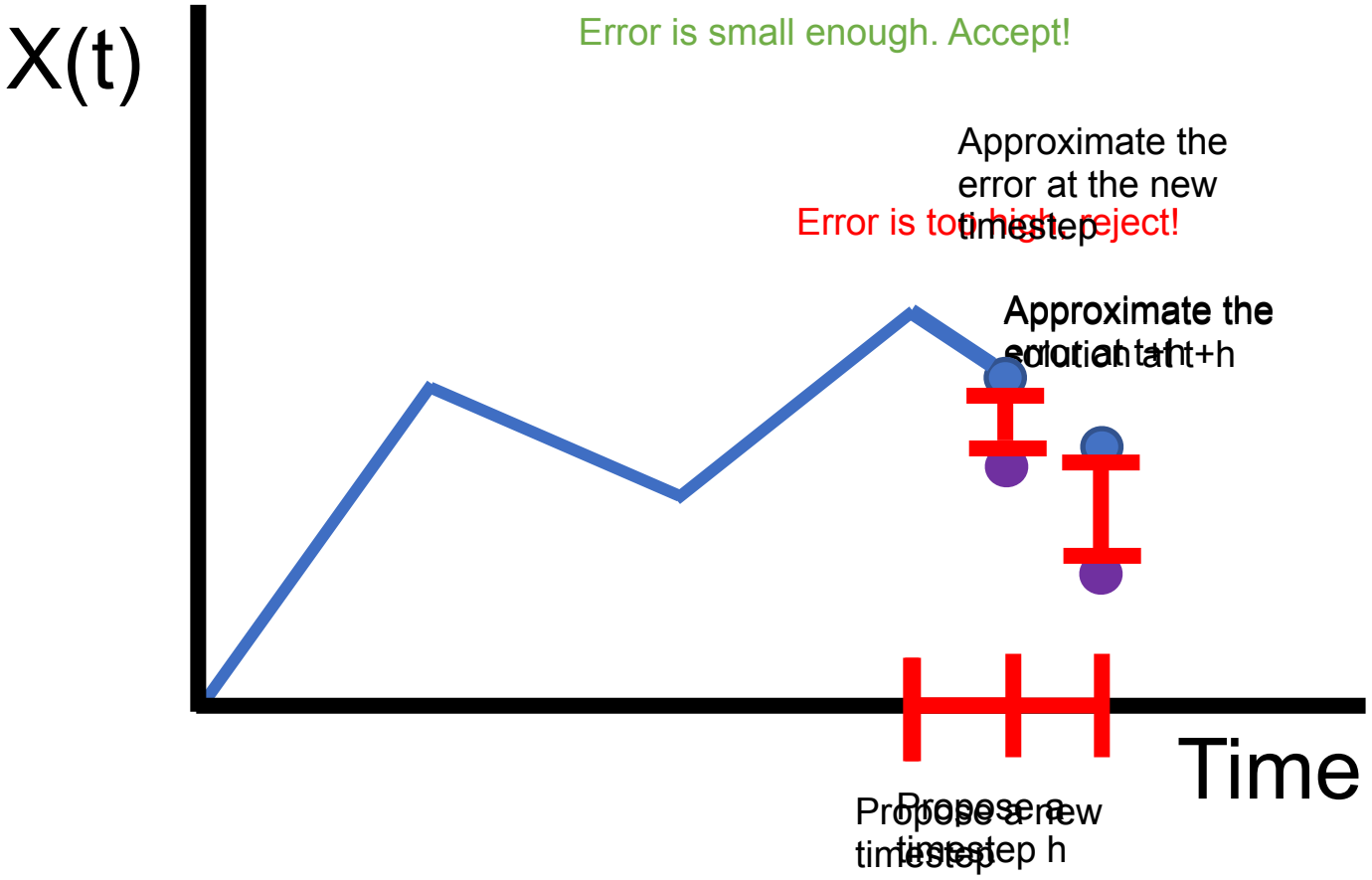
make dua
SciML/Dif

Notifications

Indeed, AD on its own gives the incorrect answer… but why?

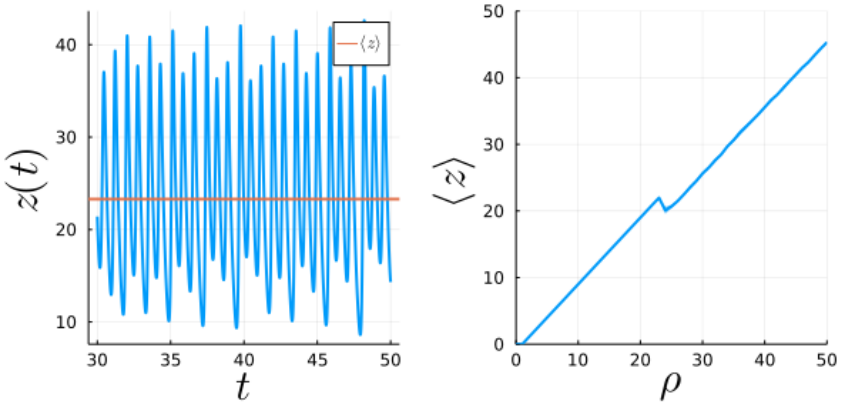
```
# Original AD
Zygote.gradient(p->sum(concrete_solve(prob, Tsit5(), u0, p, sensealg = ForwardDiffSensitivity(), saveat = 0.1, internalnorm = (u,t) -> sum
(abs2,u/length(u)), abstol=1e-12, reltol=1e-12)), p
) ([29.755582164326086, 10.206643764088689, 53.37700890093473, 3.5509327396481583],)

# Forward Sensitivity
Zygote.gradient(p->sum(concrete_solve(prob, Tsit5(), u0, p, sensealg = ForwardSensitivity(), saveat = 0.1, abstol=1e-12, reltol=1e-12)), p
) ([37.607133325673956, 35.92458894240918, 19.601050929858797, 3.6443048514269707],)

# Corrected AD
Zygote.gradient(p->sum(concrete_solve(prob, Tsit5(), u0, p, sensealg = ForwardDiffSensitivity(), saveat = 0.1, abstol=1e-12, reltol=1e-12)),
p) ([37.607133316972764, 35.92458895352116, 19.601050925013986, 3.644304853859423],)
```

How adaptivity works



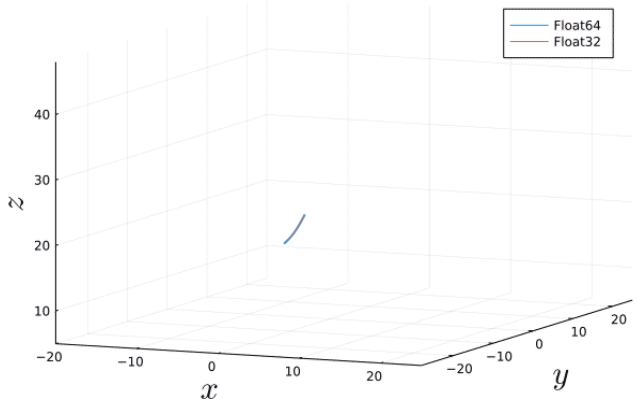
Any more cases where AD is incorrect?

Differentiation of Chaotic Systems: Shadow Adjoints



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

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



- AD and finite differencing fails!

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

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

- Shadowing methods in DiffEqSensitivity.jl

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

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

https://frankschae.github.io/post/shadowing/

Conclusion Part 1:

Be careful about how you compute derivatives of equation solvers

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

$$u' = f(u, p, t)$$

$$du = f(u, p, t)dt + g(u, p, t)dW_t$$

$$\vdots$$

Optimization.jl: Unified Optimization Interface

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

Integrals.jl: Unified Quadrature Interface

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

Unified Partial Differential Equation Interface

$$u_t = u_{xx} + f(u)$$
$$u_{tt} = u_{xx} + f(u)$$
$$\vdots$$

The SciML Common Interface for Julia Equation Solvers

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

New SciML Docs: Comprehensive Documentation of Differentiable Simulation

HOME MODELING SOLVERS ANALYSIS MACHINE LEARNING DEVELOPER TOOLS Search...

EQUATION SOLVERS
- LinearSolve
- NonlinearSolve
- DifferentialEquations
- Integrals
- Optimization
- JumpProcesses

THIRD-PARTY EQUATION SOLVERS
- LowRankIntegrators
- FractionalDiffEq
- ManifoldDiffEq

INVERSE PROBLEMS / ESTIMATION
- SciMLSensitivity
- DiffEqParamEstim
- DiffEqBayes

PDE SOLVERS
- MethodOfLines
- NeuralPDE
- NeuralOperators
- FEniCS
- HighDimPDE
- DiffEqOperators

THIRD-PARTY PDE SOLVERS
- Trixi
- Gridap
- ApproxFun
- VoronoiFVM

ADVANCED SOLVER APIS
- OrdinaryDiffEq
- DiffEqGPU

Detailed Overview of the SciML Software Ecosystem

Version v0.2

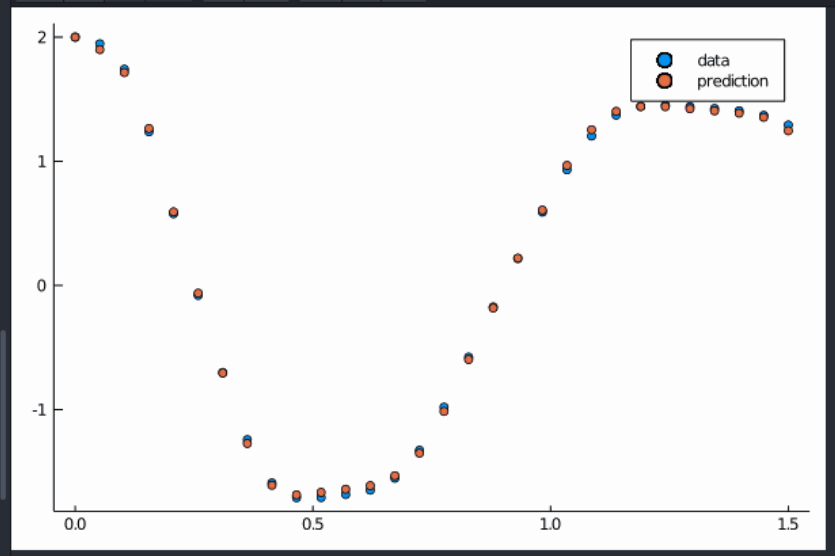
- Want to chat with someone? Check out our chat room and forums.
- Want to see our code? Check out the SciML Github organization.

And for diving into the details, use the bar on the top to navigate to the submodule of interest!

Part 2:

Methods which improve the fitting process

```julia
#0.001162 seconds (16.50 k allocations: 1.109 MiB)
#0.001148 seconds (16.50 k allocations: 1.109 MiB)

1639.7921118000004 / 0.001154



using DiffEqFlux, OrdinaryDiffEq, Flux, Optim, Plots

u0 = Float32[2.0; 0.0]
datasize = 30
tspan = (0.0f0, 1.5f0)
tsteps = range(tspan[1], tspan[2], length = datasize)

function trueODEfunc(du, u, p, t)
    true_A = [-0.1 2.0; -2.0 -0.1]
    du .= ((u.^3)'true_A)'
end

prob_trueode = ODEProblem(trueODEfunc, u0, tspan)
ode_data = Array(solve(prob_trueode, Tsit5(), saveat = tsteps))

dudt2 = FastChain((x, p) -> x.^3,
                  FastDense(2, 50, tanh),
                  FastDense(50, 2))    > FastChain
neural_ode_f(u,p,t) = dudt2(u,p)    > neural_ode_f
pinit = initial_params(dudt2)    > Vector{Float32} with 252 elements
prob = ODEProblem(neural_ode_f, u0, tspan, pinit)    > ODEProblem with uType Array{Float32,1} and tType

function predict_neuralode(p)
  tmp_prob = remake(prob,p=p)
  Array(solve(tmp_prob,Tsit5(),saveat=tsteps))
end    > predict_neuralode
```



