Chris Rackauckas, JuliaHub and MIT
Yingbo Ma, JuliaHub

JuliaHub

Connecting Scientific Machine Learning with Acausal Modeling

This talk is high-level, talking about what is done rather than the core algorithms. For a longer discussion on the core algorithms, see book.sciml.ai and other longer training sources (the new ModelingToolkitCourse notes!)

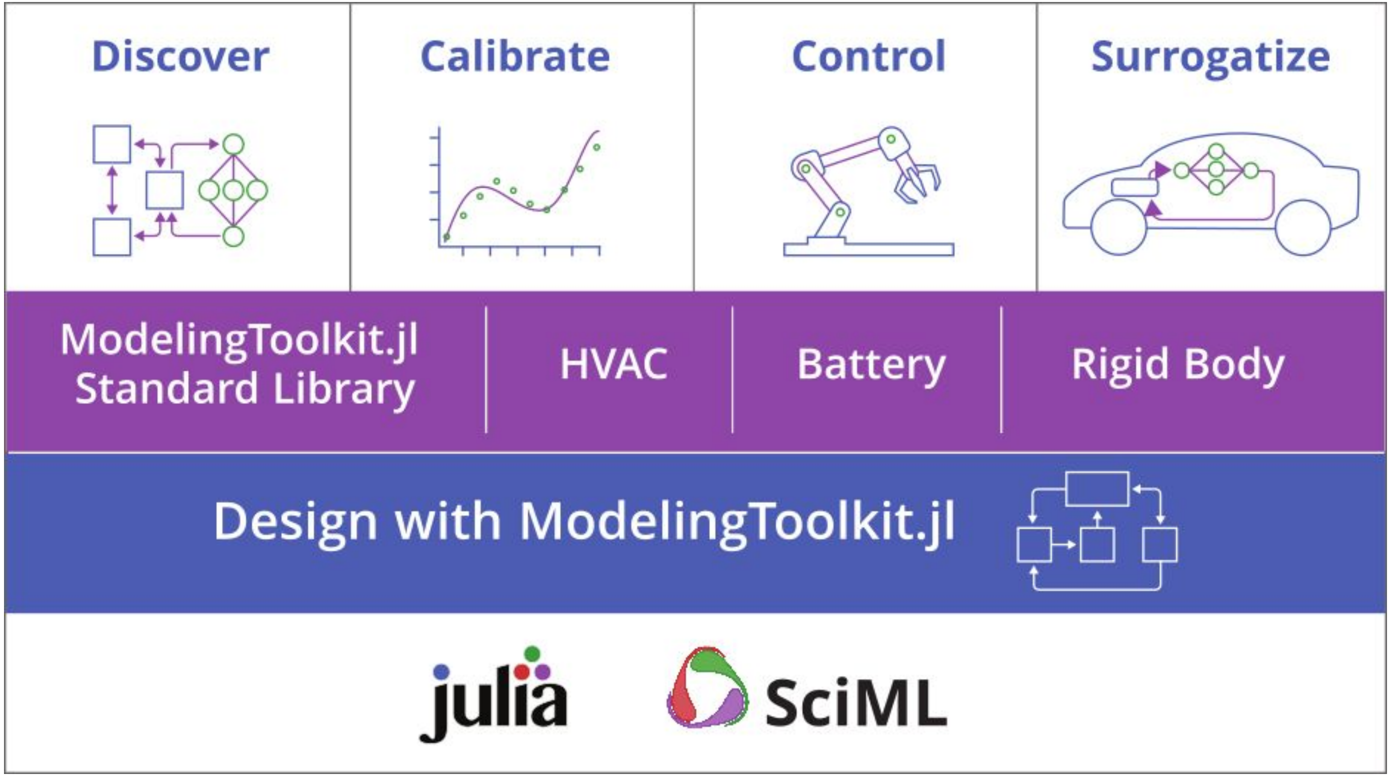
High Level Point:

SciML is the connection between modeling and ML

ModelingToolkit.jl is a modeling system built around symbolic-numeric methods.

Symbolic-Numeric-ML computing is our next step

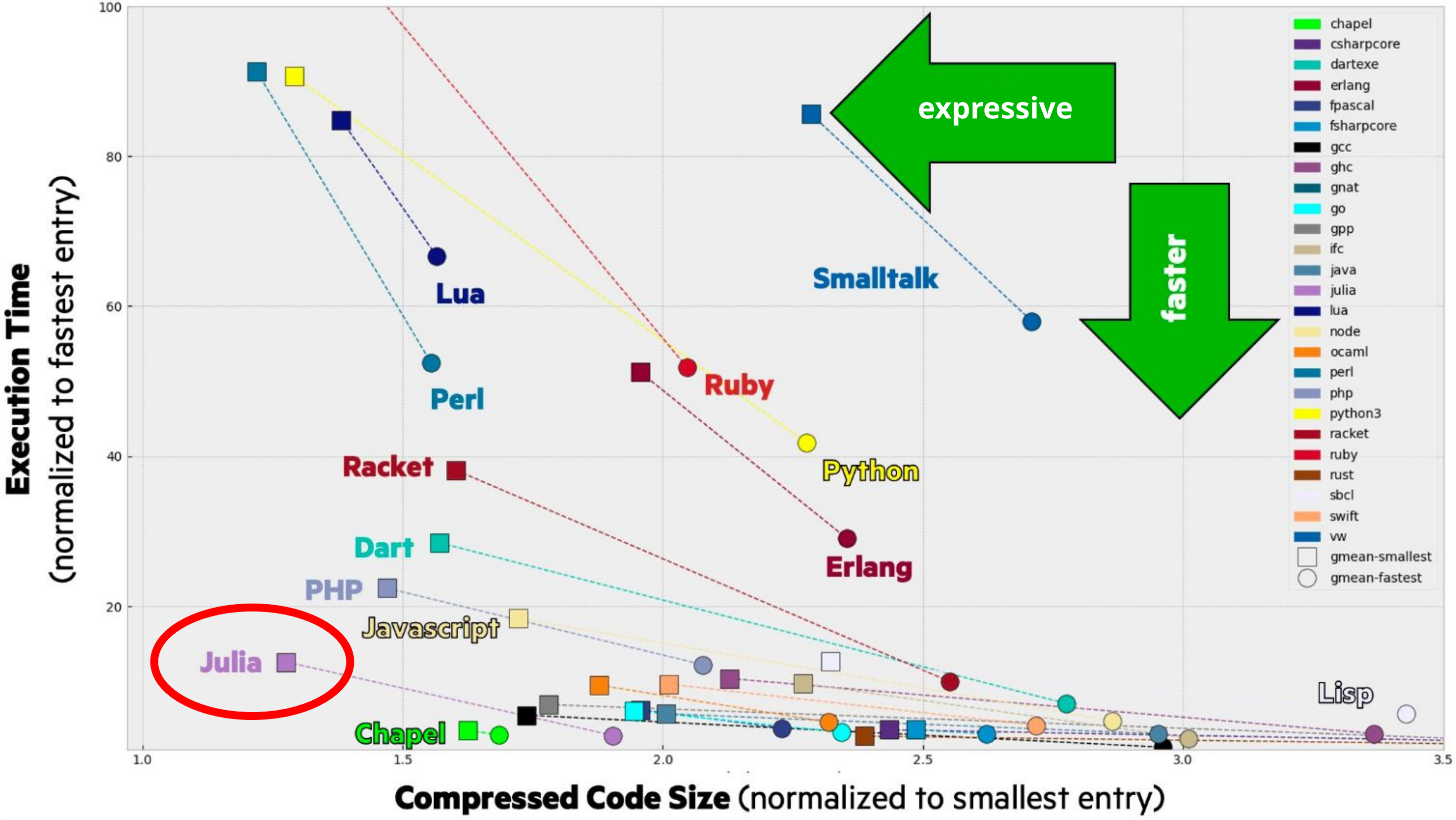
Building an Ecosystem on Open Source Foundations



JuliaHub

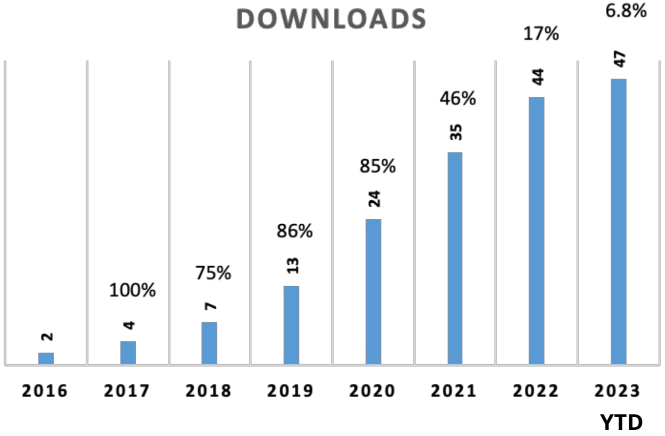
Julia Language and SciML

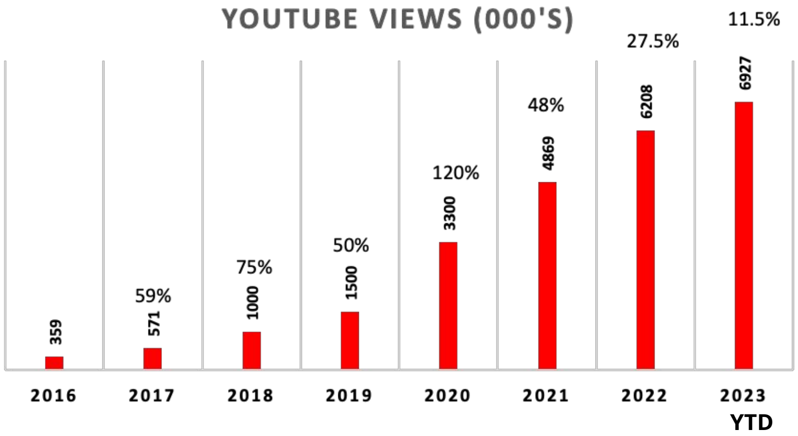
Julia is a high-level language that is faster than R and Python

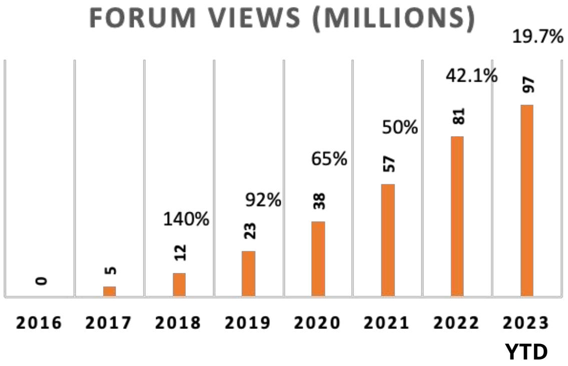


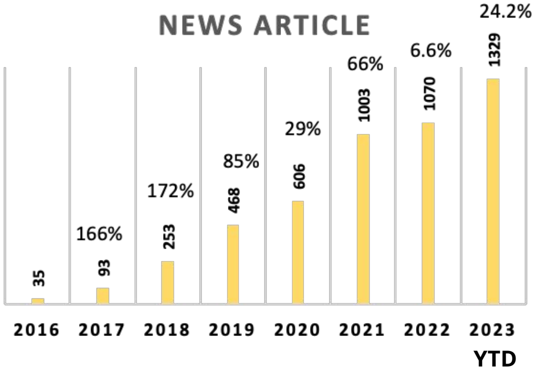
Computer Language Benchmarks Game: all-language summary (May 10, 2022)
Tweeted by the Chapel folks: https://twitter.com/ChapelLanguage/status/1484581096604016647

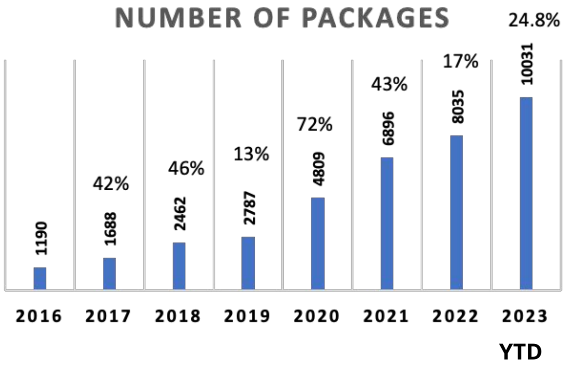
The Julia Community Is Growing!











