Surrogate Methods in Scientific Machine Learning and Extending Mathematical Compilation Techniques Beyond Differentiation

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

Idea of Surrogates: Anti-Amortize Compute Costs of Large-Scale Simulations

Pay Now, Use Later

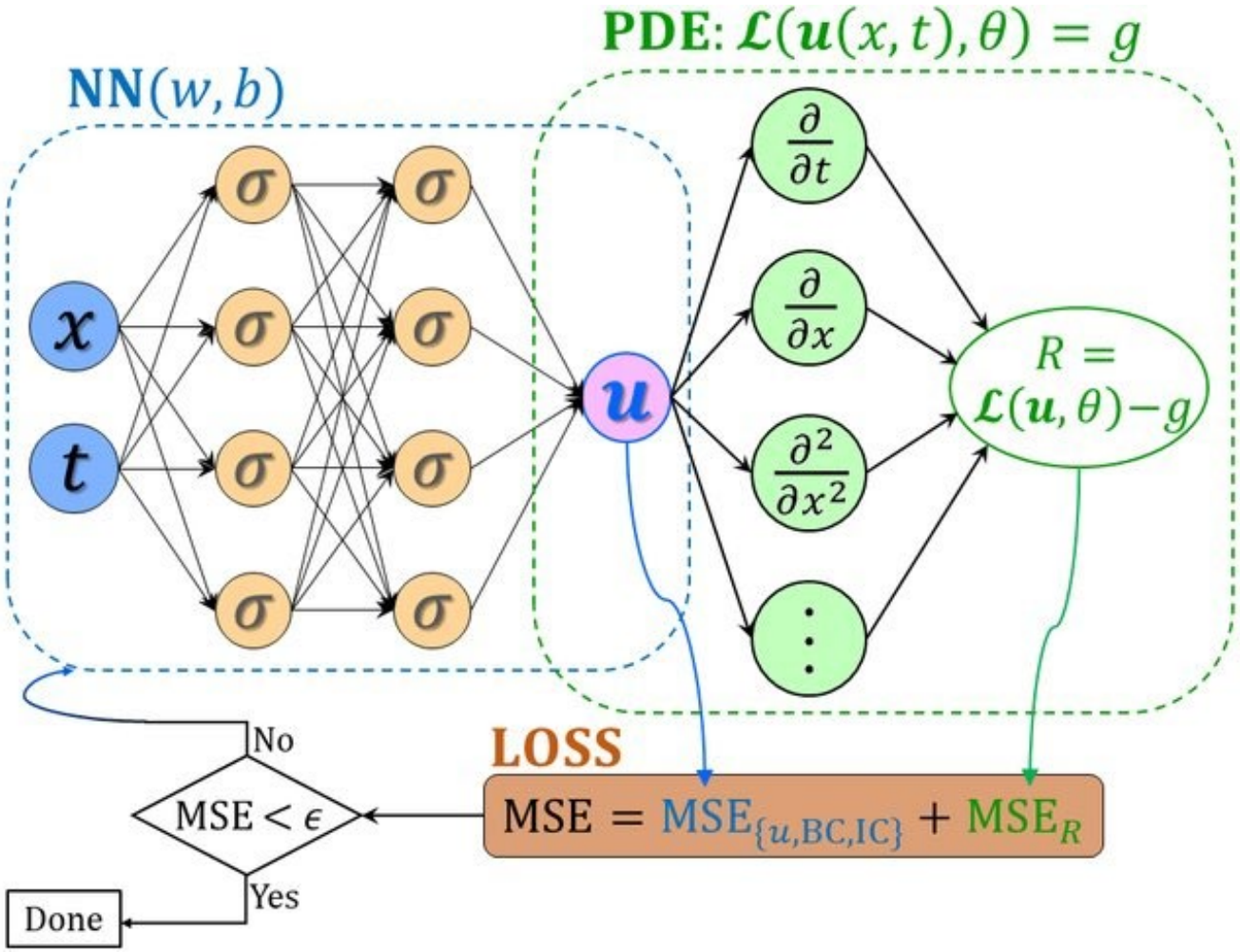
Example: control a drone with a sophisticated fluid dynamics model

Pre-solve everything before hand

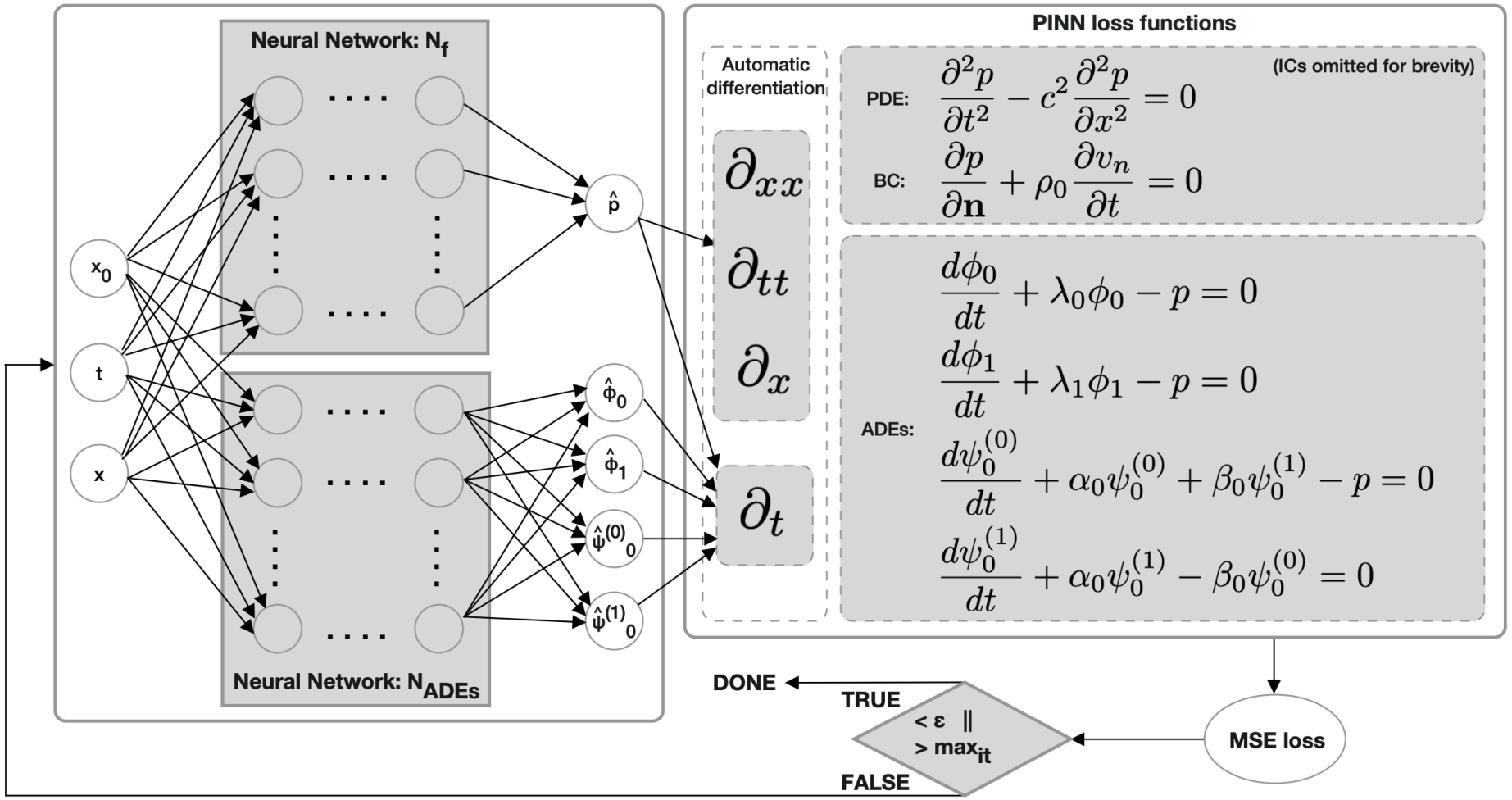
Put a neural network onto the drone that directly approximates the solution

Physics-Informed Neural Network Surrogates

Physics-Informed Neural Networks



Physics-Informed Neural Networks with Expanded Inputs



Kosambi-Karhunen-Loeve Theorem for Surrogates of Stochastic Differential Equations

Theorem. The variables Z_i have a joint Gaussian distribution and are stochastically independent if the original process $\{X_t\}_t$ is Gaussian.

In the Gaussian case, since the variables Z_i are independent, we can say more:

$$\lim_{N\to\infty} \sum_{i=1}^{N} e_i(t) Z_i(\omega) = X_t(\omega)$$

almost surely.

Idea: expand input space to include Z_i

Neural Operators

Motivation: Green's Functions

For a general PDE of the form:

$$(\mathcal{L}_a u)(x) = f(x), \quad x \in D$$

$$u(x) = 0, \quad x \in \partial D$$

Under fairly general conditions on $\mathcal{L}_a$, we may define the Green's function $G : D \times D \to \mathbb{R}$ as the unique solution to the problem

$$\mathcal{L}_a G(x, \cdot) = \delta_x$$

where δ_x is the delta measure on $\mathbb{R}^d$ centered at x. Note that G will depend on the coefficient a thus we will henceforth denote it as G_a. Then operator $\mathcal{F}_{true}$ can be written as an integral operator of green function:

$$u(x) = \int_D G_a(x, y) f(y)\, dy$$

Neural Operators

$$u(x) = \sigma\left(W v(x) + \int_D \kappa(x, y) v(y)\, \mathrm{d}\nu(y) + b(x)\right) \qquad \forall x \in D$$

Input: function v(x)
Output: function u(x)

Operation: Kernel integration

Neural Operators

Definition 1 (Iterative updates) *Define the update to the representation* $v_t \mapsto v_{t+1}$ *by*

$$v_{t+1}(x) := \sigma\Big(W v_t(x) + \big(\mathcal{K}(a;\phi)v_t\big)(x)\Big), \qquad \forall x \in D$$

Reasoning: Green's Function
requires linearity of the PDE.

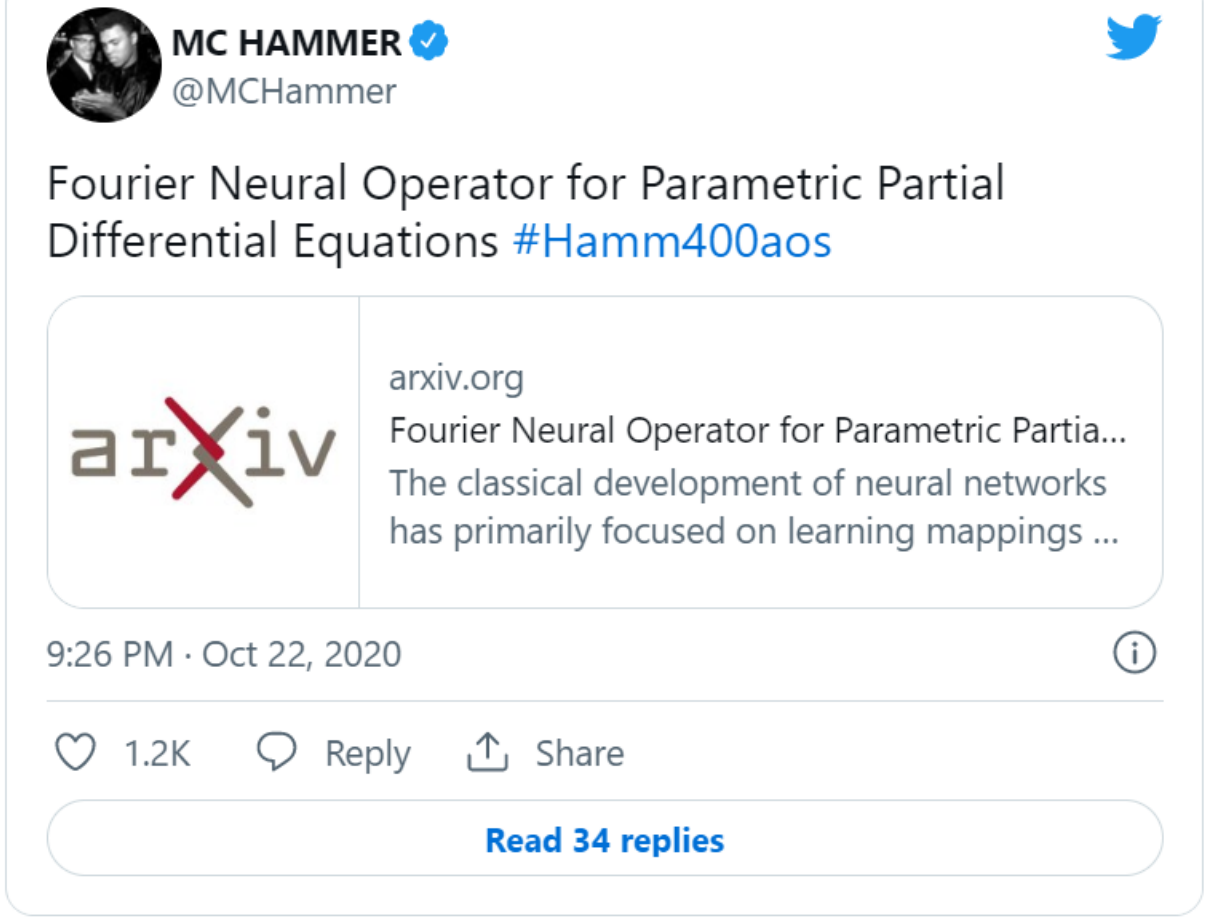
Use local linearity => time steps

Training Neural Operators

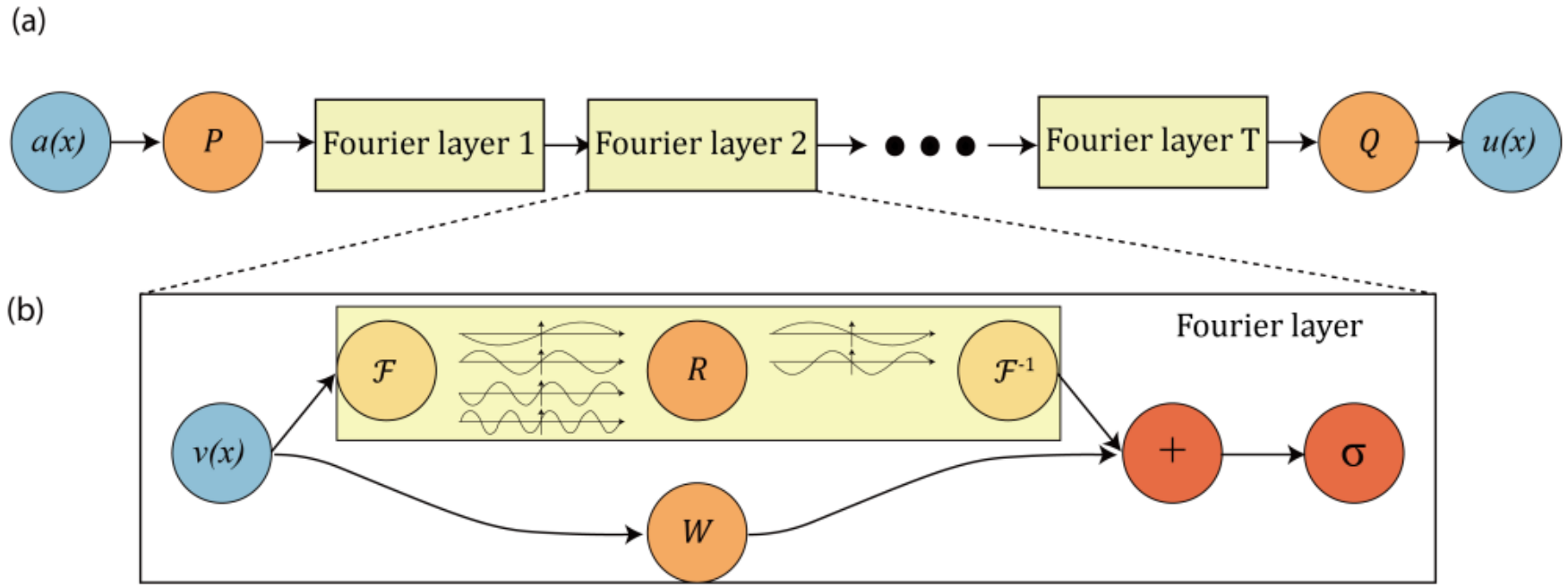
1. Generate data of PDE solution using classical method with many different inputs.
2. Train the neural operator to match the input/output behavior of the PDE solver

Note that a PDE solver is still required!!!

Fourier Neural Operators



Fourier Neural Operators



(a) The full architecture of neural operator: start from input a. 1. Lift to a higher dimension channel space by a neural network P. 2. Apply four layers of integral operators and activation functions. 3. Project back to the target dimension by a neural network Q. Output u. **(b) Fourier layers**: Start from input v. On top: apply the Fourier transform $\mathcal{F}$; a linear transform R on the lower Fourier modes and filters out the higher modes; then apply the inverse Fourier transform $\mathcal{F}^{-1}$. On the bottom: apply a local linear transform W.

Figure 2: **top:** The architecture of the neural operators; **bottom:** Fourier layer.