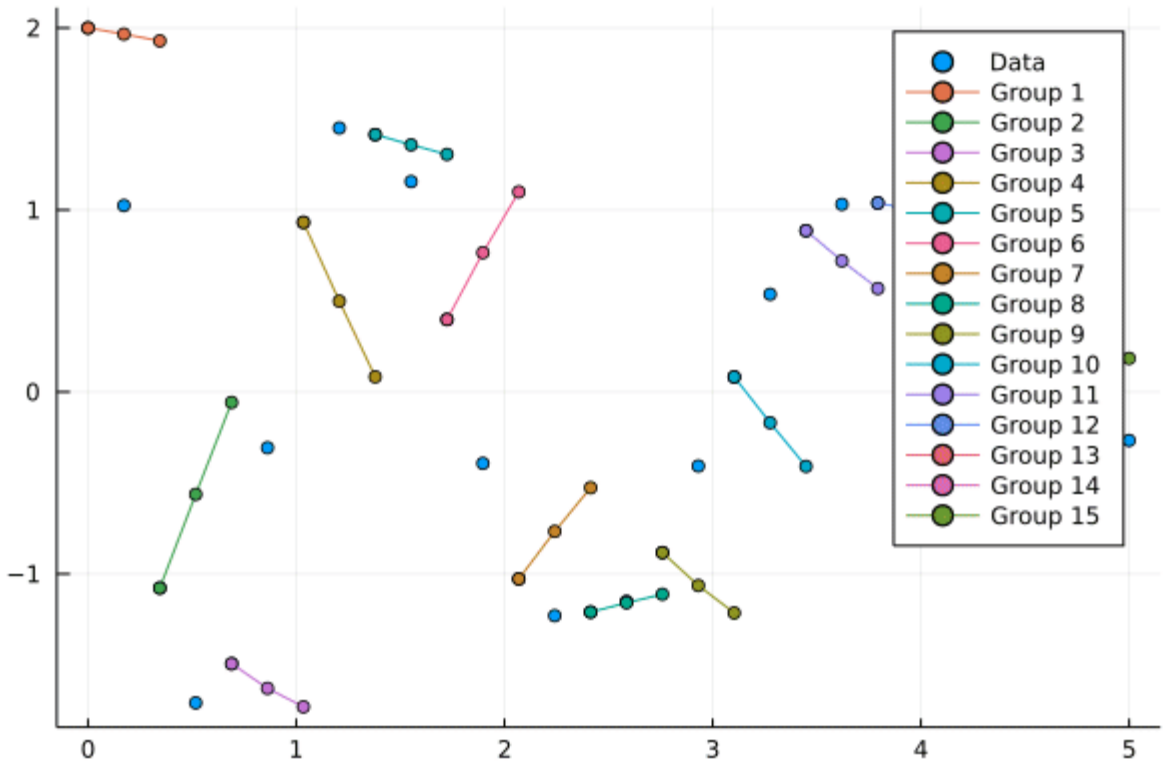

```
0.09795238f0
0.09594628f0
0.091021195f0
0.074081644f0
0.07087004f0
0.06550323f0
0.06374359f0
0.058643457f0
0.055588786f0
0.05309863f0
0.05249826f0
0.05105587f0
0.051051125f0
0.051051125f0
0.051051125f0

julia>
```

Fitting by running the simulator and doing gradient-based optimization

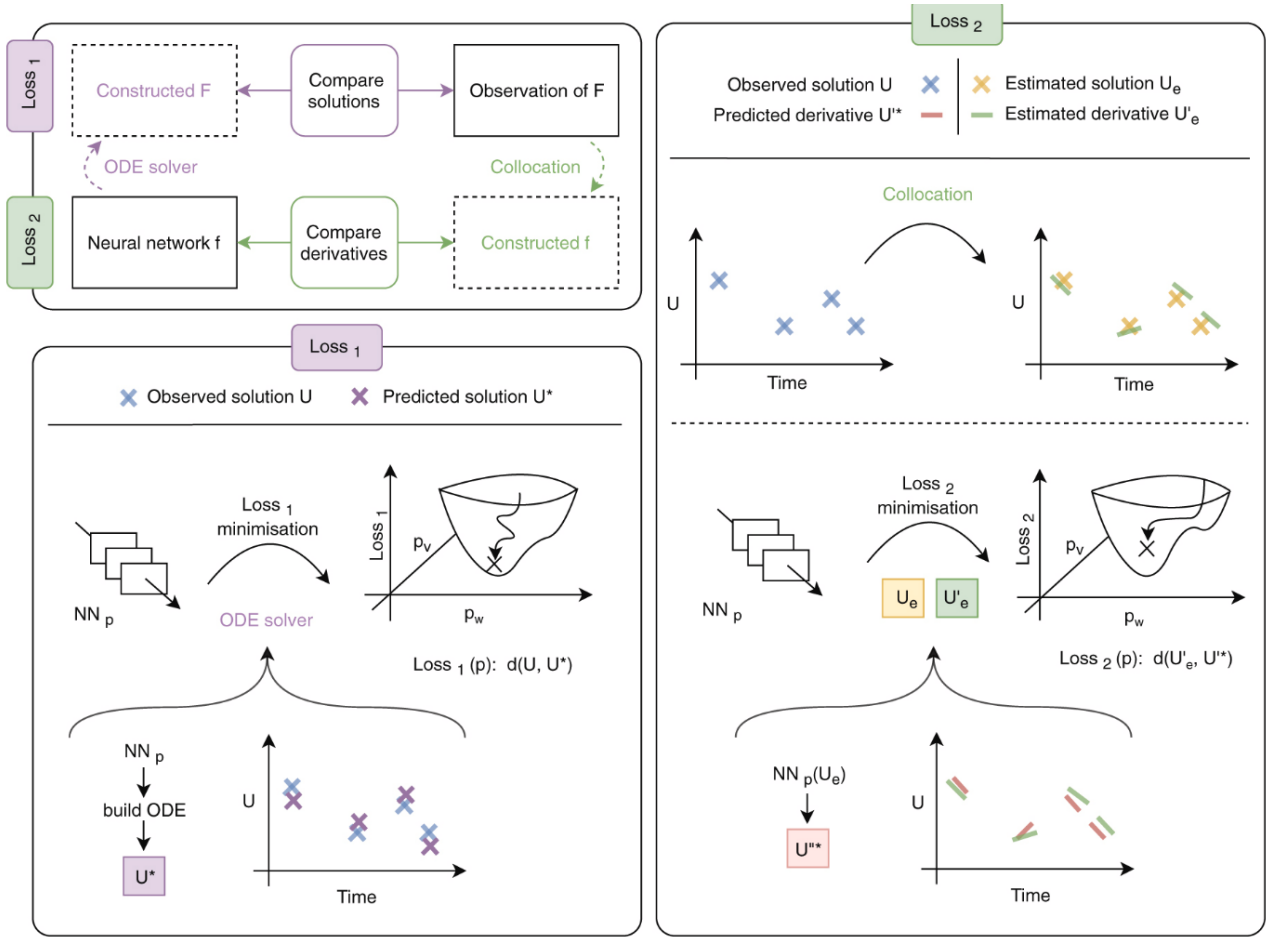
= single shooting

Single shooting is not numerically robust. Other loss functions are required in practice!

Some Alternative Loss Functions: Multiple Shooting and Collocation



Multiple Shooting Methods



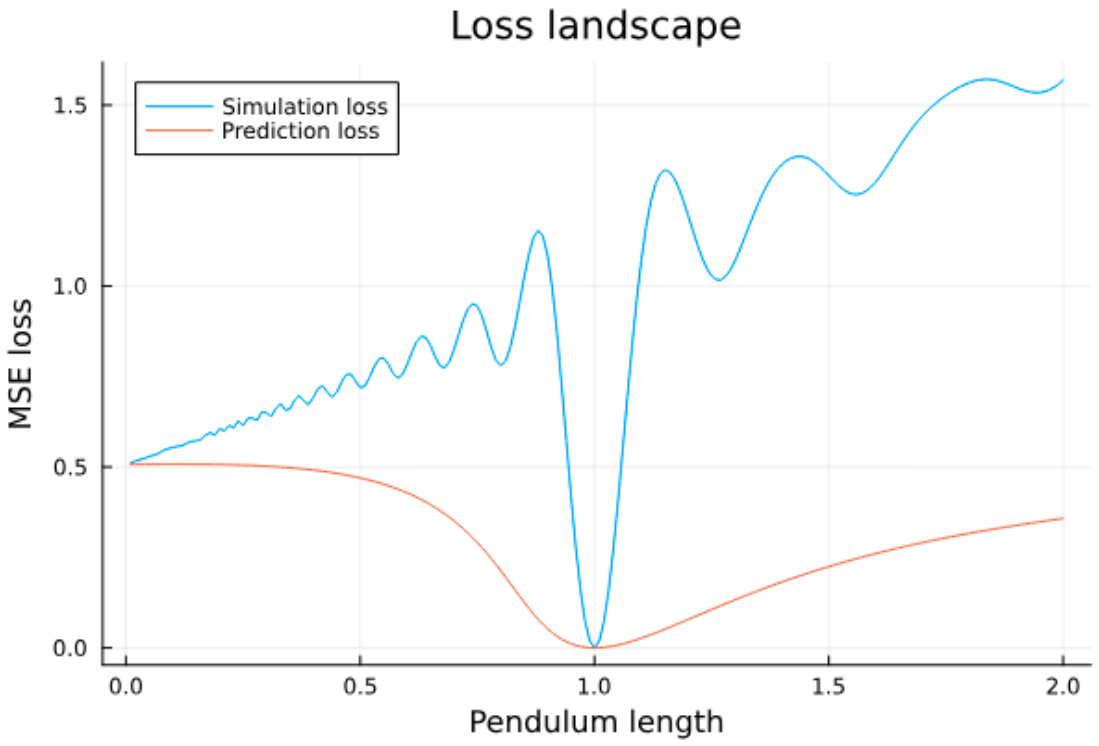
Turan, E. M., & Jäschke, J. (2021). Multiple shooting with neural differential equations. *arXiv preprint arXiv:2109.06786*.

Roesch, Elisabeth, Christopher Rackauckas, and Michael PH Stumpf. "Collocation based training of neural ordinary differential equations." *Statistical Applications in Genetics and Molecular Biology* (2021).

Prediction Error Method (PEM)

```julia
function simulator(du, u, p, t) # Pendulum dynamics
    g = 9.82 # Gravitational constant
    L = p isa Number ? p : p[1] # Length of the pendulum
    gL = g / L
    θ = u[1]
    dθ = u[2]
    du[1] = dθ
    du[2] = -gL * sin(θ)
end
```

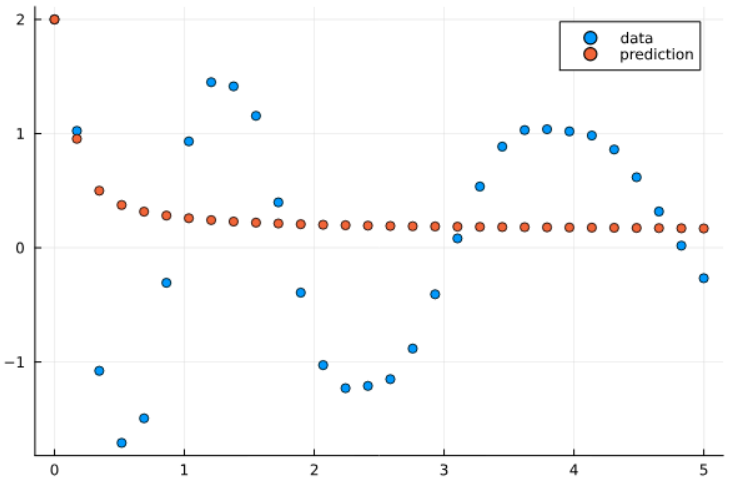
```julia
function predictor(du, u, p, t)
    g = 9.82
    L, K, y = p # pendulum length, observer gain and measurements
    gL = g / L
    θ = u[1]
    dθ = u[2]
    yt = y(t)
    e = yt - θ
    du[1] = dθ + K * e
    du[2] = -gL * sin(θ)
end
```



Use a modified simulator which is always filtered towards the data points

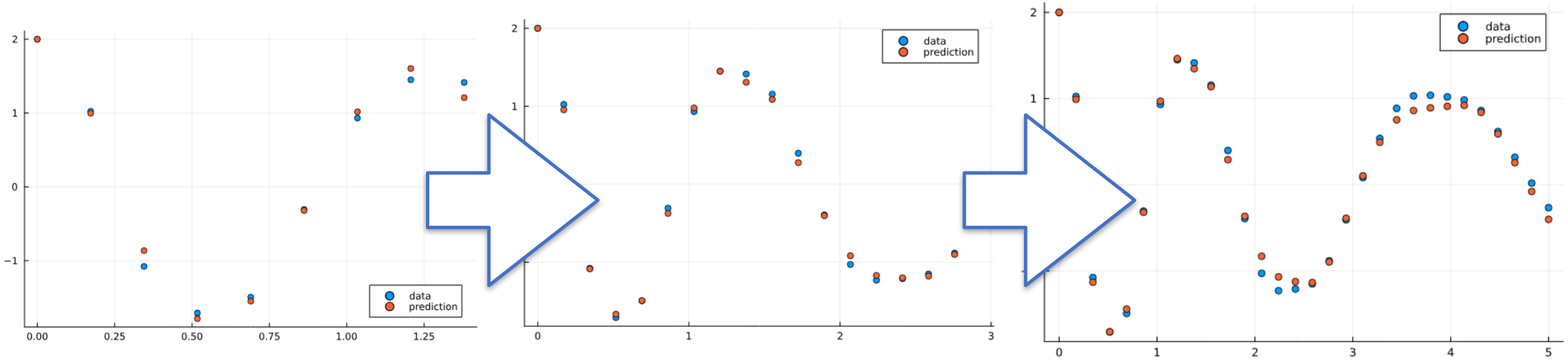
https://docs.sciml.ai/SciMLSensitivity/dev/examples/ode/prediction_error_method/

