# Differentiable Simulation, Neural ODEs, and Universal ODEs: The Real Bits

# What can go wrong, and what to do about it.

**Chris Rackauckas**

VP of Modeling and Simulation,
*Julia Computing*

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

Director of Scientific Research,
*Pumas-AI*

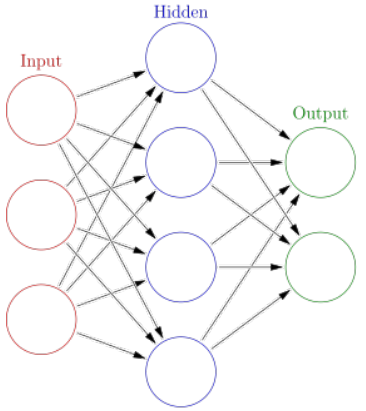
**Outline: Differentiable Simulation requires more than just sticking automatic differentiation on a simulator.**

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

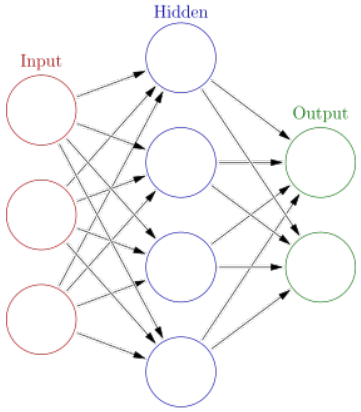
# Prologue: Why do Differentiable Simulation?

# Universal (Approximator) Differential Equations

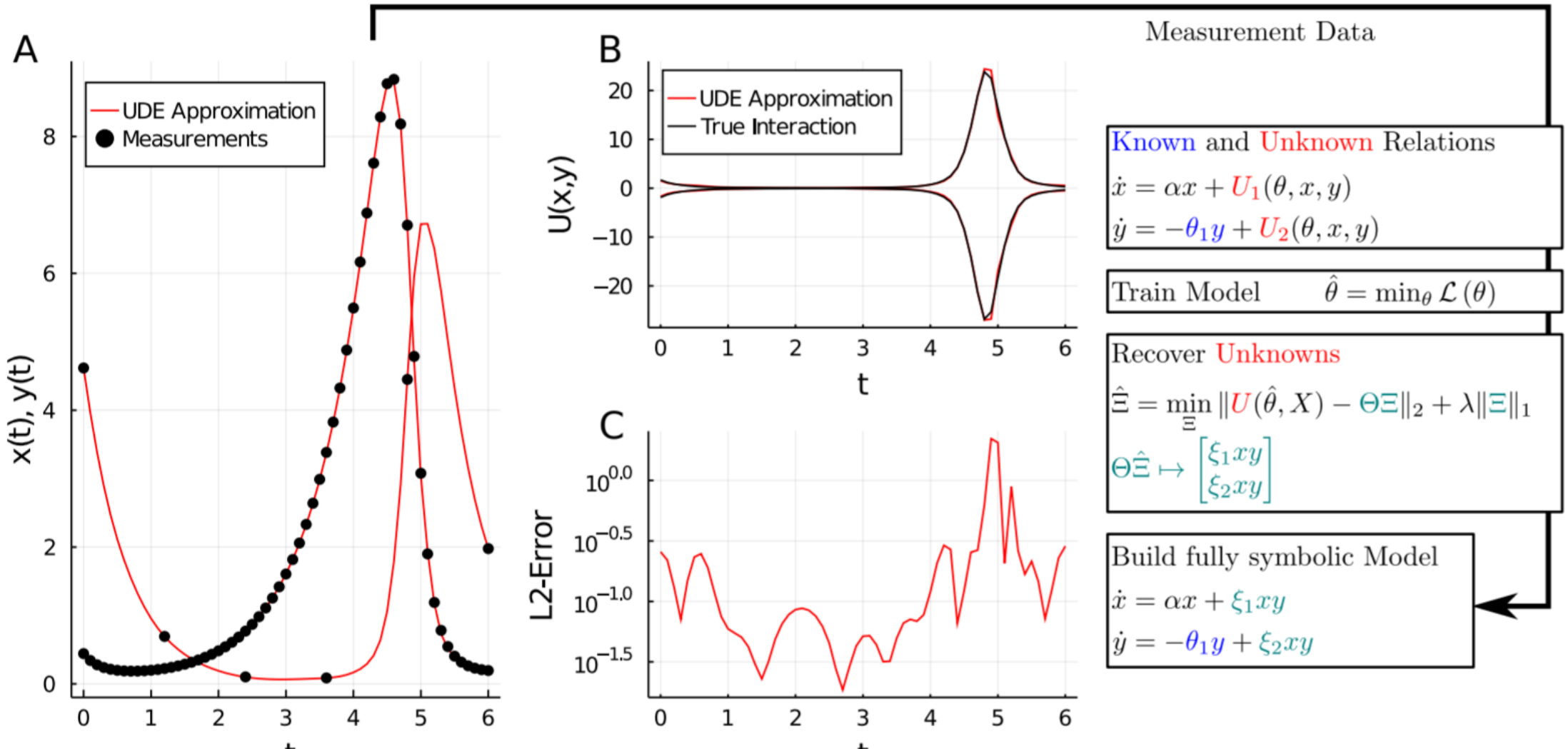
$$u' = f(u, \quad , t)$$




$$x' = \alpha x \quad +$$
$$y' = -\beta y +$$



# Universal (Approximator) Differential Equations



# UODEs show accurate extrapolation and generalization

Run the code yourself!

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

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

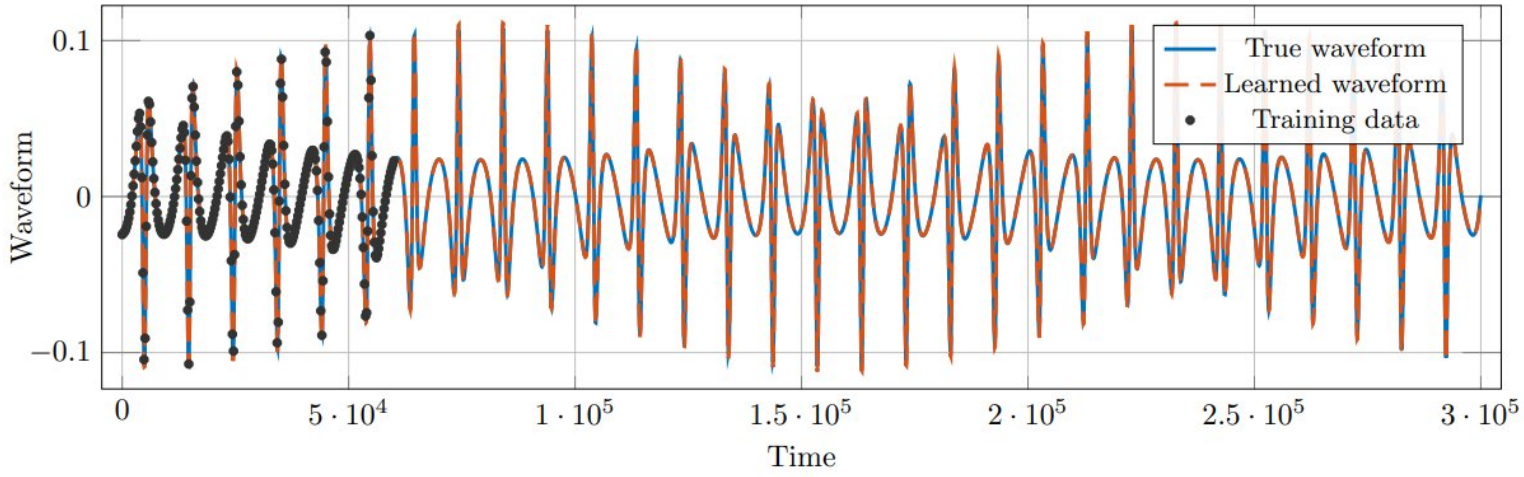
$$\dot{\phi} = \frac{(1 + e\cos(\chi))^2}{Mp^{3/2}}\big(1 + \mathcal{F}_1(\cos(\chi), p, e)\big), \tag{5a}$$

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

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

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

Example using binary black hold dynamics with LIGO gravitational wave data




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

# SciML Shows how to build Earthquake-Safe Buildings

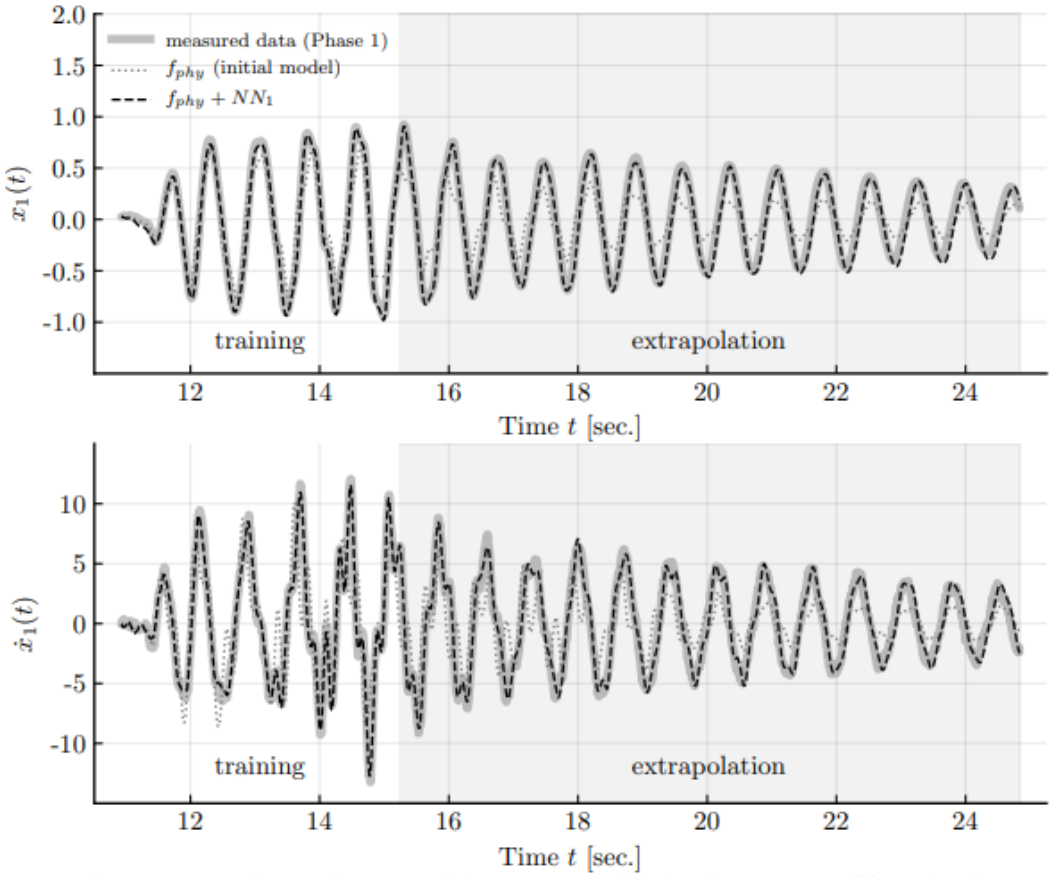



Figure 12: Comparison of time history of the response for displacement $x_1(t)$ and velocity $\dot{x}_1(t)$ for the NSD experiment (Phase 1).

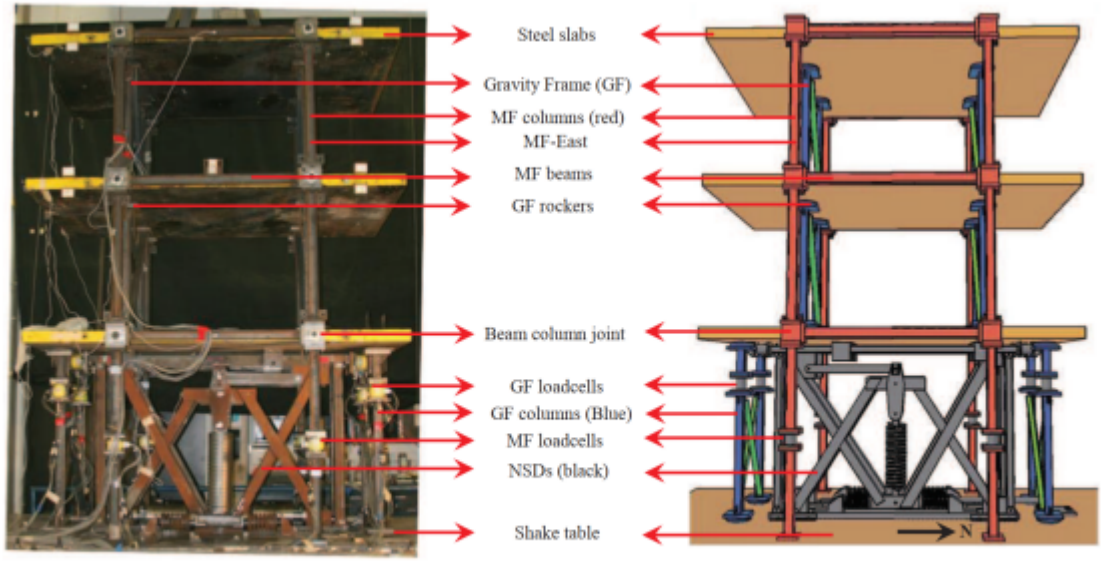



Figure 10: The structural system equipped with a negative stiffness device in between the first floor and the shake table.

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

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

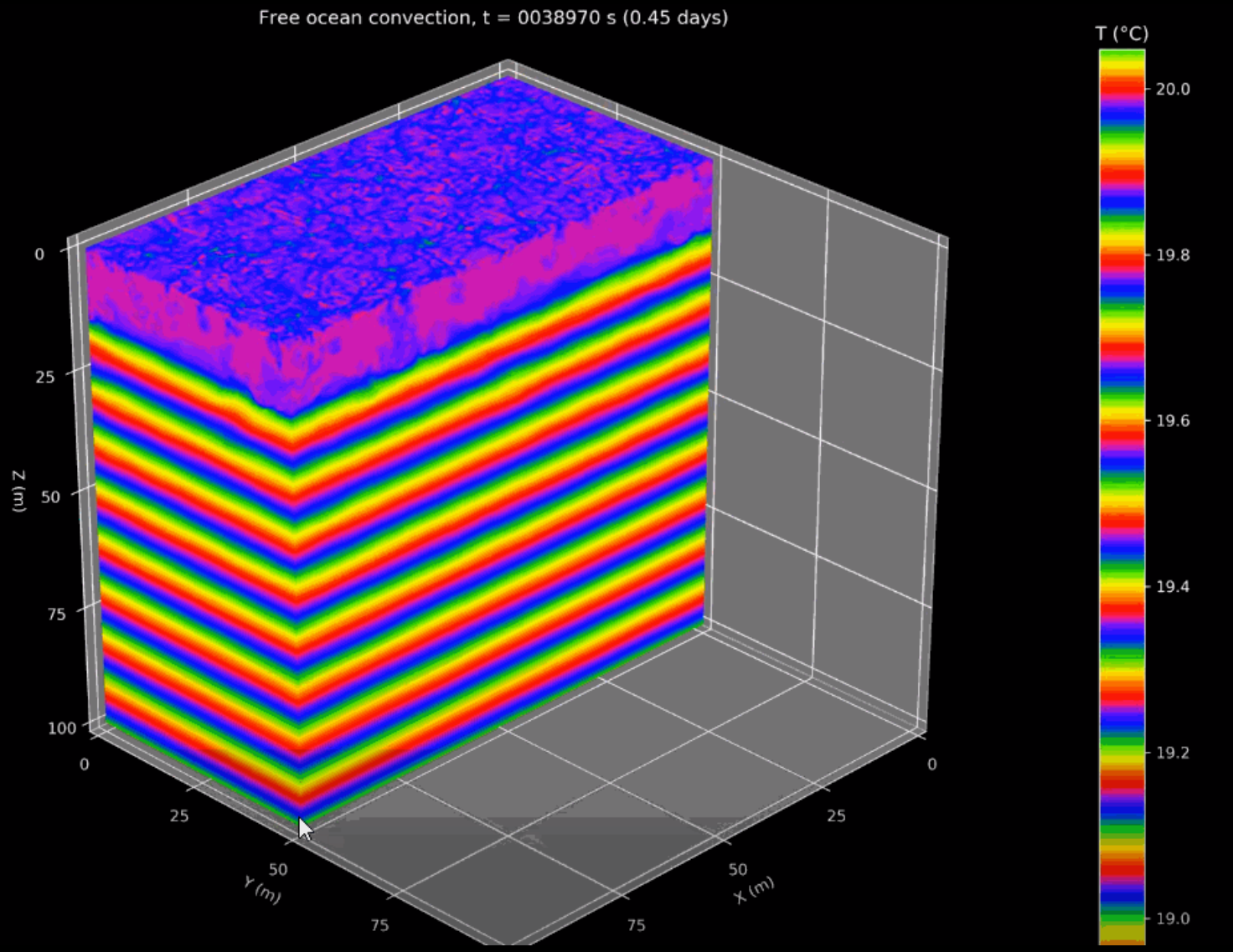
**Chris Rackauckas: Accurate and Efficient Physics-Informed Learning Through Differentiable Simulation**

# Does doing such methods require differentiation of the simulator?

# High fidelity surrogates of ocean columns for climate models

3D simulations are high resolution but too expensive.

