ModelingToolkit: A Composable Modeling Language in Julia

Yingbo Ma

About me

Senior year undergraduate mathematics major

Research engineer at Julia Computing working on modeling and numerics

About me

I was interested in differential equations when I was in high school and found Julia by Googling

Went to JuliaCon 2016 https://juliacon.org/2016/

Spent my summer at the Julia Lab and learned numerical linear algebra and ordinary differential equations

Julia

Multiple Dispatch: a higher abstraction on functions. One can dispatch on the specificity of all arguments instead of just the first one.

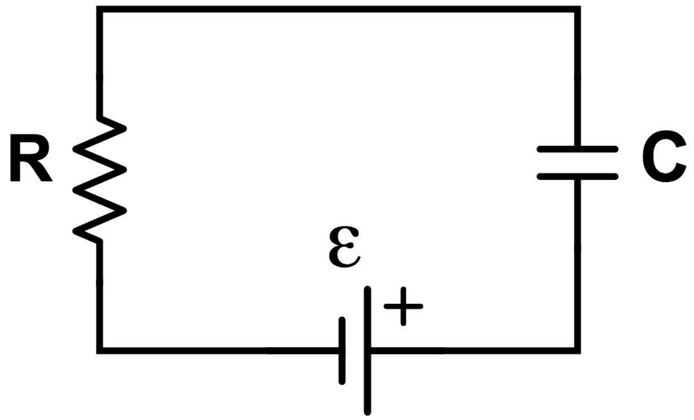
```
foo(a::Number, b::Number) = 1

foo(a::Complex, b::Real) = 2

foo(a::Real, b::Complex) = 3

foo(a::Real, b::Real) = 4


foo(1+2im, 2+2im) == 1

foo(1+2im, 2) == 2

foo(2, 1+im) == 3

foo(1, 1) == 4
```

Definition of ODEs

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

- f is called the derivative function or the right hand side of the ODE
- u is called the dependent variable or the states.
- p is the parameters.
- t is the independent variable.

Limitations of ODEs in Engineering



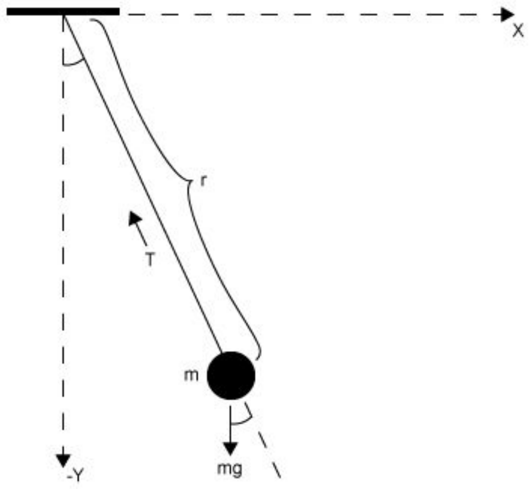
Kirchhoff's current law:

$$\sum_{k=1}^{n} I_k = 0$$

Kirchhoff's voltage law:

$$\sum_{k=1}^{n} V_k = 0$$

Limitations of ODEs in Engineering: No explicit constraints



$$
\begin{aligned}
0 &= T(t)x(t) - x''(t) \\
0 &= T(t)y(t) - g - y''(t) \\
0 &= x(t)^2 + y(t)^2 - r^2,
\end{aligned}
$$

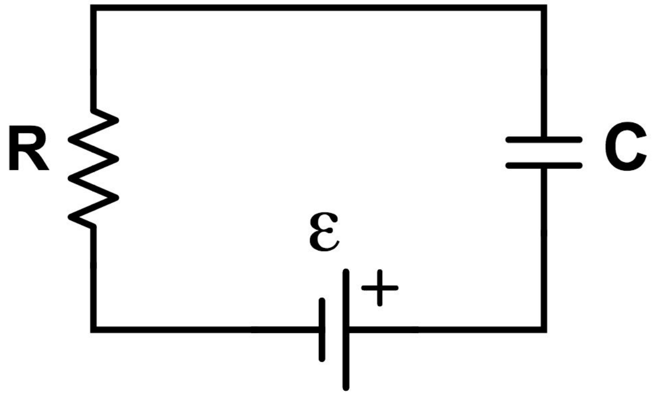
ODEs with explicit constraints: differential-algebraic equation (DAEs)

$$0 = F(u'(t), u(t), p, t)$$

E.g.

$$\begin{aligned} 0 &= T(t)x(t) - x''(t) \\ 0 &= T(t)y(t) - g - y''(t) \\ 0 &= x(t)^2 + y(t)^2 - r^2, \end{aligned}$$

Composable modeling and DAEs



$$\frac{dv}{dt} = \frac{V - v}{CR}$$

Model each physical component individually and connect them together instead of simplify the system into an ODE by hand.