Simple Tricks: Growing the Time Interval



Doing the optimization in a single pass may not be robust,

Successively grow the interval



Let's go back to this example

Run the code yourself!

https://github.com/Astroinformatics/ScientificMachineLearning/blob/main/neuralode_gw.ipynb

Upon denoting $\mathbf{x} = (\phi, \chi, p, e)$, we propose the following family of UDEs to describe the two-body relativistic dynamics:

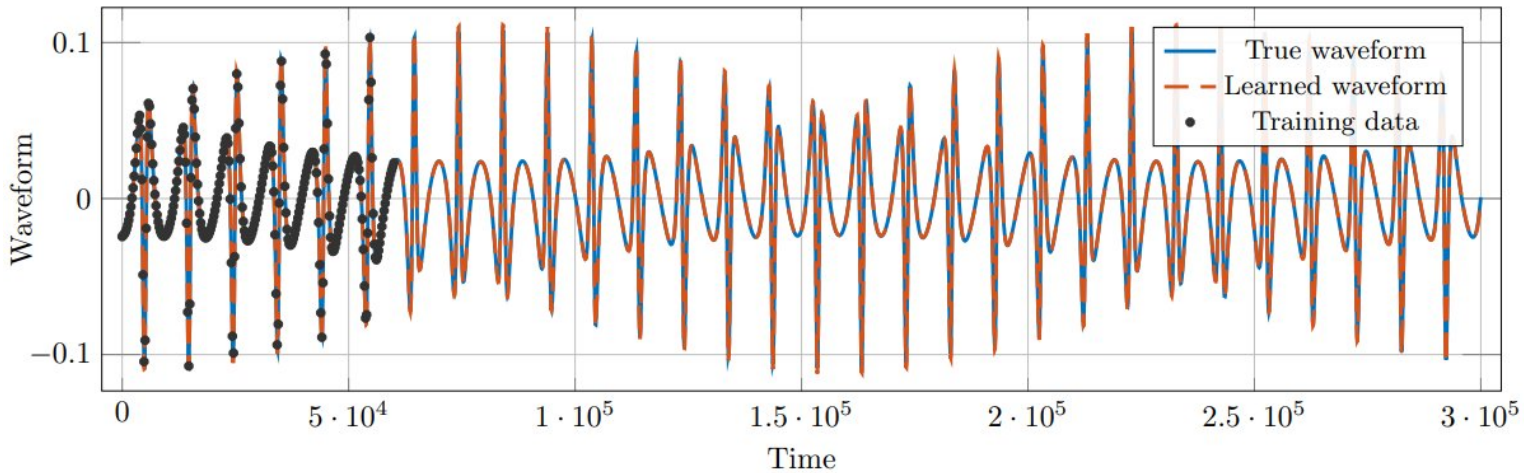
$$\dot{\phi} = \frac{(1 + e\cos(\chi))^2}{Mp^{3/2}}\big(1 + \mathcal{F}_1(\cos(\chi), p, e)\big), \tag{5a}$$

$$\dot{\chi} = \frac{(1 + e\cos(\chi))^2}{Mp^{3/2}}\big(1 + \mathcal{F}_2(\cos(\chi), p, e)\big), \tag{5b}$$

$$\dot{p} = \mathcal{F}_3(p, e), \tag{5c}$$

$$\dot{e} = \mathcal{F}_4(p, e), \tag{5d}$$

Example using binary black hold dynamics with LIGO gravitational wave data



Keith, Brendan, Akshay Khadse, and Scott E. Field. "Learning orbital dynamics of binary black hole systems from gravitational wave measurements." Physical Review Research 3, no. 4 (2021): 043101.

Let's go back to this example

```
NN_params = NN_params .* 0 + Float64(1e-4) * randn(StableRNG(2031), eltype(NN_params), size(NN_params))
```

The neural network is a residual, so start the training as a **small** perturbation!

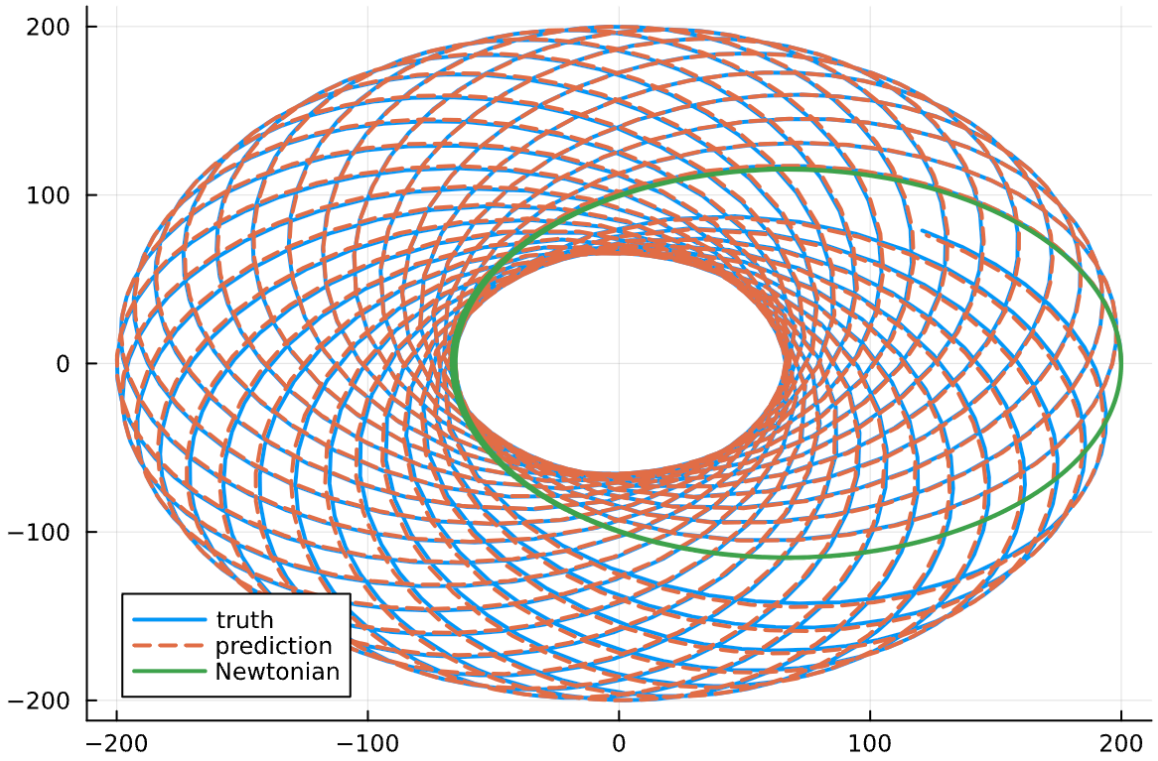
Upon denoting $\mathbf{x} = (\phi, \chi, p, e)$, we propose the following family of UDEs to describe the two-body relativistic dynamics:

$$\dot{\phi} = \frac{(1 + e\cos(\chi))^2}{Mp^{3/2}}\left(1 + \mathcal{F}_1(\cos(\chi), p, e)\right), \tag{5a}$$

$$\dot{\chi} = \frac{(1 + e\cos(\chi))^2}{Mp^{3/2}}\left(1 + \mathcal{F}_2(\cos(\chi), p, e)\right), \tag{5b}$$

$$\dot{p} = \mathcal{F}_3(p, e), \tag{5c}$$

$$\dot{e} = \mathcal{F}_4(p, e), \tag{5d}$$

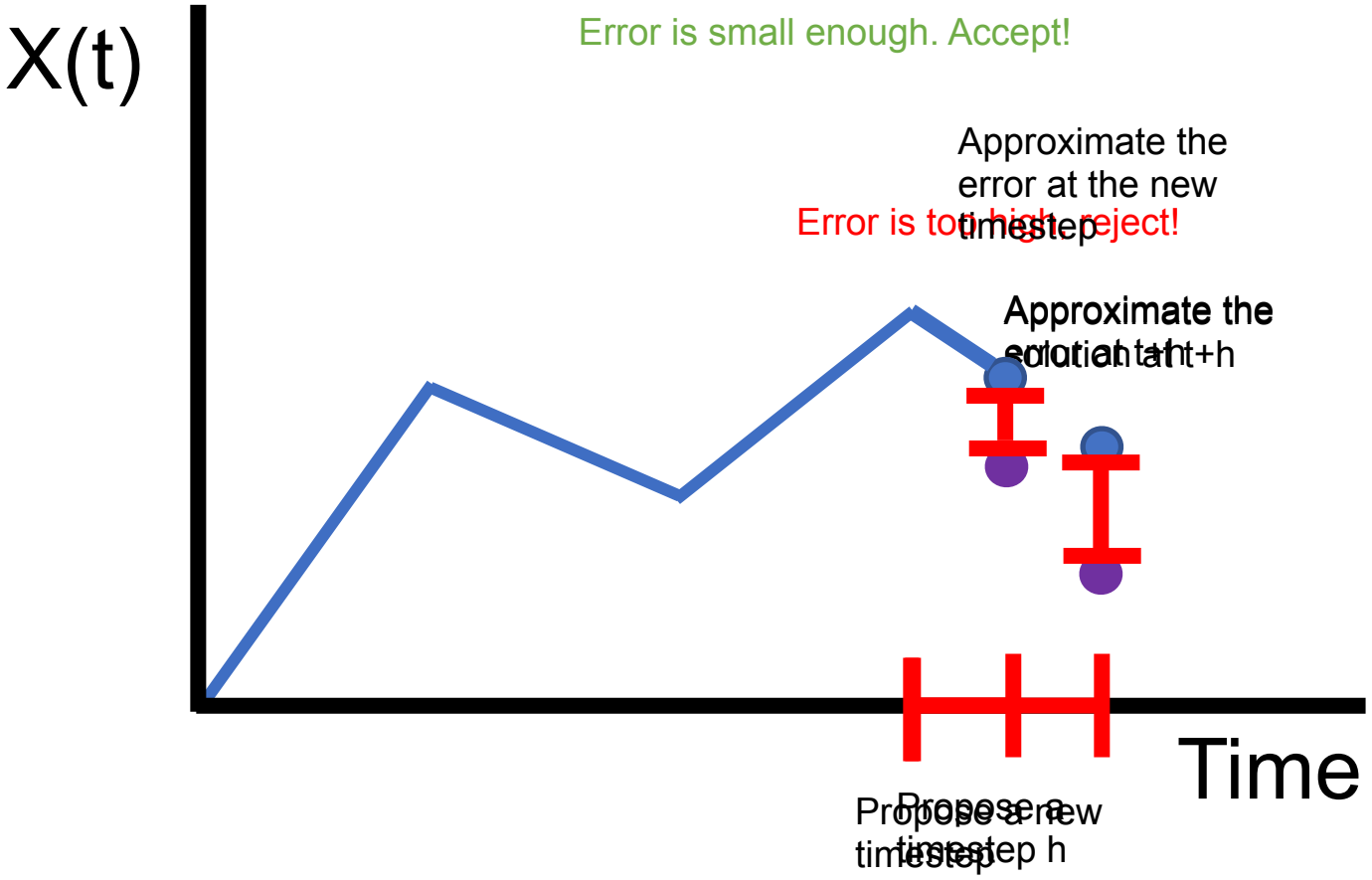


Conclusion Part 2:

Don’t use single shooting. Modify the simulation process to improve the fitting.