SciML: Common Interface for Julia Equation Solvers

- `LinearSolve.jl` $A(p)x = b$
- `NonlinearSolve.jl` $f(u,p) = 0$
- `DifferentialEquations.jl` $u' = f(u,p,t)$
- `Integrals.jl` $\int_{lb}^{ub} f(t,p)dt$
- `Optimization.jl` $$\text{minimize } f(u,p)$$ $$\text{subject to } g(u,p) \leq 0, h(u,p) = 0$$

Differential Equation Solvers: Speed

Benchmarks

- 50x faster than SciPy
- 50x faster than MATLAB
- 100x faster than deSolve in R

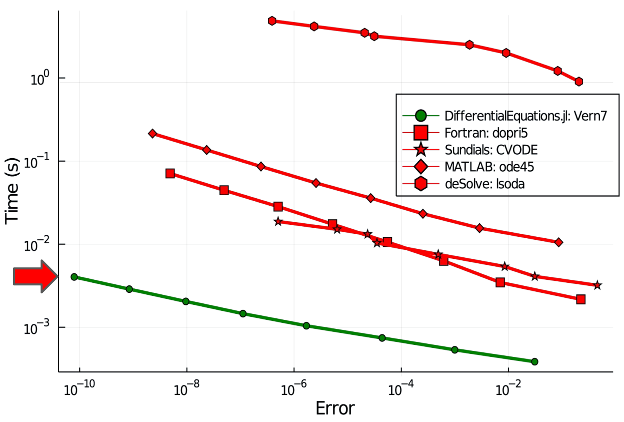
Citations

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

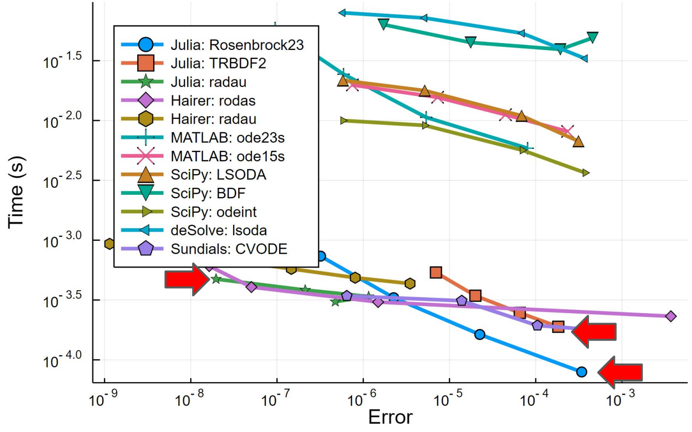
Rackauckas, Christopher, and Qing Nie. "Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia." Journal of Open Research Software 5.1 (2017).

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

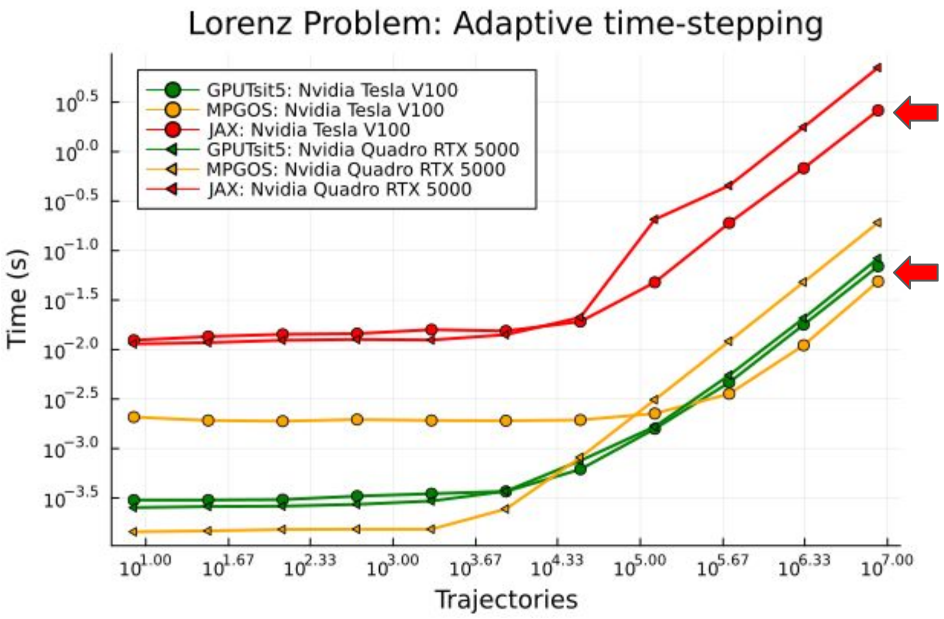
Non-Stiff ODE: Rigid Body System

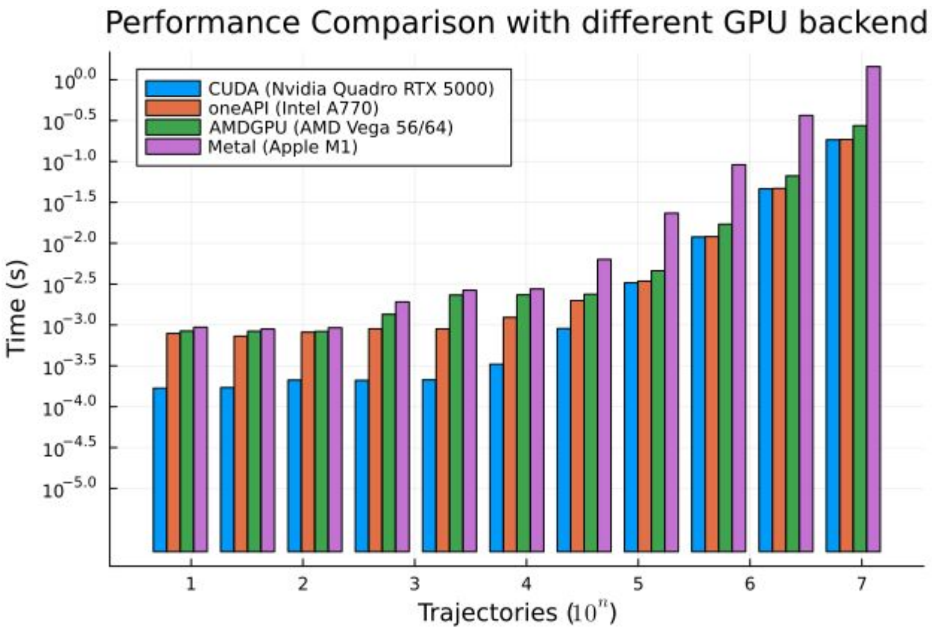


Stiff ODE: HIRES Chemical Reaction Network



GPU ODE Parallelism: 20-100x Faster than Jax and PyTorch





Matches CUDA but works on AMD, Intel and Apple GPUs

JuliaHub

Symbolic-Numerics in Scientific Machine Learning

Data + Physics = Scientific Machine Learning

What is Scientific Machine Learning (SciML)?

Scientific Computing $\leftrightarrow$ Machine Learning

Scientific Computing

- Model Building
- Robust Solvers
- Control Systems

+

Machine Learning

- Neural Nets
- Bayesian Modeling
- Automatic Differentiation

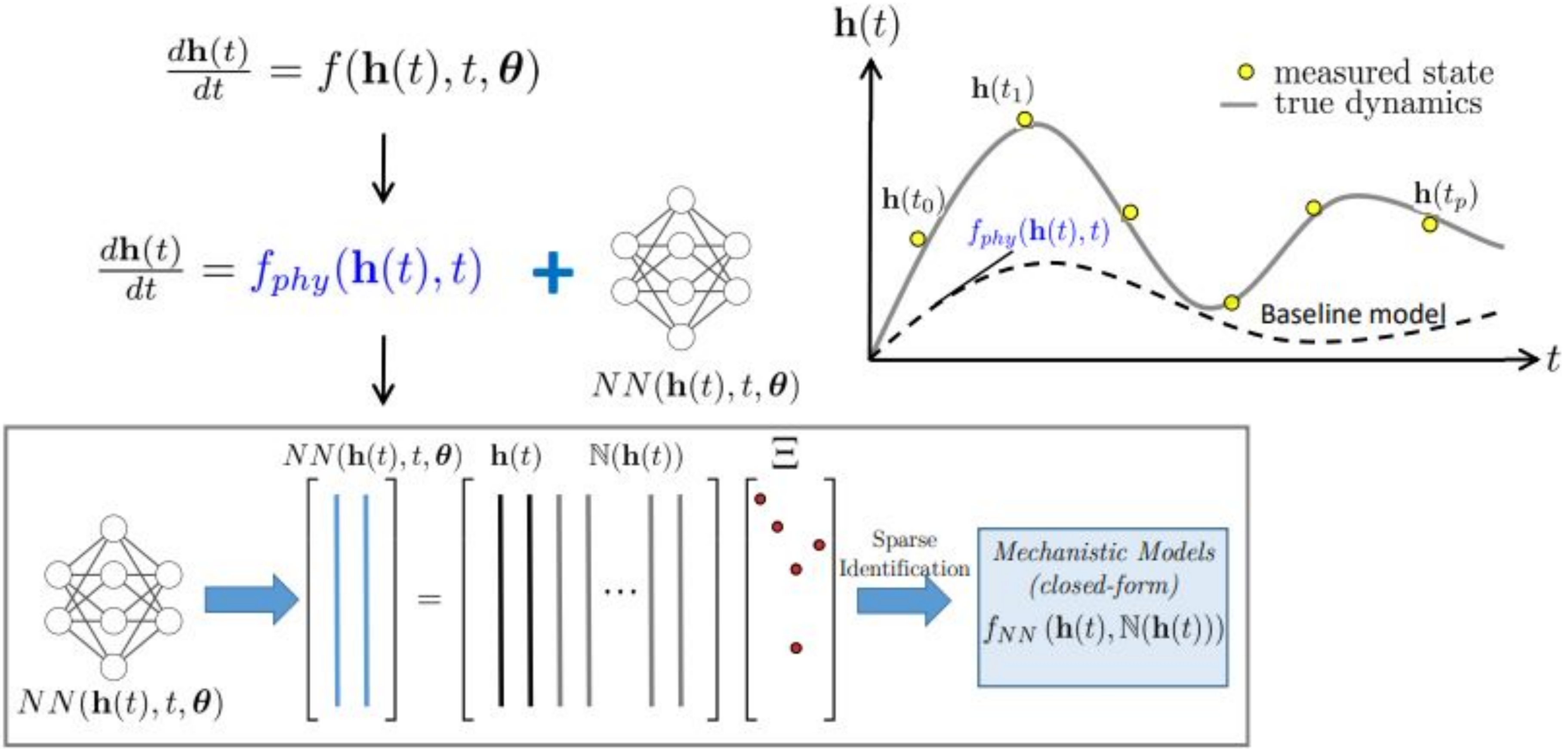
→

Scientific Machine Learning

- Differentiable Simulators
- Surrogates and ROM
- Inverse Problems & Calibration
- Automatic Equation Discovery
- Applicable to Small Data

and more ....

JuliaSim Model Discovery: Autocompleting Models with SciML



Julia SIM

Accurate Model Extrapolation Mixing in Physical Knowledge

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

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

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

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

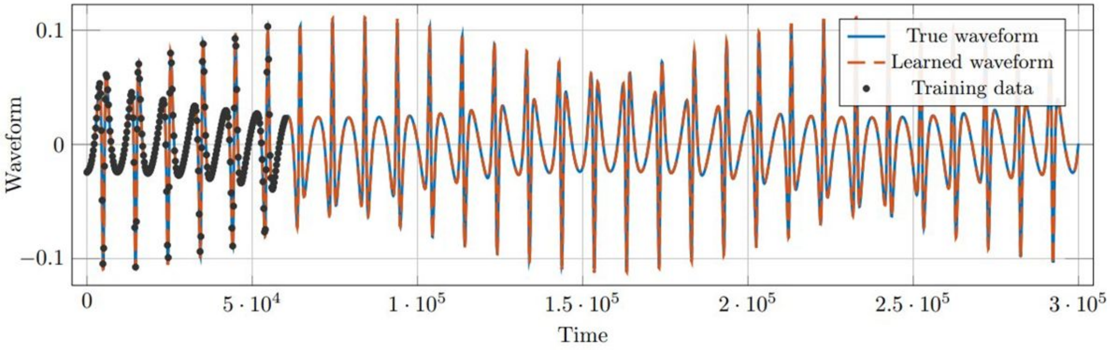
$$\dot{e} = \mathcal{F}_4(p, e), \tag{5d}$$

Keith, B., Khadse, A., & Field, S. E. (2021). Learning orbital dynamics of binary black hole systems from gravitational wave measurements. Physical Review Research, 3(4), 043101.

Automated discovery of geodesic equations from LIGO black hole data: run the code yourself!

