Physical modeling in Julia

For those about to control

Acknowledgement

This presentation contains an assortment of content contributed by multiple people

- Chris Rackauckas
- Yingbo Ma
- Probably more, thank you!

Outline

- X Differential equations
- Equation-based modeling
    - Symbolics
    - ModelingToolkit (MTK)
    - Tools on top of MTK
- MTK Standard library
- Current status
- Project ideas

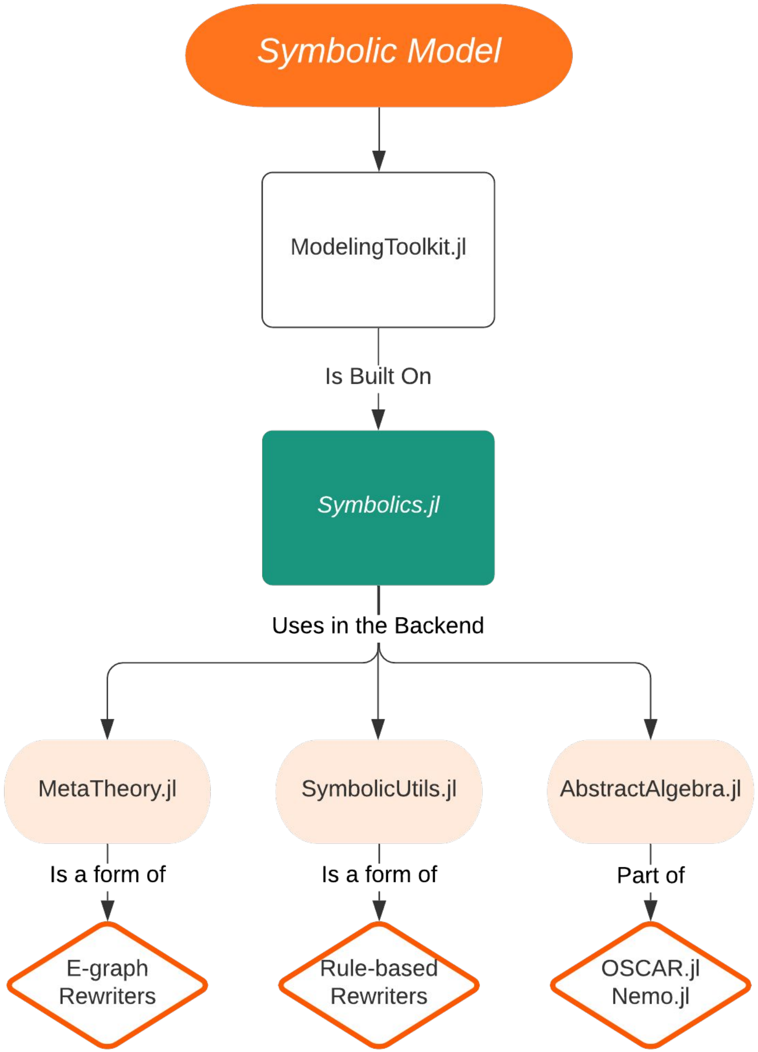
Differential equations code demo

Modeling controlled systems using DifferentialEquations.jl

- Incorporating input data
- State-feedback controller with ZoH

Equation-based modeling

- Symbolics
- ModelingToolkit (MTK)
- Tools on top of MTK




```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

Symbolics is a Type-Based Computer Algebra System (CAS)

```julia
using Symbolics
@variables a # equivalent to a::Real, symtype(a) == Sym{Real}
@variables b[1:2,1:2] # symtype(b) == SymArr{Real}
@variables c::Complex # symtype(c) == Complex{Sym{Real}}
```

The algebra and symbolic simplification rules are type-dependent

Two Worlds of Rewrite Systems Working Together

Rule-Based Rewriter: SymbolicUtils.jl

```
PLUS_RULES = [
    @rule(~x::isnotflat(+) => flatten_term(+, ~x))
    @rule(~x::needs_sorting(+) => sort_args(+, ~x))
    @ordered_acrule(~a::is_literal_number + ~b::is_literal_number => ~a + ~b)

    @acrule(*(~~x) + *(~β, ~~x) => *(1 + ~β, (~~x)...))
    @acrule(*(~α, ~~x) + *(~β, ~~x) => *(~α + ~β, (~~x)...))
    @acrule(*(~~x, ~α) + *(~~x, ~β) => *(~α + ~β, (~~x)...))

    @acrule(~x + *(~β, ~x) => *(1 + ~β, ~x))
    @acrule(*(~α::is_literal_number, ~x) + ~x => *(~α + 1, ~x))
    @rule(+(~~x::hasrepeats) => +(merge_repeats(*, ~~x)...))

