

Introduction to Scientific Machine Learning

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

Scientific Machine Learning: Extrapolating where ML Does Not

But ML Can't do that?

But was also demonstrated with the LIGO Black Hole dynamics from the gravitational wave data, and many other examples!

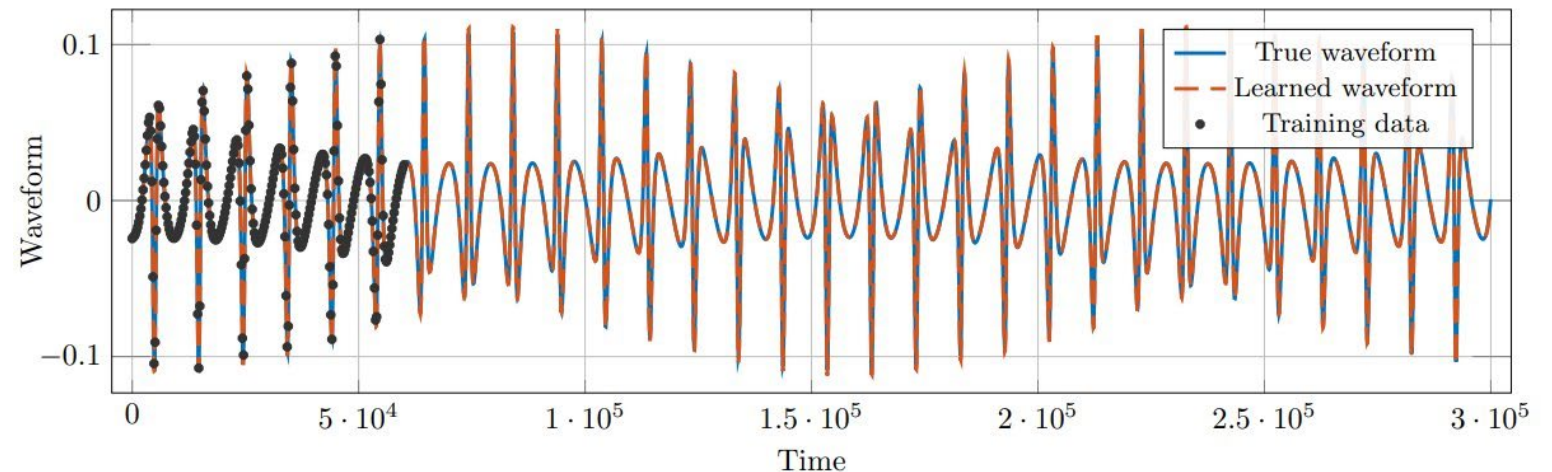
Upon denoting $\mathbf{x} = (\phi, \chi, p, e)$, we propose the following family of UDEs to describe the two-body relativistic dynamics:

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

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

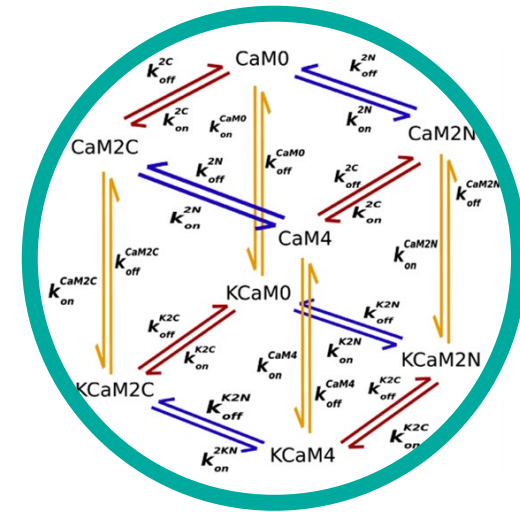
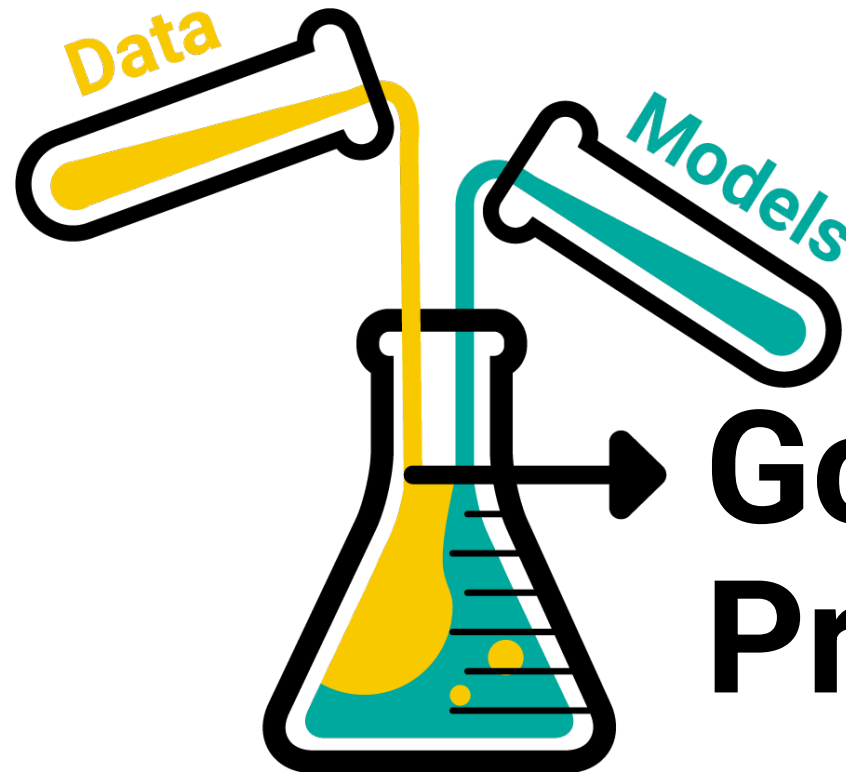
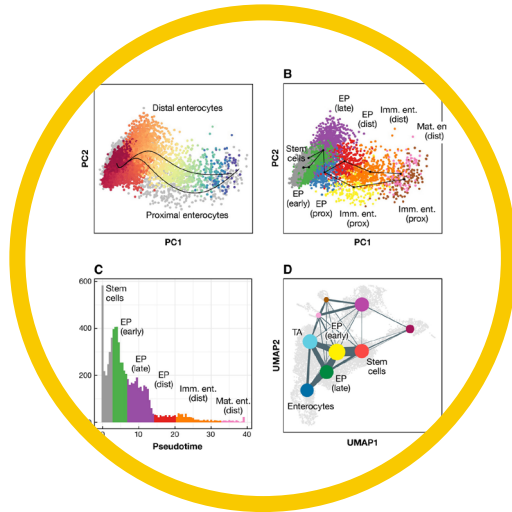
$$\dot{p} = \mathcal{F}_3(p, e), \quad (5c)$$

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



Scientific Machine Learning is model-based data-efficient machine learning

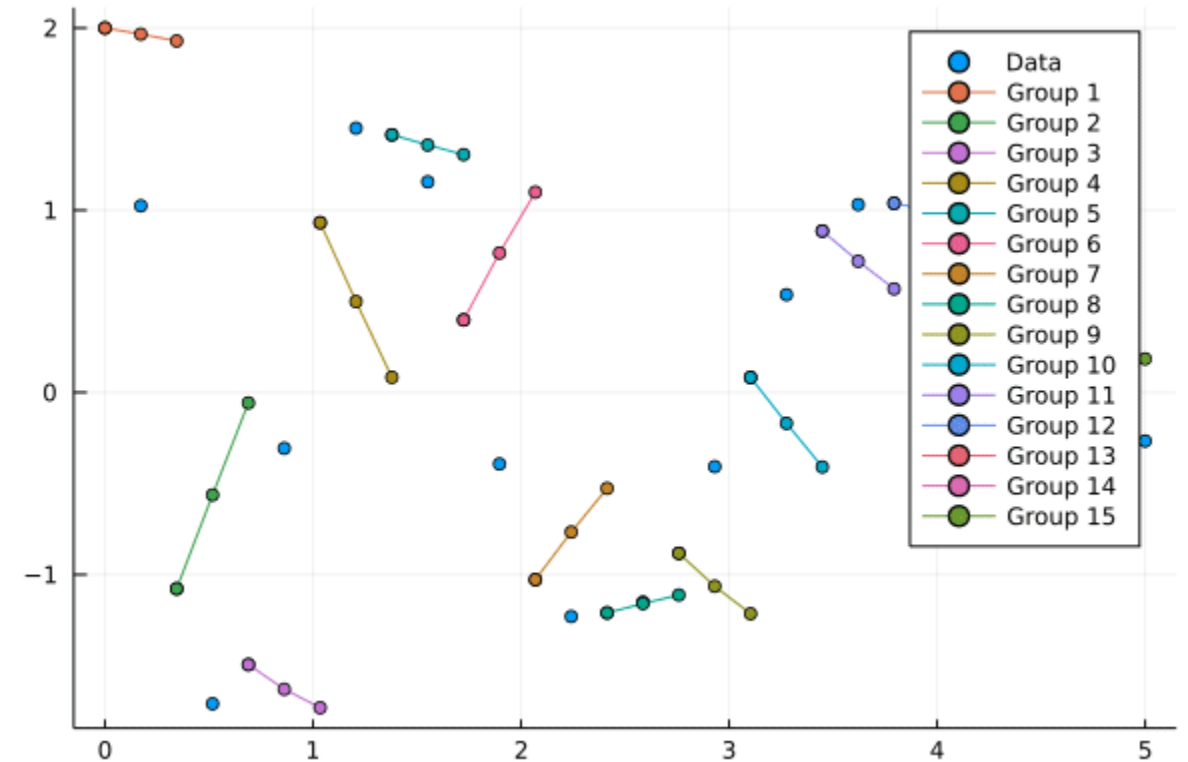
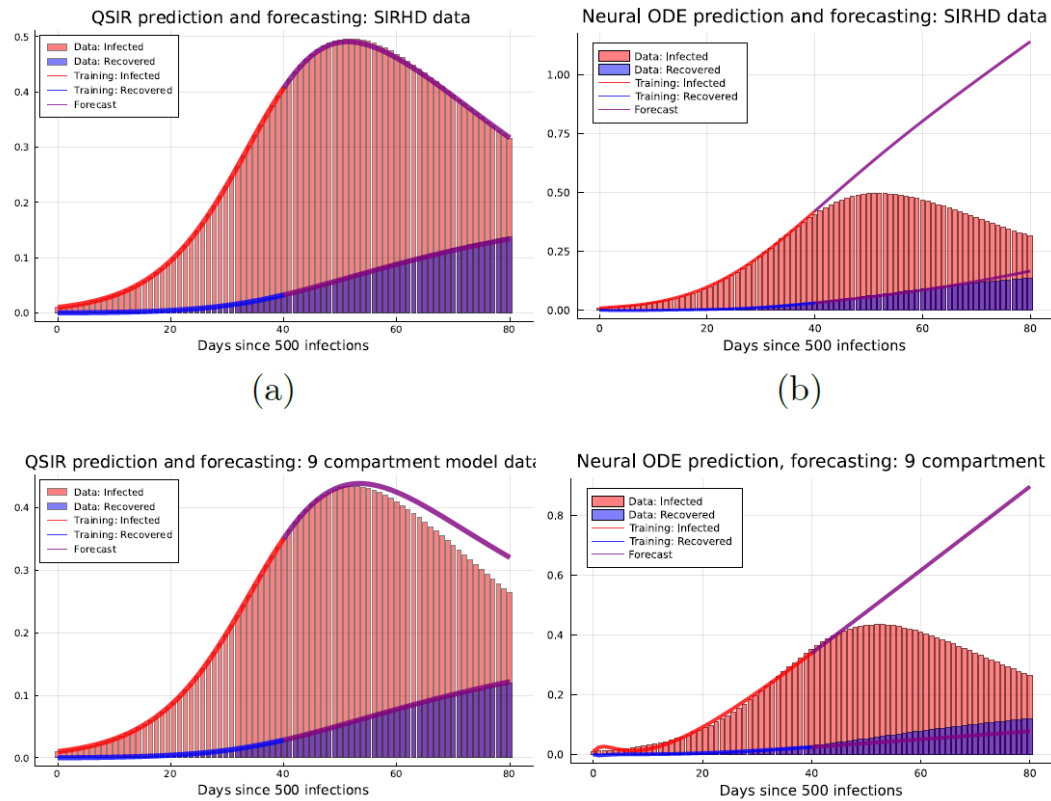
How do we simultaneously use both sources of knowledge?



Good Predictions

Outline

Mixing equation discovery into epidemic modeling workflows will revolutionize the field



1. Scientific Machine Learning Applications

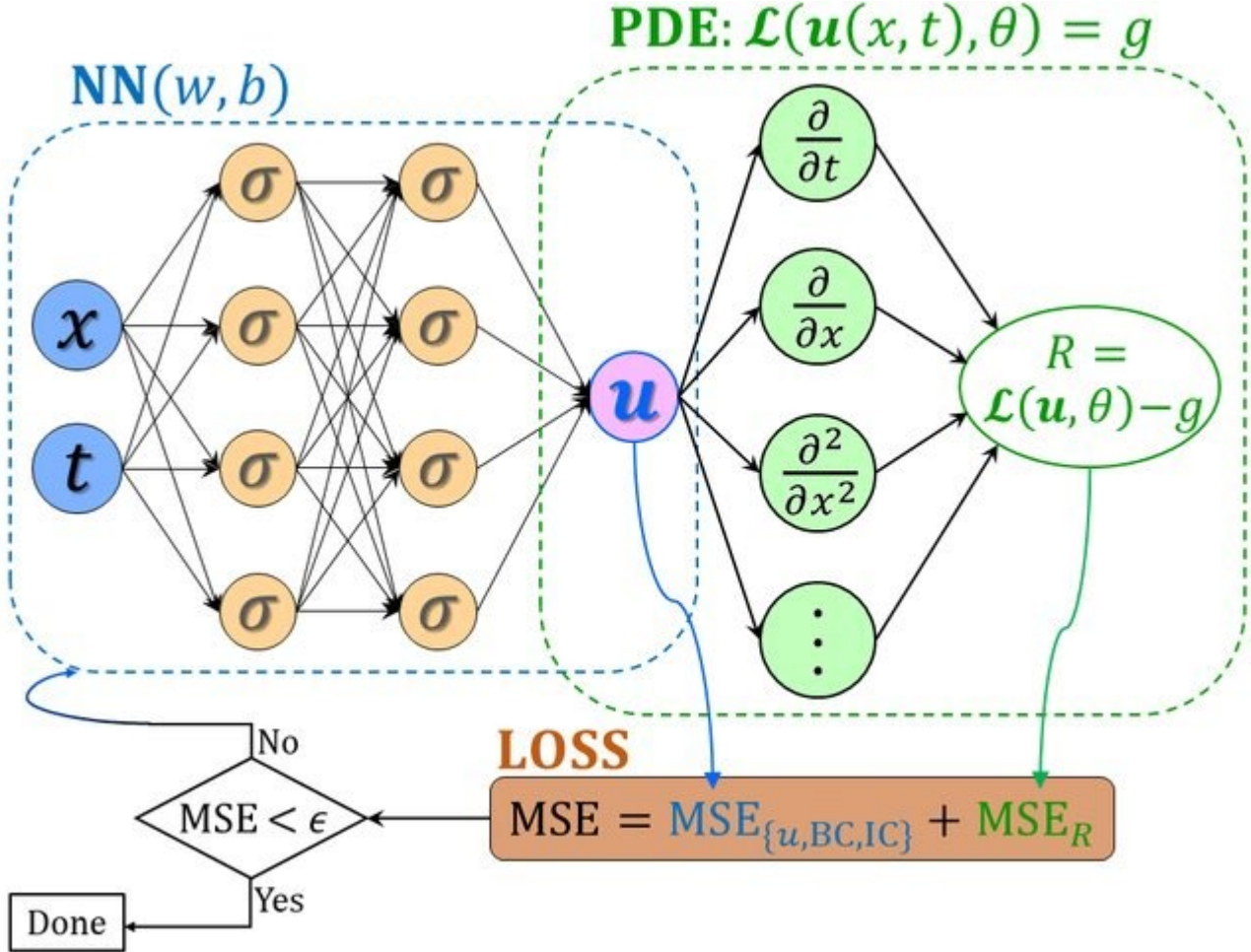
Domain knowledge with machine learning

2. Scientific Machine Learning Software

Fast and automated simulation and model discovery

First method: Physics-Informed Neural Networks

Physics-Informed Neural Networks



Approximate the PDE solution u as a neural network

Make a loss function be that its derivatives must solve the PDE

Use gradient descent

...

Now the neural network is u which solves the PDE!

ModelingToolkit's General PDE Solver: Physics-Informed Neural Networks

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

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

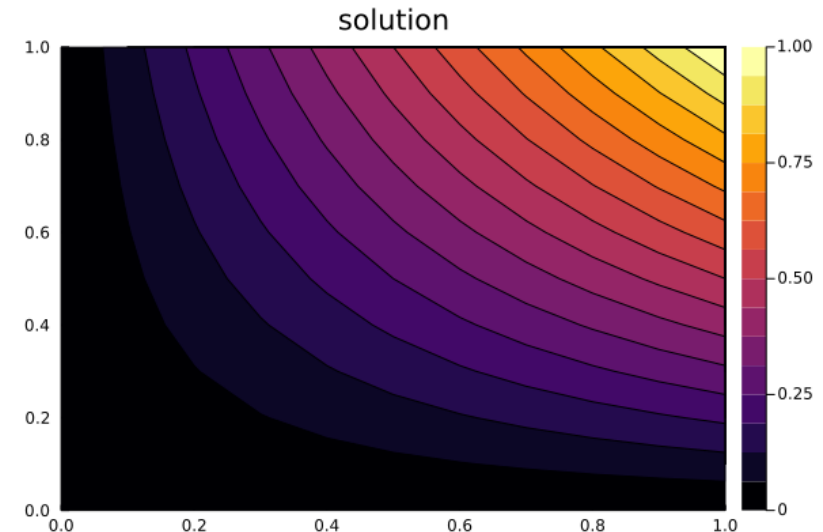
# 2D PDE
eq = Dxx(u(x,y)) + Dyy(u(x,y)) ~ -sin(pi*x)*sin(pi*y)