**Can we learn faster models?**



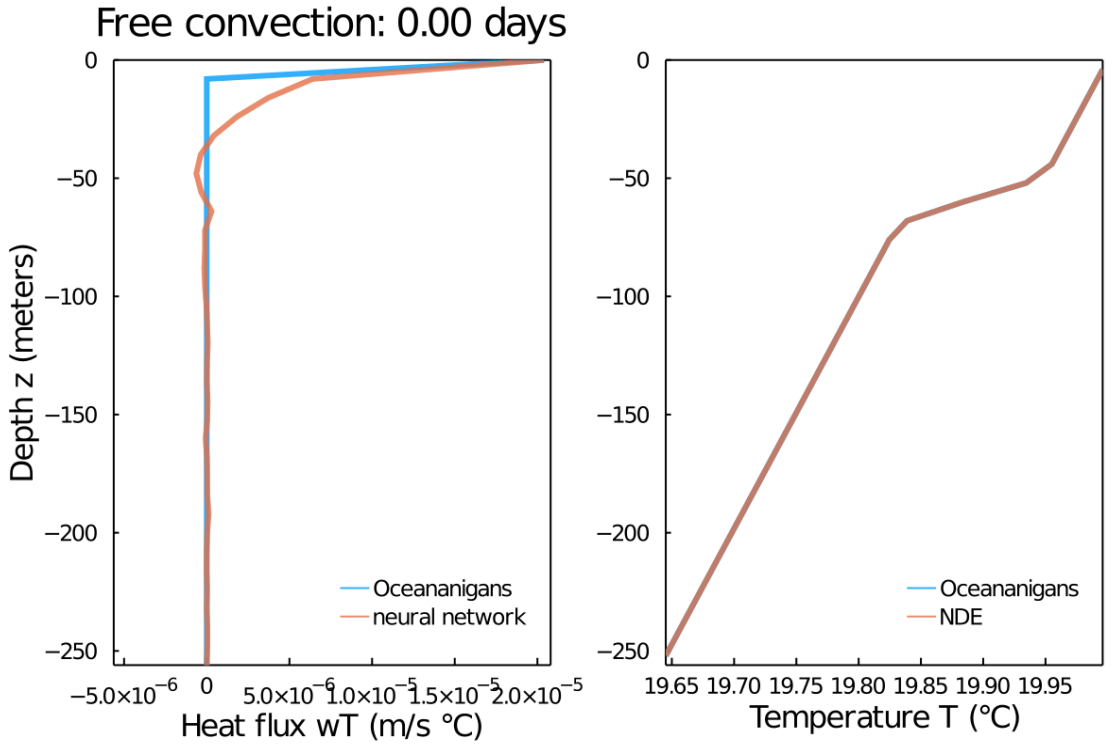
# Neural Networks Infused into Known Partial Differential Equations

## Derive a 1D approximation to the 3D model

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

## Incorporate the "convective adjustment"

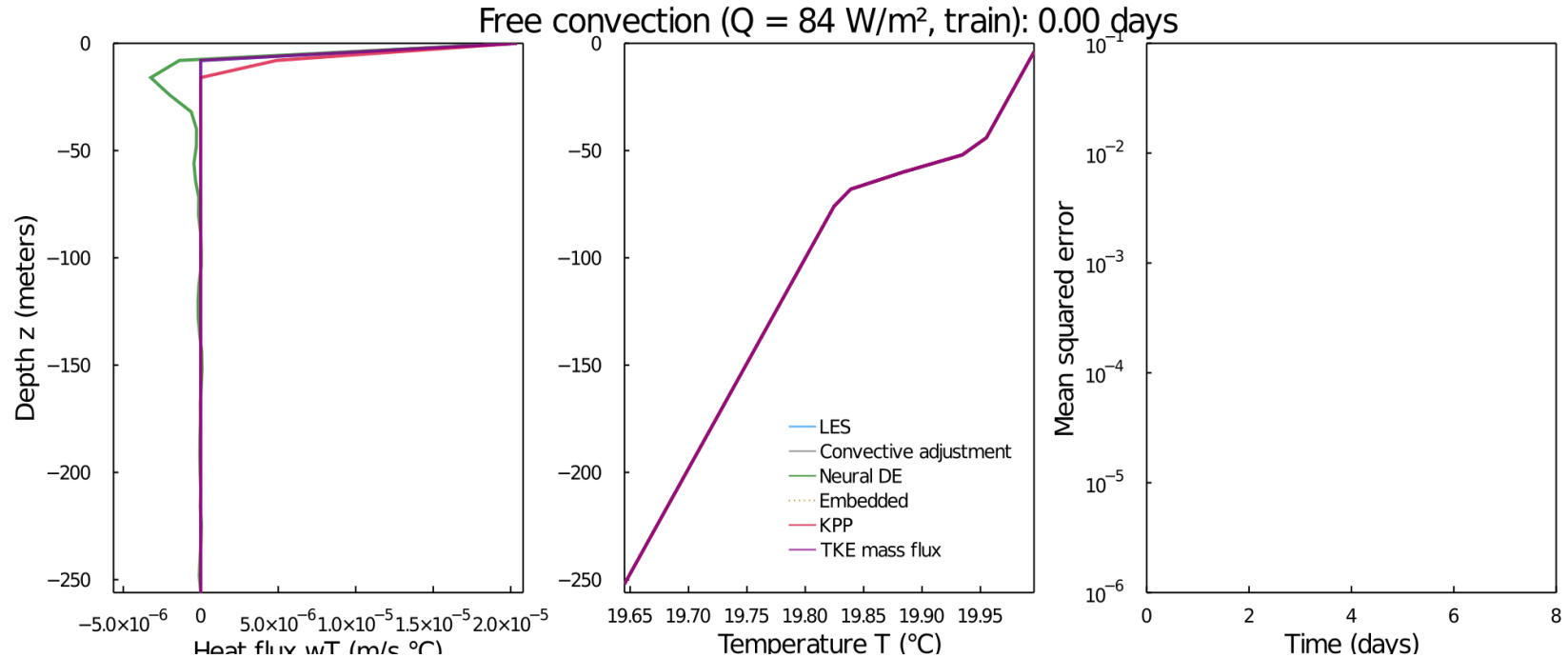
$$K = \begin{cases} 0 & \text{if } \partial_z T > 0 \\ 100\ \text{m}^2/\text{s} & \text{if } \partial_z T < 0 \end{cases}$$




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

**Only okay, but why?**

# Good Engineering Principles: Integral Control!



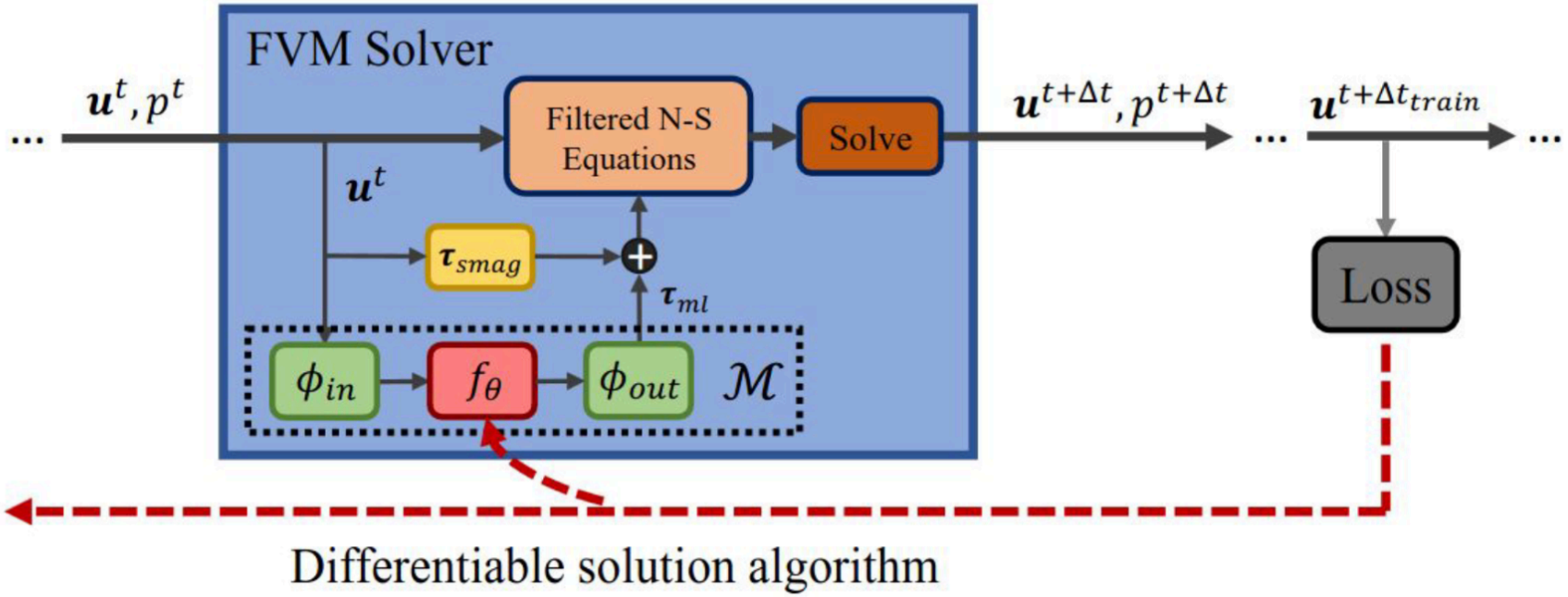

$$\frac{\partial T}{\partial t} = -\frac{\partial}{\partial z}\underbrace{\left(\text{NN} - K\frac{\partial T}{\partial z}\right)}_{\overline{w'T'}}$$

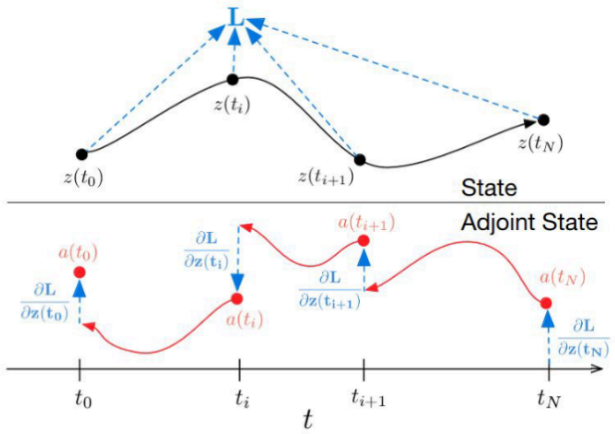
$$loss(T_{NN}, T) = |T_{NN}(z,t) - T(z,t)|^2$$

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

# Part 1: Differentiation of Solvers

# You saw this earlier this week



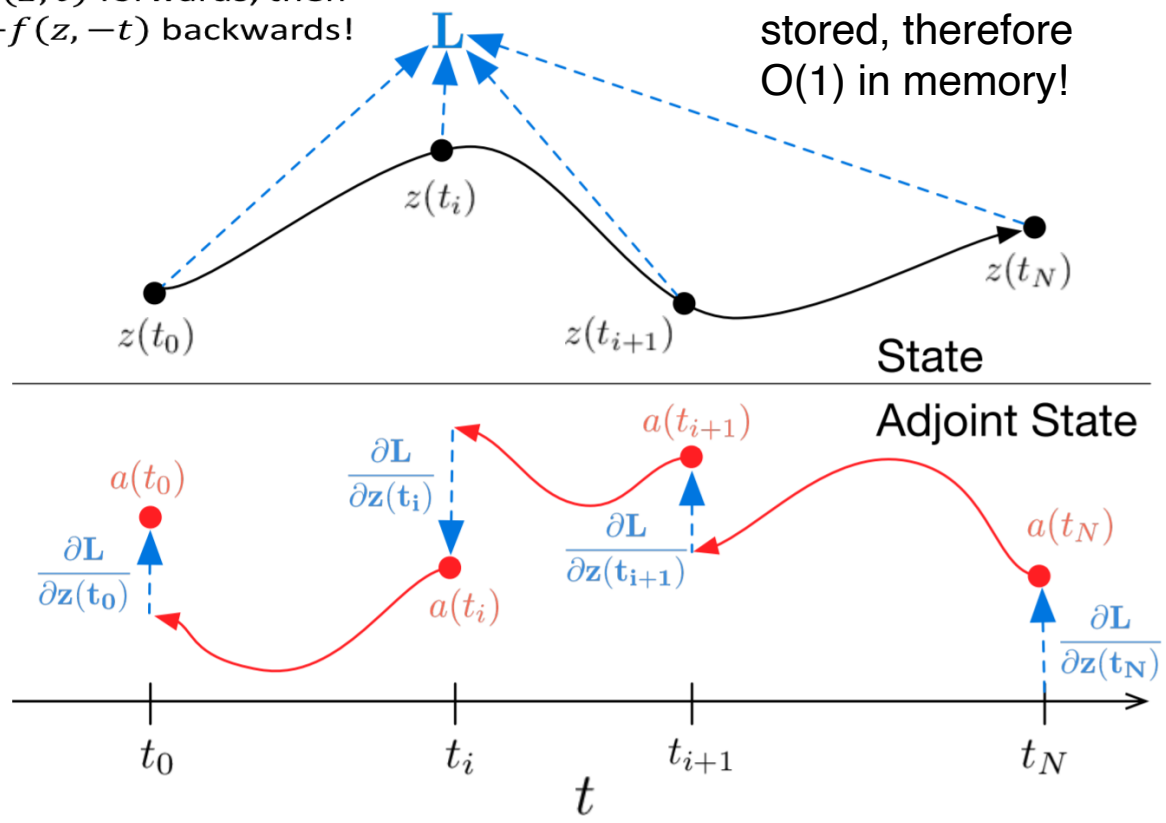




Goal:
Leverage rich developments in adjoint-based techniques for numerical solutions of partial differential equations.

# Machine Learning Neural Ordinary Differential Equations

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

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




The adjoint equation is an ODE!

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

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

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


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

# But… really?

# Differentiating Ordinary Differential Equations: The Trick

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

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

To derive this adjoint, introduce the Lagrange multiplier $\lambda$ to form:

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

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

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

# Differentiating Ordinary Differential Equations: Integration By Parts

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

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

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

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

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

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

# Differentiating Ordinary Differential Equations: The Final Form

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

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

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

$$\lambda(T) = 0$$

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

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

# Differentiating Ordinary Differential Equations: Summary

**Summary:**

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

$$\lambda(T) = 0$$

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

# Differentiating Ordinary Differential Equations: Step 2 Details

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

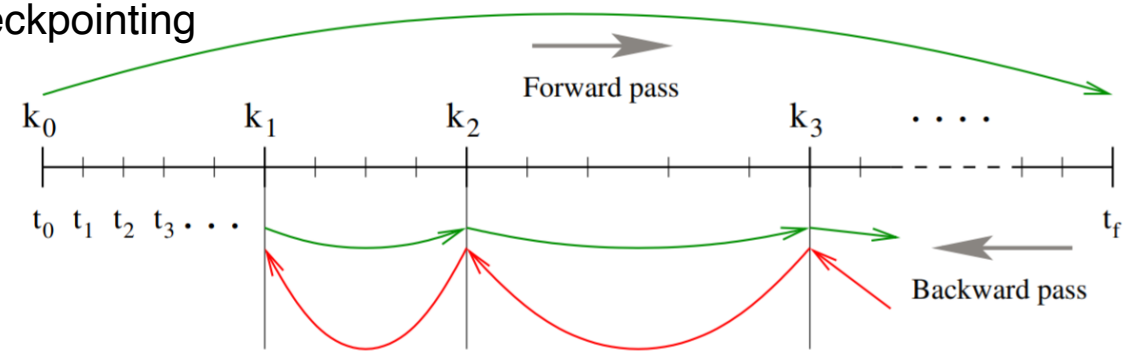
$\lambda(T) = 0$

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

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

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

3. Checkpointing



# How the gradient (adjoint) is calculated also matters!

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

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

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

## 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?**

# Let’s try this on Burger’s!

# Oh no! Oh no! Oh no no no no no!

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

Advection Equation:

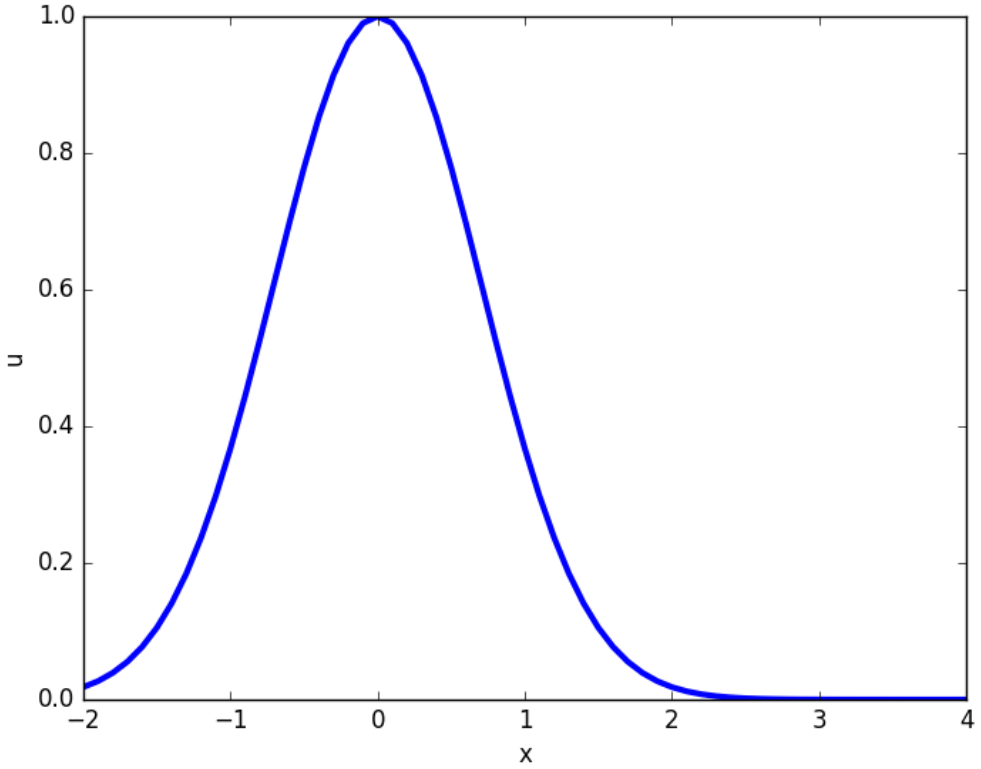
$$\frac{\partial u}{\partial t} + \frac{a(\partial u)}{\partial x} = 0$$