Fourier Transformations of Derivative Operators: Conv to Multiplication

$\Delta = \frac{d^2}{dx^2}$. The claim is that

$$\Delta = \begin{bmatrix} -(\pi)^2 & 0 & 0 \\ 0 & -(2\pi)^2 & \ddots \\ 0 & \ddots & \ddots \end{bmatrix} = D$$

is diagonal in the Fourier basis.

$$\frac{d^2}{dx^2} \sin(n\pi x) = -(n\pi)^2$$

and so

$$\begin{aligned} \frac{d^2}{dx^2} \sum_{n=1}^{\infty} b_n \sin(n\pi x) &= \sum_{n=1}^{\infty} b_n \frac{d^2}{dx^2} \sin(n\pi x) \\ &= -\sum_{n=1}^{\infty} (n\pi)^2 b_n \sin(n\pi x) \end{aligned}$$

Fourier Transformations of Derivative Operators: Conv to Multiplication

4.2 Solution to the Poisson Equation

$$\Delta u = f$$

Now diagonalize Δ. Notice that it is diagonal in the Fourier basis, and so we write the diagonalization of $\Delta = \mathcal{F}^{-1} D \mathcal{F}$ and get

$$\mathcal{F}^{-1} D \mathcal{F} u = f$$

$$u = \mathcal{F}^{-1} D^{-1} \mathcal{F} f.$$

Fourier Neural Operators

with $i = \sqrt{-1}$ the imaginary unit. By letting $\kappa(x, y) = \kappa(x - y)$ for some $\kappa : D \to \mathbb{C}^{m \times n}$ in (16) and applying the convolution theorem, we find that

$$u(x) = \mathcal{F}^{-1}\big(\mathcal{F}(\kappa) \cdot \mathcal{F}(v)\big)(x) \qquad \forall x \in D.$$

We therefore propose to directly parameterize κ by its Fourier coefficients. We write

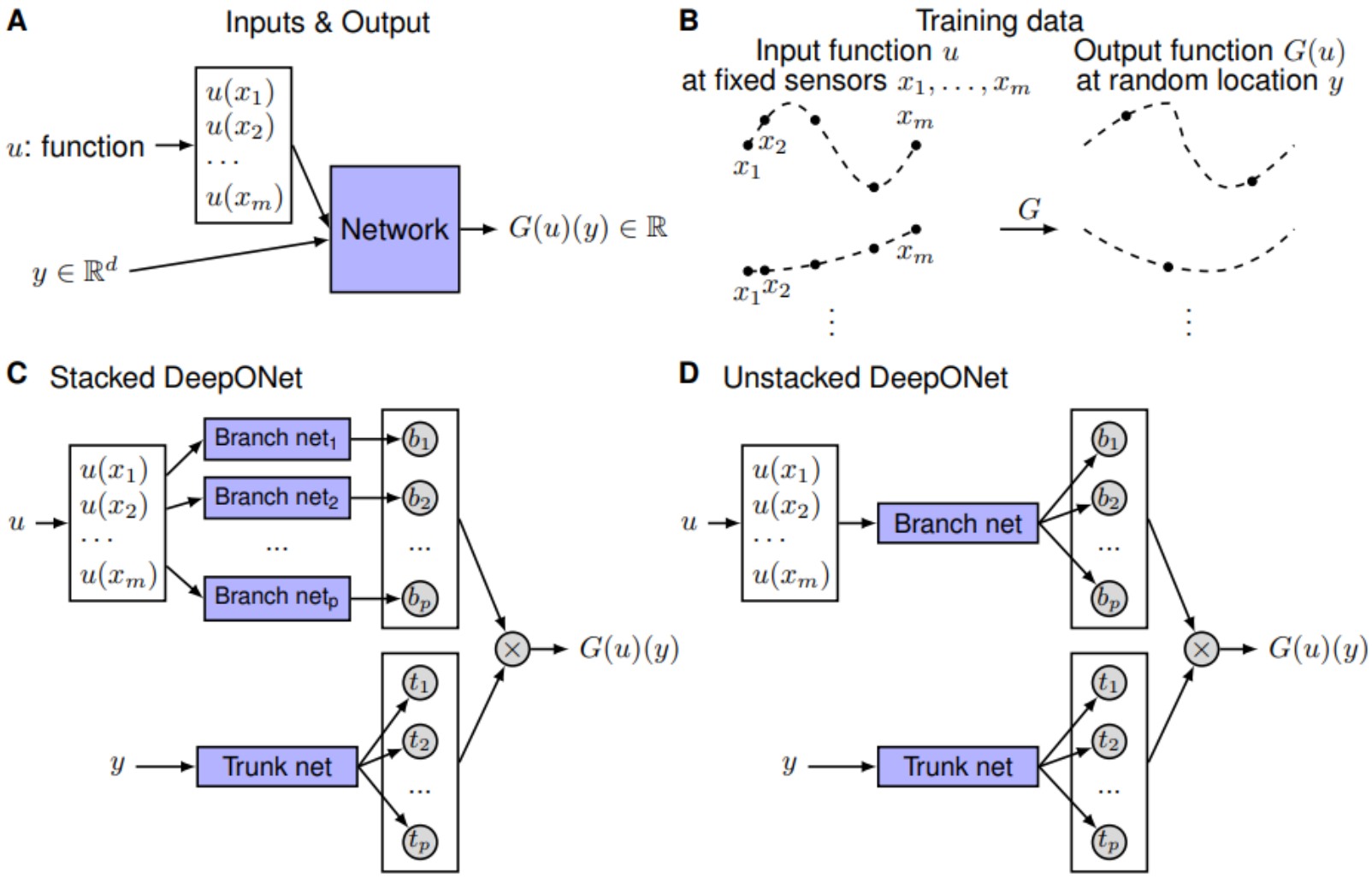
$$u(x) = \mathcal{F}^{-1}\big(R_\phi \cdot \mathcal{F}(v)\big)(x) \qquad \forall x \in D \tag{28}$$

Fourier Neural Operators with NeuralOperators.jl

Fourier Neural Operator

```
model = Chain(
    # lift (d + 1)-dimensional vector field to n-dimensional vector field
    # here, d == 1 and n == 64
    Dense(2, 64),
    # map each hidden representation to the next by integral kernel operator
    OperatorKernel(64=>64, (16, ), FourierTransform, gelu),
    OperatorKernel(64=>64, (16, ), FourierTransform, gelu),
    OperatorKernel(64=>64, (16, ), FourierTransform, gelu),
    OperatorKernel(64=>64, (16, ), FourierTransform),
    # project back to the scalar field of interest space
    Dense(64, 128, gelu),
    Dense(128, 1),
)
```

DeepONet


A
Inputs & Output
u: function
$u(x_1)$
$u(x_2)$
...
$u(x_m)$
Network
$G(u)(y) \in \mathbb{R}$
$y \in \mathbb{R}^d$
B
Training data
Input function u
at fixed sensors $x_1, \ldots, x_m$
Output function $G(u)$
at random location y
G
C
Stacked DeepONet
u
Branch net$_1$
Branch net$_2$
Branch net$_p$
b_1
b_2
b_p
y
Trunk net
t_1
t_2
t_p
$G(u)(y)$
D
Unstacked DeepONet
Branch net
Trunk net

DeepONet as a Linear Basis Representation

we propose is shown in Fig. 1C, and the details are as follows. First there is a "trunk" network, which takes y as the input and outputs $[t_1, t_2, \ldots, t_p]^T \in \mathbb{R}^p$. In addition to the trunk network, there are p "branch" networks, and each of them takes $[u(x_1), u(x_2), \ldots, u(x_m)]^T$ as the input and outputs a scalar $b_k \in \mathbb{R}$ for $k = 1, 2, \ldots, p$. We merge them together as in Eq. 1:

$$G(u)(y) \approx \sum_{k=1}^{p} b_k t_k.$$

The standard DeepONet structure is a linear approximation of the target operator, where the trunk net and branch net learn the coefficients and basis. On the other hand, the neural operator is a non-linear approximation, which makes it constructively more expressive

DeepONets with NeuralOperators.jl

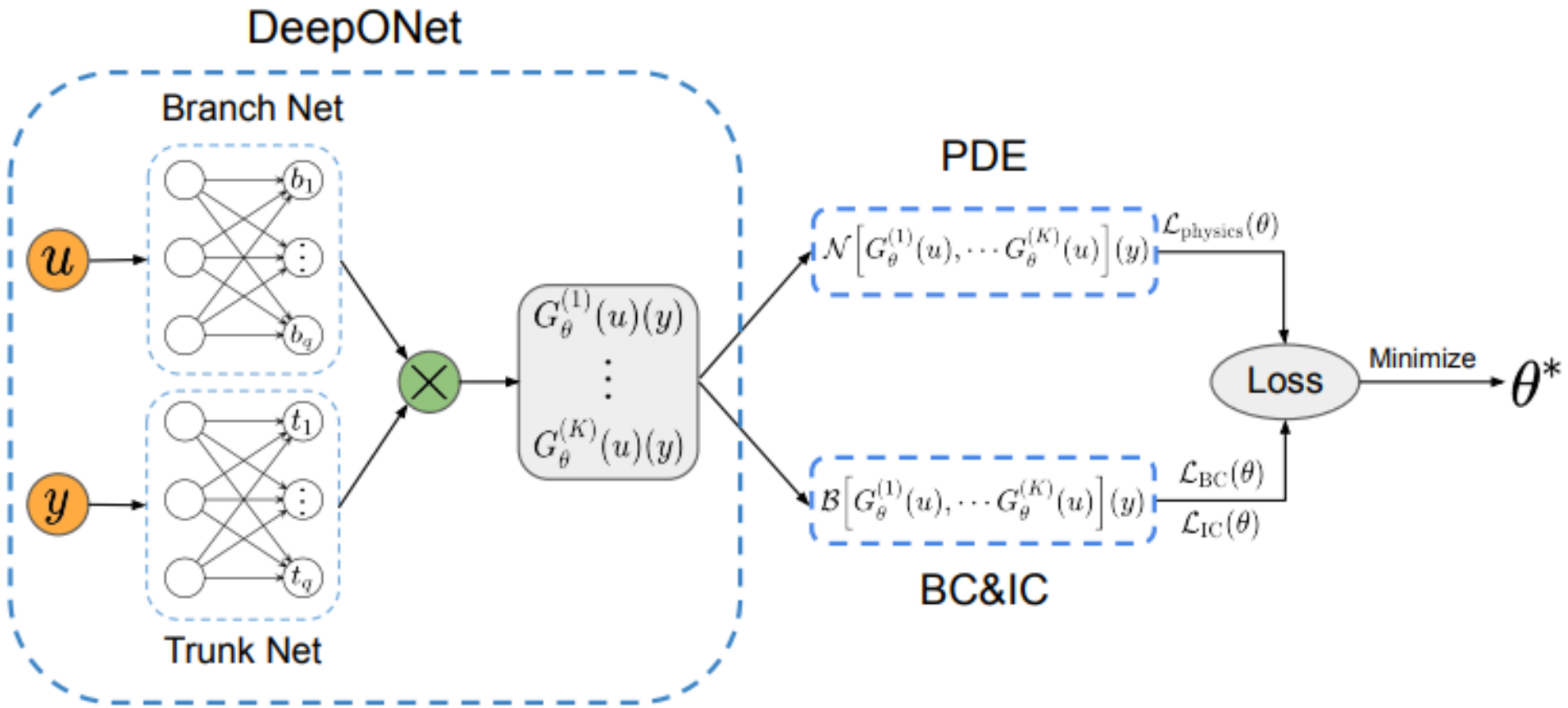
DeepONet

```
# tuple of Ints for branch net architecture and then for trunk net,
# followed by activations for branch and trunk respectively
model = DeepONet((32, 64, 72), (24, 64, 72), σ, tanh)
```

Or specify branch and trunk as separate `Chain` from Flux and pass to `DeepONet`

```
branch = Chain(Dense(32, 64, σ), Dense(64, 72, σ))
trunk = Chain(Dense(24, 64, tanh), Dense(64, 72, tanh))
model = DeepONet(branch, trunk)
```

Physics-Informed DeepONets



Physics-Informed DeepONets

Training: Table 4 summarizes the computational cost (hours) of training different models The size of different models as well as network architectures are listed Table 3. All networks are trained using a single A100 card. It can be observed that training a physics-informed DeepONet model is generally slower than training a conventional DeepONet. This is expected as physics-informed DeepONets require to compute the PDE residual via automatic differentiation, yielding a lager computational graph, and, therefore, a higher computational cost.

Case	Model	Training time (hours)
Gravity pendulum	Physics-informed neural network	0.12
	Physics-informed DeepONet	1.63
Linear ODE	Physics-informed DeepONet	1.33
Stiff ODE	Physics-informed DeepONet	7.60
Wave equation	Physics-informed DeepONet	3.00
Diffusion-reaction equation	Physics-informed DeepONet	1.48
KDV equation	Physics-informed DeepONet	2.17
	DeepONet	0.35

Table 4: Computational cost (hours) for training different models across the different benchmarks and architectures employed in this work. Reported timings are obtained on a single Nvidia V100 GPU.

Physics-Informed Neural Operators

Method	# data samples	# additional PDE instances	Solution error (w)	Time cost
PINNs	-	-	18.7%	4577s
PINO	0	0	**0.9%**	608s
PINO	4k	0	**0.9%**	536s
PINO	4k	160k	**0.9%**	**473s**

Table 2: Equation-solving on Kolmogorov flow $Re = 500$.

Reservoir Computing Methods

Origins

During the training of Recurrent Neural Networks (RNN) was observed that only the weights in the last layer presented the most changes during training [1]

The idea that immediately came to mind was to only train the last layer

As a result the following models were proposed:

- **Echo State Networks** (ESN) [2]
- **Liquid State Machines** (LSM) [3]

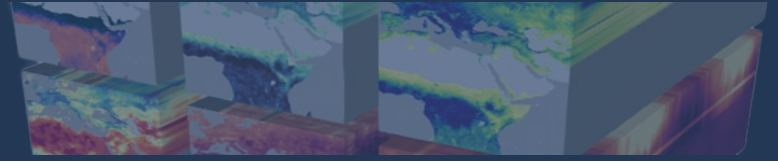
Training - ESN

All the input points are passed in the reservoir and the corresponding states $\mathbf{x}(t)$ are collected into the state matrix ($\mathbf{X} \in R^{N \times T}$) over the training period n=1,…,T. The evolution of the states is governed by [5]:

$$\mathbf{x}(t+1)=(1-\alpha)\cdot\mathbf{x}(t) + \alpha\cdot f[\mathbf{W}\mathbf{x}(t)+\mathbf{W}_{in}\mathbf{u}(t)]$$

At every input point corresponds an output vector, collected into a matrix $\mathbf{V}_{target} \in R^{L \times T}$. The output layer ($\mathbf{W}_{out}$) is computed at the end as linear regression of the teacher output on the reservoir states. The most used approach is Ridge Regression [6]:

$$\mathbf{W}_{out} = \mathbf{V}_{target}\mathbf{X}^T(\mathbf{X}\mathbf{X}^T + \beta\mathbf{I})^{-1}$$



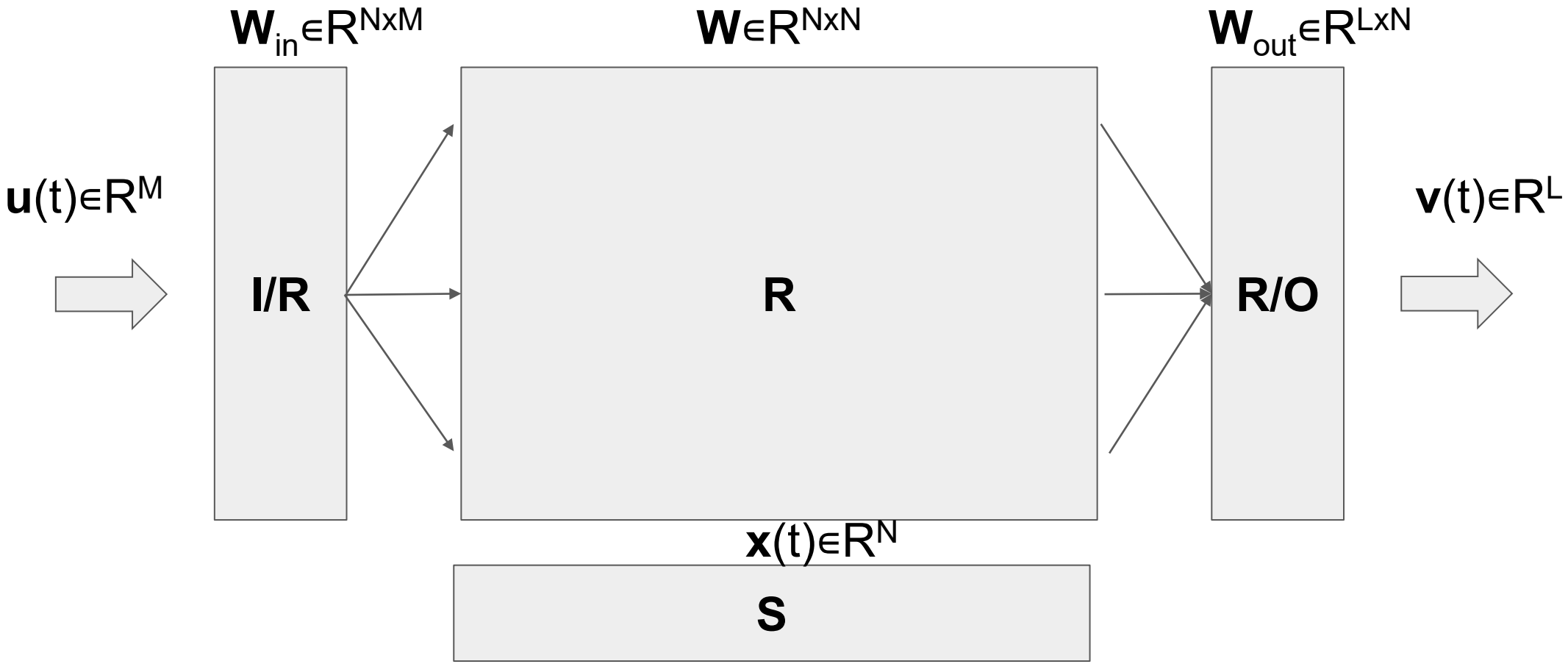
Architecture - ESN

Input layer ($\mathbf{W}_{in}$) and reservoir ($\mathbf{W}$) are fixed at the start and do not change during training. The output layer ($\mathbf{W}_{out}$) is computed in one shot at the end of the training

$\mathbf{W}_{in}$ is a dense random matrix, with weight values sampled from a uniform distribution over $(-\sigma,\sigma)$

$\mathbf{W}$ is built from a sparse Erdos-Revy matrix, with weight values sampled from an uniform distribution over (-1, 1).
All the elements are also rescaled so that the spectral radius $\rho < 1$ [4].