# 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)]

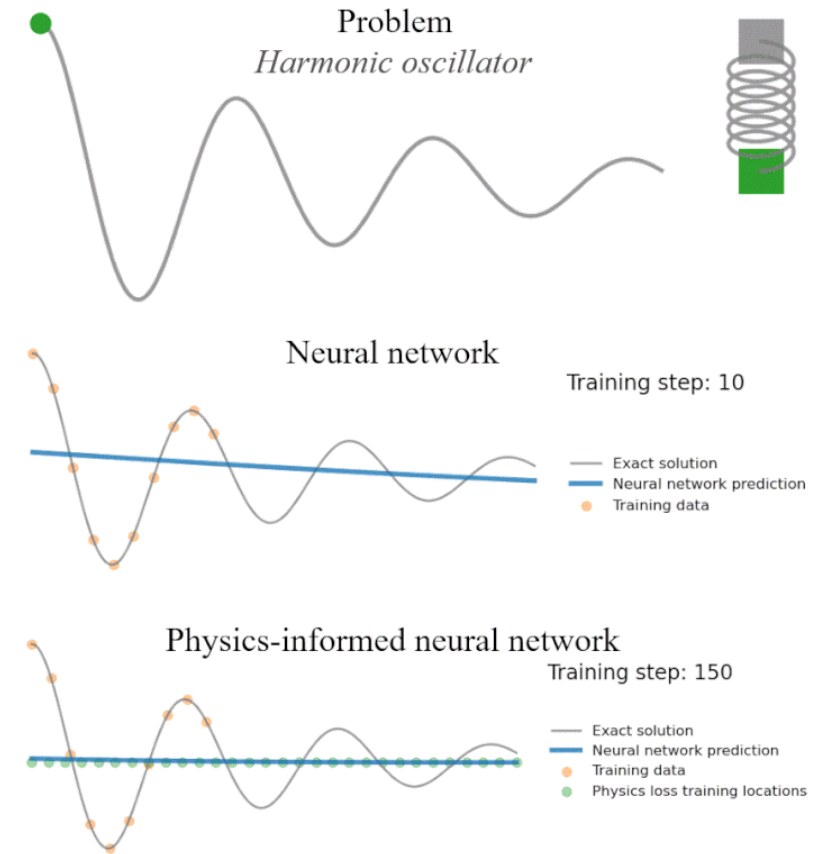
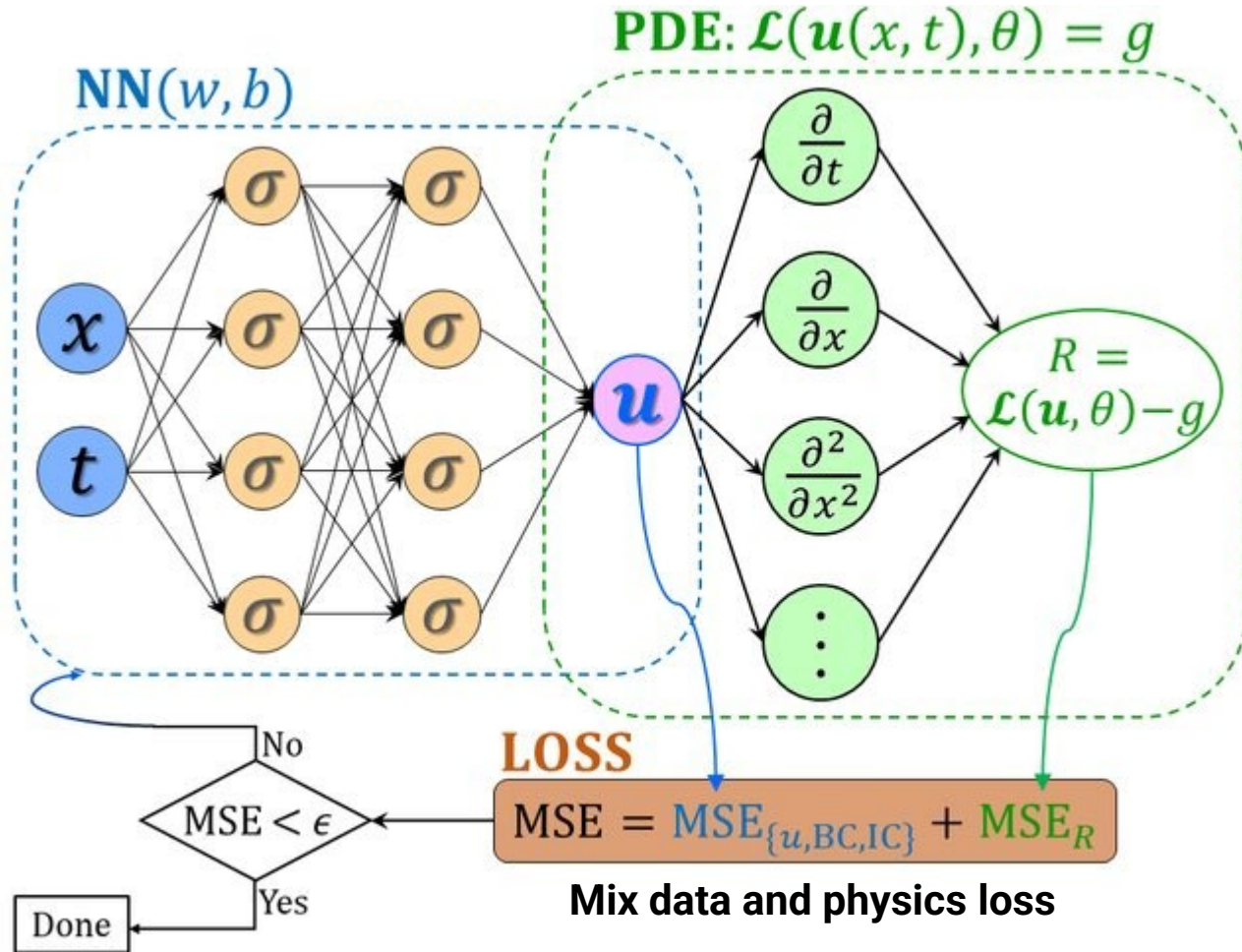
# Space and time domains
domains = [x ∈ Interval(0.0,1.0),
           y ∈ Interval(0.0,1.0)]
pde_system = PDESystem(eq,bcs,domains,[x,y],[u])
```

Easy and Customizable PINN PDE Solving with NeuralPDE.jl, JuliaCon 2021