    @ordered_acrule((~z::_iszero + ~x) => ~x)
    @rule(+(~x) => ~x)
]
```

Pros: Fast, automatically parallelized, and uses standard rules
Cons: Simplification result is dependent on rule application order!

E-Graph Based Rewriter: MetaTheory.jl

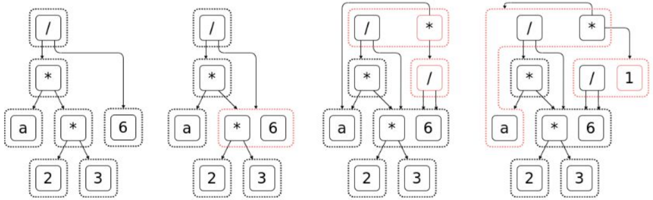


Figure 1: Equality saturation constructs the e-graph from a set of rules applied to an input expression. The four depicted e-graphs represent the process of equality saturation for the equivalent ways to write $a * (2 * 3)/6$. The dashed boxes represent equivalence classes, and regular boxes represent e-nodes.

Pros: Deterministic result based on a cost function
Cons: Requires a separate rule set

High Performance Codegen With Symbolics

```julia
jac = ModelingToolkit.sparsejacobian(vec(du),vec(u))
fjac = ModelingToolkit.build_function(jac,u,
            parallel=ModelingToolkit.MultithreadedForm())[2]
```