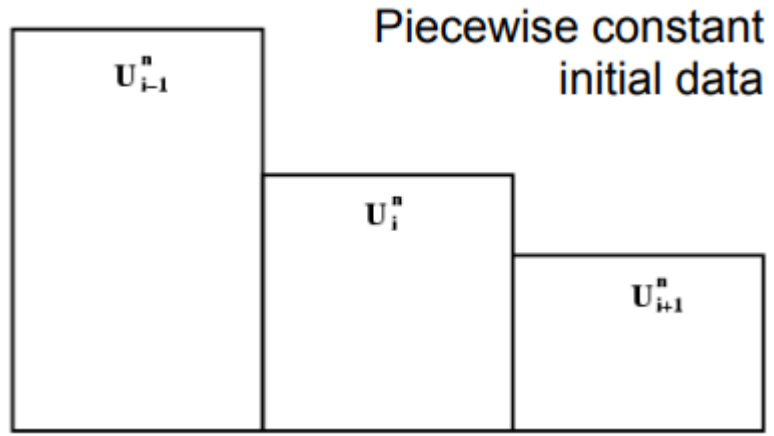
Approximating the derivative in ***X*** has two choices: forwards or backwards

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

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

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

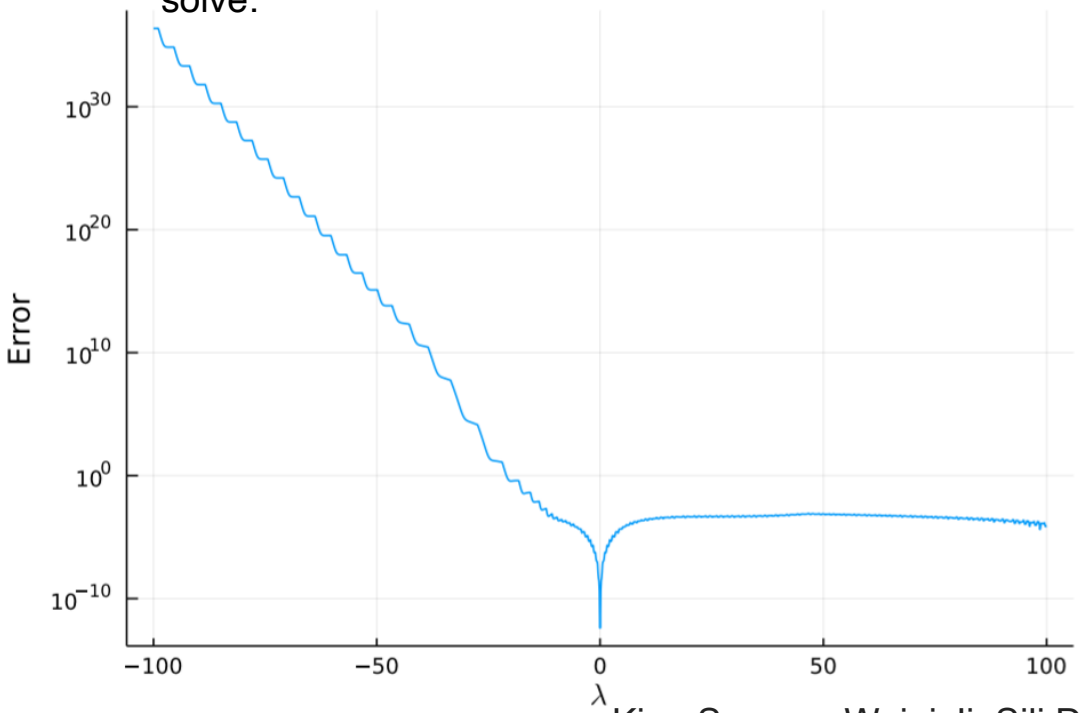


# Problems With Naïve Adjoint Approaches On Stiff Equations

**Error grows exponentially…**

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

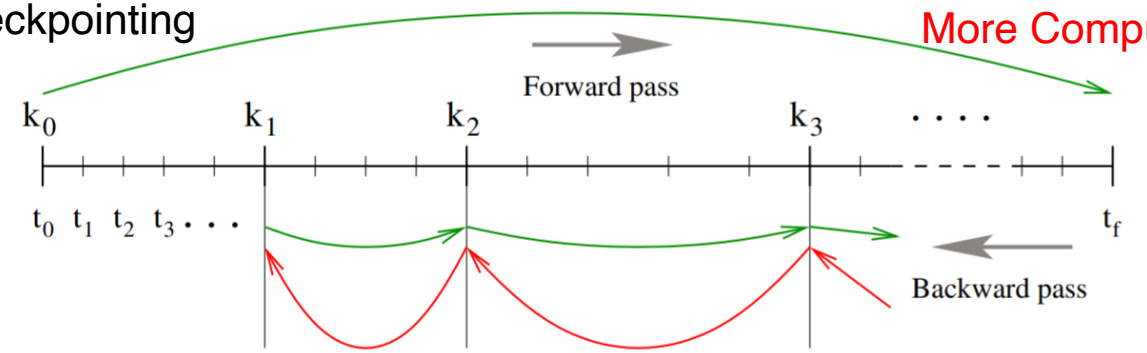



How do you get u(t) while solving backwards?
3 options!

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

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

3. Checkpointing More Compute



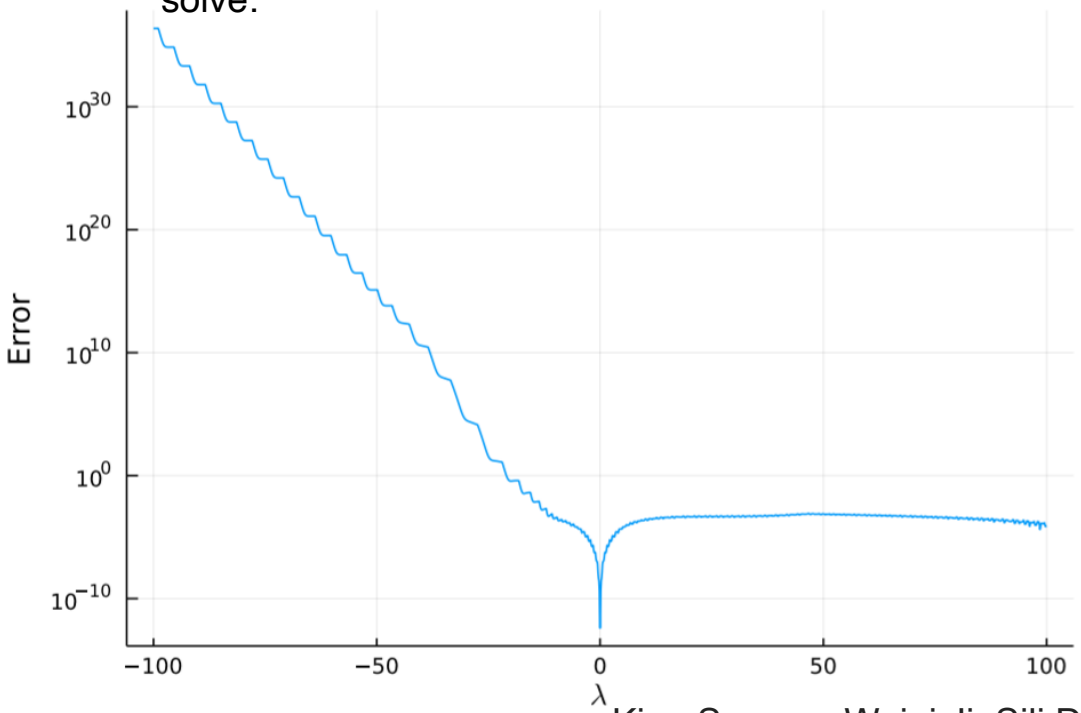

Each choices has an engineering trade-off!

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

# Problems With Naïve Adjoint Approaches On Stiff Equations

**Error grows exponentially…**

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




**Compute cost is cubic with parameter size when stiff**

Size of reverse ODE system is:

$$2states + parameters$$

Linear solves inside of stiff ODE solvers, ~cubic

Thus, adjoint cost:

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

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

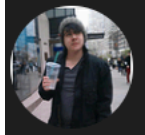
**But automatic differentiation**

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

# Symbolic Differentiation on Code

## Does Zygote differentiate symbolically?

Specific Domains · Machine Learning · question · zygote · differentiation

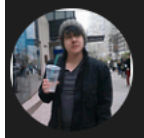


```
using Symbolics
@variables x
function f(x)
    out = one(x)
    for i in 1:5
        out *= x^i
    end
    out
end
sin(f(x)) # sin(x^15)
```

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

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

# Automatic Differentiation as Differentiation in the Language of Code



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

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

# Automatic Differentiation as Differentiation in the Language of Code

On that same example, this looks like:

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

f((1,1)) # (0.017537178497086 32, 0.36676042682811677)
dsinfx(1) # 0.3667040292067162
```

**More Details on the Algorithm, see the SciML Book:**

**book.sciml.ai**

**Chapter 10**

# What does automatic differentiation of an ODE solver give you?

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

# Wrong gradient for some sensealgs #273

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

**anhi** commented on Jun 8, 2020

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

```
using DiffEqSensitivity, OrdinaryDiffEq, Zygote

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

    return values[end]
end

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

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

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

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

Assignees

No one—assig

Labels

None yet

Projects

None yet

Milestone

No milestone

Development

Successfully
issue.