General Structure

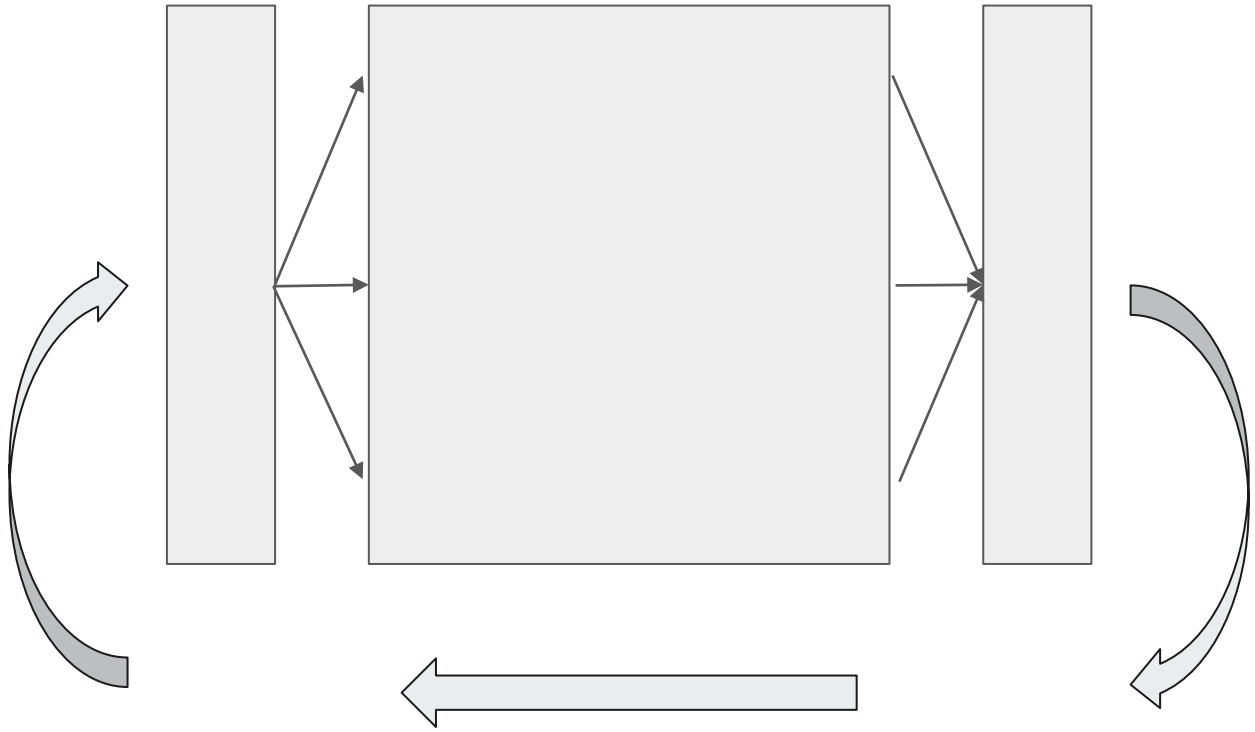


Prediction - ESN

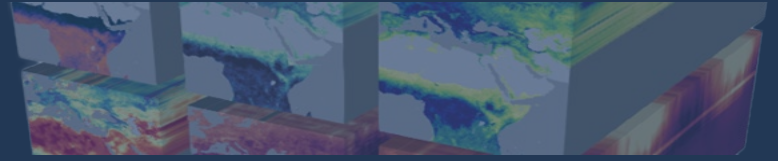
Using the setup described it is possible to obtain predictions using the following equations

$$\mathbf{v}(t+1)=g[\mathbf{W}_{out}\mathbf{x}(t)]$$

$$\mathbf{x}(t+1)=(1-\alpha)\cdot\mathbf{x}(t) + \alpha\cdot f[\mathbf{W}\mathbf{x}(t)+\mathbf{W}_{in}\mathbf{v}(t)]$$



Fully autonomous system



Reservoir Properties

- To sustain a universal computation a good reservoir needs to exhibit **complex** and **state rich** configuration space
- Systems with increasing degree of **complexity** will be better reservoir
- Better results are obtained when the system is working at the **edge of chaos**

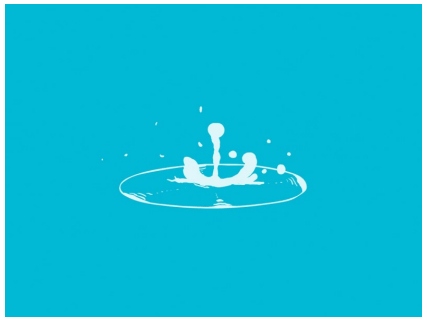
Echo State Property [7]:
the current state depends on the sequence of inputs it has been exposed to in a unique way

Fading Memory Property [2]:
the influence of the initial conditions of the dynamics weakens with time.

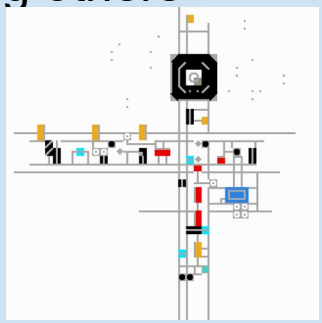
What can be used as a Reservoir?

Reservoir:
Non-linear dynamical system

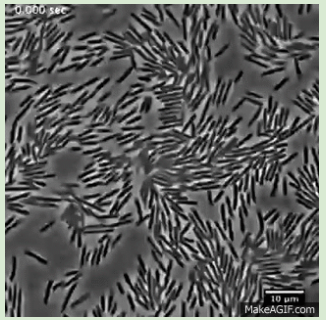
Bucket of Water:
Using different perturbations as input can obtain good classification results [8]

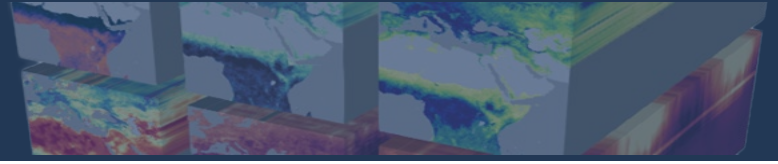


Analog Circuits:
A physical reservoir approach has also been explored in [9], among others



Bacteria:
Even E. Coli has been used as a reservoir in [10], obtaining decent results





Why is Reservoir Computing useful?

- Computationally inexpensive (Compared to standard Deep Learning approaches)
- More stable: does not show the vanishing gradient problem, typical Neural Network hurdle
- Engineering freedom: one can just pick a dynamical system as reservoir and start doing predictions
- Well suited for the prediction of time series in general, and more specifically chaotic systems. Outperforms state of the art models [11]

Prediction of Time Series

The standard approach with time series using Recurrent Neural Networks (**RNNs**) or Long Short Term Memory (**LSTM**) networks has been to feed the model an encoding of n prior time steps as training for the desired output.

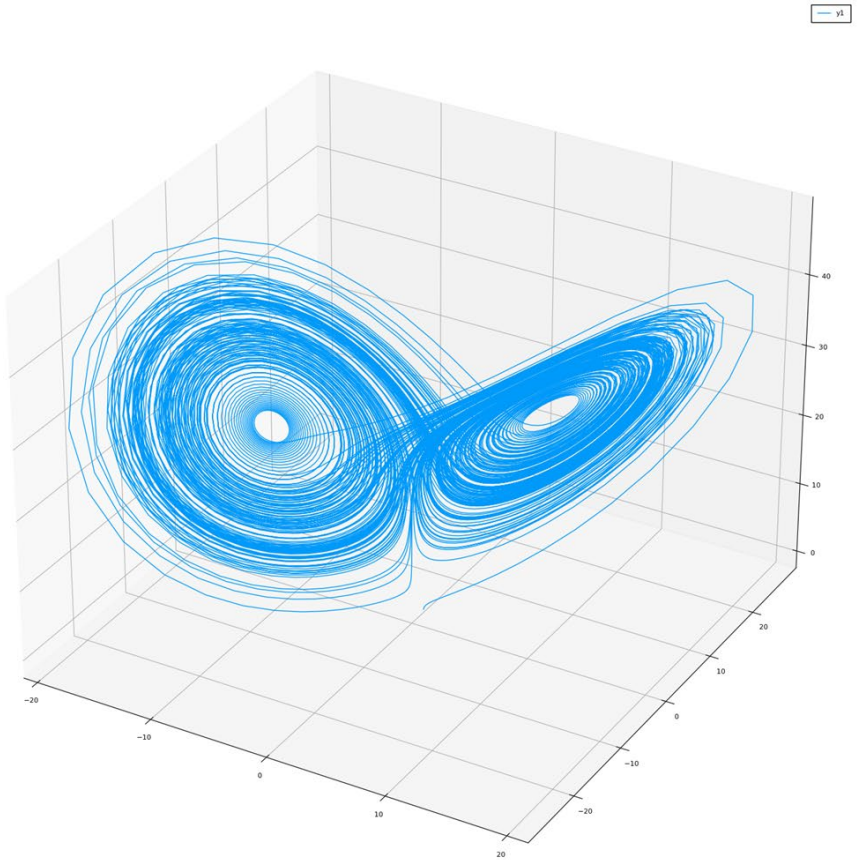
As we saw in the training section, no prior manipulation of the time series is needed for a Reservoir Computing approach.

[x(t-n), x(t-n-1),..., x(t-1)] → x(t)

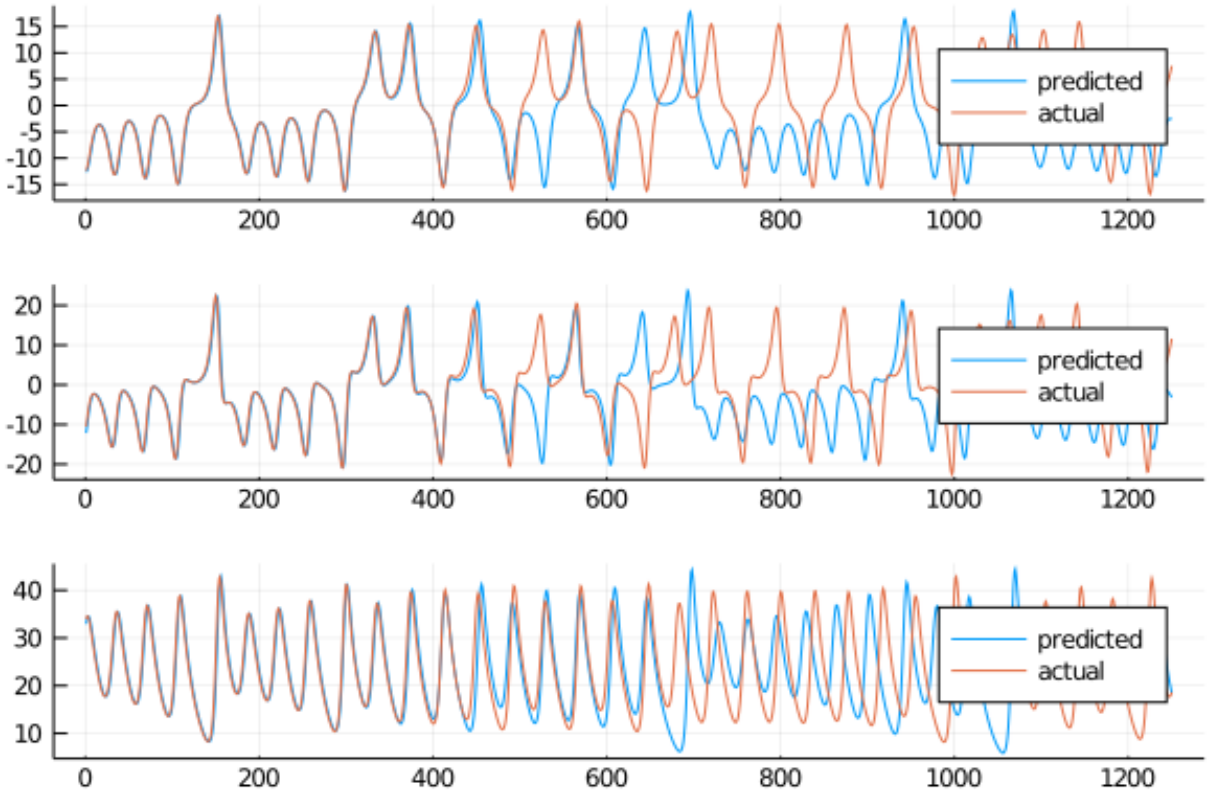
Example: Lorenz system

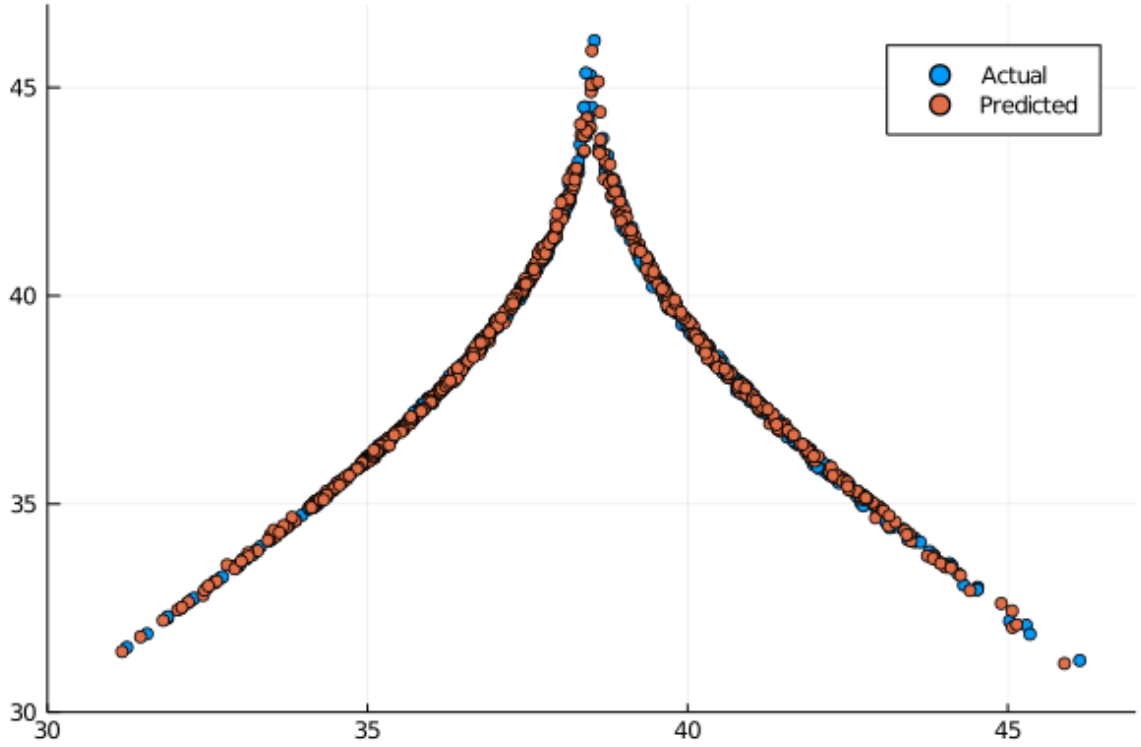
Studied by Edward Lorenz [12] is known to have **chaotic** solutions for certain parameter values