```julia
var"##MTKArg#253"[2999], var"##MTKArg#253"[3000], var"##MTKArg#253"[3001], var"##MTKArg#253"[3002], var"##
var"##MTKArg#253"[3018], var"##MTKArg#253"[3019], var"##MTKArg#253"[3020], var"##MTKArg#253"[3021], var"##
var"##MTKArg#253"[3037], var"##MTKArg#253"[3038], var"##MTKArg#253"[3039], var"##MTKArg#253"[3040], var"##
var"##MTKArg#253"[3056], var"##MTKArg#253"[3057], var"##MTKArg#253"[3058], var"##MTKArg#253"[3059], var"##
    begin
        Threads.@spawn begin
                (var"##MTIIPVar#255").nzval[1] = (getproperty(Base, :+))(-400.0, (getproperty(Base, :*))(
                (var"##MTIIPVar#255").nzval[2] = 100.0
                (var"##MTIIPVar#255").nzval[3] = 100.0
                (var"##MTIIPVar#255").nzval[4] = (getproperty(Base, :*))(-1, u_{1,1,2})
                (var"##MTIIPVar#255").nzval[5] = u_{1,1,2}
                (var"##MTIIPVar#255").nzval[6] = 200.0
                (var"##MTIIPVar#255").nzval[7] = (getproperty(Base, :+))(-400.0, (getproperty(Base, :*))(
                (var"##MTIIPVar#255").nzval[8] = 100.0
                (var"##MTIIPVar#255").nzval[9] = 100.0
                (var"##MTIIPVar#255").nzval[10] = (getproperty(Base, :*))(-1, u_{2,1,2})
                (var"##MTIIPVar#255").nzval[11] = u_{2,1,2}
                (var"##MTIIPVar#255").nzval[12] = 100.0
```

Linear indexing of the in-place operations on a sparse matrix

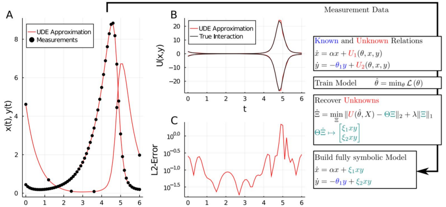
Reduction of computed operations via simplify

Automatically spawns threads based on the number of non-zeroes in the sparse Jacobian and the number of CPU cores

Symbolics code demo

What is ModelingToolkit?

A symbolic language and compilers for models.

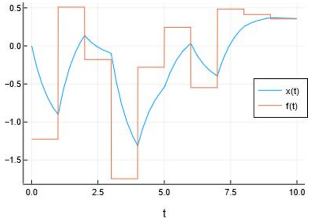


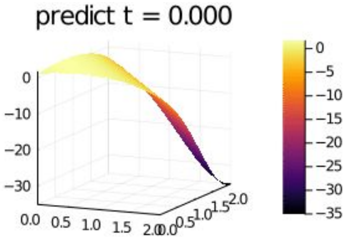
1. MTK is the symbolic side of SciML

Symbolic modeling for all numerical simulation

2. MTK is a symbolic-numeric optimizer

Automatically optimize code for ODE solvers





3. MTK is an acausal modeling system

Like Modelica, SimScape, etc.

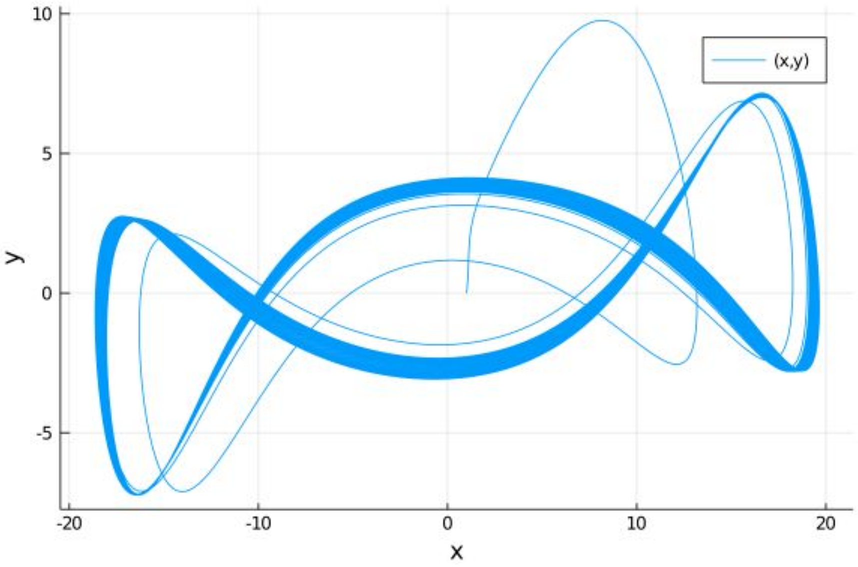
4. MTK is a DSL building tool

Catalyst.jl, PDE interfaces, Optimal control etc.

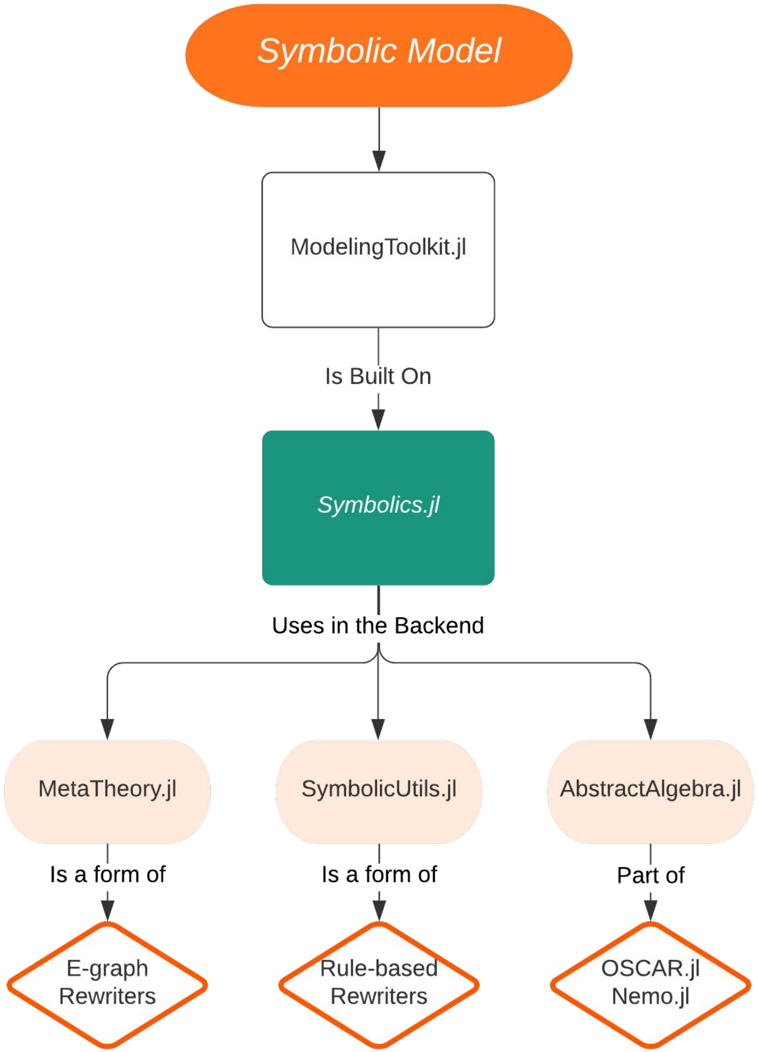
What sets MTK apart from alternatives?

- Acausal, equation based
- Open source
- Julia all the way down
- Exposes symbolic language to the user
- Very wide scope
  - ODE, PDE, … all kinds of DE
  - Optimization
  - Deep learning
  - Anything that benefits from symbolic modeling
- Differentiable

ModelingToolkit.jl – The Modeling Frontend to a Symbolic Ecosystem