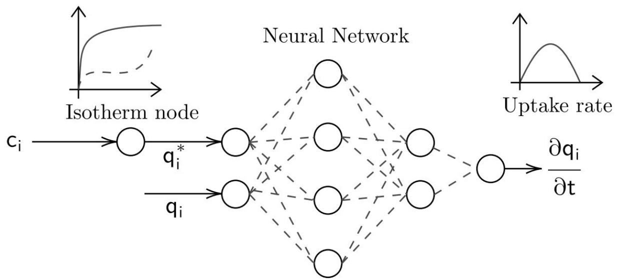
https://docs.sciml.ai/Overview/stable/showcase/blackhole/

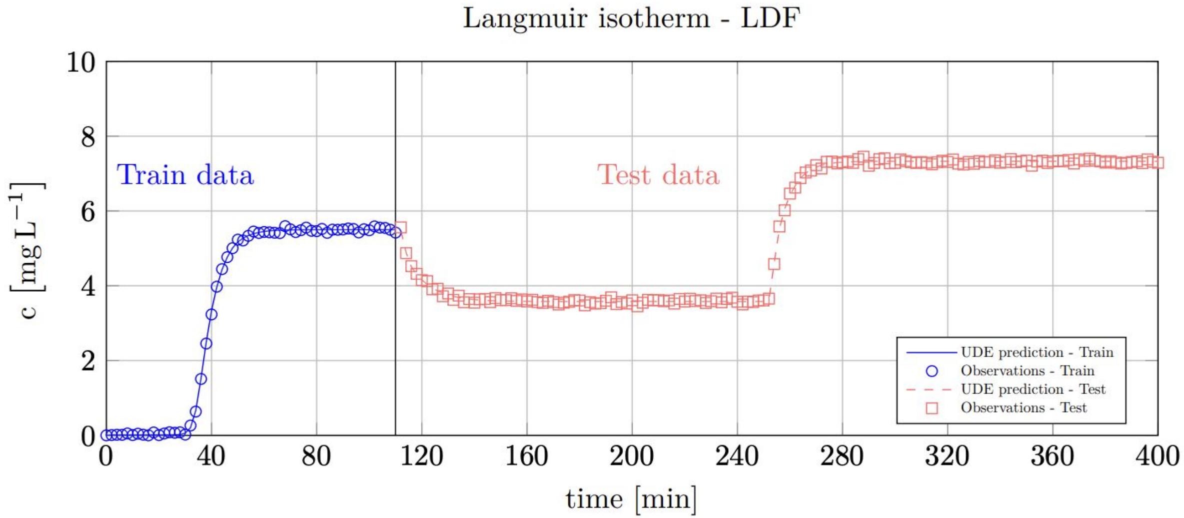
For more examples, see Scientific Machine Learning Through Symbolic Numerics, JuliaCon 2023 Keynote



Universal Differential Equations Predict Chemical Processes

$$
\begin{aligned}
&\frac{\partial c}{\partial t^*} = -\frac{1-\varepsilon}{\varepsilon}\mathrm{ANN}(q, q^*, \theta) - \frac{\partial c}{\partial x^*} + \frac{1}{Pe}\frac{\partial c^2}{\partial x^{*2}},\\
&\frac{\partial q}{\partial t^*} = \mathrm{ANN}(q, q^*, \theta),\\
&\frac{\partial c(x^* = 1, \forall t)}{\partial x^*} = 0,\\
&\frac{\partial c(x^* = 0, \forall t)}{\partial x^*} = Pe(c - c_{inlet}),\\
&c(x^* \in (0,1), t^* = 0) = c_0,\\
&q(x^* \in (0,1), t^* = 0) = q^*(c_0),\\
&q^* = f(c, p),
\end{aligned}
$$





UDEs in advection-diffusion transform the learning problem to low dimensional spaces where small data is sufficient

Universal Differential Equations Predict Chemical Processes

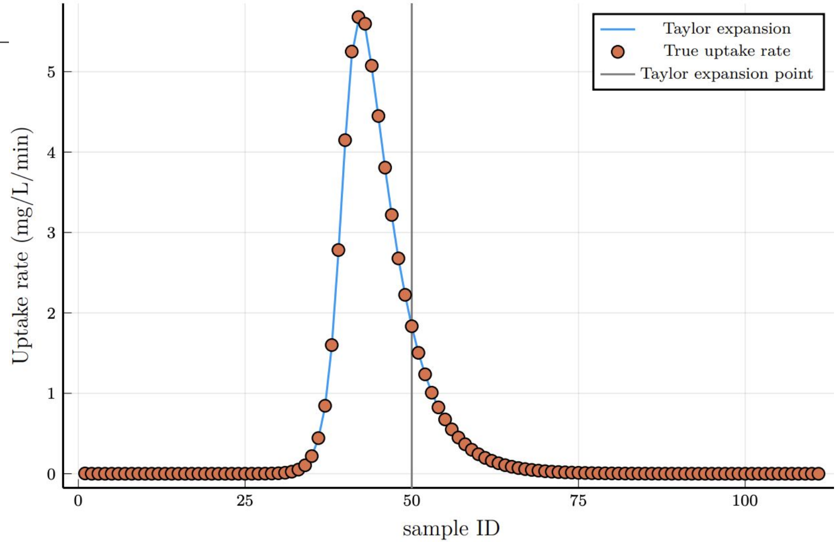
Table 5: Symbolic regression learned polynomials.

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

Recovers equations with the same 2nd order Taylor expansion

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

For more success stories, see Accurate and Efficient Physics-Informed Learning Through Differentiable Simulation



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

UDEs Effectively Recover Nonlinearities of Epidemic Models

The baseline case:

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

Replacement of all terms with neural networks:

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

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

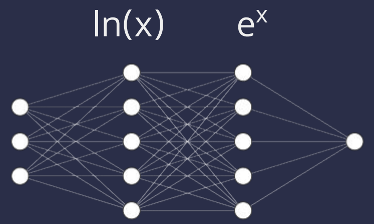
	Actual Equations	SINDY Active terms	SINDY Equations	Minimum AICC
NN_{SI}	0.85 S I	1: SI	0.74 S I	14
NN_{IR}	0.1 I	1: I	0.097 I	19
NN_{ID}	0.05 I	1: I	0.049 I	21

Table 4: SIRD: SINDY Recovered terms

Scientific Machine Learning vs. Pure ML

Physically-Informed Machine Learning

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



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

New Architecture: DigitalEcho

Smoother, more accurate results

JuliaHub

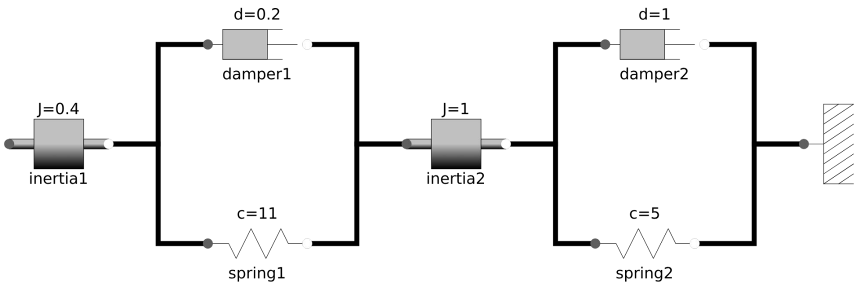
Two Questions to Link Acausal Modeling to ML:

1. How can one create ML pieces that approximate components? (Surrogates)
2. How can one create components which embed ML? (ModelingToolkit/JuliaSimCompiler)

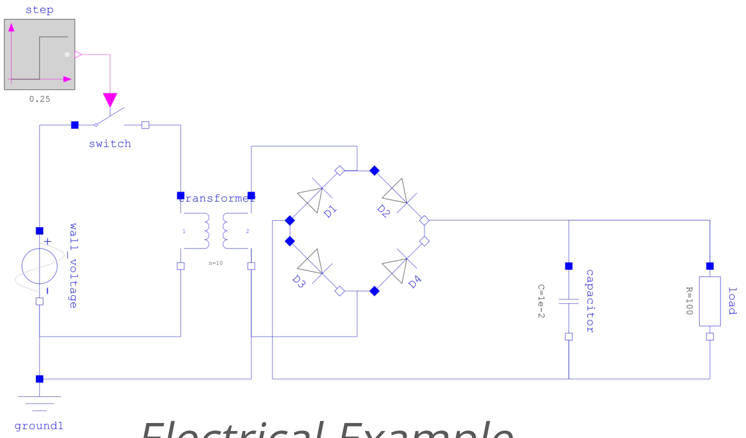
JuliaHub

Introducing: **ModelingToolkit**

ModelingToolkit.jl = Component Based Modeling



Mechanical Example



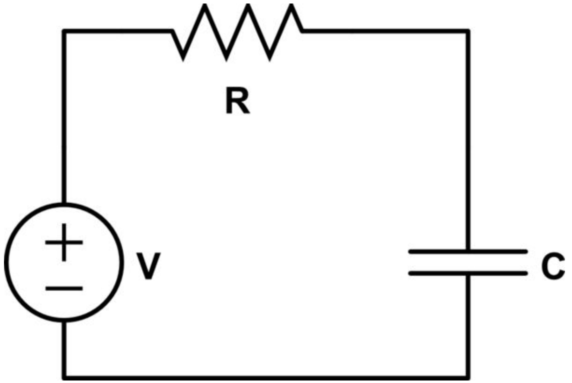
Electrical Example

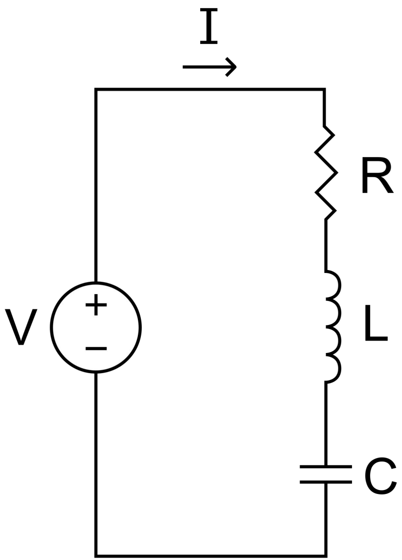
Acausal Modeling Benefits:

- Natural & analogous to real life schematics
- Easier to edit and adapt compared to Block-Diagram modeling
- Efficient: both in human and computational time
- Libraries and Subsystems: Don't Repeat Yourself

Causal vs Acausal Modeling

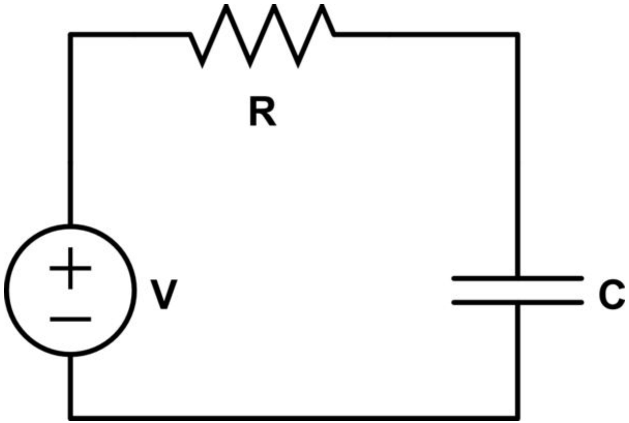
Show it, don't tell it!





Let's use some elementary circuit examples to demonstrate the difference.

RC Circuit: Causal



Kirchhoff's Voltage Law

$$V_R + V_C = V$$

Kirchhoff's Current Law

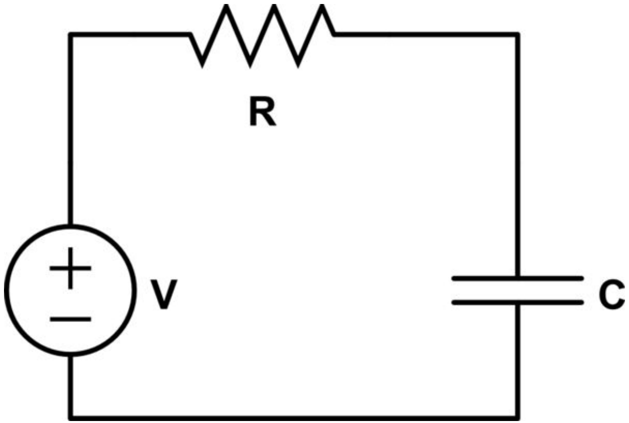
$$I_R = I_C$$

Device equations

$$V_R = I_R R$$

$$I_C = C\dot{V}_C$$

RC Circuit: Causal



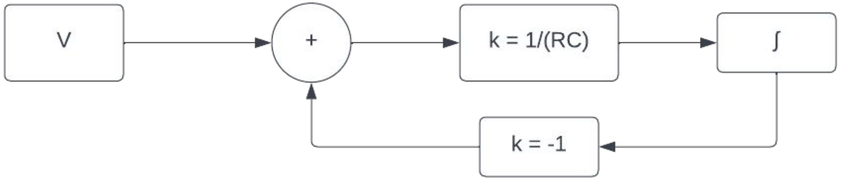
$$V_R + V_C = V \qquad I_R = I_C$$

$$V_R = I_R R \qquad I_C = C\dot{V}_C$$

$$\dot{V}_C = \frac{I_C}{C} = \frac{I_R}{C} = \frac{V_R/R}{C} = \frac{(V - V_C)/R}{C} = \frac{V - V_C}{RC}$$

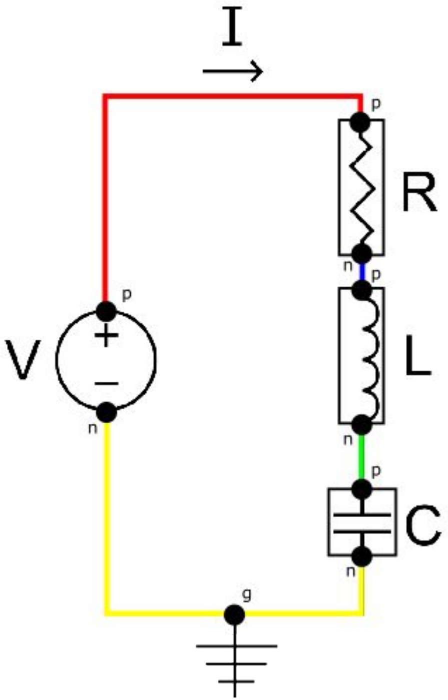
$$\dot{V}_C = \frac{V - V_C}{RC}$$

RC Circuit: Causal




```
systems = @named begin
    Vc_int = Integrator()
    adder = Add(k1 = 1, k2 = -1)
    c_gain = Gain(1 / (R * C))
    voltage_source = Step(start_time = 2, height = 1)
end

causal_rc_eqs = [connect(voltage_source.output, adder.input1)
                 connect(adder.output, c_gain.input)
                 connect(c_gain.output, Vc_int.input)
                 connect(Vc_int.output, adder.input2)]
@named casual_rc = ODESystem(causal_rc_eqs, t; systems)
```

RLC Circuit: Acausal (Component Based Modeling)




```
systems = @named begin
    resistor = Resistor(; R)
    capacitor = Capacitor(; C)
    inductor = Inductor(; L)
    source = Voltage()
    ground = Ground()
end

rlc_eqs = [connect(source.p, resistor.p)
           connect(resistor.n, inductor.p)
           connect(inductor.n, capacitor.p)
           connect(capacitor.n, source.n, ground.g)]

@named acasual_rlc = ODESystem(rlc_eqs, t; systems)
```

Human time: ~1 min

How Acausal Modeling Works: Connections

Acausal Connections

Electrical

For the Electrical domain, the across variable is *voltage* and the through variable *current*. Therefore

- Energy Dissipation:

$$\partial voltage/\partial t \cdot capacitance = current$$

- Flow:

$$current \cdot resistance = voltage$$

Translational

For the translation domain, choosing *velocity* for the across variable and *force* for the through gives

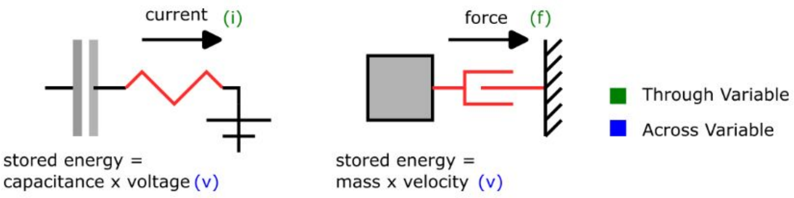
- Energy Dissipation:

$$\partial velocity/\partial t \cdot mass = force$$

- Flow:

$$force \cdot (1/damping) = velocity$$

The diagram here shows the similarity of problems in different physical domains.