Composable modeling

Simple circuits components are characterized by v and i.

```julia
@parameters t
@connector function Pin(;name)
    sts = @variables v(t)=1.0 i(t)=1.0 [connect = Flow]
    ODESystem(Equation[], t, sts, []; name=name)
end

function Ground(;name)
    @named g = Pin()
    eqs = [g.v ~ 0]
    compose(ODESystem(eqs, t, [], []; name=name), g)
end
```

Object oriented programming

To simplify the modeling process more, one can abstract away the intrinsic physical constraints of a class of physical components, and extend more specific equations later.

Composable modeling

Object-oriented design:

We can define a base model, which contains fundamental physical constraints. This is analogous to boundary conditions.

```
function OnePort(;name)
    @named p = Pin()
    @named n = Pin()
    sts = @variables v(t)=1.0 i(t)=1.0
    eqs = [
           v ~ p.v - n.v
           0 ~ p.i + n.i
           i ~ p.i
          ]
    compose(ODESystem(eqs, t, sts, [];
            name=name), p, n)
end
```

Composable modeling

Object-oriented design:

Simple electronic components are extended model from the base model. This is like “interior” conditions.

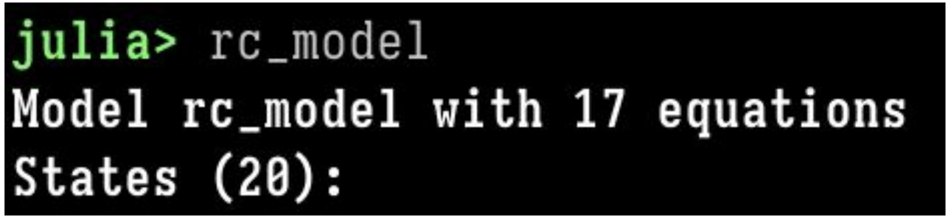
```
function Resistor(;name, R = 1.0)
    @named oneport = OnePort()
    @unpack v, i = oneport
    ps = @parameters R=R
    eqs = [
           v ~ i * R
          ]
    extend(ODESystem(eqs, t, [], ps; name=name), oneport)
end
function Capacitor(;name, C = 1.0)
    @named oneport = OnePort()
    @unpack v, i = oneport
    ps = @parameters C=C
    D = Differential(t)
    eqs = [
           D(v) ~ i / C
          ]
    extend(ODESystem(eqs, t, [], ps; name=name), oneport)
end
```

RC circuit

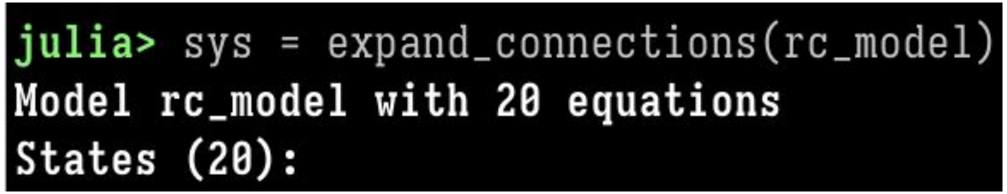
```
R = 1.0; C = 1.0; V = 1.0
@named resistor = Resistor(R=R)
@named capacitor = Capacitor(C=C)
@named source = ConstantVoltage(V=V)
@named ground = Ground()

rc_eqs = [connect(source.p, resistor.p)
          connect(resistor.n, capacitor.p)
          connect(capacitor.n, source.n)
          connect(capacitor.n, ground.g)]
@named rc_model = ODESystem(rc_eqs, t)
rc_model = compose(rc_model, [resistor, capacitor, source,
ground])
```

RC circuit

```
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)
          connect(capacitor.n, ground.g)]
@named rc_model = ODESystem(rc_eqs, t)
rc_model = compose(rc_model, [resistor, capacitor, source,
ground])
```