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$



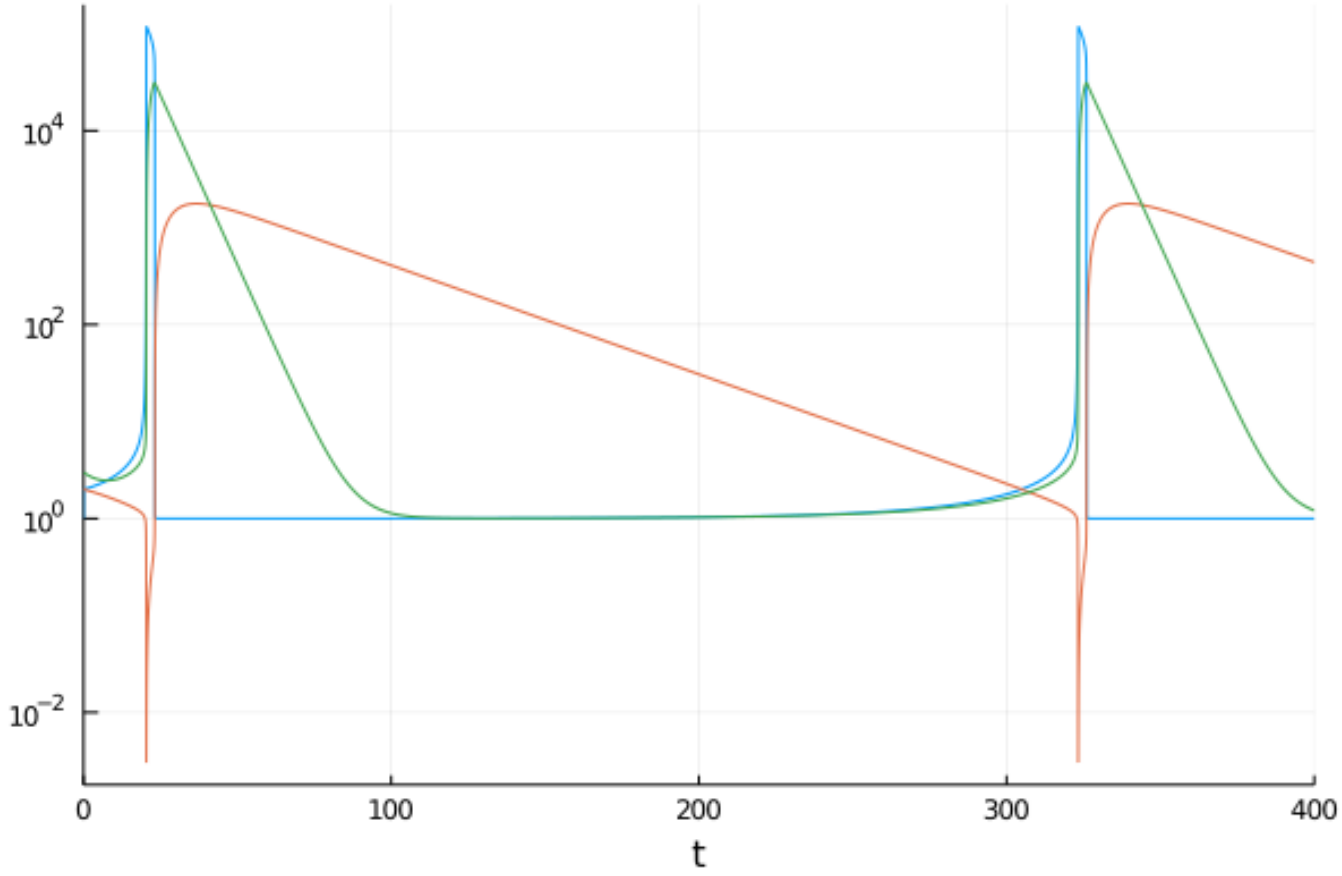
ESN Results

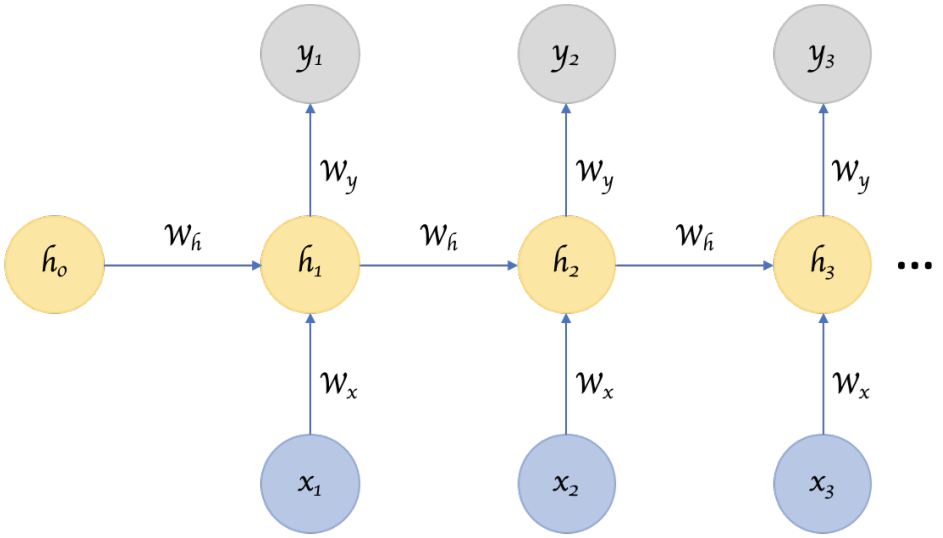

predicted
actual
predicted
actual
predicted
actual


Actual
Predicted

Continuous Time Reservoir Computing

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





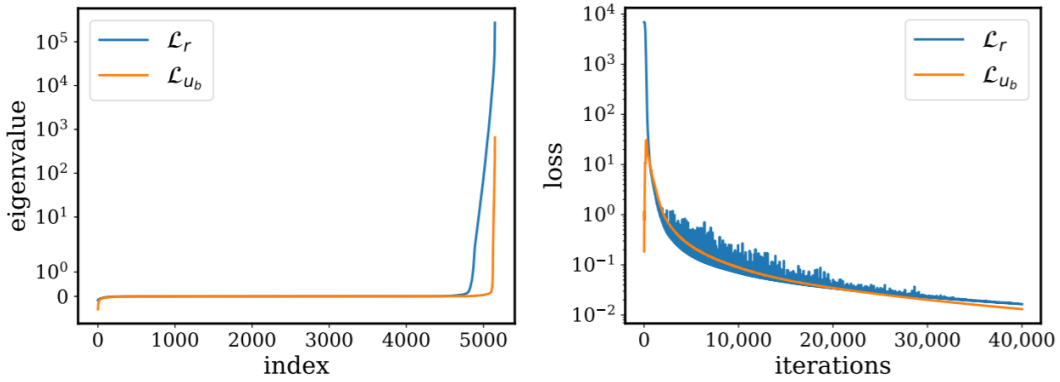
Recurrent neural network? No!

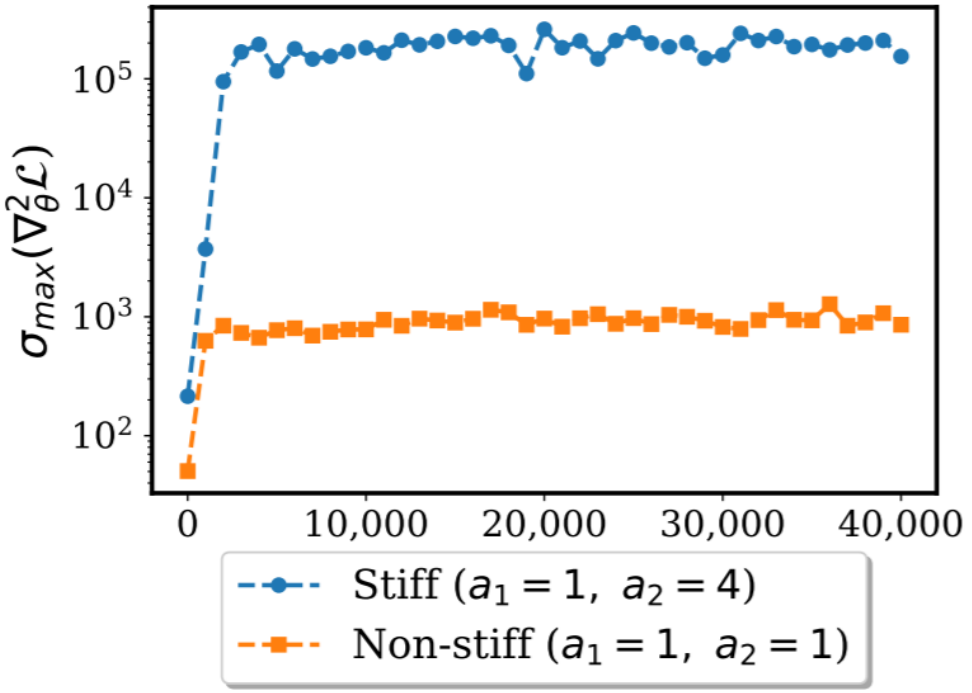
1. It's an explicit method! (Euler's)
2. Uniform steps will not capture the spikes!

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!



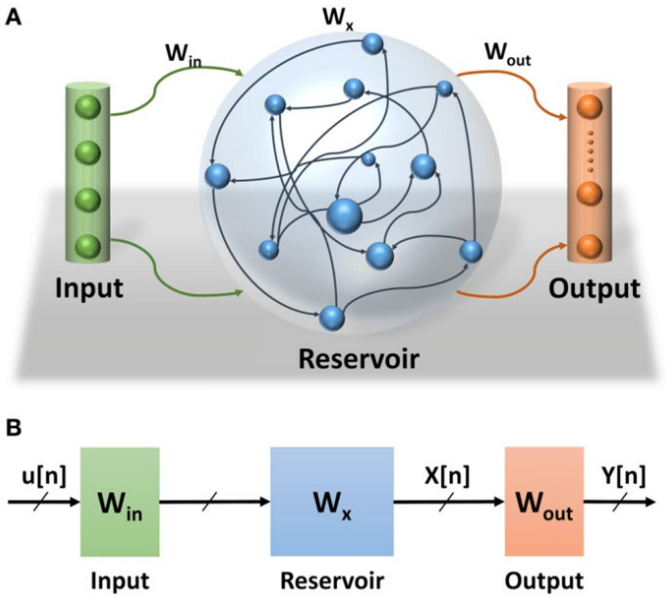


Understanding and mitigating gradient pathologies in physics-informed neural networks

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)

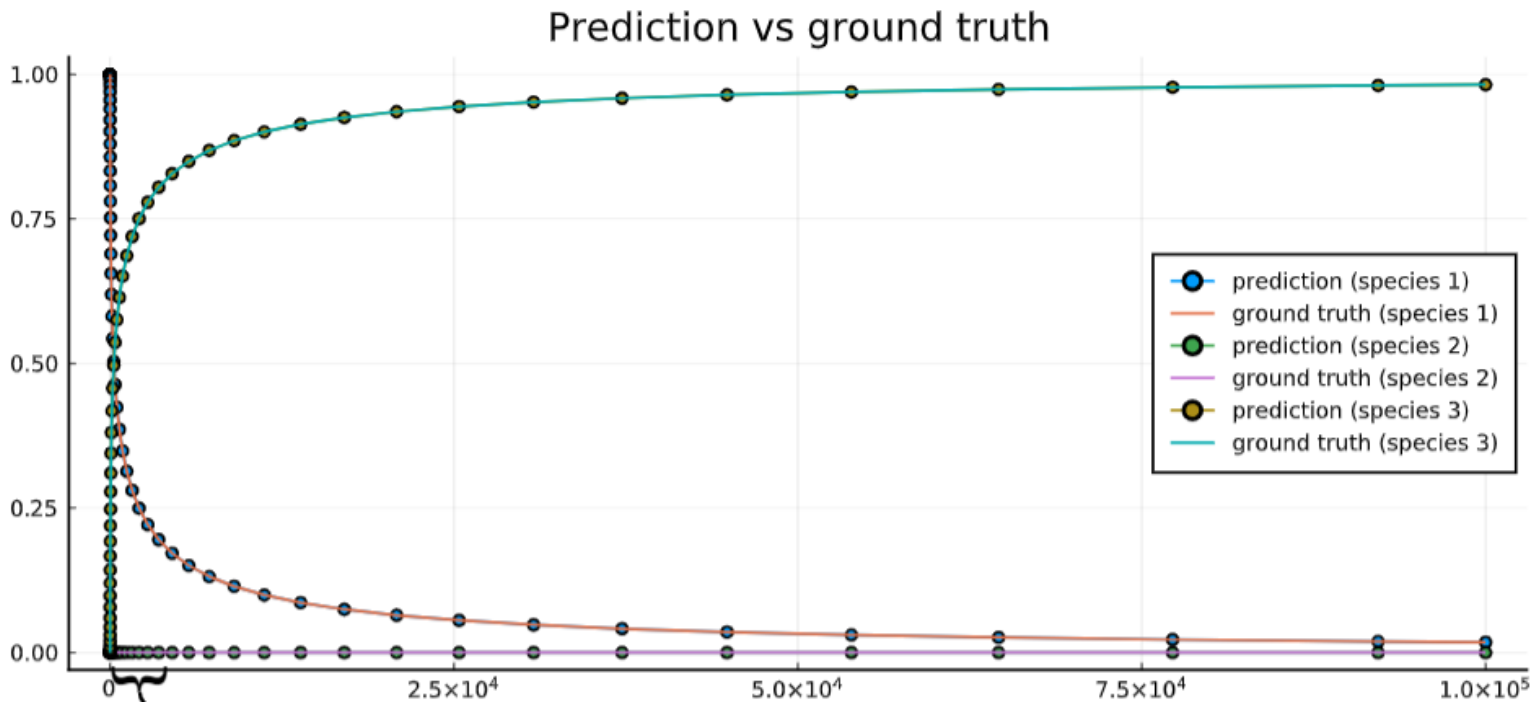
Get W_{out} at many parameters of the system

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

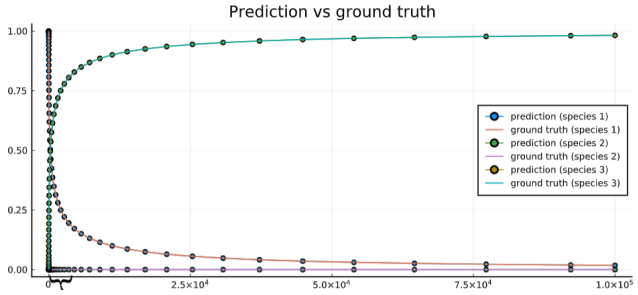
Classic stiff ODE
Used to test and break integrators
Volatile early transient

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

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

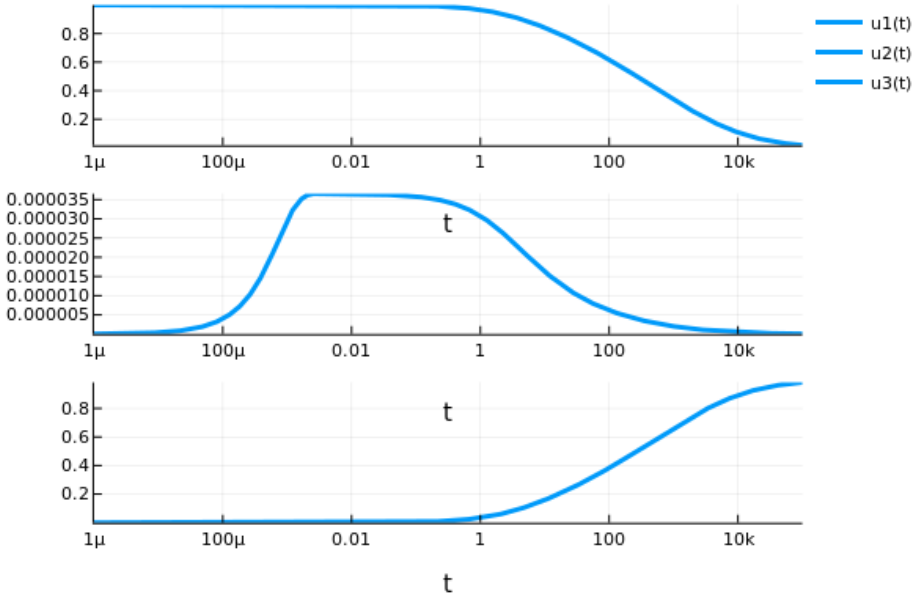
Ranjan Anantharaman, Yingbo Ma, Shashi Gowda, Chris Laughman, Viral Shah, Alan Edelman, Chris Rackauckas

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

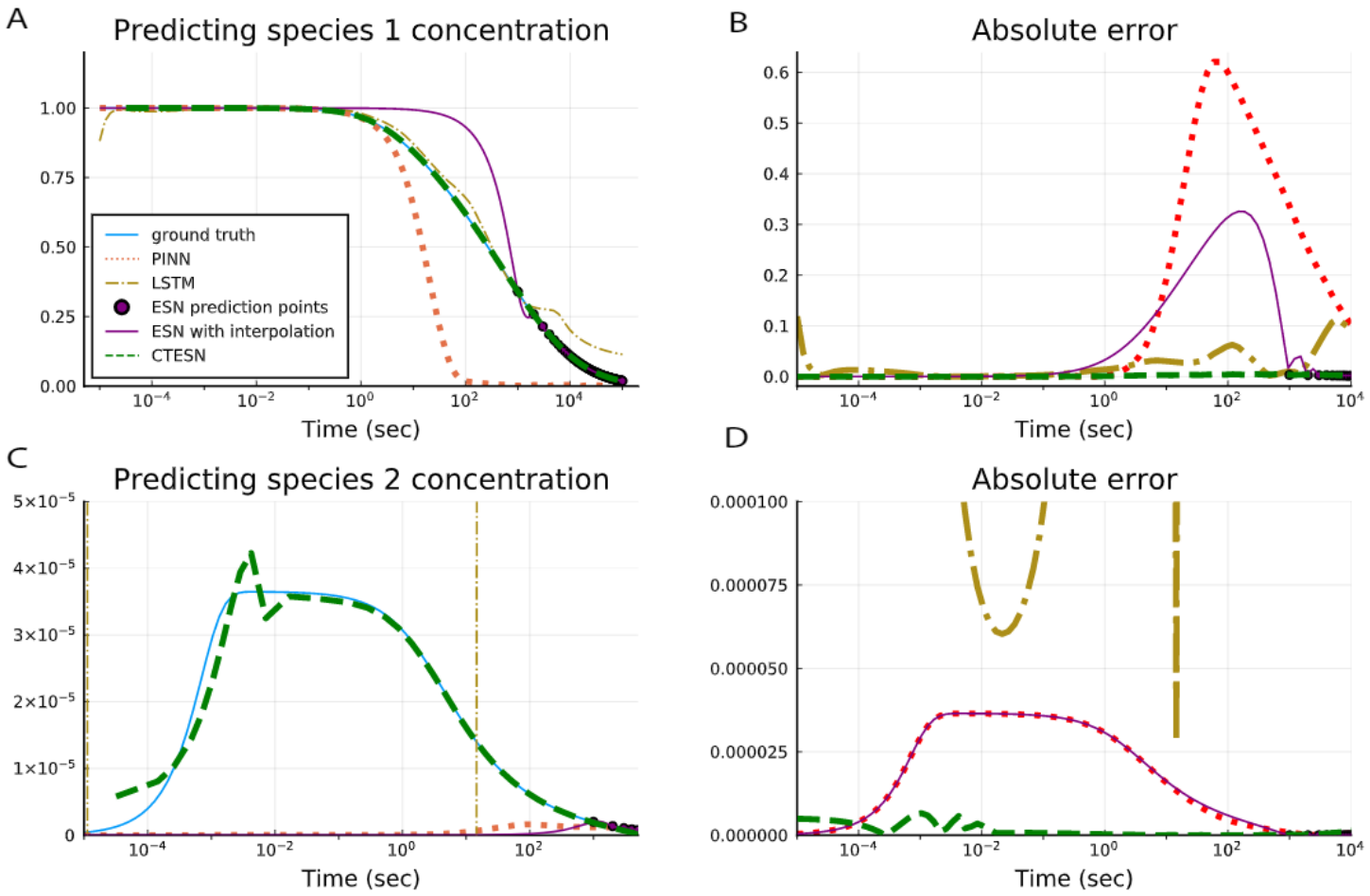
Classic stiff ODE
Used to test and break integrators
Volatile early transient