Single sig
PumasAI

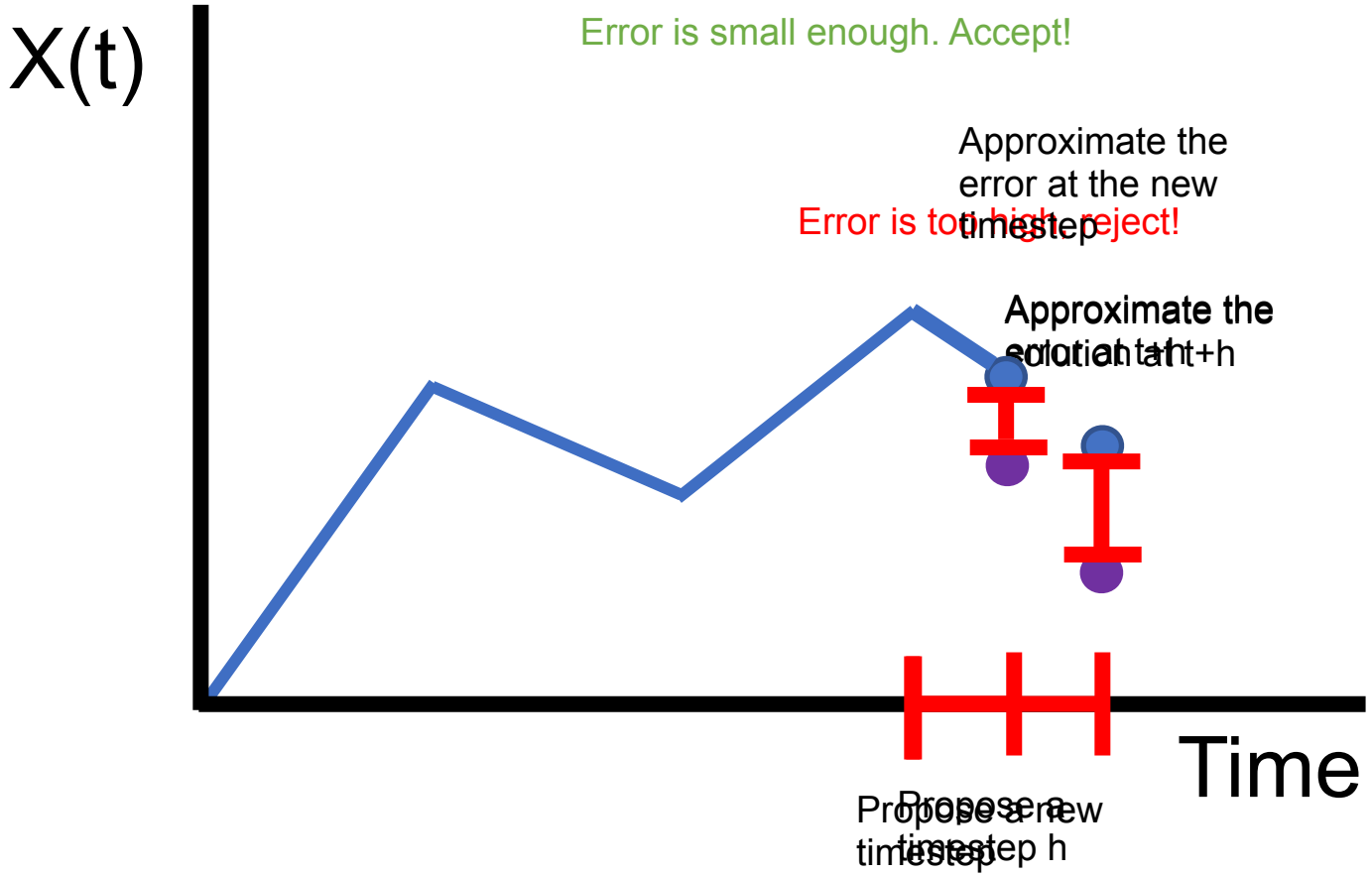
make dua
SciML/Dif

Notifications

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

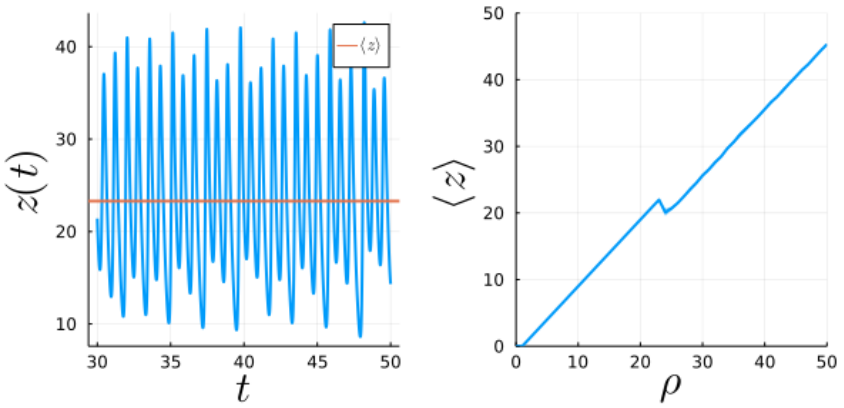
```
# Original AD
Zygote.gradient(p->sum(concrete_solve(prob, Tsit5(), u0, p, sensealg = ForwardDiffSensitivity(), saveat = 0.1, internalnorm = (u,t) -> sum
(abs2,u/length(u)), abstol=1e-12, reltol=1e-12)), p
) | ([29.755582164326086, 10.206643764088689, 53.37700890093473, 3.5509327396481583],)

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

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

# How adaptivity works



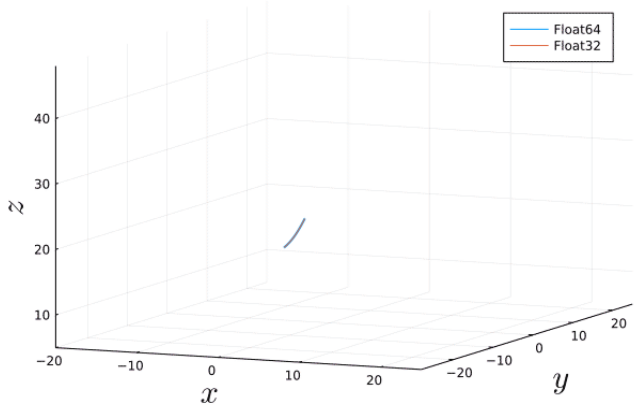
# Any more cases where AD is incorrect?

# Differentiation of Chaotic Systems: Shadow Adjoints




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

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




- AD and finite differencing fails!

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

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

- Shadowing methods in DiffEqSensitivity.jl

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

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

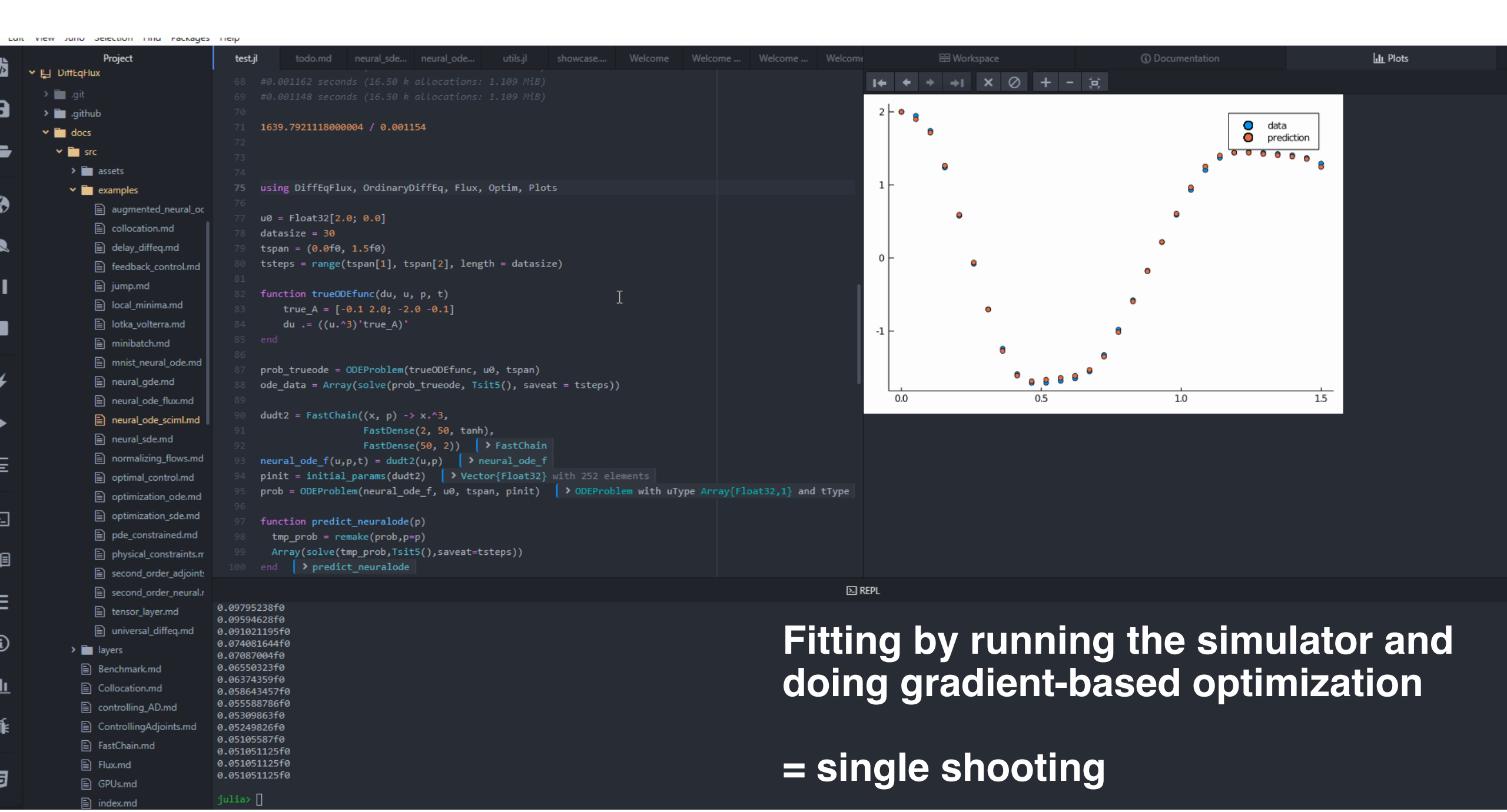
https://frankschae.github.io/post/shadowing/

# Conclusion Part 1:

# Be careful about how you compute derivatives of equation solvers

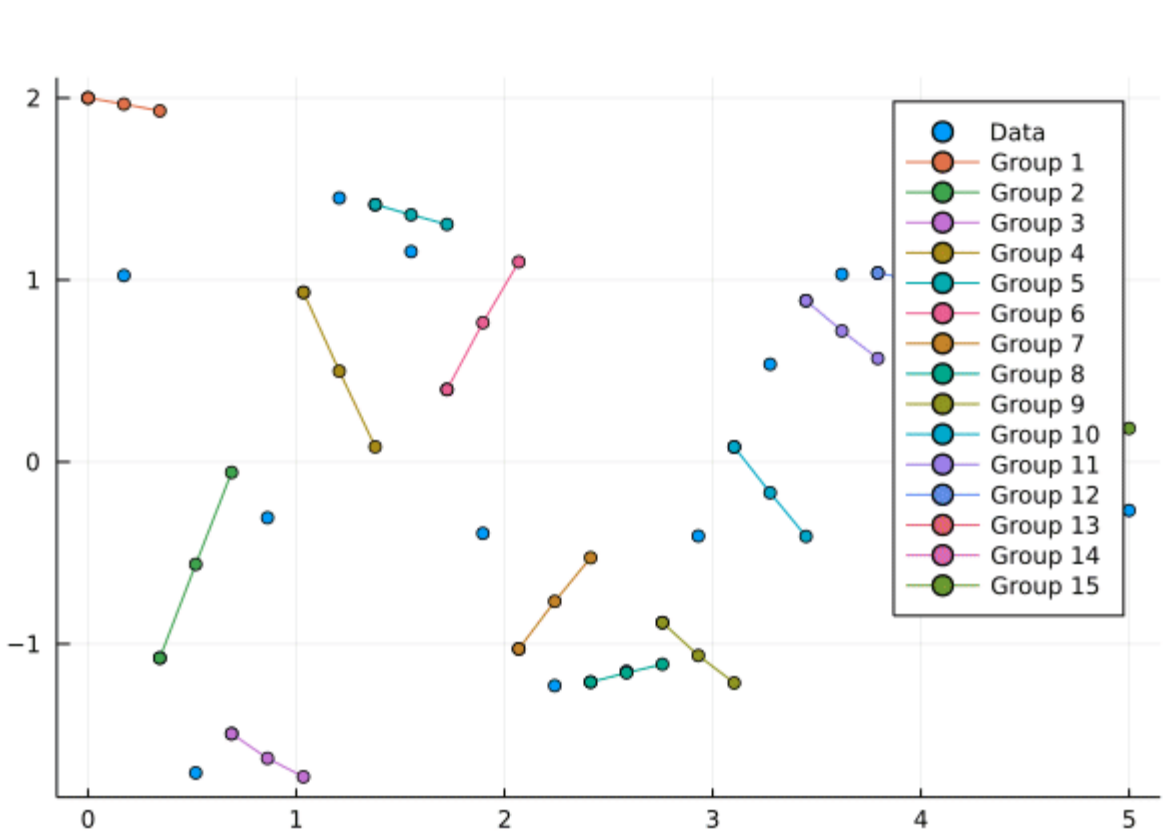
# Part 2:

# Methods which improve the fitting process


Project
DiffEqFlux
.git
.github
docs
src
assets
examples
augmented_neural_oc
collocation.md
delay_diffeq.md
feedback_control.md
jump.md
local_minima.md
lotka_volterra.md
minibatch.md
mnist_neural_ode.md
neural_gde.md
neural_ode_flux.md
neural_ode_sciml.md
neural_sde.md
normalizing_flows.md
optimal_control.md
optimization_ode.md
optimization_sde.md
pde_constrained.md
physical_constraints.m
second_order_adjoint
second_order_neural.
tensor_layer.md
universal_diffeq.md
layers
Benchmark.md
Collocation.md
controlling_AD.md
ControllingAdjoints.md
FastChain.md
Flux.md
GPUs.md
index.md
test.jl
todo.md
neural_sde...
neural_ode...
utils.jl
showcase....
Welcome
Welcome ...
Welcome ...
Welcome
Workspace
Documentation
Plots
#0.001162 seconds (16.50 k allocations: 1.109 MiB)
#0.001148 seconds (16.50 k allocations: 1.109 MiB)
1639.7921118000004 / 0.001154
using DiffEqFlux, OrdinaryDiffEq, Flux, Optim, Plots
u0 = Float32[2.0; 0.0]
datasize = 30
tspan = (0.0f0, 1.5f0)
tsteps = range(tspan[1], tspan[2], length = datasize)
function trueODEfunc(du, u, p, t)
true_A = [-0.1 2.0; -2.0 -0.1]
du .= ((u.^3)'true_A)'
end
prob_trueode = ODEProblem(trueODEfunc, u0, tspan)
ode_data = Array(solve(prob_trueode, Tsit5(), saveat = tsteps))
dudt2 = FastChain((x, p) -> x.^3,
FastDense(2, 50, tanh),
FastDense(50, 2))
FastChain
neural_ode_f(u,p,t) = dudt2(u,p)
neural_ode_f
pinit = initial_params(dudt2)
Vector{Float32} with 252 elements
prob = ODEProblem(neural_ode_f, u0, tspan, pinit)
ODEProblem with uType Array{Float32,1} and tType
function predict_neuralode(p)
tmp_prob = remake(prob,p=p)
Array(solve(tmp_prob,Tsit5(),saveat=tsteps))
end
predict_neuralode
data
prediction
2
1
0
-1
0.0
0.5
1.0
1.5
REPL
0.09795238f0
0.09594628f0
0.091021195f0
0.074081644f0
0.07087004f0
0.06550323f0
0.06374359f0
0.058643457f0
0.055588786f0
0.05309863f0
0.05249826f0