```
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))
```



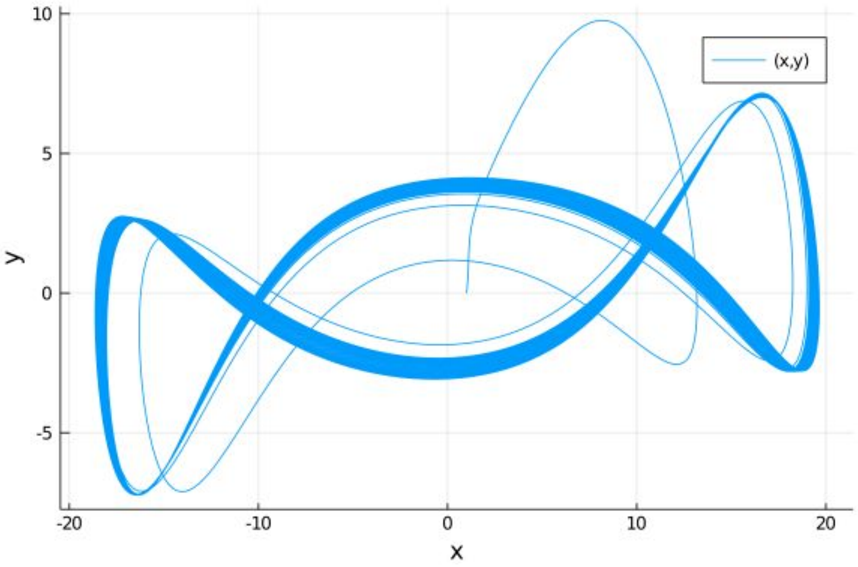

```julia
using ModelingToolkit, OrdinaryDiffEq

@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.jl – The Modeling Frontend to a Symbolic Ecosystem




```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))
```

Specified

Solved for

Dynamics

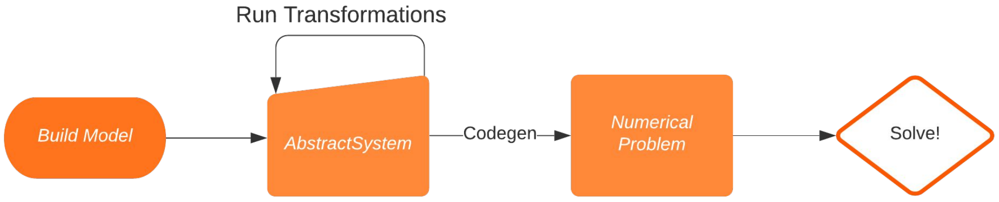
Specify params

ModelingToolkit is "Actually" About Stable Transformations of Models

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

$$\Downarrow$$

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




```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))
```

What is a compiler?

- Transforms code to other code
  - C to assembly
  - Julia to LLVM, LLVM to assembly
  - Complicated model to simple model
- A compiler has one or many *compiler passes*
  - Dead-code elimination
  - Expression rewriting
  - Symbolic simplification X+0 -> x
- MTK model compiler is implemented in Julia
  - Alias elimination, index reduction, tearing, order lowering
  - *You* can write an MTK compiler pass

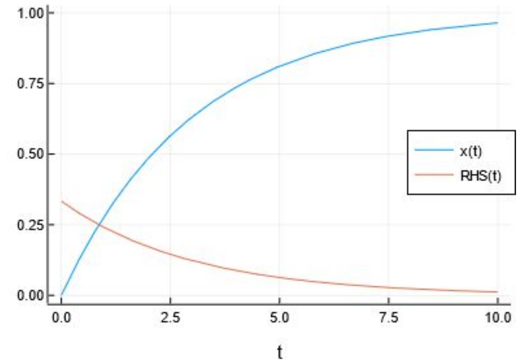
What Kinds of Transformations Do You Get?