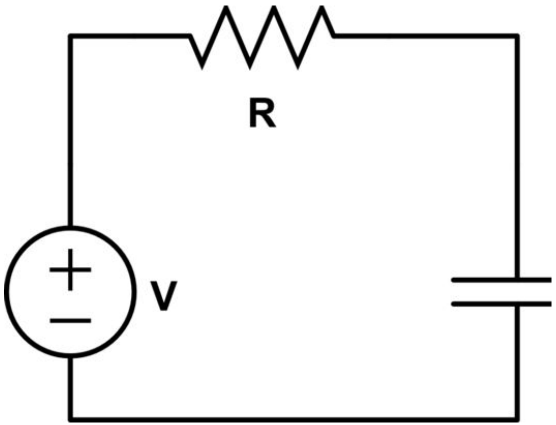


Connecting nodes generates equations:

- Across variables are equal
- Through variables sum to zero

RC Circuit: Acausal

```
equations(expand_connections(rc_model))
```

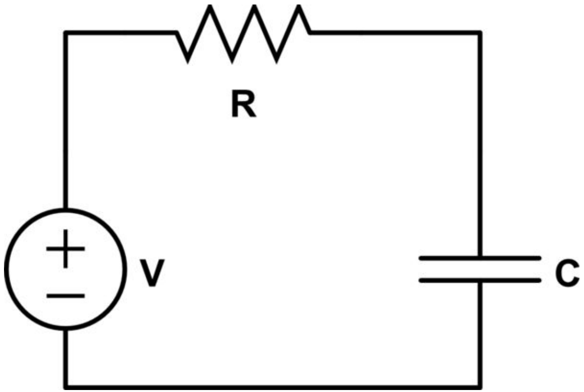


$$
\begin{aligned}
resistor_{+}v(t) &= -resistor_{+}n_{+}v(t) + resistor_{+}p_{+}v(t) \\
0 &= resistor_{+}n_{+}i(t) + resistor_{+}p_{+}i(t) \\
resistor_{+}i(t) &= resistor_{+}p_{+}i(t) \\
resistor_{+}v(t) &= resistor_{+}Rresistor_{+}i(t) \\
capacitor_{+}v(t) &= -capacitor_{+}n_{+}v(t) + capacitor_{+}p_{+}v(t) \\
0 &= capacitor_{+}n_{+}i(t) + capacitor_{+}p_{+}i(t) \\
capacitor_{+}i(t) &= capacitor_{+}p_{+}i(t) \\
\frac{\mathrm{d}capacitor_{+}v(t)}{\mathrm{d}t} &= \frac{capacitor_{+}i(t)}{capacitor_{+}C} \\
source_{+}v(t) &= -source_{+}n_{+}v(t) + source_{+}p_{+}v(t) \\
0 &= source_{+}n_{+}i(t) + source_{+}p_{+}i(t) \\
source_{+}i(t) &= source_{+}p_{+}i(t) \\
source_{+}V &= source_{+}v(t) \\
ground_{+}g_{+}v(t) &= 0 \\
\boxed{source_{+}p_{+}v(t)} &\boxed{= resistor_{+}p_{+}v(t)} \\
0 &= resistor_{+}p_{+}i(t) + source_{+}p_{+}i(t) \\
resistor_{+}n_{+}v(t) &= capacitor_{+}p_{+}v(t) \\
\boxed{0} &\boxed{= capacitor_{+}p_{+}i(t) + resistor_{+}n_{+}i(t)} \\
capacitor_{+}n_{+}v(t) &= ground_{+}g_{+}v(t) \\
capacitor_{+}n_{+}v(t) &= source_{+}n_{+}v(t) \\
0 &= capacitor_{+}n_{+}i(t) + ground_{+}g_{+}i(t) + source_{+}n_{+}i(t)
\end{aligned}
$$

Standard variables connect via equality

Flow variables connect by adding to 0

RC Circuit: Acausal




```
equations(expand_connections(rc_model))
```

$$
\begin{aligned}
resistor_{+}v\left(t\right) &= - resistor_{+}n_{+}v\left(t\right) + resistor_{+}p_{+}v\left(t\right) \\
0 &= resistor_{+}n_{+}i\left(t\right) + resistor_{+}p_{+}i\left(t\right) \\
resistor_{+}i\left(t\right) &= resistor_{+}p_{+}i\left(t\right) \\
resistor_{+}v\left(t\right) &= resistor_{+}Rresistor_{+}i\left(t\right) \\
capacitor_{+}v\left(t\right) &= - capacitor_{+}n_{+}v\left(t\right) + capacitor_{+}p_{+}v\left(t\right) \\
0 &= capacitor_{+}n_{+}i\left(t\right) + capacitor_{+}p_{+}i\left(t\right) \\
capacitor_{+}i\left(t\right) &= capacitor_{+}p_{+}i\left(t\right) \\
\frac{\mathrm{d}capacitor_{+}v\left(t\right)}{\mathrm{d}t} &= \frac{capacitor_{+}i\left(t\right)}{capacitor_{+}C} \\
source_{+}v\left(t\right) &= - source_{+}n_{+}v\left(t\right) + source_{+}p_{+}v\left(t\right) \\
0 &= source_{+}n_{+}i\left(t\right) + source_{+}p_{+}i\left(t\right) \\
source_{+}i\left(t\right) &= source_{+}p_{+}i\left(t\right) \\
source_{+}V &= source_{+}v\left(t\right) \\
ground_{+}g_{+}v\left(t\right) &= 0 \\
source_{+}p_{+}v\left(t\right) &= resistor_{+}p_{+}v\left(t\right) \\
0 &= resistor_{+}p_{+}i\left(t\right) + source_{+}p_{+}i\left(t\right) \\
resistor_{+}n_{+}v\left(t\right) &= capacitor_{+}p_{+}v\left(t\right) \\
0 &= capacitor_{+}p_{+}i\left(t\right) + resistor_{+}n_{+}i\left(t\right) \\
capacitor_{+}n_{+}v\left(t\right) &= ground_{+}g_{+}v\left(t\right) \\
capacitor_{+}n_{+}v\left(t\right) &= source_{+}n_{+}v\left(t\right) \\
0 &= capacitor_{+}n_{+}i\left(t\right) + ground_{+}g_{+}i\left(t\right) + source_{+}n_{+}i\left(t\right)
\end{aligned}
$$

Eliminated variables are algebraically constructed

```
sys = structural_simplify(rc_model)
equations(sys)
```

$$\frac{\mathrm{d}capacitor_{+}v\left(t\right)}{\mathrm{d}t} = \frac{capacitor_{+}i\left(t\right)}{capacitor_{+}C}$$

Structural simplification finds the small set of equations to solve

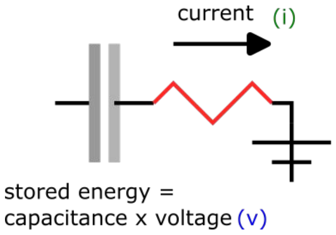
How Acausal Modeling Works: Example

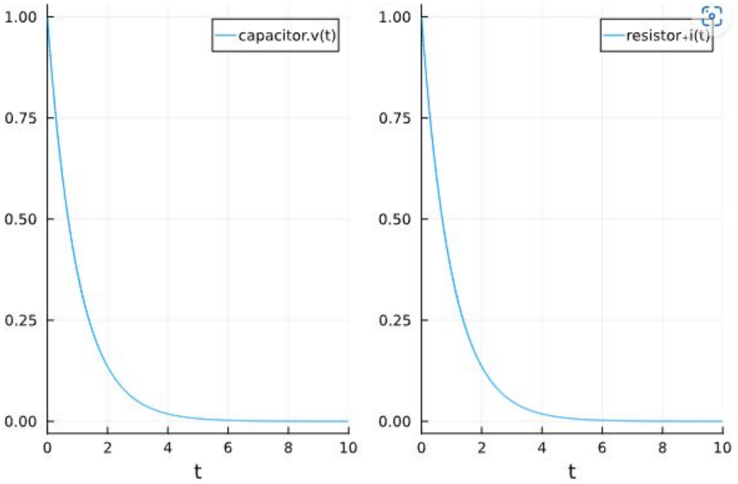
```
using ModelingToolkitStandardLibrary.Electrical, ModelingToolkit, DifferentialEquations
using Plots

@parameters t

@named resistor = Resistor(R = 1)
@named capacitor = Capacitor(C = 1)
@named ground = Ground()

eqs = [connect(capacitor.p, resistor.p)
    connect(resistor.n, ground.g, capacitor.n)]

@named model = ODESystem(eqs, t; systems = [resistor, capacitor, ground])

sys = structural_simplify(model)

println.(equations(sys))
```

```
Differential(t)(capacitor₊v(t)) ~ capacitor₊i(t) / capacitor₊C
```

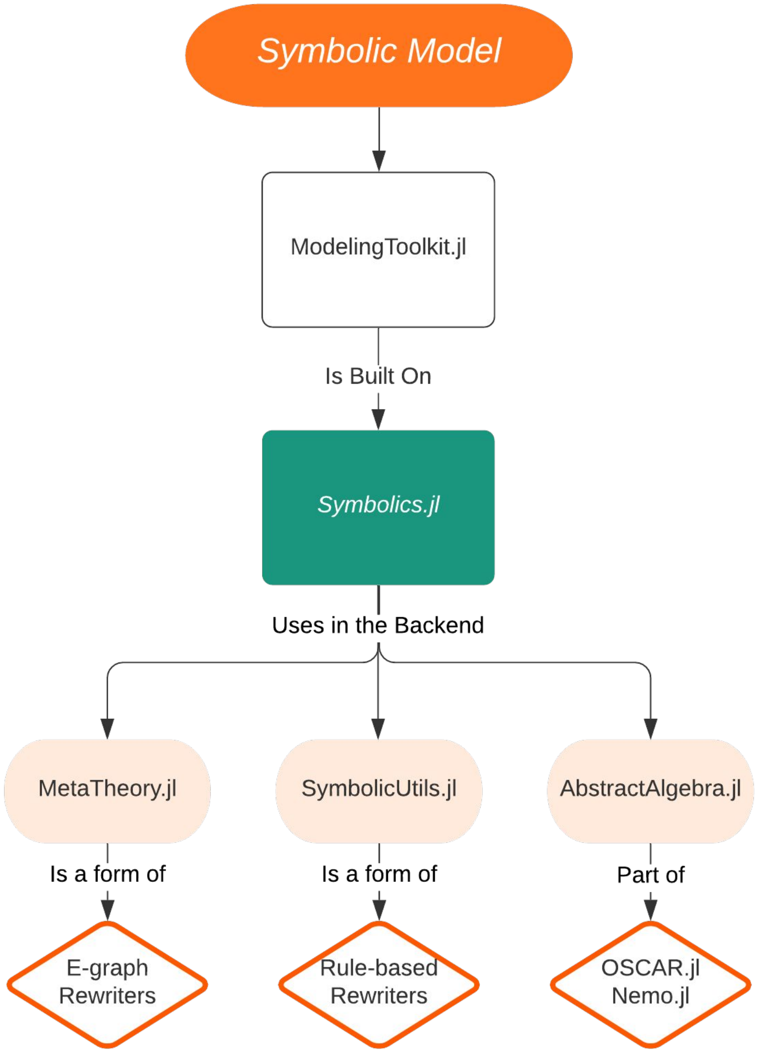
The solution shows what we would expect, a non-linear dissipation of voltage and related decrease in current flow...

```
prob = ODEProblem(sys, [1.0], (0, 10.0), [])
sol = solve(prob)

p1 = plot(sol, idxs = [capacitor.v])
p2 = plot(sol, idxs = [resistor.i])
plot(p1, p2)
```





ModelingToolkit's Symbolic-Numeric Manipulations




```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))
```

