

Data-Efficient Model Discovery with Scientific Machine Learning

Chris Rackauckas

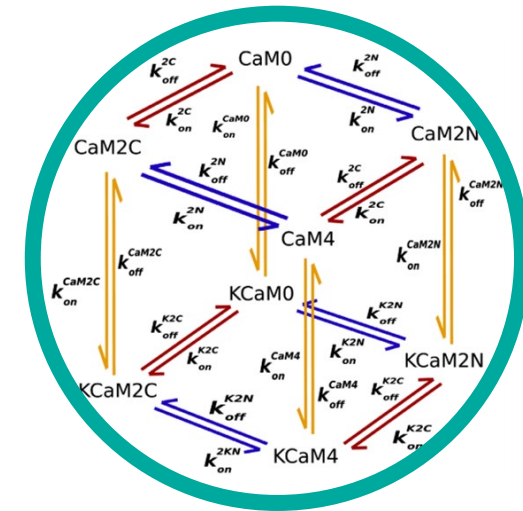
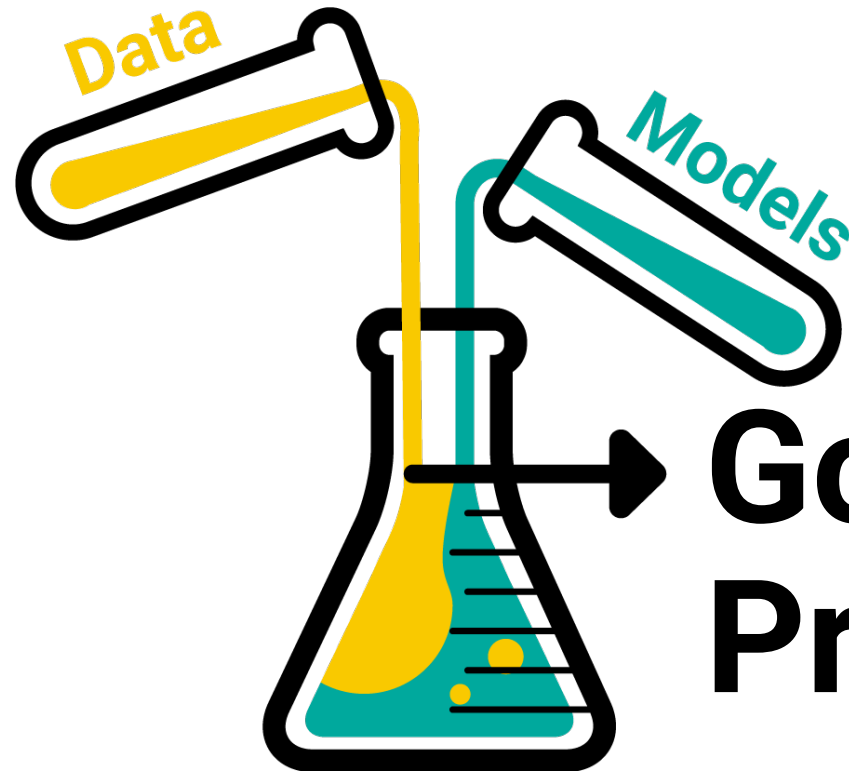
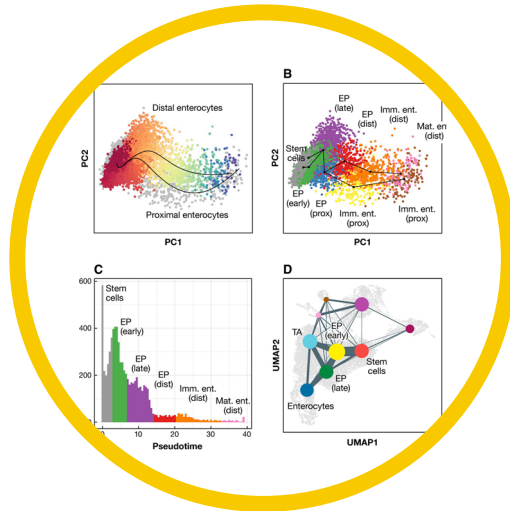
**Director of Modeling and Simulation,
*Julia Computing***

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

**Director of Scientific Research,
*Pumas-AI***

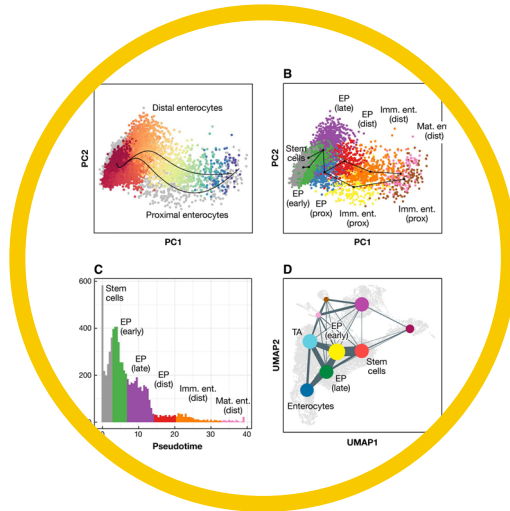
Scientific Machine Learning is model-based data-efficient machine learning

How do we simultaneously use both sources of knowledge?

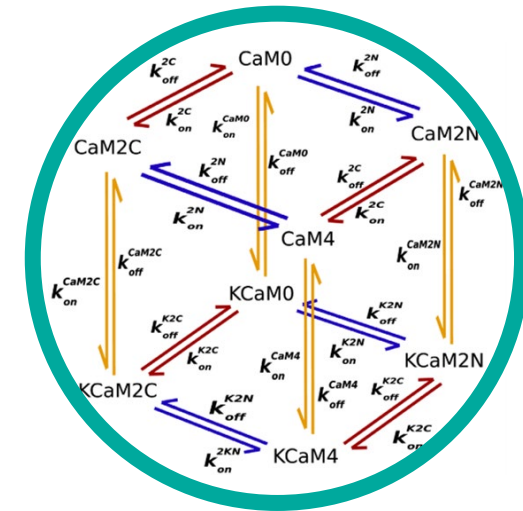
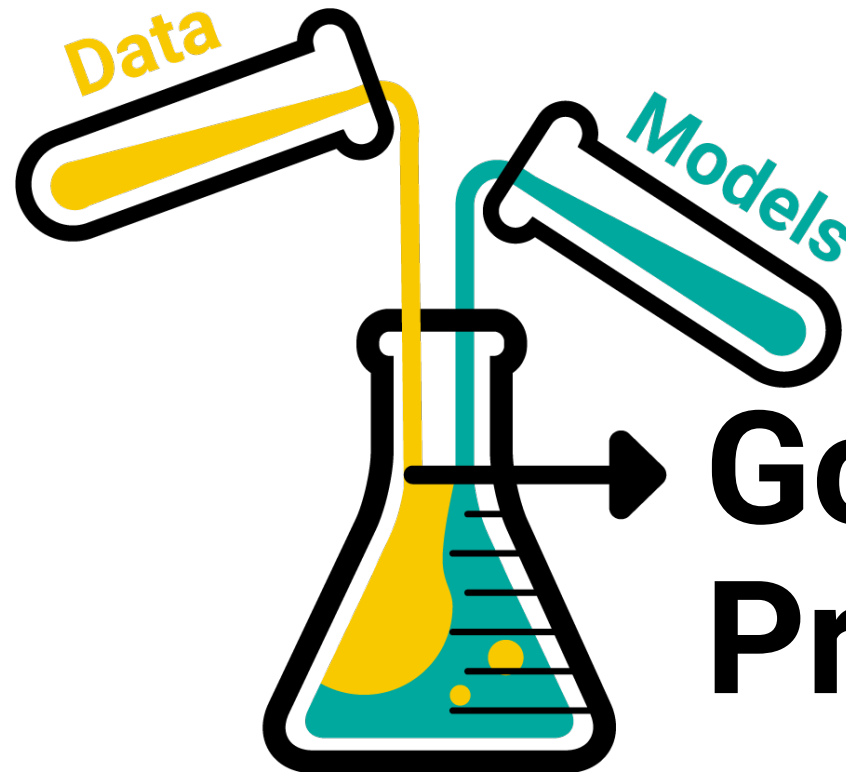


Scientific Machine Learning is model-based data-efficient machine learning

How do we simultaneously use both sources of knowledge?

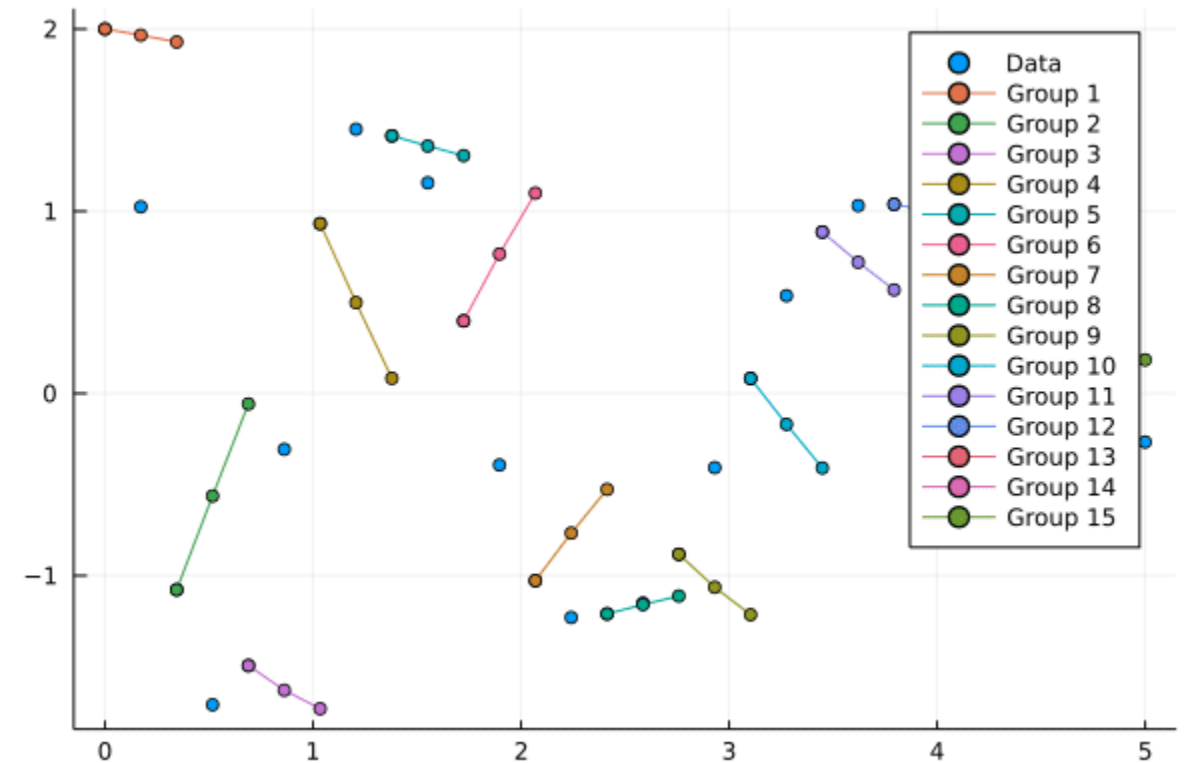
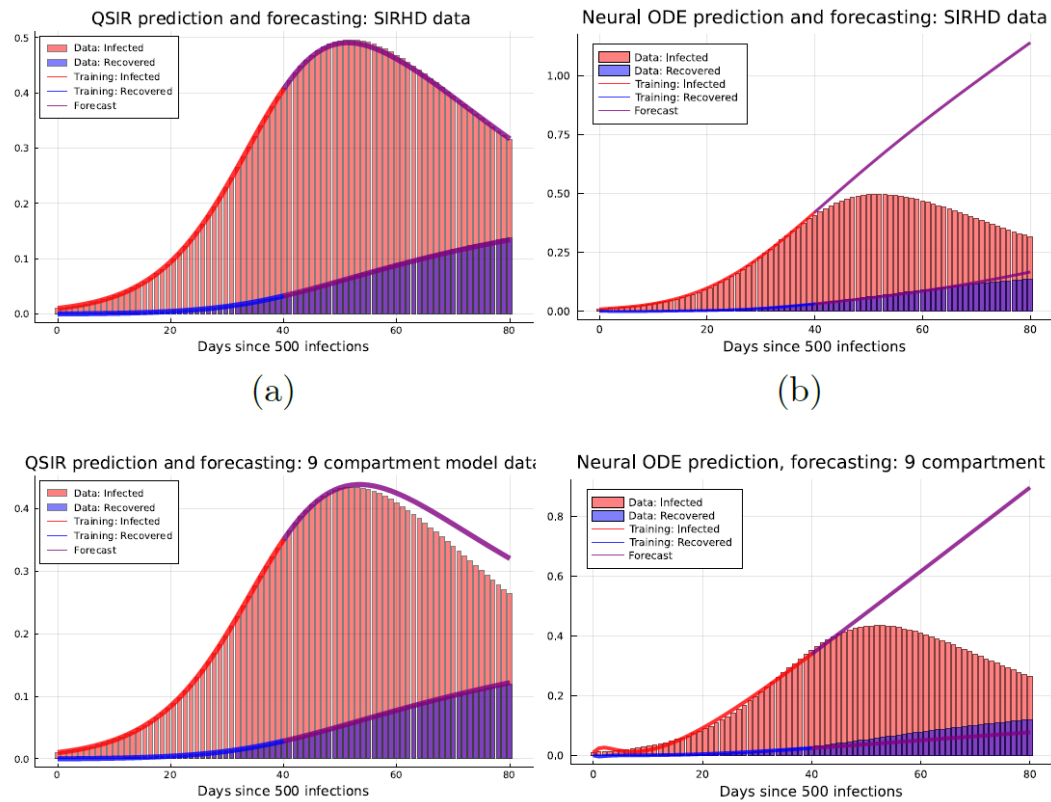


Numerical Analysis?



Outline

Mixing equation discovery into epidemic modeling workflows will revolutionize the field



1. Scientific Machine Learning Applications

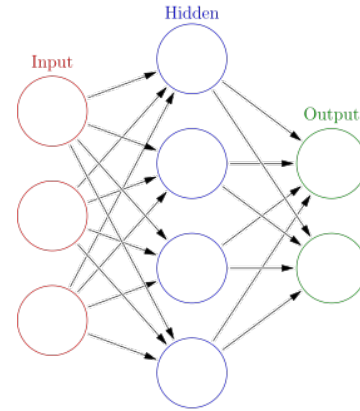
Domain knowledge with machine learning

2. Scientific Machine Learning Software

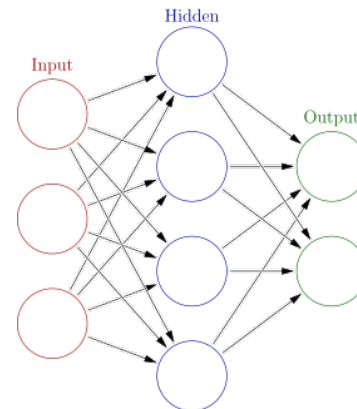
Fast and automated simulation and model discovery

Universal (Approximator) Differential Equations

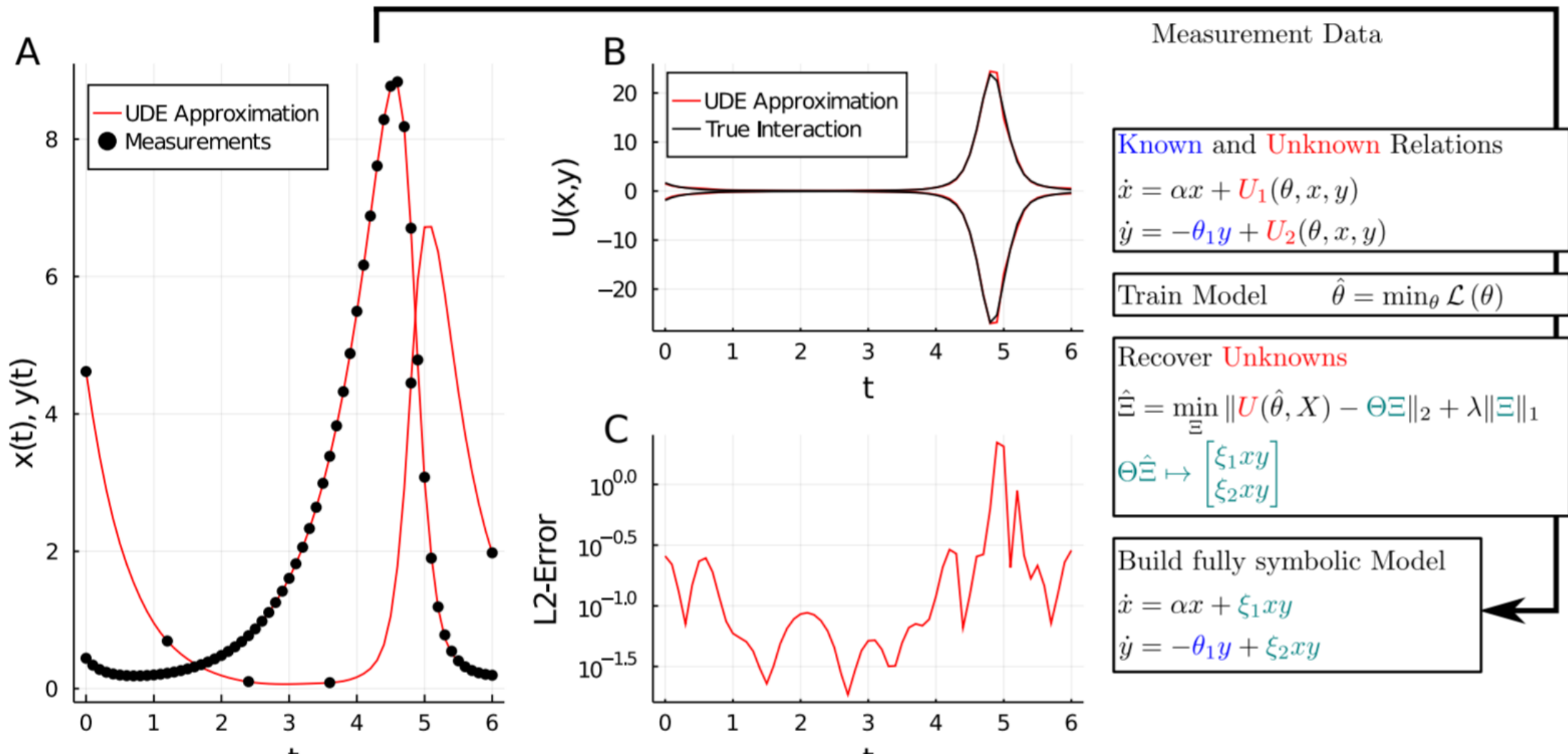
$$u' = f(u, t)$$



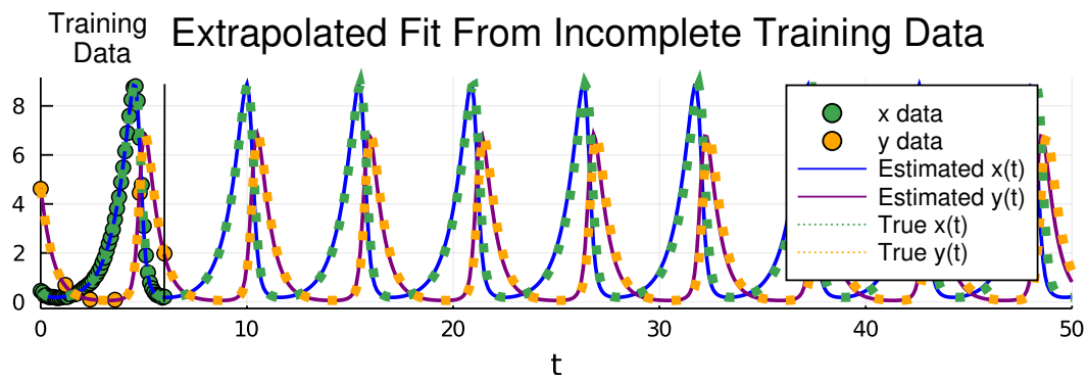
$$\begin{aligned} x' &= \alpha x + \\ y' &= -\beta y + \end{aligned}$$



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!

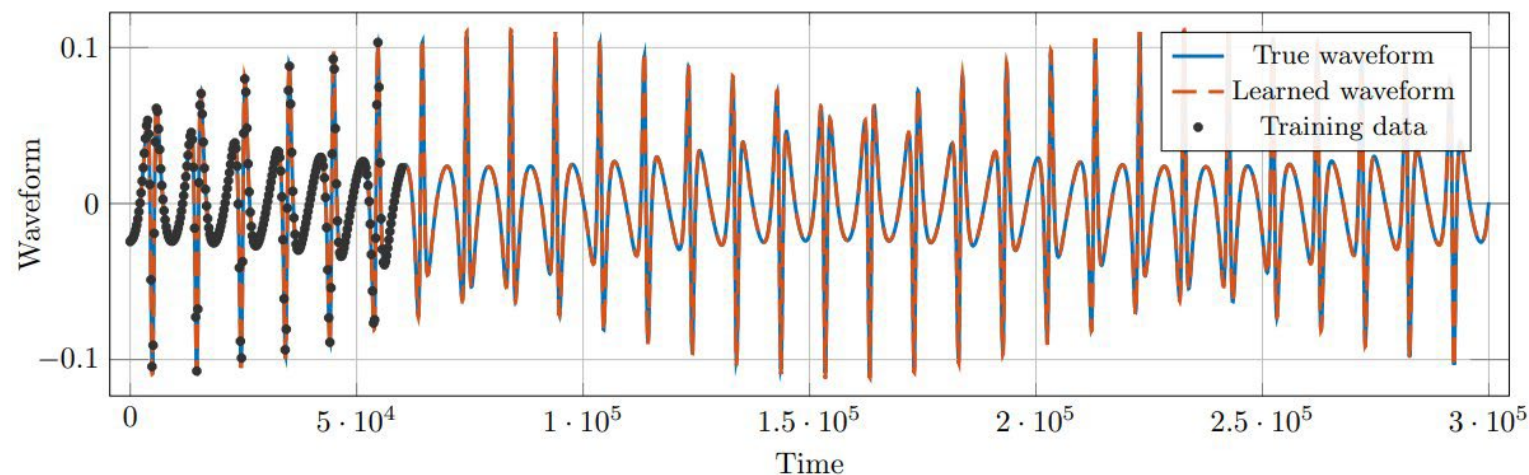
Upon denoting $\mathbf{x} = (\phi, \chi, p, e)$, we propose the following family of UDEs to describe the two-body relativistic dynamics:

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

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

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

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



SciML Shows how to build Earthquake-Safe Buildings

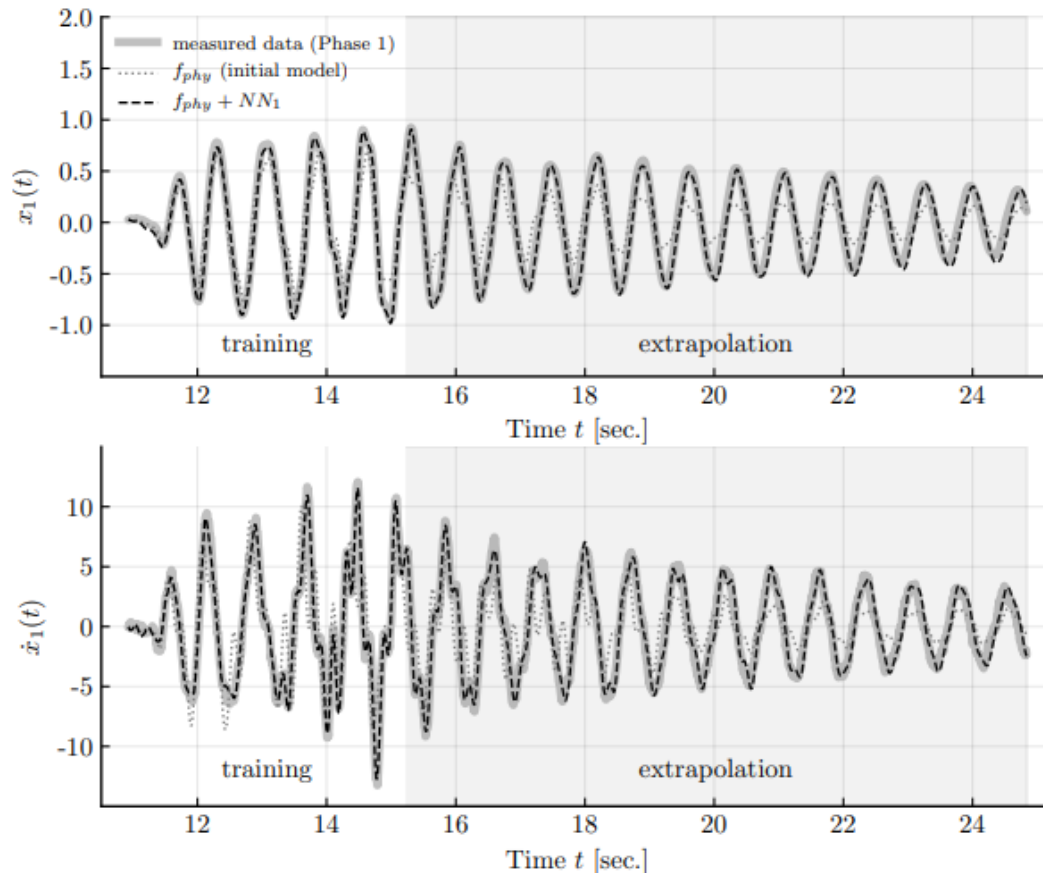


Figure 12: Comparison of time history of the response for displacement $x_1(t)$ and velocity $\dot{x}_1(t)$ for the NSD experiment (Phase 1).