0.05105587f0
0.051051125f0
0.051051125f0
0.051051125f0
julia>
Fitting by running the simulator and doing gradient-based optimization
= single shooting

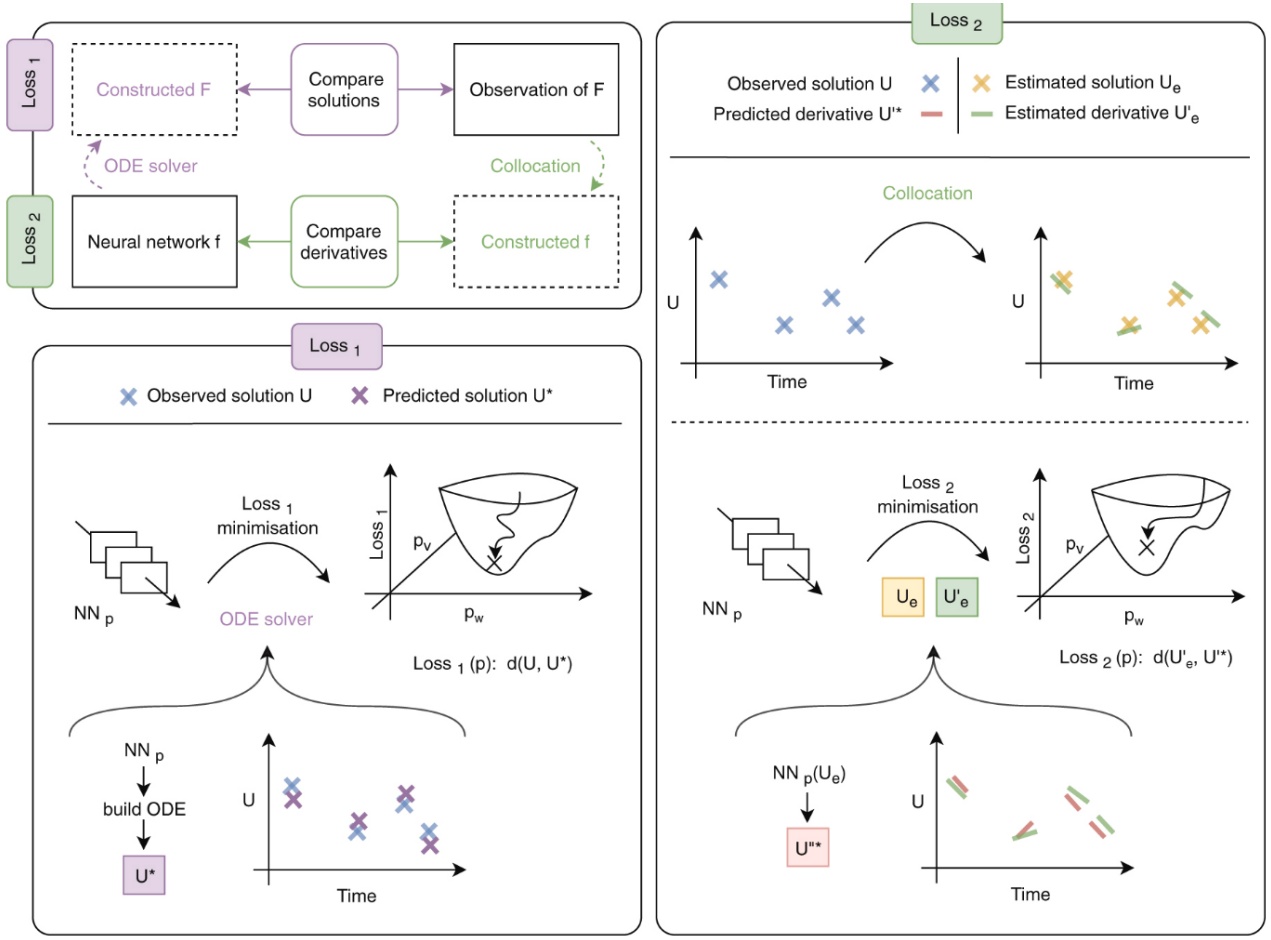
**Single shooting is not numerically robust. Other loss functions are required in practice!**

# Some Alternative Loss Functions: Multiple Shooting and Collocation




## Multiple Shooting Methods

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

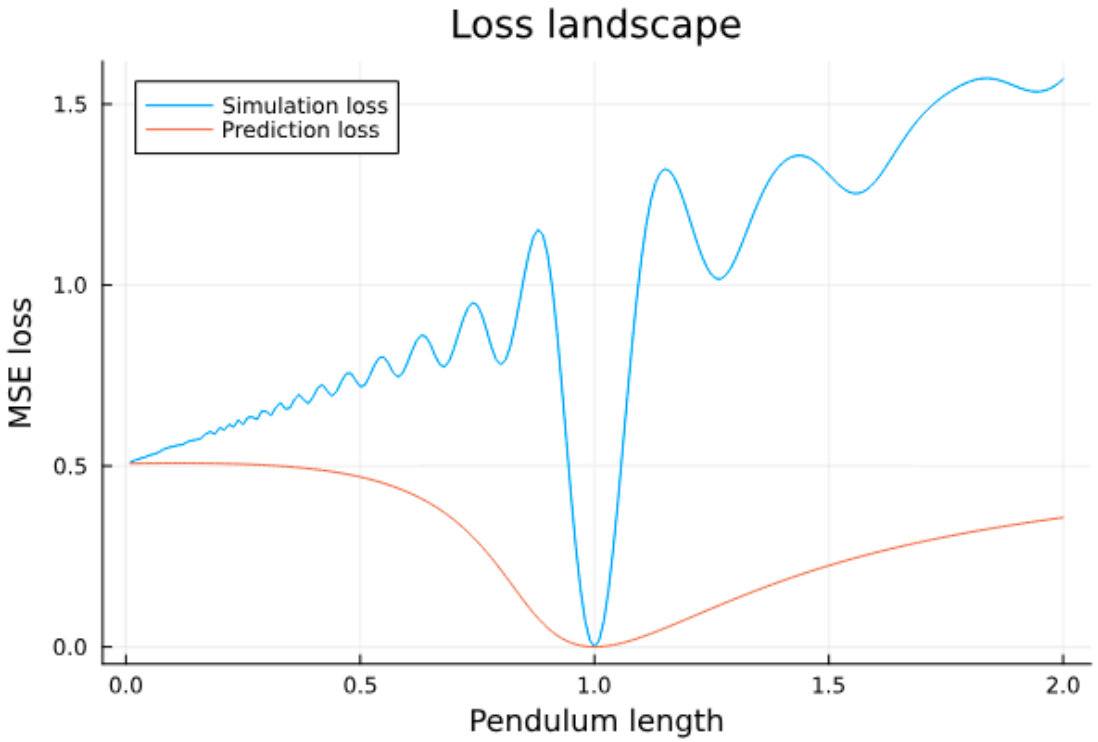



Roesch, Elisabeth, Christopher Rackauckas, and Michael PH Stumpf. "Collocation based training of neural ordinary differential equations." *Statistical Applications in Genetics and Molecular Biology* (2021).

# Prediction Error Method (PEM)

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

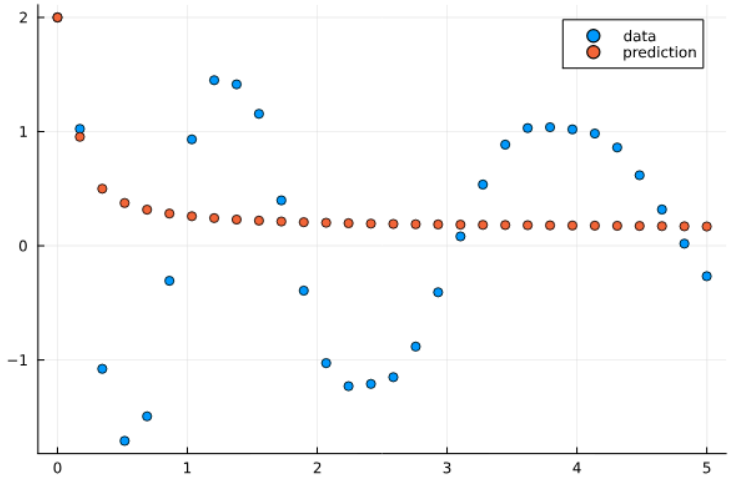
```julia
function predictor(du, u, p, t)
    g = 9.82
    L, K, y = p # pendulum length, observer gain and measurements
    gL = g / L
    θ = u[1]
    dθ = u[2]
    yt = y(t)
    e = yt - θ
    du[1] = dθ + K * e
    du[2] = -gL * sin(θ)
end
```




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

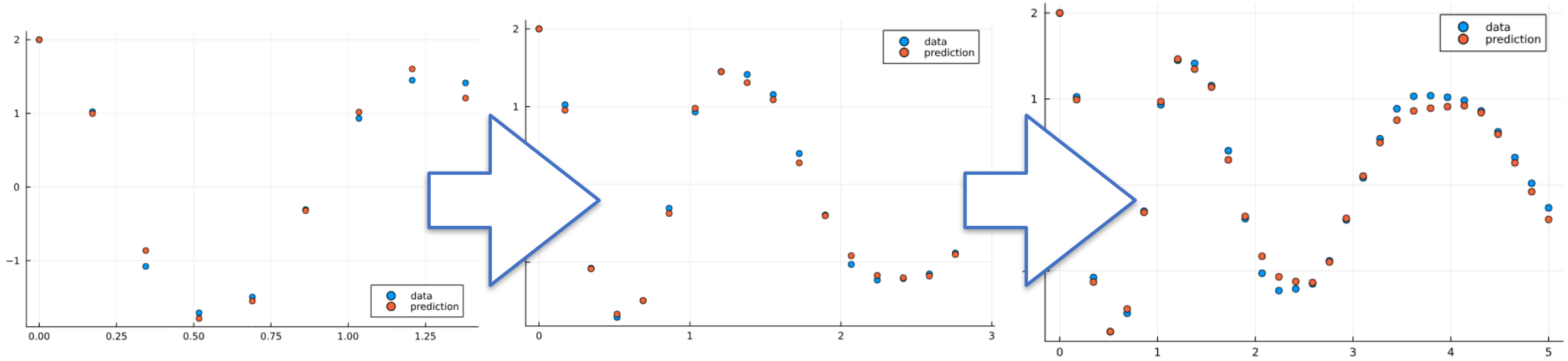
https://docs.sciml.ai/SciMLSensitivity/dev/examples/ode/prediction_error_method/

# Simple Tricks: Growing the Time Interval




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

**Successively grow the interval**



# Let's go back to this example

Run the code yourself!

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

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

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

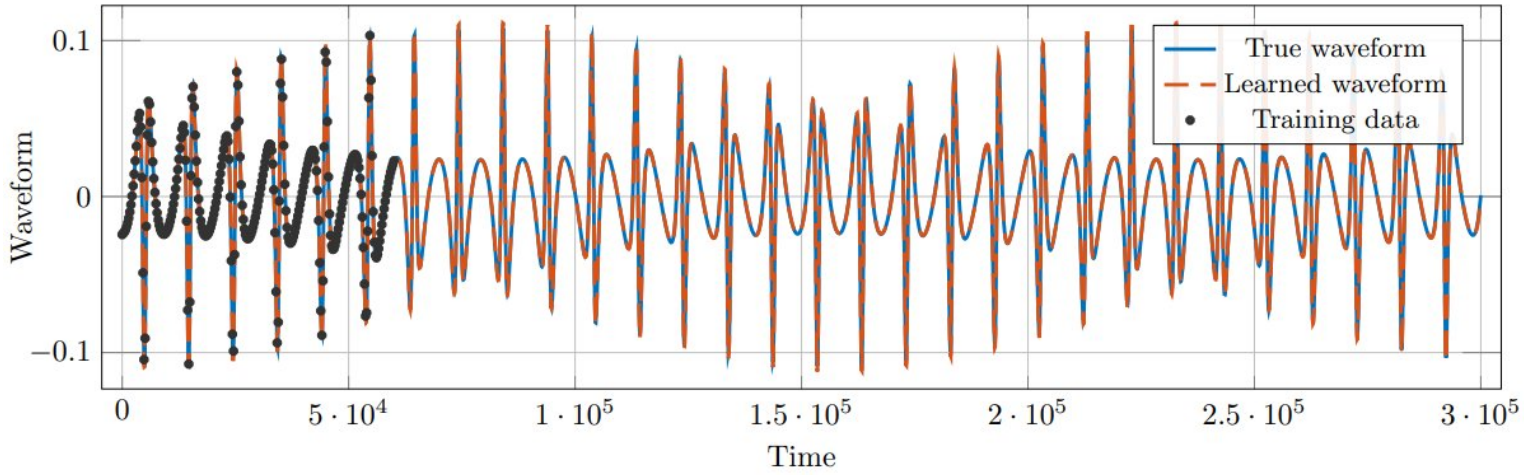
$$\dot{\chi} = \frac{(1 + e\cos(\chi))^2}{Mp^{3/2}}\big(1 + \mathcal{F}_2(\cos(\chi), p, e)\big), \tag{5b}$$

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

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

Example using binary black hold dynamics with LIGO gravitational wave data

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



# Let's go back to this example

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

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

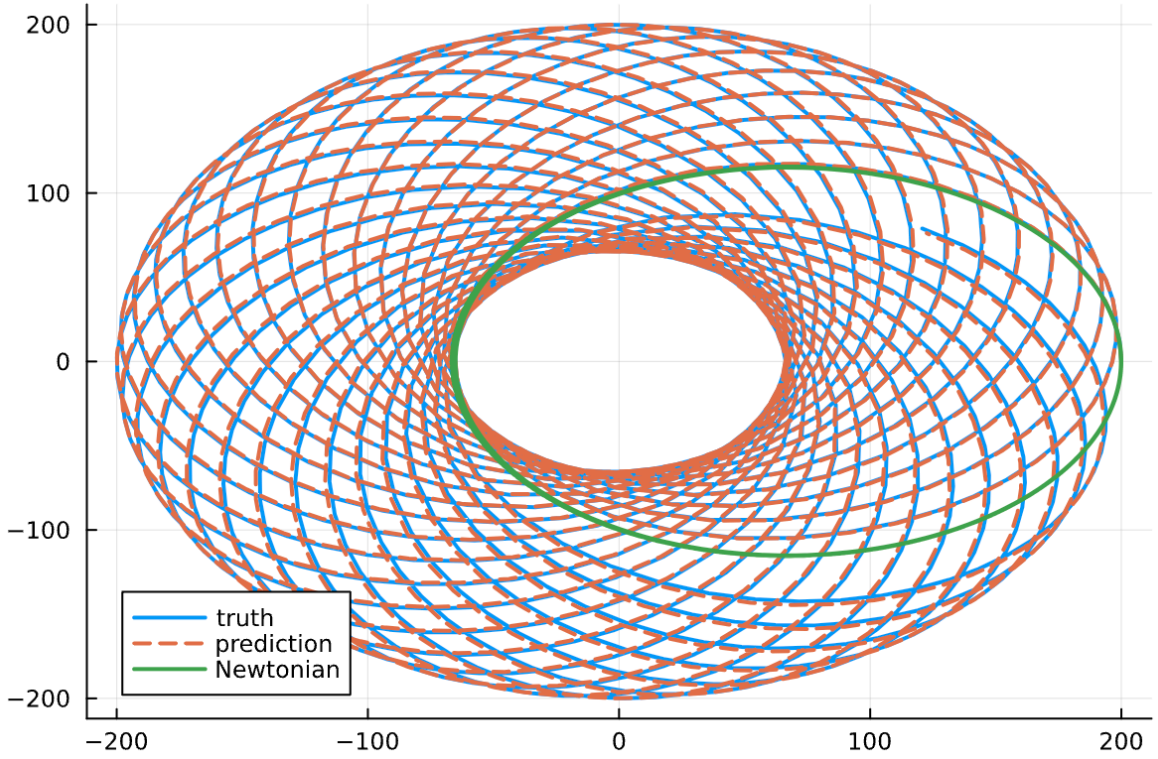
Upon denoting $\mathbf{x} = (\phi, \chi, p, e)$, we propose the following family of UDEs to describe the two-body relativistic dynamics:

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

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

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

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

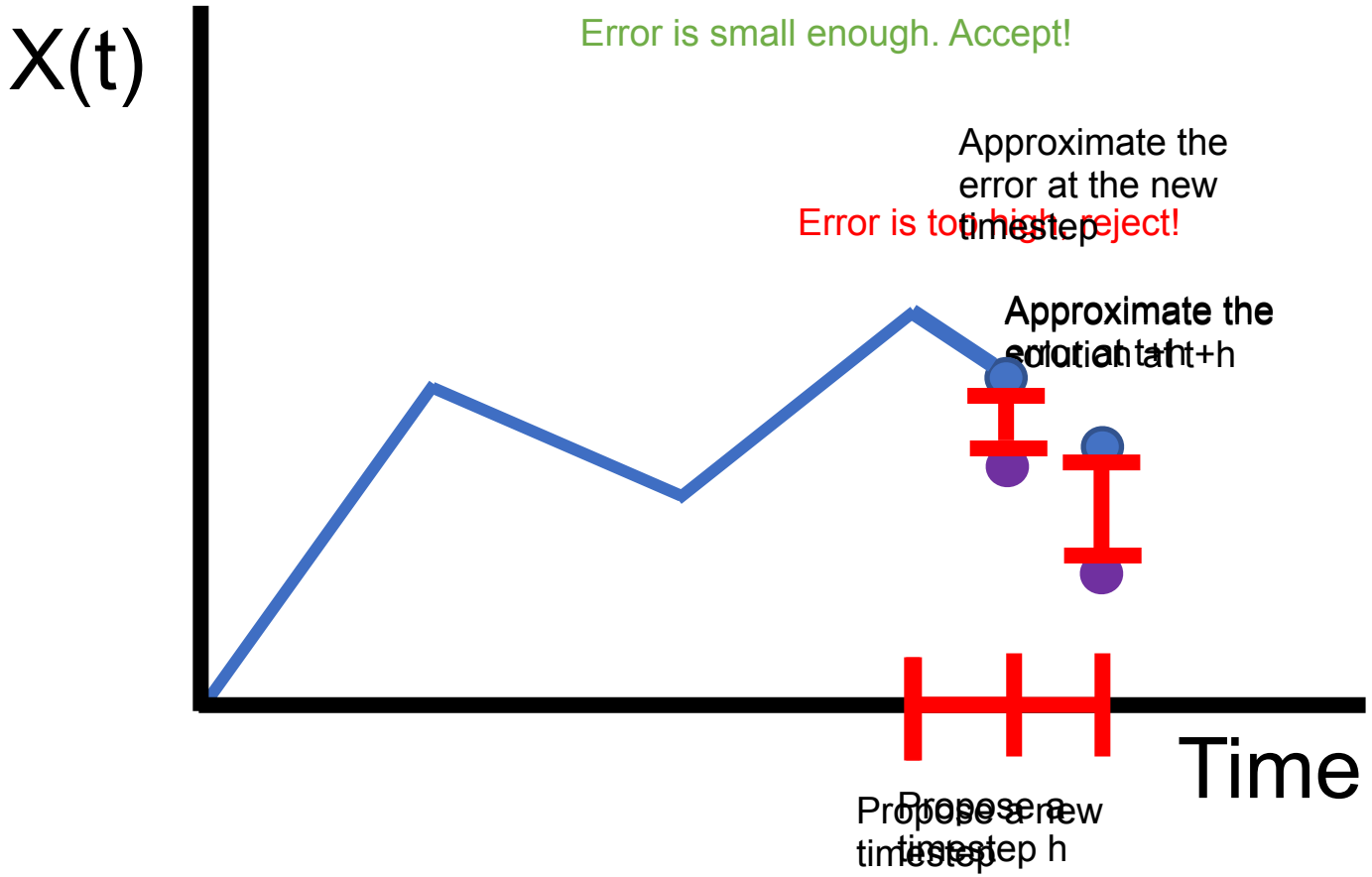


## Conclusion Part 2:

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

# Sidebar: A note on Neural Network Architectures in ODEs

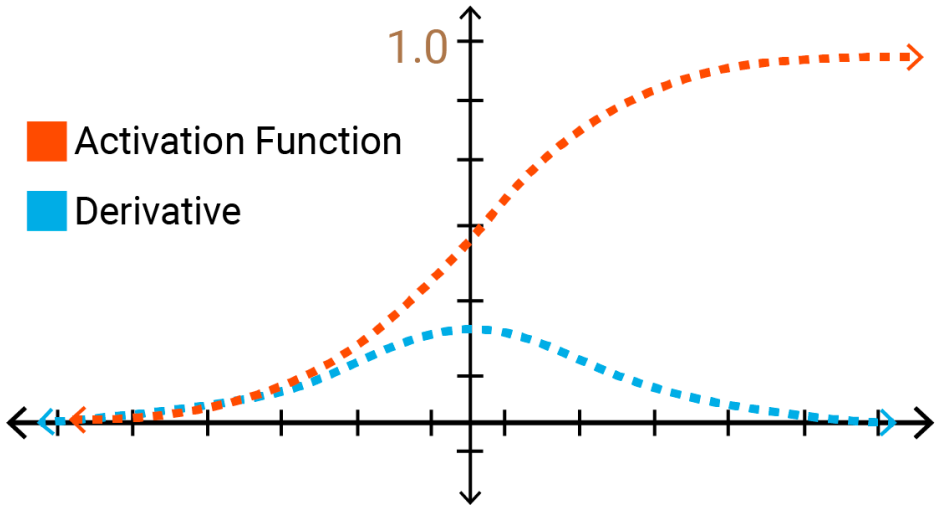
# ODE Solvers don't always go forwards!




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

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

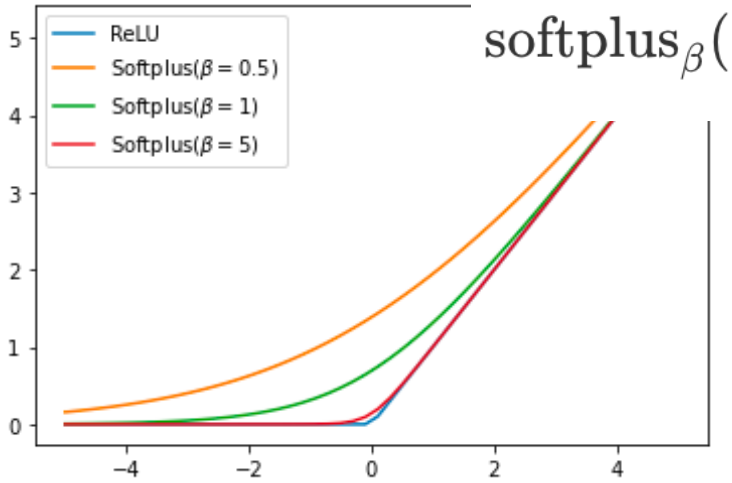
# Be Aware of Vanishing Gradients




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

Solutions:

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

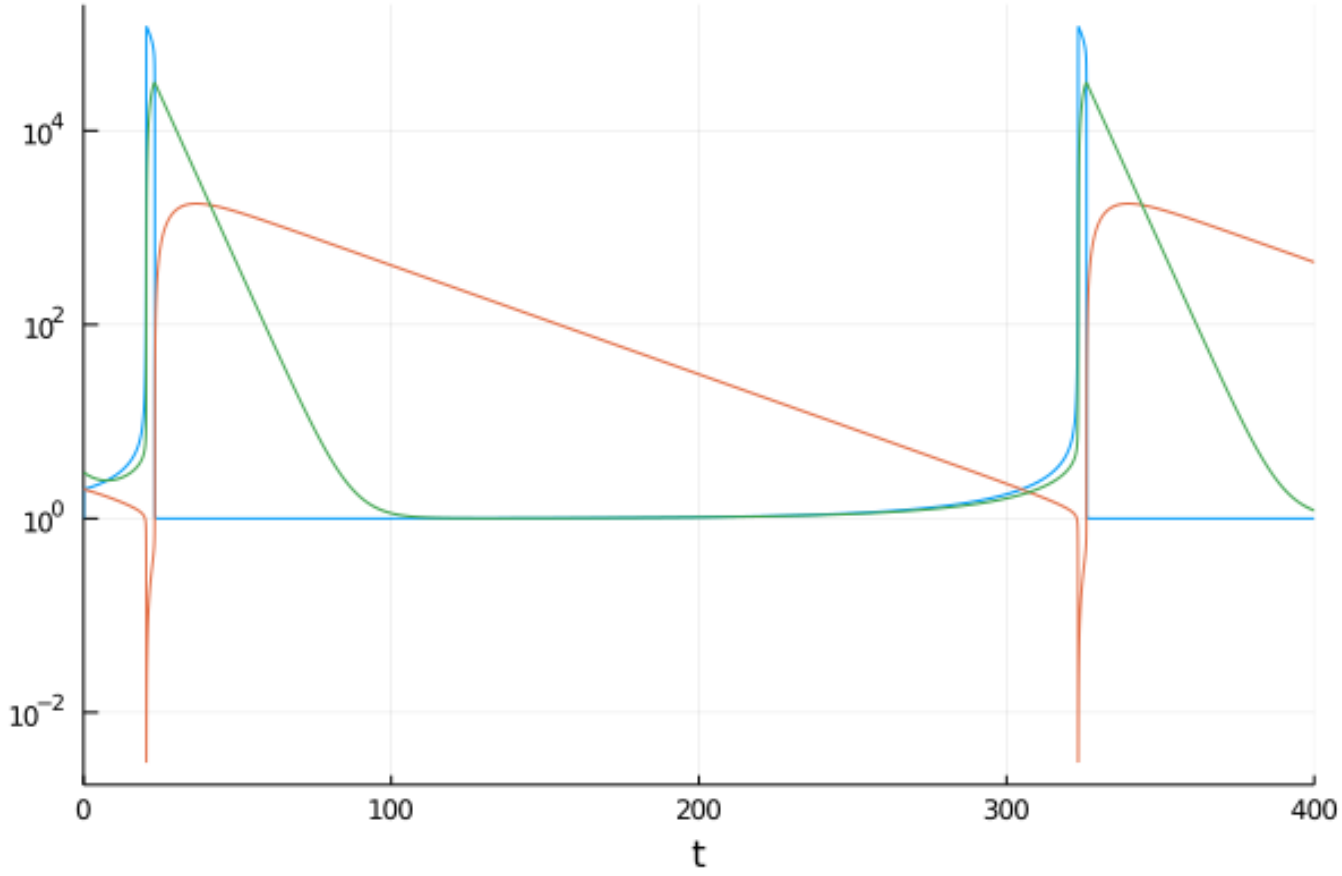


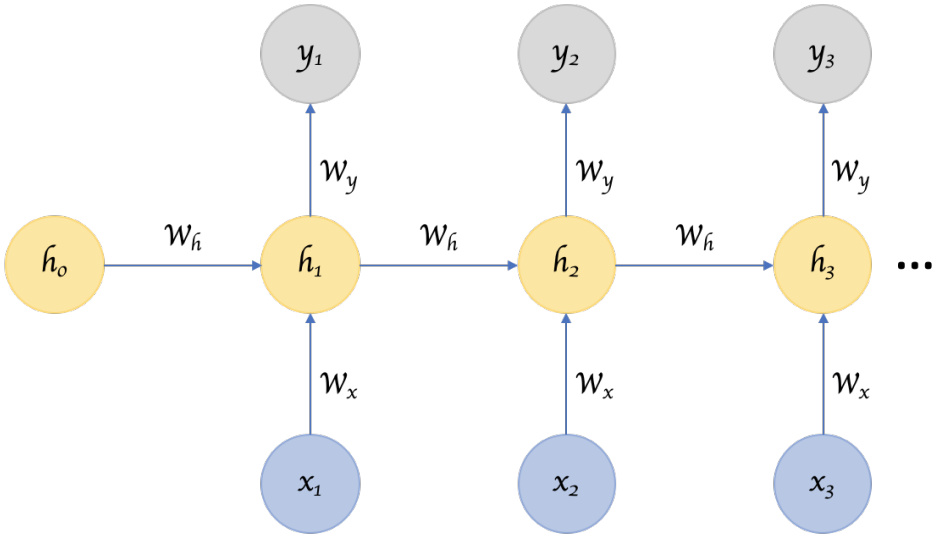

$$\text{softplus}_\beta(x) = \frac{1}{\beta}\log(1 + e^{\beta x})$$