Symbolics.jl

MTK

DiffEq

ModelingToolkit: Acausal Component-Based Modeling Heavily Inspired By Modelica

```julia
@mtkmodel RCModel begin
  @components begin
    resistor = Resistor(R = 1.0)
    capacitor = Capacitor(C = 1.0)
    source = ConstantVoltage(V = 1.0)
    ground = Ground()
  end
  @equations begin
    connect(source.p, resistor.p)
    connect(resistor.n, capacitor.p)
    connect(capacitor.n, source.n)
    connect(capacitor.n, ground.g)
  end
end

@mtkbuild rc_model = RCModel(resistor.R = 2.0)
u0 = [rc_model.capacitor.v => 0.0]
prob = ODEProblem(rc_model, u0, (0, 10.0))
sol = solve(prob)
plot(sol)
```

- Fully open source modeling language
- Comes with the “standard” transformations required for component-based modeling (tearing, Pantelides algorithm, etc.)
- Fully open source standard library based on the Modelica Standard Library
  - Currently incomplete and taking contributions!
- Allows users to customize and write their own symbolic model transformations and alternative front ends

Example of Tearing Nonlinear Systems

```julia
@variables u1 u2 u3 u4 u5
eqs = [
    0 ~ u1 - sin(u5),
    0 ~ u2 - cos(u1),
    0 ~ u3 - hypot(u1, u2),
    0 ~ u4 - hypot(u2, u3),
    0 ~ u5 - hypot(u4, u1),
]
@named sys = NonlinearSystem(eqs, [u1, u2, u3, u4, u5], [])
```

$$
\begin{aligned}
0 &= u1 - \sin\left(u5 \right) \\
0 &= u2 - \cos\left(u1 \right) \\
0 &= u3 - \mathrm{hypot}\left(u1, u2 \right) \\
0 &= u4 - \mathrm{hypot}\left(u2, u3 \right) \\
0 &= u5 - \mathrm{hypot}\left(u4, u1 \right)
\end{aligned}
$$

Example of Tearing Nonlinear Systems

```
sys = structural_simplify(sys)
```

$$0 = u5 - \mathrm{hypot}\left(u4, u1 \right)$$

It automatically reduced your 5 equation system to 1!

```
observed(sys)
```

$$\begin{aligned}
u1 &= \sin\left(u5 \right) \\
u2 &= \cos\left(u1 \right) \\
u3 &= \mathrm{hypot}\left(u1, u2 \right) \\
u4 &= \mathrm{hypot}\left(u2, u3 \right)
\end{aligned}$$

Example of Tearing Nonlinear Systems

```julia
u0 = [u5 .=> 1.0]
prob = NonlinearProblem(sys, u0)
sol = solve(prob, NewtonRaphson())
```

```
u: 1-element Vector{Float64}:
 1.6069926947050053
```

Only solves one equation numerically

```julia
sol[u1]
```

```
0.9993449829954304
```

```julia
sol[u2]
```

```
0.5408533677250150
```

But can generate the other variables

Soon: Exact ODE Reduction

$$\begin{cases} \dot{x}_1 = x_1^2 + 2x_1x_2, \\ \dot{x}_2 = x_2^2 + x_3 + x_4, \\ \dot{x}_3 = x_2 + x_4, \\ \dot{x}_4 = x_1 + x_3 \end{cases}$$

An example of an exact reduction in this case would be the following set of new variables

$$y_1 = x_1 + x_2 \quad \text{and} \quad y_2 = x_3 + x_4$$

The important feature of variables y_1, y_2 is that their derivatives can be written in terms of y_1 and y_2 only:

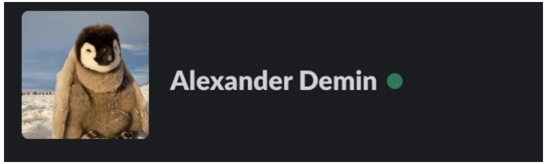
$$\dot{y}_1 = \dot{x}_1 + \dot{x}_2 = y_1^2 + y_2$$

and

$$\dot{y}_2 = \dot{x}_3 + \dot{x}_4 = y_1 + y_2$$

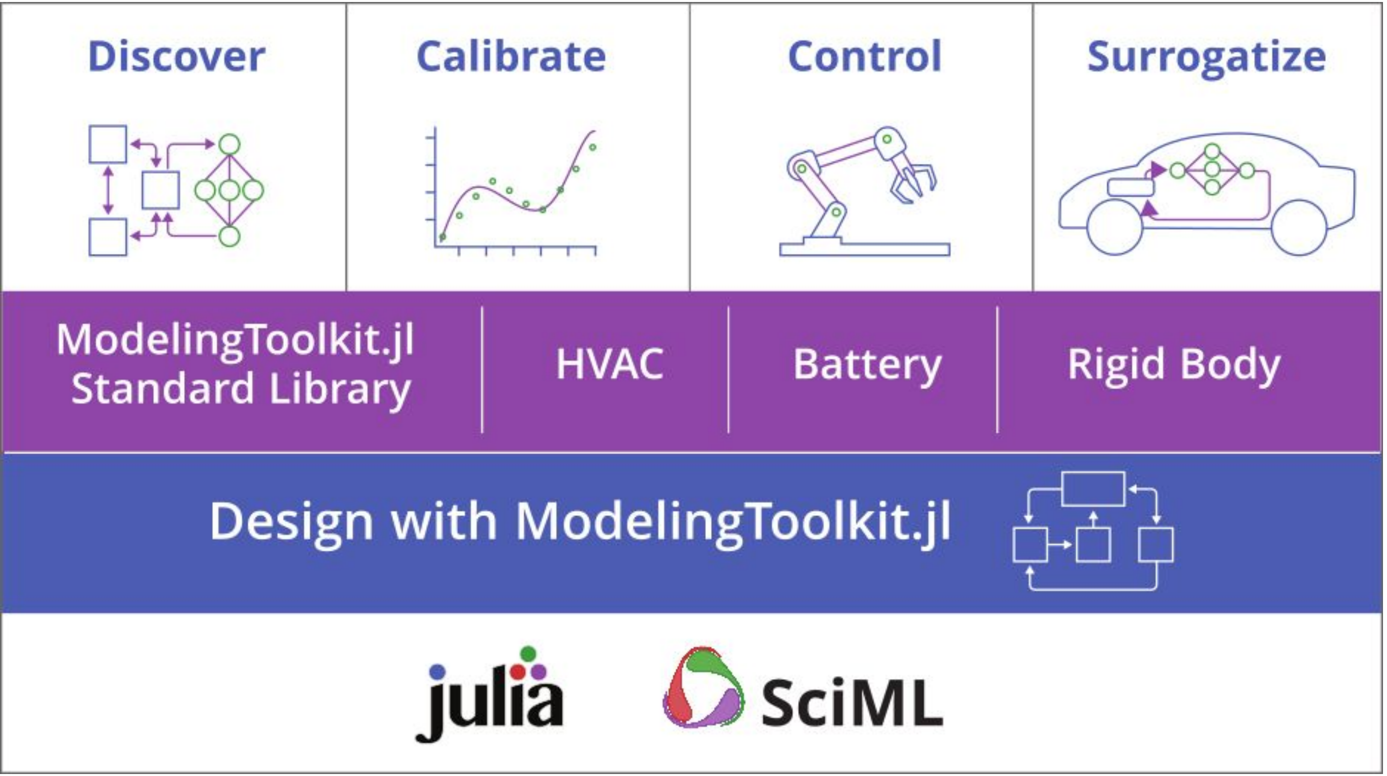
Therefore, the original system can be **reduced exactly** to the following system:

$$\begin{cases} \dot{y}_1 = y_1^2 + y_2, \\ \dot{y}_2 = y_1 + y_2 \end{cases}$$



JuliaSim

Building an Ecosystem on Open Source Foundations



JuliaSim enhances and extends ModelingToolkit for industrial users

Transform ModelingToolkit models into digital twins with easy calibration to data.

Documentation at: help.juliahub.com

JuliaSim: Accelerate Modeling with Component Libraries

Model Libraries

Standard Library

- Build electrical, mechanical, thermal and magnetic models

HVAC

- Buildings use 40% of the world's energy
- Don't let it use that much of yours

Batteries

- Scale physics-based models to large packs
- Understand thermal and deradation properties

Multibody

- Models, robots, engines and turbines
- Mix rigid body to mass-spring components

4 more libraries in the roadmap:

- Media
- Fluid
- Aerial Vehicles
- Process Modeling

This roadmap is not fixed and is looking for input from you!

Catapult Project

10/11/2022

Brad Carman

Model History: >1,000x over Simulink, and Beyond

2.5kHz → 10kHz →

2000

- Inverse Model: Transfer functions
- Forward Model: **Simulink**

2014

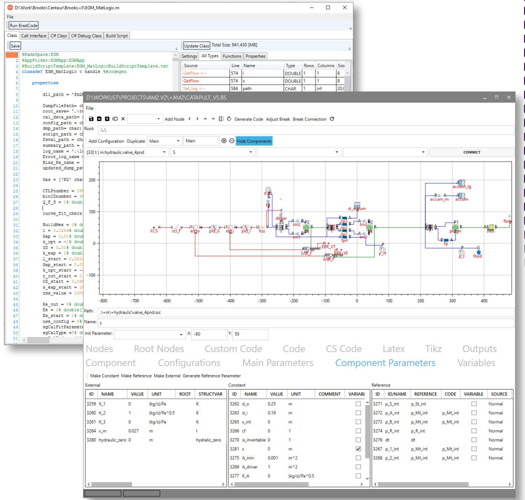
- I joined Instron
- Built Implicit Newton/Euler Equation Based model in pure **Matlab** with inverse and subset model generator using Symbolic Toolbox
- Increased model accuracy with elimination of assumptions and increased complexity
- Worked well, but...
  - *Slow*
  - *Hard to update and maintain*

2017

- Attempted to move to **SimScape**
- Successfully transitioned model with improved speed, but required many workarounds and hacks
- *Never released...*

2020

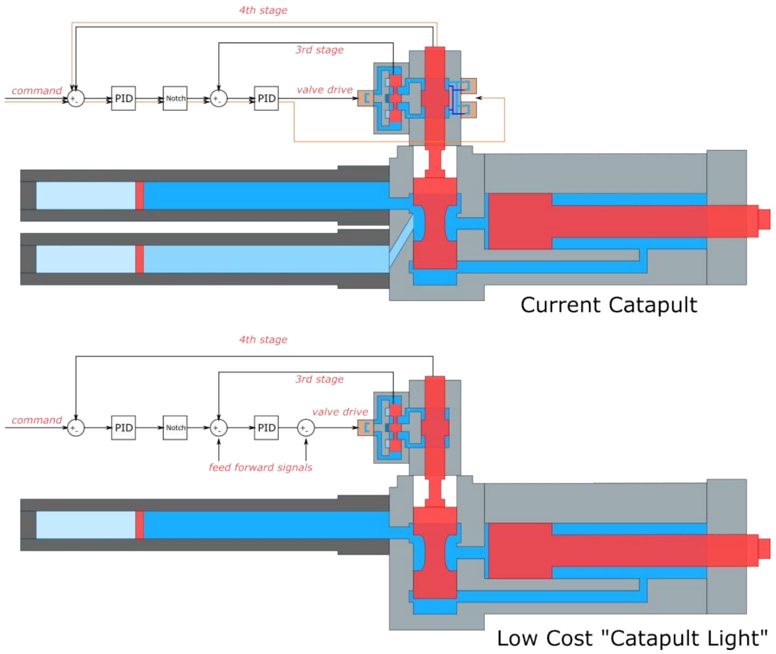
- Moved to **Julia**
- Developed *EmbeddedJulia* library, *ModelingToolkitComponents.jl* and successfully transitioned model to **JuliaSim**



Matlab2CSharp and SimScape Manager

>1000x performance improvements over Simulink!

Catapult Light Design using JuliaSim

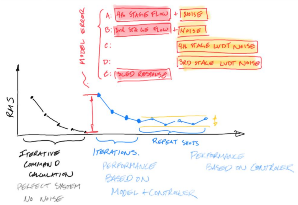


Goal: Eliminate Expensive Multi-Mode System, Design Low Cost Single Mode "Catapult Light" System

Strategy: optimize controller and hardware to provide acceptable performance.

How: use simulation to optimize controller configuration and tuning, real life testing is prohibitive in cost and time. Simulation required for 1 data point is:

- 25 runs for iterative command calculation
- 25 runs for simulated iterations
- 5 runs for repeat shots
- x10 signals = total of 550 model runs



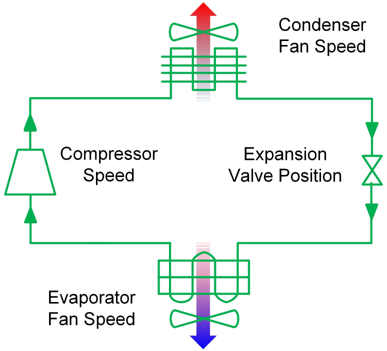
Equivalent to ~2 days of real life testing

Current Matlab simulation time (1 data point): 10 minutes
Required Data Points for Iterative Design Optimization: 1000+
□ 166+ Days of Matlab Simulation Time
□ 8 Hours JuliaSim Time

Accelerated Simulation of HVAC Systems

- Model of vapor compression cycle model
- Contains 8,000 stiff differential algebraic equations
- Reference Dymola model took 35.3 seconds to simulate.
- JuliaSim version took 5.8 seconds.
- Speed of factor of nearly **6x**.