- Analytically calculate Jacobians, Hessians, etc.
- Automatically determine sparsity patterns
- Automatically parallelize the generated code
- Automatically simplify the model and eliminate redundant variables
- Automatically transform equations to require positivity
- Etc…

```julia
using DifferentialEquations, ModelingToolkit, Plots

@variables t x(t) RHS(t)  # independent and dependent variables
@parameters τ         # parameters
D = Differential(t) # define an operator for the differentiation w.

# your first ODE, consisting of a single equation, indicated by ~
@named fol_separate = ODESystem([ RHS  ~ (1 - x)/τ,
                                  D(x) ~ RHS ])

simplesys = structural_simplify(fol_separate)
print(equations(simplesys))
#[Differential(t)(x(t)) ~ (τ^-1)*(1 - x(t))]

prob = ODEProblem(structural_simplify(fol_separate), [x => 0.0],
                                           (0.0,10.0), [τ => 3.0])

sol = solve(prob)
plot(sol, vars=[x,RHS])
```

Removes One Equation!

structural_simplify: The typical Modelica Transforms + more



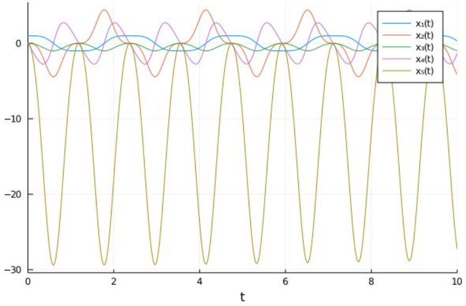
What Kinds of Transformations Do You Get? DAE Index Reduction

```julia
using DifferentialEquations, ModelingToolkit
using LinearAlgebra, Plots

function pendulum!(du, u, p, t)
    x, dx, y, dy, T = u
    g, L = p
    du[1] = dx
    du[2] = T*x
    du[3] = dy
    du[4] = T*y - g
    du[5] = x^2 + y^2 - L^2
    return nothing
end
pendulum_fun! = ODEFunction(pendulum!,
                            mass_matrix=Diagonal([1,1,1,1,0]))
u0 = [1.0, 0, 0, 0, 0]
p = [9.8, 1]
tspan = (0, 10.0)
pendulum_prob = ODEProblem(pendulum_fun!, u0, tspan, p)
sol = solve(pendulum_prob)
```

Let me fix that for you…

```julia
@named traced_sys = modelingtoolkitize(pendulum_prob)
pendulum_sys = structural_simplify(traced_sys)
prob = ODAEProblem(pendulum_sys, [], tspan)
sol = solve(prob, Tsit5(),abstol=1e-8,reltol=1e-8)
plot(sol, vars=states(traced_sys))
```

structural_simplify: The typical Modelica transforms




```
┌ Warning: dt <= dtmin. Aborting. There is either an error in your model specification or the true solution is unstable.
└ @ SciMLBase C:\Users\accou\.julia\packages\SciMLBase\AOoIW\src\integrator_interface.jl:345
```

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!

Composable (Acausal) Modeling via Subsystems

```
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)
         ]

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

Describe how the subsystems relate

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:
Contains main model transforms

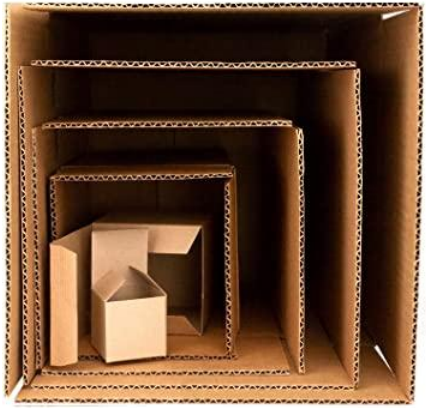
```
sys = structural_simplify(rc_model)
```

```
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)
```

Compose vs. extend

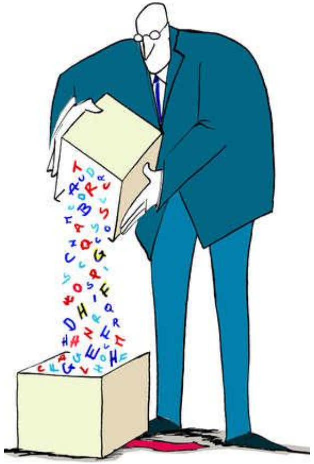
compose

Place an instance of a sub-system into an outer system



extend

Copy the *contents* of sub-system into inheriting system



Compose vs. extend

compose

```
function Inertia(;name, J=1.0, phi_start=0.0, w_start=0.0, a_start=0.0)
   @named flange_a = Flange()
   @named flange_b = Flange()
   @parameters J=J
   sts = @variables begin
       phi(t)=phi_start
       w(t)=w_start
       a(t)=a_start
   end
   eqs = [
       phi ~ flange_a.phi
       phi ~ flange_b.phi
       D(phi) ~ w
       D(w) ~ a
       J*a ~ flange_a.tau + flange_b.tau
   ]
   return compose(ODESystem(eqs, t, sts, [J]; name=name), flange_a, flange_b)
end
julia> @named J = Inertia(J=1)
Model J with 5 equations
States (7):
 phi(t) [defaults to 0.0]
 w(t) [defaults to 0.0]
 a(t) [defaults to 0.0]
 flange_a₊phi(t) [defaults to 0.0]
 flange_a₊tau(t) [defaults to 0.0]
 flange_b₊phi(t) [defaults to 0.0]
 flange_b₊tau(t) [defaults to 0.0]
Parameters (1):
 J [defaults to 1]
```

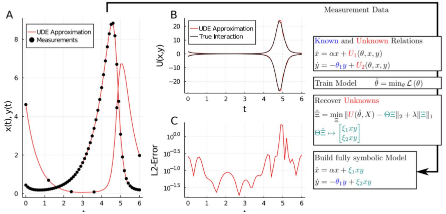
extend