```
# Neural network
dim = 2 # number of dimensions
chain = FastChain(FastDense(dim,16,Flux.σ),
                  FastDense(16,16,Flux.σ),FastDense(16,1))
# Discretization
dx = 0.05
discretization = PhysicsInformedNN(chain,GridTraining(dx))
prob = discretize(pde_system,discretization)
#Optimizer
opt = Optim.BFGS()
res = GalacticOptim.solve(prob, opt, maxiters=1000)
```

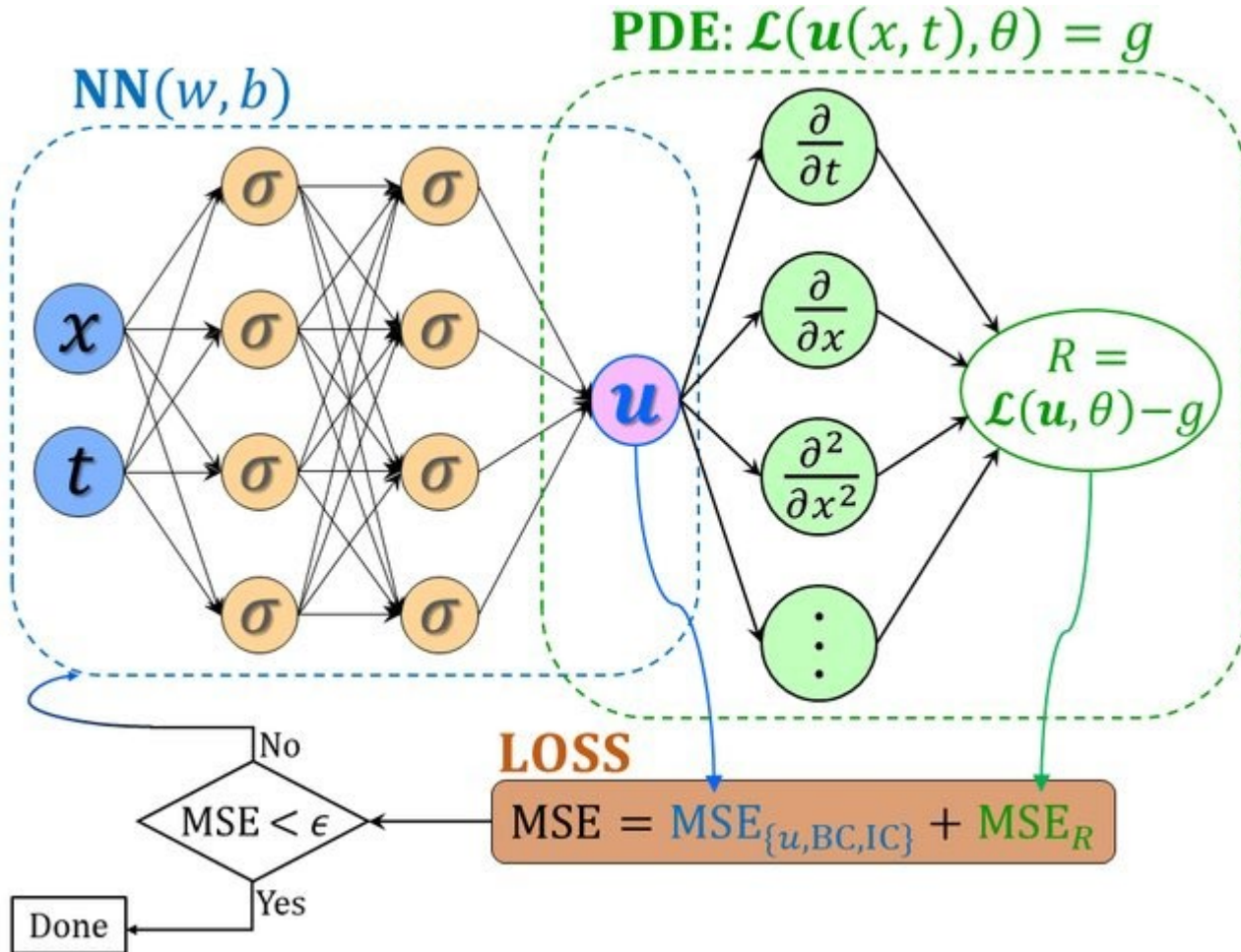


Why Use Physics-Informed Neural Networks?



Outperforms standard machine learning

Why Use Physics-Informed Neural Networks?



Generality: Easily works for any PDE

Simplicity: Doesn't require any simulation methods, just a neural network and automatic differentiation

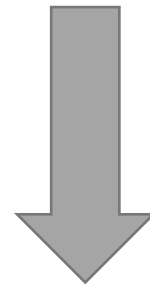
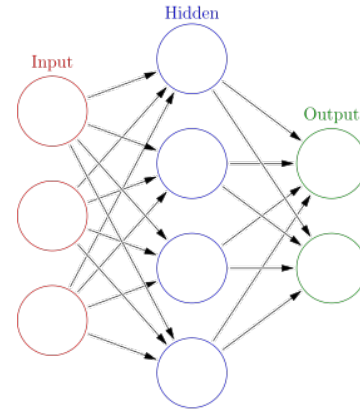
Why not use a PINN?

Performance.

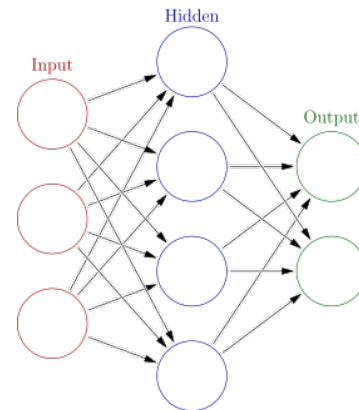
Second method: Differentiable Simulation and Universal Differential Equations

Universal (Approximator) Differential Equations

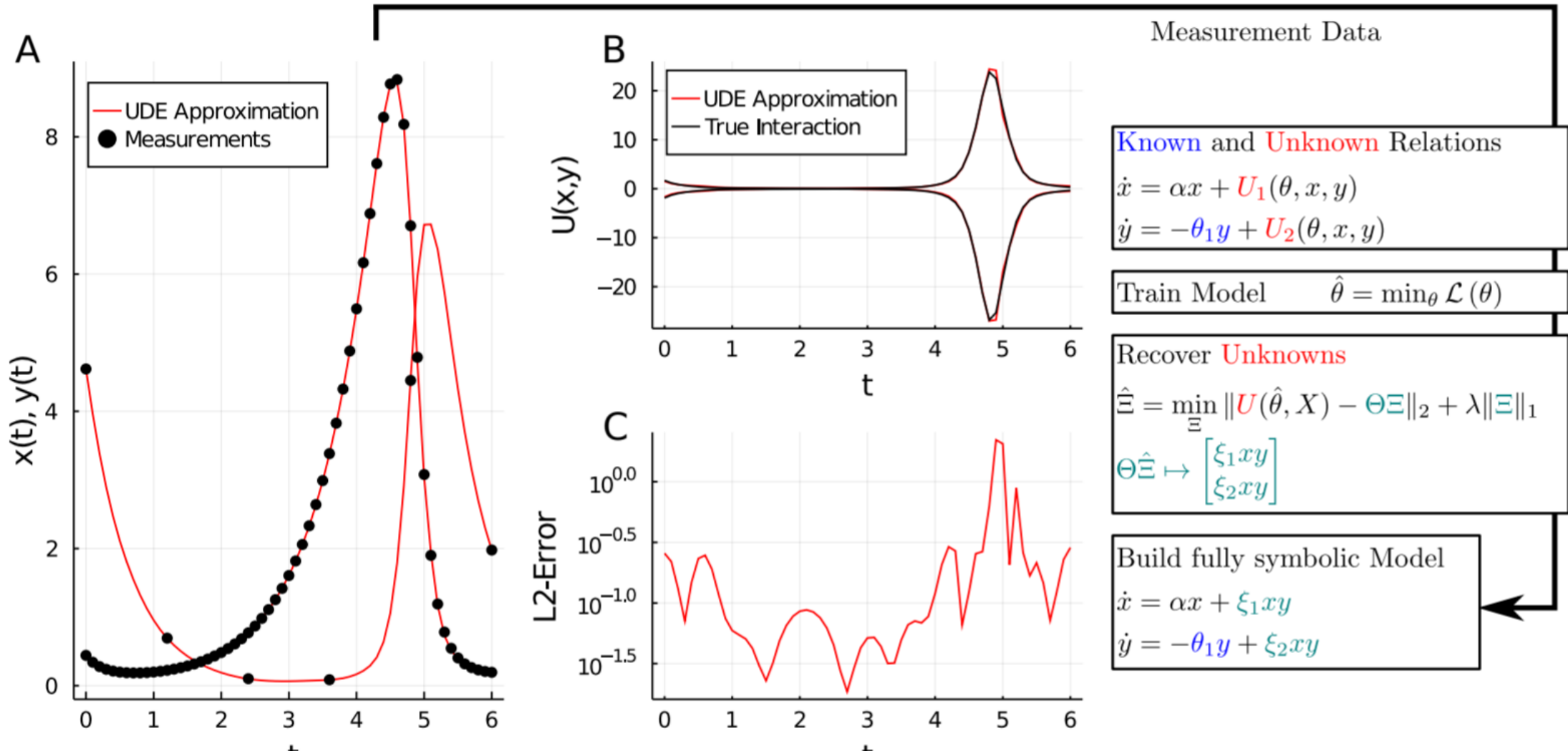
$$u' = f(u, \text{Hidden}, t)$$



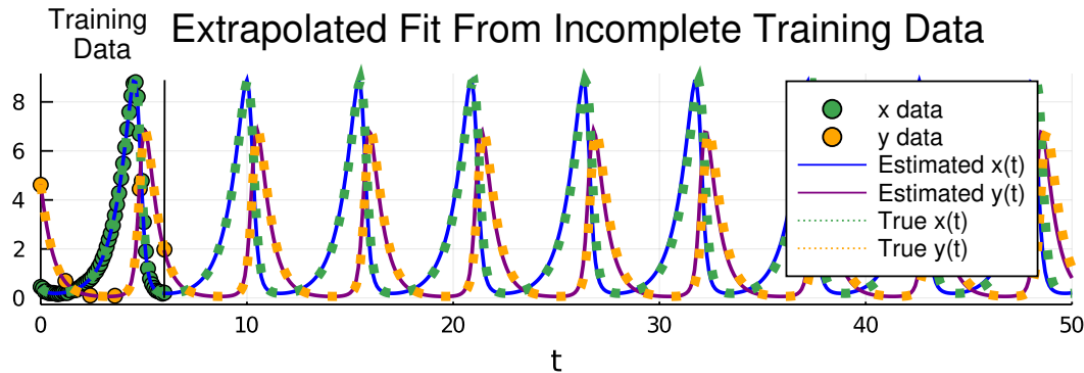
$$\begin{aligned} x' &= \alpha x + \\ y' &= -\beta y + \end{aligned}$$



Universal (Approximator) Differential Equations



UODEs show accurate extrapolation and generalization



Extrapolation is successful in Lotka-Volterra...

But was also demonstrated with the LIGO Black Hole dynamics from the gravitational wave data, and many other examples!

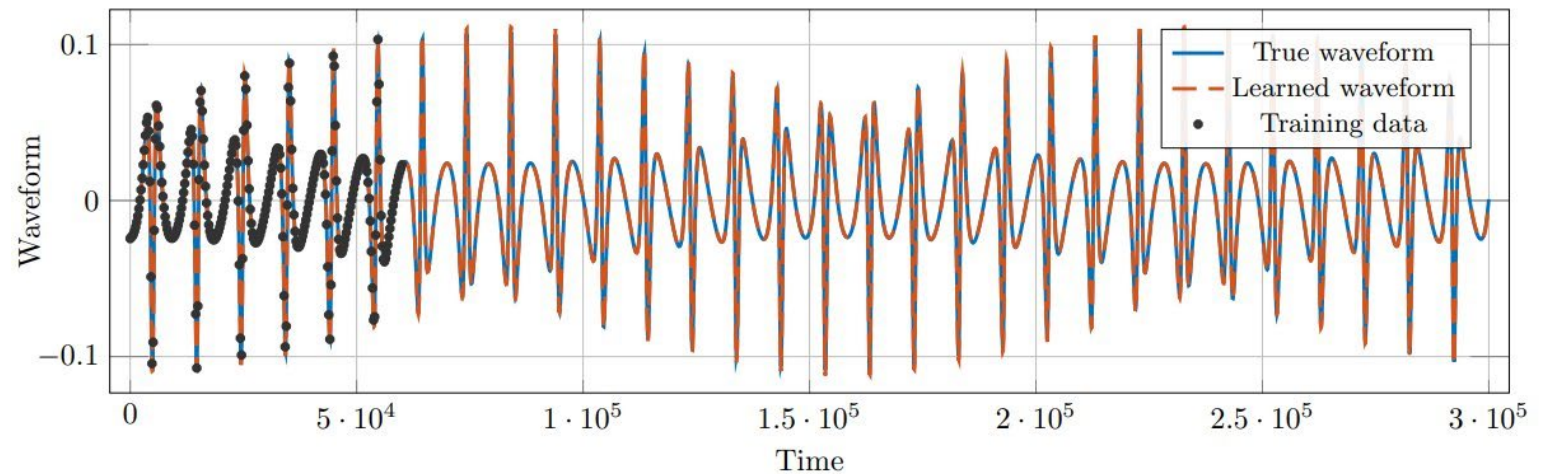
Upon denoting $\mathbf{x} = (\phi, \chi, p, e)$, we propose the following family of UDEs to describe the two-body relativistic dynamics:

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

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

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

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



SciML Shows how to build Earthquake-Safe Buildings

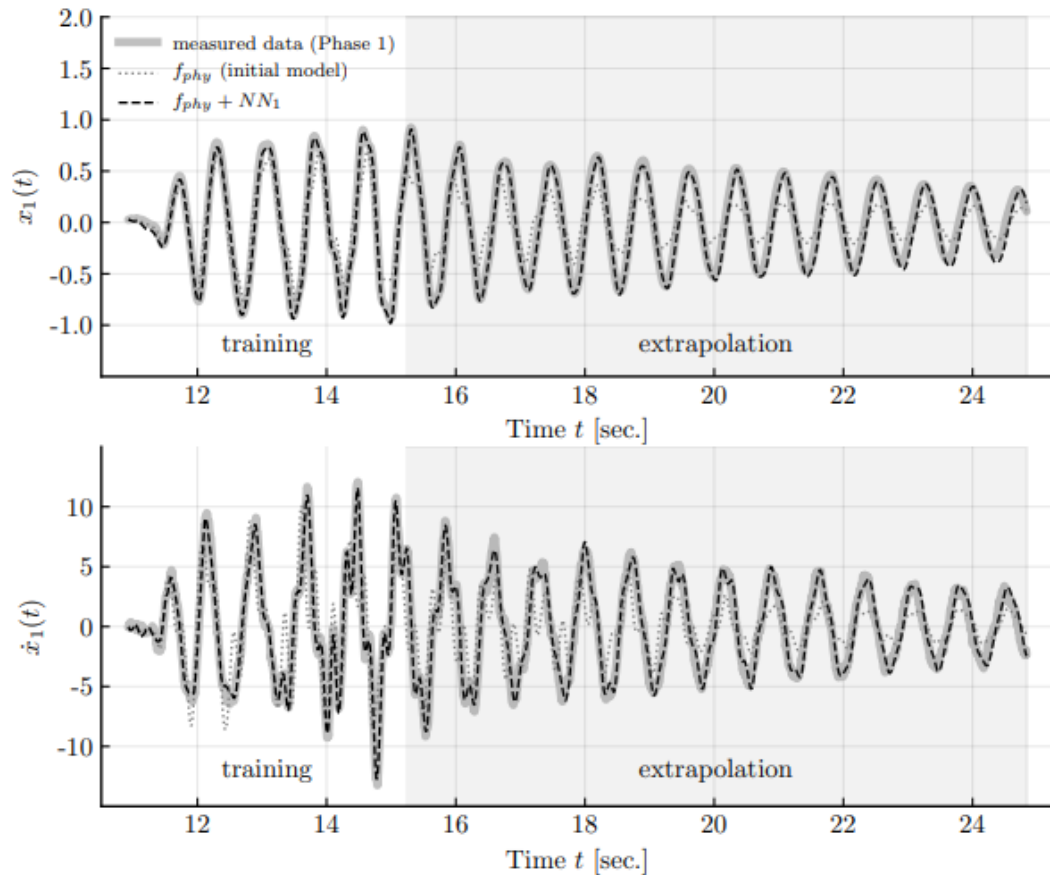


Figure 12: Comparison of time history of the response for displacement $x_1(t)$ and velocity $\dot{x}_1(t)$ for the NSD experiment (Phase 1).

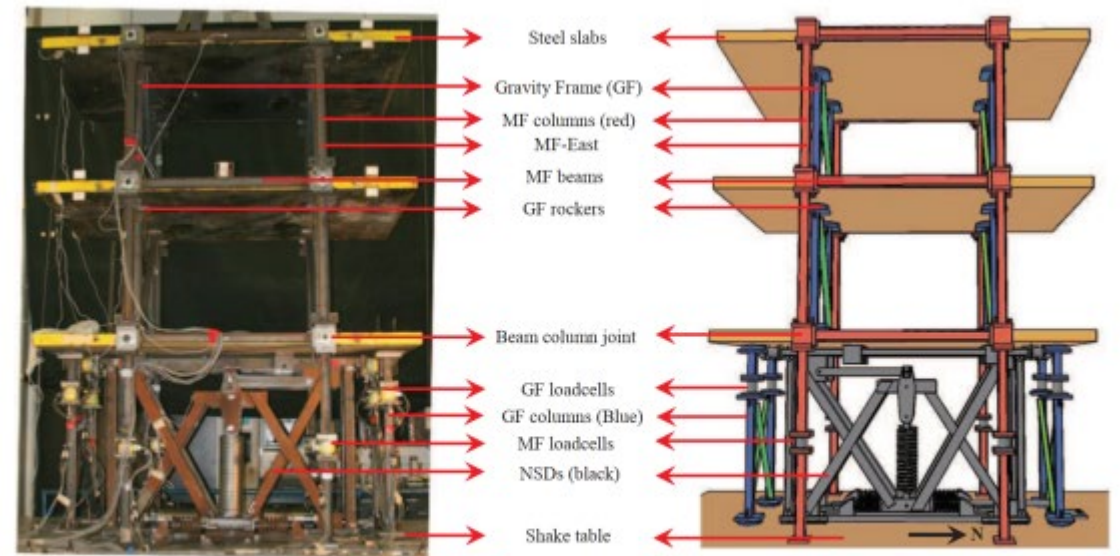


Figure 10: The structural system equipped with a negative stiffness device in between the first floor and the shake table.

Scientific machine learning for earthquake-safe buildings