Sidebar: A note on Neural Network Architectures in ODEs

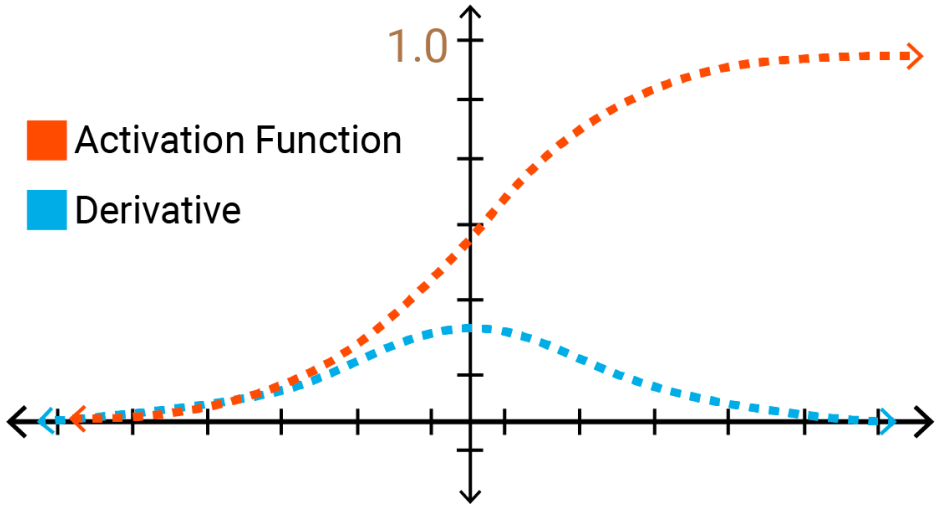
ODE Solvers don't always go forwards!



If you're using an adaptive ODE solver, you cannot assume that the next step will be forward in time from the previous one.

I.e., neural networks with state (RNN, GRU, etc.) do not give a well-defined ODE solution and will fail in adaptivity!

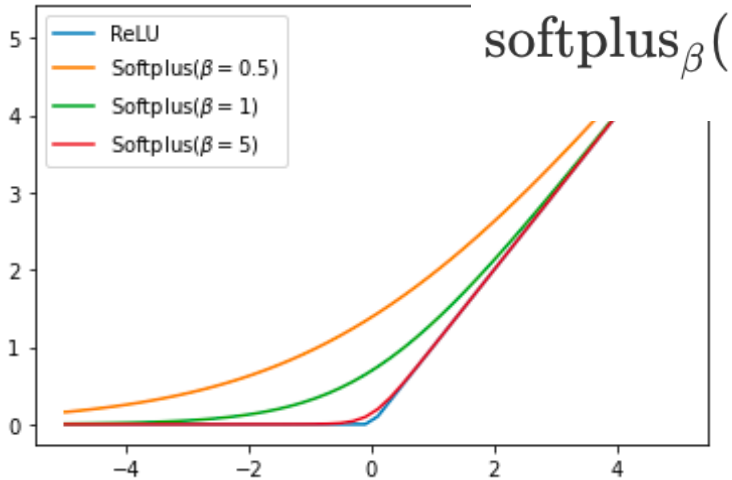
Be Aware of Vanishing Gradients



* Many loss functions have gradients which go to zero when loss functions get extreme.
* ODEs naturally amplify values (exponentially!) as time gets larger
* Consequence: gradients can become zero, making training become ineffective

Solutions:

* Never train for long intervals (successive interval growth, multiple shooting)
* Use loss functions which don't saturate (but try and keep them smooth (?))

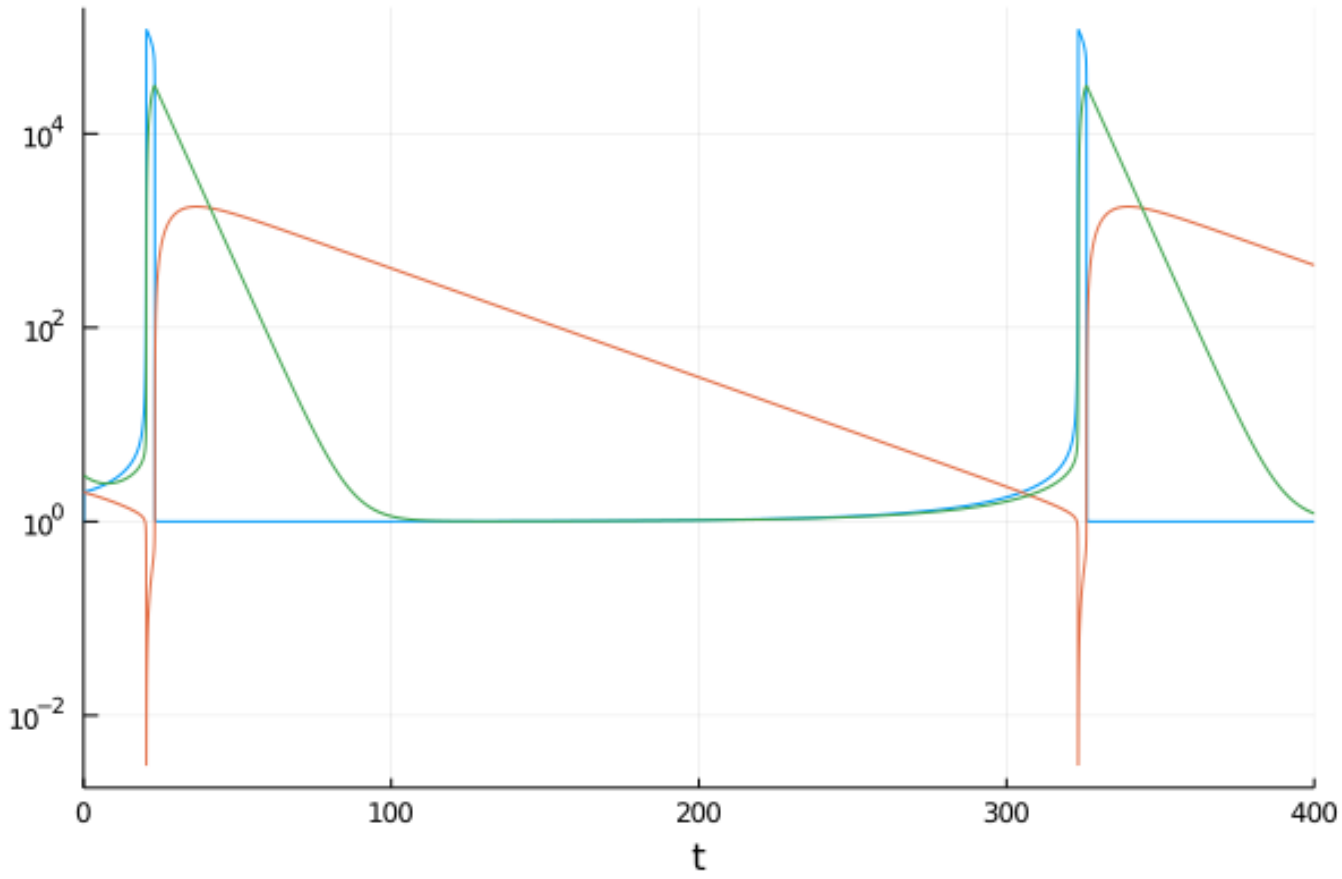


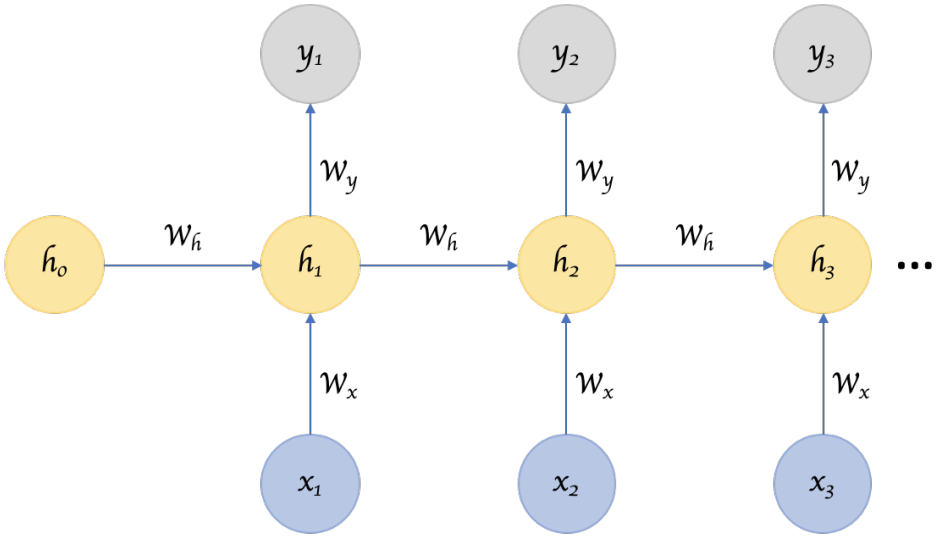
$$\mathrm{softplus}_\beta(x) = \frac{1}{\beta}\log(1 + e^{\beta x})$$

Part 3:

Methods which ignore such derivative issues that could be interesting to explore

Challenge: train a surrogate to accelerate an arbitrary highly stiff system





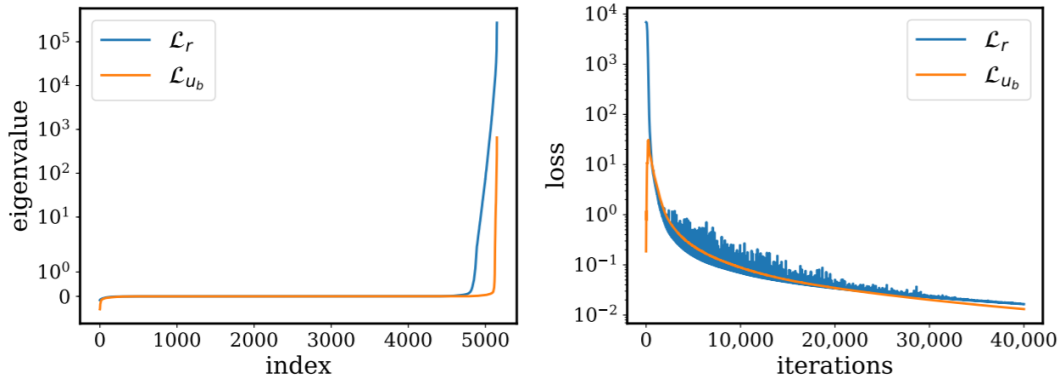
Recurrent neural network? No!

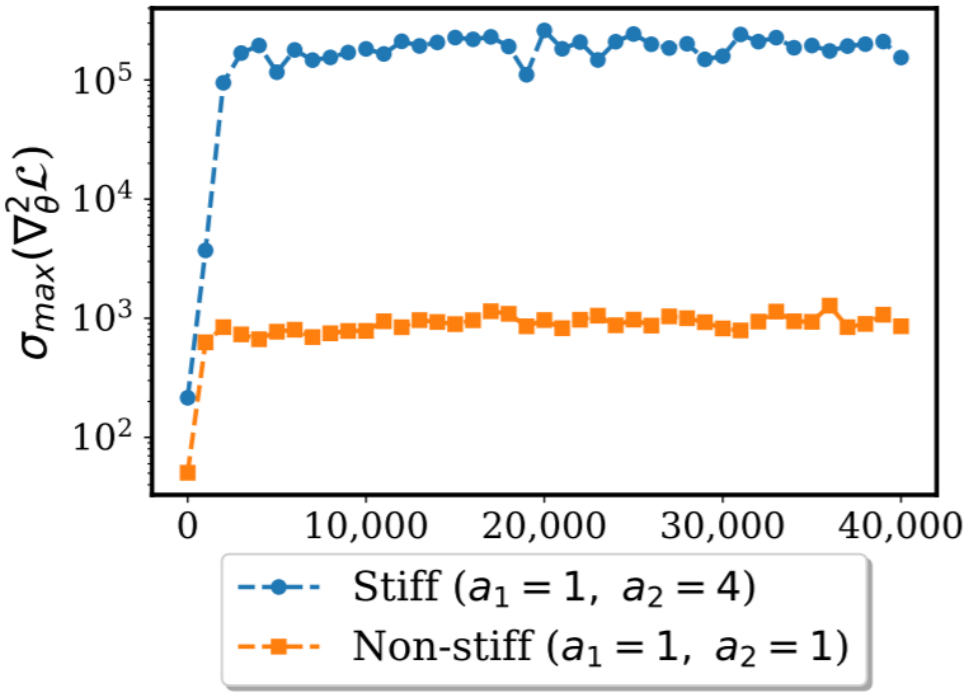
1. It's an explicit method! (Euler's)
2. Uniform steps will not capture the spikes!

Stiffness causes a problem even with many SciML approaches like Physics-Informed Neural Networks (PINNs)

1. Neural networks have difficulties matching highly ill-conditioned systems
2. Optimization techniques like gradient descent are explicit processes attempting to solving a stiff model
3. Stiffness in the model can translate to stiffness in the optimization process as it tries to find a manifold
4. Timescale separations of 10^9 and more are common in real applications

We need to utilized all of the advanced numerical knowledge for handling stiff systems to work in tandem with ML!