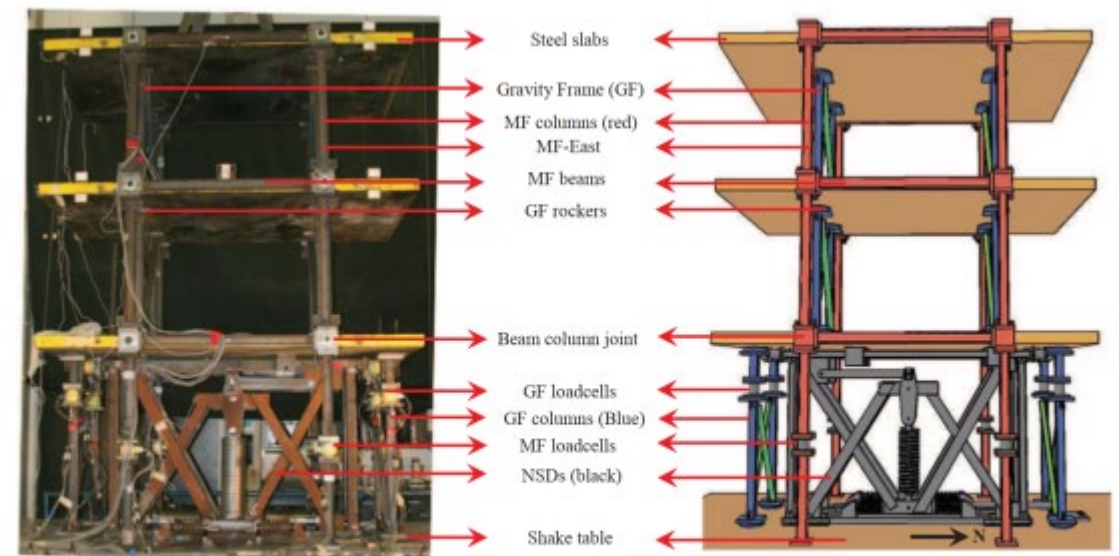


Figure 10: The structural system equipped with a negative stiffness device in between the first floor and the shake table.

Scientific machine learning for earthquake-safe buildings

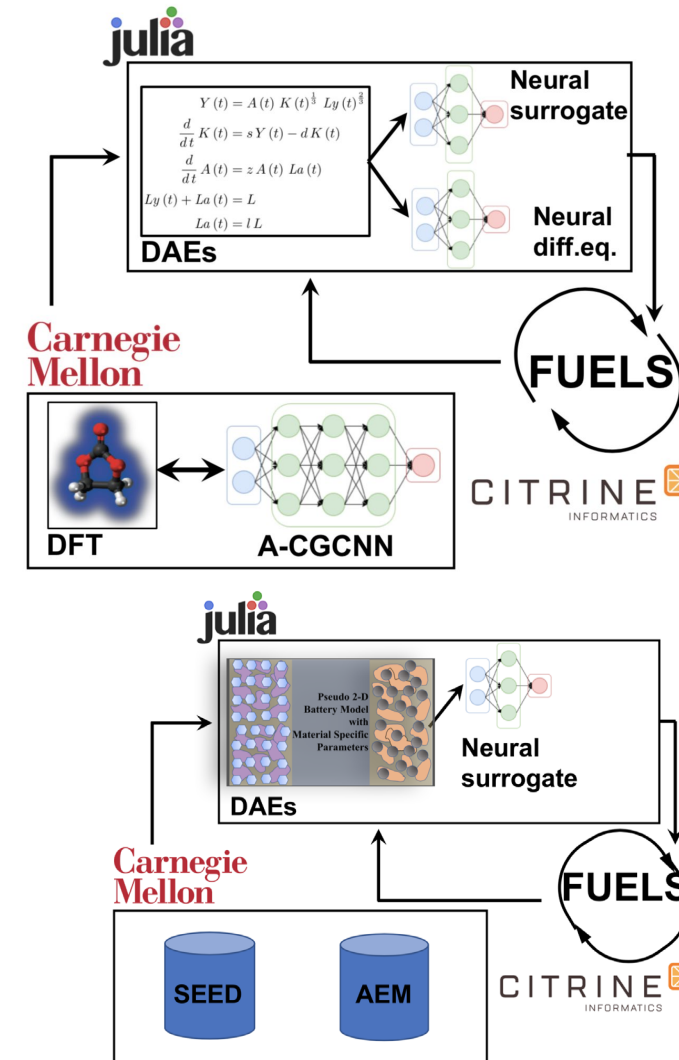
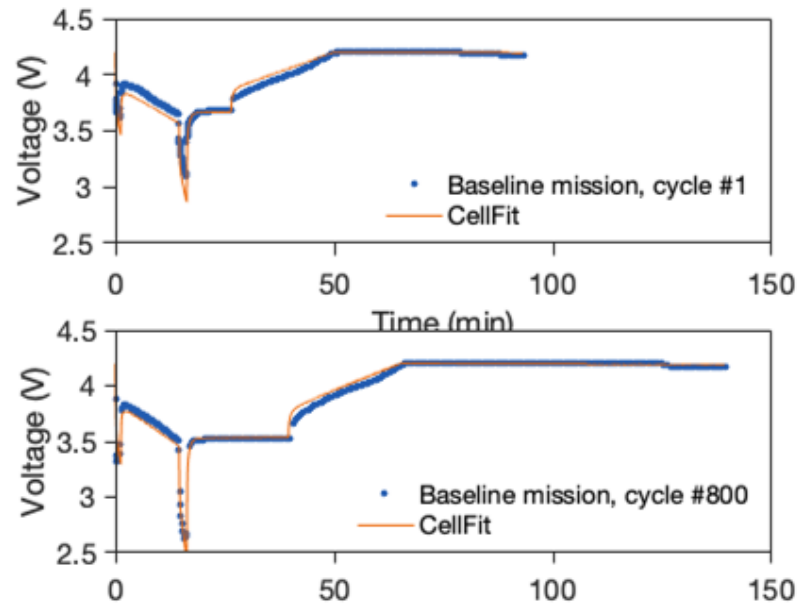
Structural identification with physics-informed neural ordinary differential equations

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

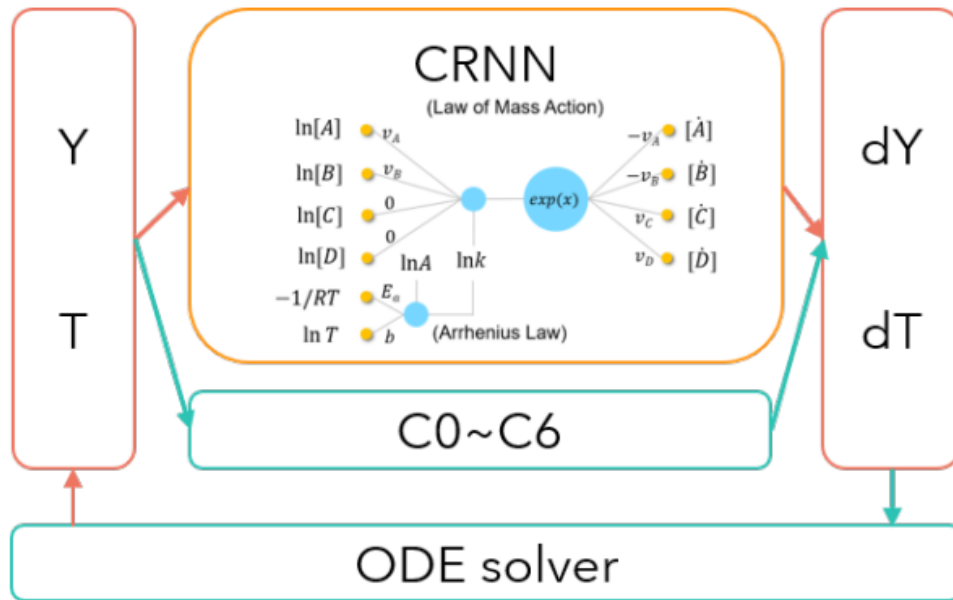
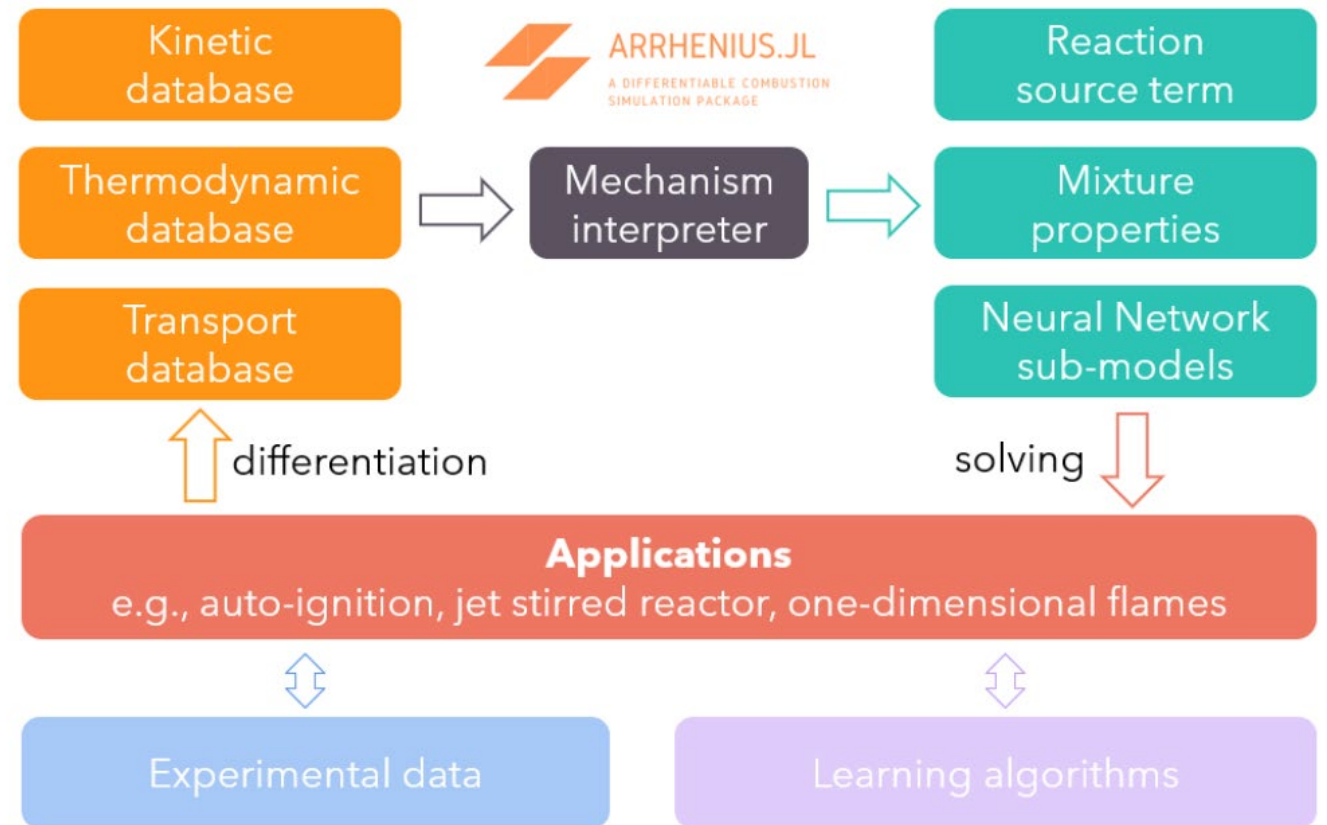


Figure 8: Schematic showing the structure of the CRNN-HyChem approach.

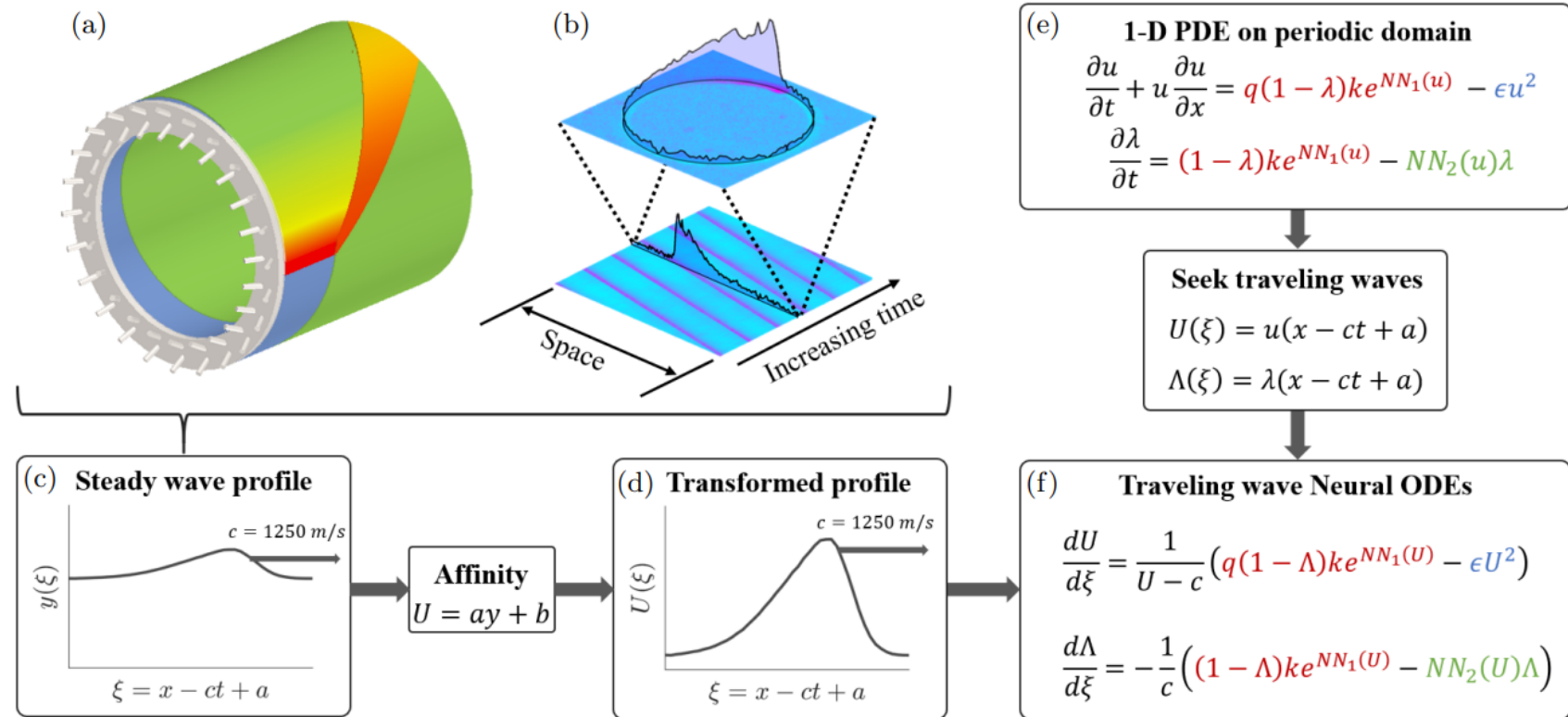
Fast automated learning of combustion models for accelerated engineering



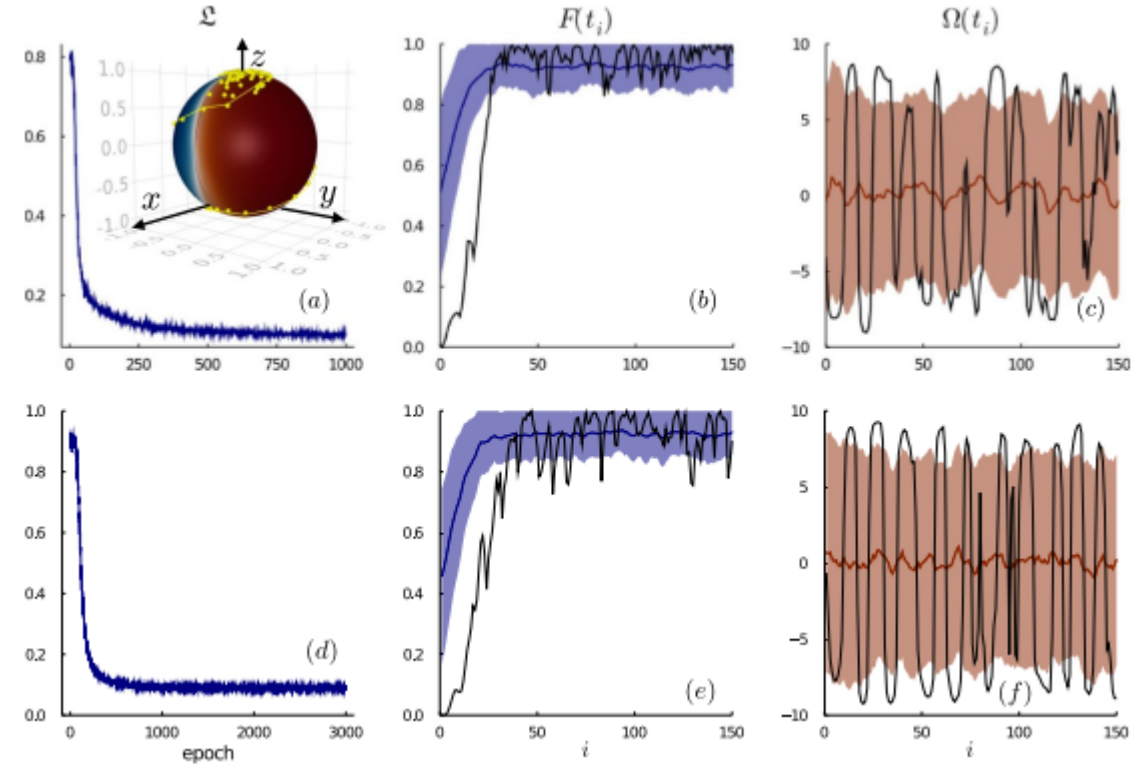
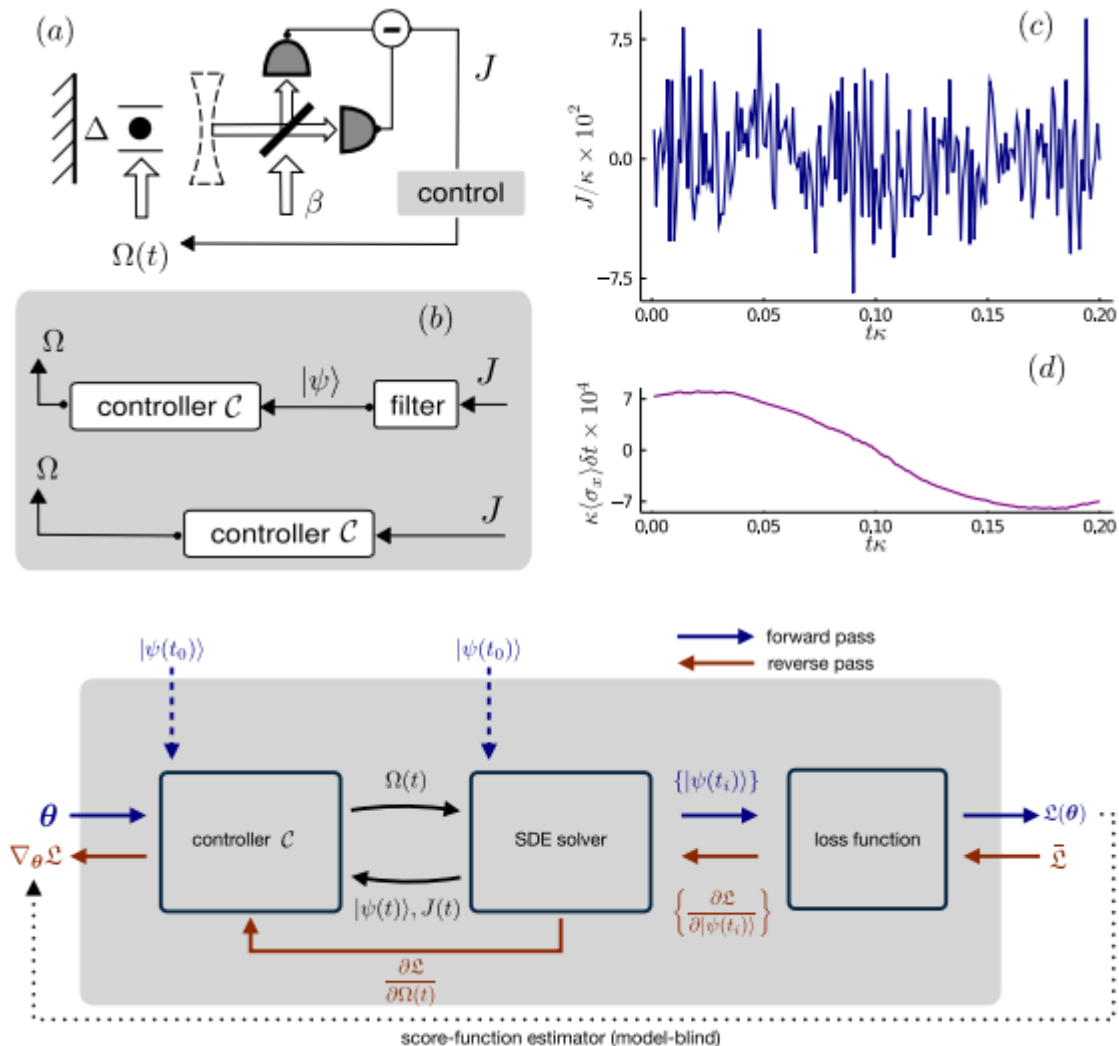
Arrhenius.jl: A Differentiable Combustion Simulation Package
Weiqi Ji, Xingyu Su, Bin Pang, Sean Joseph Cassady, Alison M. Ferris, Yajuan Li, Zhuyin Ren, Ronald Hanson, Sili Deng