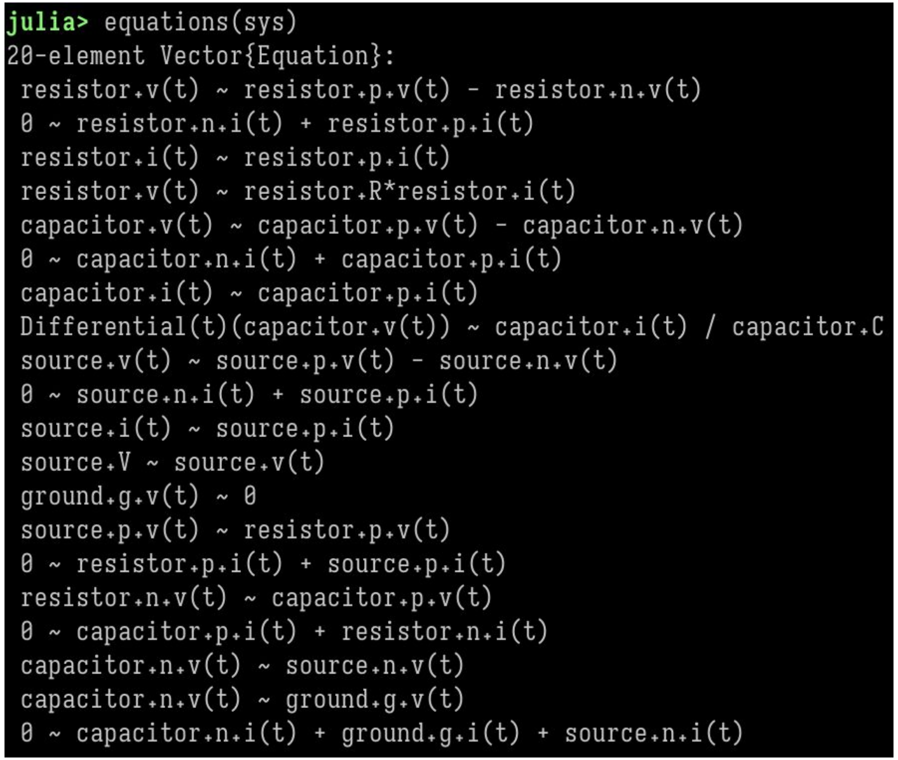
```
julia> rc_model
Model rc_model with 17 equations
States (20):
```



```
julia> sys = expand_connections(rc_model)
Model rc_model with 20 equations
States (20):
```

RC circuit

Due to the constraints, the simulation of this system is numerically expensive.



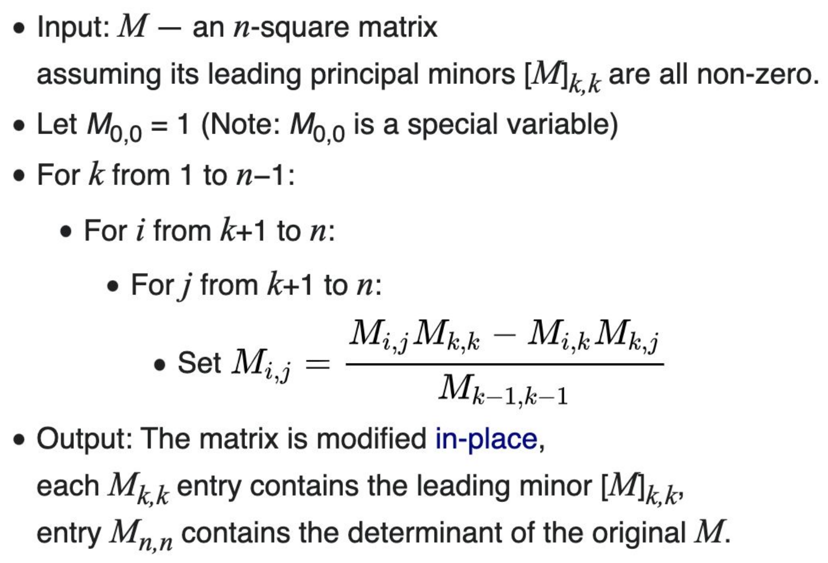
```
julia> equations(sys)
20-element Vector{Equation}:
 resistor.v(t) ~ resistor.p.v(t) - resistor.n.v(t)
 0 ~ resistor.n.i(t) + resistor.p.i(t)
 resistor.i(t) ~ resistor.p.i(t)
 resistor.v(t) ~ resistor.R*resistor.i(t)
 capacitor.v(t) ~ capacitor.p.v(t) - capacitor.n.v(t)
 0 ~ capacitor.n.i(t) + capacitor.p.i(t)
 capacitor.i(t) ~ capacitor.p.i(t)
 Differential(t)(capacitor.v(t)) ~ capacitor.i(t) / capacitor.C
 source.v(t) ~ source.p.v(t) - source.n.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
 source.p.v(t) ~ resistor.p.v(t)
 0 ~ resistor.p.i(t) + source.p.i(t)
 resistor.n.v(t) ~ capacitor.p.v(t)
 0 ~ capacitor.p.i(t) + resistor.n.i(t)
 capacitor.n.v(t) ~ source.n.v(t)
 capacitor.n.v(t) ~ ground.g.v(t)
 0 ~ capacitor.n.i(t) + ground.g.i(t) + source.n.i(t)
```

Structural simplification of DAEs

First observation: most of the algebraic equations in practice are linear with only integer coefficients.