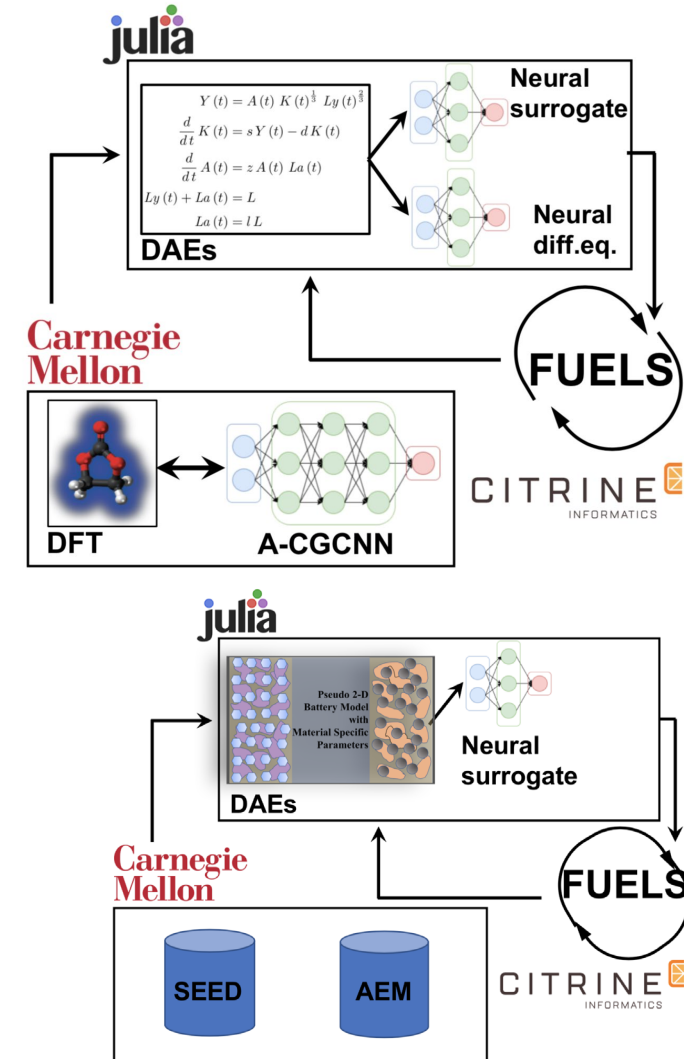
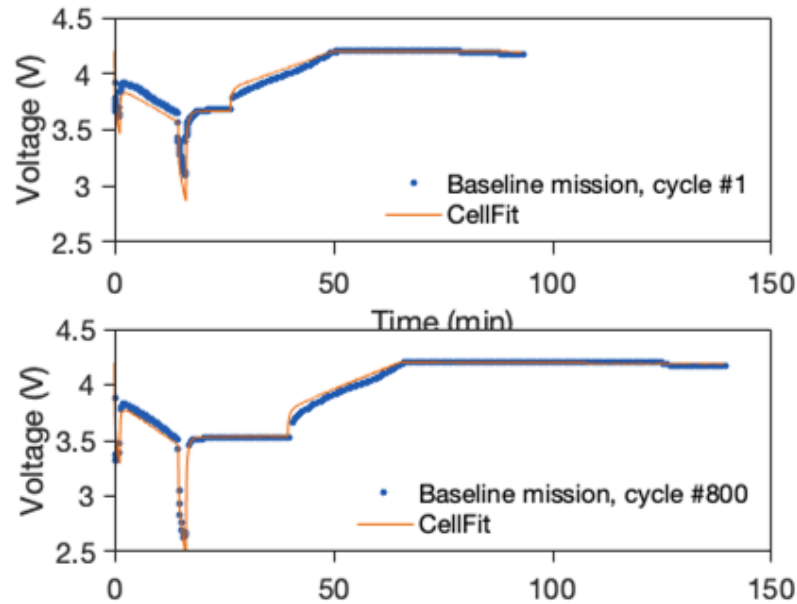
Structural identification with physics-informed neural ordinary differential equations

Lai, Zhilu, Mylonas, Charilaos, Nagarajaiah, Satish, Chatzi, Eleni

SciML for Predicts Longer Lasting Battery Materials

Researches at CMU used Universal Differential Equations to improve models of Battery Degradation to Suggest Better Battery Materials

Universal Battery Performance and Degradation Model for Electric Aircraft
 Alexander Bills, Shashank Sripad, William L. Fredericks, Matthew Guttenberg, Devin Charles, Evan Frank, Venkatasubramanian Viswanathan



SciML for Generates Predictive Combustion Models

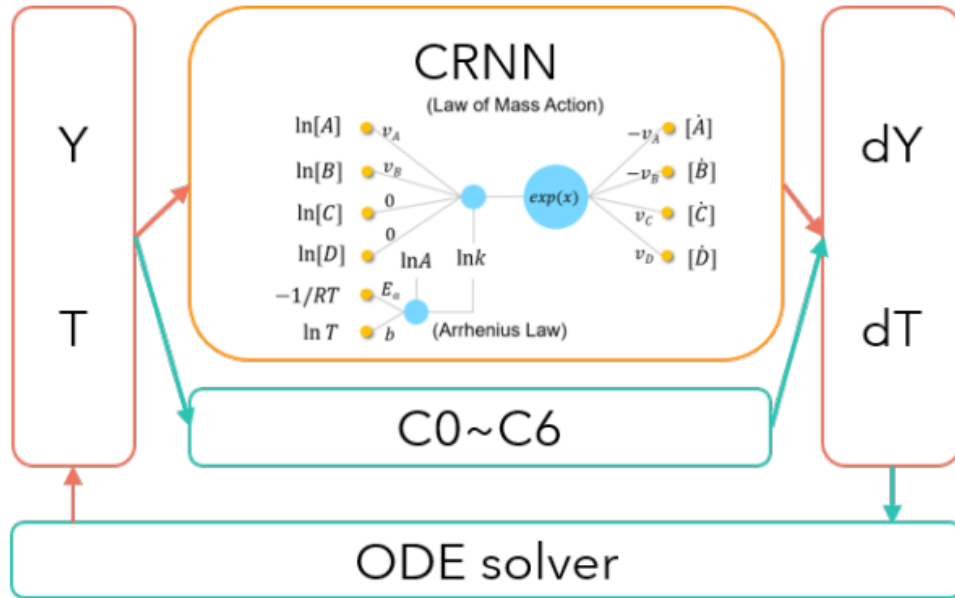
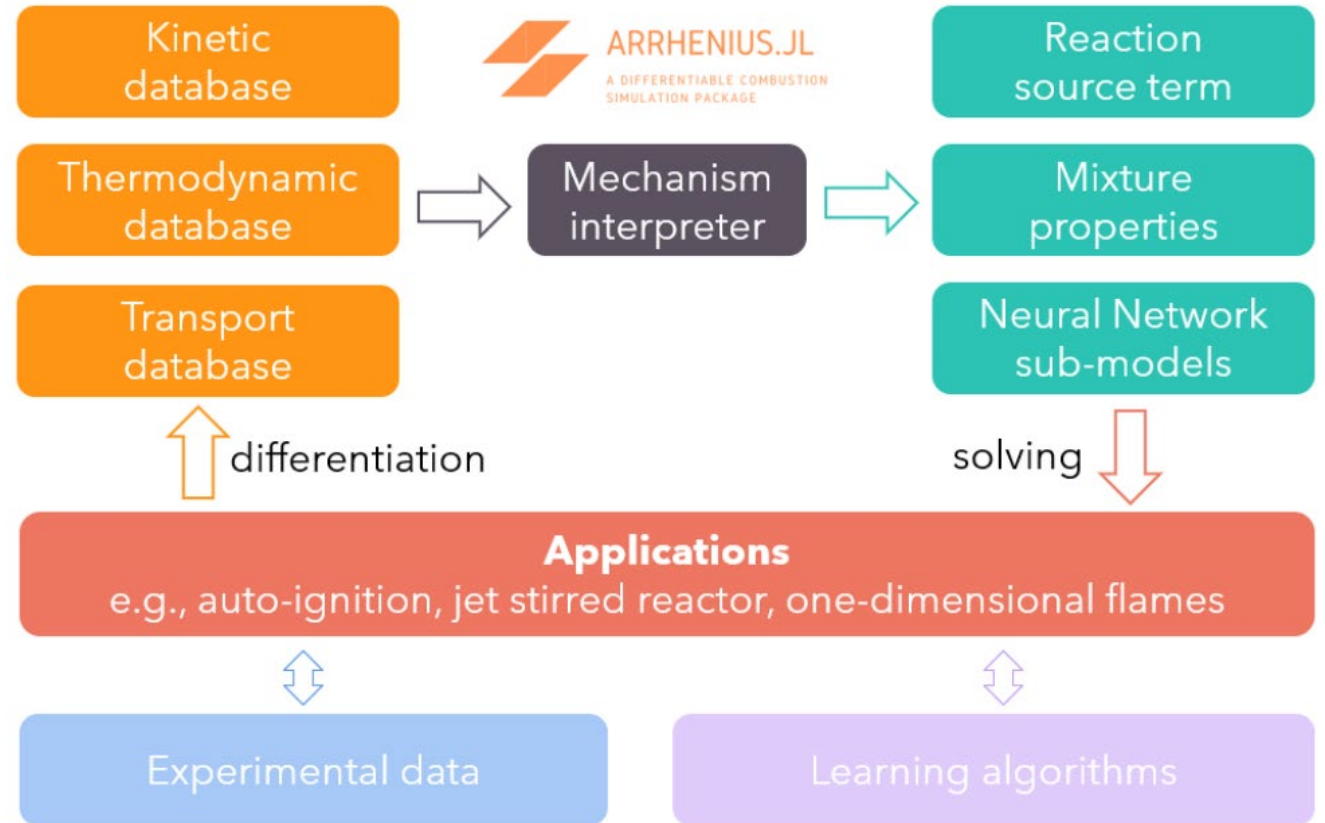


Figure 8: Schematic showing the structure of the CRNN-HyChem approach.

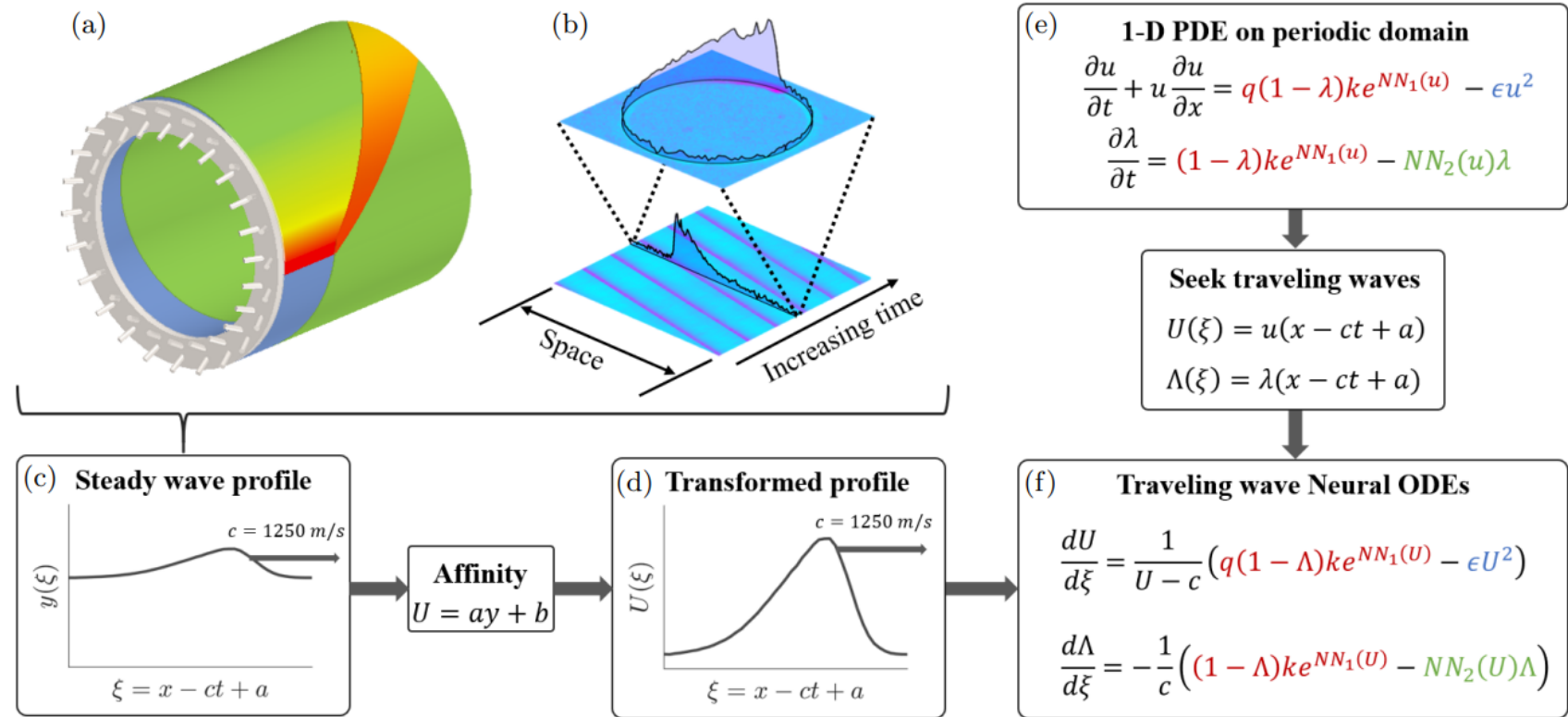


Fast automated learning of combustion models for accelerated engineering

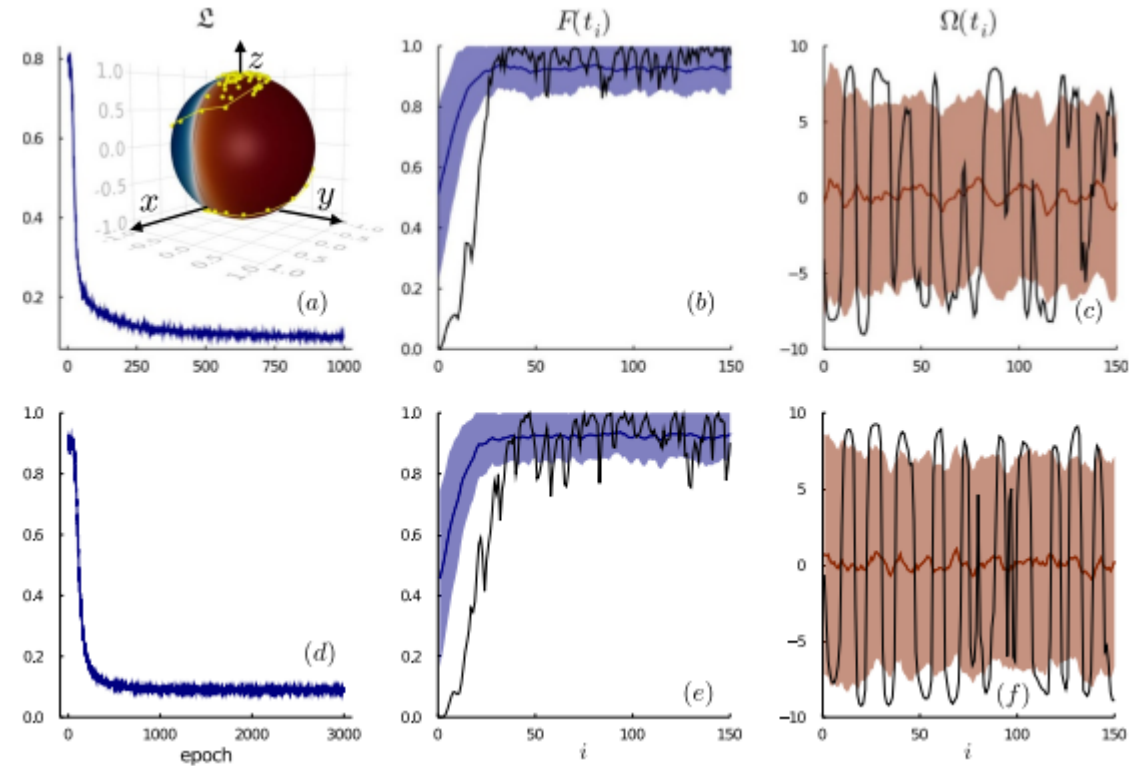
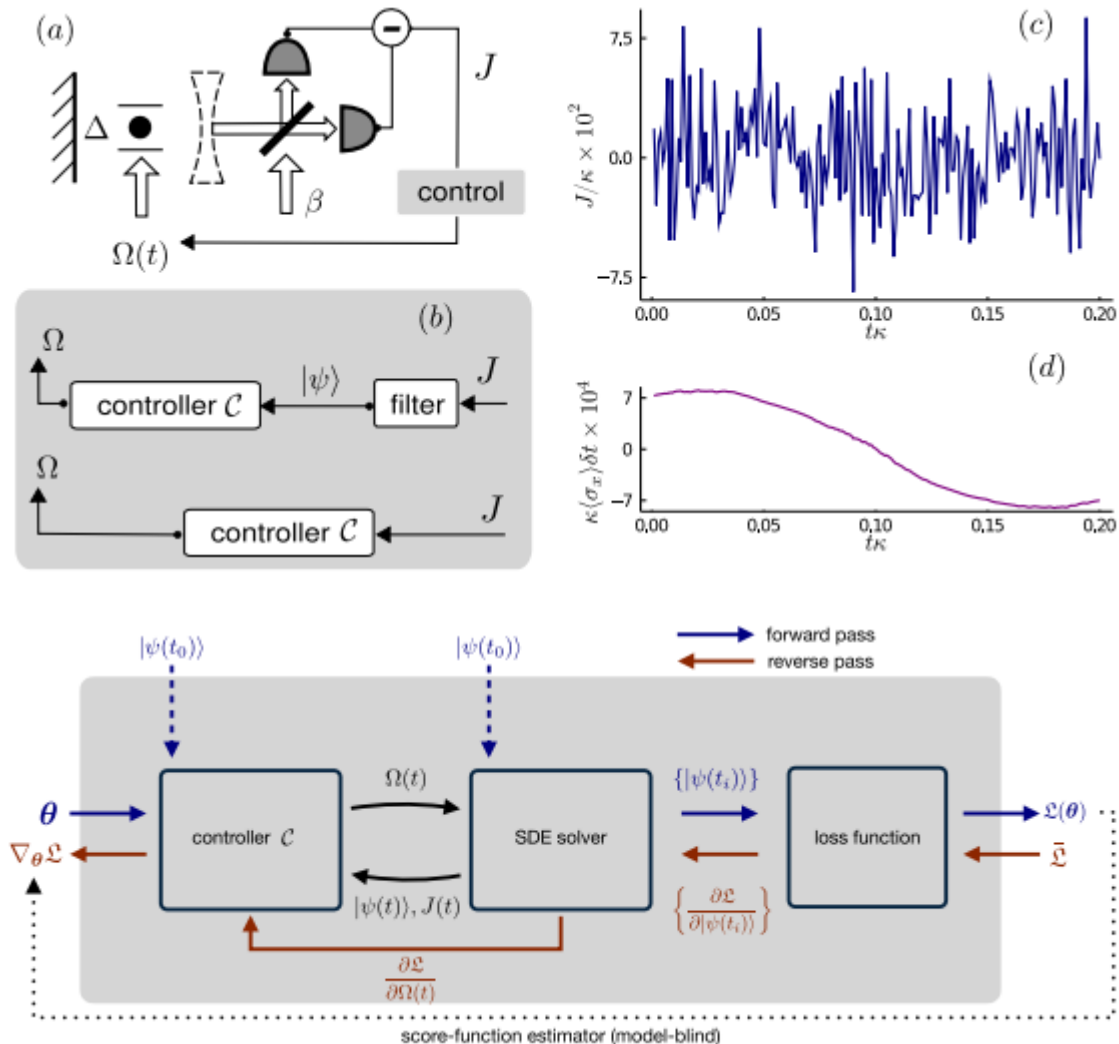
Arrhenius.jl: A Differentiable Combustion Simulation Package
 Weiqi Ji, Xingyu Su, Bin Pang, Sean Joseph Cassady, Alison M. Ferris, Yajuan Li, Zhuyin Ren, Ronald Hanson, Sili Deng

SciML for Generates Predictive Models of New Propulsion Devices

SciML predicting the properties of new propulsion devices



SciML for Controlling Qubit Preparation in Quantum Circuits



Future quantum computers will be made possible by SciML

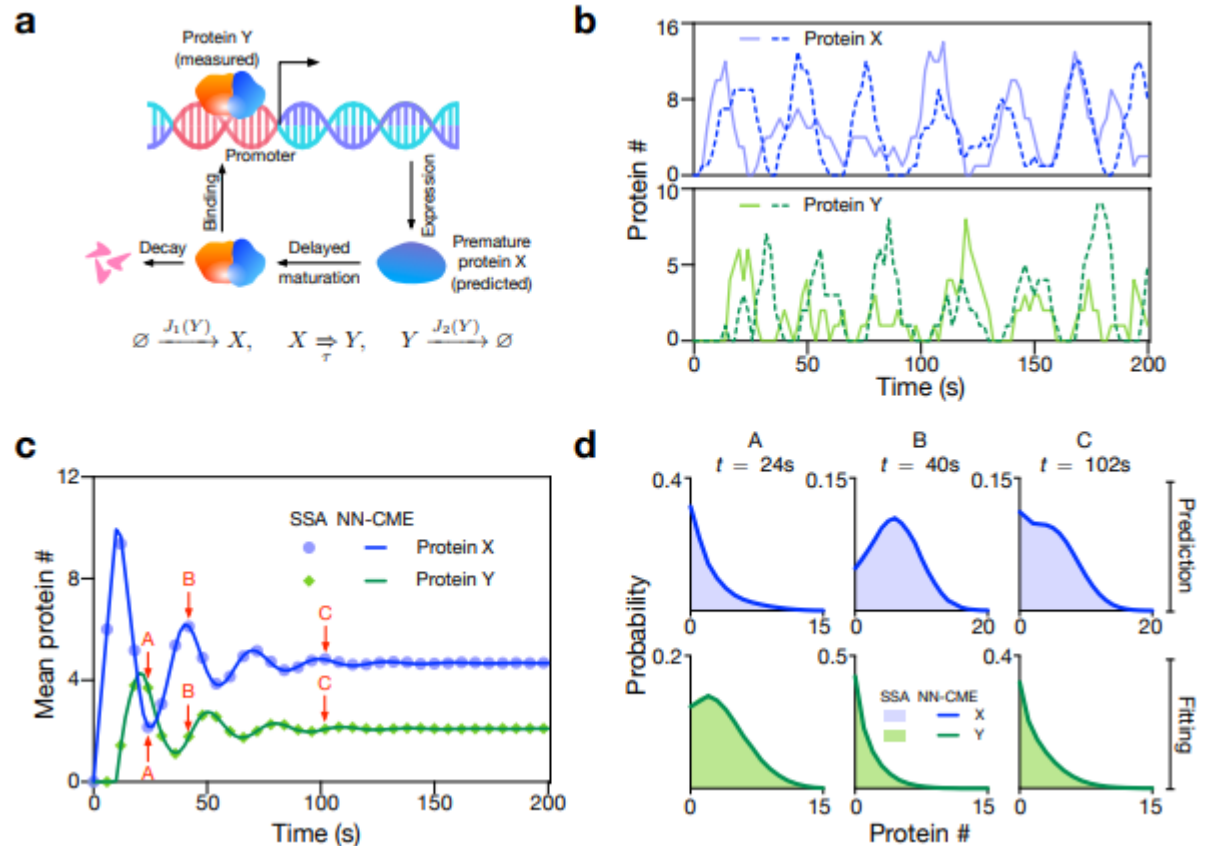
Control of stochastic quantum dynamics by differentiable programming
 Frank Schäfer, Pavel Sekatski, Martin Koppenhöfer, Christoph Bruder and Michal Kloc

SciML for Builds Models of Biological Systems

Better models of gene expression to understand biological systems

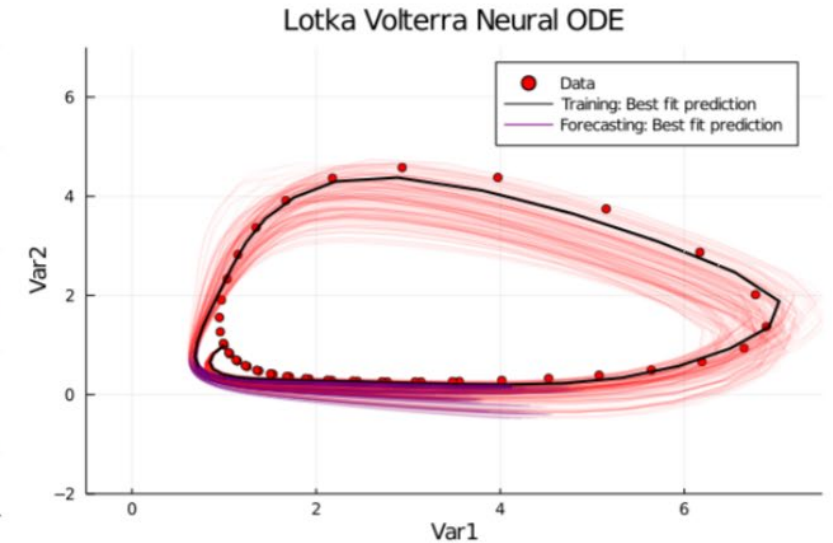
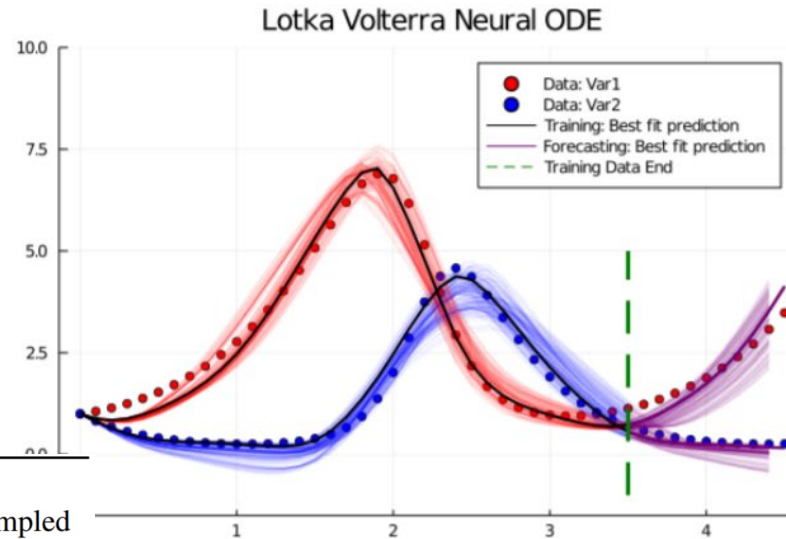
Neural network aided approximation and parameter inference of stochastic models of gene expression

Qingchao Jiang, Xiaoming Fu, Shifu Yan, Runlai Li, Wenli Du, Zhixing Cao, Feng Qian, Ramon Grima



Bayesian UODEs: Knowledge-Enhanced Model Discovery with UQ

Result: Probability of Missing Mechanisms

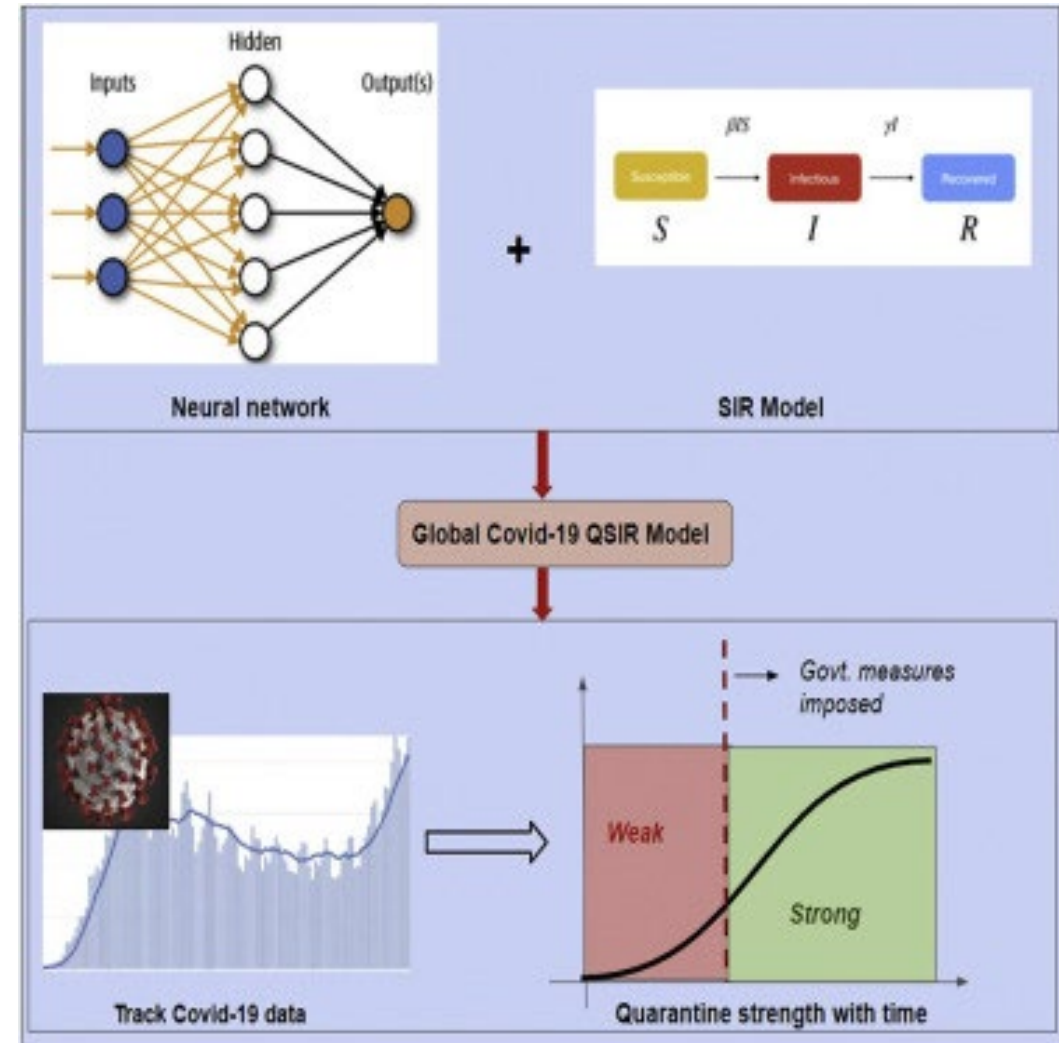
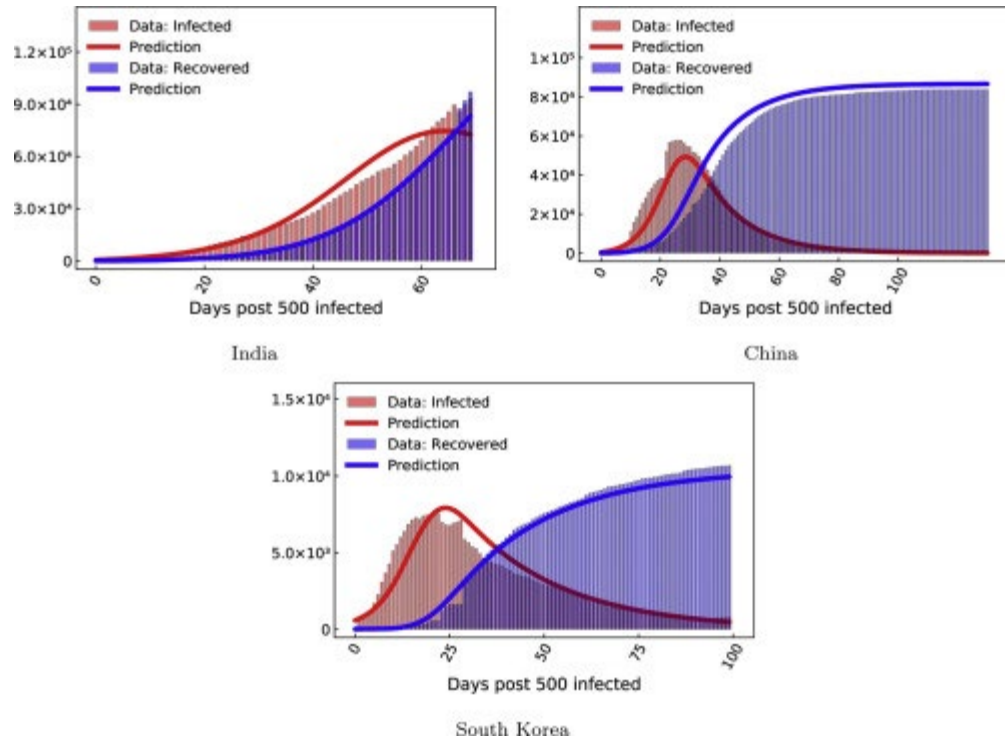