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

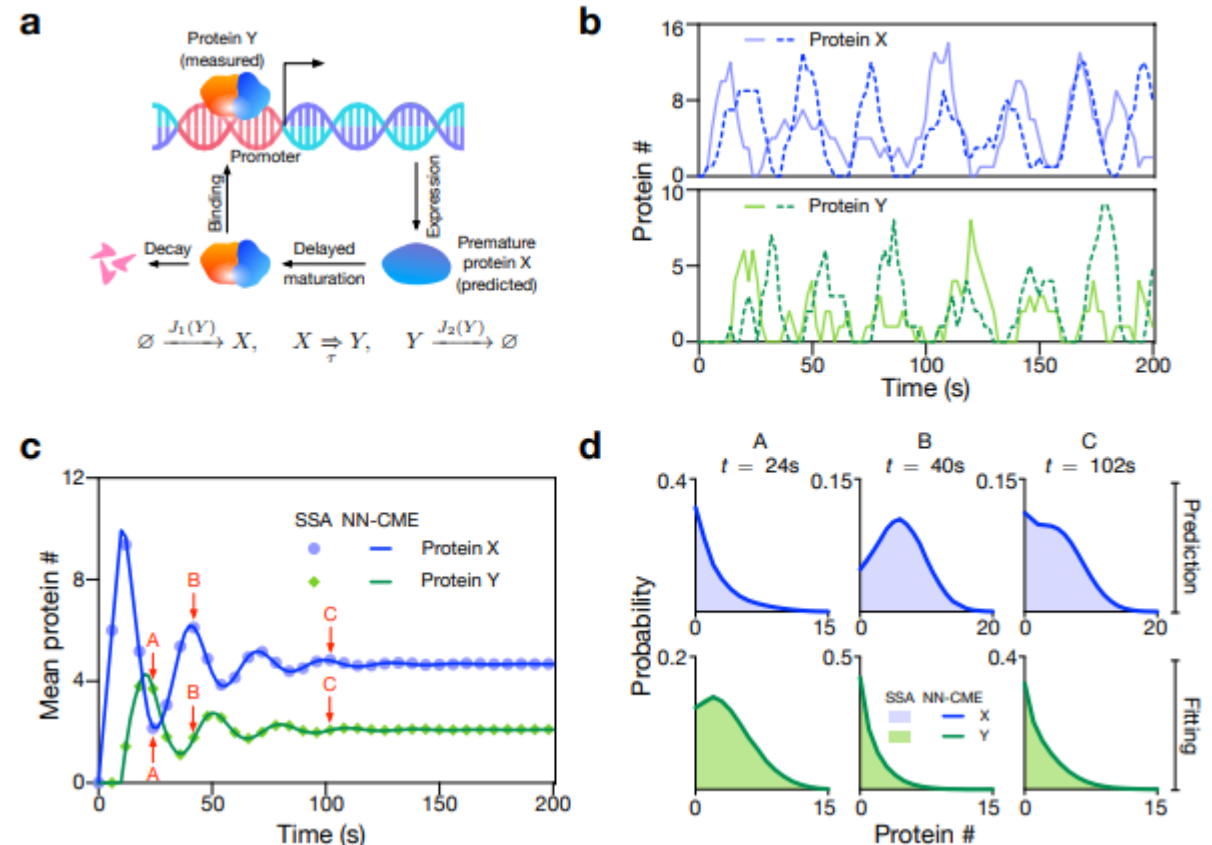
Control of stochastic quantum dynamics by differentiable programming
 Frank Schäfer, Pavel Sekatski, Martin Koppenhöfer, Christoph Bruder and Michal Kloc

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

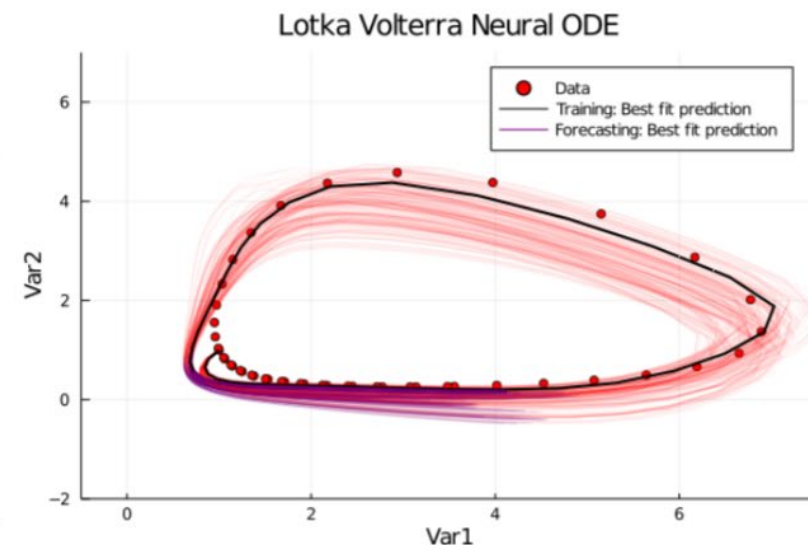
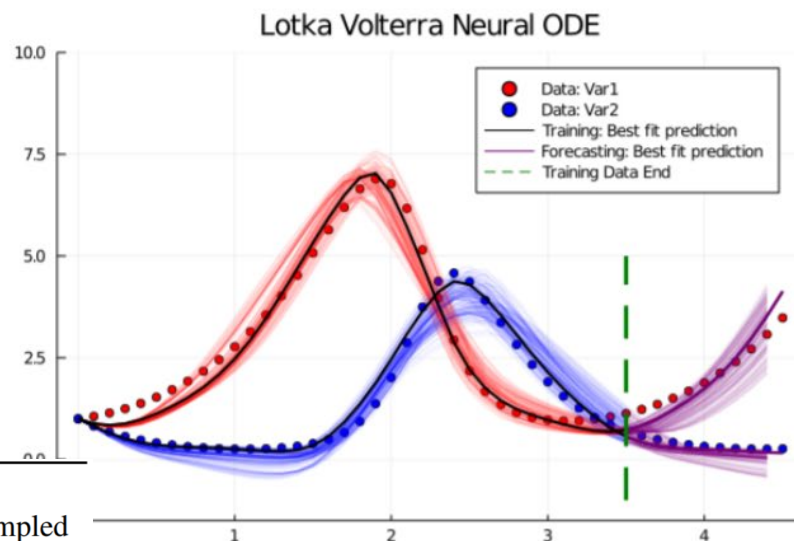
Qingchao Jiang, Xiaoming Fu, Shifu Yan, Runlai Li,
Wenli Du, Zhixing Cao, Feng Qian, Ramon Grima



Bayesian UODEs: Knowledge-Enhanced Model Discovery with UQ

Result: Probability of Missing Mechanisms

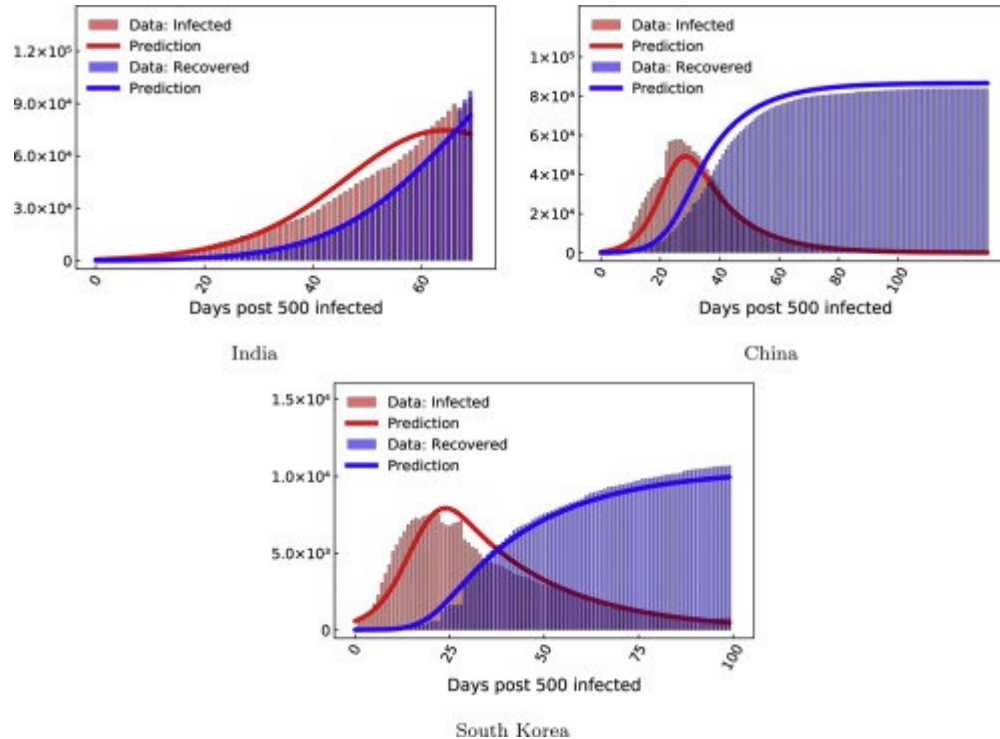
λ	Number of Active terms	Dominant terms	Error	Mean AIC score	% sampled
0.01	9	$u_1^2, u_2^2, u_1 u_2$ $u_1^2 u_2^2, u_1^2 u_2, u_2^2 u_1$	0.765	40.4	100
0.1	9	$u_1 u_2, \text{const}$ $u_1^2, u_2^2, u_1 u_2$ $u_1^2 u_2^2, u_1^2 u_2, u_2^2 u_1$	0.764	35	100
1	5	$u_1 u_2, \text{const}$ u_1^2, u_2^2, u_2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



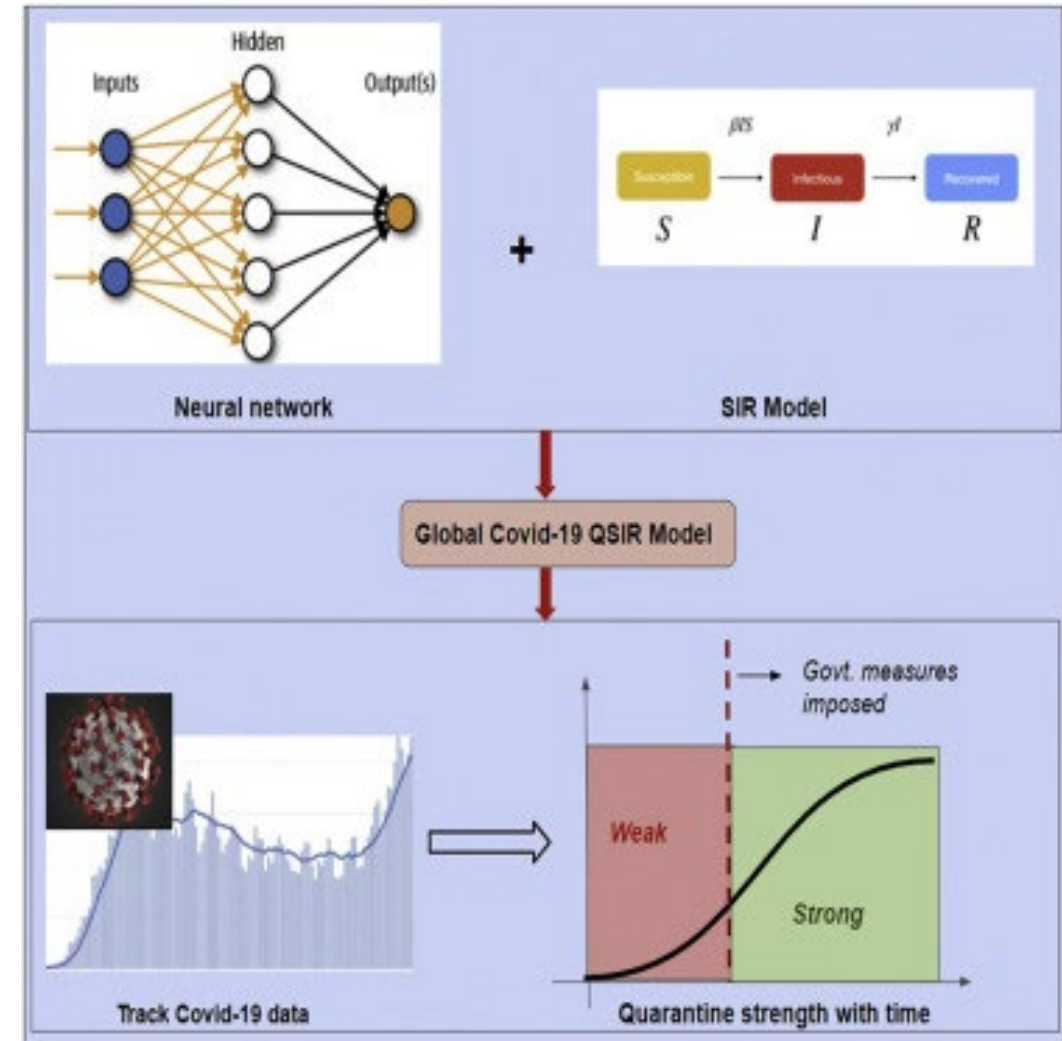
```
function lotka_volterra!(du, u, p, t)
    x, y = u
    α, β, δ, γ = p
    du[1] = dx = α*x - β*x*y - δ*x
    du[2] = dy = γ*y - δ*y
end
```



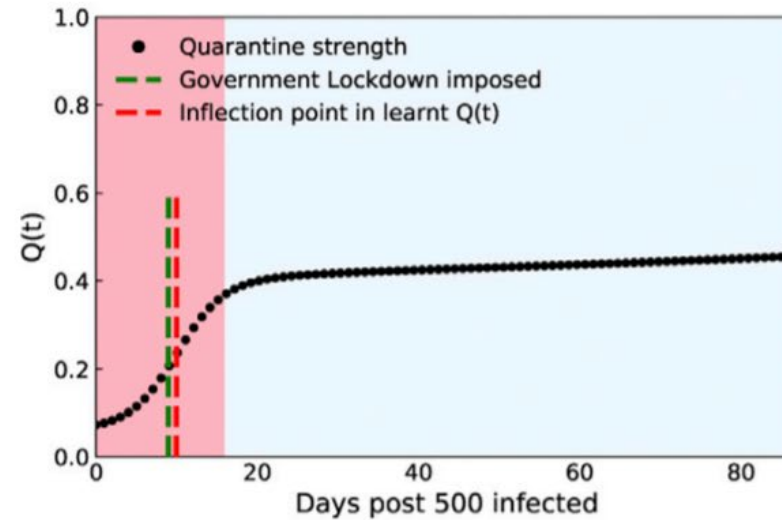
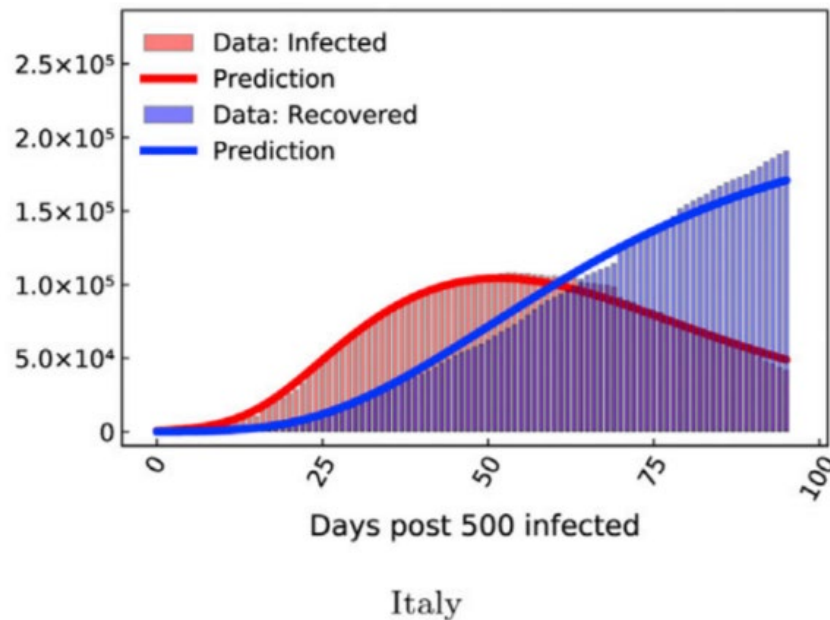
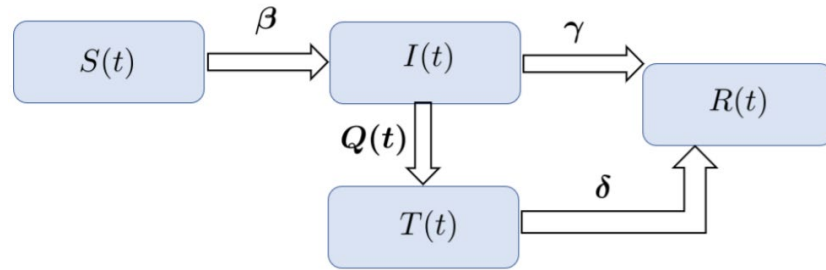
Demonstration of UDE Epidemic Models



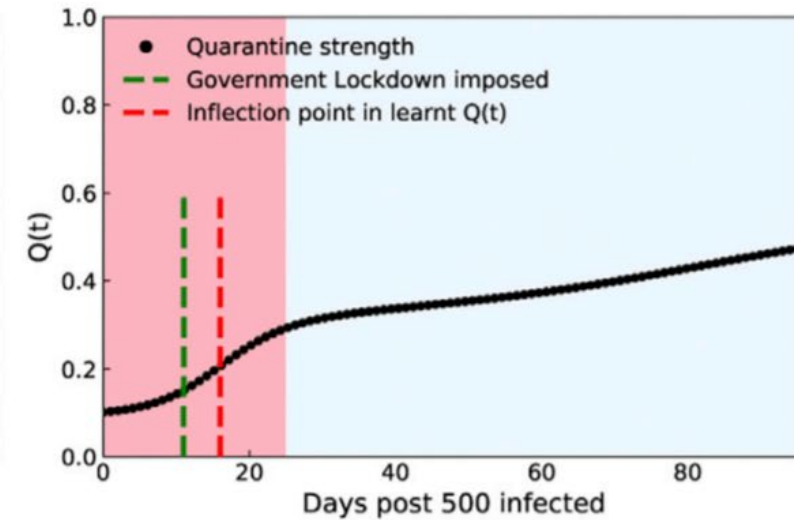
Dandekar, Raj, Chris Rackauckas, and George Barbastathis. "A machine learning aided global diagnostic and comparative tool to assess effect of quarantine control in Covid-19 spread." *Cell Patterns* (2020).



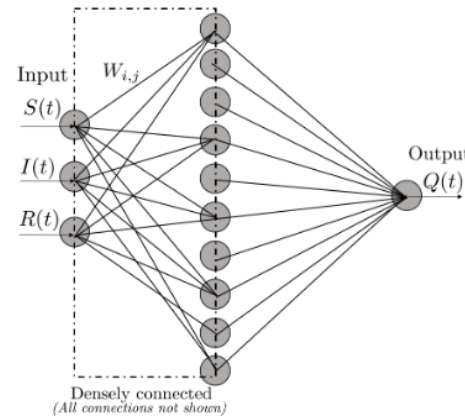
QSIR Predicts Quarantine Measure Evolution



Spain

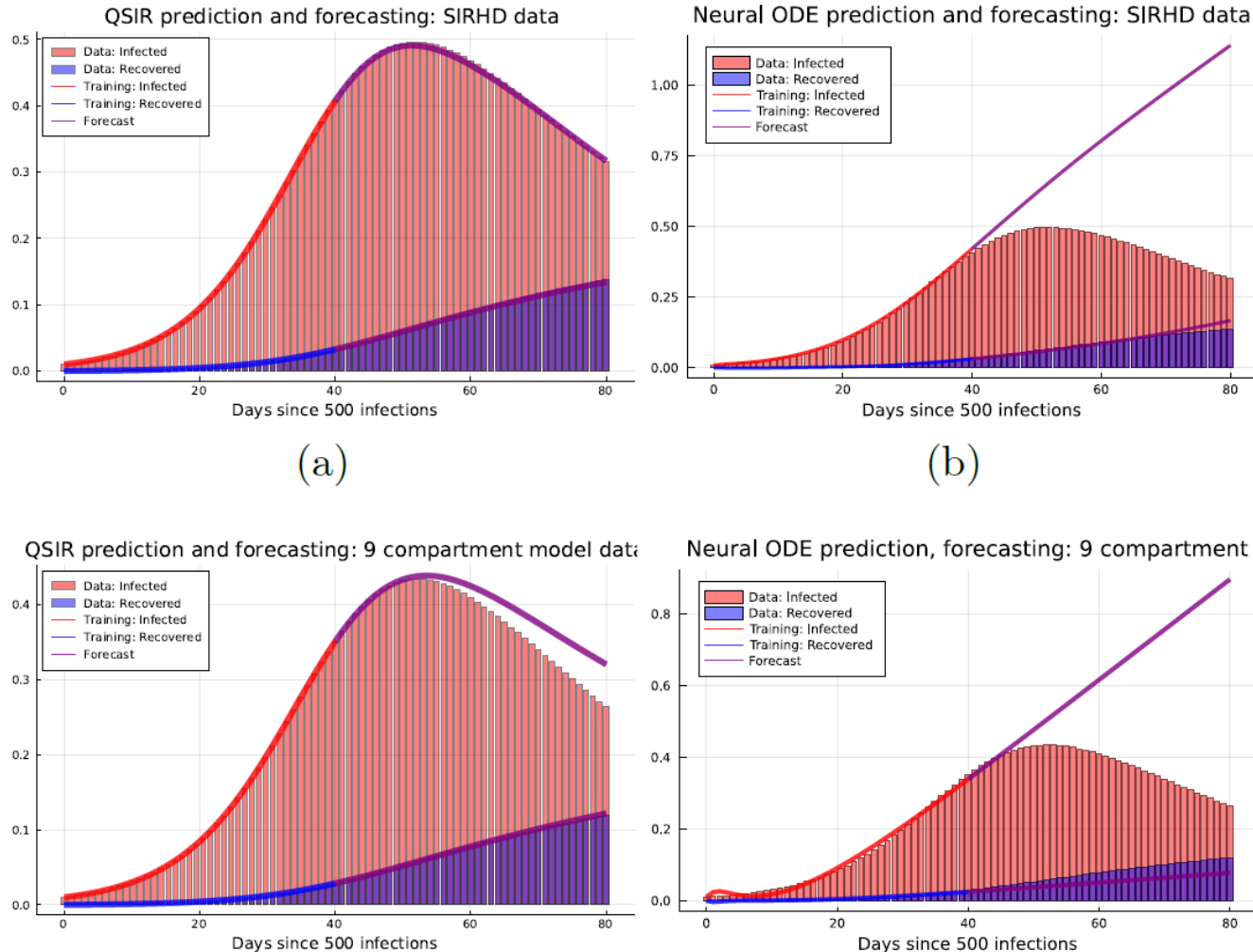


Italy



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

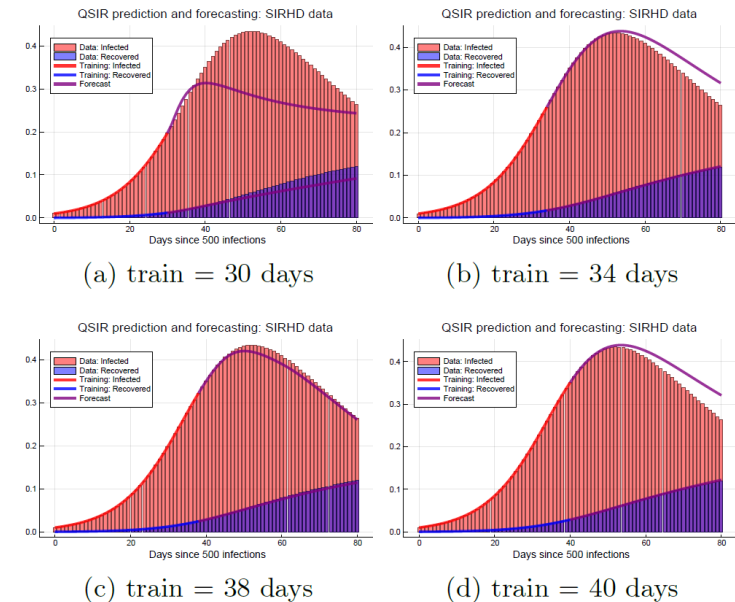


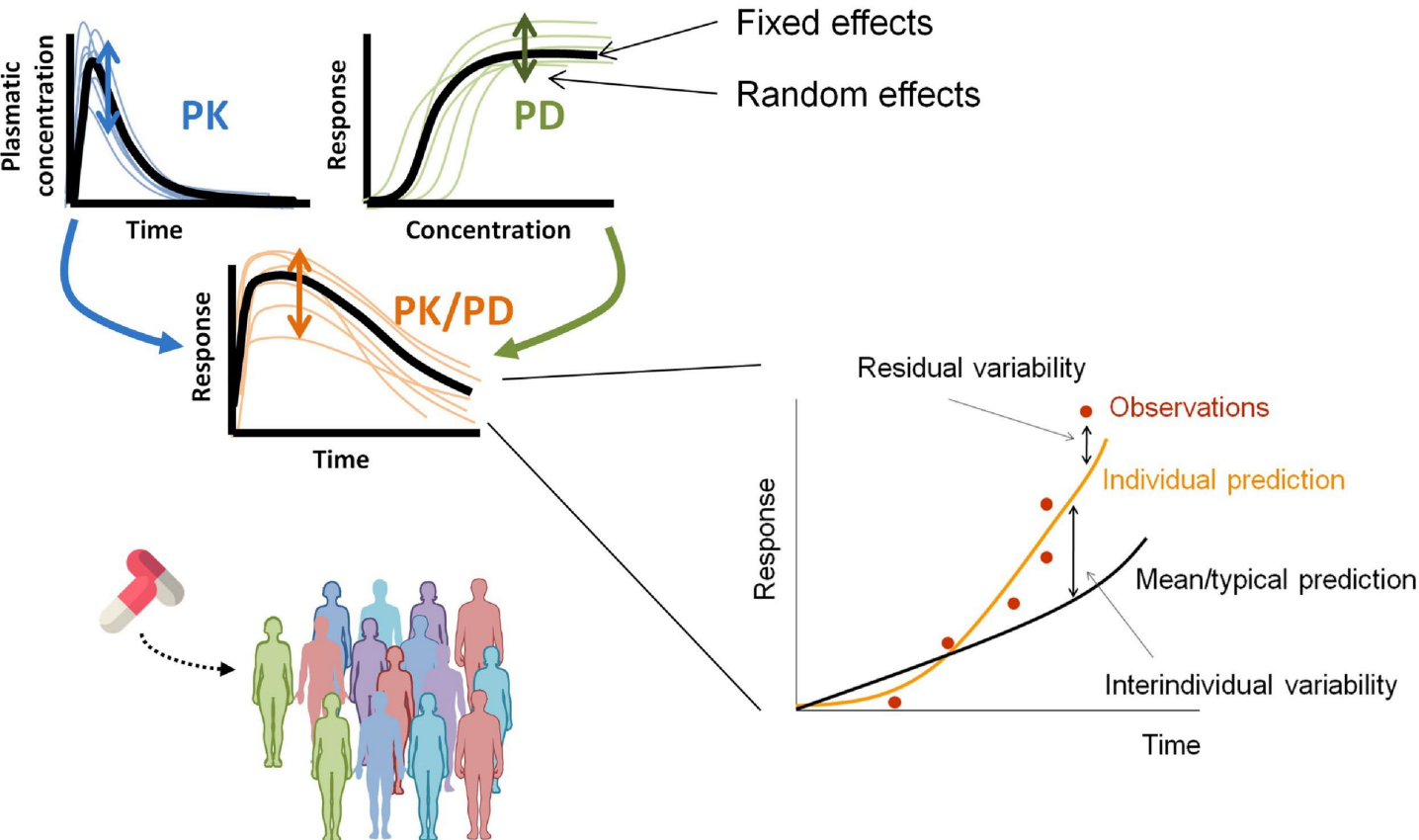
Figure 5: EpiSciML QSIR model - 9 compartment: Effect of training size on forecasting performance Figure shows the prediction and forecasting performance of the QSIR model for training data size of (a) 30, (b) 34, (c) 38, (d) 40 data points.