We run an algorithm on the linear subsystem with integer coefficients that is fast and avoids truncation errors.

Bareiss algorithm

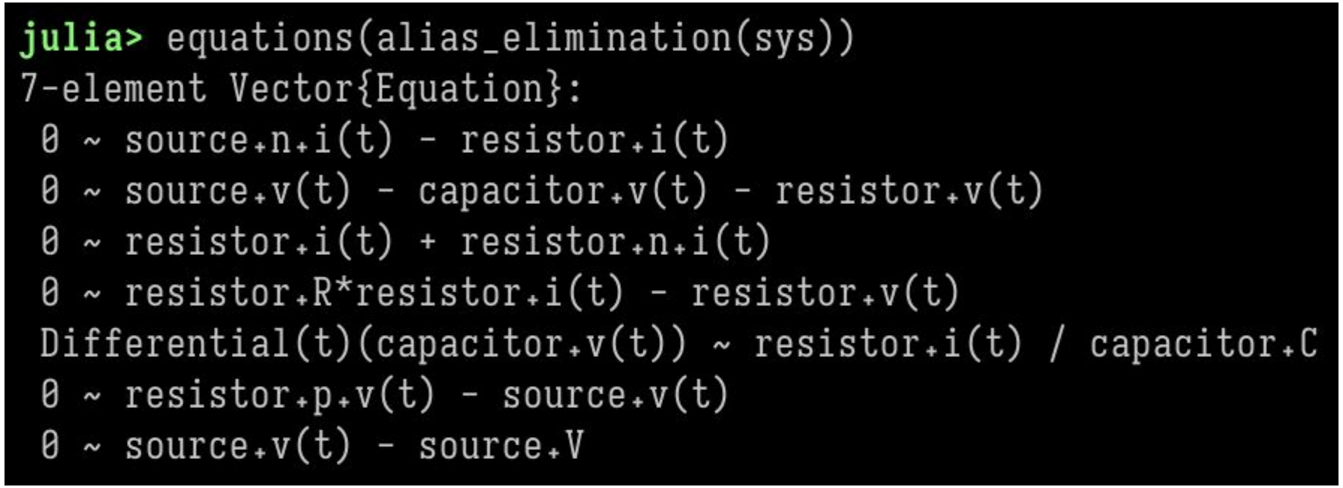


```
• Input: $M$ — an $n$-square matrix
  assuming its leading principal minors $[M]_{k,k}$ are all non-zero.
• Let $M_{0,0}$ = 1 (Note: $M_{0,0}$ is a special variable)
• For $k$ from 1 to $n$−1:
    • For $i$ from $k$+1 to $n$:
        • For $j$ from $k$+1 to $n$:
            • Set $M_{i,j} = \frac{M_{i,j}M_{k,k} - M_{i,k}M_{k,j}}{M_{k-1,k-1}}$
• Output: The matrix is modified in-place,
  each $M_{k,k}$ entry contains the leading minor $[M]_{k,k}$,
  entry $M_{n,n}$ contains the determinant of the original $M$.
```

The Bareiss algorithm exploits the fact that the determinant of integer valued matrices is also integer. Bareiss used Sylvester's determinant identity to prove that this algorithm is fraction-free.

Bareiss, Erwin H. (1968)

Alias elimination



```
julia> equations(alias_elimination(sys))
7-element Vector{Equation}:
 0 ~ source.n.i(t) - resistor.i(t)
 0 ~ source.v(t) - capacitor.v(t) - resistor.v(t)
 0 ~ resistor.i(t) + resistor.n.i(t)
 0 ~ resistor.R*resistor.i(t) - resistor.v(t)
 Differential(t)(capacitor.v(t)) ~ resistor.i(t) / capacitor.C
 0 ~ resistor.p.v(t) - source.v(t)
 0 ~ source.v(t) - source.V
```

Tearing

Consider the system of equations

```
0 = u1 - f1(u5)
0 = u2 - f2(u1)
0 = u3 - f3(u1, u2)
0 = u4 - f4(u2, u3)
0 = u5 - f5(u4, u1)
```

Suppose we know u5 already: we can transform it into

```
u1 = f1(u5)
u2 = f2(u1)
u3 = f3(u1, u2)
u4 = f4(u2, u3)
u5 = f5(u4, u1)
```

Note that we get u5 again

Tearing (cont.)