```
function lotka_volterra!(du, u, p, t)
    x, y = u
    α, β, δ, γ = p
    du[1] = dx = α*x - β*x*y
    du[2] = dy = -δ*y + γ*x*y
end
```



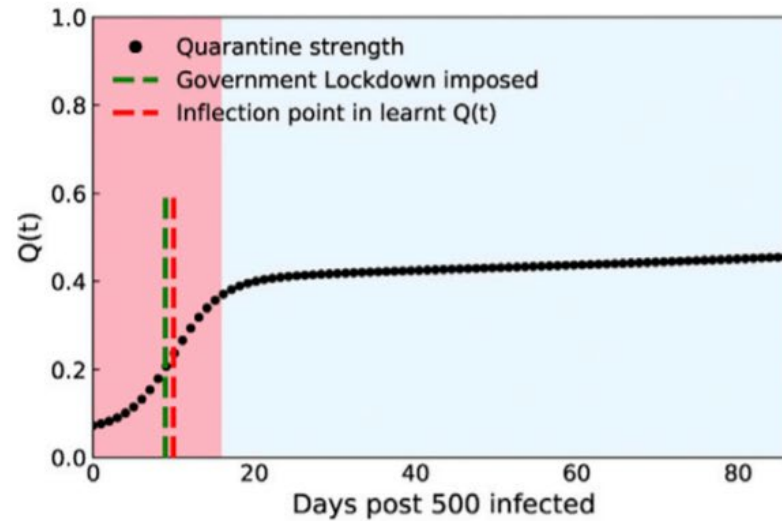
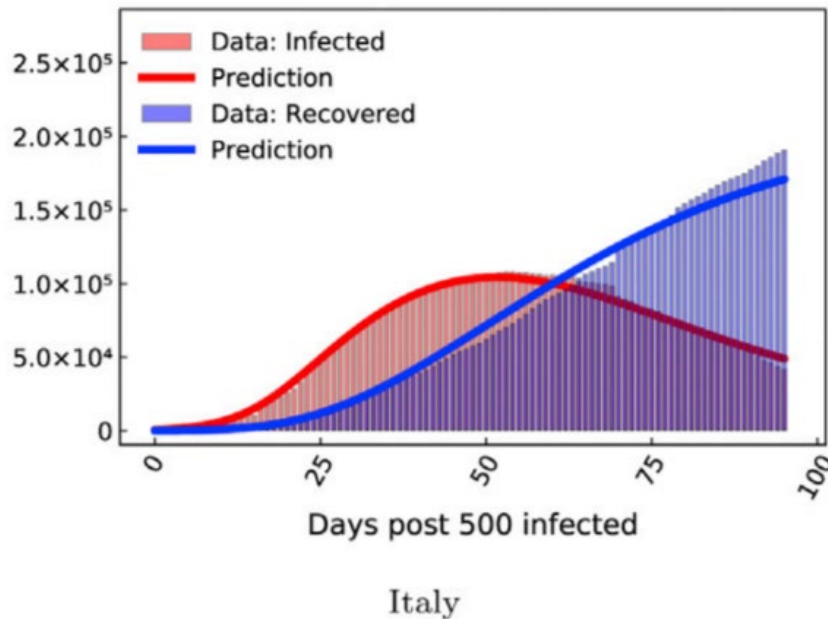
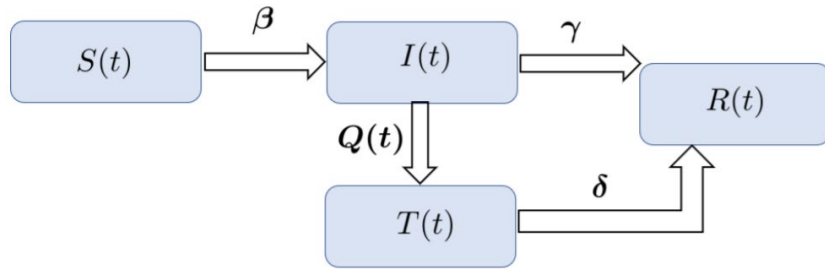
λ	Number of Active terms	Dominant terms	Error	Mean AIC score	% sampled
0.01	9	$u_1^2, u_2^2, u_1 u_2$ $u_1^2 u_2^2, u_1^2 u_2, u_2^2 u_1$ $u_1 u_2, \text{const}$	0.765	40.4	100
0.1	9	$u_1^2, u_2^2, u_1 u_2$ $u_1^2 u_2^2, u_1^2 u_2, u_2^2 u_1$ $u_1 u_2, \text{const}$	0.764	35	100
1	5	u_1^2, u_2^2, u_2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

Demonstration of UDE Epidemic Models

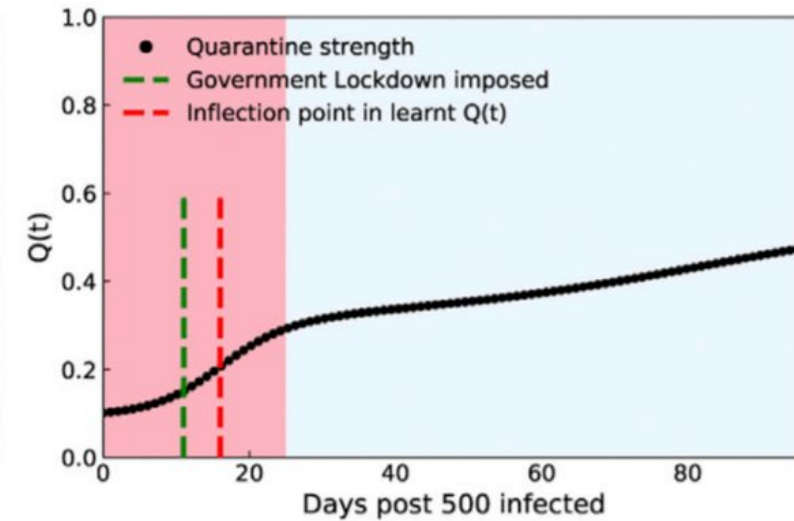


Dandekar, Raj, Chris Rackauckas, and George Barbastathis. "A machine learning aided global diagnostic and comparative tool to assess effect of quarantine control in Covid-19 spread." *Cell Patterns* (2020).

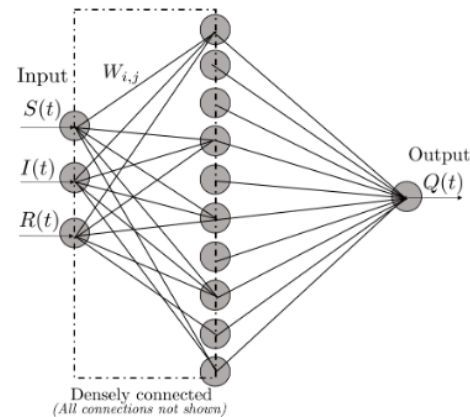
QSIR Predicts Quarantine Measure Evolution



Spain

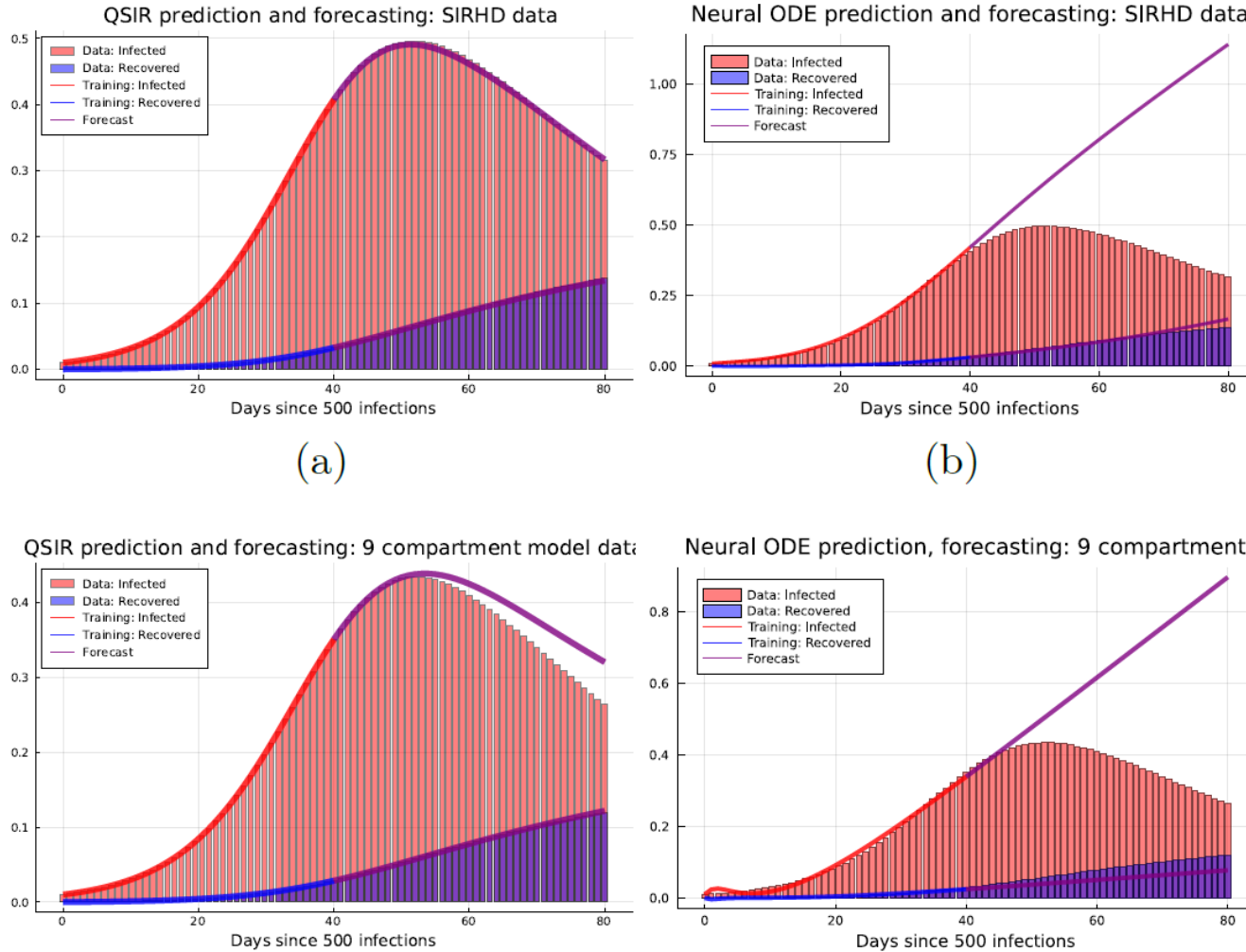


Italy



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

QSIR is robust to having small amounts of sample data



QSIR is robust to having small amounts of sample data

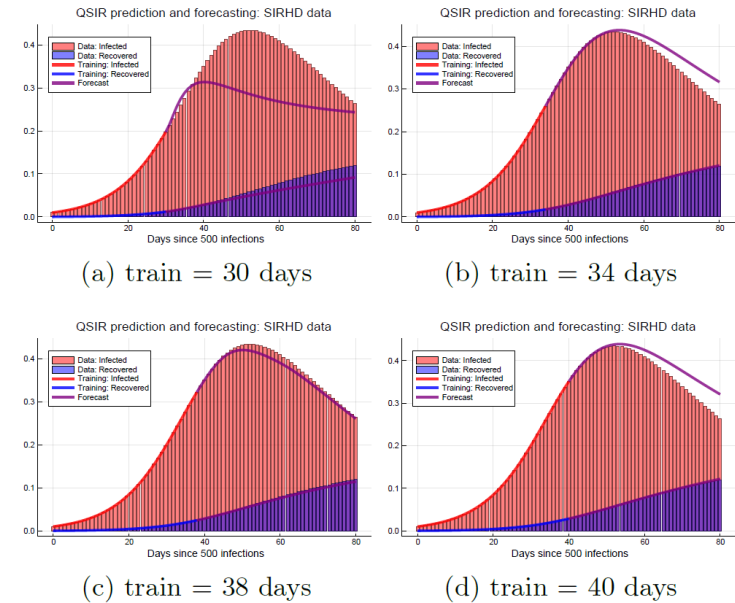


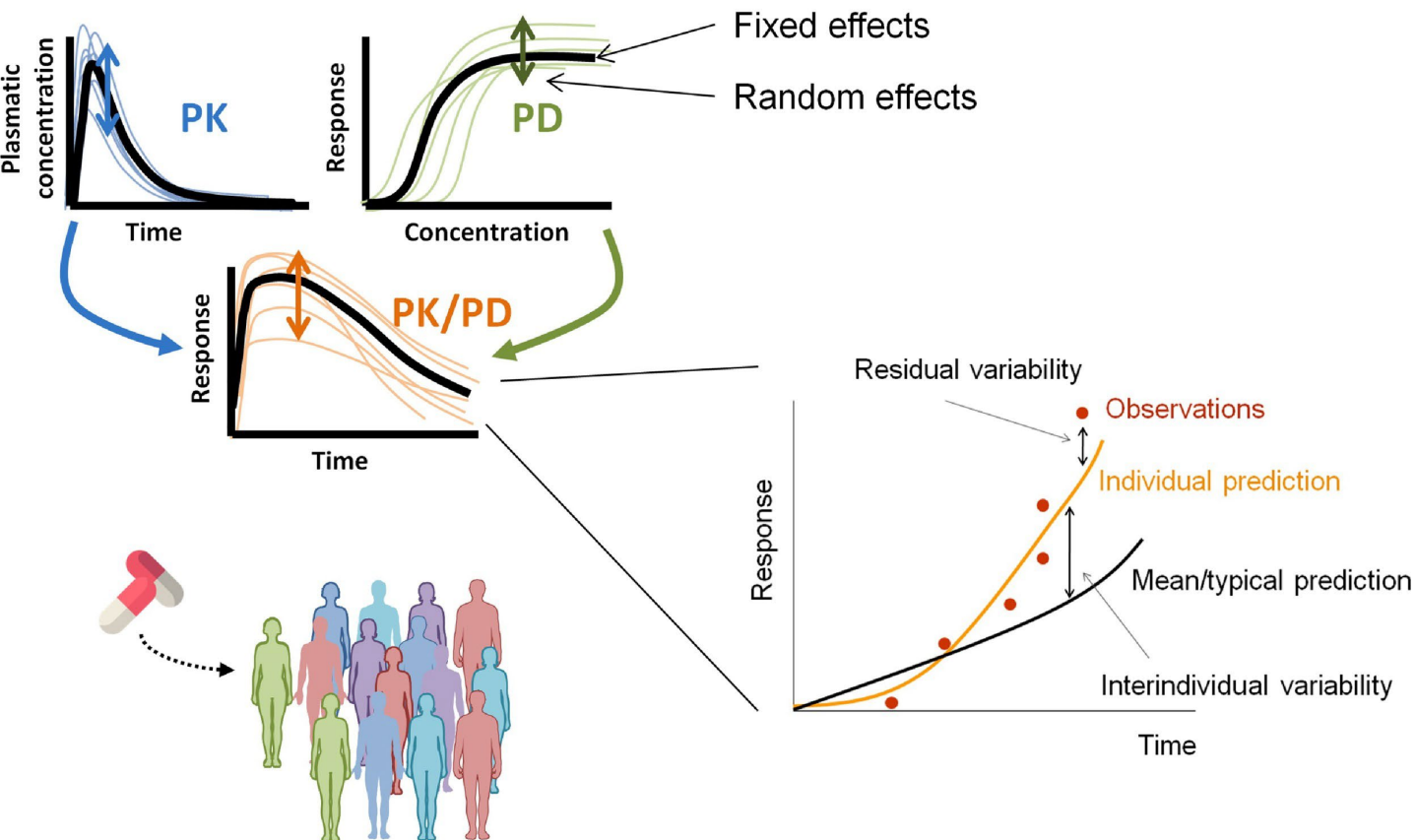
Figure 5: EpiSciML QSIR model - 9 compartment: Effect of training size on forecasting performance. Figure shows the prediction and forecasting performance of the QSIR model for training data size of (a) 30, (b) 34, (c) 38, (d) 40 data points.

DeepNLME: Integrate neural networks into traditional NLME modeling

DeepNLME is SciML-enhanced modeling for clinical trials

DeepNLME is SciML-enhanced modeling for clinical trials

Mixed-effects modeling



- 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

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



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

Structural Model (pre)

Math: Find (θ, η) such that $E[\eta] = 0$
Requires special fitting procedures (Pumas)



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

Dynamics

Intuition: 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 $E[\eta] = 0$

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

Structural Model (pre)

How can we find these models?

Intuition: 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

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

Dynamics

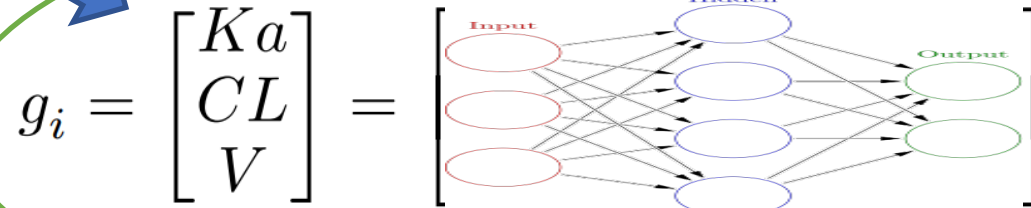
From Dynamics to Nonlinear Mixed Effects (NLME) Modeling

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 $E[\eta] = 0$



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

Structural Model (pre)

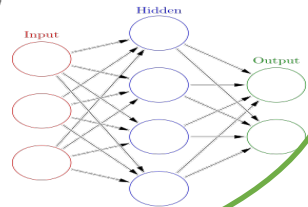
How can we find these models?

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!

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

Dynamics



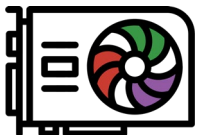
DeepNLME in Practice: Data Mining for Predictive Covariates