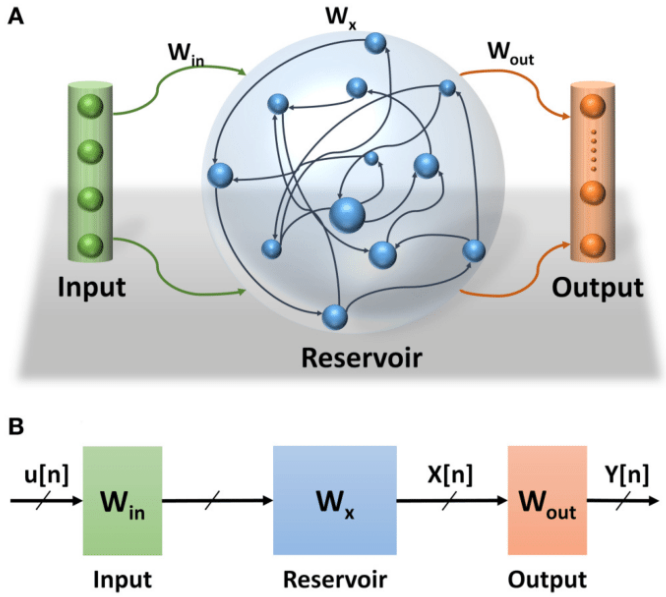


Understanding and mitigating gradient pathologies in physics-informed neural networks

Sifan Wang, Yujun Teng, Paris Perdikaris

Idea: Avoid Gradients and Use an Implicit Fit

Some precedence: echo state networks
Fix a random process and find a projection to fit the system



Adapting: continuous-time echo state networks
Build a random non-stiff ODE and find a projection to the stiff ODE

Fix $r' = \sigma(Ar + W_x x)$
Predict $x(t) = W_{out} r(t)$

Turns into a linear solve
Solve the linear system via SVD
(to manage the growth factor)

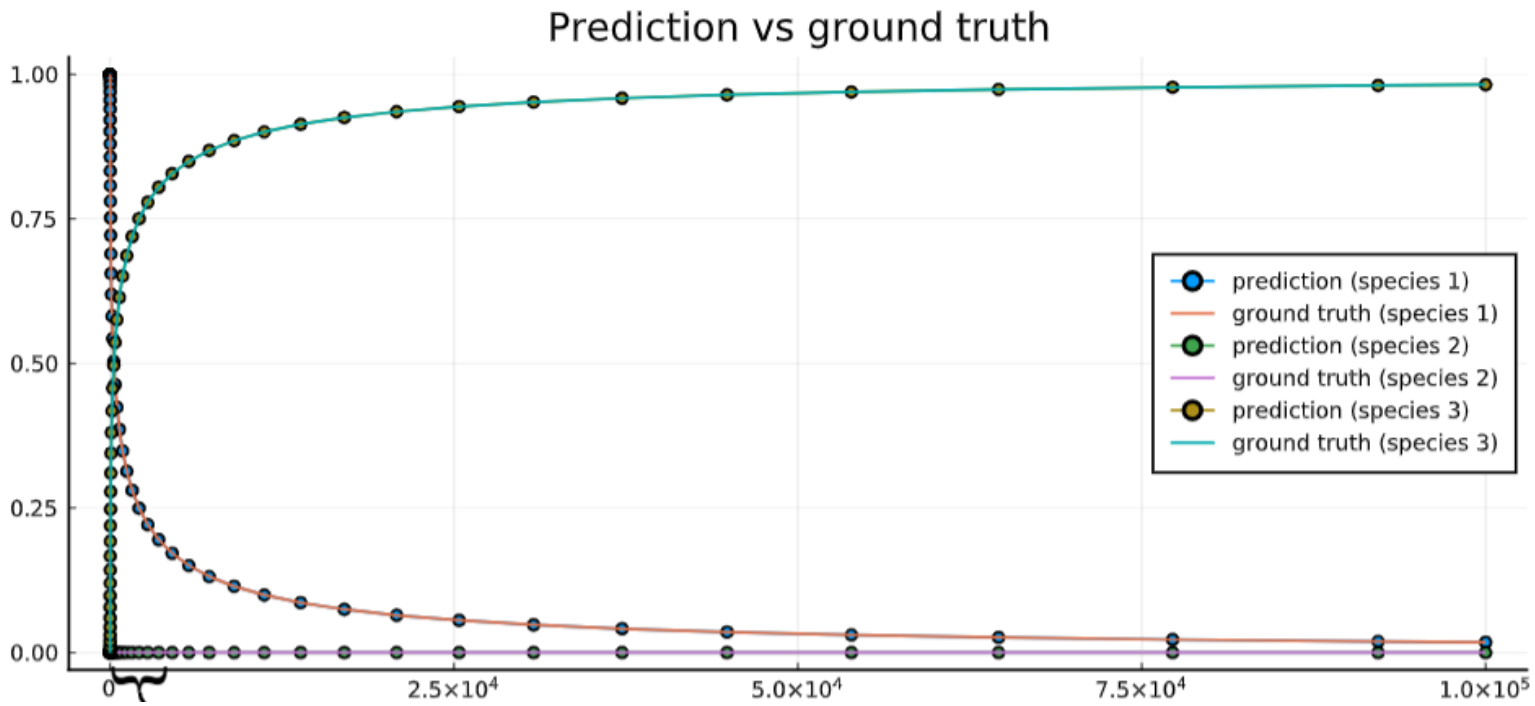
Get W_{out} at many parameters of the system

Predict behavior at new parameters via:

$$x(t) = W_{out}(p) r(t)$$

Using a Radial Basis Function constructed from the W_{out} training data

Continuous-Time Echo State Networks
Handle the stiff equations where current methods fail



Robertson's Equations

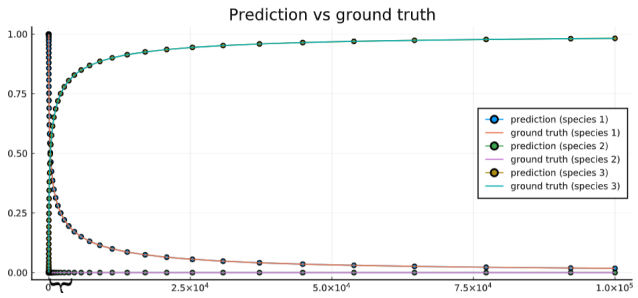
Classic stiff ODE
Used to test and break integrators
Volatile early transient

$$\dot{y_1} = -0.04y_1 + 10^4 y_2 \cdot y_3$$
$$\dot{y_2} = 0.04y_1 - 10^4 y_2 \cdot y_3 - 3 \cdot 10^7 y_2^2$$
$$\dot{y_3} = 3 \cdot 10^7 y_2^2$$

Accelerating Simulation of Stiff Nonlinear Systems using Continuous-Time Echo State Networks

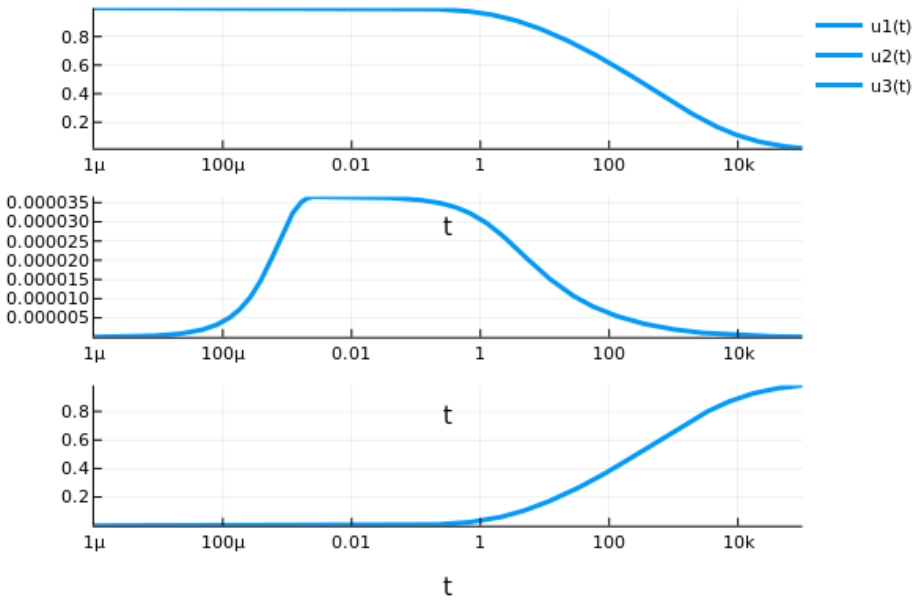
Ranjan Anantharaman, Yingbo Ma, Shashi Gowda, Chris Laughman, Viral Shah, Alan Edelman, Chris Rackauckas

Continuous-Time Echo State Networks
Handle the stiff equations where current methods fail



Log-Scale Fast Changes!

No auto-catalyst, no dynamics



Robertson's Equations

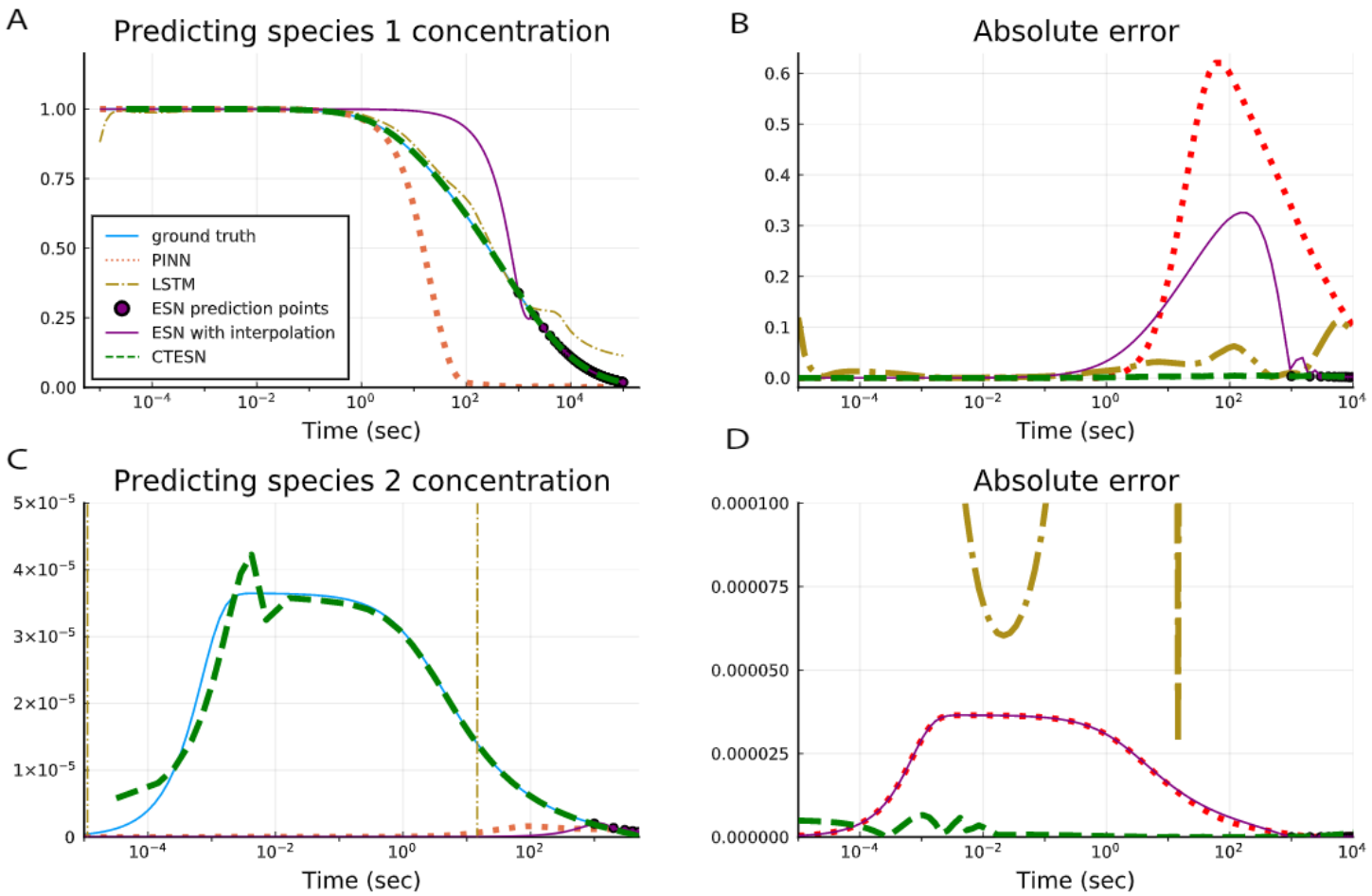
Classic stiff ODE
Used to test and break integrators
Volatile early transient

$$\dot{y_1} = -0.04y_1 + 10^4 y_2 \cdot y_3$$
$$\dot{y_2} = 0.04y_1 - 10^4 y_2 \cdot y_3 - 3 \cdot 10^7 y_2^2$$
$$\dot{y_3} = 3 \cdot 10^7 y_2^2$$