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

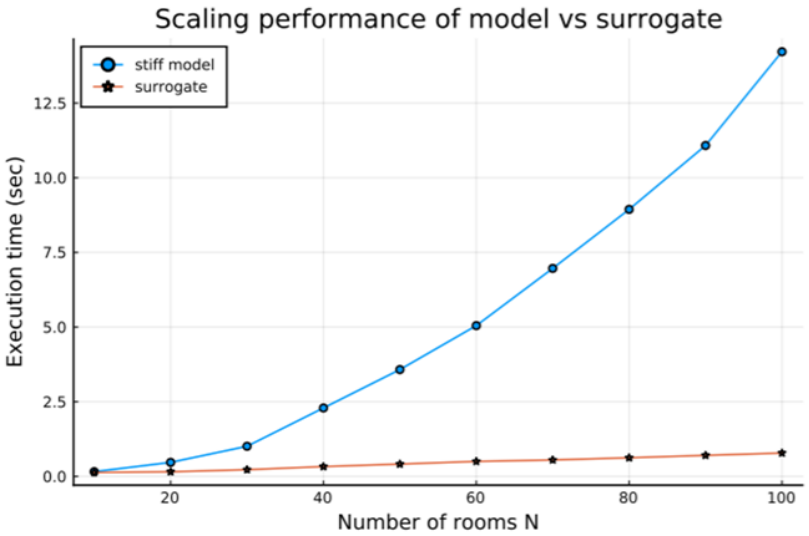
Accelerating Simulation of Stiff Nonlinear Systems using Continuous-Time Echo State Networks

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!

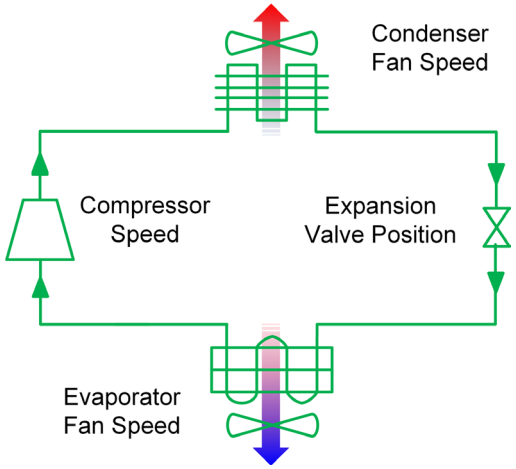
Accelerating Simulation of Stiff Nonlinear Systems using Continuous-Time Echo State Networks

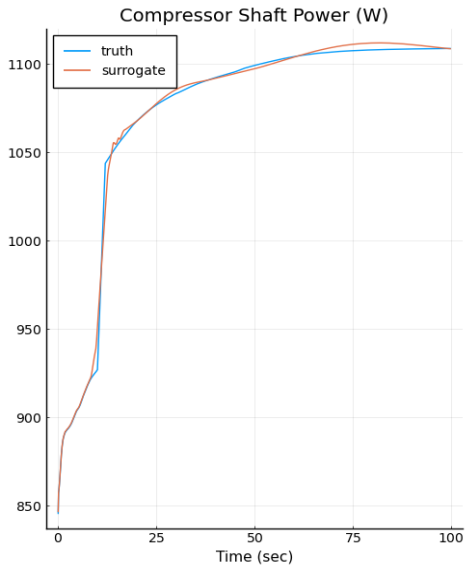
Ranjan Anantharaman, Yingbo Ma, Shashi Gowda, Chris Laughman, Viral Shah, Alan Edelman, Chris Rackauckas

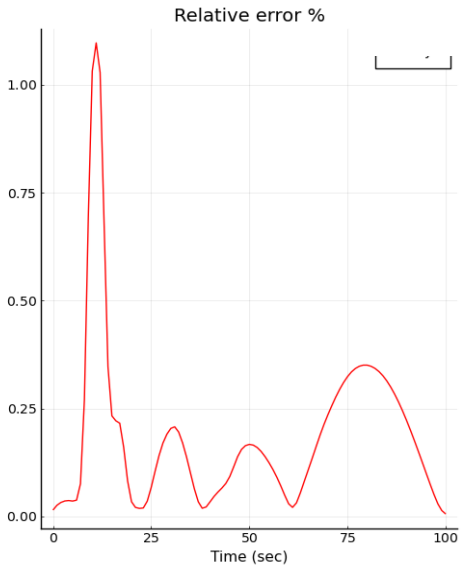
ARPA-E: Accelerated Simulation of Building Energy Efficiency

8,000 ODE Highly stiff vapor-compression cycle model

MITSUBISHI ELECTRIC
Changes for the Better







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

BioModels

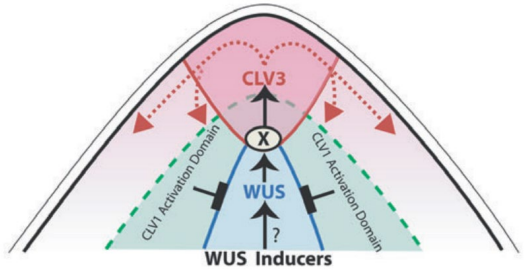
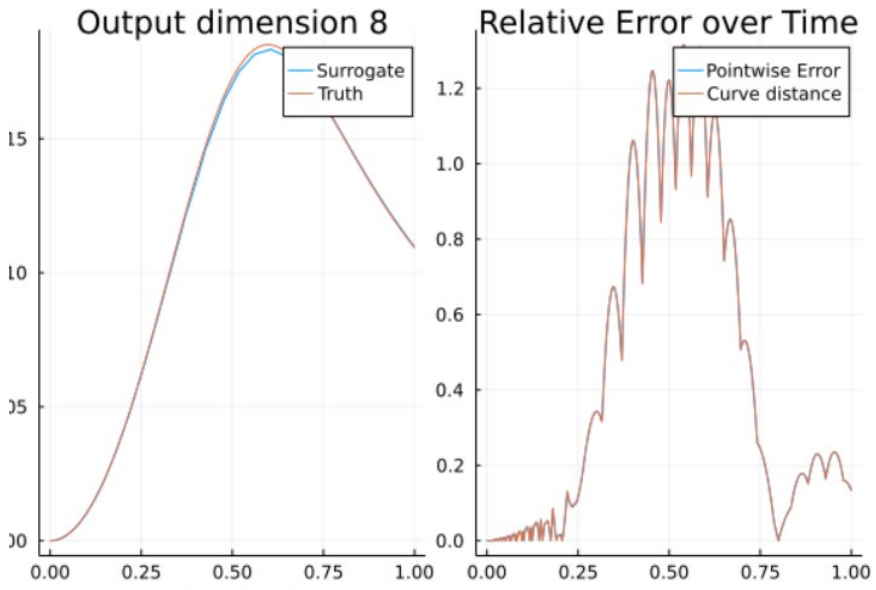


Fig. 1. A schematic of the expression domains of *CLAVATA1*, *CLAVATA3* and *WUSCHEL*. The solid arrows indicate the regulatory (indirect) interactions and the dashed arrows show the movement of the CLV3 protein.



- 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

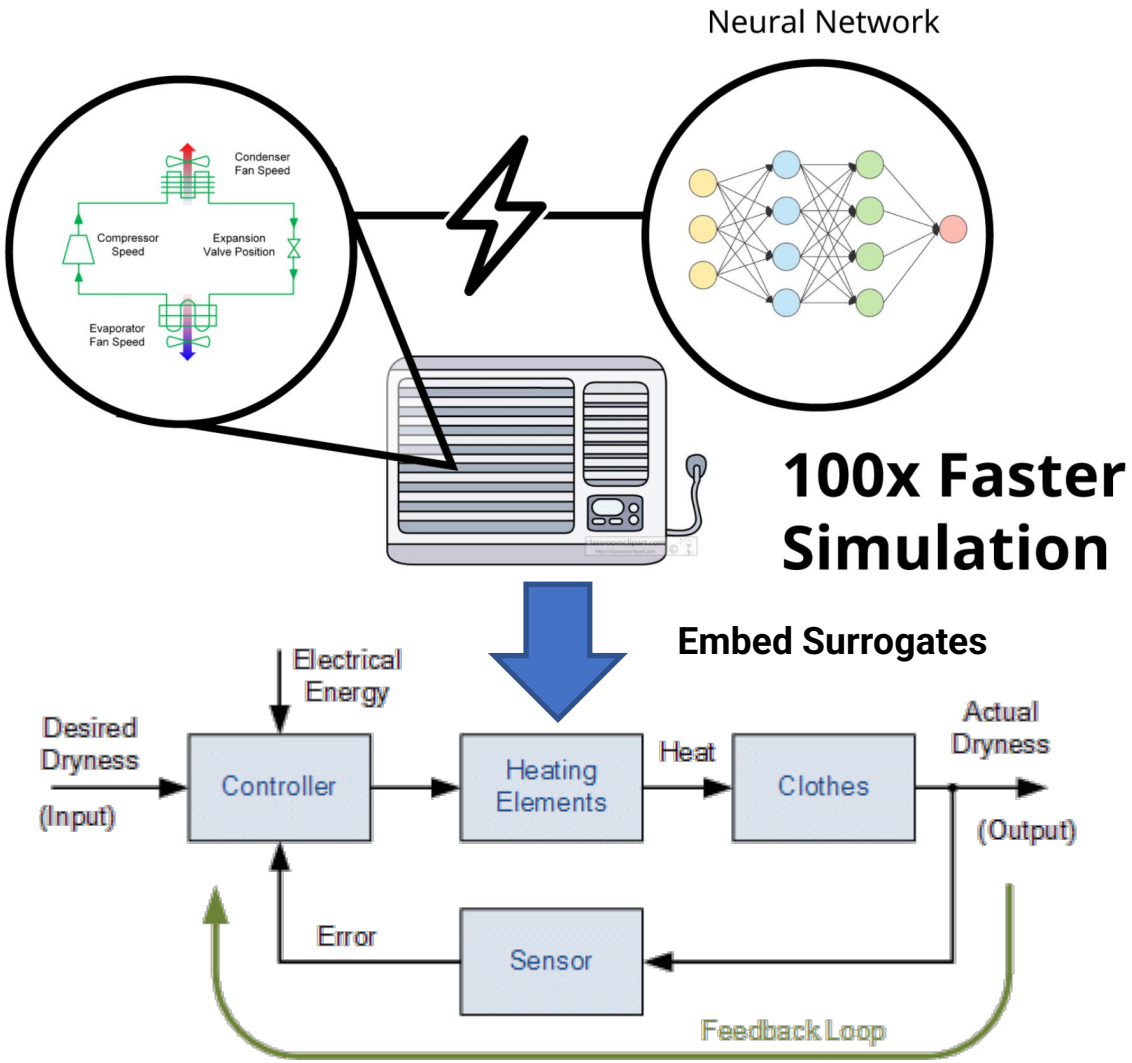
Julia vanilla outperforms MATLAB’s SBMLToolbox

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)

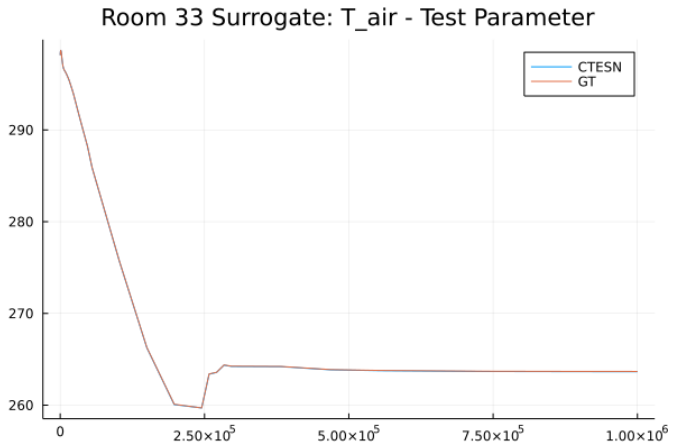
Total speedup: 100x vs MATLAB SBMLToolbox

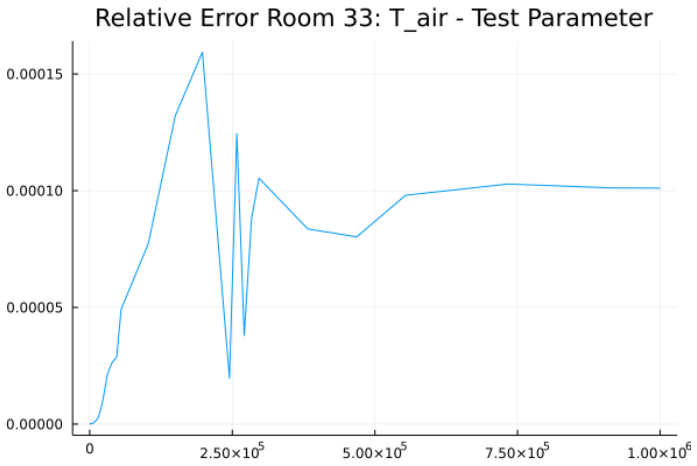
The Transformed Models are Just Components: Compose As Normal



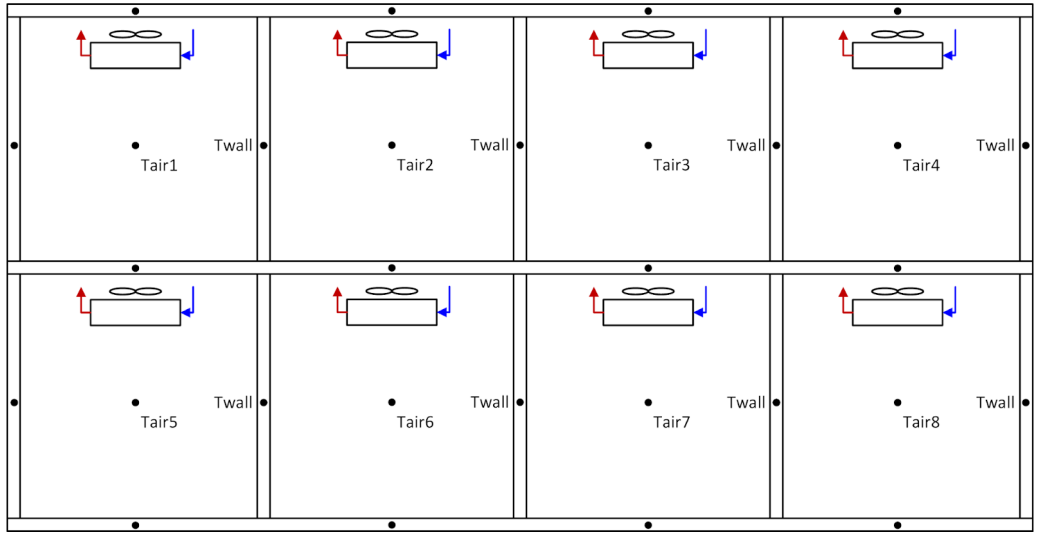
Accelerate large (100,000 ODE) simulations without retraining by using an accelerated HVAC component inside of different building models

Large Building Models 100K Equations, 80x Acceleration

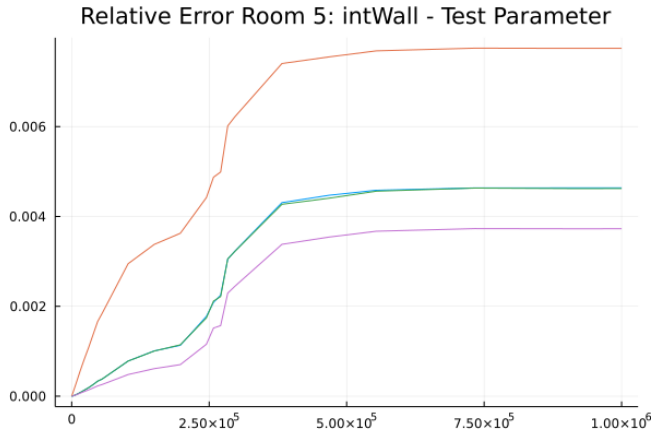


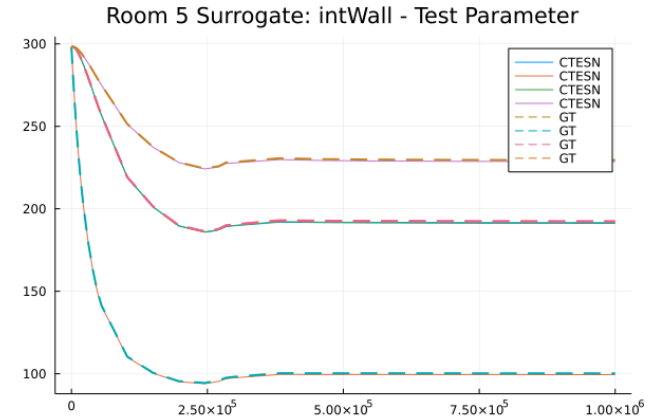


Rooms Disturbed	Training set size	Reservoir size	Prediction time	Speedup over baseline
1	100	200	0.2597 s	77x
3	100	200	0.413s	80x