MITSUBISHI ELECTRIC
Changes for the Better

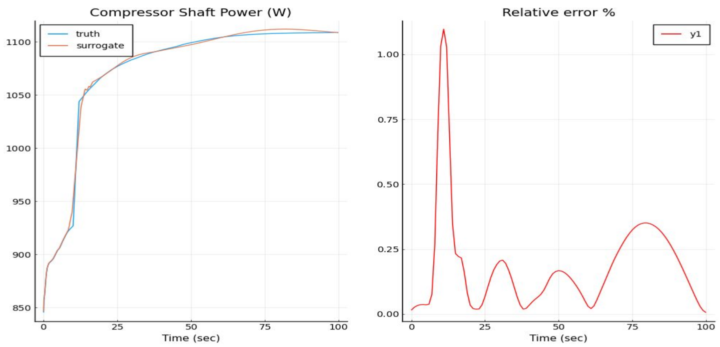


Accelerated Simulation of HVAC Systems

Next step, create surrogate model:

- Concerned with 20 specific signals inside the HVAC system
- Surrogate was up to **95x** faster than JuliaSim version
- Total speed up Dymola→Surrogate: **570x**

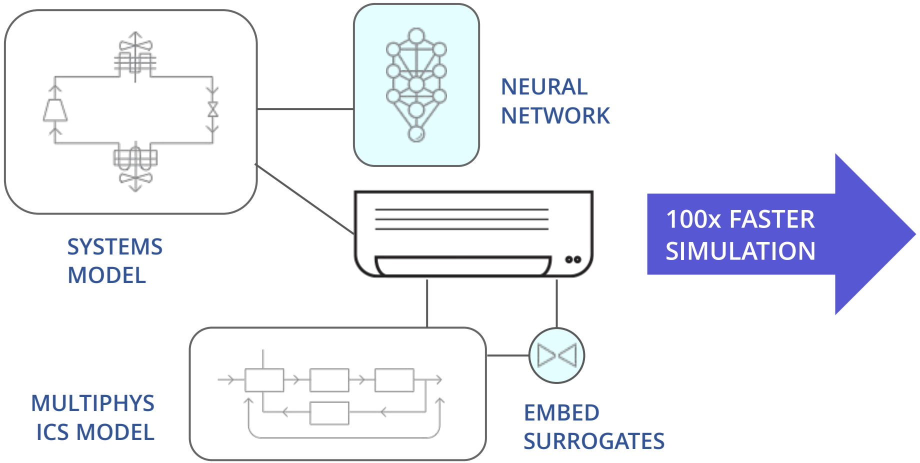
Training set size	Reservoir size	Prediction time	Speedup over baseline
100	1000	0.06 s	95x
1000	2000	0.56 s	10x



JuliaSim: Surrogate Components

JuliaSim Surrogates

You bring physics, we bring machine learning.
Together we achieve fast simulation.



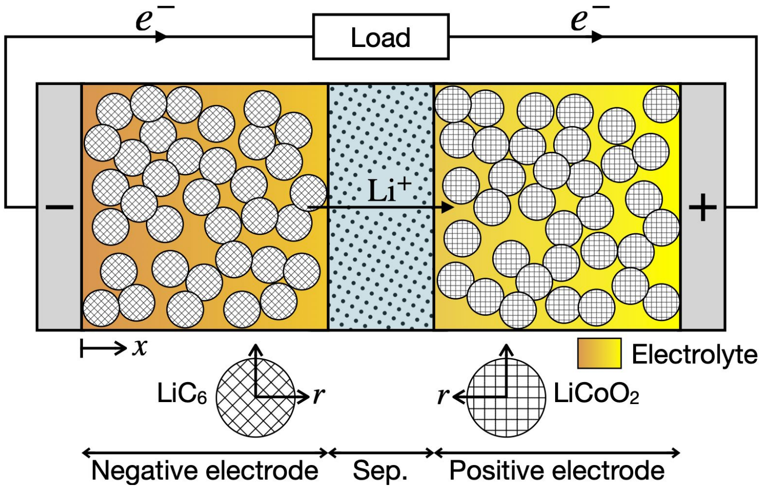
Accelerate large simulations with ML-surrogates

Connect surrogate models in Modeling Toolkit

Total speedup **570x**

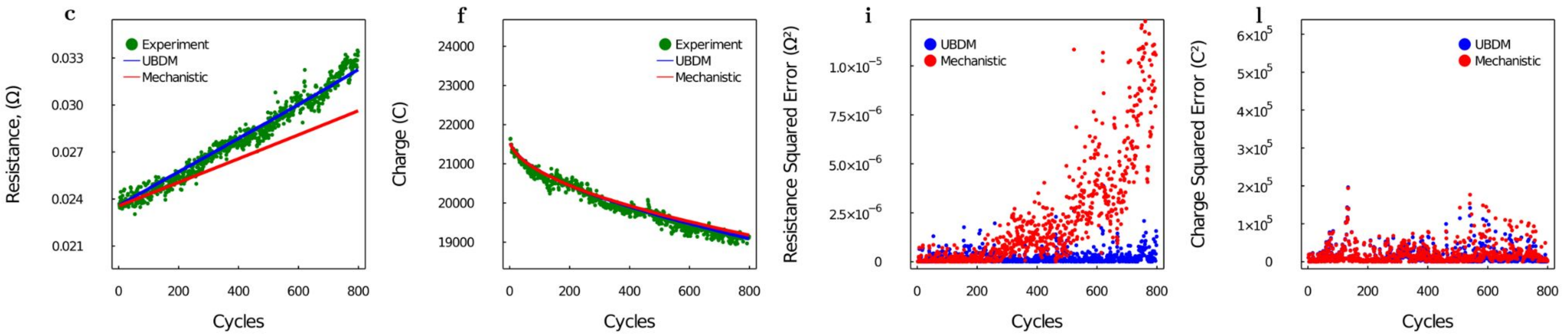
JuliaSim Batteries overview

- 2D electrochemical model of a single battery
  - 300 equations, 5 ms solve time
- Battery pack: repeat the model 200 times
  - Very long simulation time
  - Most tools cannot scale physically-accurate models to full packs



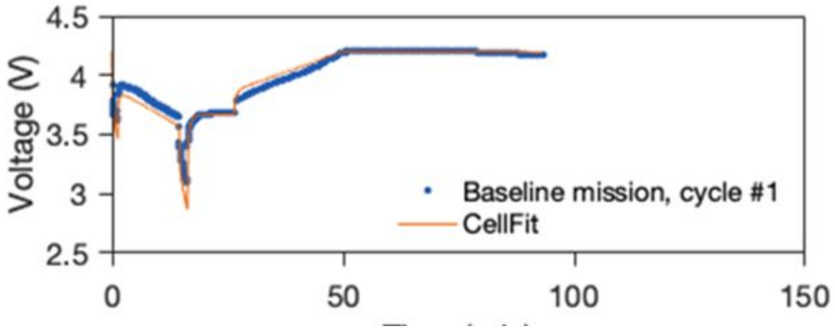
JuliaHub

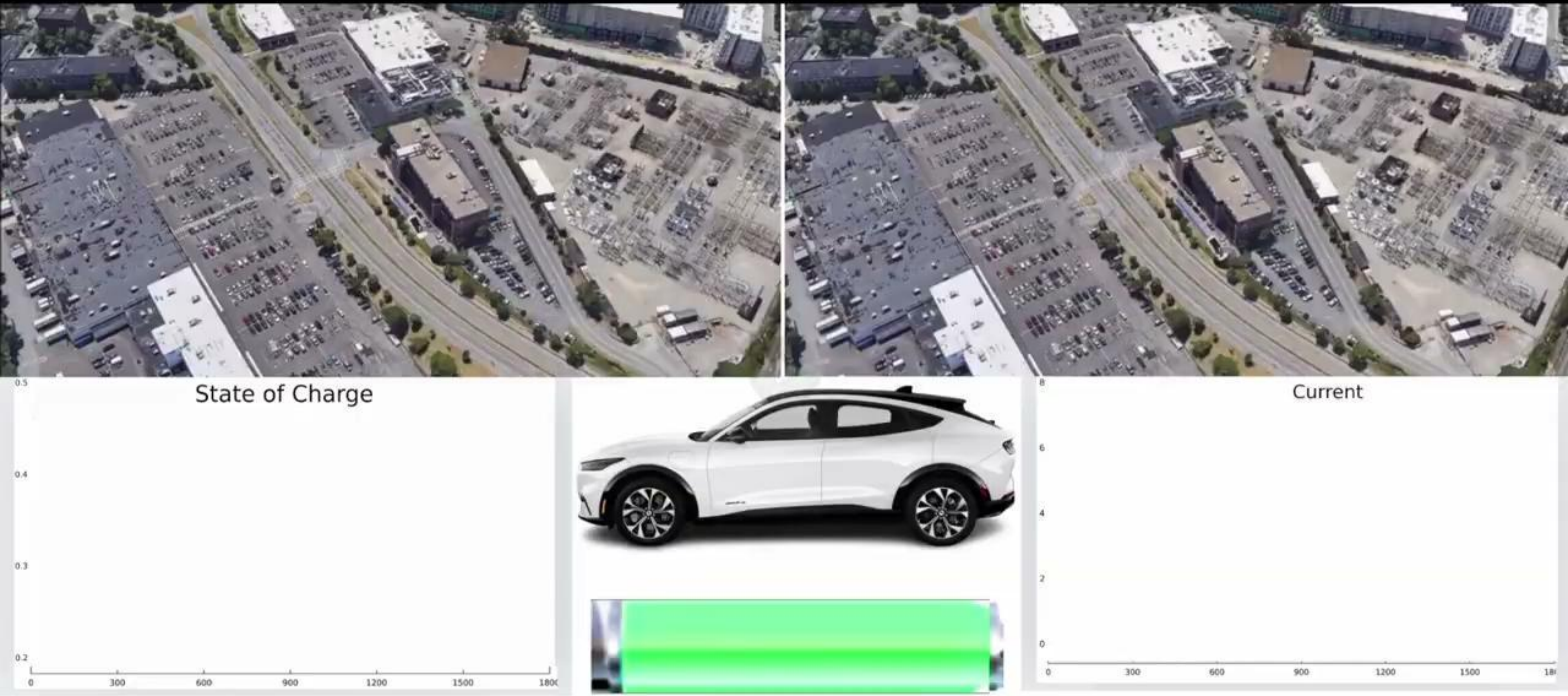
Universal Differential Equations Generate More Accurate Models of Battery Degradation



Researchers at CMU Used Universal Differential Equations to Improve Models of Battery Degradation to Suggest Better Batter Materials

UBDM = Universal Battery Degradation Model



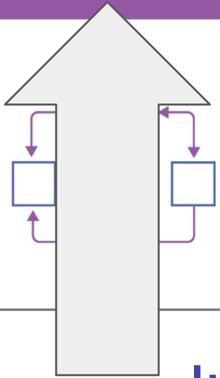

State of Charge
0.5
0.4
0.3
0.2
0
300
600
900
1200
1500
Current
8
6
4
2
0
0
300
600
900
1200
1500

JuliaHub

Introducing: **JuliaSimCompiler**
Scaling Symbolic-Numerics for ML

Introducing JuliaSimCompiler

Design	Discover	Calibrate	Control	Surrogatize
• Build realistic physical models with minimal code • Run simulations 100x faster	• Use Machine Learning to autocomplete models • Discover missing physics	• Turn models into Digital Twins • Robust nonlinear fitting with automatic differentiation	• Build robust nonlinear controls • Deploy Model-Predictive Controllers (MPC)	• Train neural networks to match models • Accelerate fast simulations by another 100x



Acausal model compilers automatically simplify and improve model code.

But can they achieve top performance on large-scale models?

JuliaSimCompiler: Better scaling of ModelingToolkit models

JuliaSimCompiler: Accelerated ModelingToolkit

MTK

```
sys = structural_simplify(complete_motor)
```

JuliaSimCompiler

```
using JuliaSimCompiler
complete_motor_ir = IRSystem(complete_motor)
sys_ir = structural_simplify(complete_motor_ir)
```

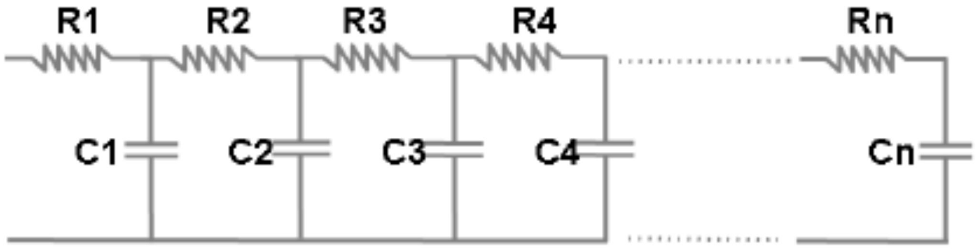
2 lines of code to turn on, enables enormous scalability improvements

Solves a major scaling problem in acasual systems

Conclusion: ModelingToolkit is a widely used open modeling platform, and with JuliaSim it's also the most scalable.