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



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

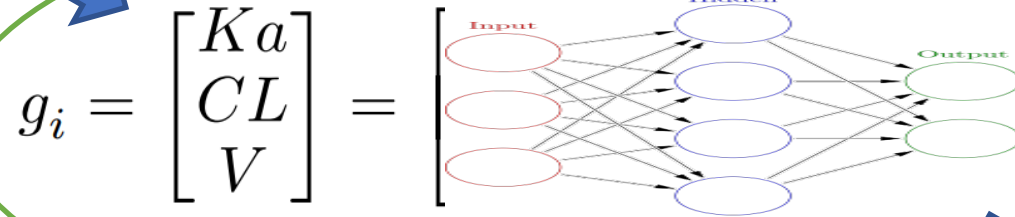
From Dynamics to Nonlinear Mixed Effects (NLME) Modeling

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)

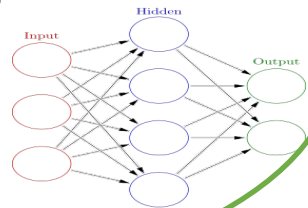
How can we find these models?

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
     $\theta \in \text{VectorDomain}(\text{lower}=[0.1, 0.0008, 0.0040, 0.1], \text{upper}=[5.0, 0.5, 0.9, 5.0])$ 
     $\Omega \in \text{PSDDomain}(3)$ 
     $\sigma^2_{\text{add}} \in \text{RealDomain}(\text{lower}=0.001, \text{init}=\text{sqrt}(0.388))$ 
    p1  $\in \text{NeuralDomain}(\text{FastChain}(\text{FastDense}(2, 50, \text{tanh}), \text{FastDense}(50, 1), (x, p) \rightarrow x.^2))$ 
    p2  $\in \text{NeuralDomain}(\text{FastChain}(\text{FastDense}(2, 50, \text{tanh}), \text{FastDense}(50, 1), (x, p) \rightarrow x.^2))$ 
  end

  @random begin  $\eta \sim \text{MvNormal}(\Omega)$  end

  @pre begin
    Ka = SEX == 0 ?  $\theta[1] + \eta[1]$  :  $\theta[4] + \eta[1]$ 
    K = nn1( $[\theta[2], \eta[2]]$ , p1)[1]
    CL = nn2( $[\theta[3] * \text{WT}, \eta[3]]$ , p2)[1]
    Vc = CL/K
    SC = CL/K/WT
  end