Scalable building model with equipment





Total speedup over original : 80x

Extended Mathematical Compiler Transformations: Abstract Interpretation More than Differentiation

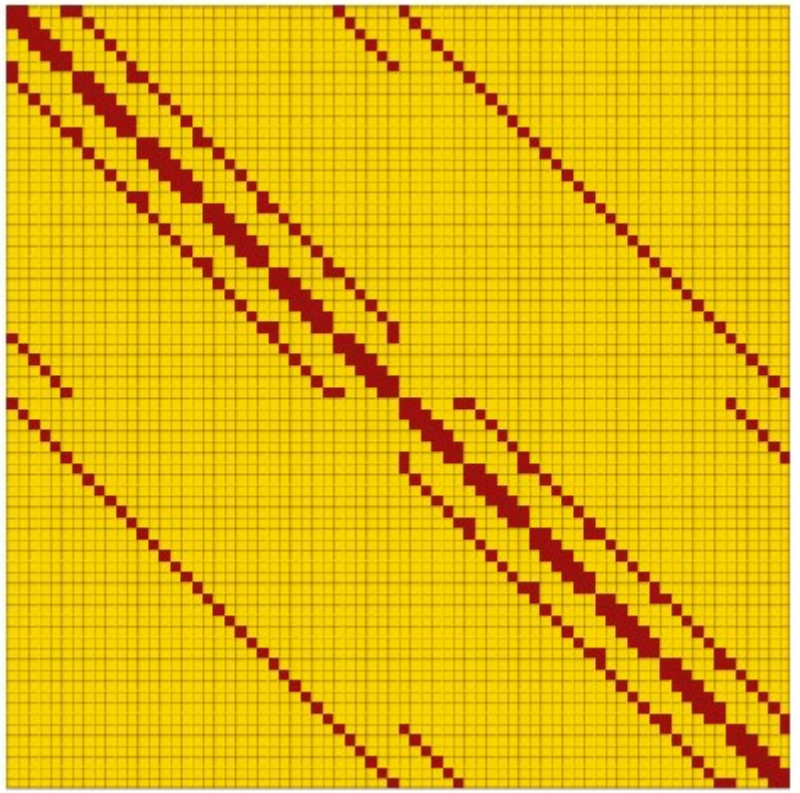
Automated Sparsity Detection: SparsityDetection.jl and ModelingToolkit.jl

```
brusselator_f(x, y, t) = ifelse((((x-0.3)^2 + (y-0.6)^2) <= 0.1^2) && (t >= 1.1), 5., 0.)
limit(a, N) = a == N+1 ? 1 : a == 0 ? N : a
function brusselator_2d_loop(du, u, p, t)
  A, B, alpha, xyd, dx, N = p; alpha = alpha/dx^2
  @inbounds for I in CartesianIndices((N, N))
      i, j = Tuple(I)
      x, y = xyd[I[1]], xyd[I[2]]
      ip1, im1, jp1, jm1 = limit(i+1, N), limit(i-1, N), limit(j+1, N), limit(j-1, N)
      du[i,j,1] = alpha*(u[im1,j,1] + u[ip1,j,1] + u[i,jp1,1] + u[i,jm1,1] - 4u[i,j,1]) +
                  B + u[i,j,1]^2*u[i,j,2] - (A + 1)*u[i,j,1] + brusselator_f(x, y, t)
      du[i,j,2] = alpha*(u[im1,j,2] + u[ip1,j,2] + u[i,jp1,2] + u[i,jm1,2] - 4u[i,j,2]) +
                  A*u[i,j,1] - u[i,j,1]^2*u[i,j,2]
    end
end
```

https://docs.sciml.ai/latest/tutorials/advanced_ode_example/#Automatic-Sparsity-Detection-1

A program is just control flow:
- Jacobians: whether two variables interact
- Hessians: whether two variables interact nonlinearly

We can generate sparsity patterns from programs via nonstandard interpretation.



51.714 seconds to 1.185 s!
50x for no effort!

Uncertainty Quantification via Metaprograms

- We have a brand new method for uncertainty quantification in ODEs. It requires only one ODE solver call and is incredibly efficient. But, nobody ever wrote the code and nobody ever needs to because it created itself.

julia

DifferentialEquations.jl and Measurements.jl

Usage diffeq

giordano Oct '17

Today I was asked whether it was possible to solve in Julia differential equations involving numbers with uncertainties. Of course the answer is yes. What I find really amazing about Julia is that the two packages don't know anything about each other, yet they can work together without any effort. Here is an short example based on this tutorial 23 : https://nbviewer.jupyter.org/gist/giordano/e82a3959d8f64301129d64d004e10b4e 99

Linear Error Propagation Theory

Any non-linear differentiable function, $f(a,b)$, of two variables, a and b, can be expanded as

$$f \approx f^0 + \frac{\partial f}{\partial a}a + \frac{\partial f}{\partial b}b$$

hence:

$$\sigma_f^2 \approx \left|\frac{\partial f}{\partial a}\right|^2 \sigma_a^2 + \left|\frac{\partial f}{\partial b}\right|^2 \sigma_b^2 + 2\frac{\partial f}{\partial a}\frac{\partial f}{\partial b}\sigma_{ab}$$

where σ_f is the standard deviation of the function f, σ_a is the standard deviation of a, σ_b is the standard deviation of b and σ_{ab} is the covariance between a and b.

In the particular case that $f = ab$, $\frac{\partial f}{\partial a} = b$, $\frac{\partial f}{\partial b} = a$. Then

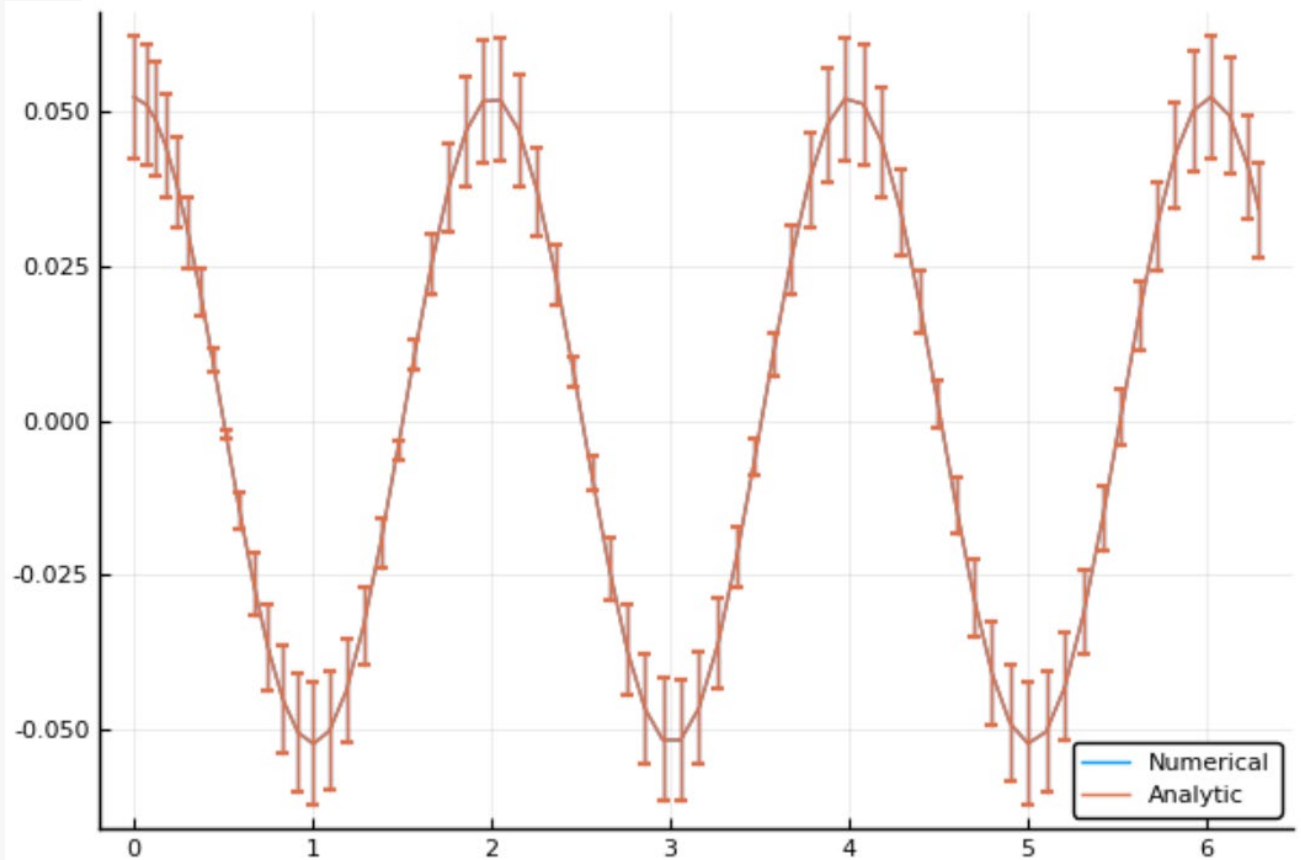
$$\sigma_f^2 \approx b^2\sigma_a^2 + a^2\sigma_b^2 + 2ab\,\sigma_{ab}$$

or

$$\left(\frac{\sigma_f}{f}\right)^2 \approx \left(\frac{\sigma_a}{a}\right)^2 + \left(\frac{\sigma_b}{b}\right)^2 + 2\left(\frac{\sigma_a}{a}\right)\left(\frac{\sigma_b}{b}\right)\rho_{ab}$$

where ρ_{ab} is the correlation between a and b.

```julia
using DifferentialEquations, Measurements, Plots

pyplot()

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

#Initial Conditions
u₀ = [0 ± 0, π / 60 ± 0.01] # Initial speed and initial angle
tspan = (0.0, 6.3)

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

#Pass to solvers
prob = ODEProblem(simplependulum, u₀, tspan)
sol = solve(prob, Tsit5(), reltol = 1e-6)

# Analytic solution
u = u₀[2] .* cos.(sqrt(g / L) .* sol.t)

plot(sol.t, getindex.(sol.u, 2), label = "Numerical")
plot!(sol.t, u, label = "Analytic")
```



Polynomial chaos expansion (PCE) provides a way to represent a random variable Y with finite variance (i.e., $\operatorname{Var}(Y) < \infty$) as a function of an M-dimensional random vector $\mathbf{X}$, using a polynomial basis that is orthogonal to the distribution of this random vector. The prototypical PCE can be written as:

$$Y = \sum_{i \in \mathbb{N}} c_i \Psi_i(\mathbf{X}).$$

$$\int_0^1 6x^5 \mathrm{d}x.$$

Exploiting the underlying uniform measure, the integration can be done exactly with a 3-point quadrature rule.

```
julia> using PolyChaos

julia> opq = Uniform01OrthoPoly(3, addQuadrature = true)
Uniform01OrthoPoly{Array{Float64,1},Uniform01Measure,Quad{Float64,Array{Float64,1}}}(3, [0.5, 0

julia> integrate(x -> 6x^5, opq)
0.9999999999999993

julia> show(opq)

Univariate orthogonal polynomials
degree:         3
#coeffs:        4
α =             [0.5, 0.5, 0.5, 0.5]
β =             [1.0, 0.0833333333333333, 0.0666666666666667, 0.0642857142857142]
```

Another Example: Free Efficient PDE Solvers!

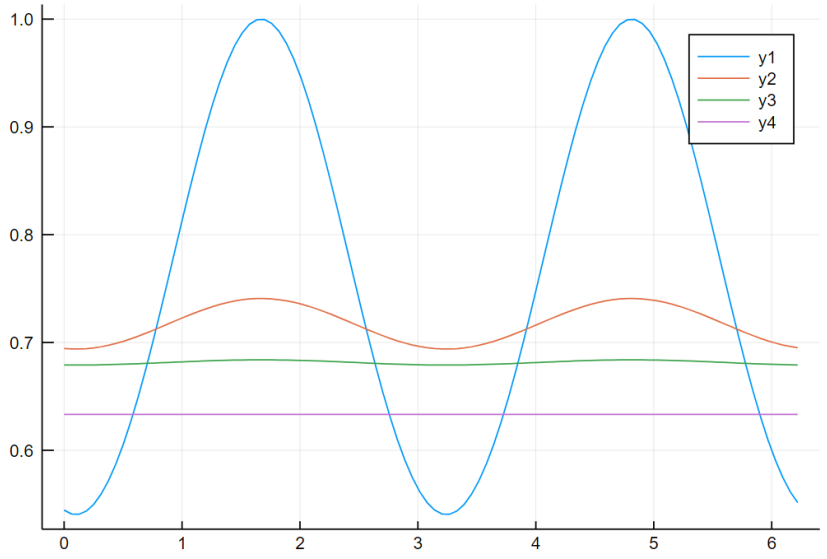
- From Sobolev Function Space Theory, we know that we can represent L^2 functions as vectors in the L^2 function space
- Example: Fourier decomposition

$$f(x) = c + \sum_k a_k \sin(kx) + b_k \cos(kx)$$

- $f(x) \in L^2$ is the infinite length vector $[c, a_k, b_k]$. It is essentially a number in L^2 space.
- An arithmetic on these "numbers" makes sense: how do you add, multiply, divide, etc. functions by other functions? You can then define this in terms of the coefficients of the decomposition.
  - Semigroup theory details how using these functions as numbers gives a mathematically rigorous way of defining PDEs and their solution
- Computationally, you can discretize a function into this space and re-write a PDE in terms of an ODE on these vector coefficients. This is known as a spectral or finite element method.

DifferentialEquations.jl + ApproxFun.jl

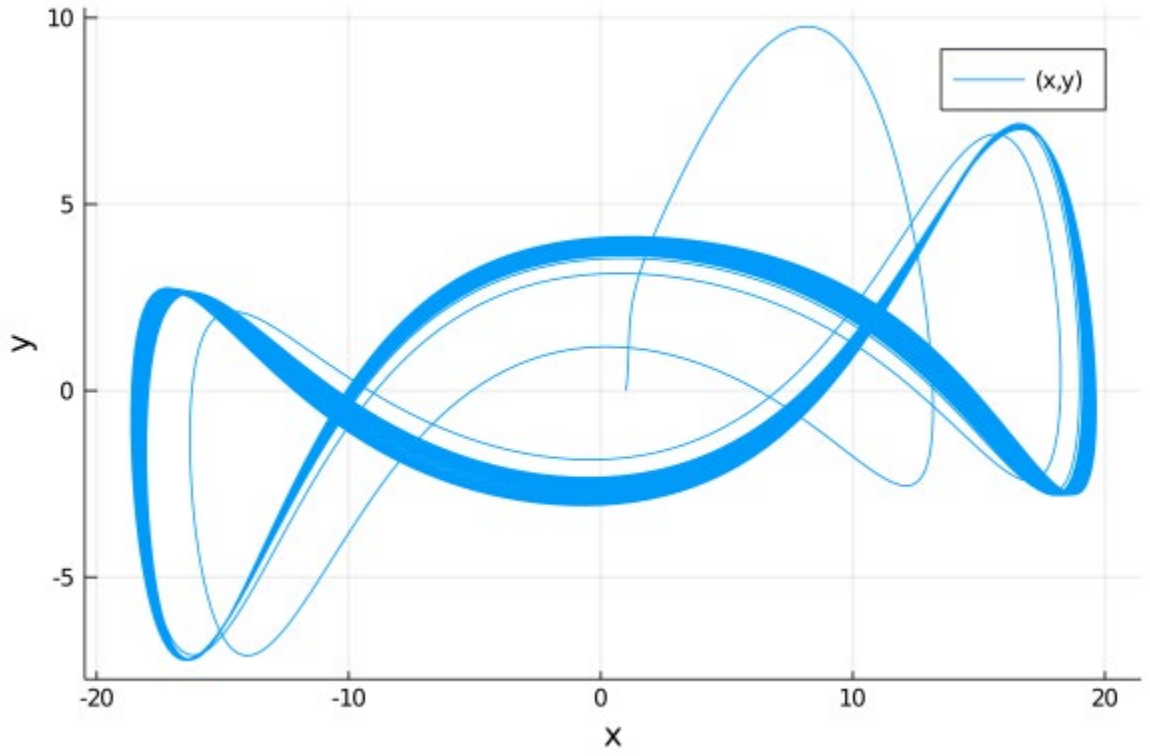
- ApproxFun.jl defines the Fun type which discretizes a function into a chosen decomposition space and treats it like a number
  - It's length adaptive: it automatically chooses the number of coefficients to use in order to specify the function to tolerance.
- DifferentialEquations.jl has ODE solvers which are generic to the Number and AbstractArray type which are used.
- What happens if you define your PDE using a Fun as an initial condition, define the PDE by its arithmetic (i.e. write down the ODE on the function), and hit solve?
  - You get some of the world's first adaptive space + adaptive time spectral/pseudospectral ODE solvers. Super efficient. Nobody wrote the code. Works fine.