Loop Rerolling

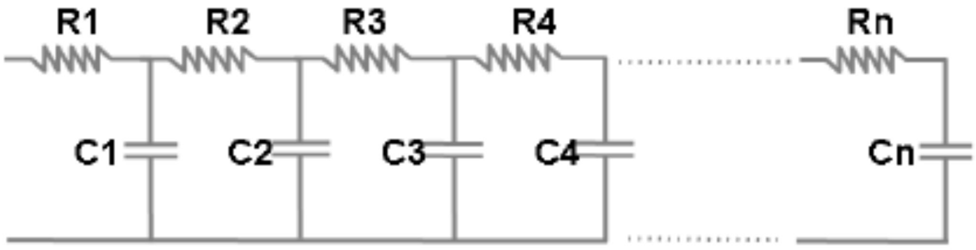
```
systems = @named begin
    sine = Sine(frequency = 10)
    source = Voltage()
    resistors[1:n] = Resistor()
    capacitors[1:n] = Capacitor()
    ground = Ground()
end;
```



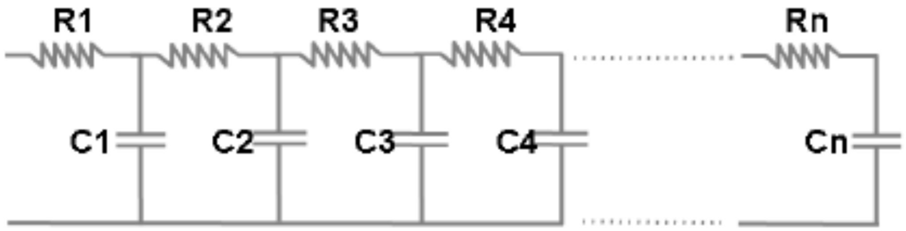
Loop Rerolling

```
resistors_1.v(t) ~ resistors_1.p.v(t) - resistors_1.n.v(t)
0 ~ resistors_1.p.i(t) + resistors_1.n.i(t)
resistors_1.i(t) ~ resistors_1.p.i(t)
resistors_1.v(t) ~ resistors_1.R*resistors_1.i(t)
resistors_2.v(t) ~ -resistors_2.n.v(t) + resistors_2.p.v(t)
0 ~ resistors_2.p.i(t) + resistors_2.n.i(t)
resistors_2.i(t) ~ resistors_2.p.i(t)
resistors_2.v(t) ~ resistors_2.R*resistors_2.i(t)
```

Variable classes:
{{resistors_1.v, resistors_2.v, ...},
{resistors_1.p.v, resistors_2.p.v}, ...}
Equation classes:
{$0 = f_1(x, y, z) = x - (y - z)$,
$0 = f_2(x, y) = x + y$, ...}

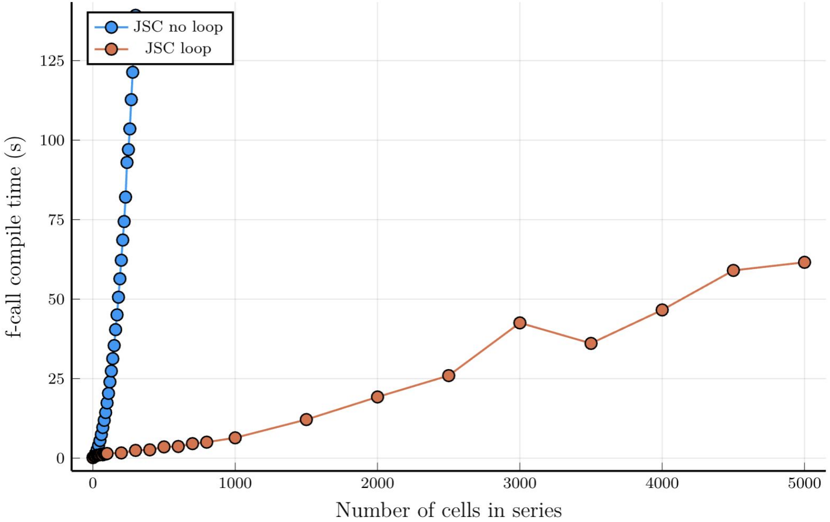


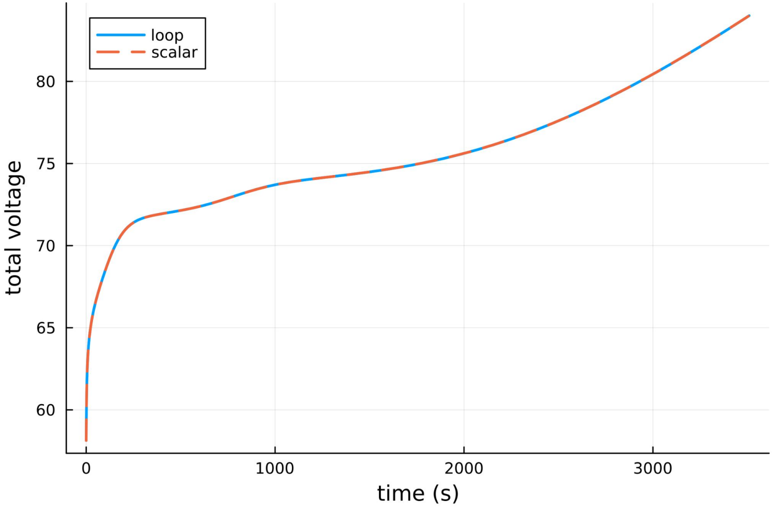
Loop Rerolling




```
for var"%33" = 1:97
    var"%34" = var"%33" - var"%32"
    var"%35" = var"%29" + var"%34"
    var"%36" = Base.getindex(var"###in 1###", var"%35")
    var"%37" = var"%30" + var"%34"
    var"%38" = Base.getindex(var"###in 1###", var"%37")
    var"%39" = var"%31" + var"%34"
    var"%40" = Base.getindex(var"###in 1###", var"%39")
    var"%41" = var"%24" * var"%38"
    var"%42" = var"%41" + var"%36"
    var"%43" = var"%40" + var"%42"
    var"%44" = var"%31" + var"%33"
    var"%45" = Base.setindex!(var"###out###", var"%43", var"%44")
end
```

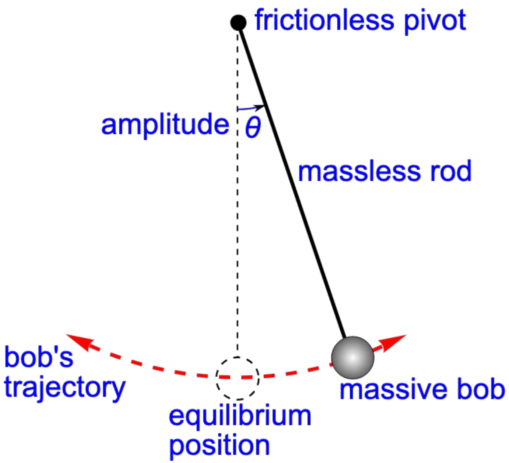
Loop Rerolling on JuliaSimBattery (Single-Particle Model (SPM), Lithium Nickel Manganese Cobalt Oxide (NMC))





JuliaHub

Inlined Linear Solver Optimization



$$\frac{\mathrm{d}\theta(t)}{\mathrm{d}t} = \dot\theta(t)$$
$$\frac{\mathrm{d}\dot\theta(t)}{\mathrm{d}t} = \theta\_tt(t)$$
$$0 = -L\sin(\theta(t))\lambda(t) - L\theta\_tt(t)\cos(\theta(t)) + (\dot\theta(t))^2 L\sin(\theta(t))$$
$$0 = -g + L\sin(\theta(t))\theta\_tt(t) - L\cos(\theta(t))\lambda(t) + (\dot\theta(t))^2 L\cos(\theta(t))$$

Algebraic variables: λ, θ_tt 😱
But they are linear! 😏🚀

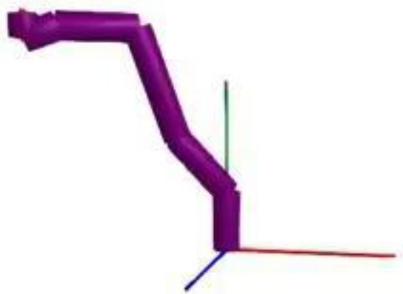
Inlined Linear Solver Optimization

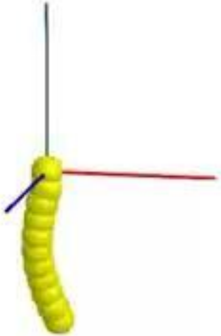
```
julia> ss = structural_simplify(sys);
┌ Info:
│   A =
│    6×6 Matrix{Union{Float64, IRElement}}:
│     -1.0   0.0   0.0   0.0 │ -(%9)           0.0
│      0.0  -1.0   0.0   0.0 │ (-1.0 * %18)   0.0
│      0.0   0.0  -1.0   0.0 │ 0.0             (%32 * cos(%47))
│      0.0   0.0   0.0  -1.0 │ 0.0             (%32 * -(sin(%110)))
│     ───────────────────────┼──────────────────────────────────────
│      0.0   1.0   0.0  -1.0 │ 0.0             0.0
│      1.0   0.0  -1.0   0.0 │ 0.0             0.0
│   b =
│    6-element Vector{Union{Float64, IRElement}}:
│     0.0
│      -(-(%16))
│      -((%32 * ((-(sin(%84)) * %24) * %24)))
│      -((%32 * ((-1.0 * (cos(%138) * %24)) * %24)))
│     0.0
│     0.0
│   vars[svar] =
│    6-element Vector{IRElement}:
│     Dt(v₁, 1, true)
│     Dt(v₂, 1, true)
│     Dt(q₁, 2, true)
│     Dt(q₂, 2, true)
│     λ
└     Dt(θ, 2, true)
```

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}.$$

$$\begin{aligned} \tilde{x}_1 &:= A_{1,1}^{-1} b_1 \\ F_1 &:= A_{1,1}^{-1} A_{1,2} \\ F_2 &:= A_{2,2} - A_{2,1} F_1 \\ F_2 x_2 &= b_2 - A_{2,1} \tilde{x}_1 \\ x_1 &= \tilde{x}_1 - F_1 x_2. \end{aligned}$$

Inlined Linear Solver Optimization: Multibody





JuliaHub

Thanks!

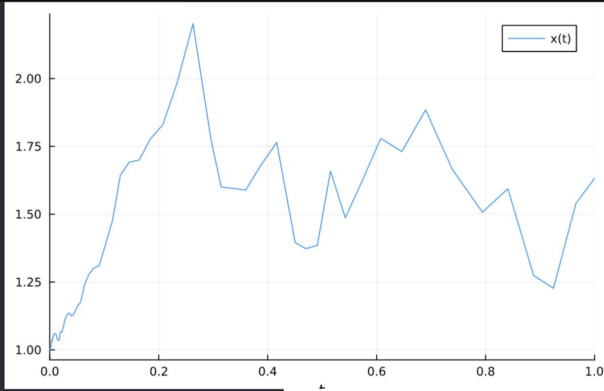
December, 2023

JuliaHub

Some Speical Things in ModelingToolkit

ModelingToolkit: Taking Acausal Component-Based Modeling Beyond DAEs

```julia
using ModelingToolkit, StochasticDelayDiffEq, Plots
@variables t x(..)
@parameters a=-4.0 b=-2.0 c=10.0 α=-1.3 β=-1.2 γ=1.1
D = Differential(t)
@brownian η
τ = 1.0
eqs = [D(x(t)) ~ a * x(t) + b * x(t - τ) + c + (α * x(t) + γ) * η]
@mtkbuild sys = System(eqs)
prob_mtk = SDDEProblem(sys, [x(t) => 1.0 + t], (0.0,1.0); constant_lags = (τ,));
sol_mtk = solve(prob_mtk, RKMil())
plot(sol_mtk)
```



- Sets up a stochastic delay differential equation (SDDE) with driving white noise
- Solved using (implicit) high-order adaptive SDDE solvers
- Can be used to model difficult components like sensors

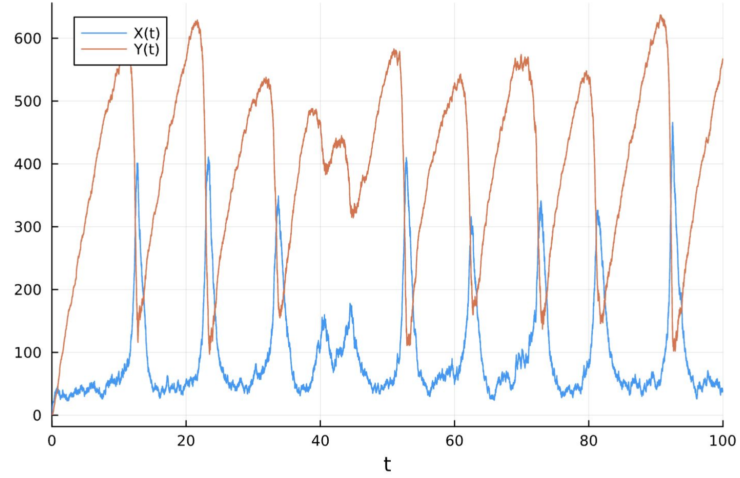
ModelingToolkit Feature Highlight: Sophisticated Stochastic Modeling

```julia
using Catalyst, DifferentialEquations, Plots, Latexify
rn = @reaction_network begin
  @parameters c₁ c₂ c₃ c₄ Ω
  (c₁/Ω^2), 2X + Y → 3X
  (c₂), X → Y
  (c₃*Ω, c₄), 0 ↔ X
end
p = [:c₁ => 0.9, :c₂ => 2, :c₃ => 1, :c₄ => 1, :Ω => 100]
u₀ = [:X => 1, :Y => 1]
tspan = (0., 100.)

dprob = DiscreteProblem(rn, u₀, tspan, p)
jprob = JumpProblem(rn, dprob, Direct())
sol = solve(jprob, SSAStepper(), saveat=10.)
plot(sol)
```