# Part 3:

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

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



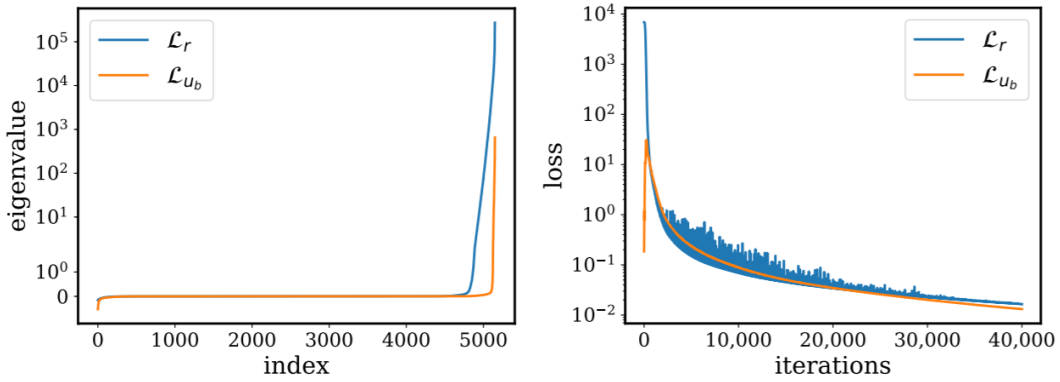

**Recurrent neural network? No!**

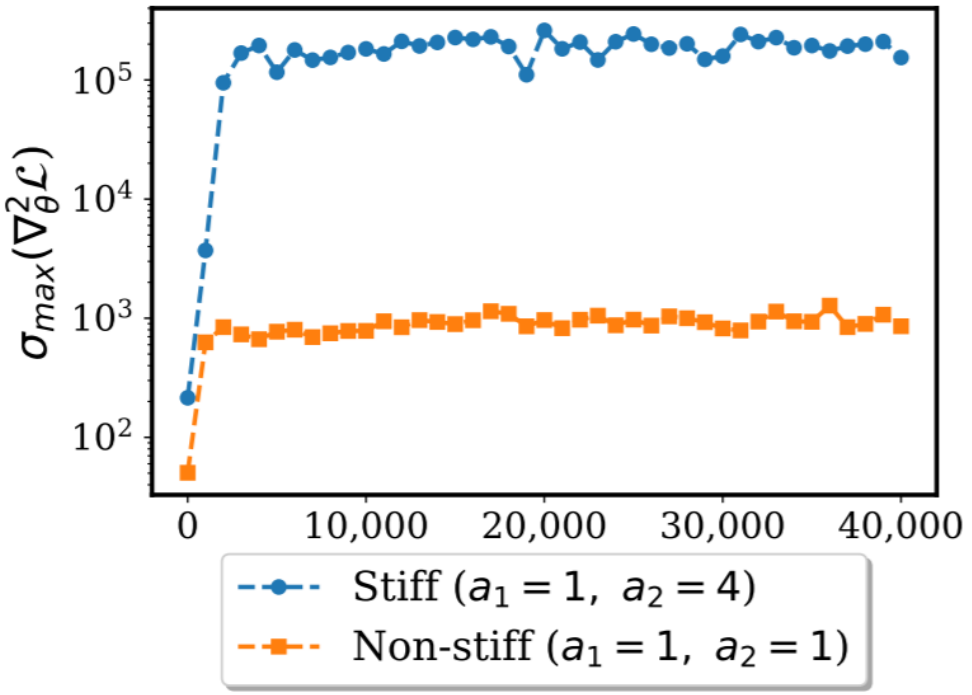
1. It's an explicit method! (Euler's)
2. Uniform steps will not capture the spikes!

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

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

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



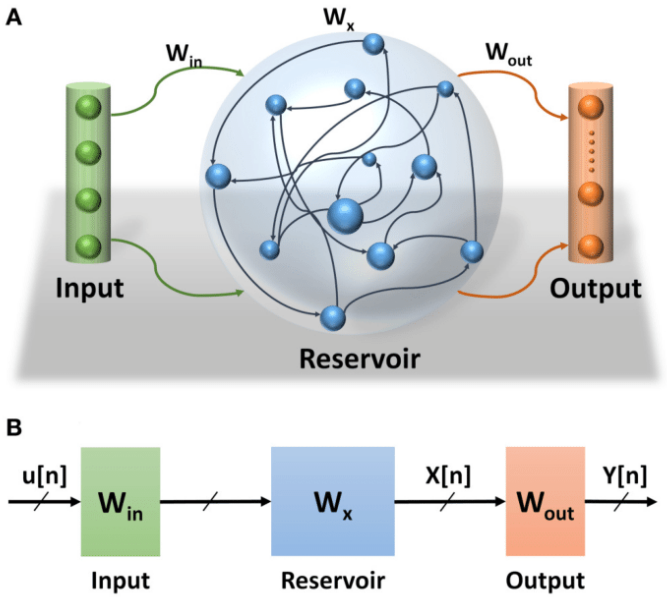




Understanding and mitigating gradient pathologies in physics-informed neural networks

Sifan Wang, Yujun Teng, Paris Perdikaris

# Idea: Avoid Gradients and Use an Implicit Fit

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




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

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

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

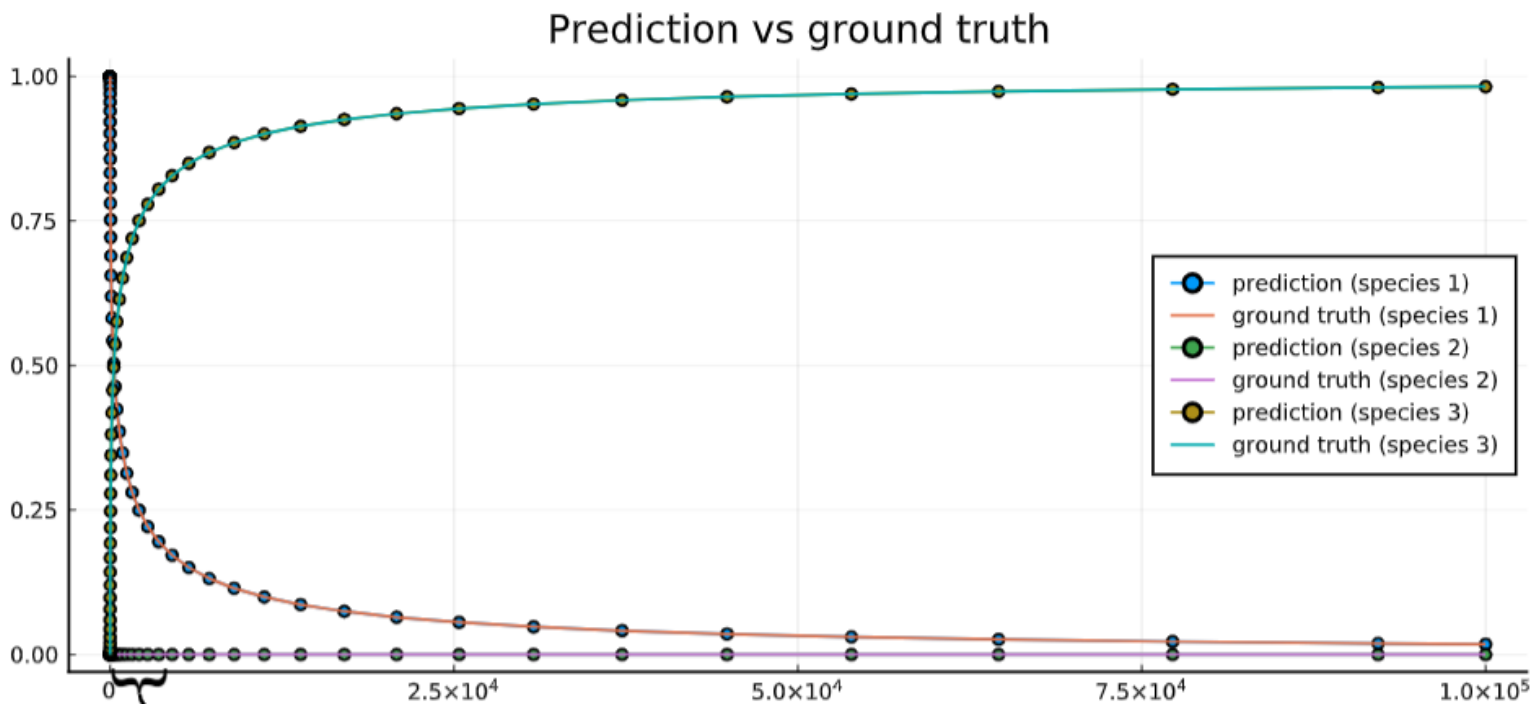
Get $W_{out}$ at many parameters of the system

Predict behavior at new parameters via:

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

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

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




## Robertson's Equations

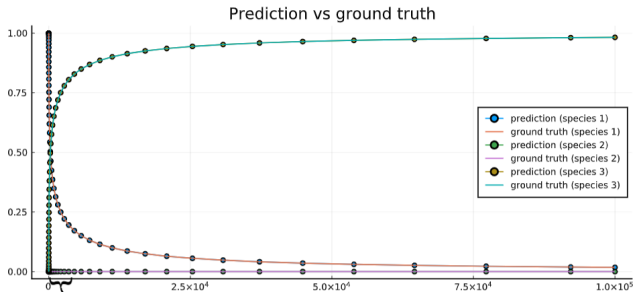
Classic stiff ODE
Used to test and break integrators
Volatile early transient

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

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

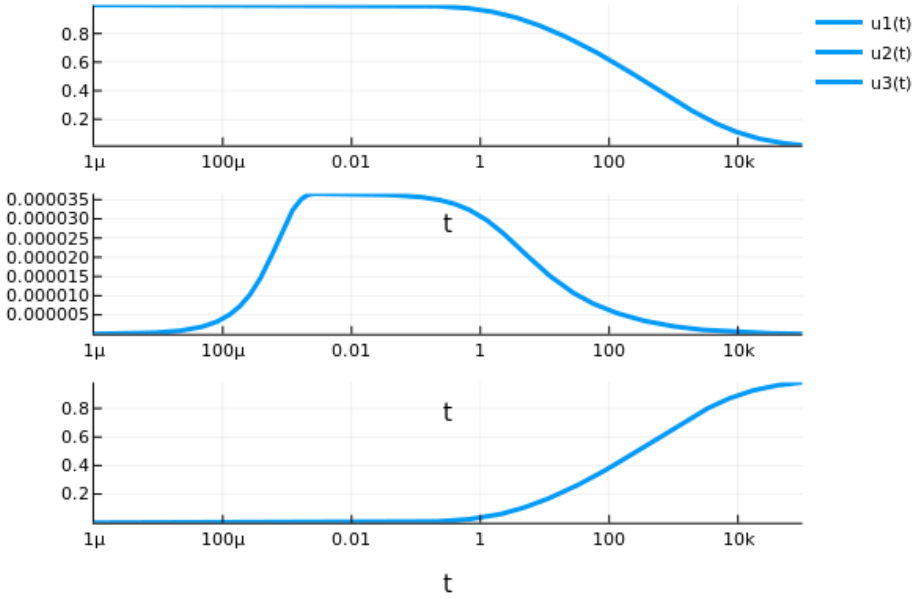
Ranjan Anantharaman, Yingbo Ma, Shashi Gowda, Chris Laughman, Viral Shah, Alan Edelman, Chris Rackauckas

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




**Log-Scale Fast Changes!**

No auto-catalyst, no dynamics




## Robertson's Equations

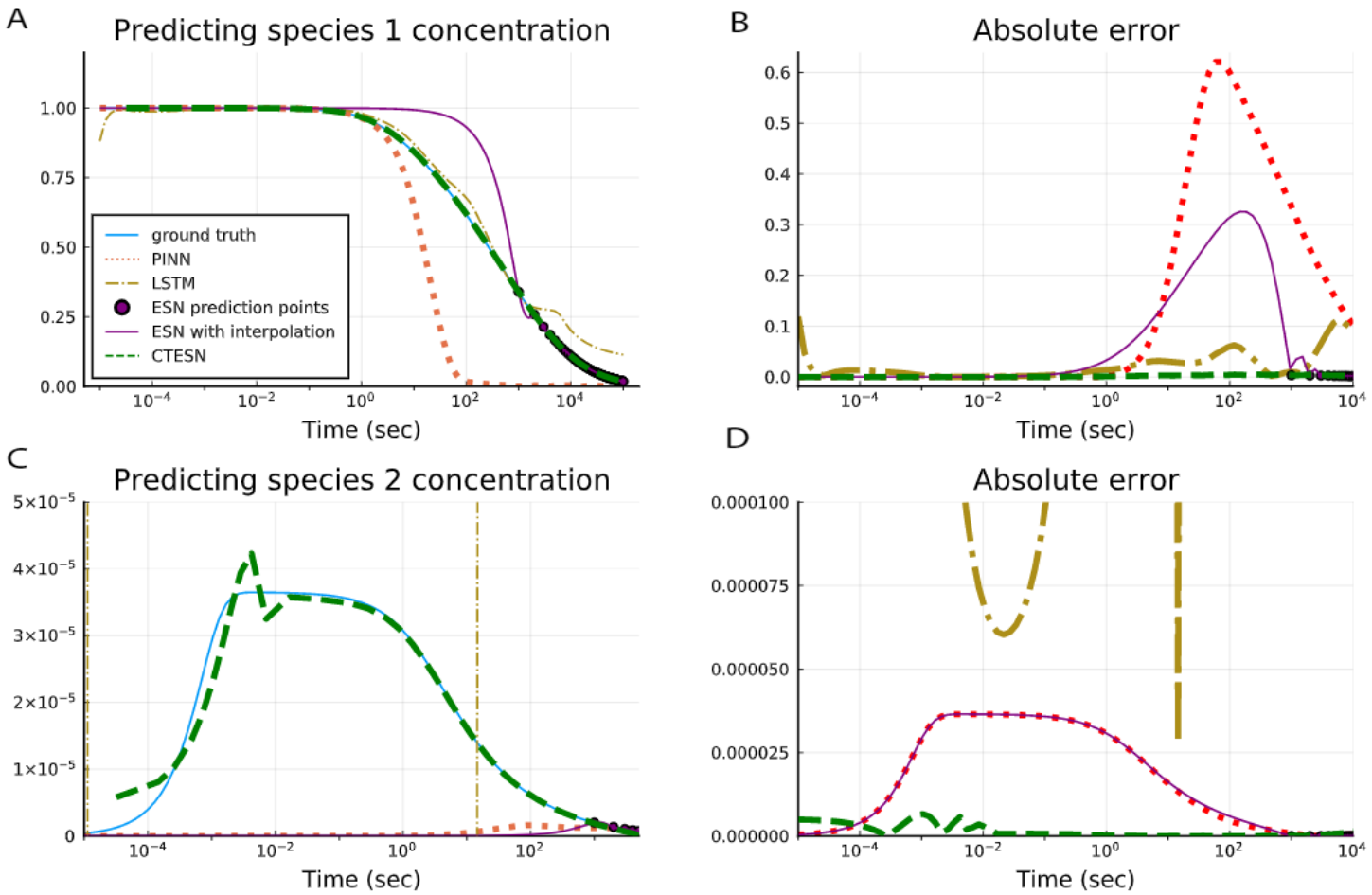
Classic stiff ODE
Used to test and break integrators
Volatile early transient

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

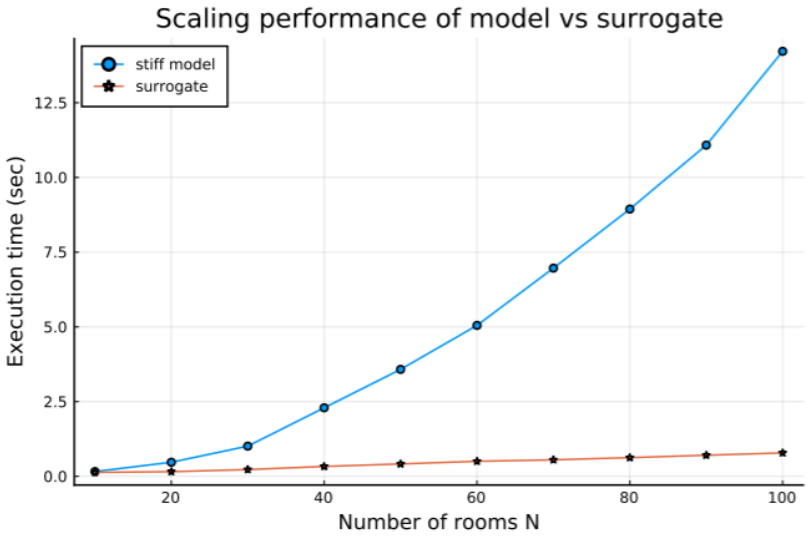
**Accelerating Simulation of Stiff Nonlinear Systems using Continuous-Time Echo State Networks**

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

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




**Only CTESNs Capture the Hard Dynamics**




**After training, 100x faster than direct simulation!**

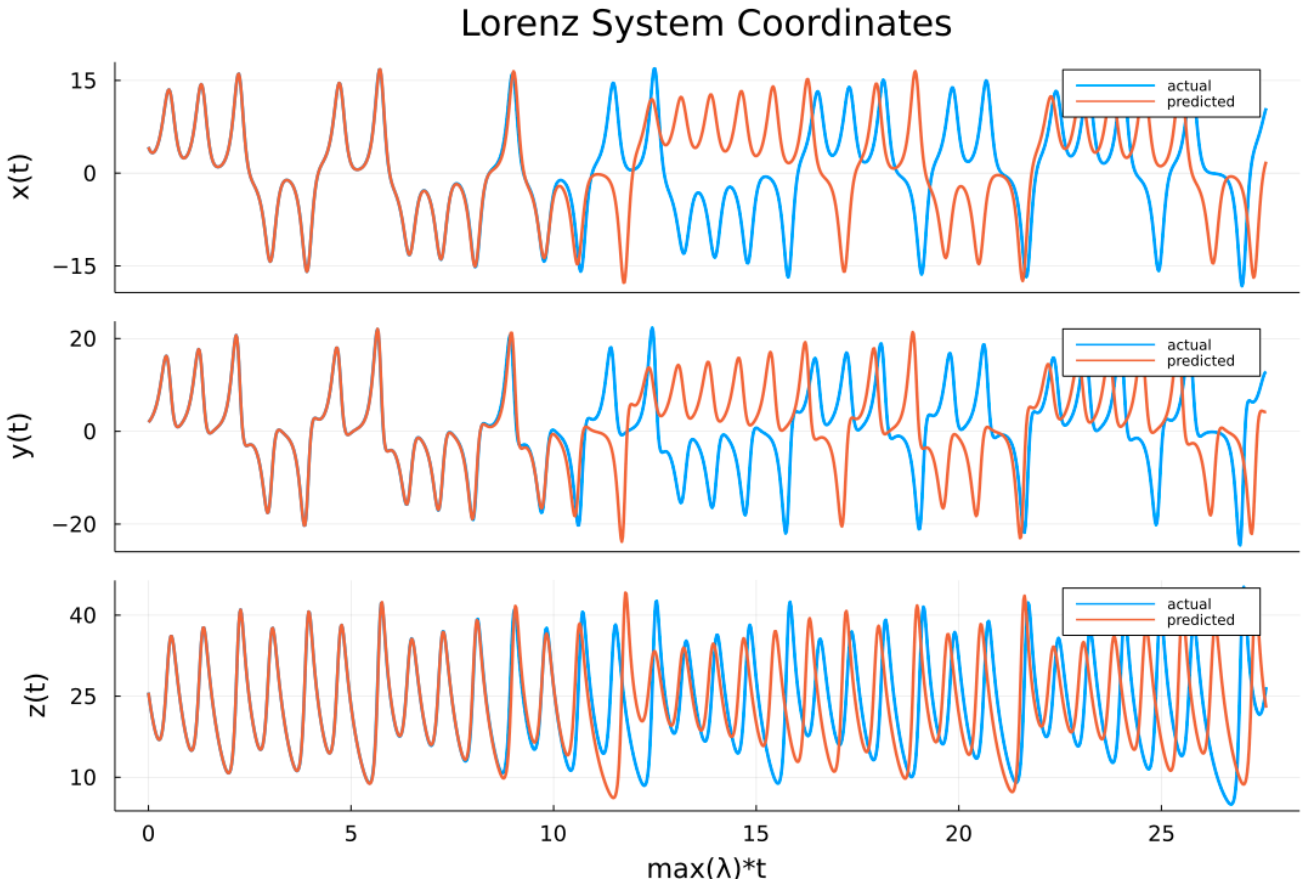
**Accelerating Simulation of Stiff Nonlinear Systems using Continuous-Time Echo State Networks**

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

# ReservoirComputing.jl

# Reservoir Computing.jl

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

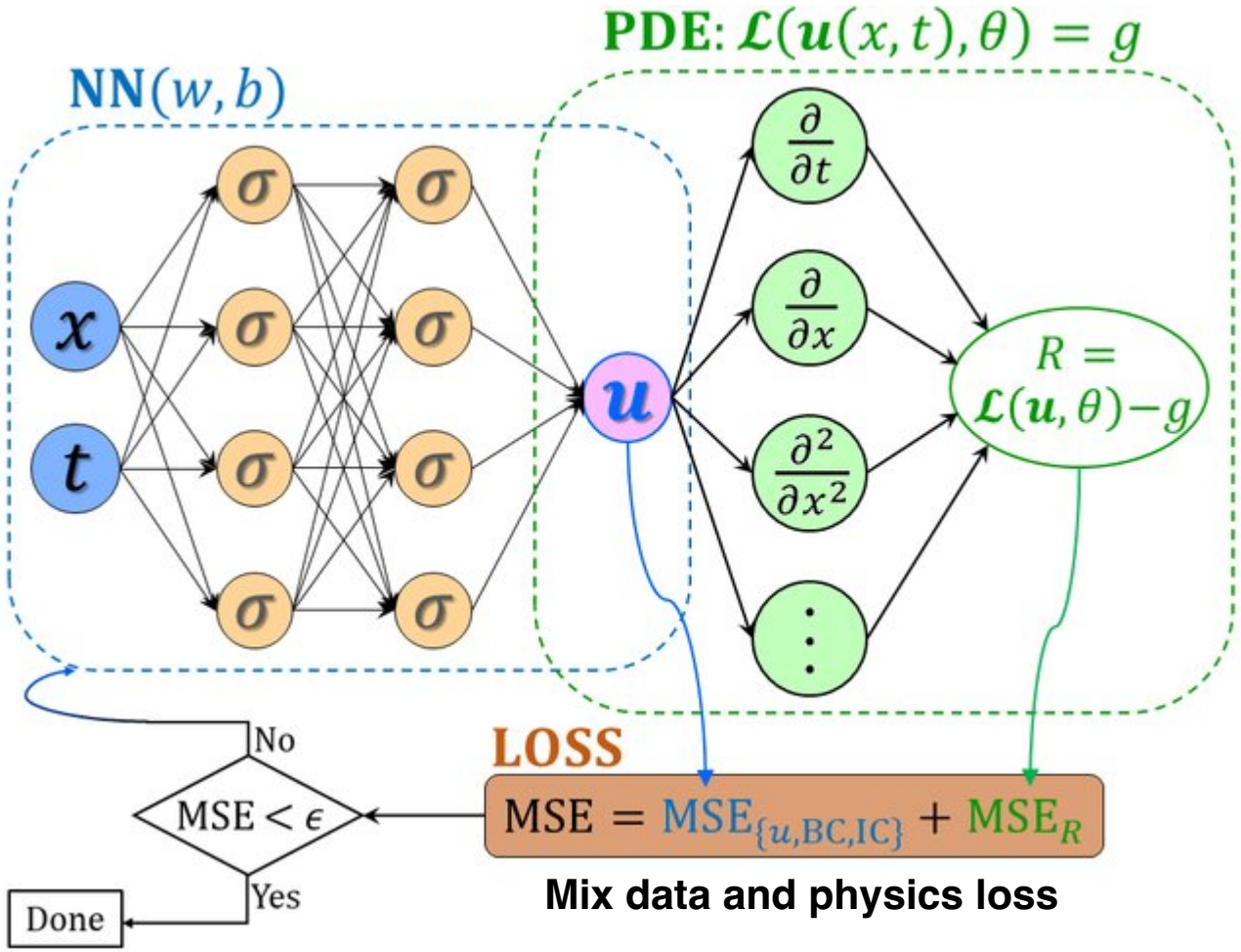


# Part 4: Performance

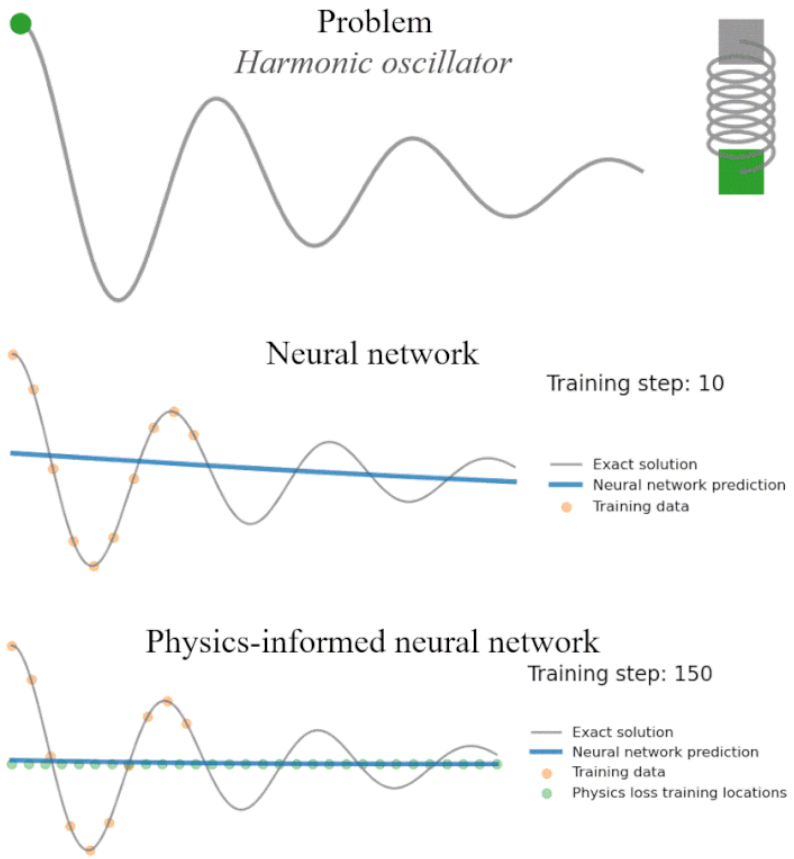
# A Deep Dive into how Performance is Different Between Deep Learning and Differentiable Simulation

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

# Why Use Physics-Informed Neural Networks?



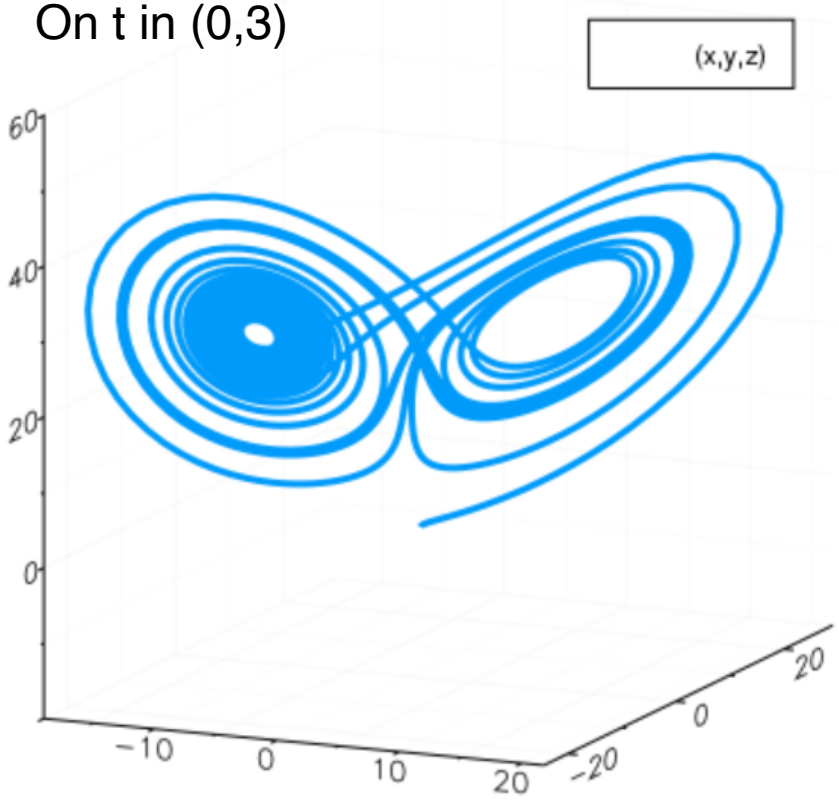

**Mix data and physics loss**




**Outperforms standard machine learning**

# Keeping Neural Networks Small Keeps Speed For Inverse Problems

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




## DeepXDE (TensorFlow Physics-Informed NN)

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

'train' took 362.351454 s
```