```
function Resistor(;name, R=1.0)
   @named oneport = OnePort()
   @unpack v, i = oneport
   pars = @parameters R=R
   eqs = [
       v ~ i * R
   ]
   extend(ODESystem(eqs, t, [], pars; name=name), oneport)
End

julia> @named R = Resistor(R=1)
Model R with 4 equations
States (6):
 v(t) [defaults to 0.0]
 i(t) [defaults to 0.0]
 p₊v(t) [defaults to 1.0]
 p₊i(t) [defaults to 1.0]
 n₊v(t) [defaults to 1.0]
 n₊i(t) [defaults to 1.0]
Parameters (1):
 R [defaults to 1]
```

MTK code demo

What is ModelingToolkit?

A symbolic language and compilers for models.

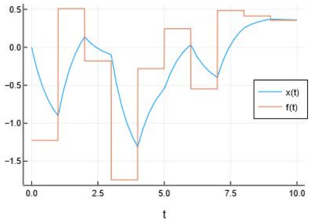


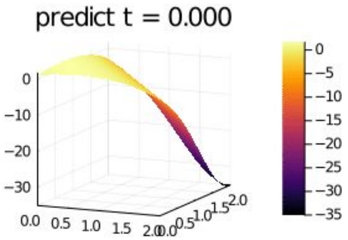
1. MTK is the symbolic side of SciML

Symbolic modeling for all numerical simulation

2. MTK is a symbolic-numeric optimizer

Automatically optimize code for ODE solvers





3. MTK is an acausal modeling system

Think like Modelica, SimScape, etc.

4. MTK is a DSL building tool

Catalyst.jl, PDE interfaces, optimal control

ModelingToolkit has System Types Matching Each SciML Numerical Domain

LinearSolve.jl: Unified Linear Solver Interface

$$A(p)x = b$$

NonlinearSolve.jl: Unified Nonlinear Solver Interface

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

DifferentialEquations.jl: Unified Interface for all Differential Equations

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

$$du = f(u,p,t)dt + g(u,p,t)dW_t$$

$$\vdots$$

GalacticOptim.jl: Unified Optimization Interface