```
model = @model begin
  @param begin
    θ ∈ VectorDomain(lower=[0.1,0.0008,0.0040,1],upper=[5.0,0.5,0.9,5.0])
    Ω ∈ PSDDomain(3)
    σ²_add ∈ RealDomain(lower=0.001,init=sqrt(0.388))
    p1 ∈ NeuralDomain(FastChain(FastDense(2,50,tanh),FastDense(50,1),(x,p)->x.^2))
    p2 ∈ NeuralDomain(FastChain(FastDense(2,50,tanh),FastDense(50,1),(x,p)->x.^2))
  end

  @random begin η ~ MvNormal(Ω) end

  @pre begin
    Ka = SEX == 0 ? θ[1] + η[1] : θ[4] + η[1]
    K = nn1([θ[2],η[2]],p1)[1]
    CL = nn2([θ[3]*WT,η[3]],p2)[1]
    Vc = CL/K
    SC = CL/K/WT
  end

  @covariates SEX WT
  @vars begin conc = Central / SC end
  @dynamics Depots1Central1
  @derived begin dv ~ @. Normal(conc, sqrt(σ²_add)) end
end
```



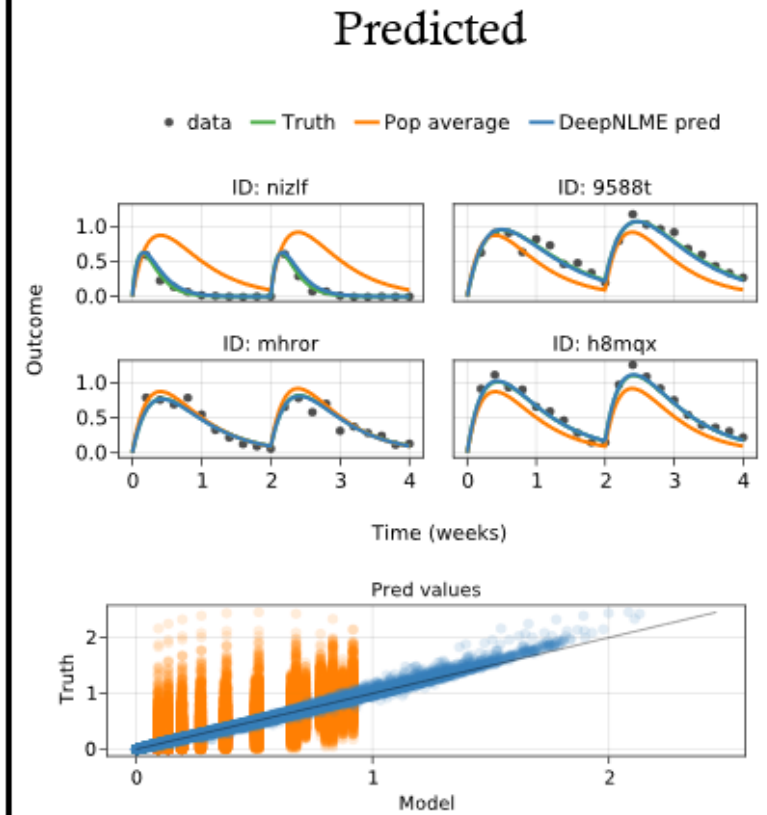
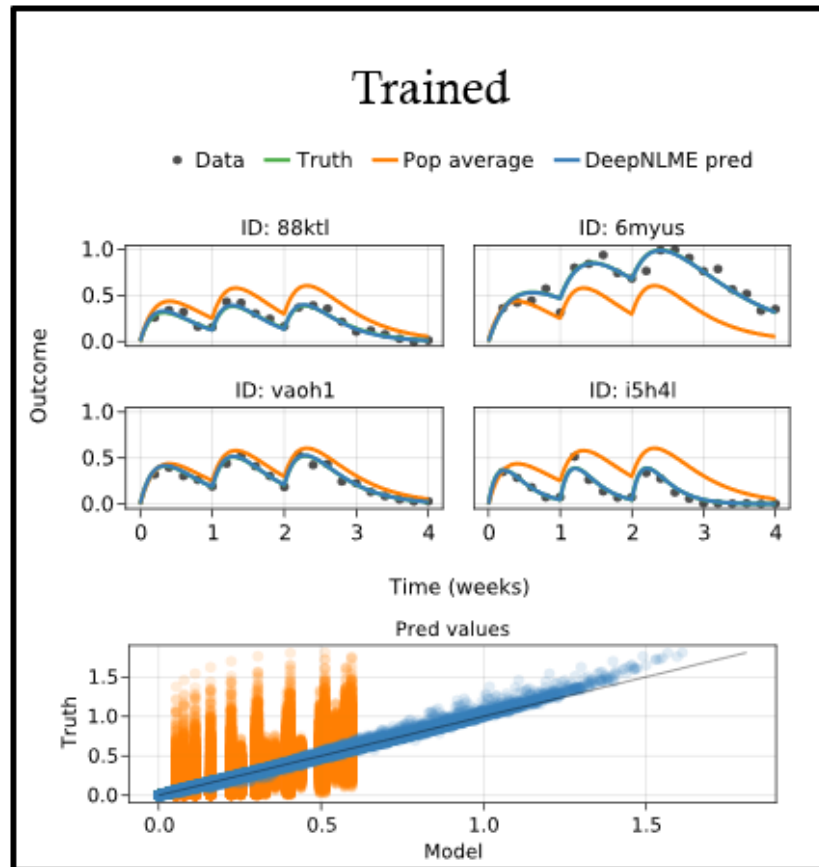
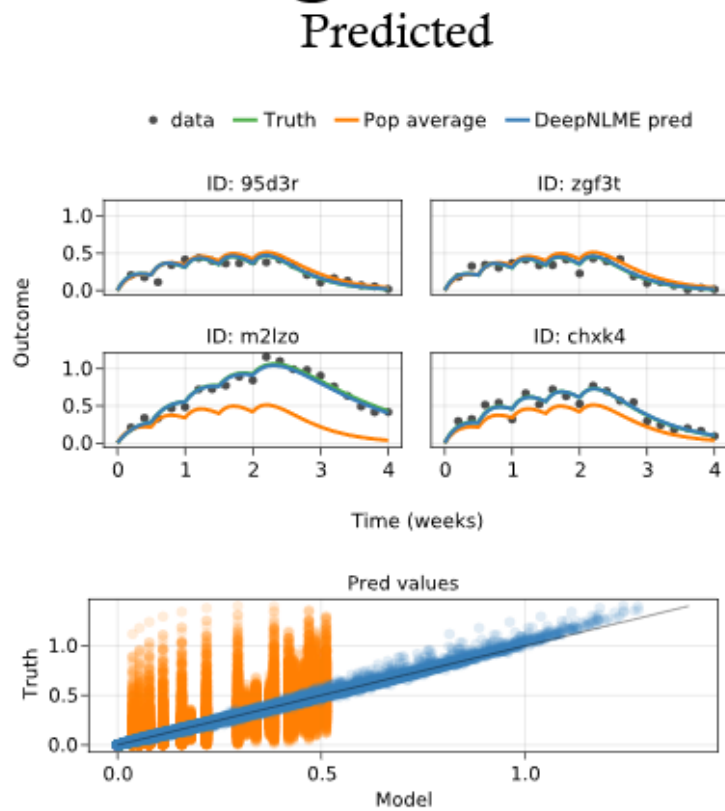
Utilize GPU acceleration for neural networks

Automate the discovery of covariate models

- Train convolutional neural networks to incorporate images as covariates
- Train transformer models to utilize natural language processing on electronic health records
- Utilize automated model discovery to prune genomics data to find the predictive subset

Currently being tested on clinical trial data

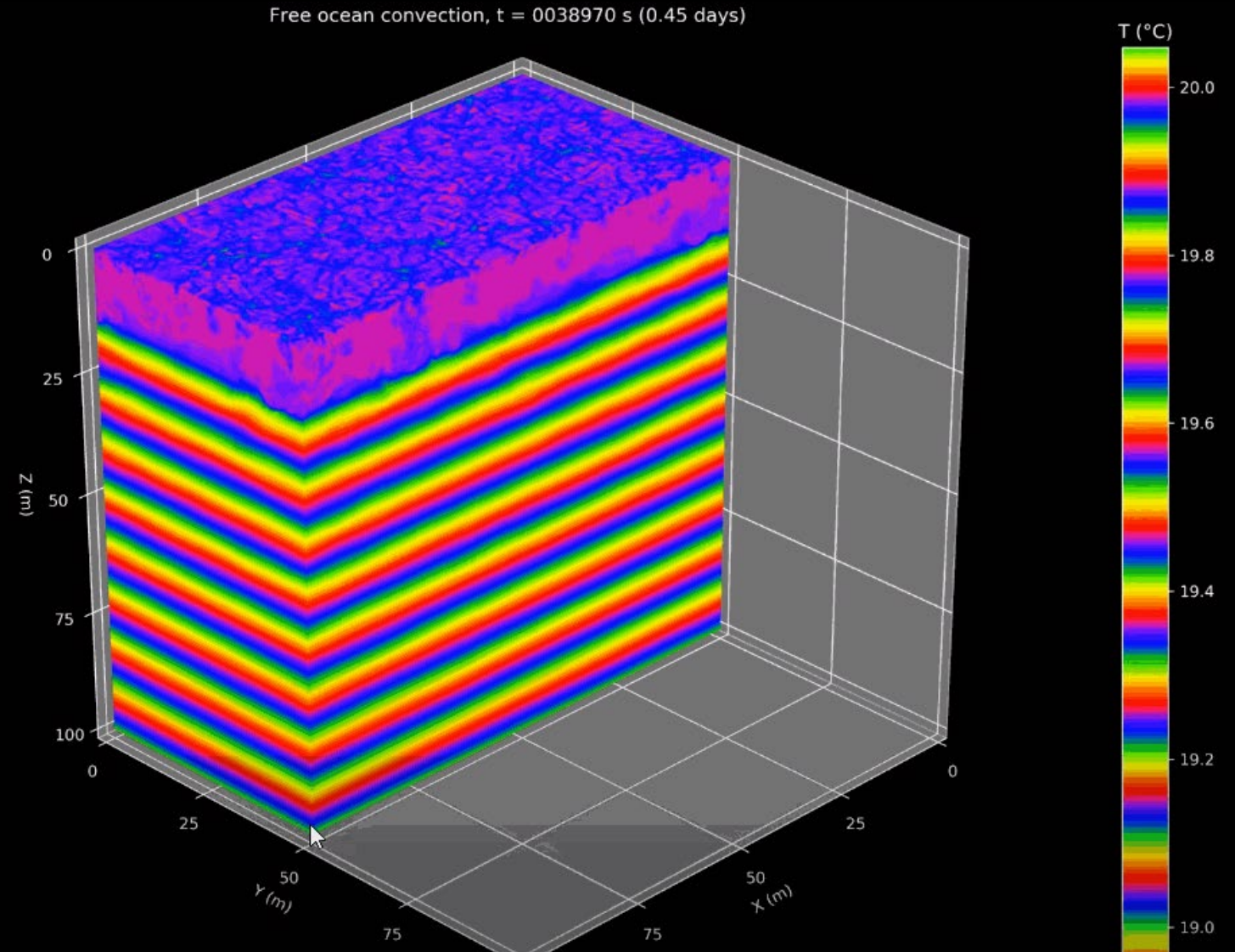
DeepNLME: Automated Construction of Patient-Specific Pharmacological Models for Individualized Dosing



High fidelity surrogates of ocean columns for climate models

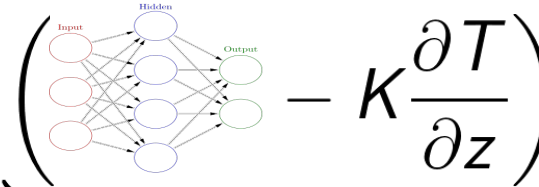
3D simulations are high resolution but too expensive.

Can we learn faster models?



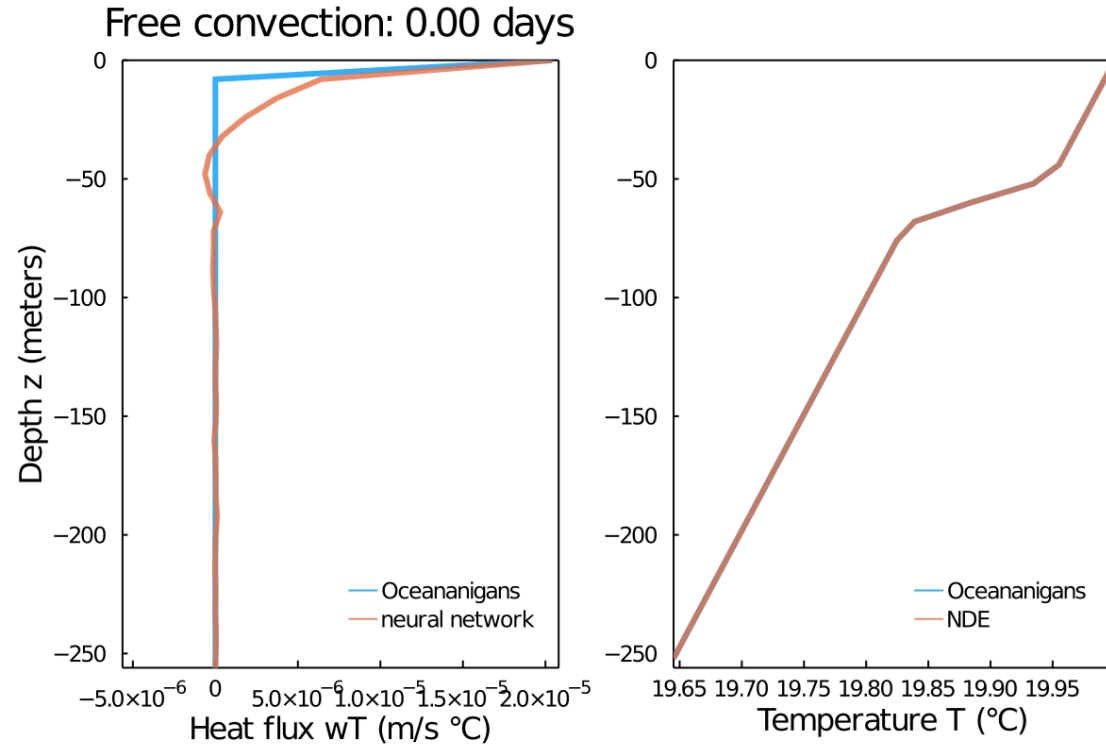
Neural Networks Infused into Known Partial Differential Equations

Derive a 1D approximation to the 3D model

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


Incorporate the “convective adjustment”

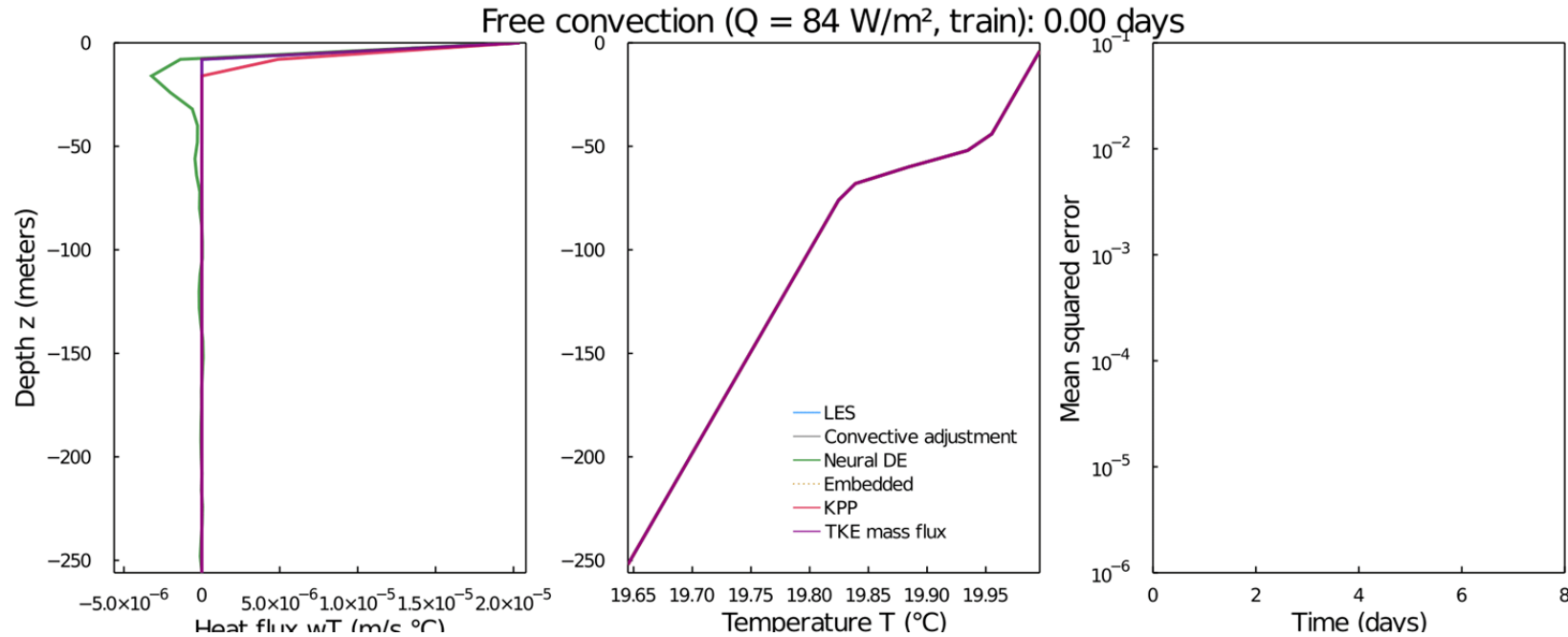
$$K = \begin{cases} 0 & \text{if } \partial_z T > 0 \\ 100 \text{ m}^2/\text{s} & \text{if } \partial_z T < 0 \end{cases}$$



$$\text{loss}(T, wT) = |NN(T) - wT|^2$$

Only okay, but why?

Good Engineering Principles: Integral Control!



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

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

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

How do we do this effectively?

SciML is a software problem.

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

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

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

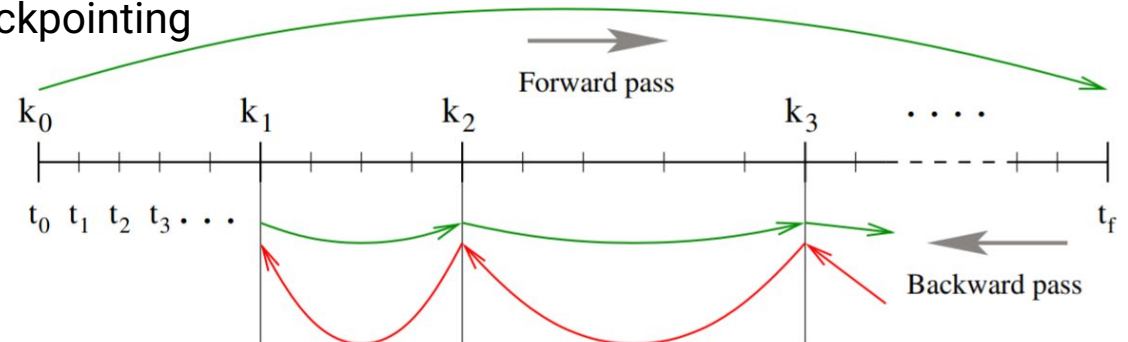
2. Solve $\lambda'(t) = -\frac{df^*}{du(t)} \lambda(t) - \left(\frac{dg}{du(t)}\right)^*$

$\lambda(T) = 0$

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

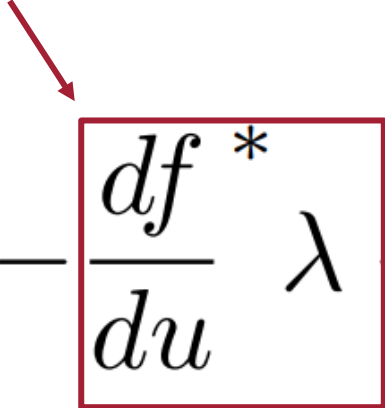
1. $u' = f(z, t)$ forwards, then
 $u' = -f(z, -t)$ backwards!
2. Store $u(t)$ while solving forwards (dense output)

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!

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


$$\lambda(T) = 0,$$

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?

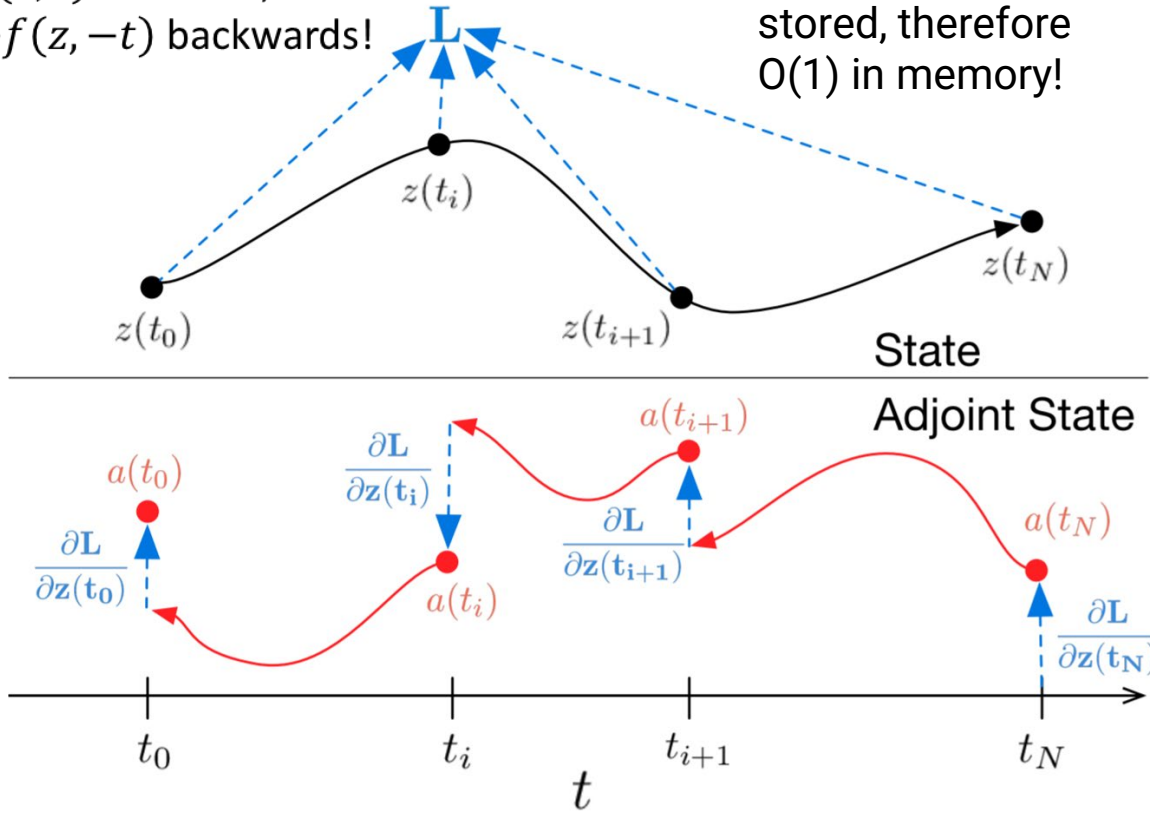
Some methods are “mathematically correct”, but “numerically incorrect”

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!

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

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:

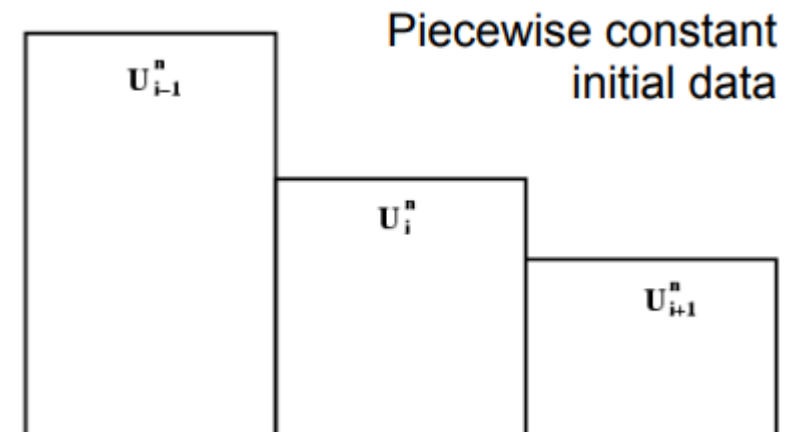
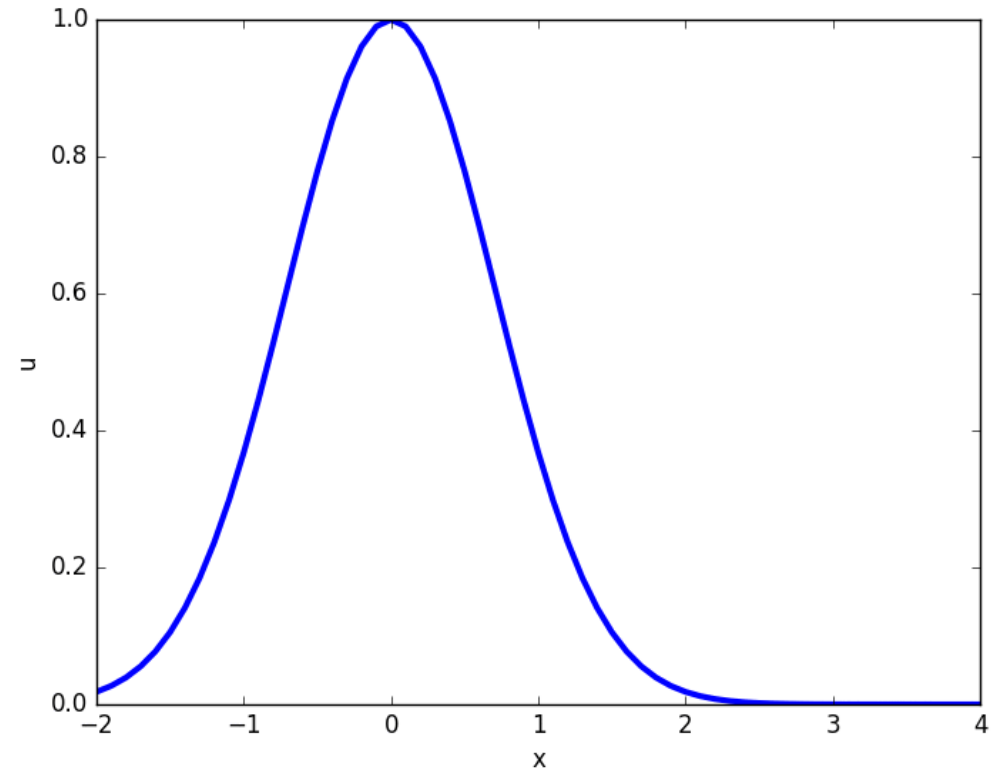
$$\frac{\partial u}{\partial t} + a \frac{\partial u}{\partial x} = 0$$

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

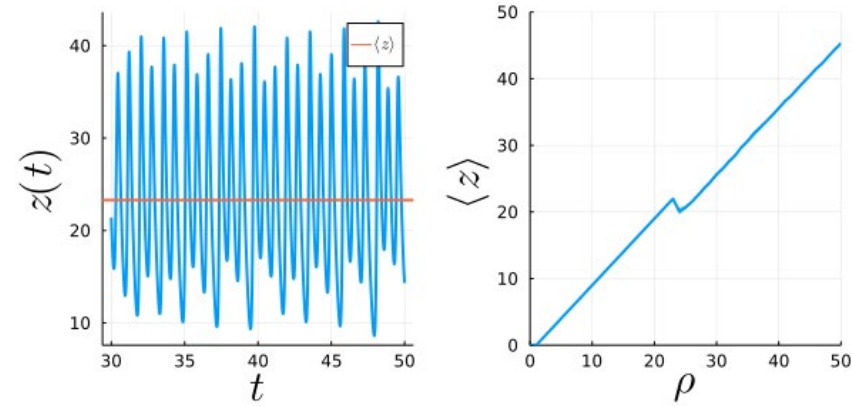
$$u'_i = -\frac{a(u_i - u_{i-1})}{\Delta x} \text{ or } u'_i = -\frac{a(u_{i+1} - u_i)}{\Delta x}?$$

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.

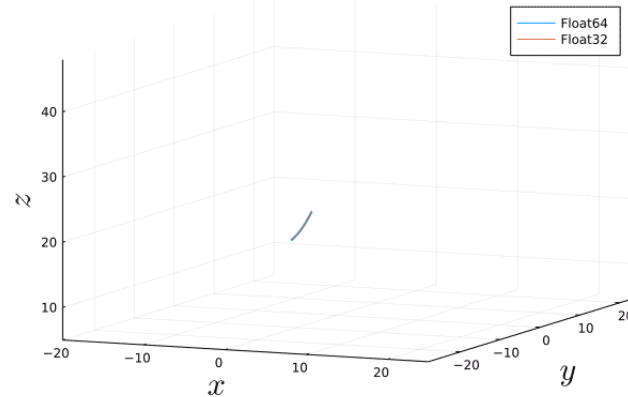


Differentiation of Chaotic Systems: Shadow Adjoints



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!

- Shadowing methods in DiffEqSensitivity.jl

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

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

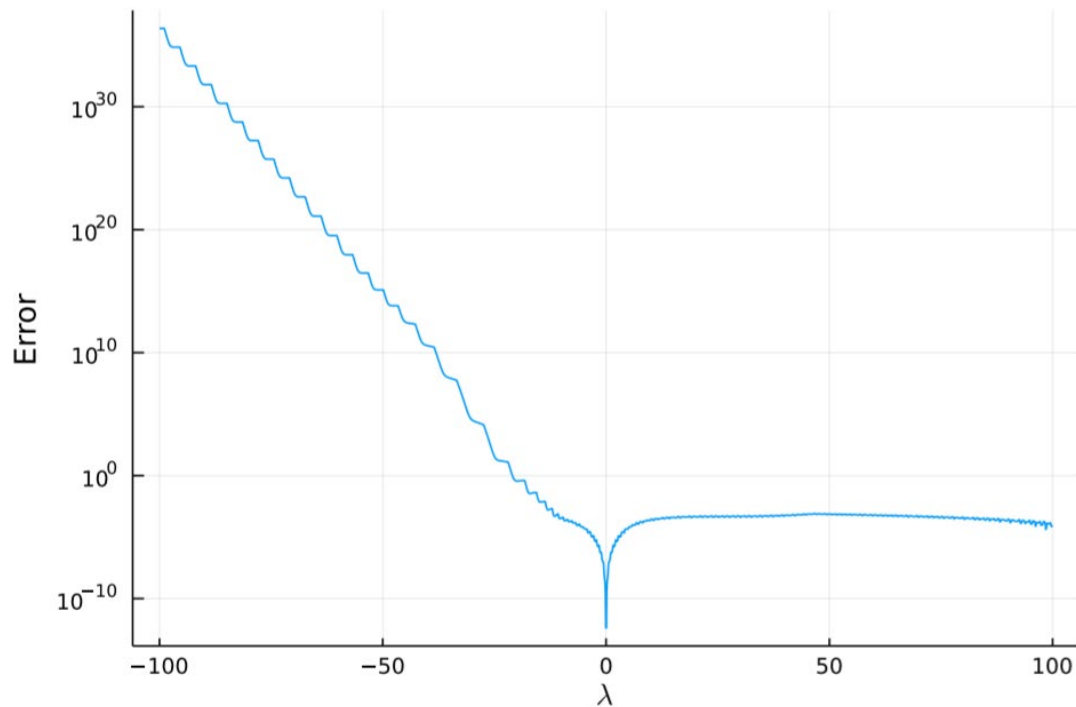
$$\left. \frac{d\langle z \rangle_{\infty}}{d\rho} \right|_{\rho=28} \approx 472 \text{ (Calculus)}$$

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

Problems With Naïve Adjoint Approaches On Stiff Equations

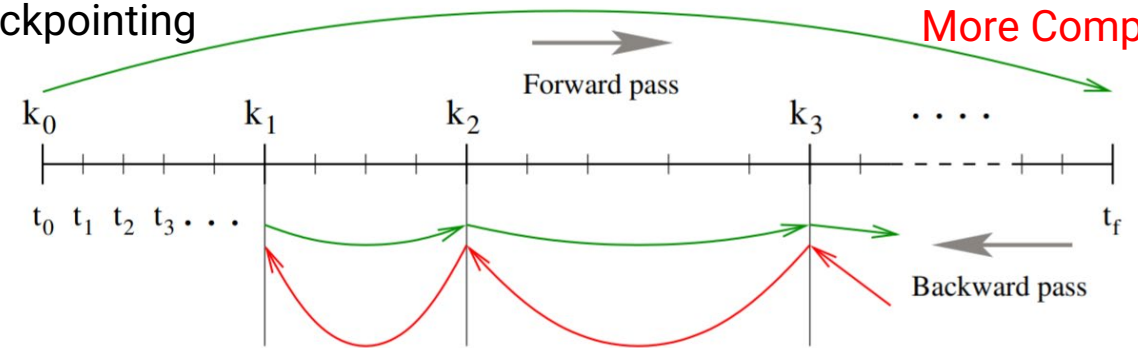
Error grows exponentially...

$u'(t) = \lambda u(t)$, 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**



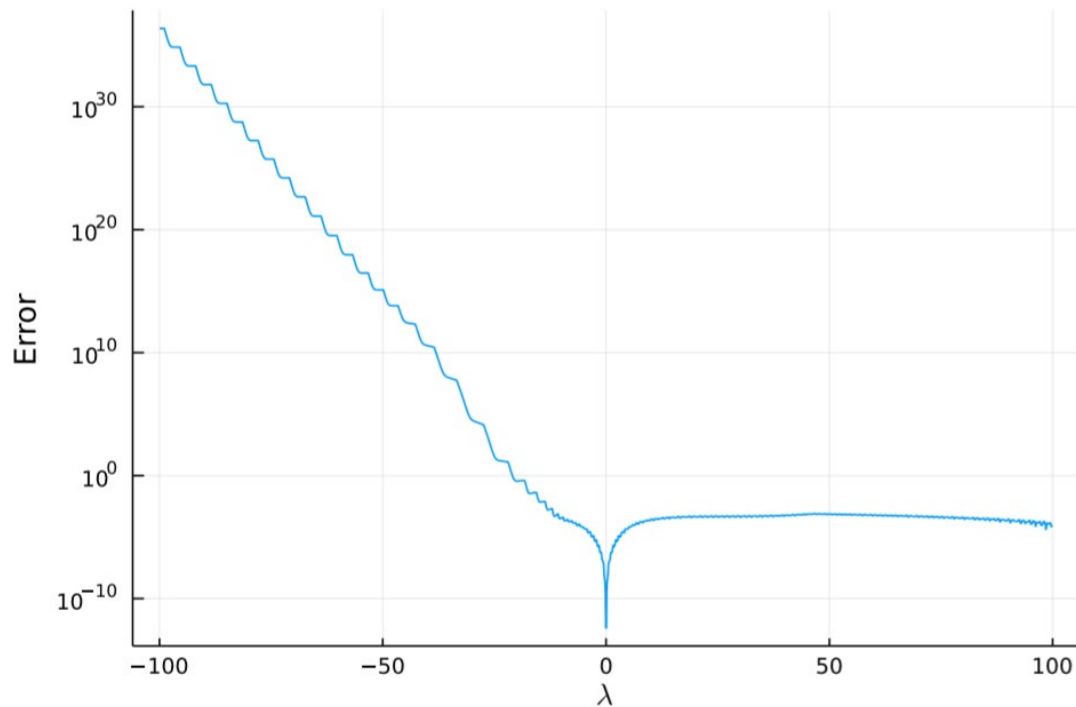
Each choice has an engineering trade-off!

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:

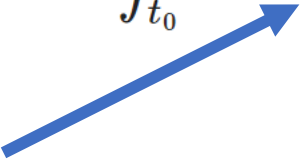
$$2states + parameters$$

Linear solves inside of stiff ODE solvers, ~cubic

Thus, adjoint cost:

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

Problems With Naïve Adjoint Approaches On Stiff Equations

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


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:

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

How do you calculate the integral?

High memory

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

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!



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.

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

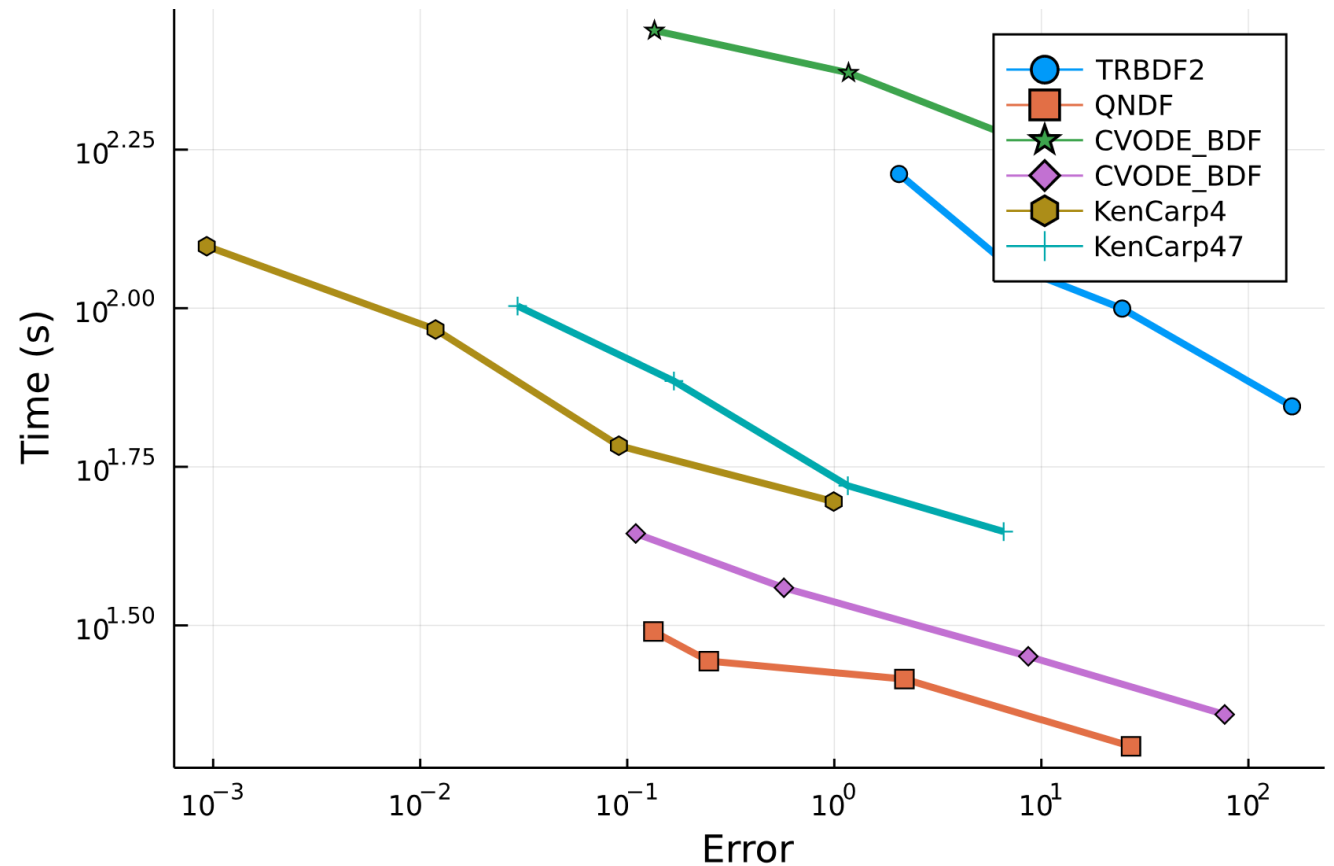
This excludes the extra 2x from symbolics and 2x from sparse parallel compilation!

<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-numeric 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



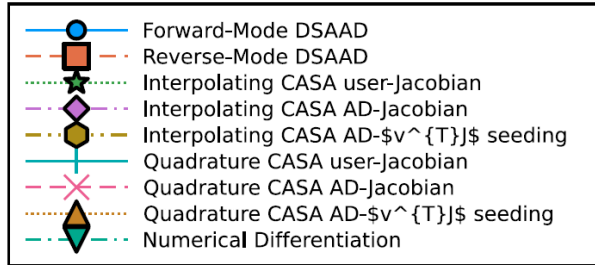
DiffEqSensitivity.jl: Every adjoint is optimized for a different case

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.

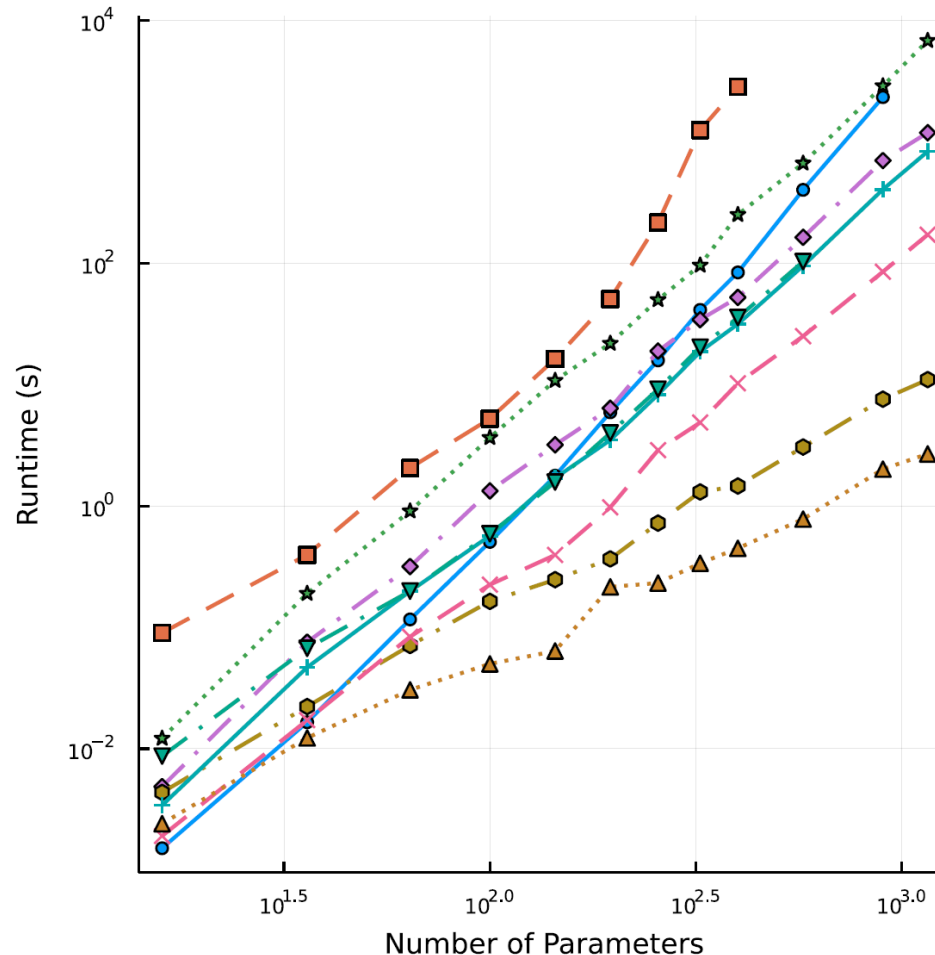
How the adjoint is calculated also matters!

Gradient calculations on a stiff PDE, varying dt

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:

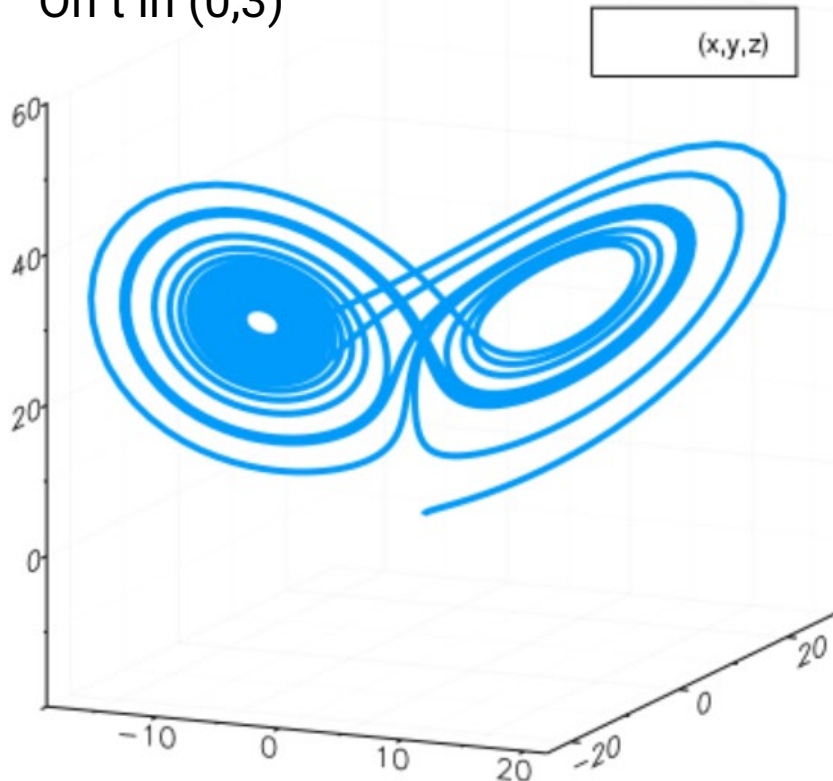
- torchsde: 1.87 seconds
- DifferentialEquations.jl: 0.00115 seconds

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

**And what about other methods for
Scientific 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)
```