  @covariates SEX WT
  @vars begin conc = Central / SC end
  @dynamics Depots1Central1
  @derived begin dv  $\sim @. \text{Normal}(\text{conc}, \text{sqrt}(\sigma^2_{\text{add}}))$  end
end
```



Utilize GPU acceleration for neural networks

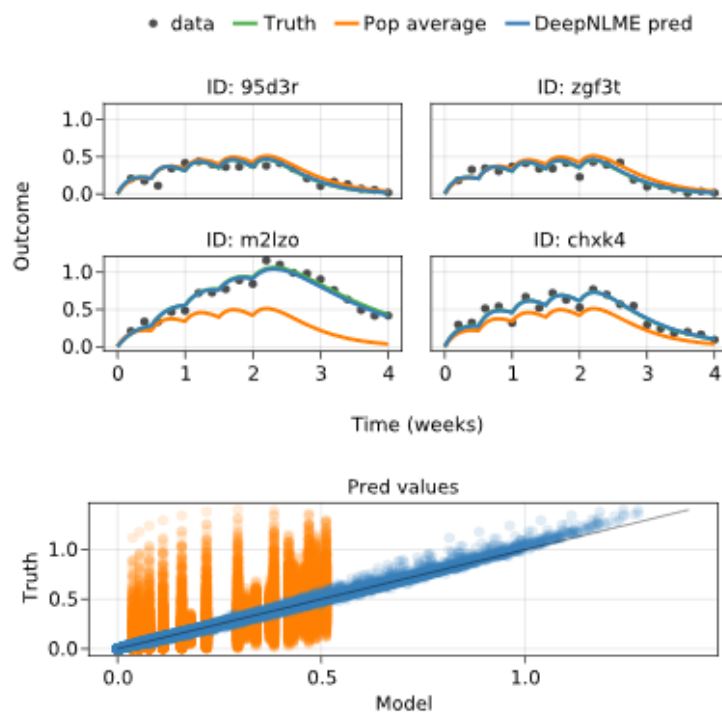
Automate the discovery of covariate models

- **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**

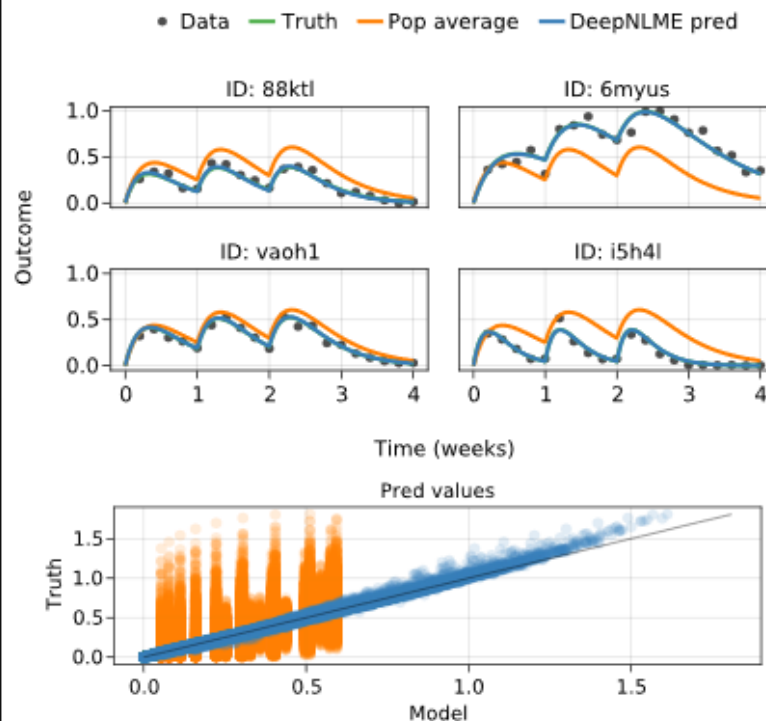
Currently being tested on clinical trial data

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

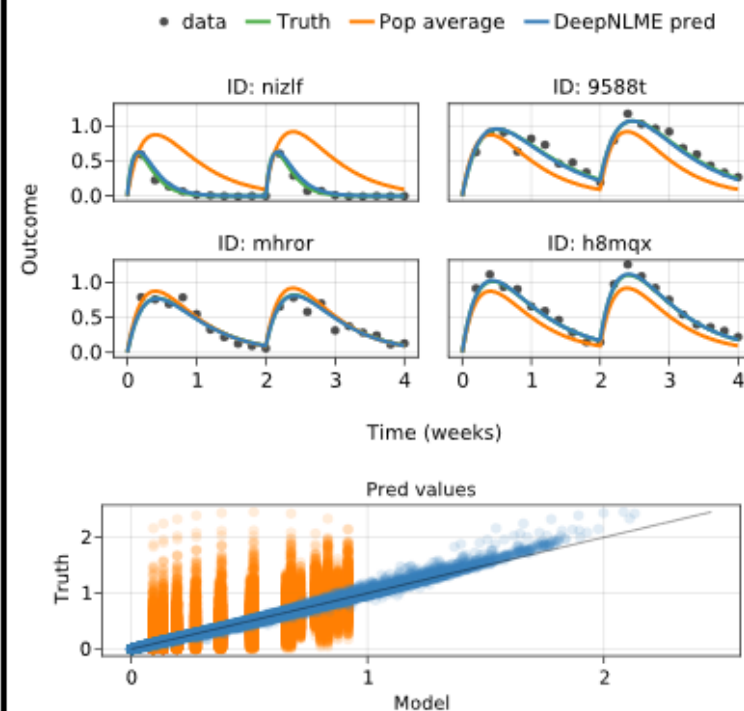
Predicted



Trained



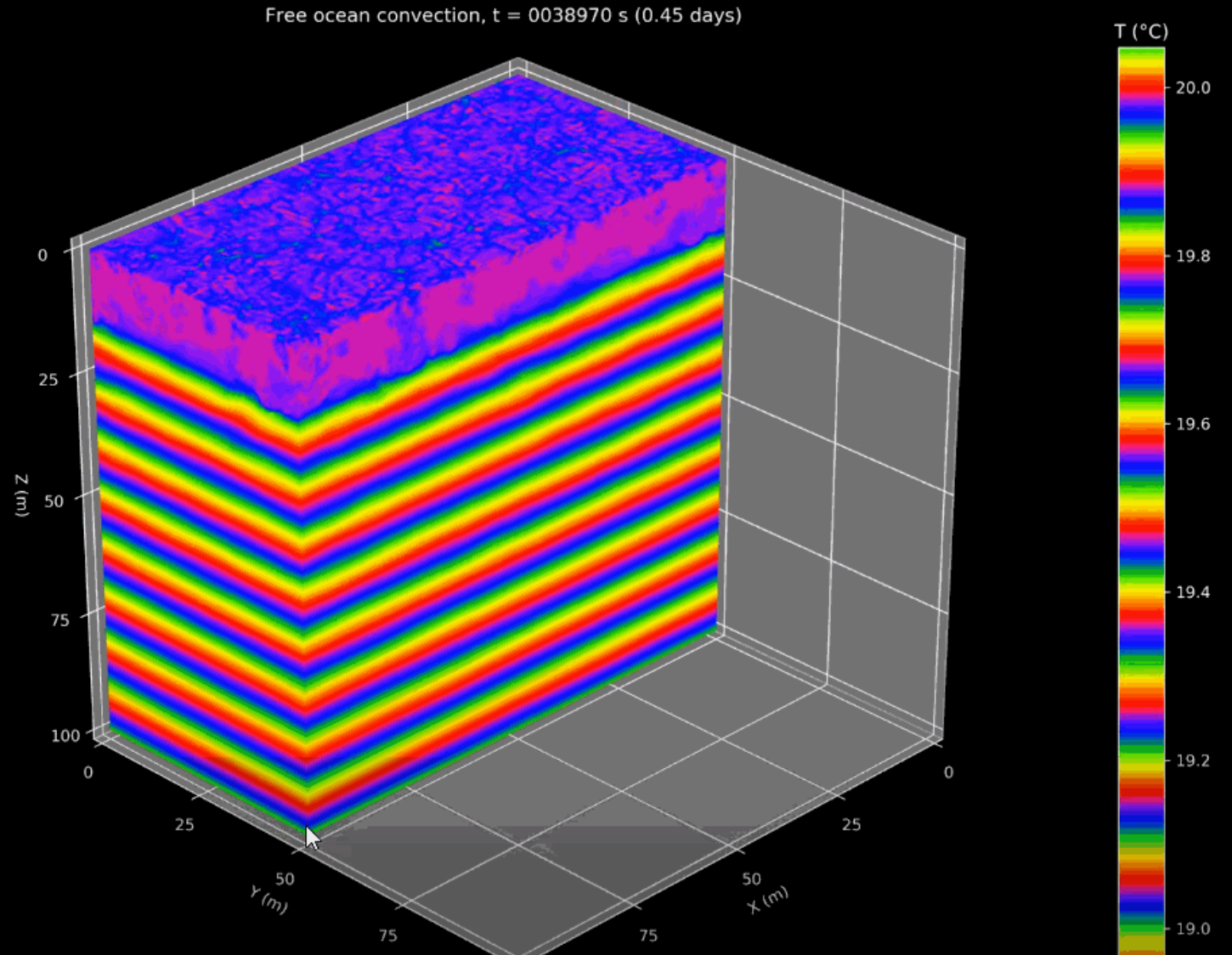
Predicted



High fidelity surrogates of ocean columns for climate models

**3D simulations are
high resolution but
too expensive.**

**Can we learn faster
models?**



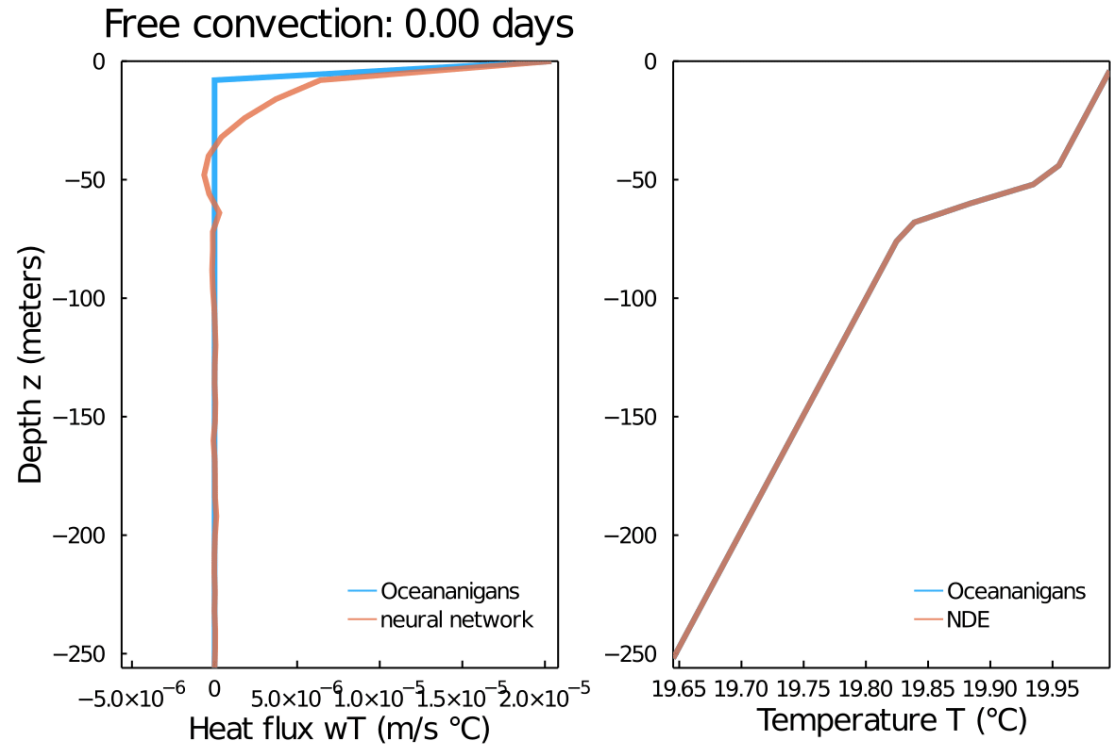
Neural Networks Infused into Known Partial Differential Equations

Derive a 1D approximation to the 3D model

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

Incorporate the “convective adjustment”

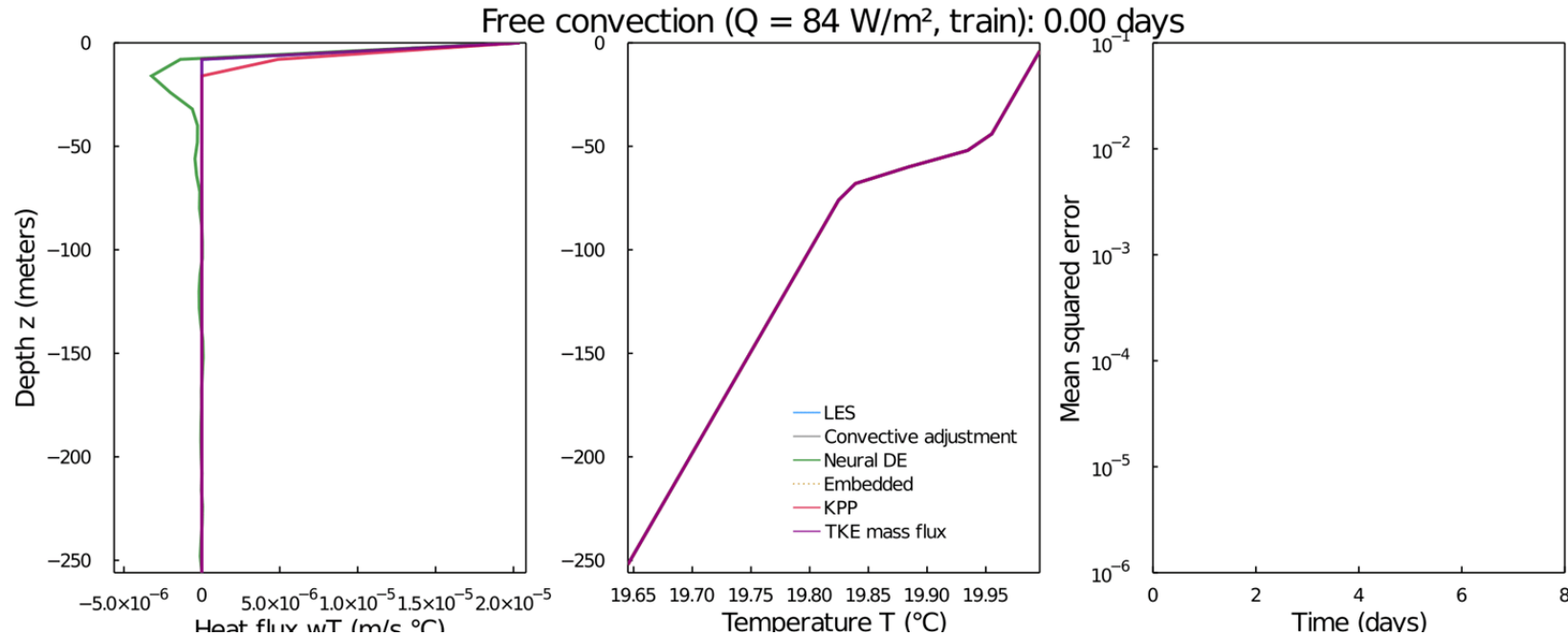
$$K = \begin{cases} 0 & \text{if } \partial_z T > 0 \\ 100 \text{ m}^2/\text{s} & \text{if } \partial_z T < 0 \end{cases}$$



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

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

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

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

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

$$\lambda(T) = 0$$

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

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

Differentiating Ordinary Differential Equations: Summary

Summary:

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

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

$$\lambda(T) = 0$$

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

Differentiating Ordinary Differential Equations: Step 2 Details

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

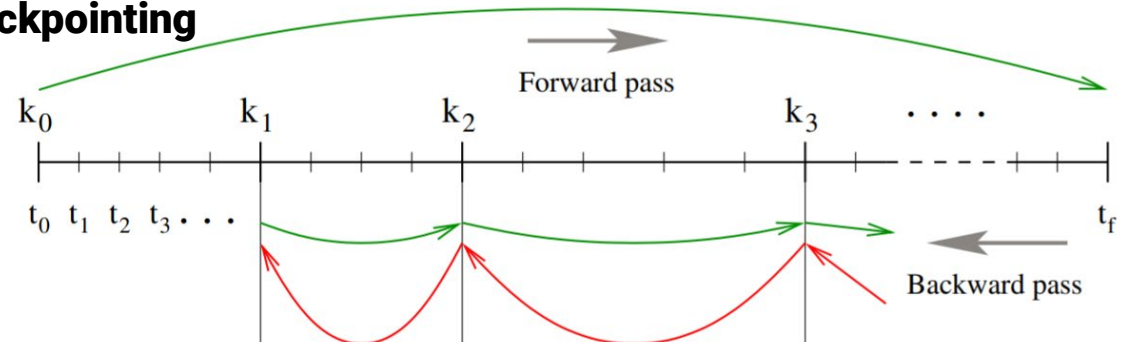
$$\lambda(T) = 0$$

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

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

2. Store $u(t)$ while solving forwards (dense output)

3. Checkpointing



How the gradient (adjoint) is calculated also matters!

This term is traditionally computed via differentiation and then multiplied to lambda

Reverse-mode embedded implementation: push-forward $f(u)$ pullback lambda
Computational cost $O(n) \rightarrow O(1)$ f evaluations and automatically uses optimized backpropagation!

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

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


Adjoint Differential Equation

Six choices for this computation:

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

Differentiating Ordinary Differential Equations: Step 3 Details

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



How do you calculate the integral?

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

What's the trade-off between these ideas?

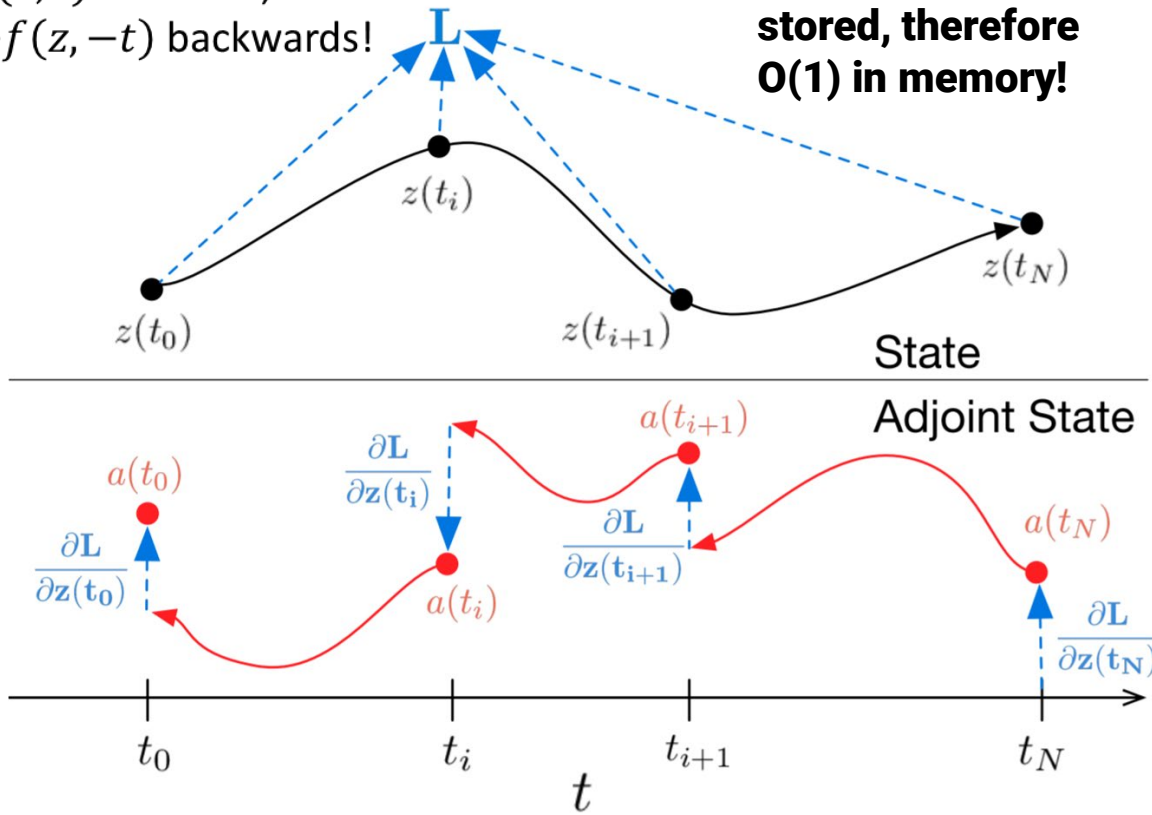
Some methods are “mathematically correct”, but “numerically incorrect”

SciML is a software problem.

Machine Learning Neural Ordinary Differential Equations

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

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



The adjoint equation is an ODE!

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

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

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

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

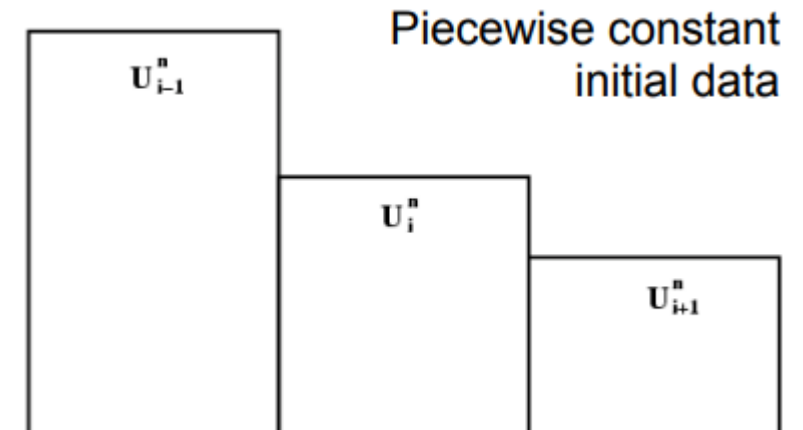
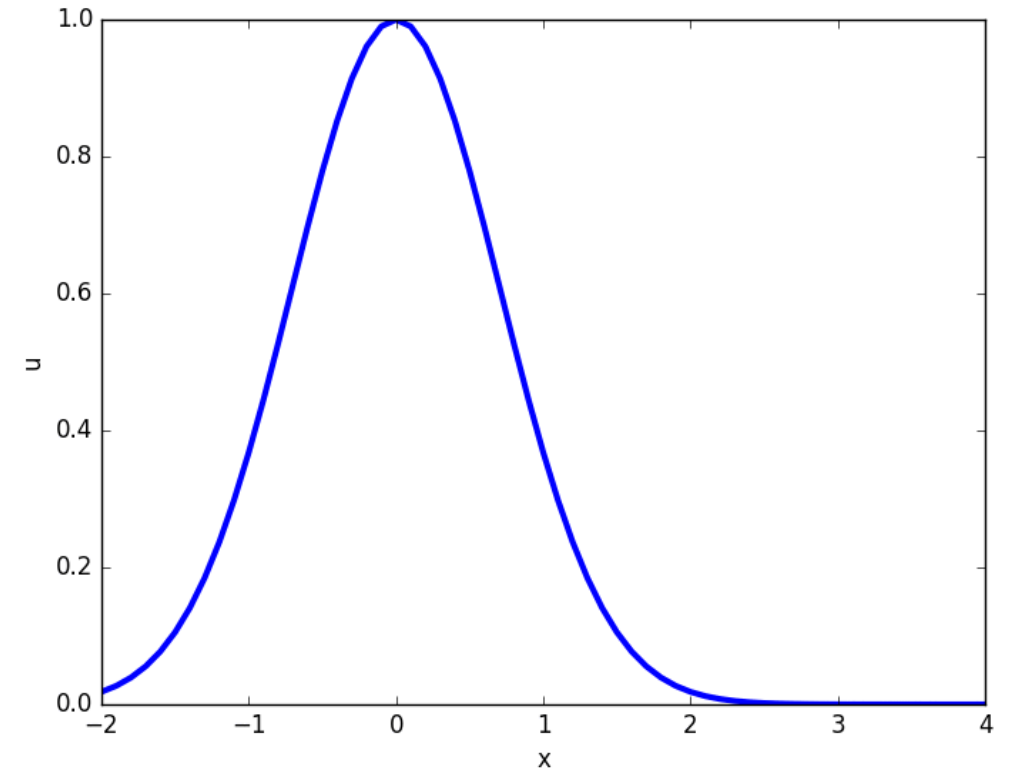
Advection Equation: $\frac{\partial u}{\partial t} + a \frac{\partial u}{\partial x} = 0$

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

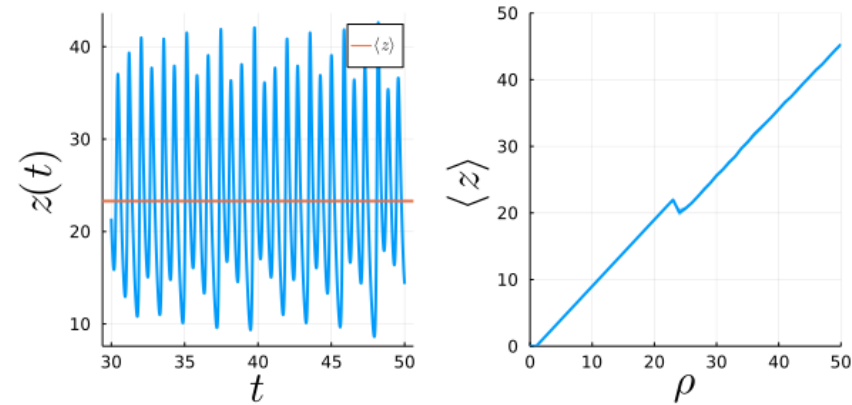
$$u'_i = -\frac{a(u_i - u_{i-1})}{\Delta x} \text{ or } u'_i = -\frac{a(u_{i+1} - u_i)}{\Delta x}?$$

If you discretize in the wrong direction you get unconditional instability

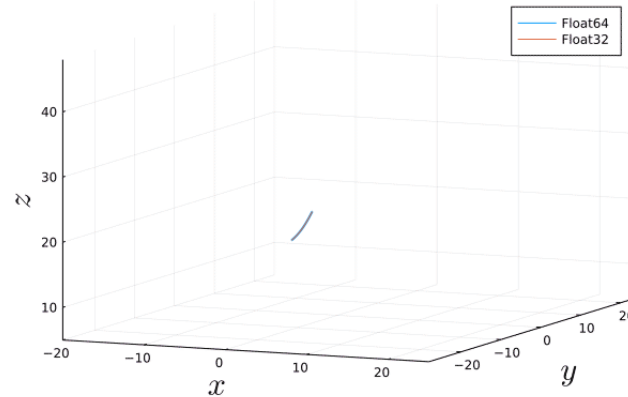
You need to understand the engineering principles and the numerical simulation properties of domain to make ML stable on it.



Differentiation of Chaotic Systems: Shadow Adjoints



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



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

- **AD** and finite differencing fails!

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

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

- **Shadowing methods** in DiffEqSensitivity.jl

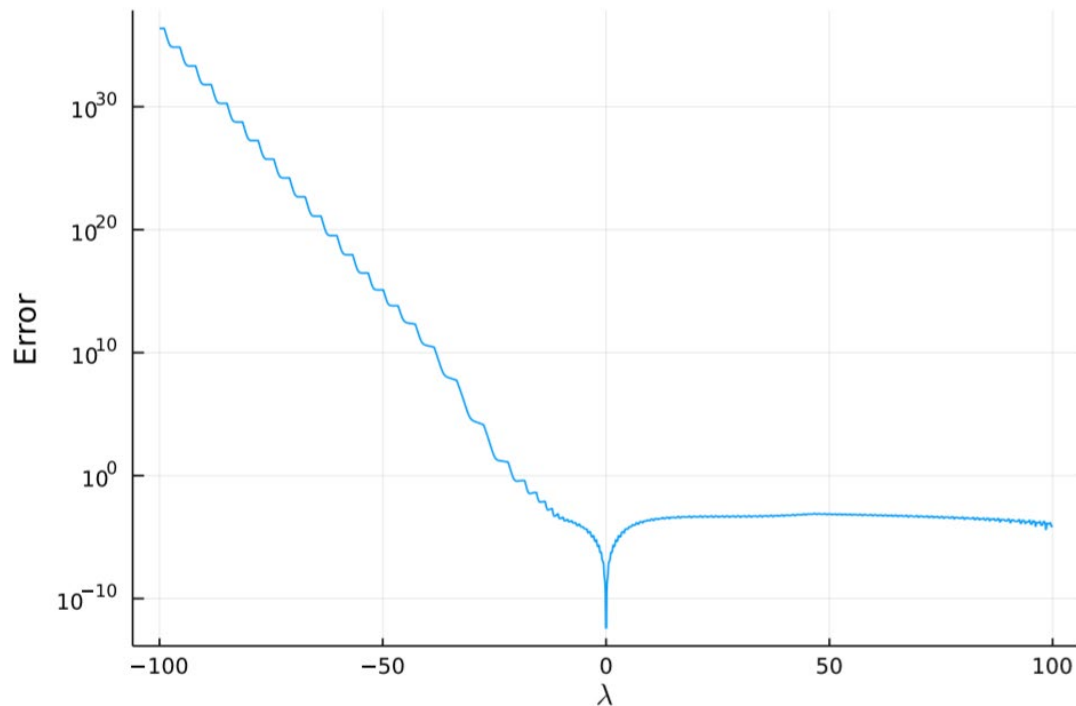
$$\left. \frac{d\langle z \rangle_{\infty}}{d\rho} \right|_{\rho=28} \approx 1.028 \text{ (LSS/AdjointLSS)}$$

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

Problems With Naïve Adjoint Approaches On Stiff Equations

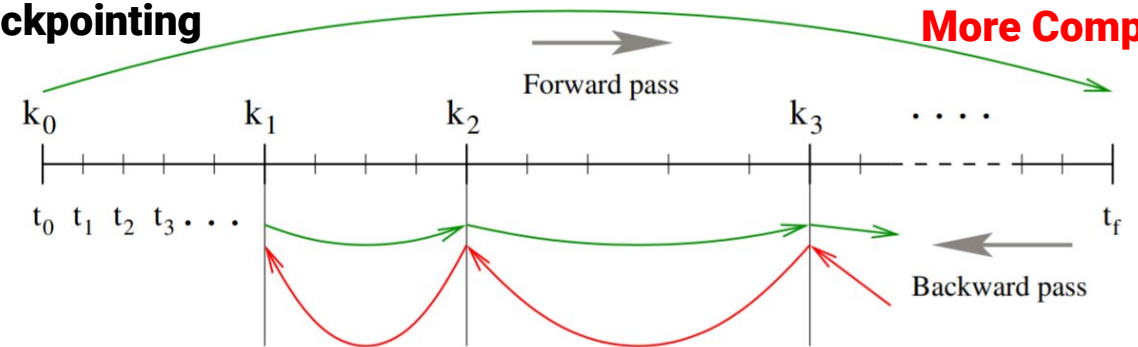
Error grows exponentially...

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



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

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



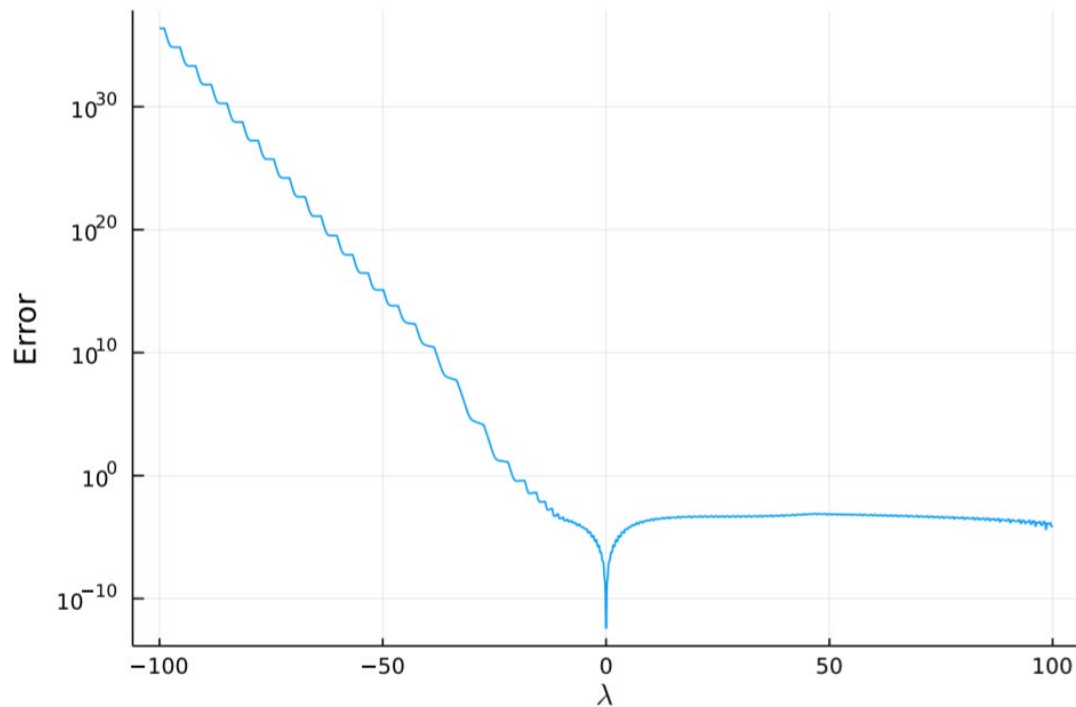
Each choice has an engineering trade-off!

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:



Compute cost is cubic with parameter size when stiff

Size of reverse ODE system is:

$$2states + parameters$$

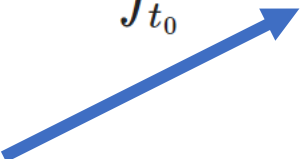
Linear solves inside of stiff ODE solvers, ~cubic

Thus, adjoint cost:

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

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

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.

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

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

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

Figure 1 is a log-log plot showing the execution time (s) versus the error for various ODE solvers. The x-axis represents the error on a logarithmic scale from 10^{-9} to 10^{-3} . The y-axis represents the time in seconds on a logarithmic scale from $10^{-4.0}$ to $10^{-1.5}$. The plot compares 12 different solvers, grouped by language: Julia (Rosenbrock23, TRBDF2, radau), Hairer (rodas, radau), MATLAB (ode23s, ode15s), SciPy (LSODA, BDF, odeint), deSolve (lsoda), and Sundials (CVODE). The plot shows that for a given error, the time generally increases as the error decreases, and that some solvers (like Julia's Rosenbrock23) are significantly faster than others (like SciPy's LSODA) for the same error level.

Foundation: Fast Differential Equation Solvers

DifferentialEquations.jl is:

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

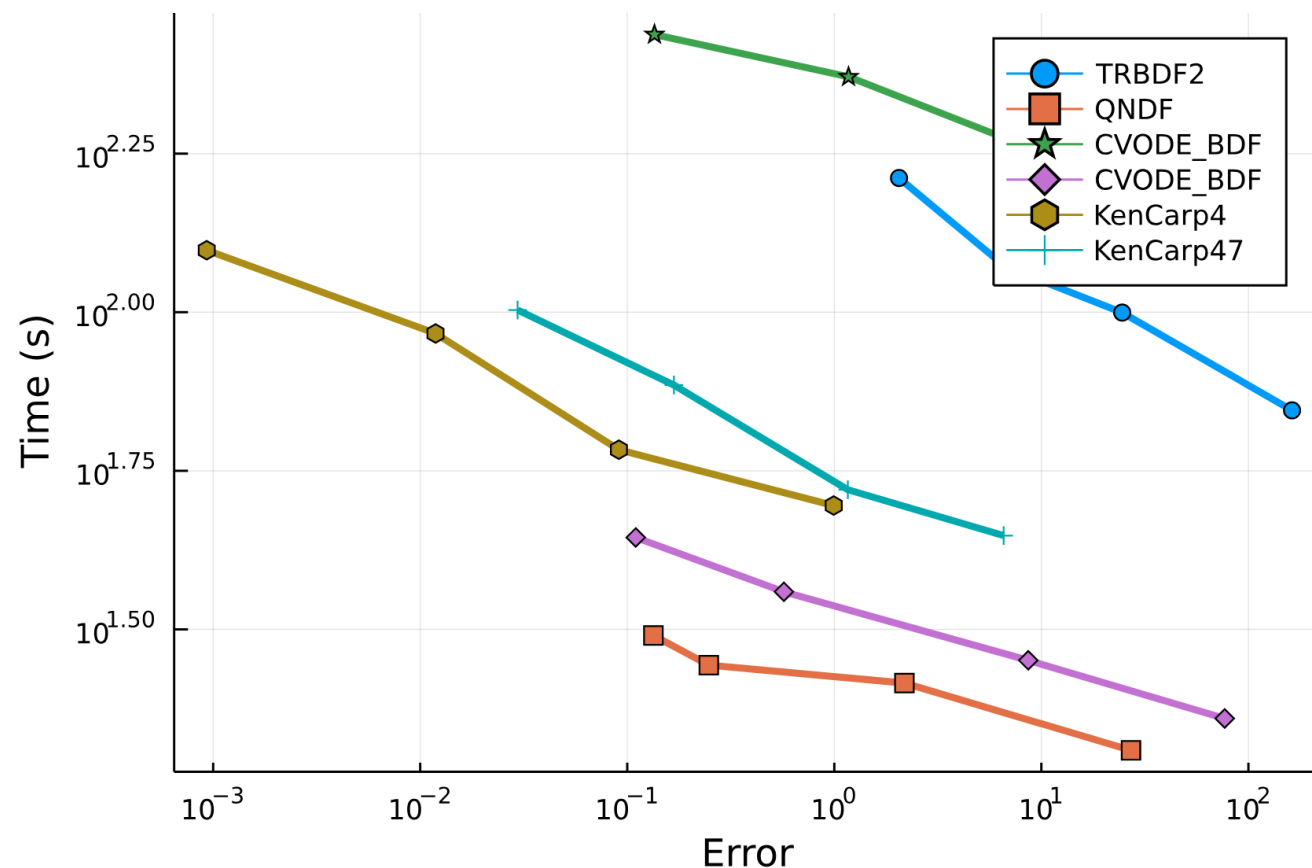
This excludes the extra 2x from symbolics and 2x from sparse parallel compilation!

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

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

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

1122 Stiff ODEs: BCR Chemical Reaction Network



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

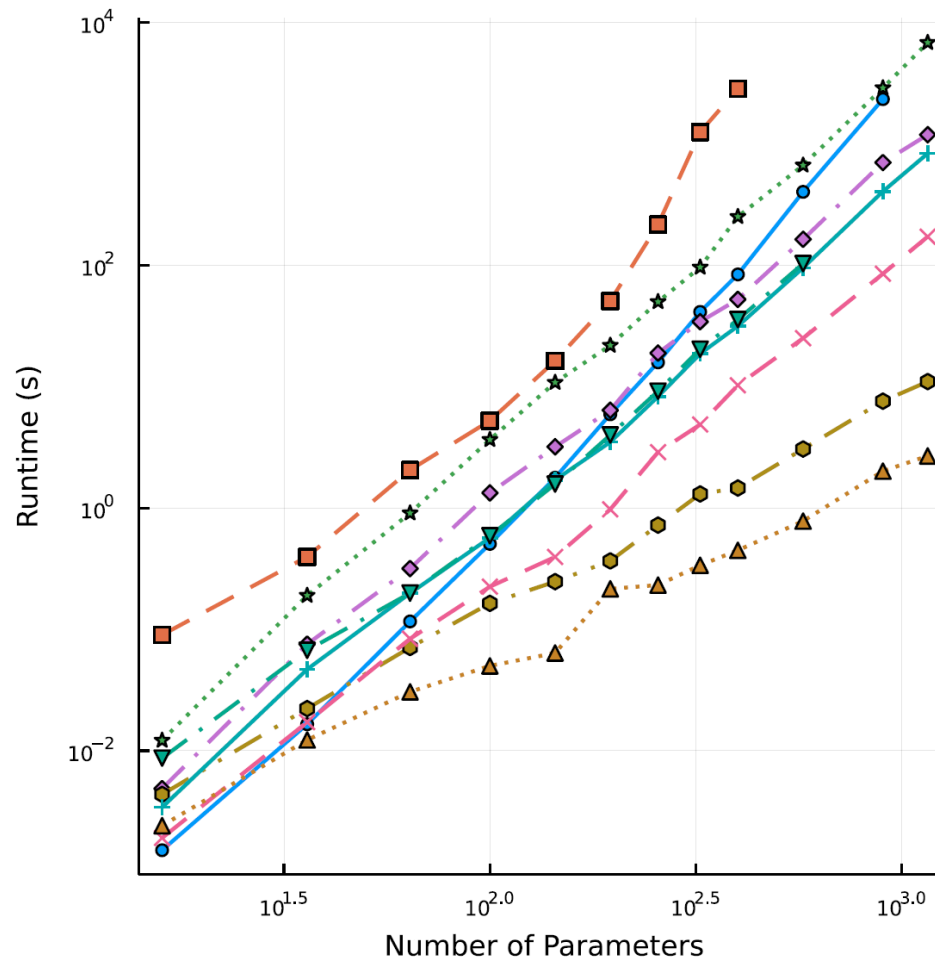
Method	Stability	Stiff Performance Scaling	Memory Usage
BacksolveAdjoint	Poor	$O((s + p)^3)$	Low. $O(1)$
InterpolatingAdjoint	Good	$O((s + p)^3)$	High. Requires full continuous solution of forward
QuadratureAdjoint	Good	$O(s^3 + p)$	Higher. Requires full continuous solution of forward and Lagrange multiplier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

How the adjoint is calculated also matters!

Gradient calculations on a stiff PDE, varying Δt

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

Sensitivity Scaling on Brusselator



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

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

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

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

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

Relative time to solve

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

Spiral Neural ODE (from original Neural ODE paper)

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

Geometric Brownian Motion of size 4

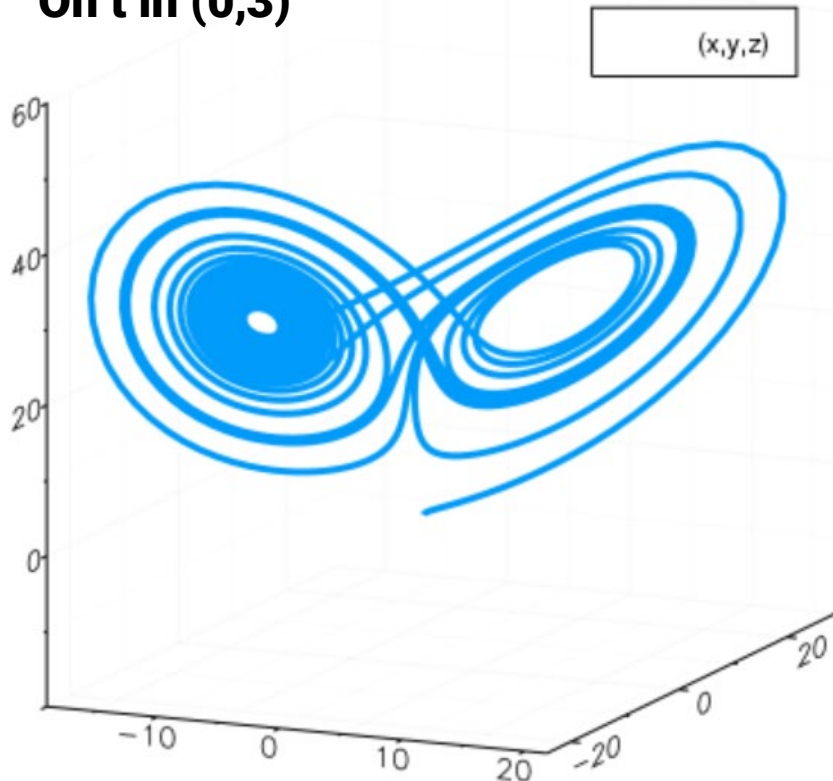
The SDE is solved 100 times. The summary of the results is as follows:

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

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

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)
```