$$\text{minimize } f(u,p)$$
$$\text{subject to } g(u,p) \leq 0, h(u,p) = 0$$

Quadrature.jl: Unified Quadrature Interface

$$\int_{lb}^{ub} f(t,p)dt$$

Unified Partial Differential Equation Interface

$$u_t = u_{xx} + f(u)$$
$$u_{tt} = u_{xx} + f(u)$$
$$\vdots$$

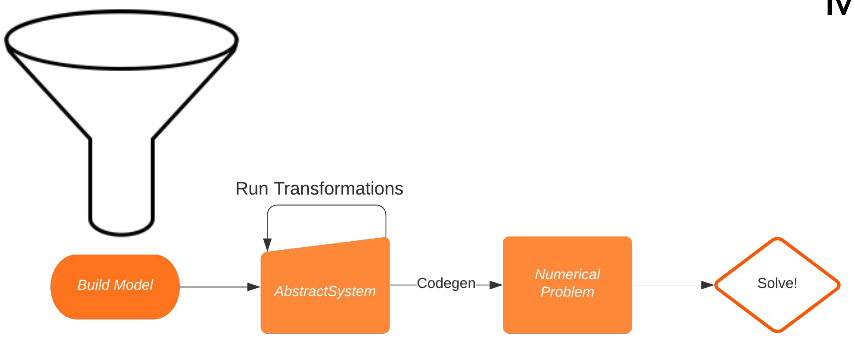
The SciML Common Interface for Julia Equation Solvers

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

Models Can Come From DSLs

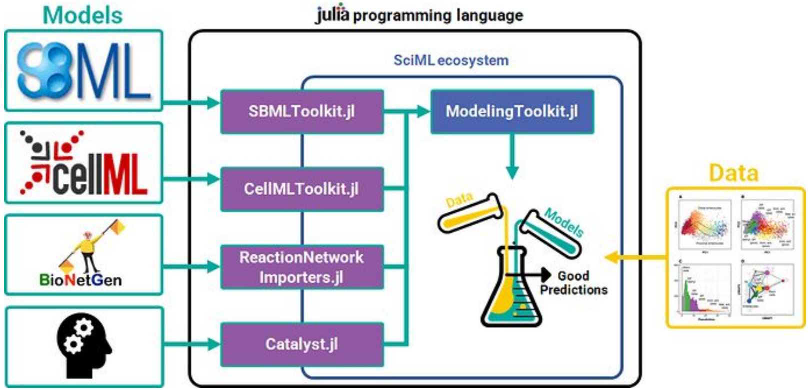
- Pumas.jl
- Catalyst.jl
- OrbitalTrajectories.jl
- AstrodynamicalModels.jl
- BlockSystems.jl
- Conductor.jl
- PowerSystemsDynamics.jl

⋮

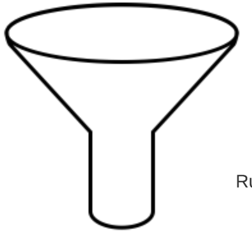


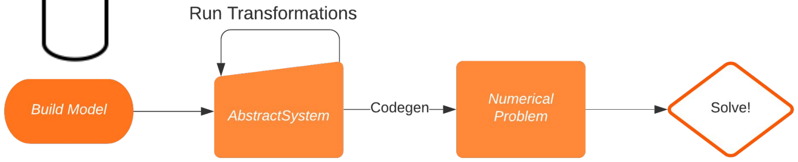
A growing ecosystem of DSLs all feed into ModelingToolkit systems.

Models Can Come From External File Formats



Systems Biology in ModelingToolkit, JuliaCon 2021



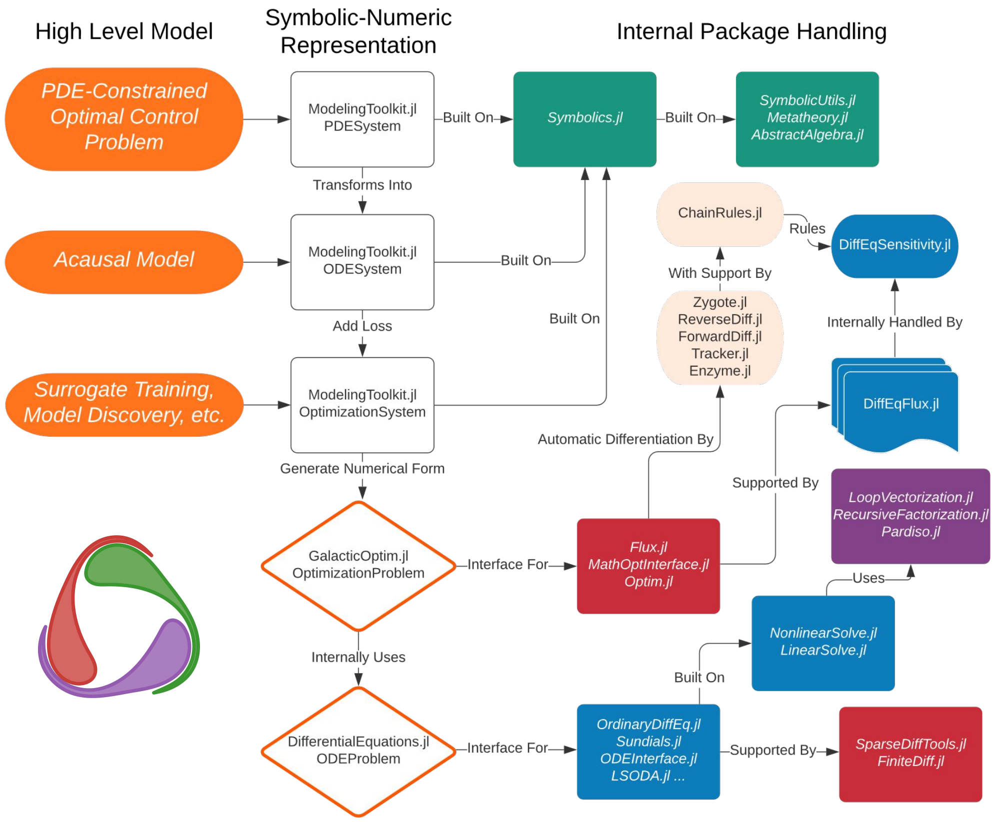


SciML's Modeling Ecosystem: ModelingToolkit's Numerical Counterpart

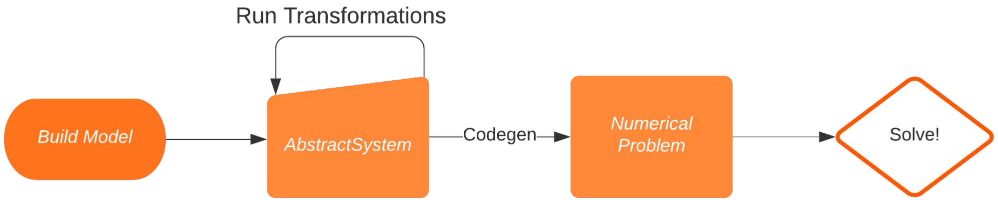
SciML's Common Interface:

- Consistent interface for all numerics
- Symbolic modeling for all forms
- Automated inverse problems and adjoints
- Composes across the whole package ecosystem
- Generic and composable programming
- Uses and embraces the work of other developers

The SciML Common Interface, Oversimplified



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.