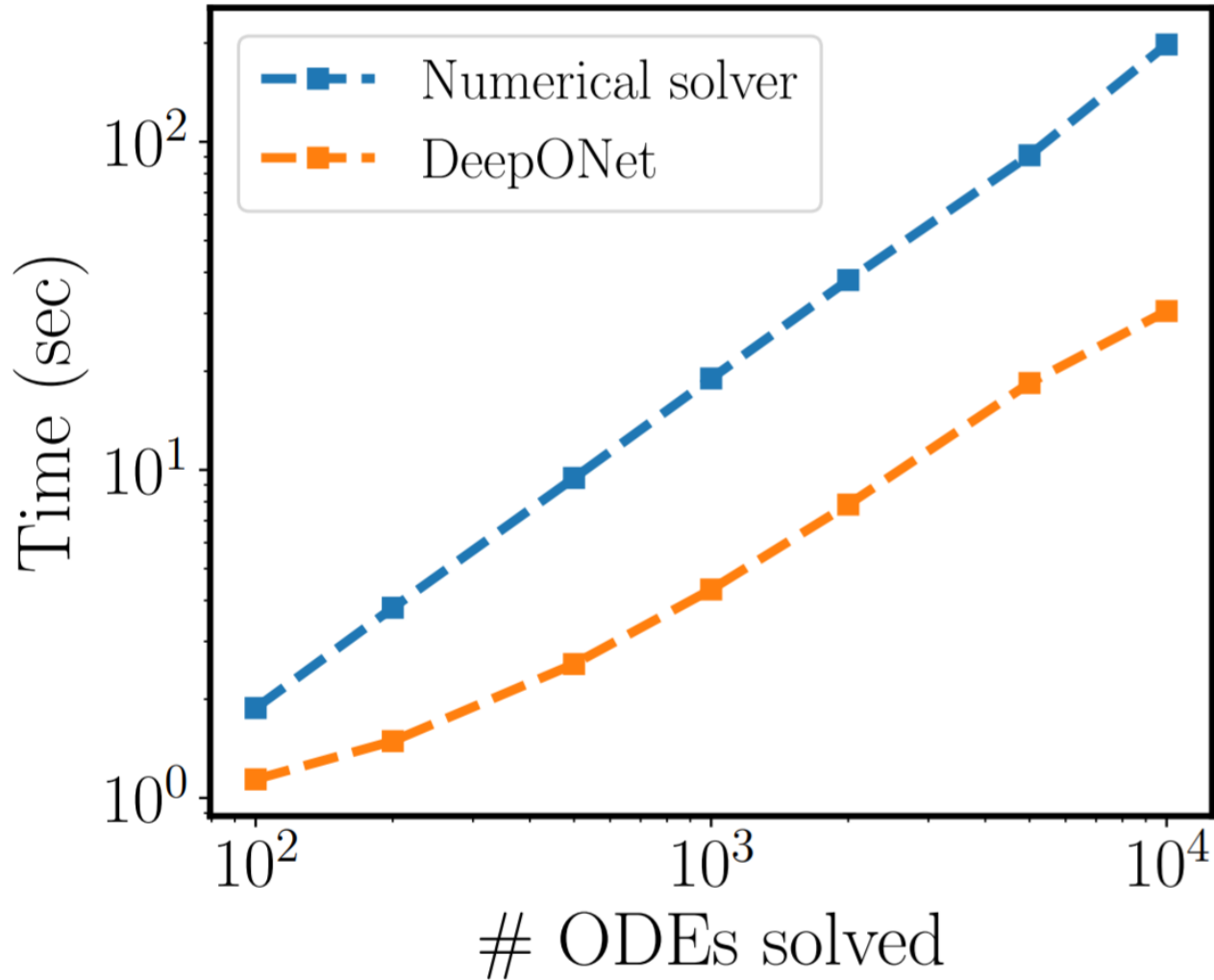
Note on Neural Networks “Outperforming” Classical Solvers