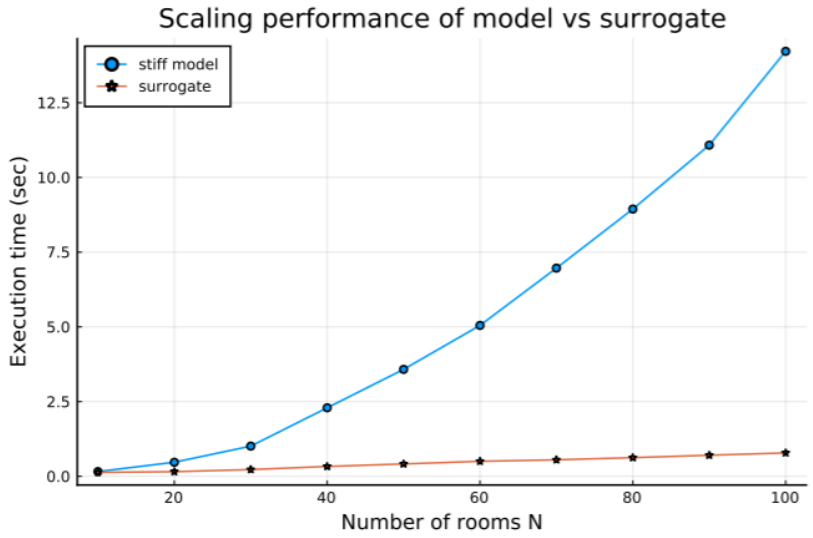
Accelerating Simulation of Stiff Nonlinear Systems using Continuous-Time Echo State Networks

Ranjan Anantharaman, Yingbo Ma, Shashi Gowda, Chris Laughman, Viral Shah, Alan Edelman, Chris Rackauckas

Continuous-Time Echo State Networks
Handle the stiff equations where current methods fail



Only CTESNs Capture the Hard Dynamics



After training, 100x faster than direct simulation!

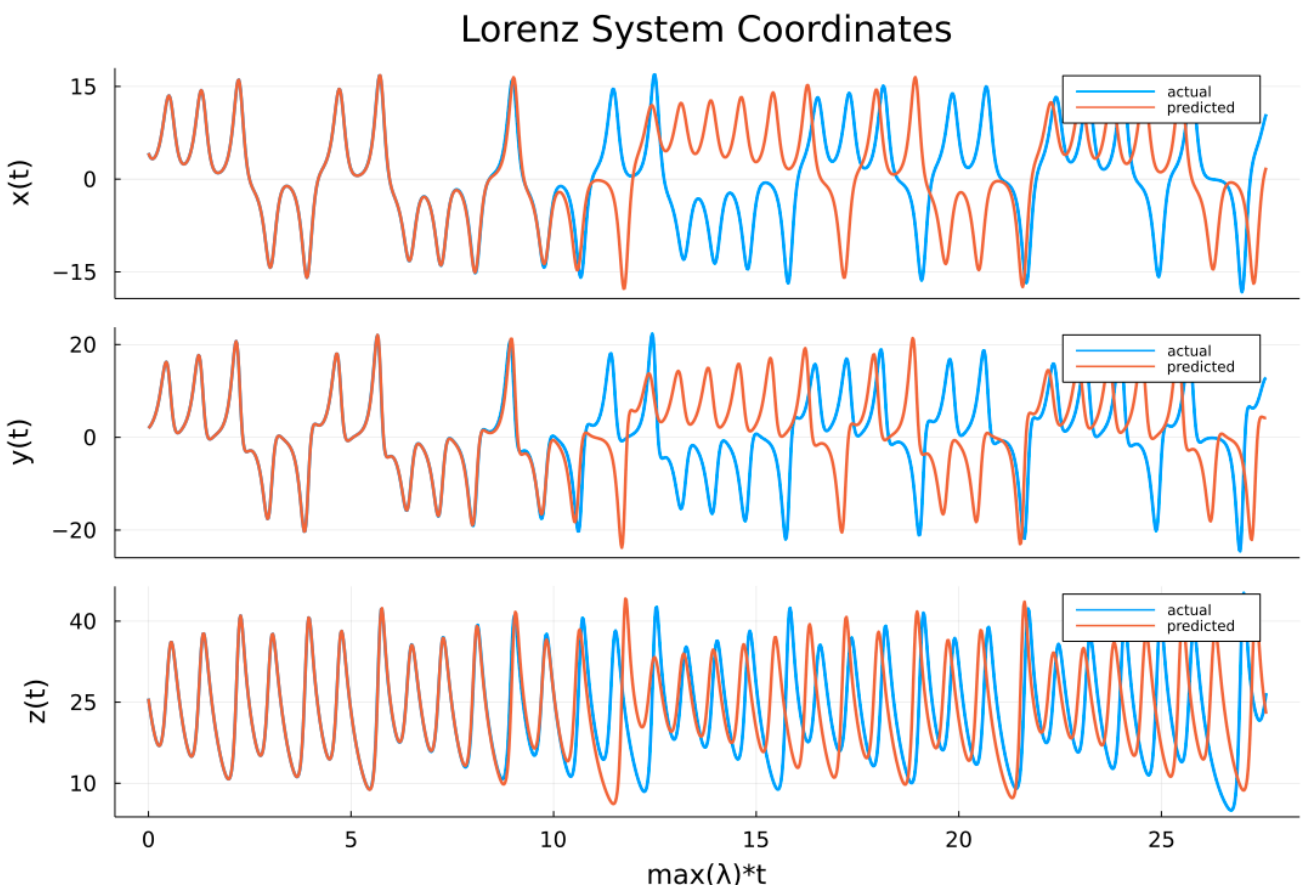
Accelerating Simulation of Stiff Nonlinear Systems using Continuous-Time Echo State Networks

Ranjan Anantharaman, Yingbo Ma, Shashi Gowda, Chris Laughman, Viral Shah, Alan Edelman, Chris Rackauckas

ReservoirComputing.jl

Reservoir Computing.jl

```
output_layer = train(esn, target_data)
output = esn(Generative(predict_len), output_layer)
```

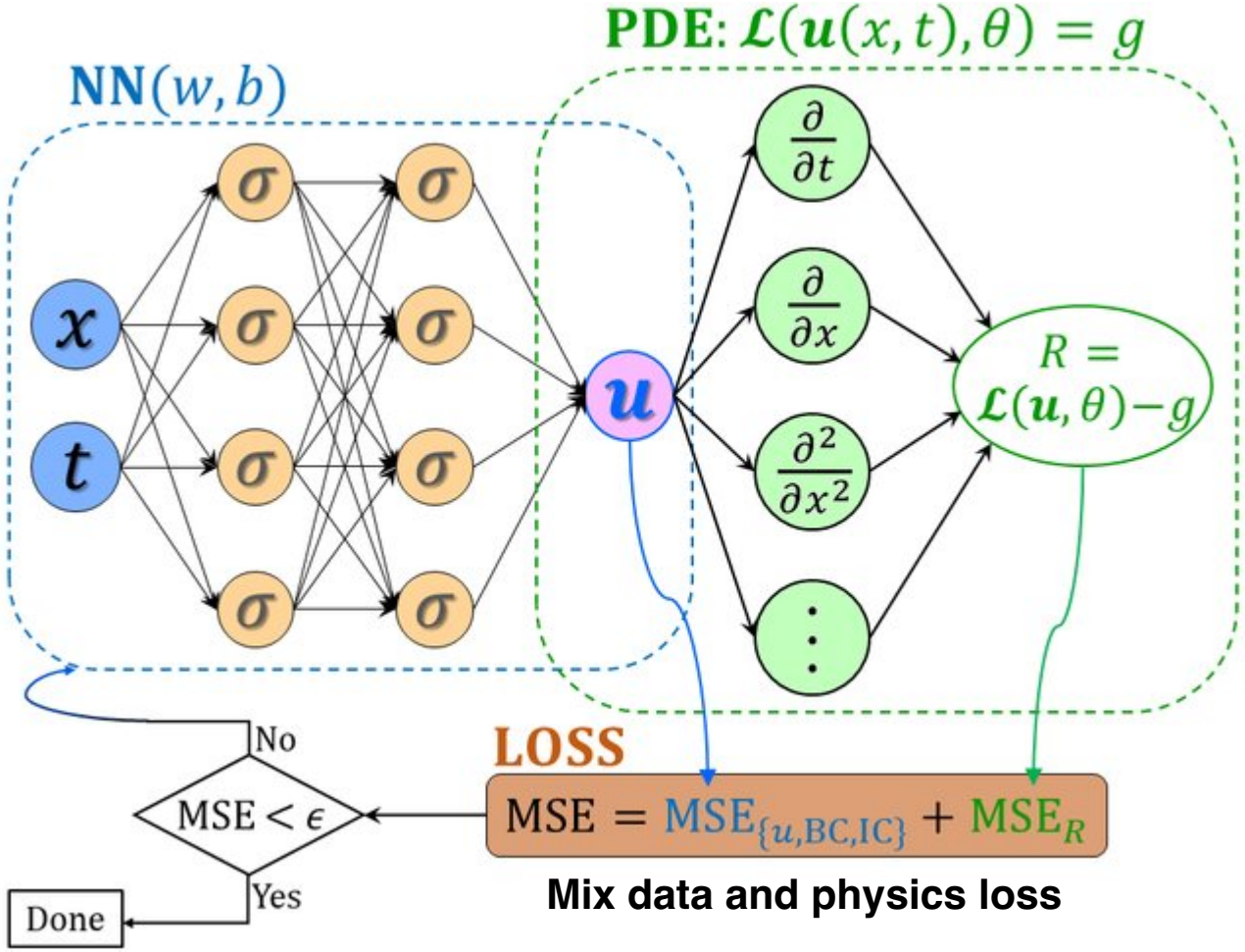


Part 4: Performance

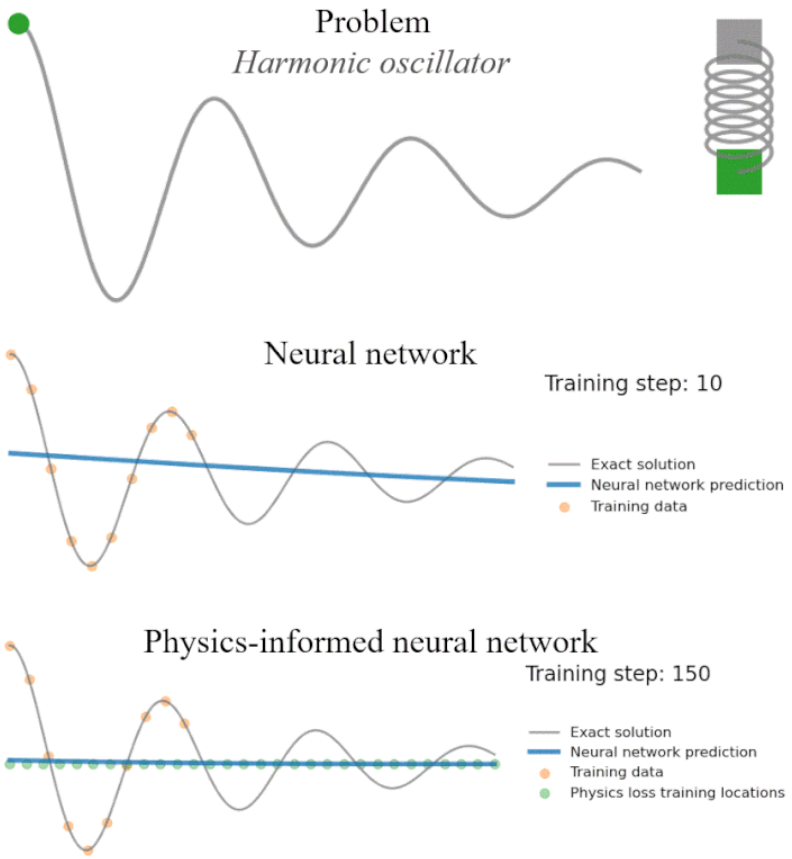
A Deep Dive into how Performance is Different Between Deep Learning and Differentiable Simulation

When/Why should this be preferred over other techniques like physics-informed neural networks (PINNs) and neural operator techniques (DeepONets)?

Why Use Physics-Informed Neural Networks?