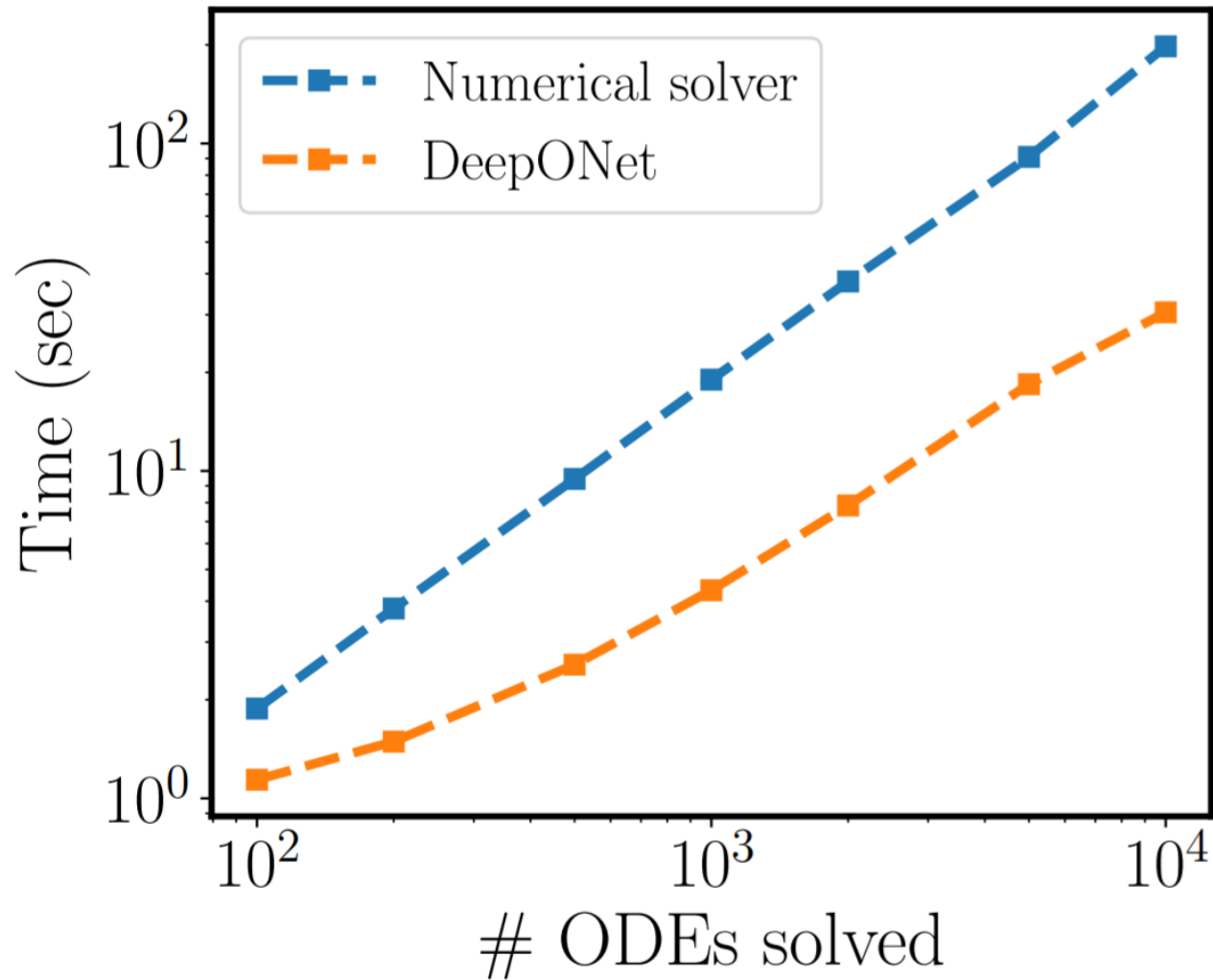
Note on Neural Networks “Outperforming” Classical Solvers

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

Sifan Wang, Paris Perdikaris

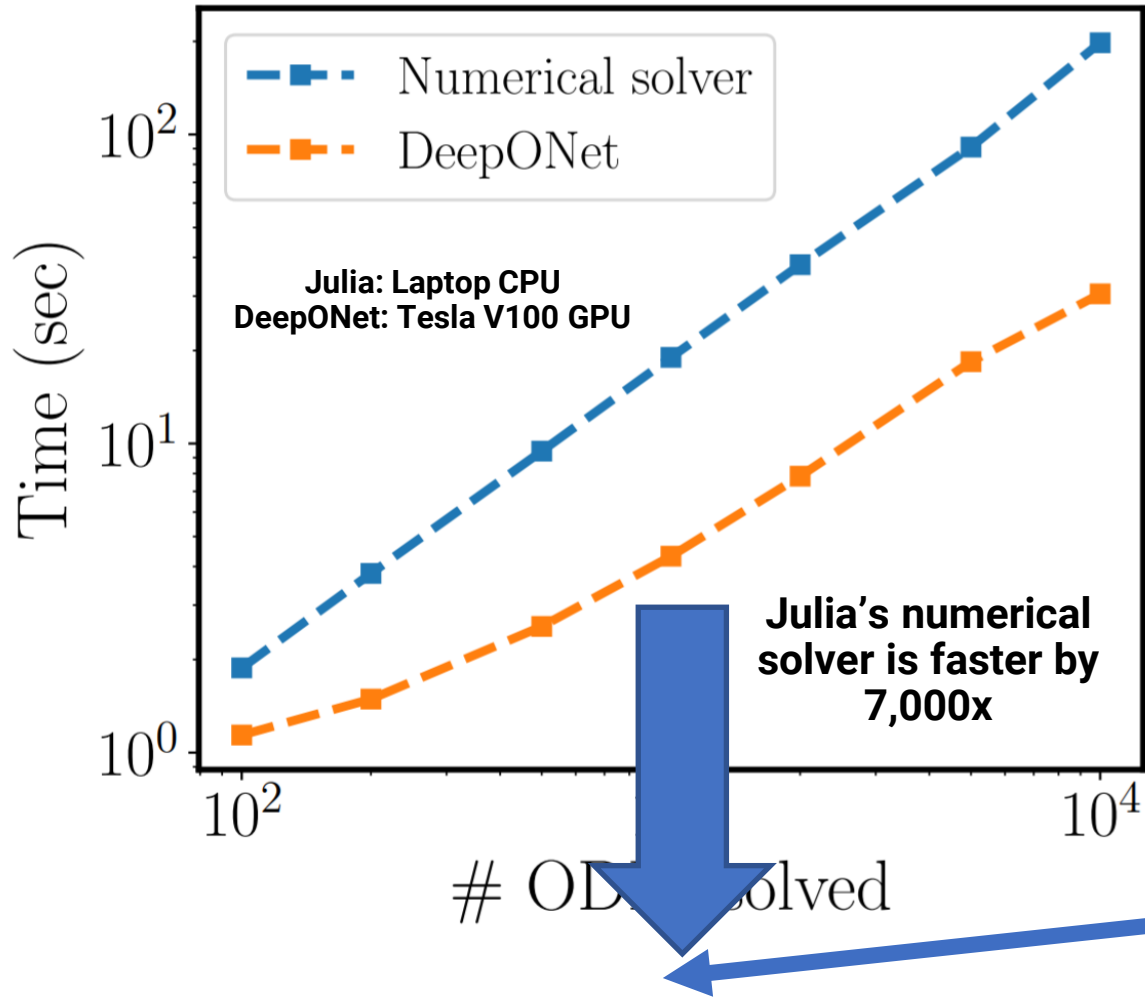
Ordinary and partial differential equations (ODEs/PDEs) play a paramount role in analyzing and simulating complex dynamic processes across all corners of science and engineering. In recent years machine learning tools are aspiring to introduce new effective ways of simulating PDEs, however existing approaches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

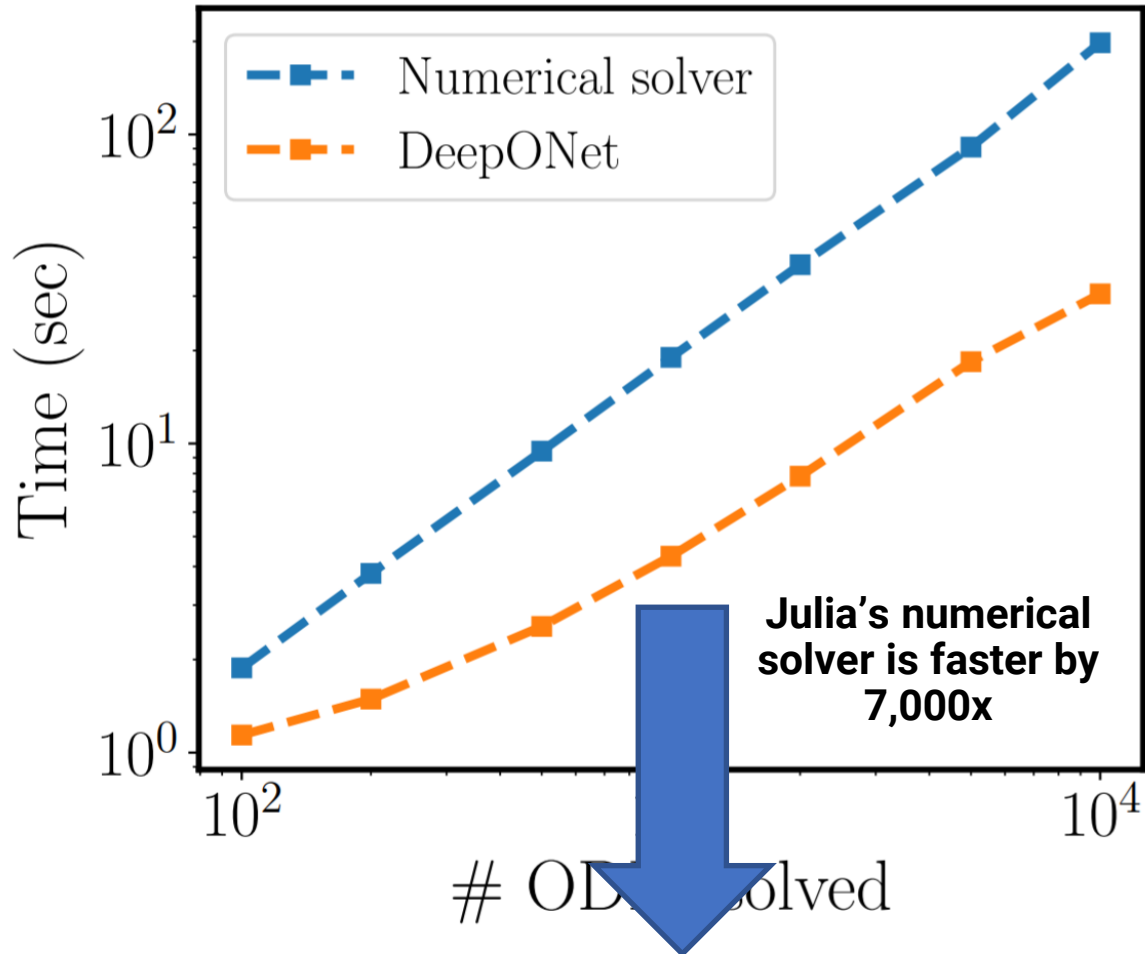
```
@time solve(monteprob, Rodas5(), EnsembleThreads(), trajectories=1000)
```

```
#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?

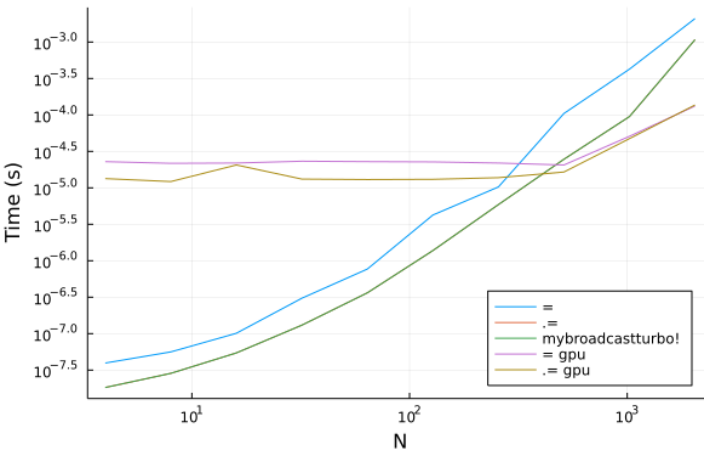


**Similar story on Fourier
Neural Operator results!**

How come so far off?

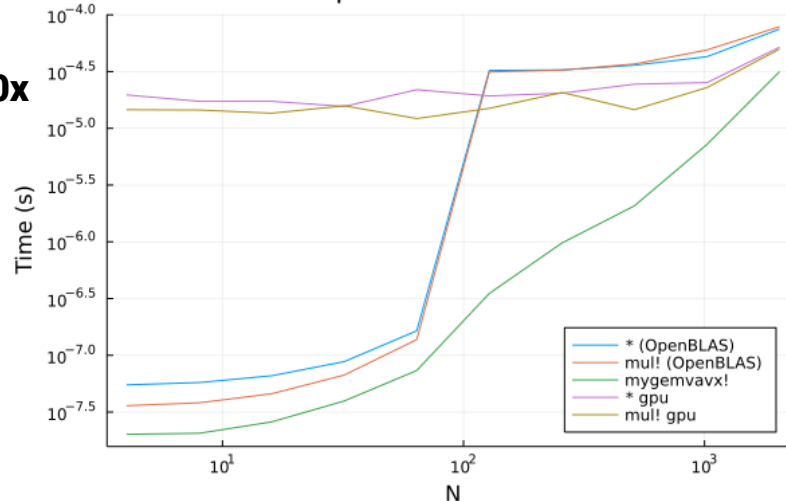
Code Optimization in Machine Learning vs Scientific Computing

Which Micro-optimizations matter for BLAS1?



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

Which Micro-optimizations matter for BLAS2?

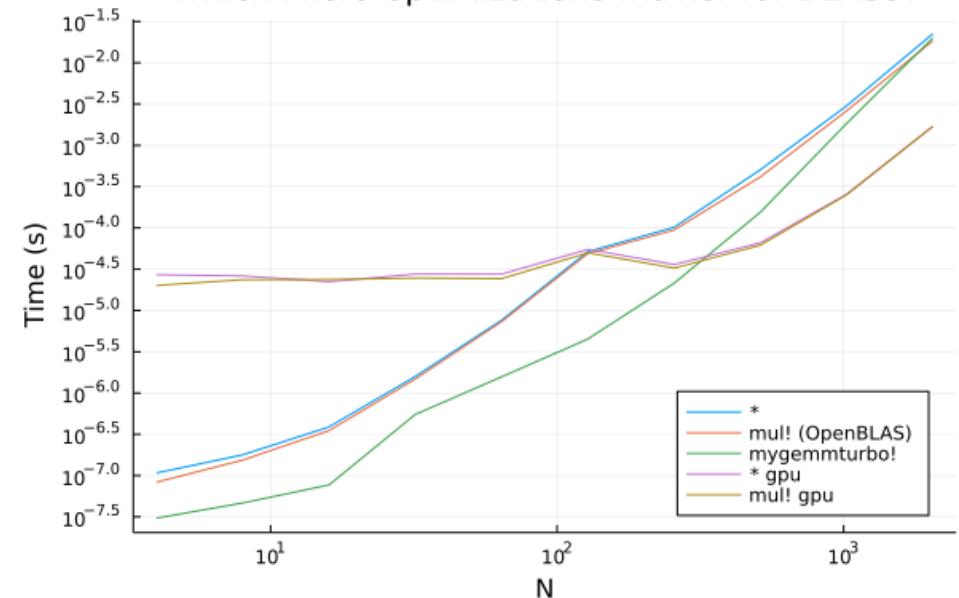


**Mutation and
Memory management: 10x**

Manual SIMD: 5x

...

Which Micro-optimizations matter for BLAS3?

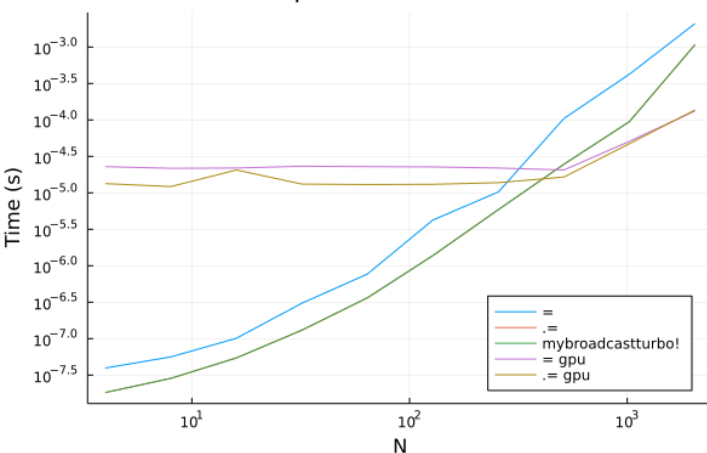


Big $O(n^3)$ operations?
Just use a GPU
Don't worry about overhead
You're fine!

Simplest code is $\sim 3x$ from optimized

What happens when you specialize computations?

Which Micro-optimizations matter for BLAS1?



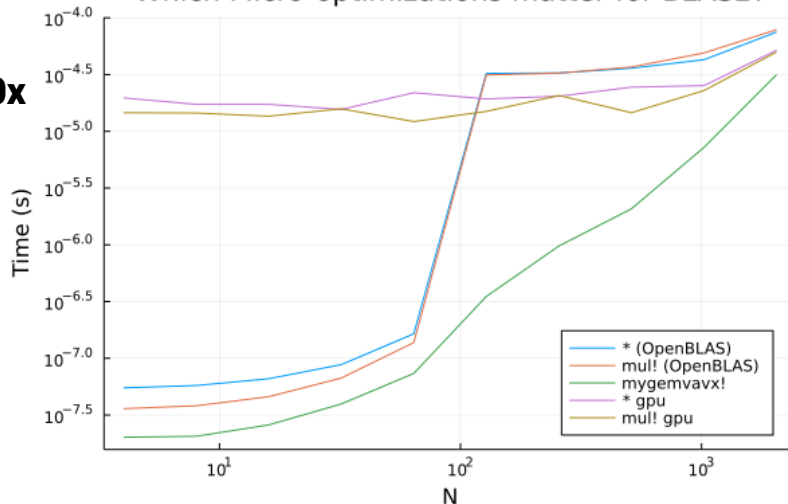
Scientific codes
 $O(n)$ and $O(n^2)$
operations

**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

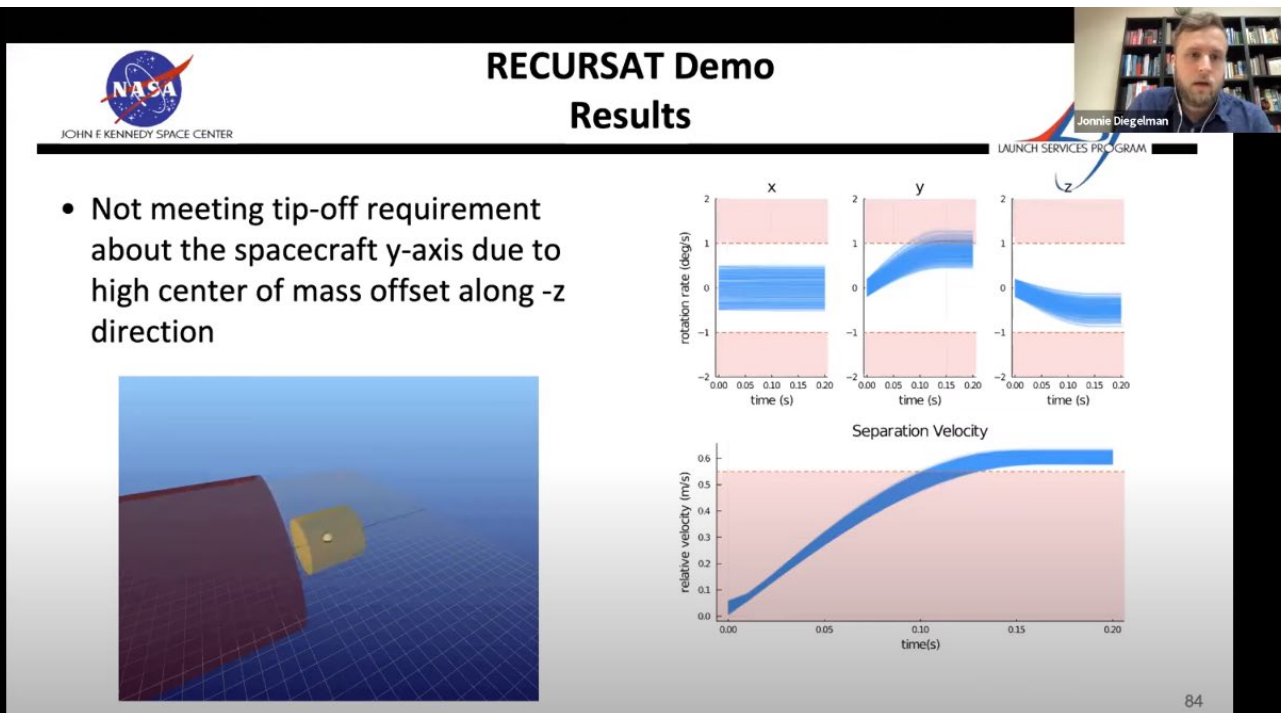
And 100 more libraries to mention...

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

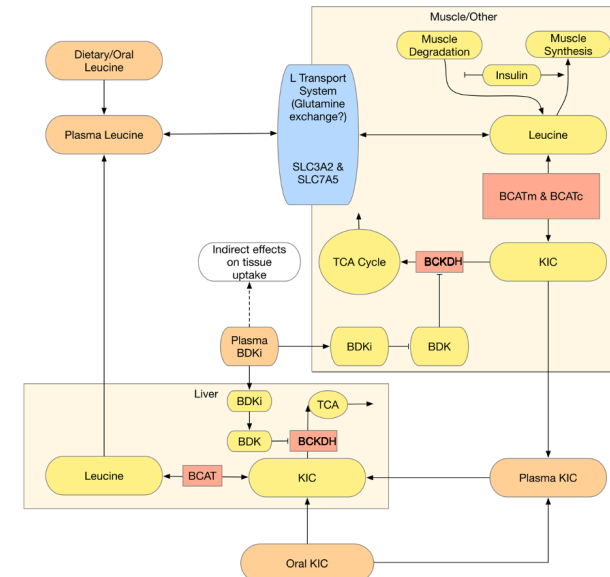


Modeling Spacecraft Separation Dynamics in Julia – SIAM CSE 2021
Jonathan Diegelman, NASA Launch Services Program and A.I. Solutions

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

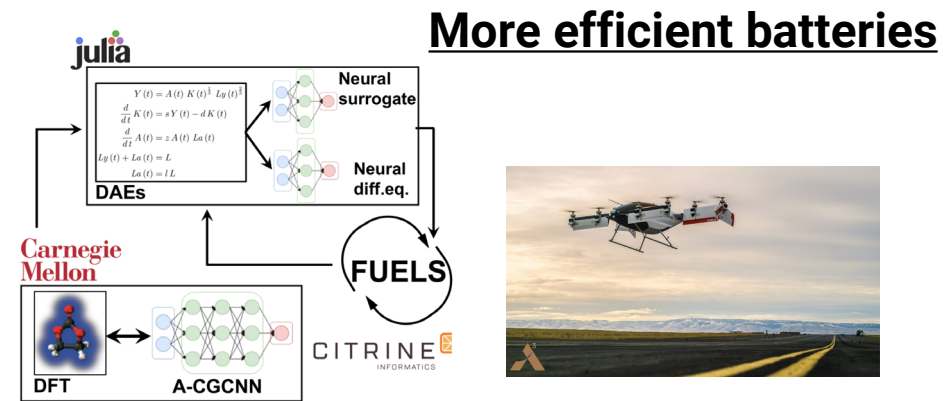


<https://juliacomputing.com/case-studies/pfizer/>

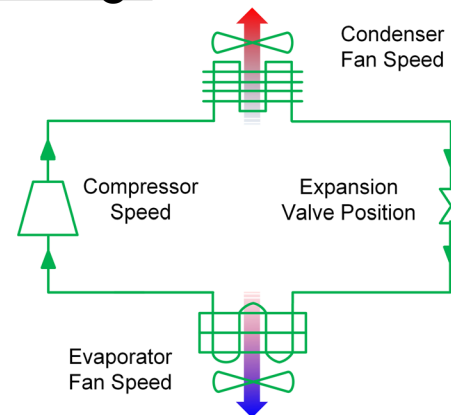
Conclusion

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

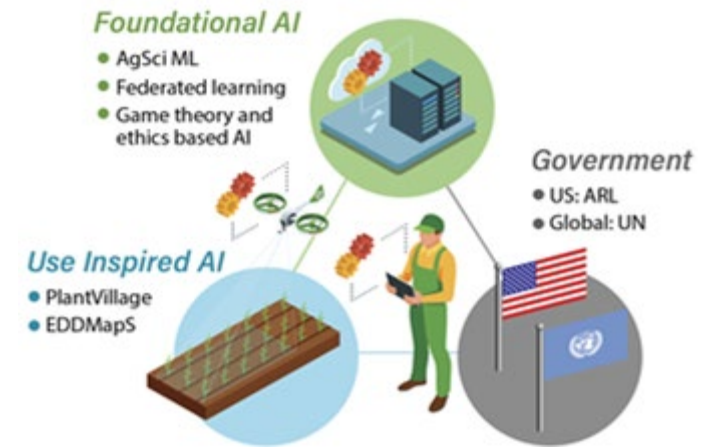
Faster Drug Development



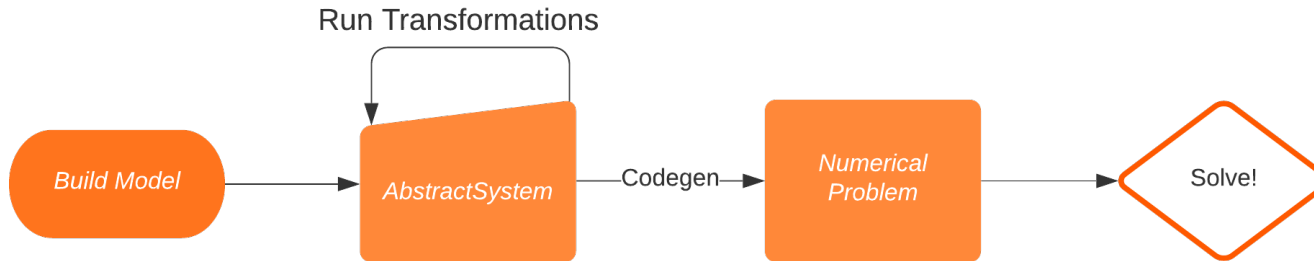
Energy Efficient Buildings



Climate modeling for improved agriculture



Machine Learning Surrogates as Approximate Transformations



If you build a machine learning method that outputs differential-algebraic equations, then it qualifies as an “approximate” stable transformation

- Take in a differential equation and the outputs to surrogate over
- Create a new differential equation system that is approximately the same input/output mapping (dimensionality reduction)
- Represent that system as an MTK model

Because it's approximate, it needs user-intervention.

We developed the continuous-time echo state network as a surrogate method which is robust to stiffness and has these properties.