## DiffEqFlux.jl (Julia UDEs)

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

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

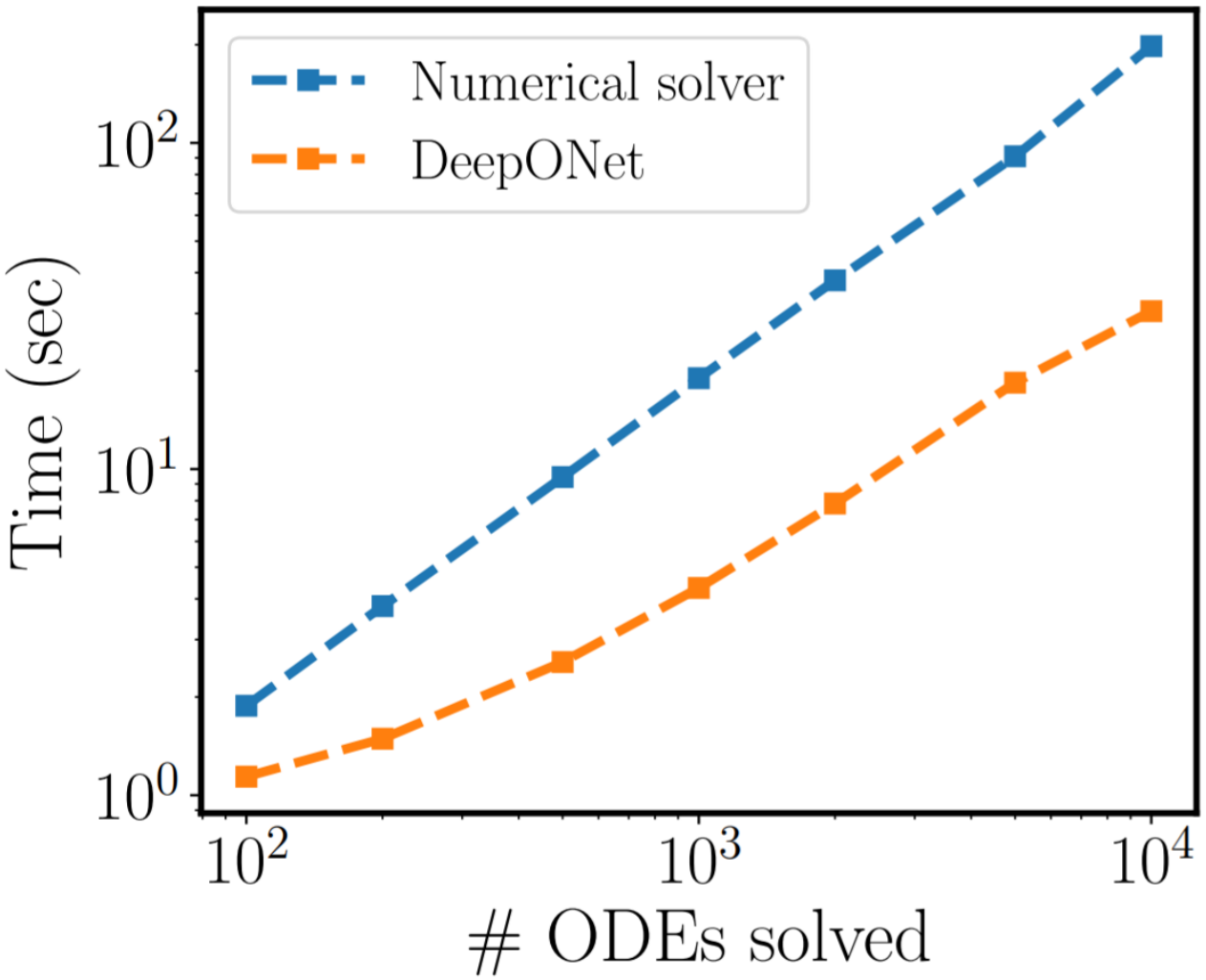
# Note on Neural Networks “Outperforming” Classical Solvers