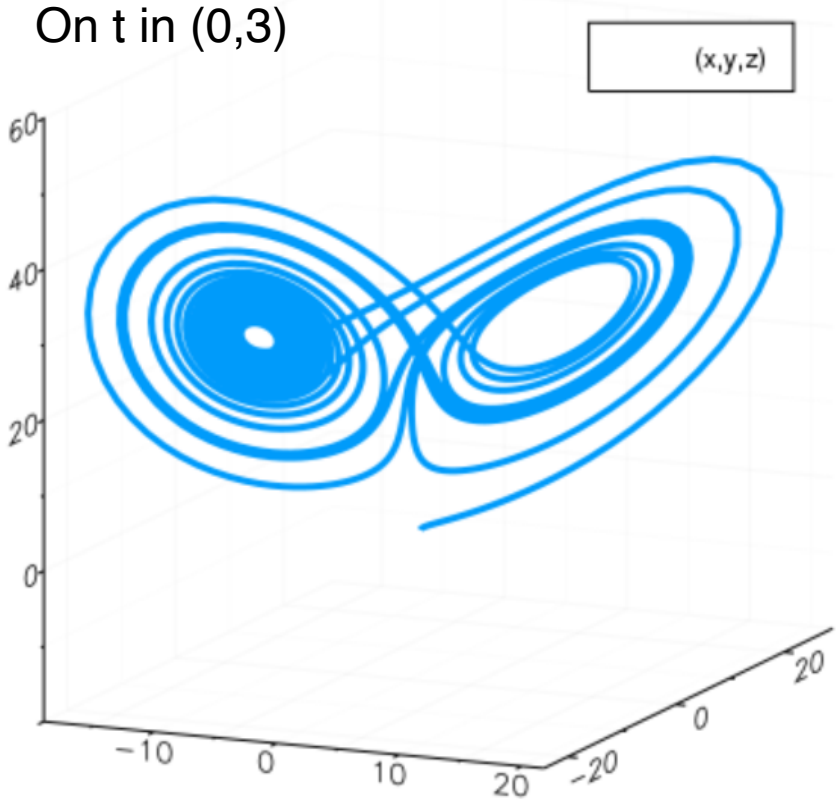
Mix data and physics loss



Outperforms standard machine learning

Keeping Neural Networks Small Keeps Speed For Inverse Problems

Problem: parameter estimation
of Lorenz equation from data
On t in (0,3)



DeepXDE (TensorFlow Physics-Informed NN)

```
Best model at step 57000:
  train loss: 5.91e-03
  test loss: 5.86e-03
  test metric: []

'train' took 362.351454 s
```

DiffEqFlux.jl (Julia UDEs)

```
opt = Opt(:LN_BOBYQA, 3)
lower_bounds!(opt,[9.0,20.0,2.0])
upper_bounds!(opt,[11.0,30.0,3.0])
min_objective!(opt, obj_short.cost_function2)
xtol_rel!(opt,1e-12)
maxeval!(opt, 10000)
@time (minf,minx,ret) = NLopt.optimize(opt,LocIniPar) # 0.1 seconds
```

```
0.032699 seconds (148.87 k allocations: 14.175 MiB)
(2.7636309213683456e-18, [10.0, 28.0, 2.66], :XTOL_REACHED)
```

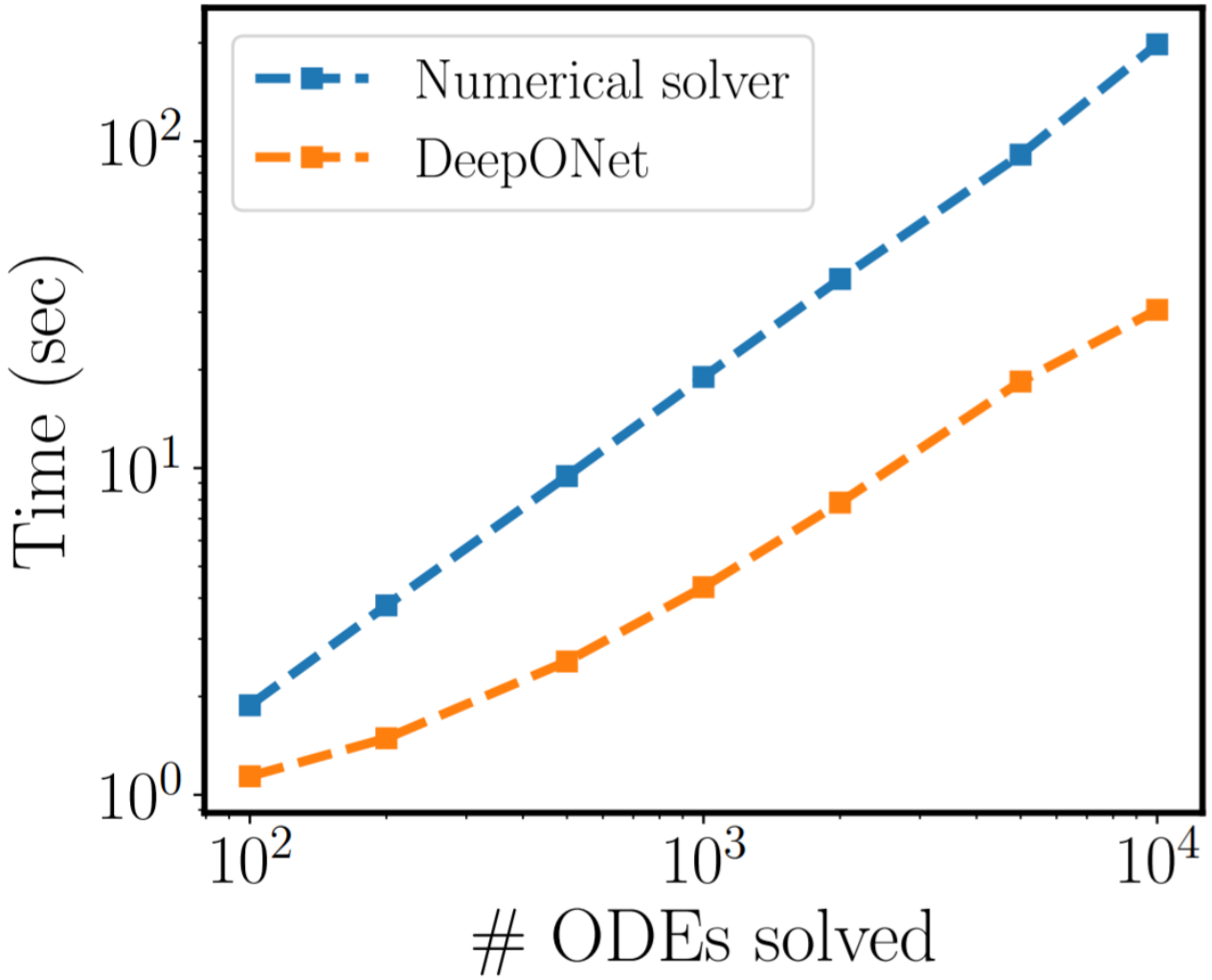
Note on Neural Networks “Outperforming” Classical Solvers

Long-time integration of parametric evolution equations with physics-informed DeepONets

Sifan Wang, Paris Perdikaris

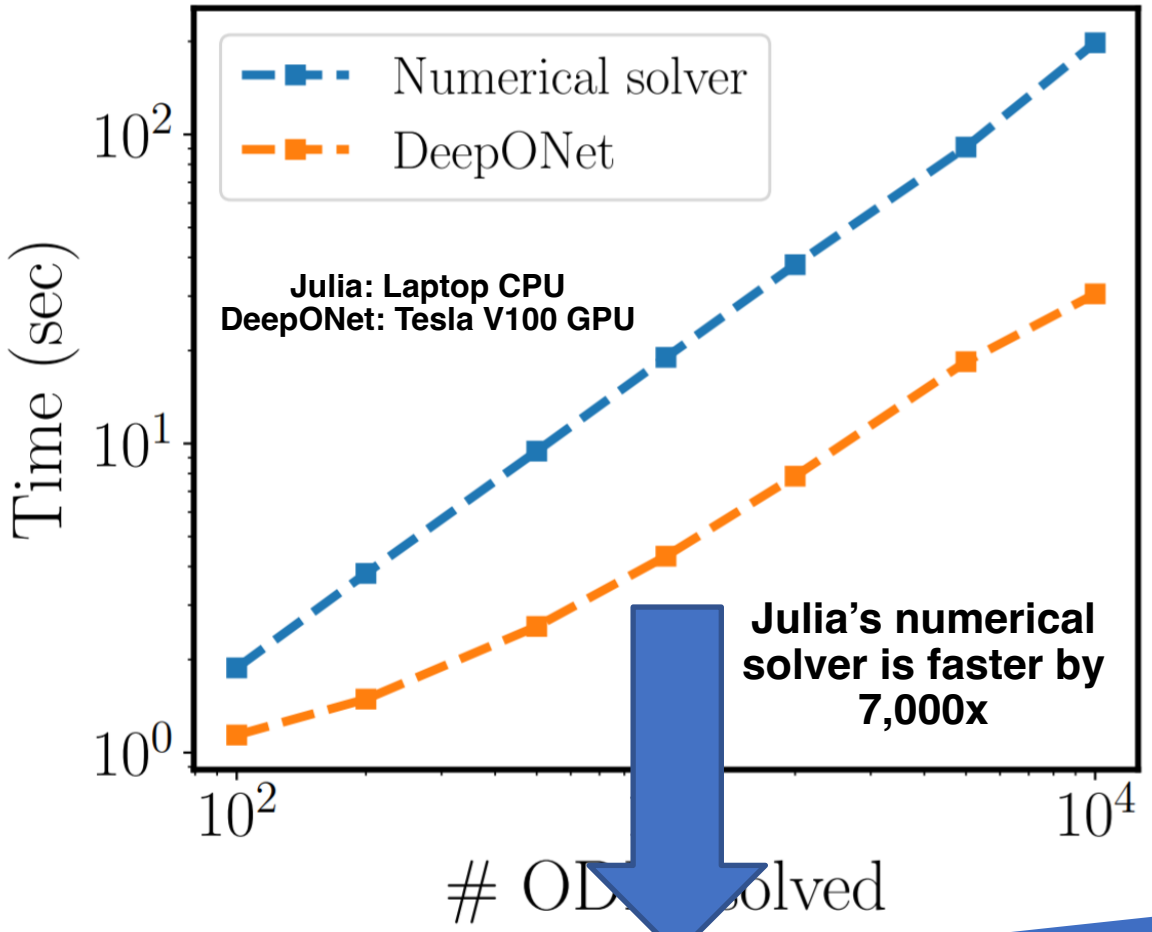
Ordinary and partial differential equations (ODEs/PDEs) play a paramount role in analyzing and simulating complex dynamic processes across all corners of science and engineering. In recent years machine learning tools are aspiring to introduce new effective ways of simulating PDEs, however existing approaches are not able to reliably return stable and accurate predictions across long temporal horizons. We aim to address this challenge by introducing an effective framework for learning infinite-dimensional operators that map random initial conditions to associated PDE solutions within a short time interval. Such latent operators can be parametrized by deep neural networks that are trained in an entirely self-supervised manner without requiring any paired input-output observations. Global long-time predictions across a range of initial conditions can be then obtained by iteratively evaluating the trained model using each prediction as the initial condition for the next evaluation step. This introduces a new approach to temporal domain decomposition that is shown to be effective in performing accurate long-time simulations for a wide range of parametric ODE and PDE systems, from wave propagation, to reaction-diffusion dynamics and stiff chemical kinetics, all at a fraction of the computational cost needed by classical numerical solvers.