```
using JuliaSim

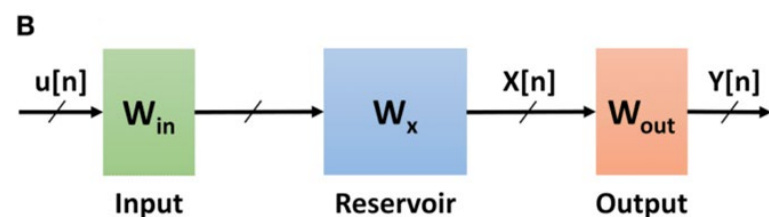
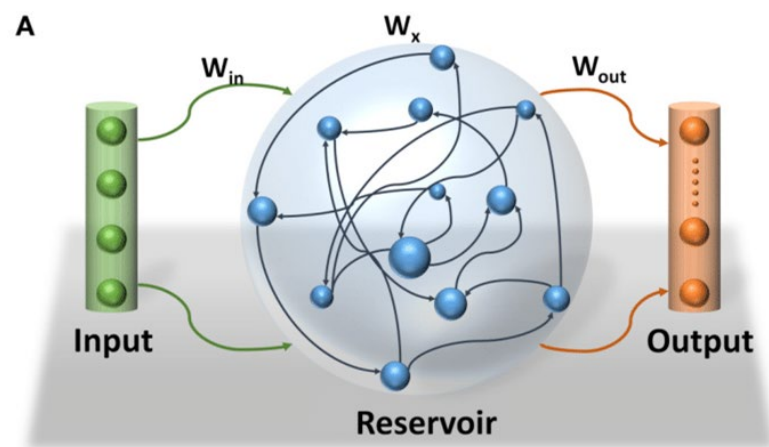
sys = ODESystem(...)
prob = ODEProblem(sys, u0, tspan, p)
param_space = [...]
surralg = LPCTESN(1000, output_function = (u,t) -> u[1:3])
sim = DEProblemSimulation(prob, reltol = 1e-12, abstol = 1e-12)

odesurrogate = JuliaSimSurrogates.surrogatize(
    sim,param_space,
    surralg,100 # n_sample_pts
)

newsys = ODESystem(odesurrogate)
```

Continuous-Time Echo State Networks: Avoid Gradients and Use an Implicit Fit

One way to visualize: reservoir computing
Fix a random dynamical process and find a projection to fit the system



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

Another interpretation: Semi-Neural ODE
Fix the parameters of the first layer and only train the last layer. By doing so, you can transform the training problem into a linear solve via SVD.

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

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

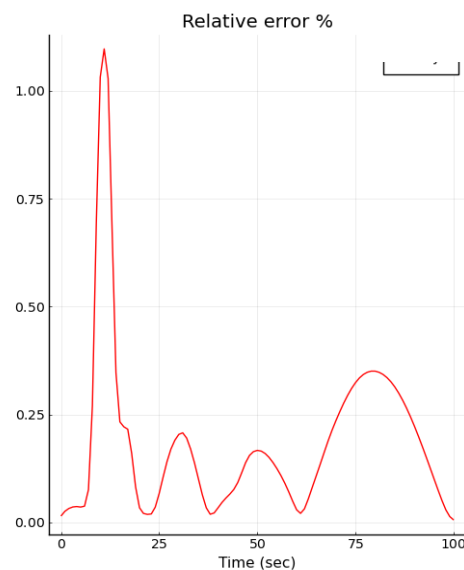
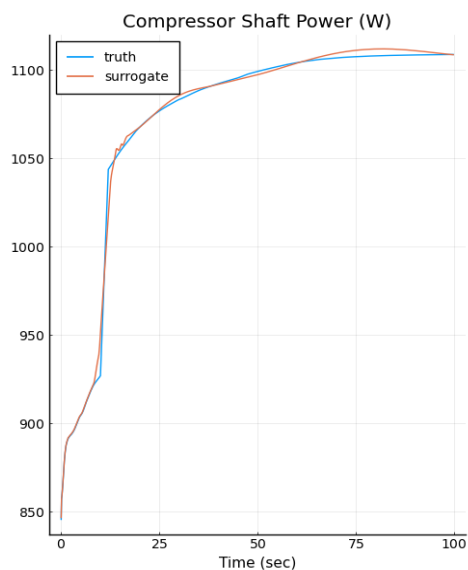
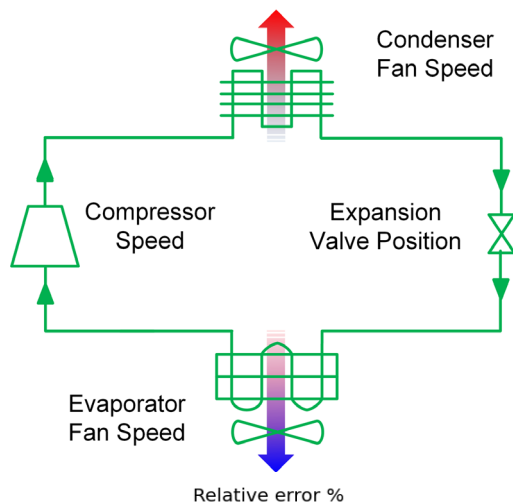
Predict behavior at new parameters via:

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

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

ARPA-E: Accelerated Simulation of Building Energy Efficiency

**8,000 ODE Highly stiff
vapor-compression
cycle model**



The Julia implementation is 6x faster than Dymola for the full cycle simulation.

- Dymola reference model: 35.3 s
- Julia (as close to) equivalent model: 5.8 s
- Could be due to details such as the linear solvers, the refrigerant property libraries, etc. More benchmarking to come.

Using CTESNs as surrogates improves simulation times between 10x-95x over the Julia baseline. Acceleration depends on the size of the reservoir in the CTESN. The surrogate approximates 20 of the observables.

Training set size	Reservoir size	Prediction time	Speedup over baseline
100	1000	0.06 s	95x
1000	2000	0.56 s	10x

Error is < 5% in all cases.

Total speedup over Dymola: 60-570x

Take Arbitrary Large Models and Automatically Accelerate with CTESNs

1265 ODE model of
spatial cell signaling in
Arabidopsis

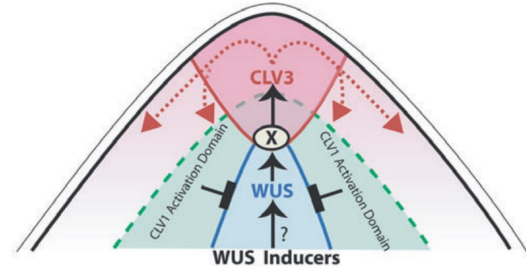
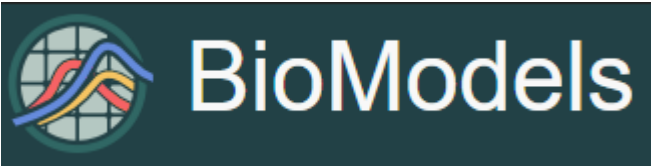


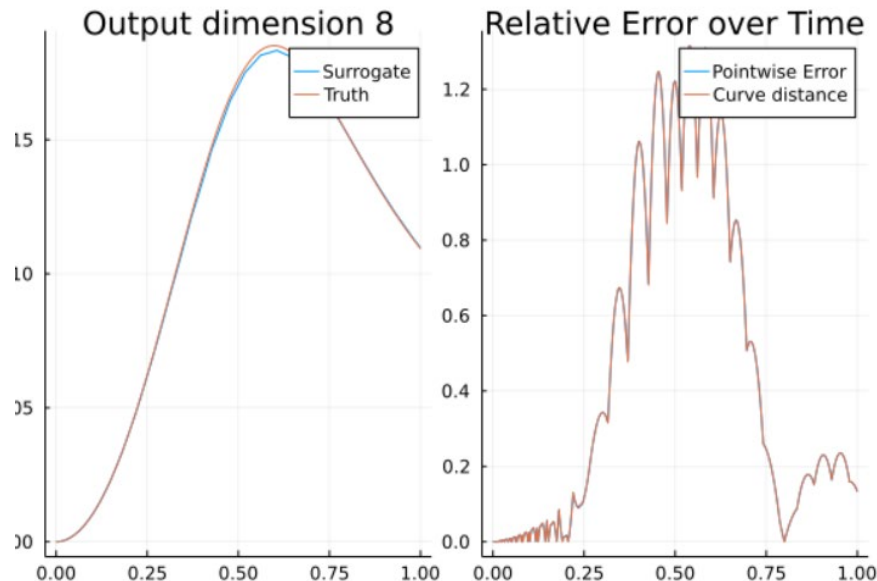
Fig. 1. A schematic of the expression domains of *CLAVATA1*, *CLAVATA3* and *WUSCHEL*. The solid arrows indicate the regulatory (indirect) interactions and the dashed arrows show the movement of the CLV3 protein.

- COPASI simulation: crashed upon reading (“not responding”)
- MATLAB SBMLToolbox: 870s to read, 1.13s to simulate
- Julia vanilla: 60s to read, 0.6s to simulate
- Julia surrogatized simulation: ~instant to read, 0.062s to simulate

Julia vanilla outperforms MATLAB’s SBMLToolbox

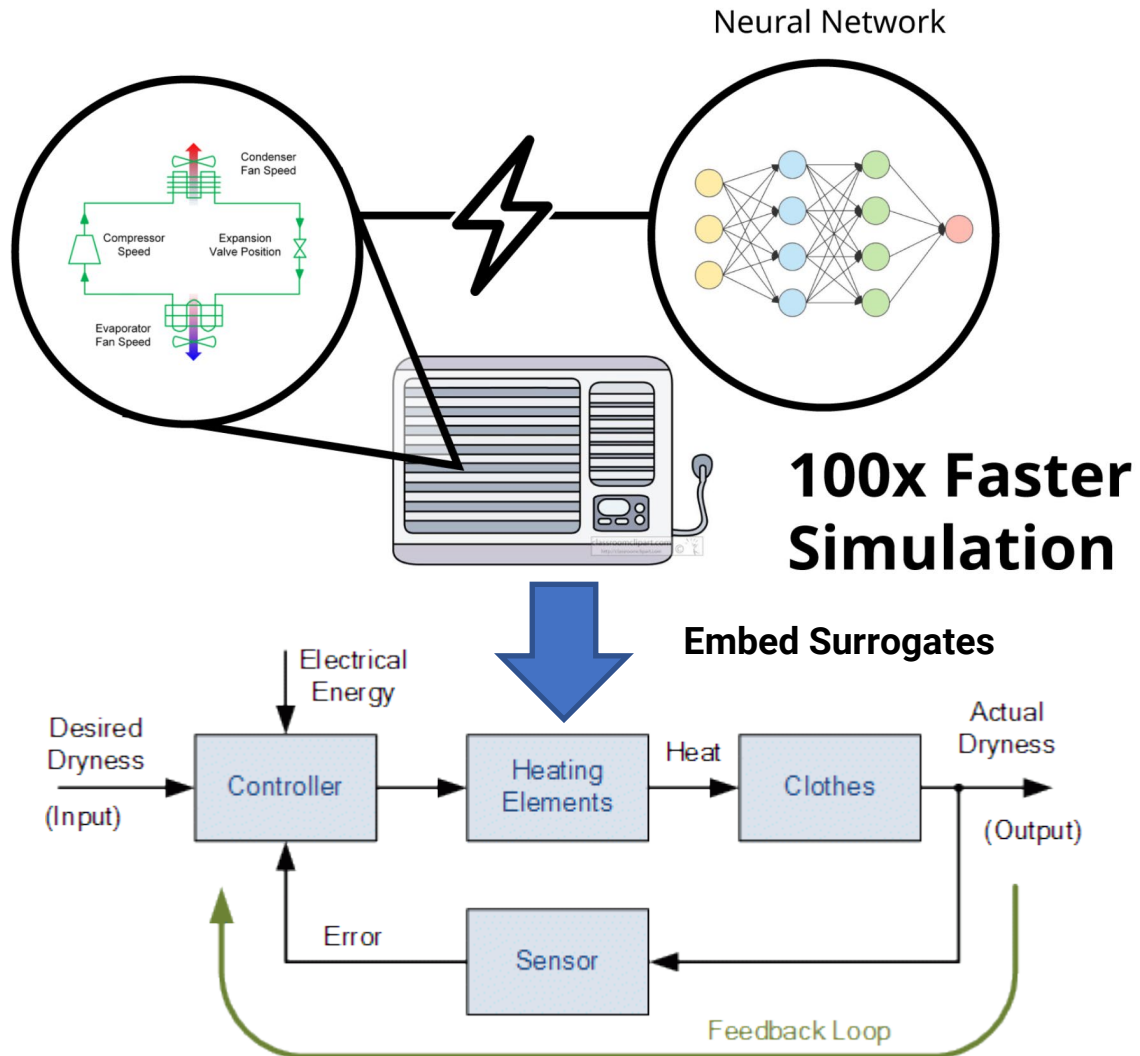
CTESN predictions at new parameters have < 5% error, are almost instant to read and 100x faster to simulate

(Julia SBML reader is incomplete: full Jacobians right now and no e-graph simplification. Probably ~10x performance left on the table)



Total speedup: 100x vs MATLAB SBMLToolbox

The Transformed Models are Just Components: Compose As Normal



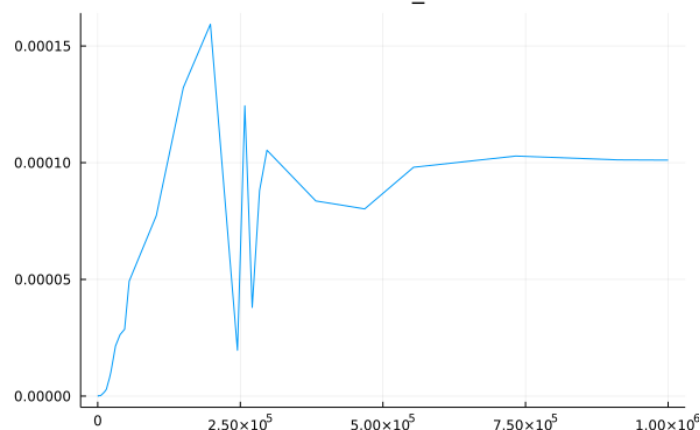
Accelerate large (100,000 ODE) simulations without retraining by using an accelerated HVAC component inside of different building models

Large Building Models 100K Equations, 80x Acceleration

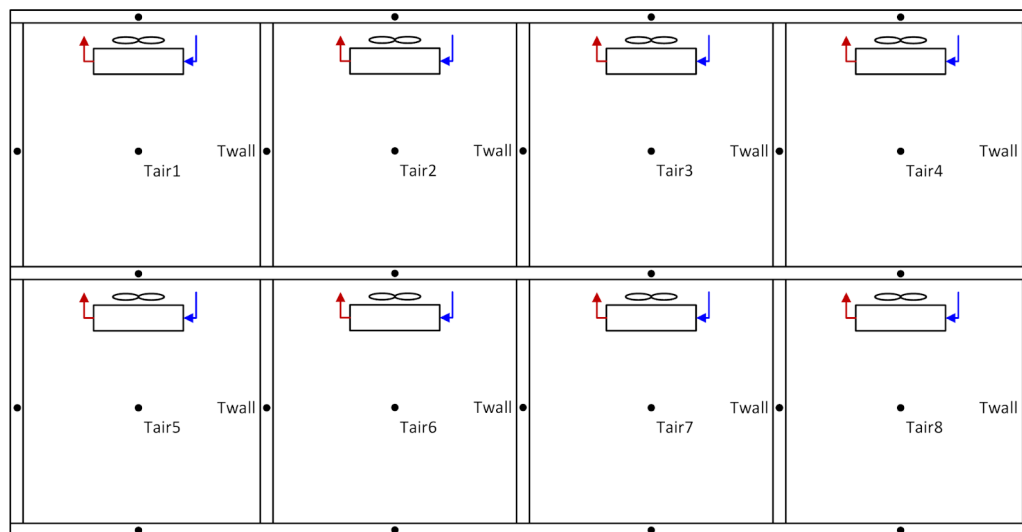
Room 33 Surrogate: T_{air} - Test Parameter



Relative Error Room 33: T_{air} - Test Parameter

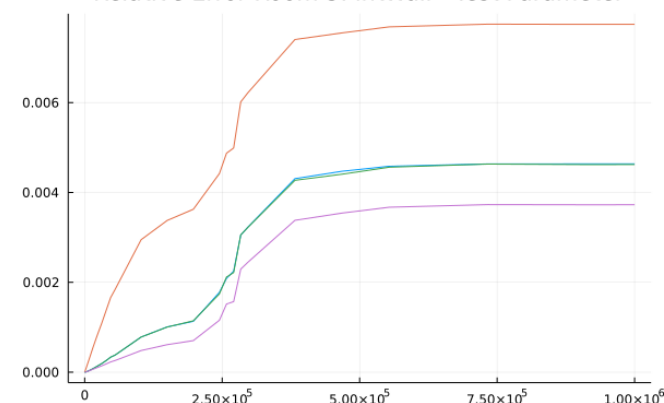


Rooms Disturbed	Training set size	Reservoir size	Prediction time	Speedup over baseline
1	100	200	0.2597 s	77x
3	100	200	0.413s	80x

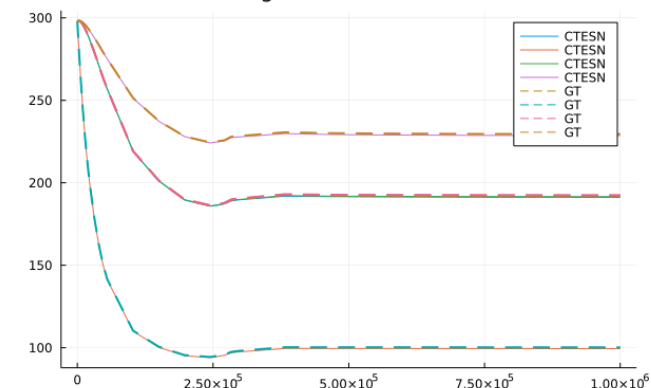


Scalable building model with equipment

Relative Error Room 5: intWall - Test Parameter

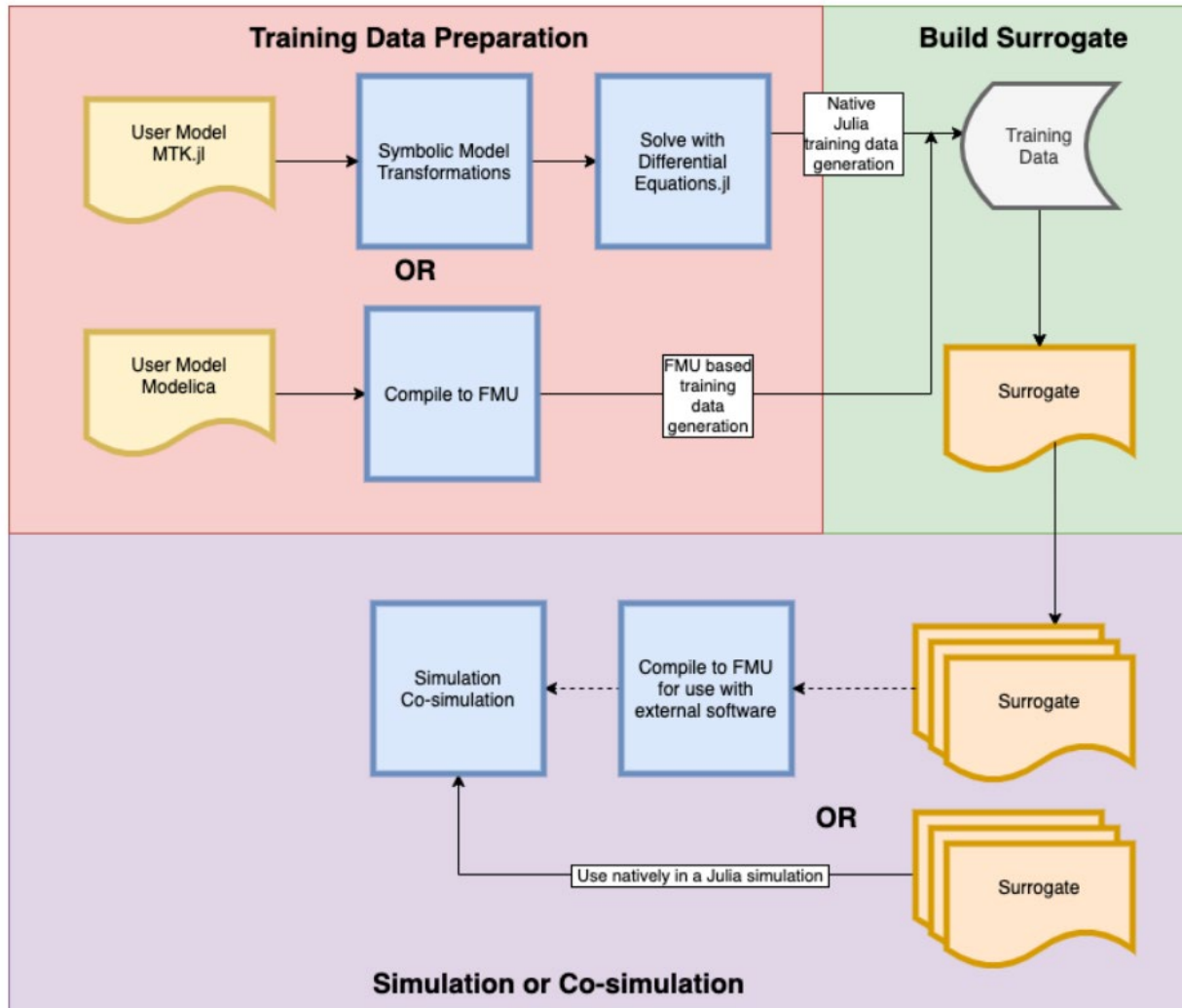


Room 5 Surrogate: intWall - Test Parameter



Total speedup over original : 80x

Surrogatization as Machine Learned Approximate Transformations



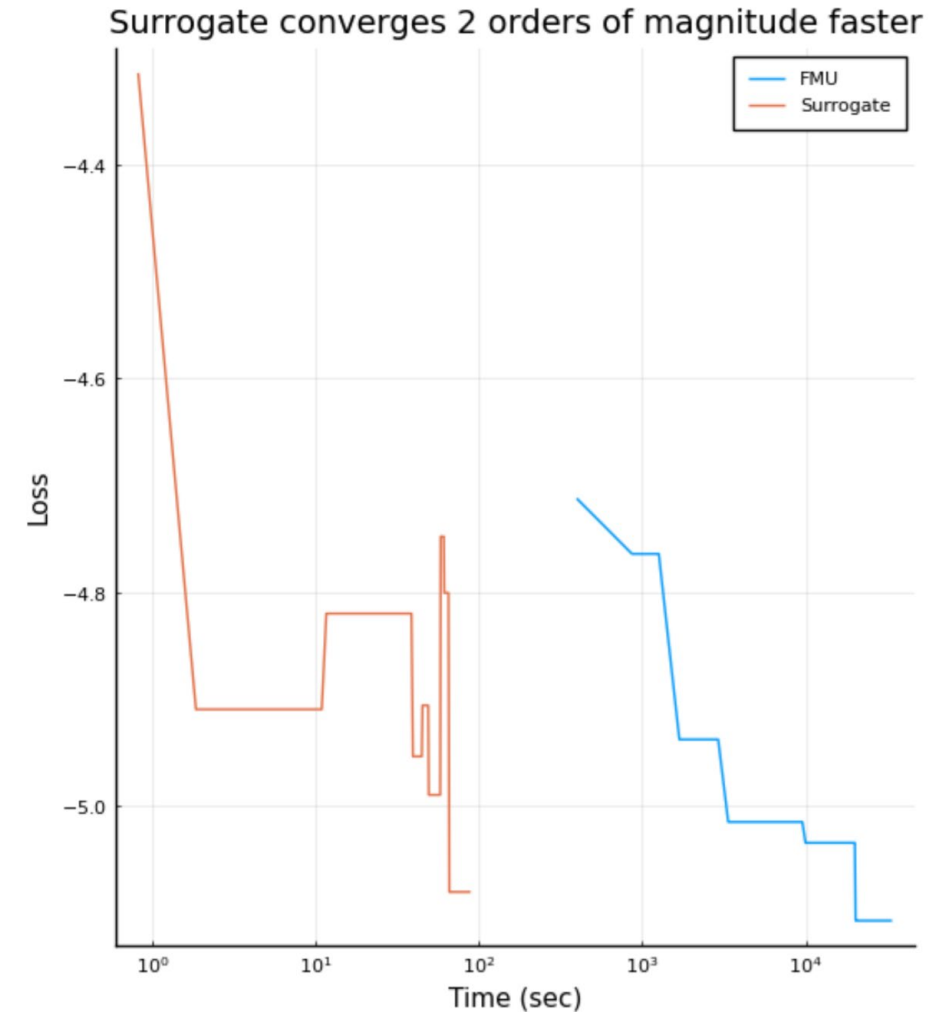
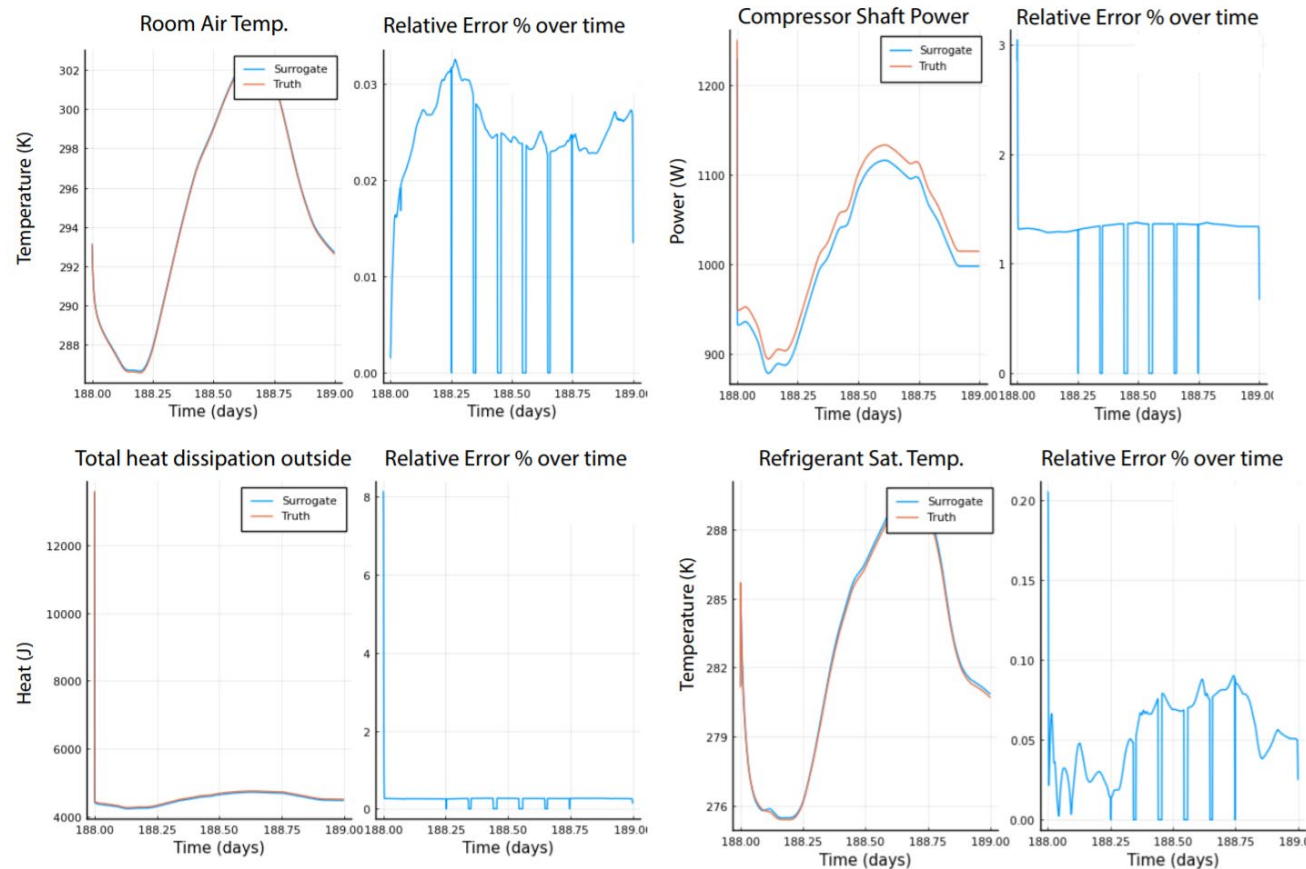
The training data source for a CTESN surrogate does not need to come from ModelingToolkit, it can come from any timeseries data source.

Training CTESNs on timeseries data sources gives a process that merges translation to ModelingToolkit with acceleration!

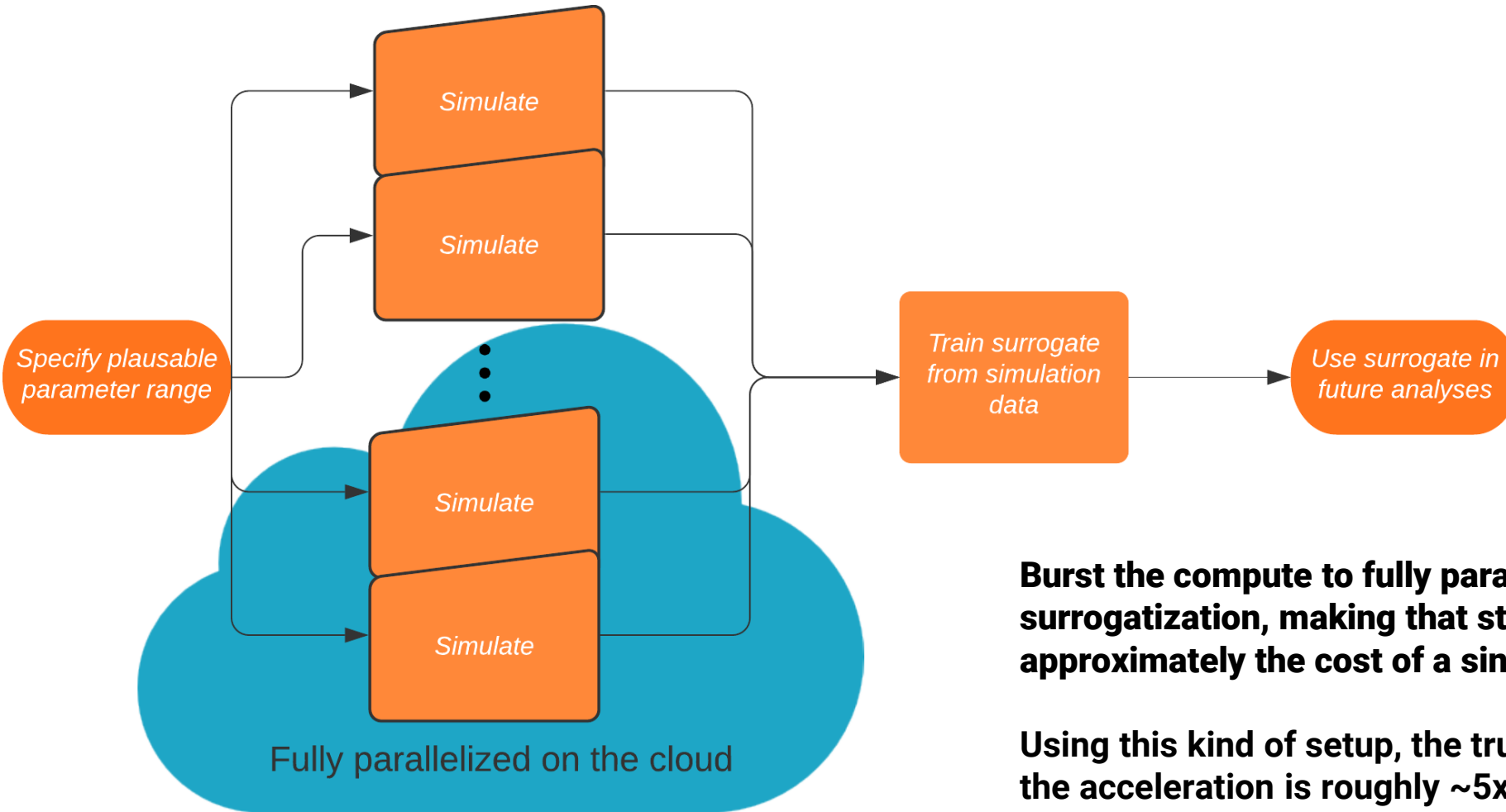
Sources that we have been experimenting with:

- **Functional Markup Units (FMUs) (Dymola, Simulink)**
- **SPICE models for electrical circuits (NgSpice, Xyce)**
- **Various PDE tools (COMSOL, Abaqus, etc.)**

340x Acceleration of a Global Optimization by Surrogatizing an FMU



Use Cloud Resources to Smartly Burst Compute and Amortize Time



Burst the compute to fully parallelize the simulations of the surrogatization, making that step of the process approximately the cost of a single simulation

Using this kind of setup, the true time cost to the user to run the acceleration is roughly $\sim 5x-10x^*$ the simulation time

This Process Can Be Bundled Up As an FMU->FMU Accelerator

The screenshot shows the JuliaSim FMU Surrogates web interface. At the top, there is a dark blue header with the logo and text 'JuliaSim FMU Surrogates' on the left, and links for 'Library', 'Julia', 'JuliaHub', and 'Contact Us' on the right. Below the header is a large, dark blue rectangular box with a white upload icon and the text 'Upload FMU' and 'Supported files:fmu'. Underneath this box are three buttons: 'LOAD', 'EXPORT', and 'RESET'. Below the buttons is a section for configuration options. The first option is 'Algorithm', which has a dropdown menu currently showing 'Algorithm *'. The second option is 'Reservoir Size', which has a slider ranging from 1 to 5001, with a blue dot indicating the current value is near 1. The third option is 'Number of Sample Points', which has a slider ranging from 1 to 1001, with a blue dot indicating the current value is near 1. The fourth option, 'Time Span', is partially visible at the bottom of the interface.

By moving the model transformation process to the runtime itself, ModelingToolkit can be used as a transformation and compilation system by other front ends.

Other talks at the Modelica conference also exploit this feature.

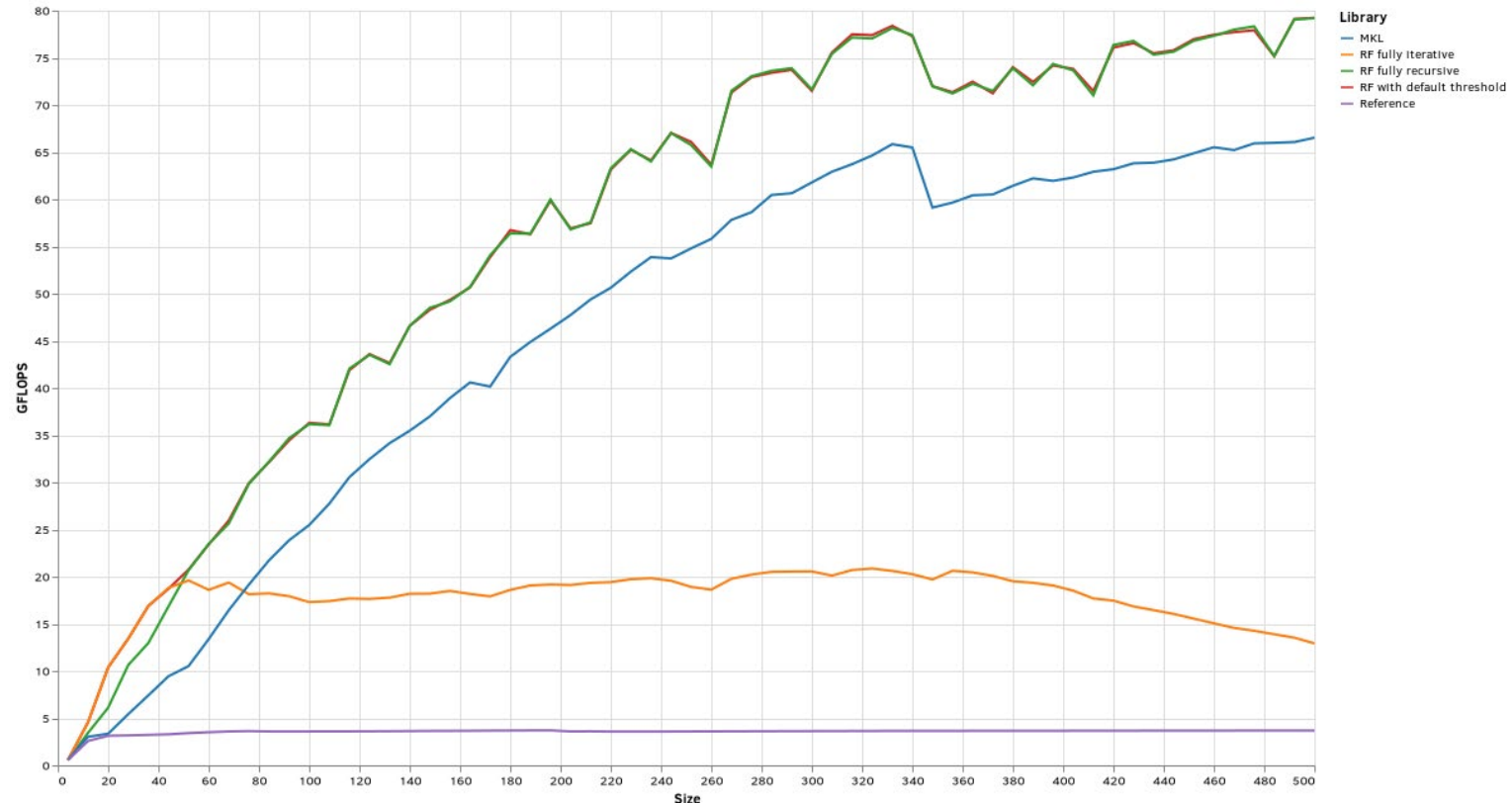
One Part of Performance: Improved Linear Solvers

**New numerical linear algebra stack,
designed for ML, outperforms MKL on
modern high-end AMD CPUs**

**Roadmap to Julia BLAS and
LinearAlgebra | Chris Elrod |
JuliaCon2021**

Size: (17, 17)
RecursiveFactorization: MedianGFLOPS = 3.053
MKL: MedianGFLOPS = 2.047
OpenBLAS: MedianGFLOPS = 2.509

Size: (486, 486)
RecursiveFactorization: MedianGFLOPS = 61.48
MKL: MedianGFLOPS = 44.45
OpenBLAS: MedianGFLOPS = 30.56



<https://github.com/YingboMa/RecursiveFactorization.jl/pull/28>

The Limitations of Tracing Representations: Quasi-Static Models

Representable, 8x