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

Sifan Wang, Paris Perdikaris

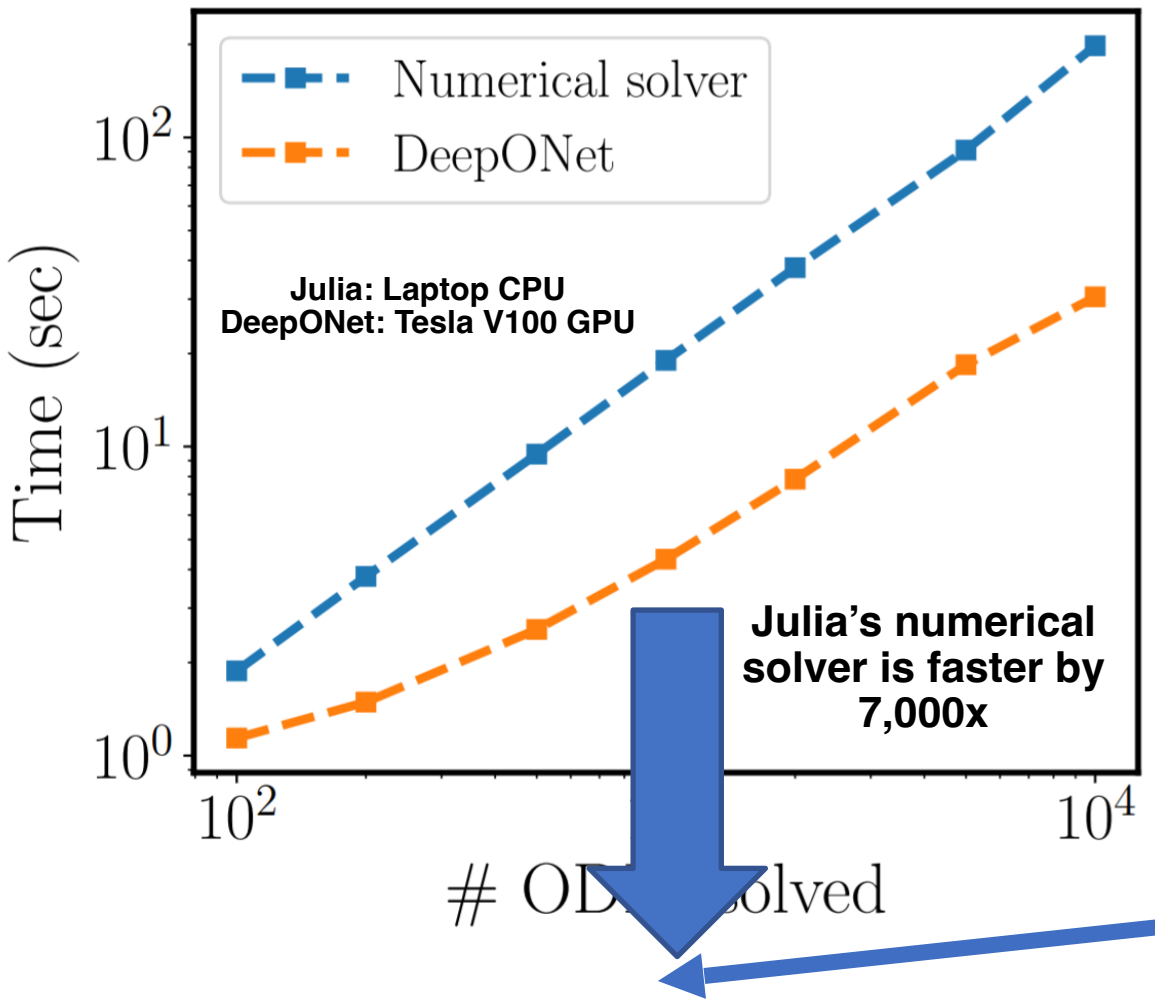
Ordinary and partial differential equations (ODEs/PDEs) play a paramount role in analyzing and simulating complex dynamic processes across all corners of science and engineering. In recent years machine learning tools are aspiring to introduce new effective ways of simulating PDEs, however existing approaches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

# Note on Neural Networks "Outperforming" Classical Solvers



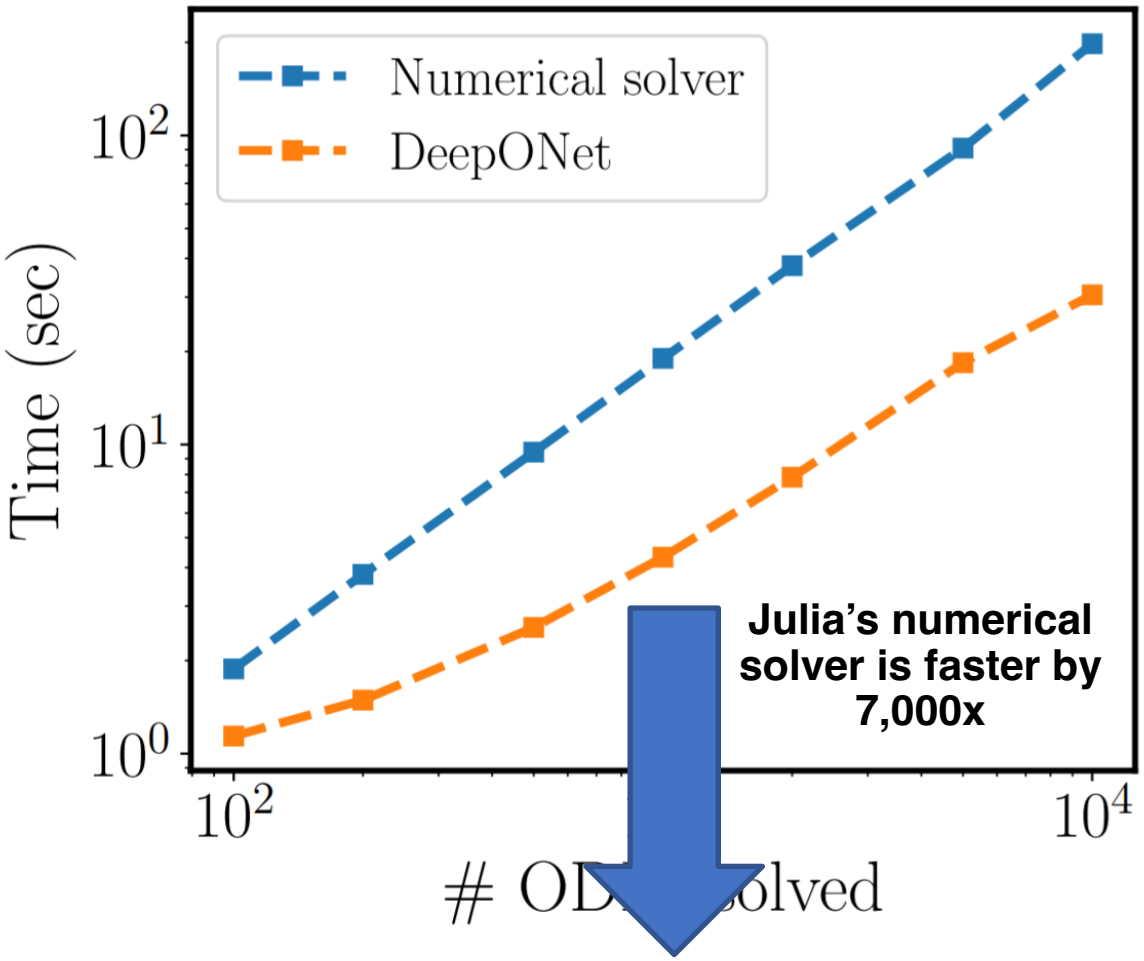

**Oh no, we're doomed!**

# Wait a second?




```julia
using ModelingToolkit, OrdinaryDiffEq, StaticArrays

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

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

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

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

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

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

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

# Wait a second?

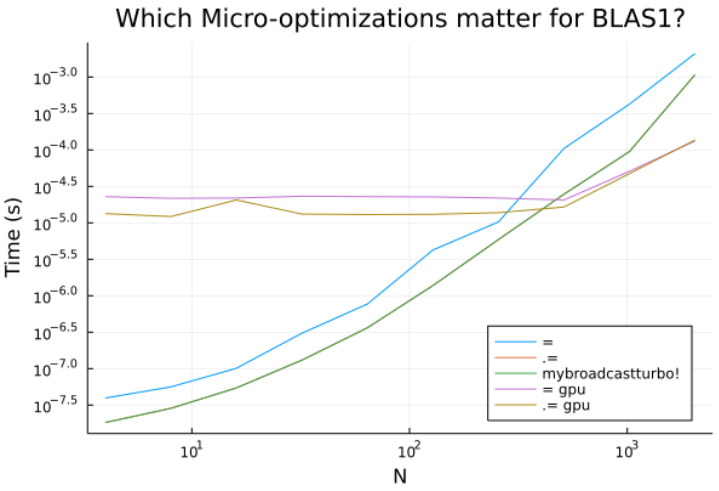



**Similar story on Fourier Neural Operator results!**

**How come so far off?**

**If Differentiable Simulation techniques are easily >1000x more efficient, then why doesn't everyone "see" that?**

# Code Optimization in Machine Learning vs Scientific Computing

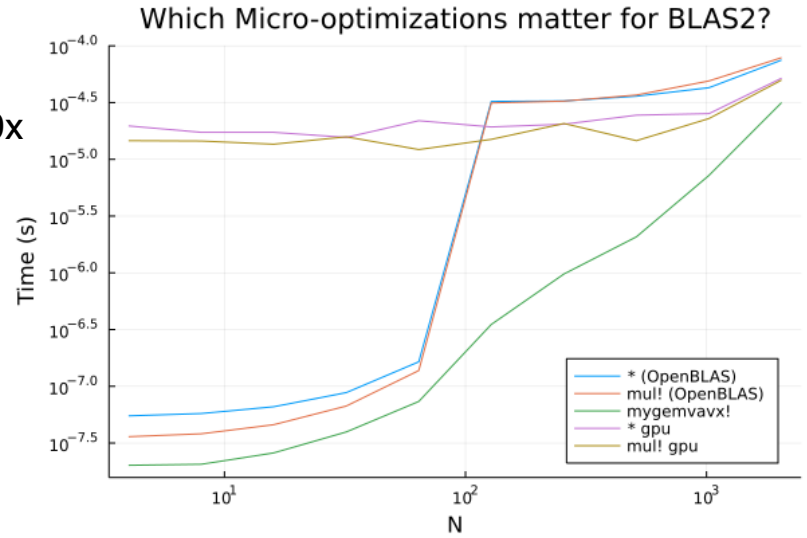


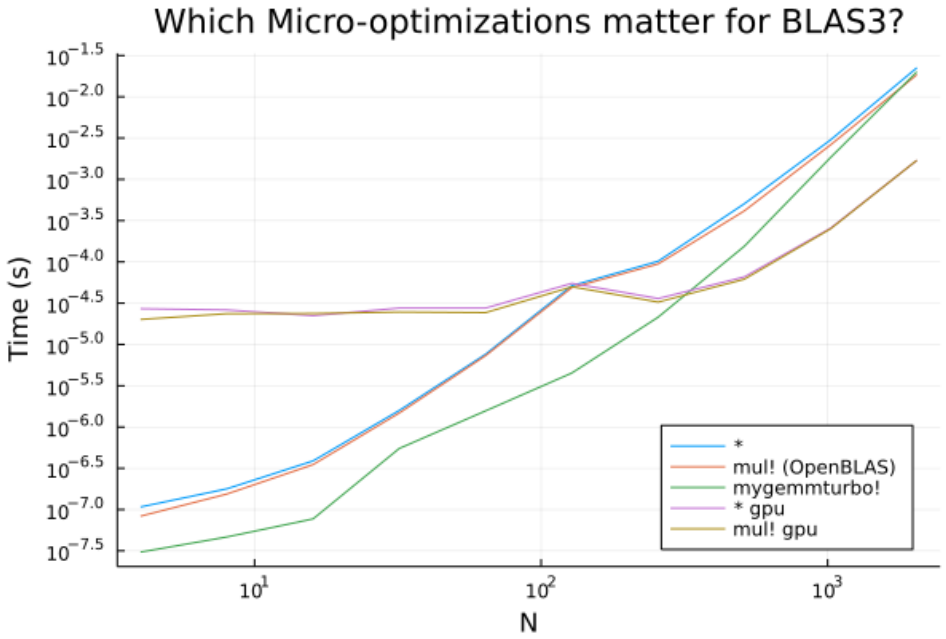

Scientific codes
O(n) and O(n^2)
operations

Mutation and
Memory management: 10x

Manual SIMD: 5x

…

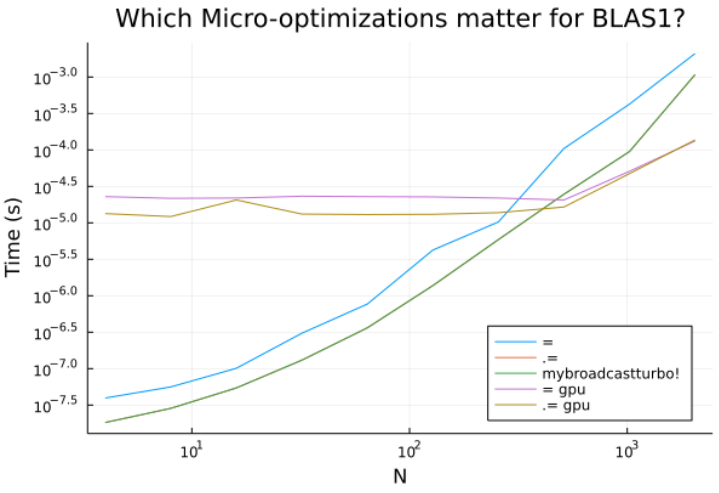



Big O(n^3) operations?
Just use a GPU
Don’t worry about overhead
You’re fine!

**Simplest code is ~3x from optimized**

# What happens when you specialize computations?

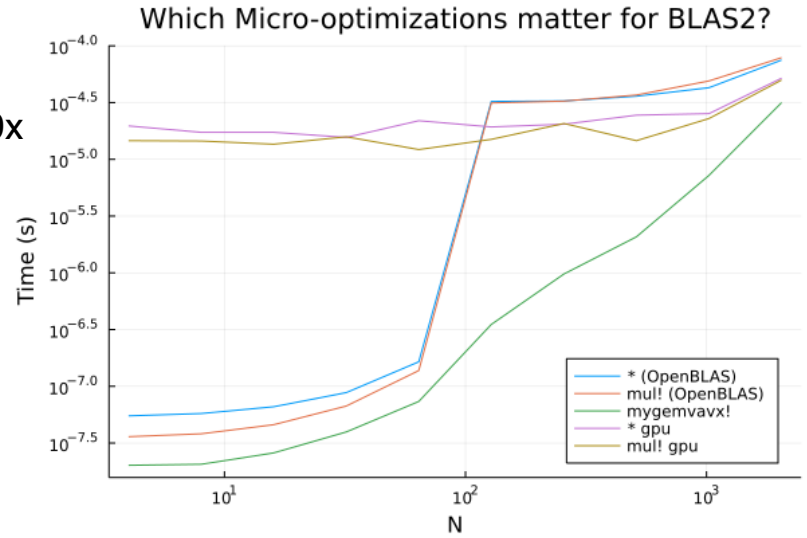



Scientific codes O(n) and O(n^2) operations

Mutation and Memory management: 10x

Manual SIMD: 5x

…




SimpleChains.jl

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

(10x Jax on CPU)

Details in the release blog post

Only for size ~100 layers and below!

# SimpleChains + StaticArray Neural ODEs

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

p_nn = SimpleChains.init_params(sc)

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

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

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

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

**About a 5x improvement**

**~1000x in a nonlinear mixed effects context**

**Tutorial should be up in a few days**

**Caveat: Requires sufficiently small ODEs (<20)**

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

# SciML Open Source Software Organization
# sciml.ai

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

**And 100 more libraries to mention…**

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

---

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