A story that merges AD-based compiler techniques with surrogate generation

(Julia Computing JuliaSim Pitch)

ModelingToolkit.jl – A staged programming paradigm for modeling and simulation

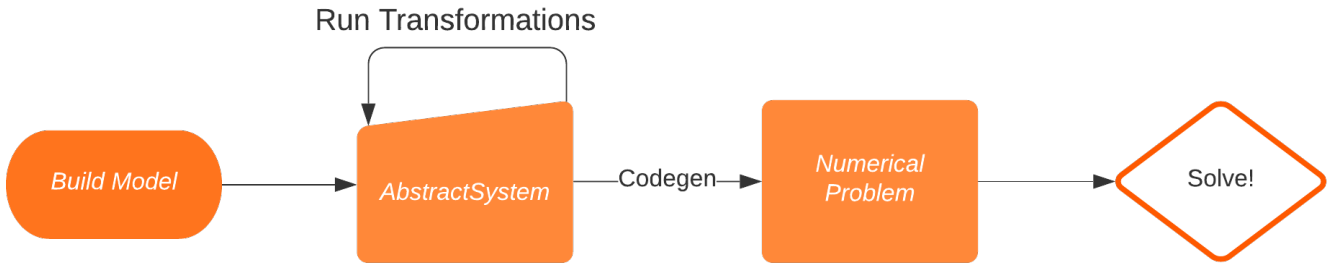



```julia
using ModelingToolkit, OrdinaryDiffEq

@parameters t σ ρ β
@variables x(t) y(t) z(t)
D = Differential(t)

eqs = [D(D(x)) ~ σ*(y-x),
       D(y) ~ x*(ρ-z)-y,
       D(z) ~ x*y - β*z]

sys = ODESystem(eqs)
sys = ode_order_lowering(sys)

u0 = [D(x) => 2.0,
      x => 1.0,
      y => 0.0,
      z => 0.0]

p  = [σ => 28.0,
      ρ => 10.0,
      β => 8/3]

tspan = (0.0,100.0)
prob = ODEProblem(sys,u0,tspan,p,jac=true)
sol = solve(prob,Tsit5())
using Plots; plot(sol,vars=(x,y))
```

ModelingToolkit is Staged Stable Transformations of Models

$$u'' = f(u)$$

$$u' = x$$
$$x' = f(u)$$



In the compiler world, GCC is an example of a compiler with many lowering steps while LLVM is an example of a compiler with a well-documented IR designed for stable transformations

```
using ModelingToolkit, OrdinaryDiffEq

@parameters t σ ρ β
@variables x(t) y(t) z(t)
D = Differential(t)

eqs = [D(D(x)) ~ σ*(y-x),
       D(y) ~ x*(ρ-z)-y,
       D(z) ~ x*y - β*z]

sys = ODESystem(eqs)
sys = ode_order_lowering(sys)

u0 = [D(x) => 2.0,
      x => 1.0,
      y => 0.0,
      z => 0.0]

p  = [σ => 28.0,
      ρ => 10.0,
      β => 8/3]

tspan = (0.0,100.0)
prob = ODEProblem(sys,u0,tspan,p,jac=true)
sol = solve(prob,Tsit5())
using Plots; plot(sol,vars=(x,y))
```

Gives an ODESystem

What Kinds of Transformations Do You Get? DAE Index Reduction

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

Not solvable by standard numerical solvers!

Differentiate the last equation twice, do a few substitutions…

$$
\begin{aligned}
x' &= v_x \\
v_x' &= xT \\
y' &= v_y \\
v_y' &= yT - g \\
0 &= 2\left(v_x^2 + v_y^2 + y(yT - g) + Tx^2\right)
\end{aligned}
$$

Easy to solve!

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

Composable (Acausal) Modeling via Subsystems

```julia
using JuliaSim

R = 1.0
C = 1.0
V = 1.0
@named resistor = Resistor(R=R)
@named capacitor = Capacitor(C=C)
@named source = ConstantVoltage(V=V)
@named ground = Ground()

rc_eqs = [
          connect(source.p, resistor.p)
          connect(resistor.n, capacitor.p)
          connect(capacitor.n, source.n, ground.g)
         ]
```

Describe how the subsystems relate

```julia
@named rc_model = ODESystem(rc_eqs,
                  systems=[resistor, capacitor, source, ground])
```

Build a system of subsystems!

```
julia> equations(rc_model)
16-element Vector{Equation}:
 0 ~ resistor₊p₊i(t) + source₊p₊i(t)
 source₊p₊v(t) ~ resistor₊p₊v(t)
 0 ~ capacitor₊p₊i(t) + resistor₊n₊i(t)
 resistor₊n₊v(t) ~ capacitor₊p₊v(t)
 ⋮
 Differential(t)(capacitor₊v(t)) ~ capacitor₊p₊i(t)*(capacitor₊C^-1)
 source₊V ~ source₊p₊v(t) - source₊n₊v(t)
 0 ~ source₊n₊i(t) + source₊p₊i(t)
 ground₊g₊v(t) ~ 0
```

structural_simplify: the analogue of the standard Dymola/Modia compilation pass

```julia
sys = structural_simplify(rc_model)
```

```
julia> equations(sys)
2-element Vector{Equation}:
 0 ~ capacitor₊v(t) + resistor₊R*capacitor₊p₊i(t) - source₊V
 Differential(t)(capacitor₊v(t)) ~ capacitor₊p₊i(t)*(capacitor₊C^-1)
```

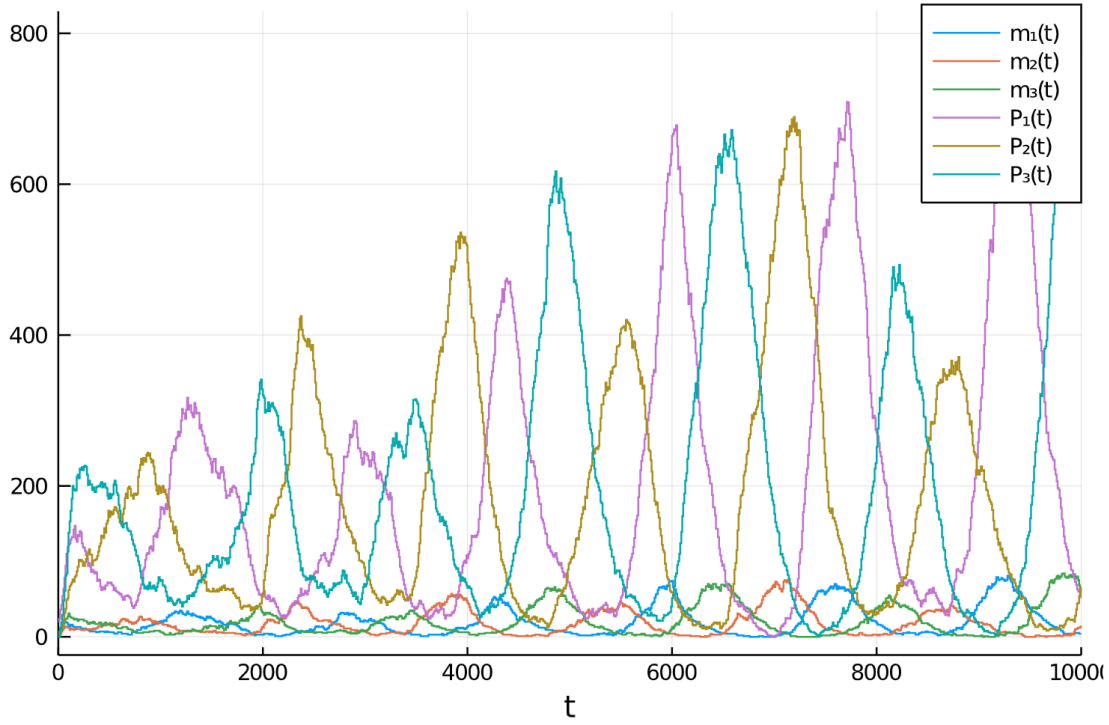
What Kinds of Transformations Do You Get? Moment Closures For Free!

Chemical Reaction Systems as Stochastic Models

```
using Catalyst, DifferentialEquations, Plots, Latexify
repressilator = @reaction_network begin
    hillr(P₃,α,K,n), ∅ --> m₁
    hillr(P₁,α,K,n), ∅ --> m₂
    hillr(P₂,α,K,n), ∅ --> m₃
    (δ,γ), m₁ ↔ ∅
    (δ,γ), m₂ ↔ ∅
    (δ,γ), m₃ ↔ ∅
    β, m₁ --> m₁ + P₁
    β, m₂ --> m₂ + P₂
    β, m₃ --> m₃ + P₃
    μ, P₁ --> ∅
    μ, P₂ --> ∅
    μ, P₃ --> ∅
end α K n δ γ β μ;

u₀ = [0,0,0,20,0,0]
dprob = DiscreteProblem(repressilator, u₀, tspan, p)
jprob = JumpProblem(repressilator, dprob, Direct(),
                    save_positions=(false,false))
sol = solve(jprob, SSAStepper(), saveat=10.)
plot(sol)
```

$$\sum_{i=1}^{N} s_{ij} X_i \xrightarrow{k_j} \sum_{i=1}^{N} r_{ij} X_i, \quad j = 1, \ldots, R,$$

$$\frac{dP(\mathbf{n},t)}{dt} = \sum_{r=1}^{R} \left[a_r(\mathbf{n} - S_r) P(\mathbf{n} - S_r, t) - a_r(\mathbf{n}) P(\mathbf{n}, t) \right]$$



What Kinds of Transformations Do You Get? Moment Closures For Free!

Chemical Reaction Systems as Stochastic Models

$$\sum_{i=1}^{N} s_{ij} X_i \xrightarrow{k_j} \sum_{i=1}^{N} r_{ij} X_i, \quad j = 1, \ldots, R,$$

$$\frac{dP(\mathbf{n},t)}{dt} = \sum_{r=1}^{R} \left[a_r(\mathbf{n} - S_r) P(\mathbf{n} - S_r, t) - a_r(\mathbf{n}) P(\mathbf{n}, t) \right]$$

Who the heck would want to do that by hand?

You can write out the moments…

$$\begin{aligned} \sum_{\mathbf{n}} n_i \frac{dP(\mathbf{n},t)}{dt} &= \sum_{n_1}^{\infty} \sum_{n_2}^{\infty} \cdots \sum_{n_N}^{\infty} n_i \frac{dP(\mathbf{n},t)}{dt} \\ &= \sum_{r} \sum_{\mathbf{n}} n_i a_r(\mathbf{n} - S_r) P(\mathbf{n} - S_r, t) - n_i a_r(\mathbf{n}) P(\mathbf{n}, t) \end{aligned}$$

What Kinds of Transformations Do You Get? Moment Closures For Free!

```julia
using Catalyst
rn = @reaction_network begin
  # also including system-size parameter Ω
  (c₁/Ω^2), 2X + Y → 3X
  (c₂), X → Y
  (c₃*Ω, c₄), 0 ↔ X
end c₁ c₂ c₃ c₄ Ω

using MomentClosure
raw_eqs = generate_raw_moment_eqs(rn, 2,
              combinatoric_ratelaw=false)

using Latexify
latexify(raw_eqs)
```

$$\frac{d\mu_{10}}{dt} = c_3\Omega + c_1\mu_{21}\Omega^{-2} - c_2\mu_{10} - c_4\mu_{10} - c_1\mu_{11}\Omega^{-2}$$
$$\frac{d\mu_{01}}{dt} = c_2\mu_{10} + c_1\mu_{11}\Omega^{-2} - c_1\mu_{21}\Omega^{-2}$$
$$\frac{d\mu_{20}}{dt} = c_2\mu_{10} + c_3\Omega + c_4\mu_{10} + 2c_1\mu_{31}\Omega^{-2} + 2c_3\Omega\mu_{10} - 2c_2\mu_{20} - 2c_4\mu_{20} - c_1\mu_{11}\Omega^{-2} - c_1\mu_{21}\Omega^{-2}$$
$$\frac{d\mu_{11}}{dt} = c_2\mu_{20} + c_1\mu_{11}\Omega^{-2} + c_1\mu_{22}\Omega^{-2} + c_3\Omega\mu_{01} - c_2\mu_{10} - c_2\mu_{11} - c_4\mu_{11} - c_1\mu_{12}\Omega^{-2} - c_1\mu_{31}\Omega^{-2}$$
$$\frac{d\mu_{02}}{dt} = c_2\mu_{10} + c_1\mu_{21}\Omega^{-2} + 2c_2\mu_{11} + 2c_1\mu_{12}\Omega^{-2} - c_1\mu_{11}\Omega^{-2} - 2c_1\mu_{22}\Omega^{-2}$$

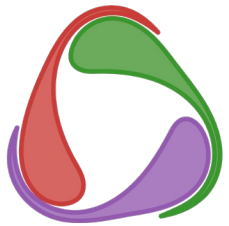
It spits out the ODESystem for the means and variances, now go forth and solve!

ModelingToolkit System Example: NonlinearSystem

NonlinearSolve.jl: Unified Nonlinear Solver Interface

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

- Systems can take other AbstractSystems as subsystems
- sturctural_simplify is “universal simplification”
- The syntax is all similar
- Each system has a SciMLProblem type and a unified solver library



The SciML Common Interface for Julia Equation Solvers

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

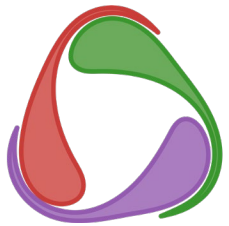
```julia
using ModelingToolkit, NonlinearSolve

@variables x y z
@parameters σ ρ β

# Define a nonlinear system
eqs = [0 ~ σ*(y-x),
       0 ~ x*(ρ-z)-y,
       0 ~ x*y - β*z]
ns = NonlinearSystem(eqs, [x,y,z], [σ,ρ,β])

guess = [x => 1.0,
         y => 0.0,
         z => 0.0]

ps = [
      σ => 10.0
      ρ => 26.0
      β => 8/3
      ]

prob = NonlinearProblem(ns,guess,ps)
sol = solve(prob,NewtonRaphson())
```

ModelingToolkit System Example: NonlinearSystem

NonlinearSolve.jl: Unified Nonlinear Solver Interface

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

structural_simplify:
The God of Transforms

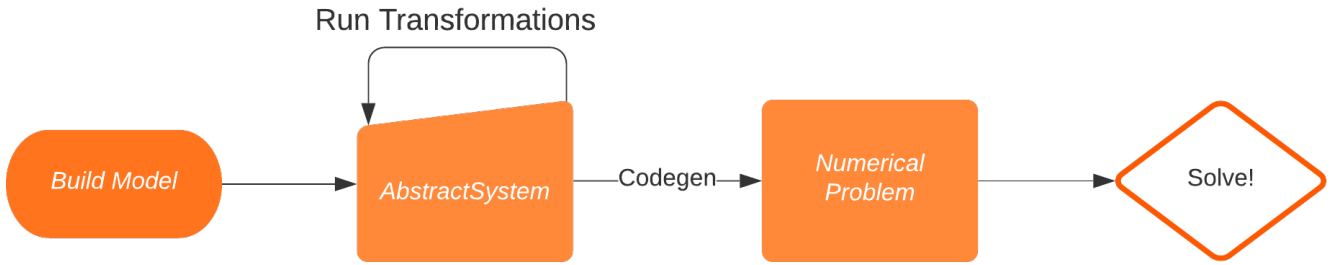


Newton method cost: O(n^3)
O(1^3) << O(5^3)!