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

Sifan Wang, Paris Perdikaris

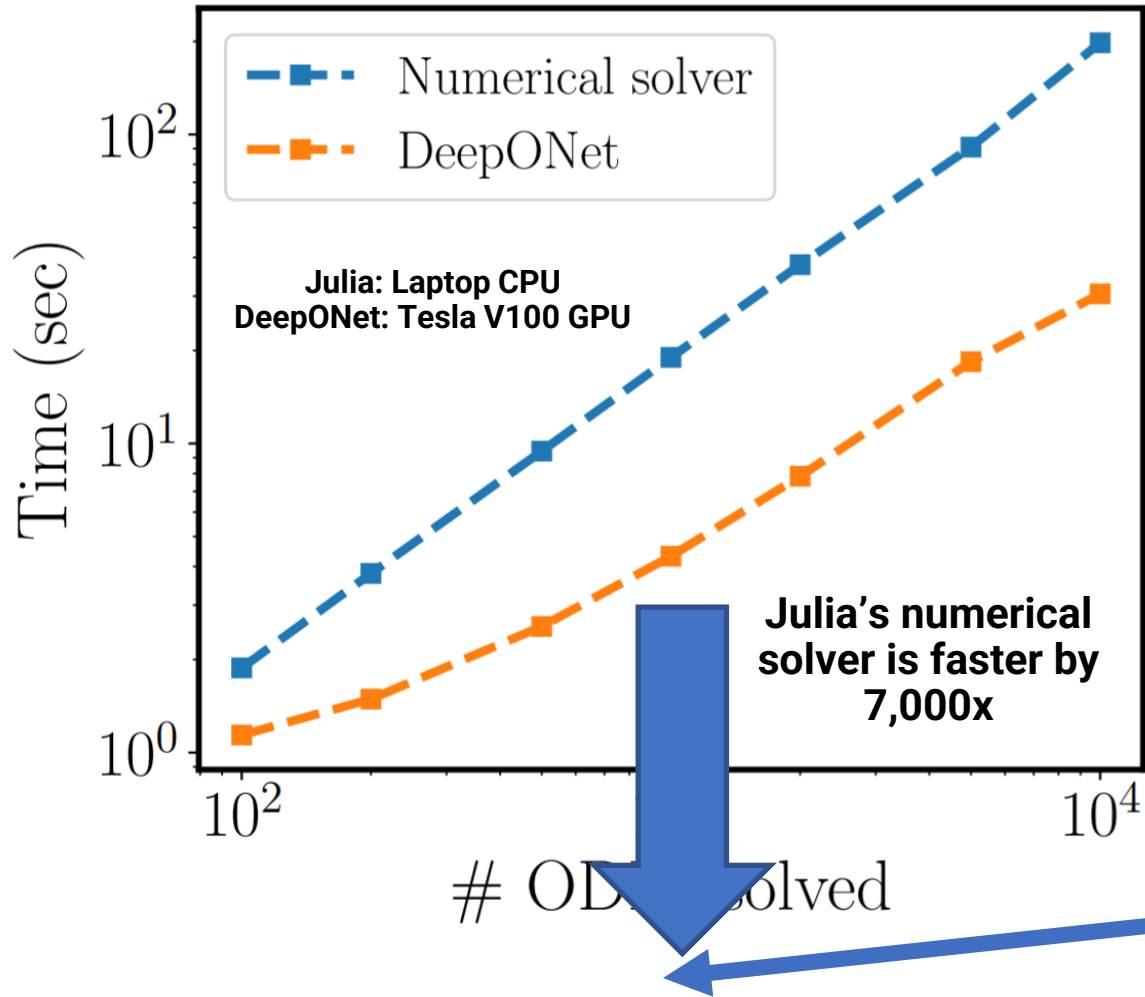
Ordinary and partial differential equations (ODEs/PDEs) play a paramount role in analyzing and simulating complex dynamic processes across all corners of science and engineering. In recent years machine learning tools are aspiring to introduce new effective ways of simulating PDEs, however existing approaches 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?



```
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*y22-k3*y2*y3
```

```
        D(y3) ~ k2*y22]
```

```
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)
```

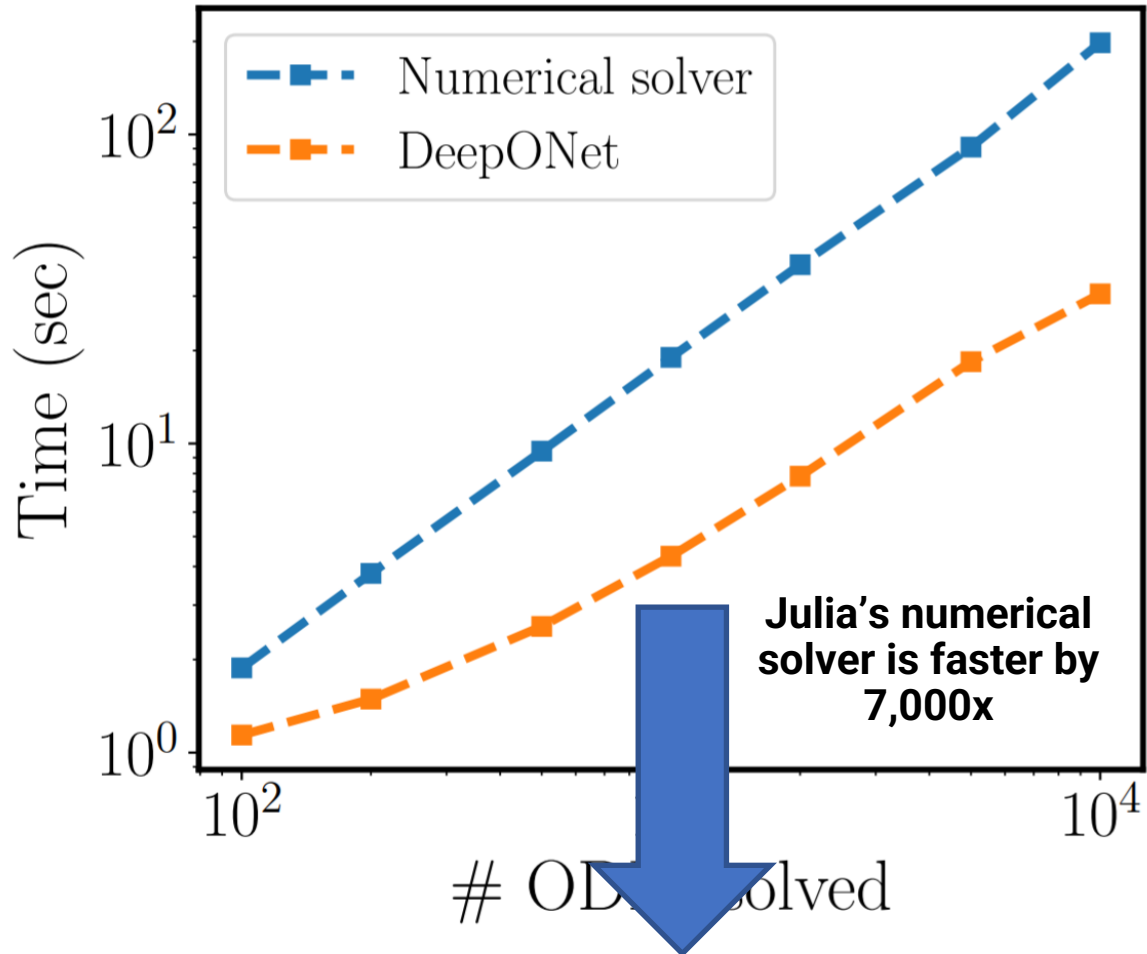
```
@time solve(monteprob, Rodas5(), EnsembleThreads(), trajectories=1000)
```

```
#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?

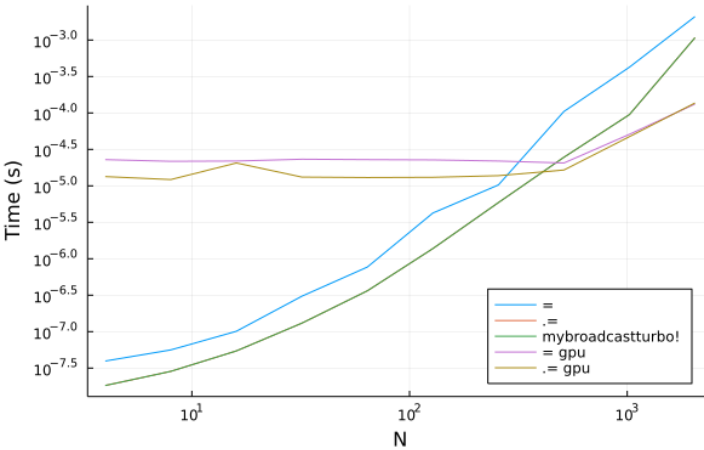


Similar story on Fourier Neural Operator results!

How come so far off?

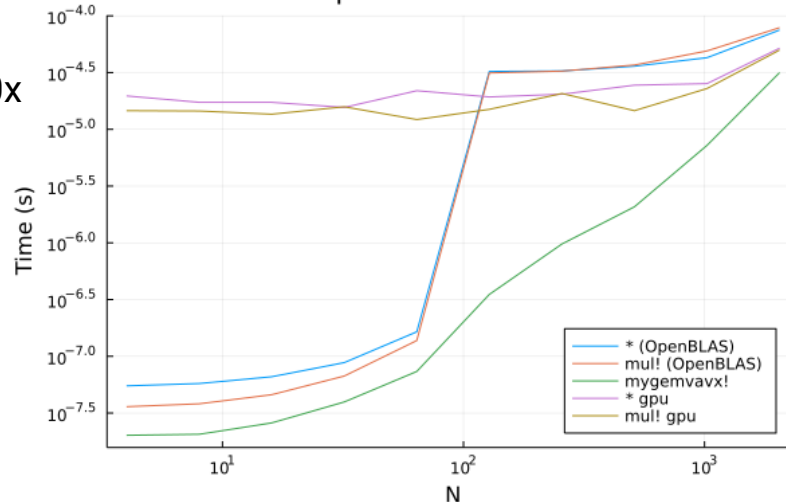
Code Optimization in Machine Learning vs Scientific Computing

Which Micro-optimizations matter for BLAS1?



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

Which Micro-optimizations matter for BLAS2?

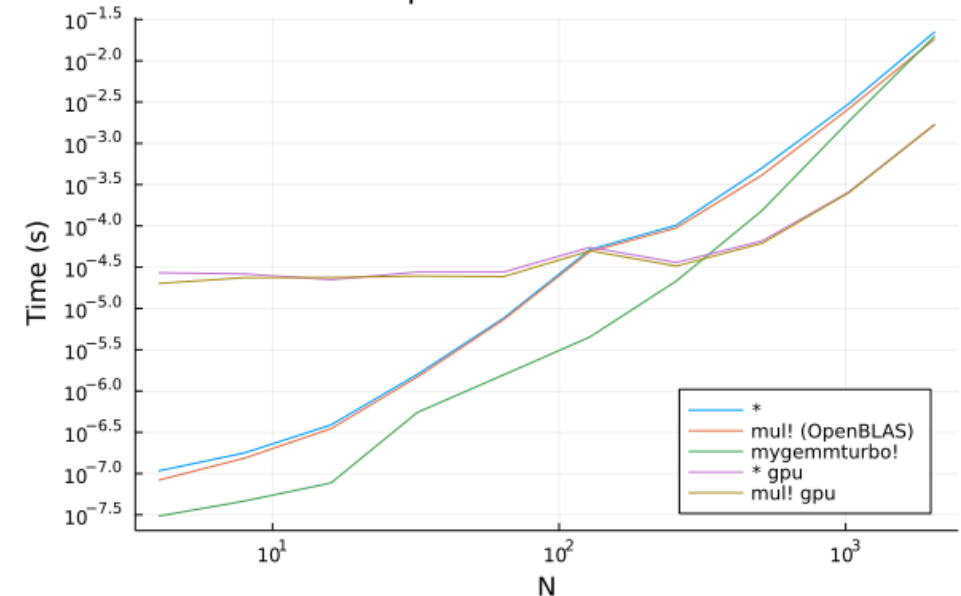


Mutation and
 Memory management: 10x

Manual SIMD: 5x

...

Which Micro-optimizations matter for BLAS3?

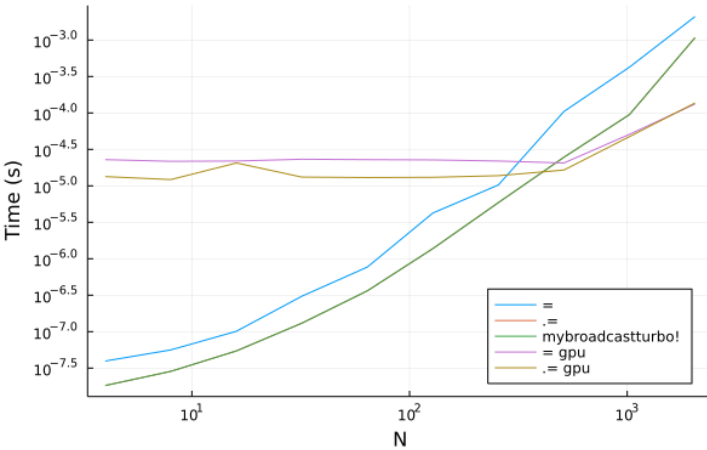


Big $O(n^3)$ operations?
 Just use a GPU
 Don't worry about overhead
 You're fine!

Simplest code is ~3x from optimized

What happens when you specialize computations?

Which Micro-optimizations matter for BLAS1?



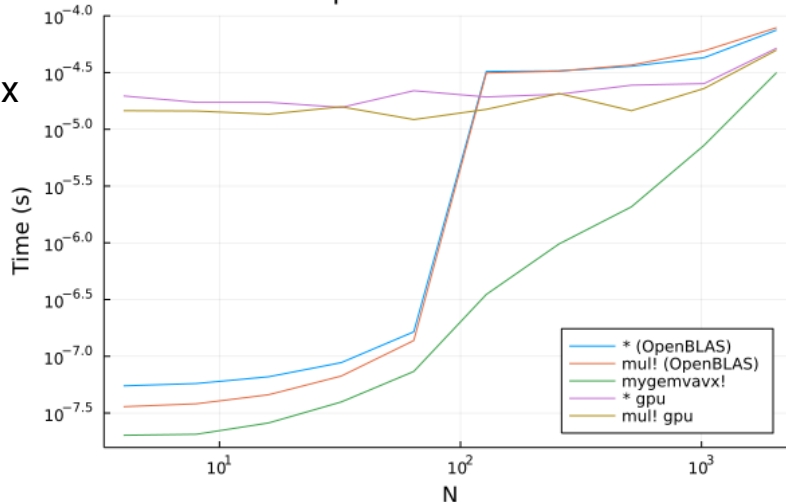
Scientific codes
 $O(n)$ and $O(n^2)$
operations

Mutation and
Memory management: 10x

Manual SIMD: 5x

...

Which Micro-optimizations matter for BLAS2?



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!

What happens when you specialize computations?

Moral of the Story

General computations are generally less optimized

Physics-informed neural networks are an extremely general solver...
QED

Differentiable simulation scales extremely well, if and only if you work on the implementation issues which arise in every equation type.

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!

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

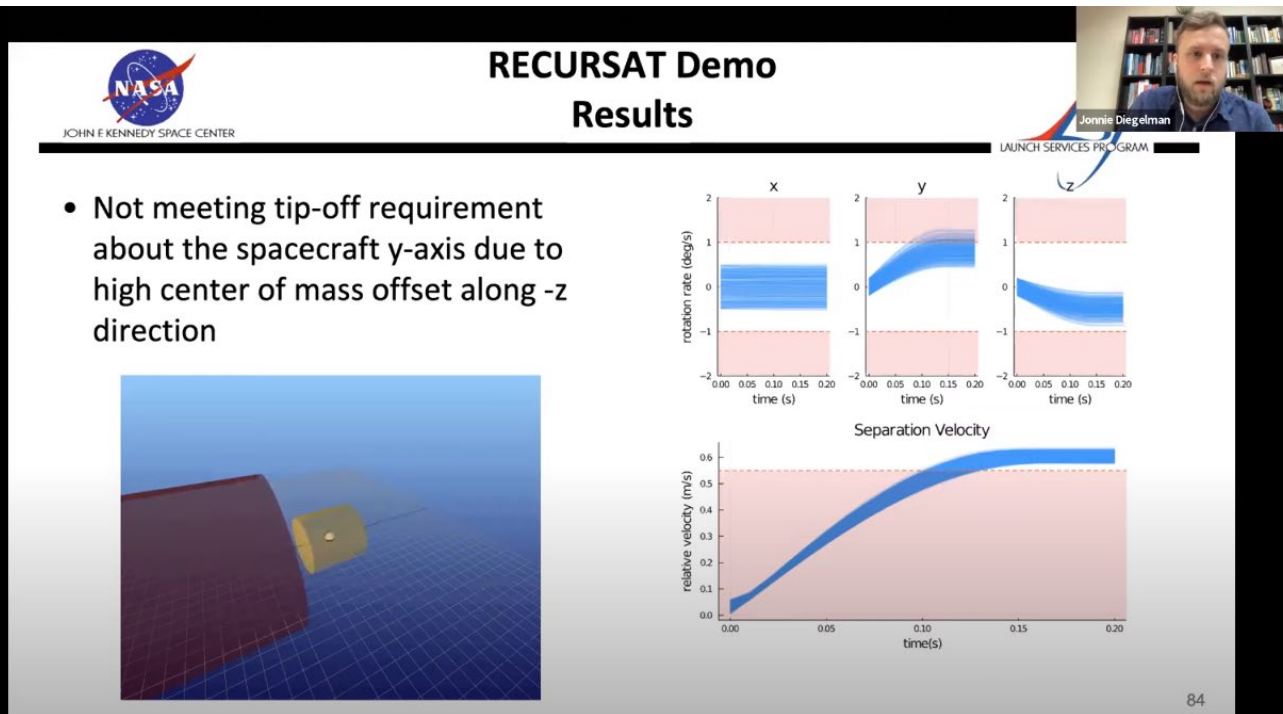
And 100 more libraries to mention...

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**



SciML OSS Org is Impacting Many Modeling and Simulation Applications

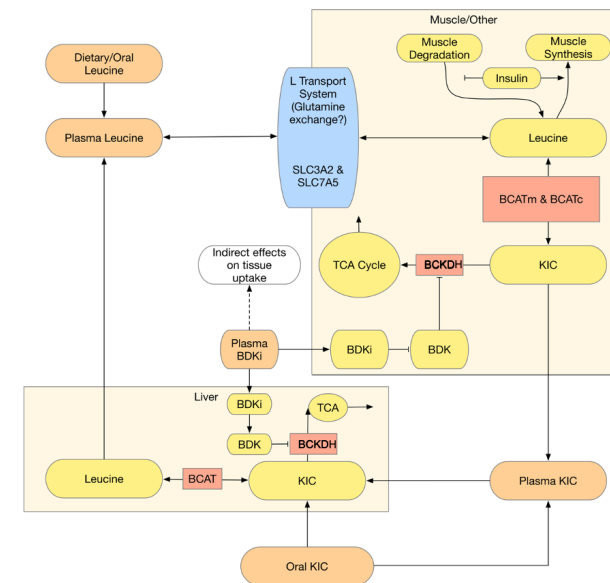


Modeling Spacecraft Separation Dynamics in Julia – SIAM CSE 2021
Jonathan Diegelman, NASA Launch Services Program and A.I. Solutions

15,000x acceleration over Simulink using Julia's ModelingToolkit.jl

175x acceleration for Pfizer's quantitative systems pharmacology team via automated GPU acceleration

2020: American Conference on Pharmacometrics (ACoP) Quality Award

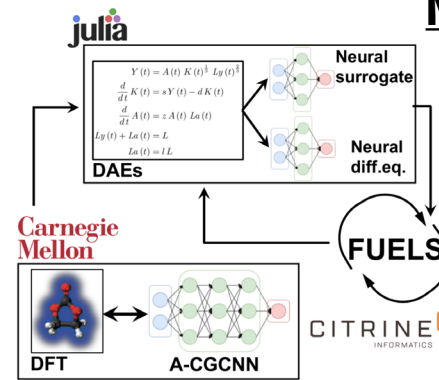


<https://juliacomputing.com/case-studies/pfizer/>

Conclusion

Bridging computational science and machine learning helps improve all aspects of discovery

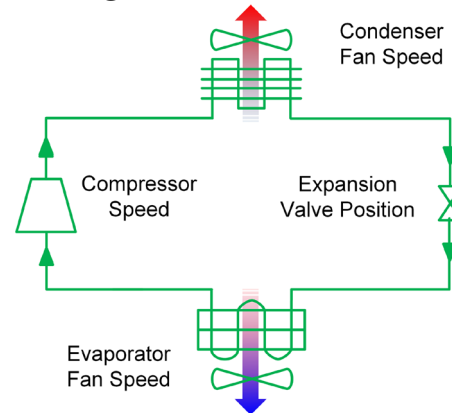
Faster Drug Development



More efficient batteries



Energy Efficient Buildings



Climate modeling for improved agriculture