Note on Neural Networks "Outperforming" Classical Solvers



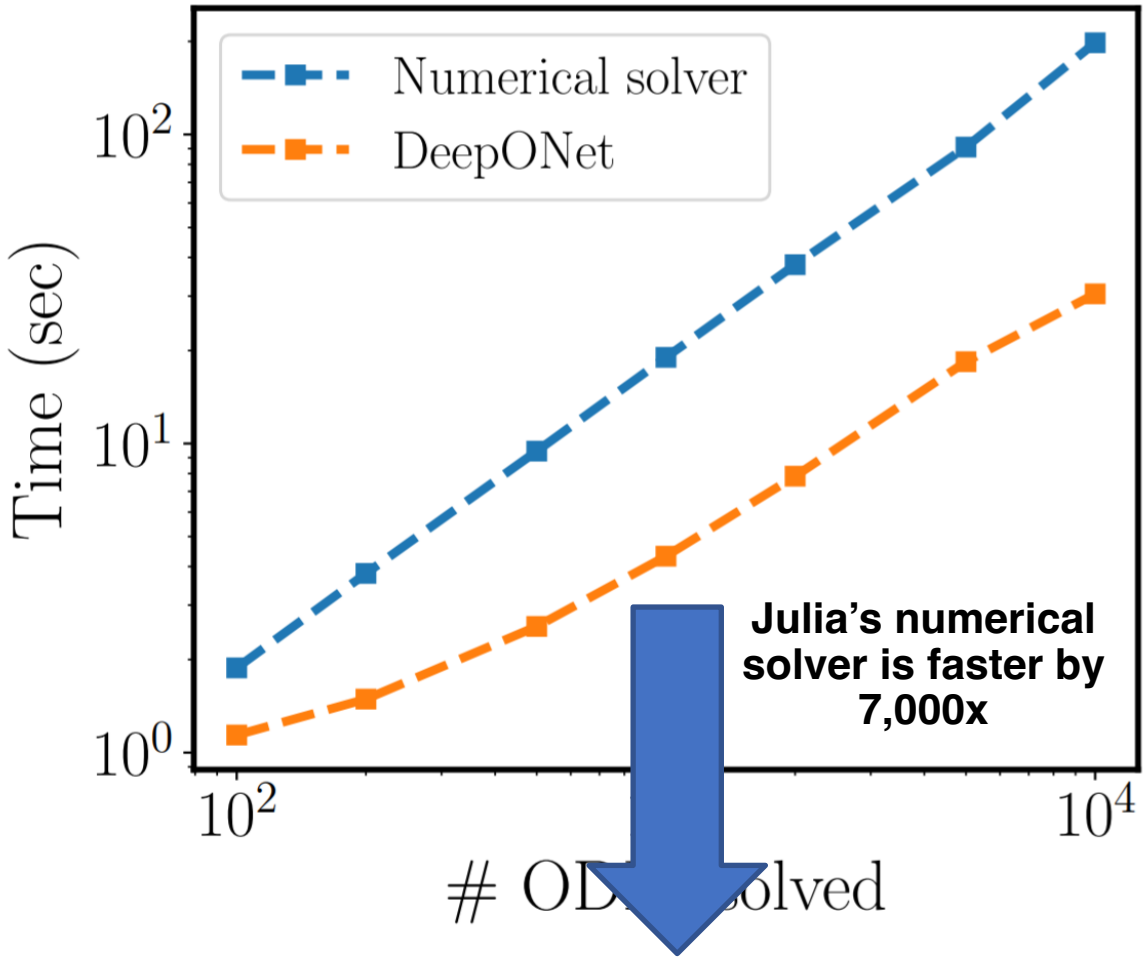
Oh no, we're doomed!

Wait a second?




```julia
using ModelingToolkit, OrdinaryDiffEq, StaticArrays

@variables t y₁(t) y₂(t) y₃(t)
@parameters k₁ k₂ k₃
D = Differential(t)
eqs = [D(y₁) ~ -k₁*y₁+k₃*y₂*y₃
       D(y₂) ~  k₁*y₁-k₂*y₂^2-k₃*y₂*y₃
       D(y₃) ~  k₂*y₂^2]

sys = ODESystem(eqs, t)
prob = ODEProblem{false}(sys,SA[y₁=>1f0,y₂=>0f0,y₃=>0f0],(0f0,500f0),
                         SA[k₁=>4f-2,k₂=>3f7,k₃=>1f4],jac=true)

N = 1000
y₁s = rand(Float32,N)
y₂s = 1f-4 .* rand(Float32,N)
y₃s = rand(Float32,N)

function prob_func(prob,i,repeat)
    remake(prob,p=SA[y₁s[i],y₂s[i],y₃s[i]])
end

monteprob = EnsembleProblem(prob, prob_func = prob_func, safetycopy=false)
solve(monteprob,Rodas5(),EnsembleThreads(),trajectories=1000)

@time solve(monteprob,Rodas5(),EnsembleThreads(),trajectories=1000)

#0.006486 seconds (172.26 k allocations: 16.740 MiB)
#0.006024 seconds (172.26 k allocations: 16.740 MiB)
#0.007074 seconds (172.26 k allocations: 16.740 MiB)
```

Wait a second?

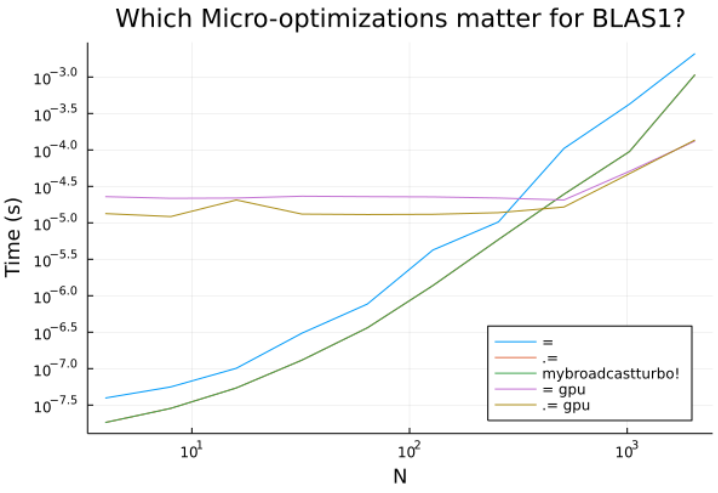


Similar story on Fourier Neural Operator results!

How come so far off?

If Differentiable Simulation techniques are easily >1000x more efficient, then why doesn’t everyone “see” that?

Code Optimization in Machine Learning vs Scientific Computing

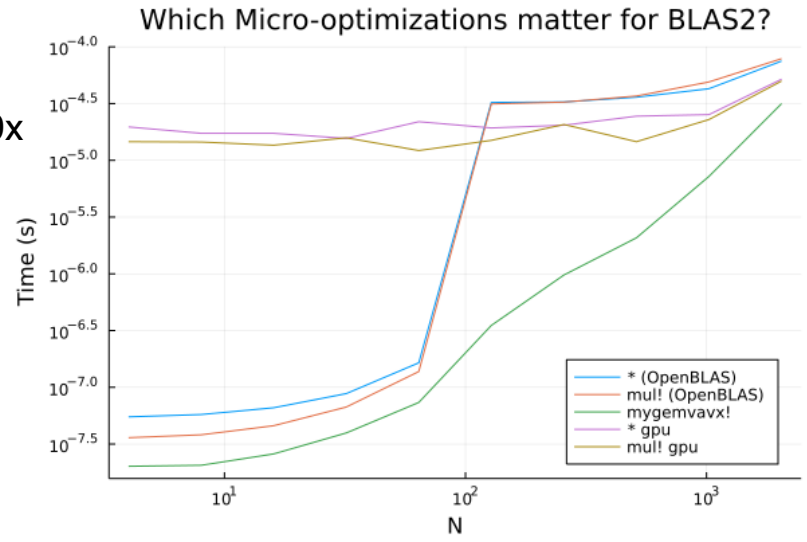


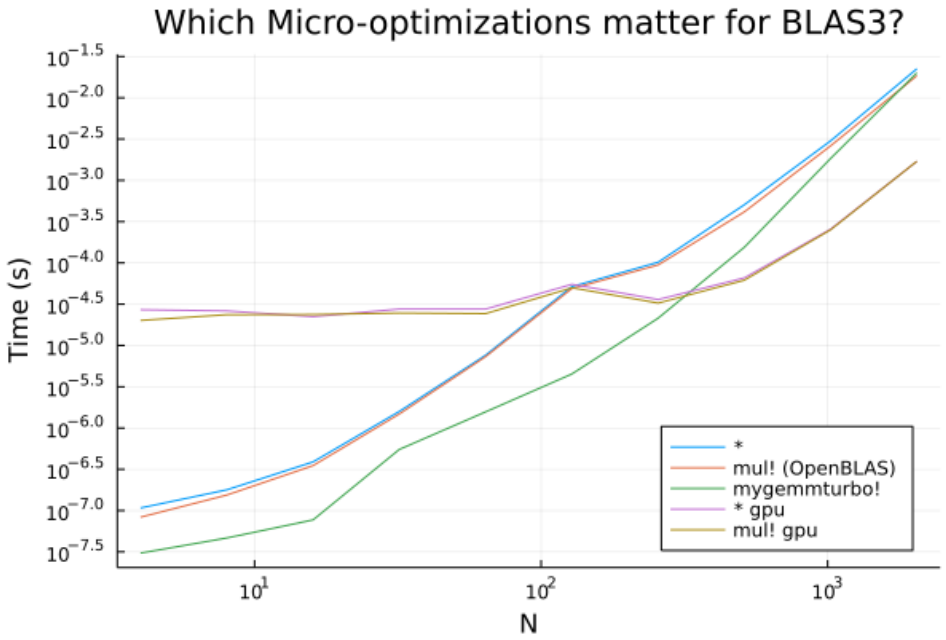
Scientific codes
O(n) and O(n^2)
operations

Mutation and
Memory management: 10x

Manual SIMD: 5x

…

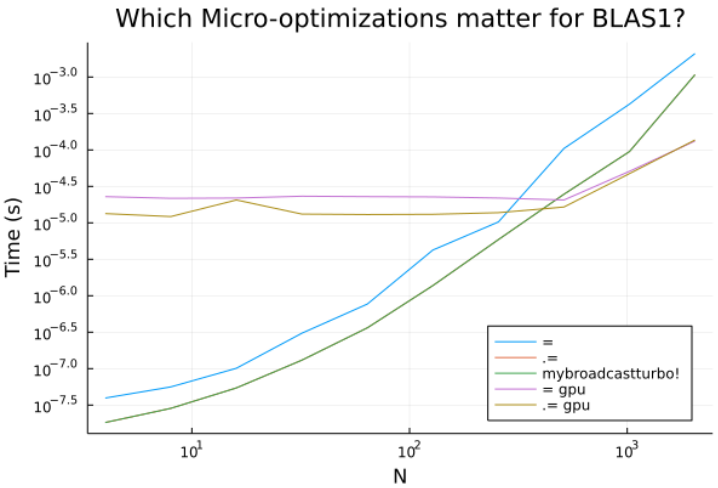




Big O(n^3) operations?
Just use a GPU
Don't worry about overhead
You're fine!

Simplest code is ~3x from optimized

What happens when you specialize computations?

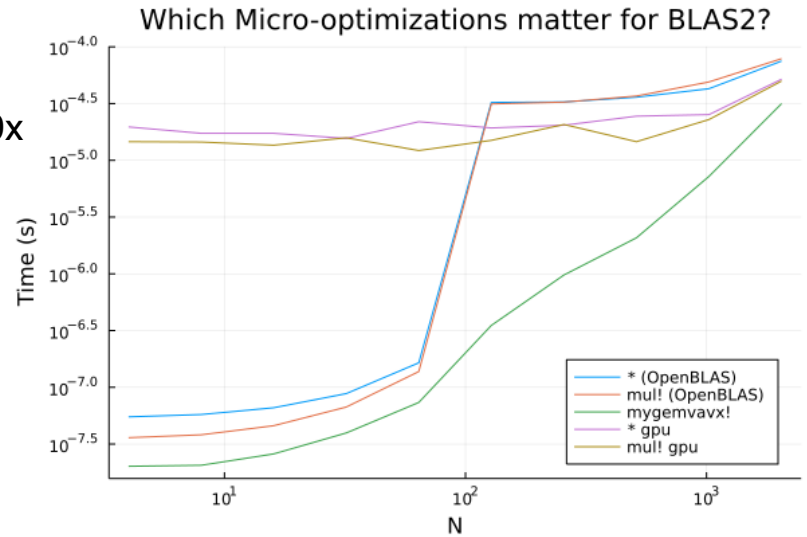


Scientific codes O(n) and O(n^2) operations

Mutation and
Memory management: 10x

Manual SIMD: 5x

…



SimpleChains.jl

Doing small network scientific machine learning in Julia on CPU 5x faster than PyTorch on GPU

(10x Jax on CPU)

Details in the release blog post

Only for size ~100 layers and below!

SimpleChains + StaticArray Neural ODEs

```
sc = SimpleChain(
                static(2),
                Activation(x -> x.^3),
                TurboDense{true}(tanh, static(50)),
                TurboDense{true}(identity, static(2))
            )

p_nn = SimpleChains.init_params(sc)

f(u,p,t) = sc(u,p)
```

his function is plugged into an ODE solver and the L2 loss is calculated from
ıe numerical solution and the NeuralODE output.

```
prob_nn = ODEProblem(f, u0, tspan)

function predict_neuralode(p)
    Array(solve(prob_nn,
Tsit5();p=p,saveat=tsteps,sensealg=QuadratureAdjoint(autojacvec=Zygote
VJP())))
end
```

About a 5x improvement

~1000x in a nonlinear mixed effects context

Tutorial should be up in a few days

Caveat: Requires sufficiently small ODEs (<20)

Let’s dive into some performance optimizations and see what’s required in practice on Burger’s Equation

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