```
function f(x)
  for i in 1:3
    x = 2 * x
  end
  return x
end
```

Not Representable

```
function f(x)
  if x < 3:
    return 3 * x^2
  else
    return -4 * x
  end
end
```

Representable

```
f(x) = IfElse.ifelse(x<3,3*x^2,-4*x)
```

Quasi-Static:

Is the computation input
value independent?

ML model representations in Jax,
Tensorflow, etc. use quasi-static
representations for model optimizations.

Not representable, not quasi-static

```
function factorial(x)
  out = x
  while x > 1
    x -= 1
    out *= x
  end
  out
end
```

[Useful Algorithms That Are Not Optimized By Jax, PyTorch, or
Tensorflow](#)

Stochasticlifestyle.com

Final Note: Using Compilers and Transformations Beyond Differentiation

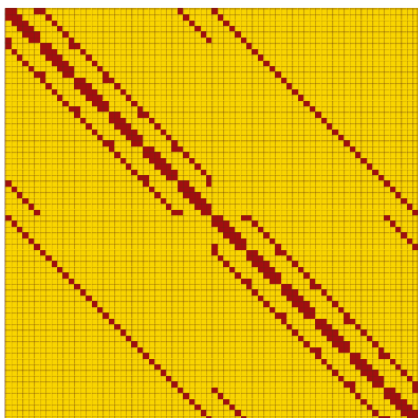


Figure 1: Sparsity pattern of the Jacobian of the Brusselator code in Listing 1 with input and output tensors of size $6 \times 6 \times 2 = 72$.

Automatic Sparsity Detection

Table 1: Hessian sparsity construction for a program taking as input a vector of length 4. The 4×4 sparsity pattern for each intermediate value is shown. The provenance polynomial has the same hessian sparsity pattern.

code fragment	polynomial	sparsity
<code>deg2rad(x[1])</code>	x_1	
<code>log(x[1])</code>	x_1^2	
<code>x[1] + x[4]</code>	$x_1 + x_4$	
<code>x[1] * x[4]</code>	$x_1 x_4$	
<code>q = x[1]/x[4]</code>	$x_1 x_4^2$	
<code>asin(q)*x[3]</code>	$(x_1^2 x_4^2) x_3$	

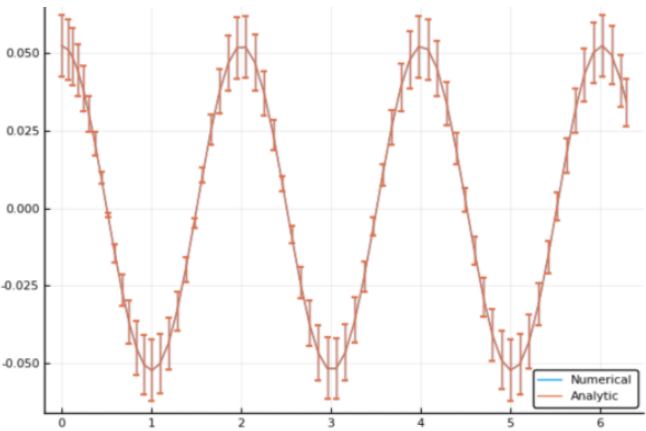
```
g = 9.79 ± 0.02 # Gravitational constants
L = 1.00 ± 0.01 # Length of the pendulum

# Initial speed & angle, time span
u_o = [0 ± 0, π/60 ± 0.01]
tspan = (0.0, 6.3)

# Define the problem
function pendulum(du, u, p, t)
    θ = u[1]
    dθ = u[2]
    du[1] = dθ
    du[2] = -(g/L)*θ
end

# Pass to solvers
prop = ODEProblem(pendulum, u_o, tspan)
sol = solve(prop, Tsit5(), reltol = 1e-6)

# Analytic solution
u = u_o[2] .* cos.(sqrt(g/L) .* sol.t)
```



Rackauckas et al. *DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia*. 2017. ([Journal of Open Research Software](#))

Giordano. *Uncertainty propagation with functionally correlated quantities* ([arXiv.1610.08716](#))

Compiler-Based Intrusive Uncertainty Quantification

[Generalizing Automatic Differentiation to Automatic Sparsity, Uncertainty, Stability, and Parallelism](#), [StochasticLifestyle.com](#)

Julia's Modeling and Simulation Advantage Extends to the Symbolic Realm



ModelingToolkit, Modelica, and Modia: The Composable Modeling Future in Julia

Chris Rackauckas



Fredrik Bagge Carlson 8:02 PM

Started out a simple computation using SymPy today and noticed that it took a while. Spent the time I was waiting for SymPy to return to read the docs for Symbolics.jl, reimplemented the same thing using Symbolics and 20 minutes later I had the answer, before SymPy was done. The same computation using Symbolics.jl took

```
1.552 ms (10755 allocations: 424.75 KiB)
```

(admittedly, after the time-to-first simplify of about 5 seconds 😊)
Symbolics.jl may be a bit rough around the edges, but sure looks promising!

2370x speedup over SymPy on real-world robotics application

<https://github.com/JuliaSymbolics/SymbolicUtils.jl/pull/254>

Symbolics:

```
julia> x = setup(vis, 3); @time M = mass_matrix(x);
3.721925 seconds (7.73 M allocations: 462.482 MiB, 2.41% gc time, 99.20% compilation time)

julia> for n in 3:7; x = setup(vis, n); @time M = mass_matrix(x); end
0.006976 seconds (79.61 k allocations: 3.028 MiB)
0.007024 seconds (80.71 k allocations: 3.072 MiB)
0.007367 seconds (81.56 k allocations: 3.106 MiB)
0.007341 seconds (82.07 k allocations: 3.131 MiB)
0.007315 seconds (82.66 k allocations: 3.157 MiB)
```

SymPy:

```
julia> x = setup(vis, 3, true); @time M = mass_matrix(x);
16.333147 seconds (31.71 M allocations: 1.870 GiB, 2.18% gc time)

julia> for n in 3:7; x = setup(vis, n, true); @time M = mass_matrix(x); end
2.899745 seconds (78.61 k allocations: 2.414 MiB)
4.737822 seconds (78.61 k allocations: 2.414 MiB)
9.462235 seconds (78.61 k allocations: 2.414 MiB)
16.858959 seconds (78.61 k allocations: 2.414 MiB)
17.311054 seconds (78.61 k allocations: 2.414 MiB)
```

Reason for Julia's Advantage? Engineering a Community

SciML's Common Interface:

- **One consistent interface for all numerics**
- **Symbolic modeling for all forms**
- **Automated inverse problems and adjoints**
- **Composes across the whole package ecosystem**
- **Fully embraces generic programming**
- **Uses and embraces the work of other developers**

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

Youtube: Differential Equations in 2021

Every improvement by every package developer feeds into one pipeline

The SciML Common Interface, Oversimplified