The SciML Common Interface for Julia Equation Solvers
https://scimlbase.sciml.ai/dev/

```julia
using ModelingToolkit
@parameters t
@variables u1(t) u2(t) u3(t) u4(t) u5(t)
eqs = [
       0 ~ u1 - sin(u5),
       0 ~ u2 - cos(u1),
       0 ~ u3 - hypot(u1, u2),
       0 ~ u4 - hypot(u2, u3),
       0 ~ u5 - hypot(u4, u1),
]
sys = NonlinearSystem(eqs, [u1, u2, u3, u4, u5], [])
simple_sys = structural_simplify(sys)
equations(simple_sys)

# unknowns: u5
# u1 := sin(u5)
# u2 := cos(u1)
# u3 := hypot(u1, u2)
# u4 := hypot(u2, u3)
# solve for
# 0 ~ u5(t) - hypot(hypot(cos(sin(u5(t))),
#                   hypot(sin(u5(t)), cos(sin(u5(t))))),
#                   sin(u5(t)))
```

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

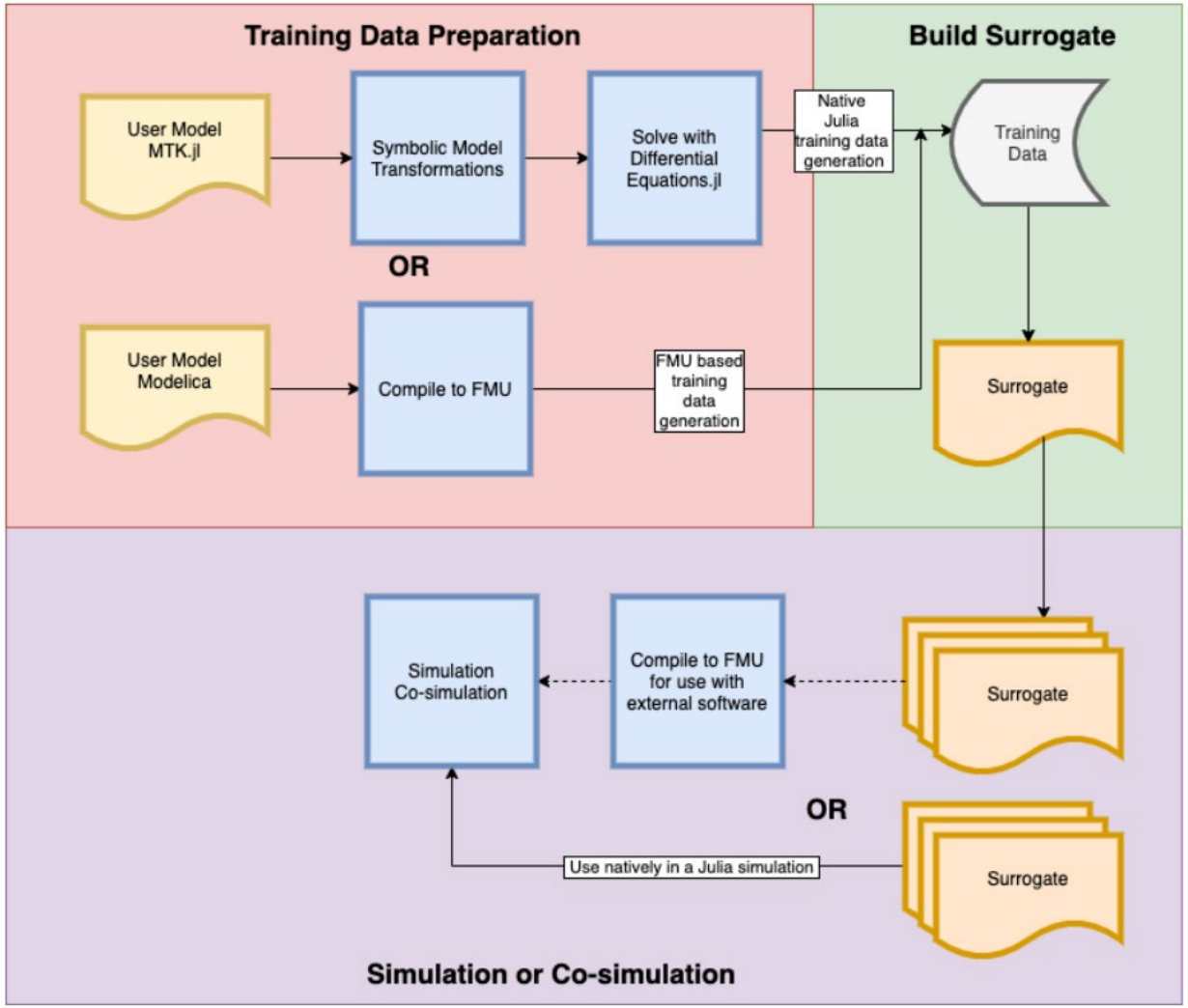
- Take in a differential equation and the outputs to surrogatize over
- Create a new differential equation system that is approximately the same input/output mapping (dimensionality reduction)
- Represent that system as an MTK model

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)
```

Surrogatization as Machine Learned Approximate Transformations



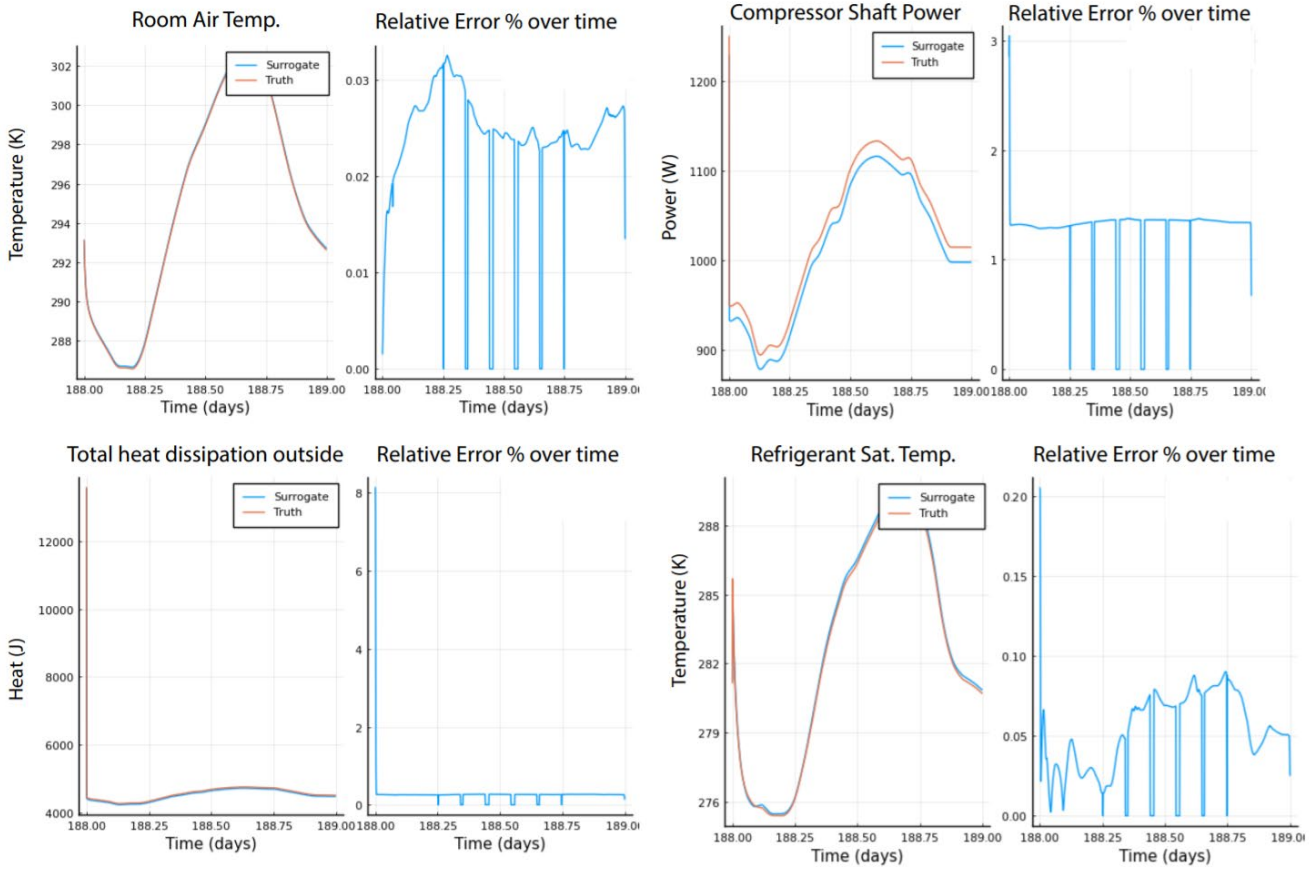
The training data source for a CTESN surrogate does not need to come from ModelingToolkit, it can come from any timeseries data source.

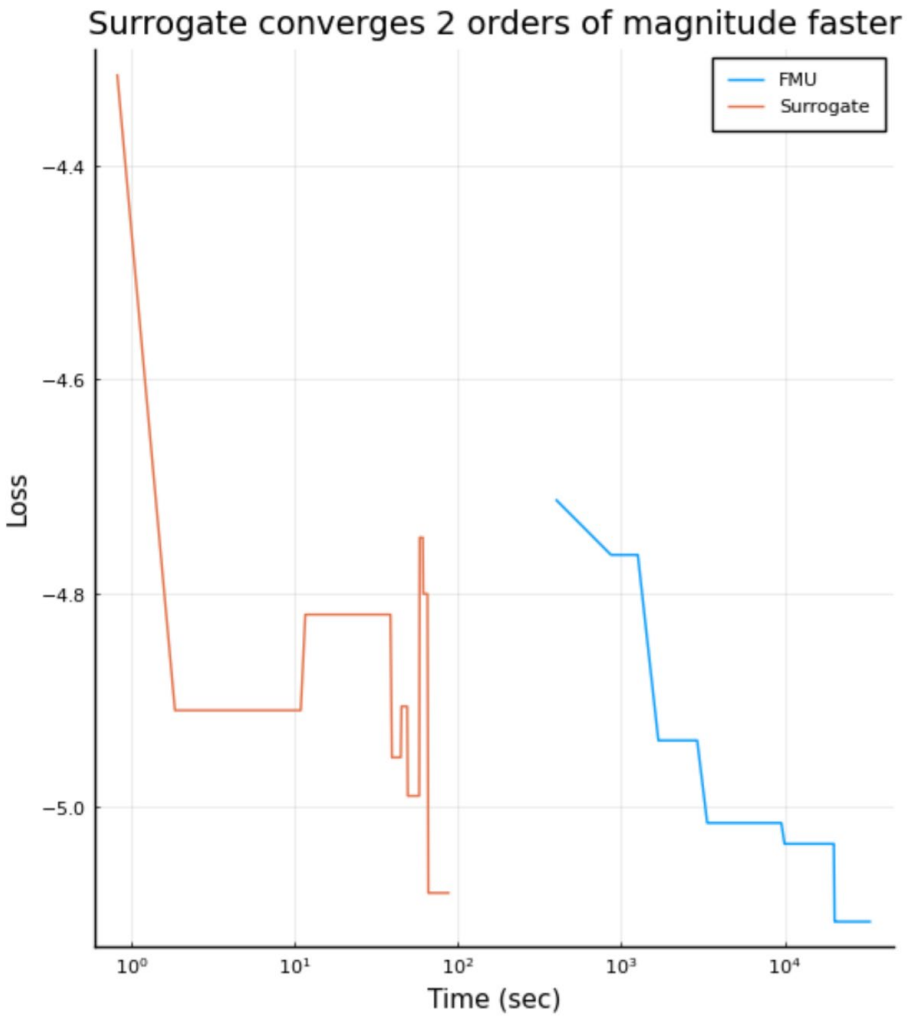
Training CTESNs on timeseries data sources gives a process that merges translation to ModelingToolkit with acceleration!

Sources that we have been experimenting with:

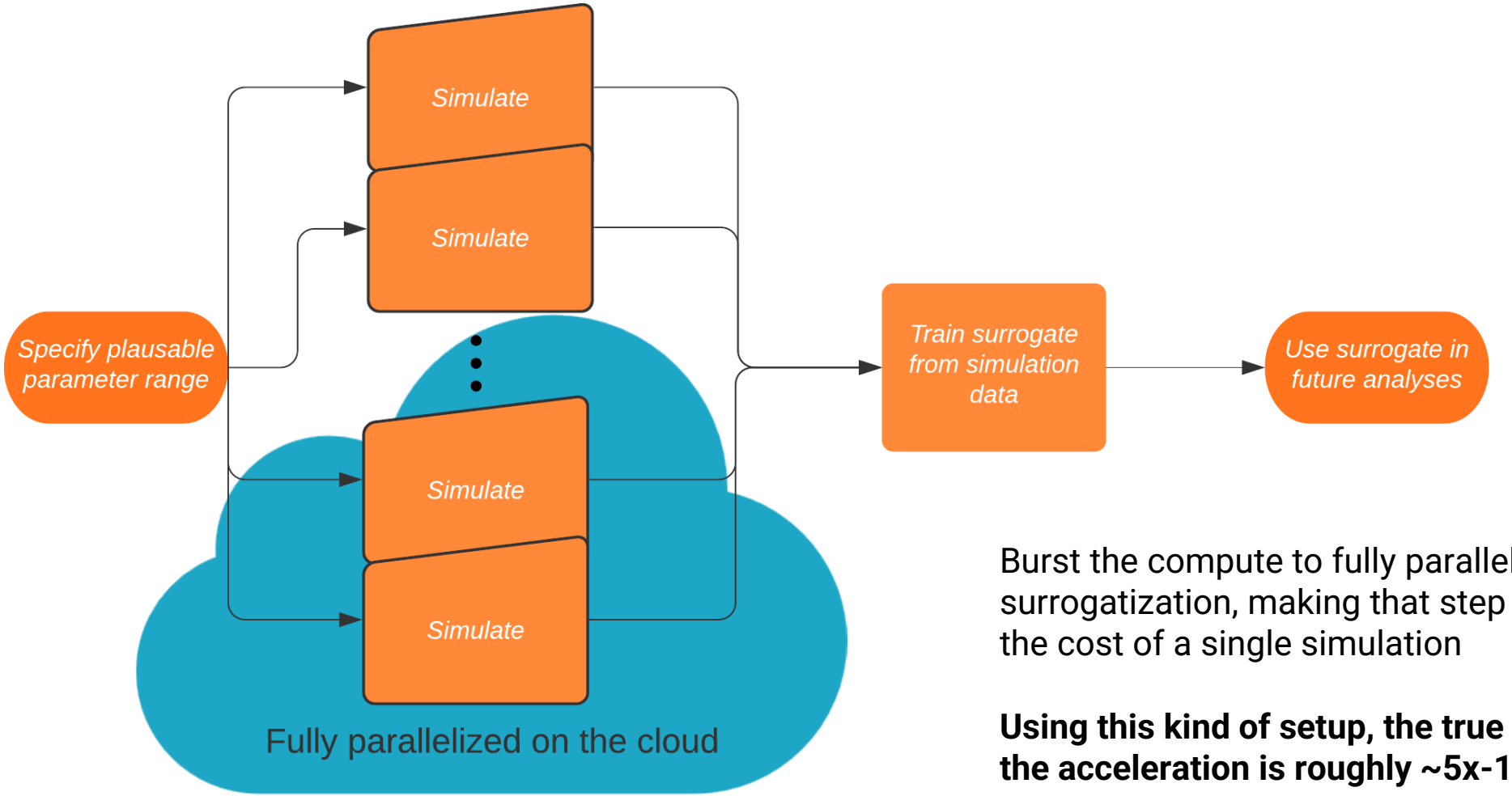
- Functional Markup Units (FMUs) (Dymola, Simulink)
- SPICE models for electrical circuits (NgSpice, Xyce)
- Various PDE tools (COMSOL, Abaqus, etc.)

340x Acceleration of a Global Optimization by Surrogatizing an FMU





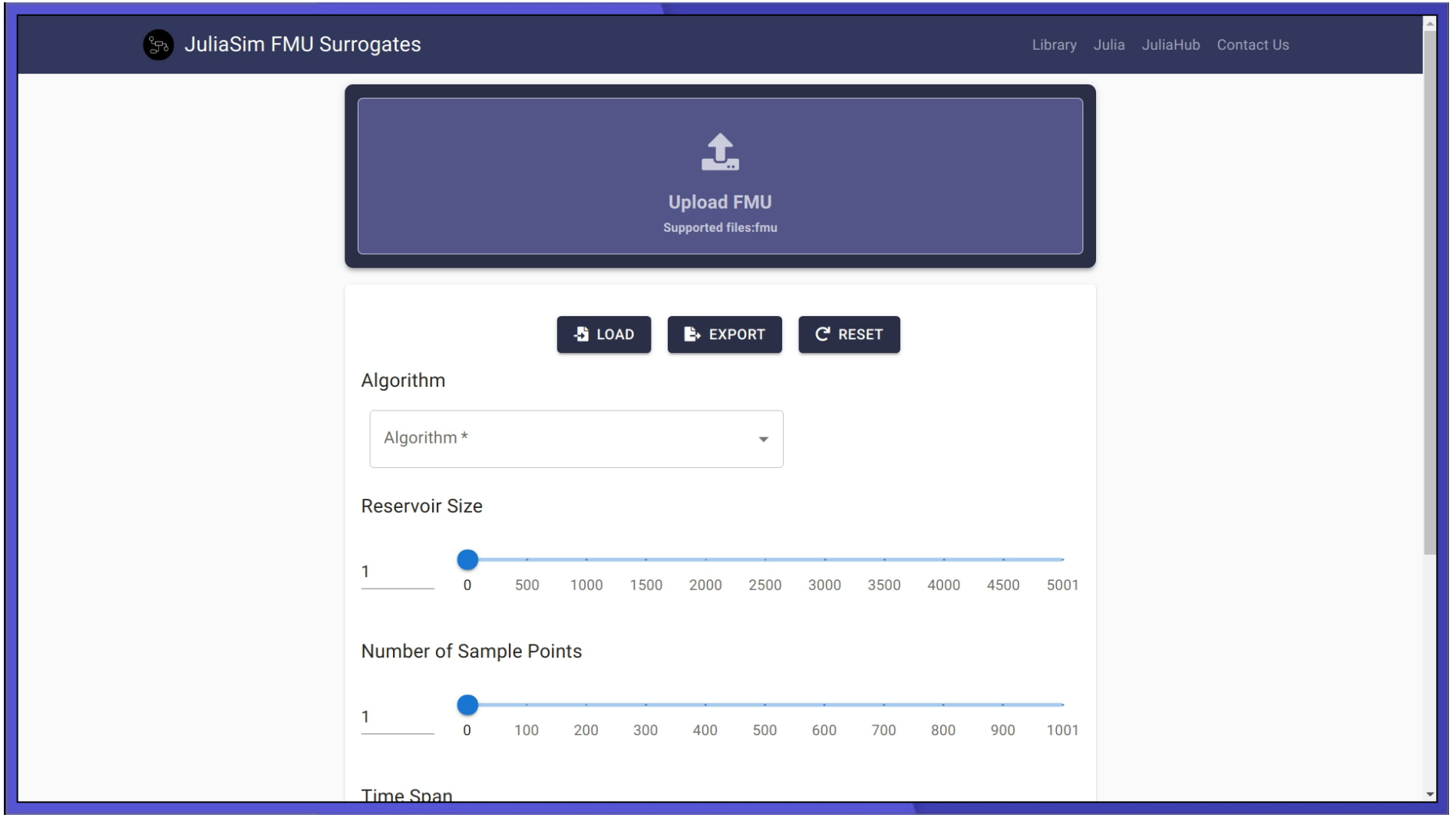
Use Cloud Resources to Smartly Burst Compute and Amortize Time



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 ~5x-10x* the simulation time

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



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.