```julia
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)
```

PDE

ModelingToolkit’s General PDE Specifications

```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])
```

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = -sin(\pi x) * sin(\pi y)$$

with the boundary conditions:

$$\begin{aligned} u(0,y) &= 0 \\ u(1,y) &= -sin(\pi) * sin(\pi y) \\ u(x,0) &= 0, \\ u(x,1) &= -sin(\pi x) * sin(\pi) \end{aligned}$$

on the space domain:

$$x \in [0,1], y \in [0,1]$$

ModelingToolkit's General PDE Specifications

```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])
```

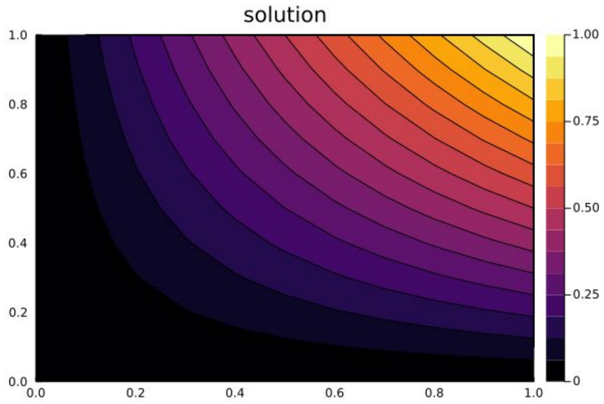
Solving PDEs is generally about transforming mathematical equations into other forms.

See "Solving Partial Differential Equations in Julia", JuliaCon 2018

ModelingToolkit's General PDE Solver: Finite Difference Method

```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")
```

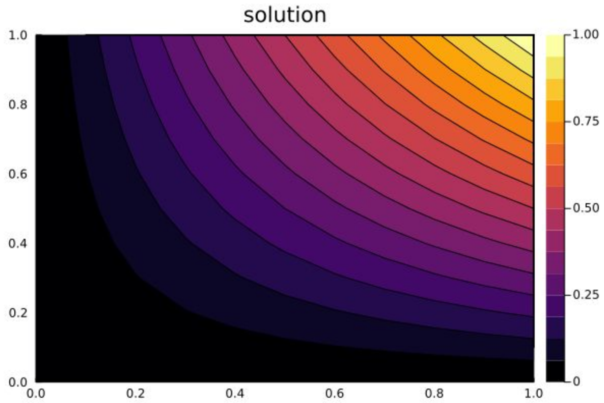


ModelingToolkit's General PDE Solver: Physics-Informed Neural Networks

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

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

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

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

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

Coming soon:

Finite Volume methods
Spectral methods
Finite element methods
…

All PDE solving with unified interface

Looking to collaborate with all Julia PDE developers to make this a reality.

ModelingToolkitStandardLibrary

Modeled after Modelica stdlib

- Blocks
  - PID
  - StateSpace
  - FirstOrder
  - …
- Mechanical
  - Rotational
- Electrical
- Magnetic
- Thermal

MTK: Work in progress-Current status

Solid

- Standard compiler transforms
- Composable modeling in continuous time

Basic functionality

- Symbolic events
- Component library
- Units

Needs more work

- Discrete time
- General events
- Documentation
- Optimization problems
    - Optimal control
- Helpful error messages
- Array variables
- Input-output
- Linearization
- Trimming
- Inverting models
- GUI

Project ideas

- Model validity checker
    - Add metadata to model (https://github.com/SciML/ModelingToolkit.jl/pull/1560)
    - Post hoc solution validation?
    - Online solution validation as callback?
- Composable code callbacks
    - Symbolic callbacks supported
    - How about non-symbolic callbacks?
- Bayesian estimation of model parameters
    - Prior-metadata for parameters
    - Code-gen for likelihood evaluation

Julia computing