Suppose we know u5 already: we can transform it into

```
u1 = f1(u5)
u2 = f2(u1)
u3 = f3(u1, u2)
u4 = f4(u2, u3)
u5 = f5(u4, u1)
```

Hence, we can reduce it into a single nonlinear equation by substitution

```
0 = u5 - f5(f4(f2(f1(u5)), f3(f1(u5),
f2(f1(u5)))), f1(u5))
```

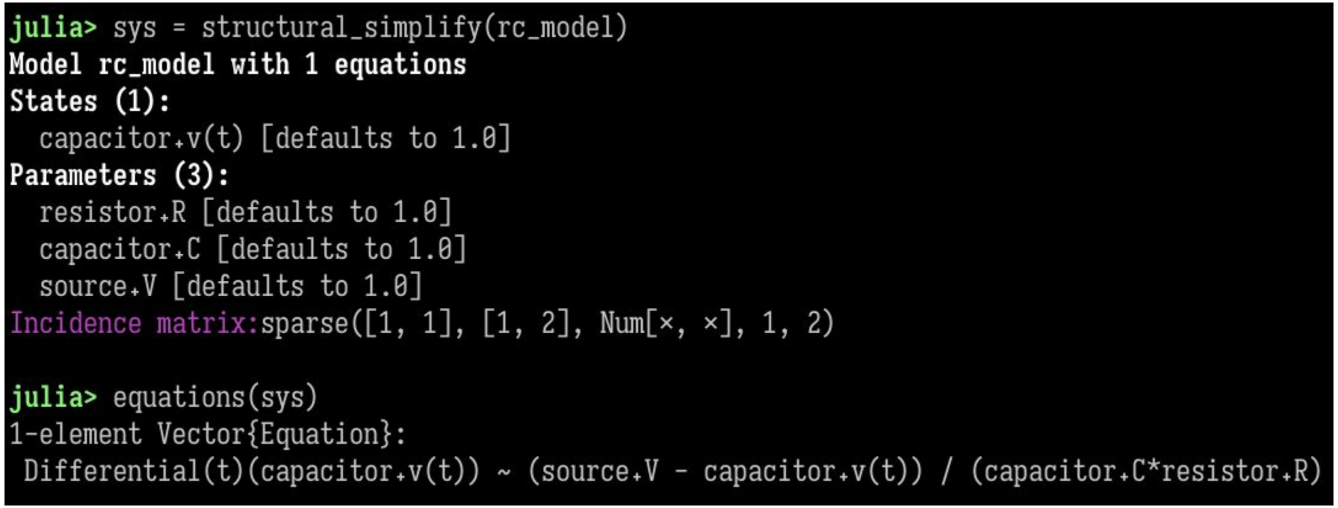
Tearing (cont.)

We can formulate this process in terms of graphs.

1. Run the bipartite graph matching algorithm to assign each equation an unknown.
2. Contract the bipartite graph of equations and unknowns to only a graph of unknowns.
3. Remove all acyclic subgraph.

All Together

$$\frac{dv}{dt} = \frac{V - v}{CR}$$



```
julia> sys = structural_simplify(rc_model)
Model rc_model with 1 equations
States (1):
  capacitor₊v(t) [defaults to 1.0]
Parameters (3):
  resistor₊R [defaults to 1.0]
  capacitor₊C [defaults to 1.0]
  source₊V [defaults to 1.0]
Incidence matrix:sparse([1, 1], [1, 2], Num[×, ×], 1, 2)

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

Acknowledgement

My collaborators: Shashi Gowda (MIT), Chris Rackauckas, Keno Fischer, Viral Shah (Julia Computing), Chris Laughman (MERL)

Modelica community: Hilding Elmquist, Martin Otter

Advisor: Dr. Bedřich Sousedík

References

Julia Language official website: https://julialang.org/

ModelingToolkit.jl code: https://github.com/SciML/ModelingToolkit.jl

Yingbo Ma, Shashi Gowda, Ranjan Anantharaman, Chris Laughman, Viral Shah, and Chris Rackauckas. "Modelingtoolkit: A composable graph transformation system for equation-based modeling." arXiv:2103.05244 (2021).

Martin Otter and Hilding Elmqvist. "Transformation of differential algebraic array equations to index one form." In Proceedings of the 12th International Modelica Conference. Linköping University Electronic Press, 2017.