Chemical Reaction Systems as Stochastic Models

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

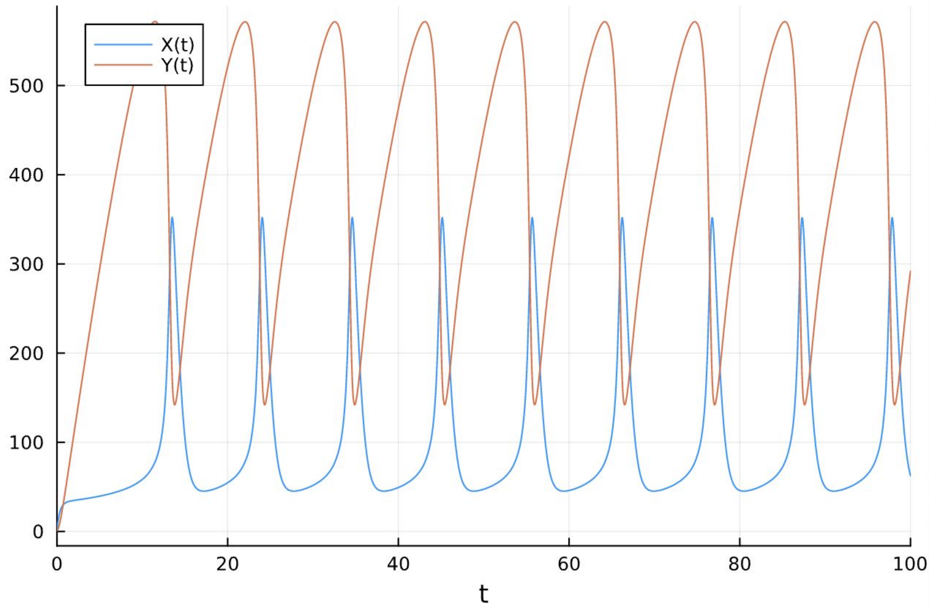
$$\frac{dP(\mathbf{n}, t)}{dt} = \sum_{r=1}^{R} \left[a_r(\mathbf{n} - S_r) P(\mathbf{n} - S_r, t) - a_r(\mathbf{n}) P(\mathbf{n}, t) \right]$$



ModelingToolkit Feature Highlight: Sophisticated Stochastic Modeling

Transform the stochastic model into an approximating deterministic model:

```
prob = ODEProblem(rn, u₀, tspan, p)
sol = solve(prob)
plot(sol)
```



ModelingToolkit Feature Highlight: Sophisticated Stochastic Modeling

Transform the stochastic model into an
approximating deterministic model
Of means and moments

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

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

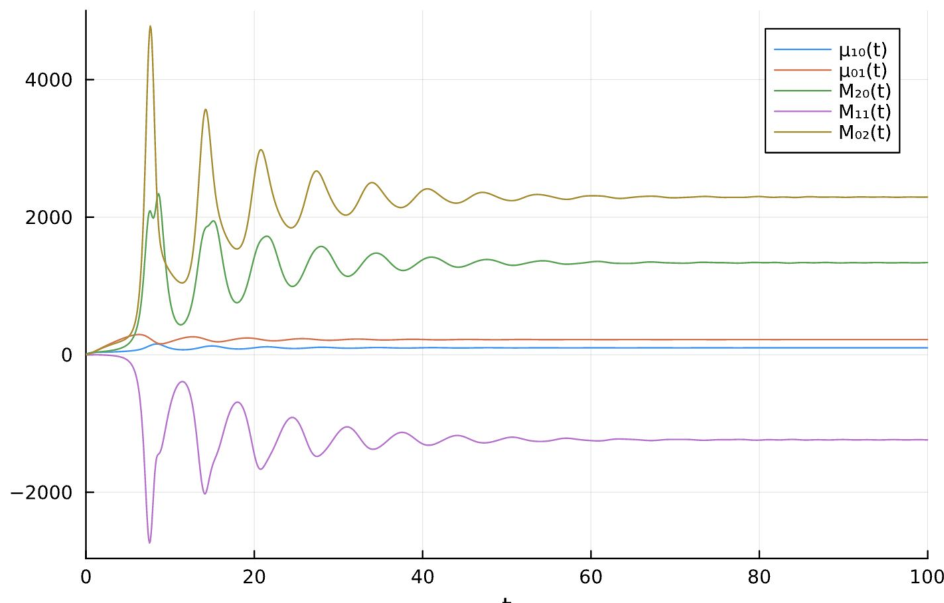
You can write out the moments…

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

This sounds like a problem for a symbolic modeling tool to figure out for you…

ModelingToolkit Feature Highlight: Sophisticated Stochastic Modeling

```julia
using MomentClosure
central_eqs = generate_central_moment_eqs(rn, 2, combinatoric_ratelaws=false)
closed_raw_eqs = moment_closure(central_eqs, "normal")
u₀map = deterministic_IC([1, 1], closed_raw_eqs)
tspan = (0., 100.)
p = [0.9,2,1,1,100]
oprob = ODEProblem(closed_raw_eqs, u₀map, tspan, p)
sol = solve(oprob)
plot(sol)
```

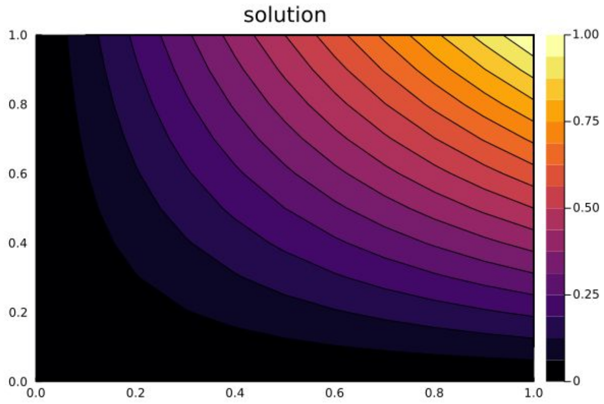


Solution for the means and variances computed via an ODE!

ModelingToolkit PDEs: Method Of Lines Finite Difference

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

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

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

# Boundary conditions
bcs = [u(0,y) ~ 0.f0, u(1,y) ~ -sin(pi*1)*sin(pi*y),
       u(x,0) ~ 0.f0, u(x,1) ~ -sin(pi*x)*sin(pi*1)]
# Space and time domains
domains = [x ∈ Interval(0.0,1.0),
           y ∈ Interval(0.0,1.0)]
pde_system = PDESystem(eq,bcs,domains,[x,y],[u])
```

```julia
# Transform it into a symbolic NonlinearSystem via Finite Differences
discretization = MOLFiniteDifference([x=>dx,y=>dy], nothing, centered_order=2)

prob = discretize(pdesys,discretization)
sol = solve(prob)

using Plots
xs,ys = [infimum(d.domain):dx:supremum(d.domain) for d in domains]
u_sol = reshape(sol.u, (length(xs),length(ys)))
plot(xs, ys, u_sol, linetype=:contourf,title = "solution")
```

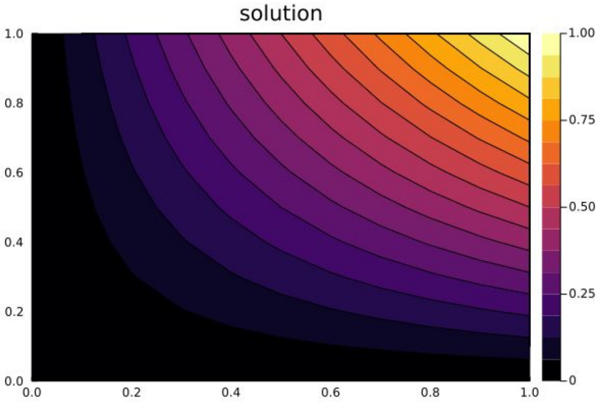


ModelingToolkit PDEs: Physics-Informed Neural Networks

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

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

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

# Boundary conditions
bcs = [u(0,y) ~ 0.f0, u(1,y) ~ -sin(pi*1)*sin(pi*y),
       u(x,0) ~ 0.f0, u(x,1) ~ -sin(pi*x)*sin(pi*1)]
# Space and time domains
domains = [x ∈ Interval(0.0,1.0),
           y ∈ Interval(0.0,1.0)]
pde_system = PDESystem(eq,bcs,domains,[x,y],[u])
```

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

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



ModelingToolkit PDEs: Extensible PDE Interface

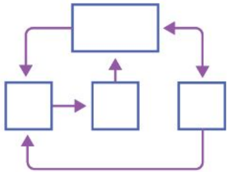
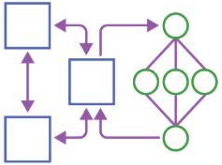
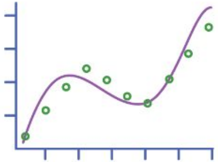
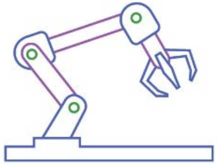
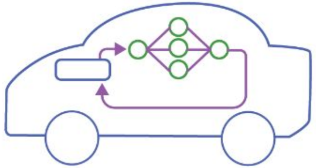
```julia
using ModelingToolkit
import ModelingToolkit: Interval, infimum, supremum

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

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

# Boundary conditions
bcs = [u(0,y) ~ 0.f0, u(1,y) ~ -sin(pi*1)*sin(pi*y),
       u(x,0) ~ 0.f0, u(x,1) ~ -sin(pi*x)*sin(pi*1)]
# Space and time domains
domains = [x ∈ Interval(0.0,1.0),
           y ∈ Interval(0.0,1.0)]
pde_system = PDESystem(eq,bcs,domains,[x,y],[u])
```

Coming soon:

Finite Volume methods
Spectral methods
Finite element methods
…

Attempting to unify the difference PDE discretization methods into a one interface compatible with component-based modeling!

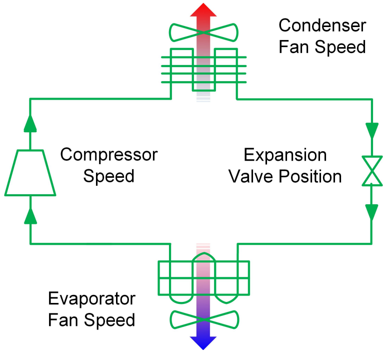
JuliaSim at a Glance

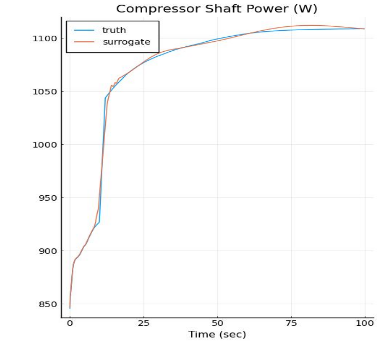
Design	Discover	Calibrate	Control	Surrogatize
• Build realistic physical models with minimal code • Run simulations 100x faster	• Use Machine Learning to autocomplete models • Discover missing physics	• Turn models into Digital Twins • Robust nonlinear fitting with automatic differentiation	• Build robust nonlinear controls • Deploy Model-Predictive Controllers (MPC)	• Train neural networks to match models • Accelerate fast simulations by another 100x
				

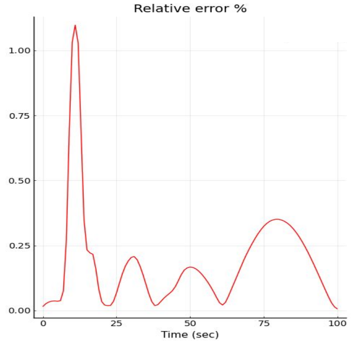
ARPA-E
Accelerated Simulation of Building Energy Efficiency

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

Home · JuliaHub Enterprise - Google Chrome

Home · JuliaHub Enterpr | Main Interface ⊂ JuliaSim

https://juliahub.com/ui/Home

JuliaHub

Press / to search

Hello, JuliaSim

Home
Applications
Notebooks
Projects
Files
Package Registry
Jobs
Datasets
Resources
Admin

JuliaHub is the entry point for all things Julia: explore the ecosystem, contribute packages, and easily run code in the cloud on big machines and on-demand clusters.

Your Jobs

more...

JuliaSim	Stop	Connect	Logs
Julia IDE	Stop	Connect	Logs
JuliaSim	Stop	Connect	Logs
JuliaSim IDE	Stop	Connect	Logs
JuliaSim IDE	Stop	Connect	Logs
JuliaSim	Stop	Connect	Logs
JuliaSim	Stop	Connect	Logs

Applications

more...

Julia SIM

JuliaSim IDE

Access a fast precompiled JuliaSim instance

Launch

Julia SIM

JuliaSim

Modern Modeling and Simulation Powered by Machine Learning

Connect

New Packages

HydroTools	0.1.0
Crossterm	0.2.1
TensorOperationsTBLIS	0.1.0
Vahana	1.0.0
KMA_jll	1.3.21+0
Vensim2MTK	0.1.0
PolarizedTypes	0.1.0
VLBISkyModels	0.2.1
InverseStatMech	1.0.0
SMLMBoxer	0.1.0

Recently Updated Packages

CommonOPF	0.3.3
ModalDecisionTrees	0.1.5
HTMLForge	0.3.0
TestingUtilities	1.6.5
NMF	1.0.2
CellListMap	0.8.21
GeometricSolutions	0.3.14
GeometricBase	0.7.2
PlayingCards	0.3.2
SolverTest	0.3.11

Popular Packages

more...

Popular Tags