ModelingToolkit.jl, An IR and Compiler for Scientific Models

Chris Rackauckas

December 29, 2019

A lot of people are building modeling languages for their specific domains. However, while the syntax my vary greatly between these domain-specific languages (DSLs), the internals of modeling frameworks are surprisingly similar: building differential equations, calculating Jacobians, etc.

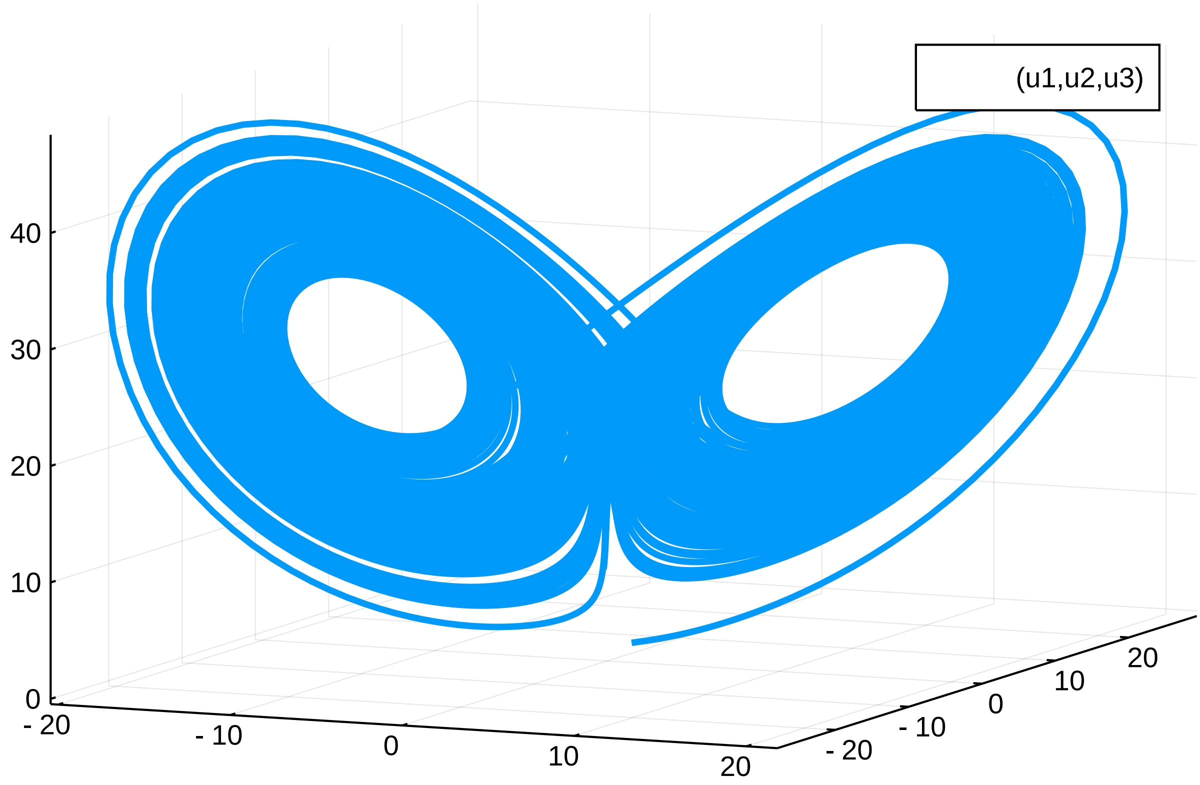
ModelingToolkit.jl is metamodeling systemitized After building our third modeling interface, we realized that this problem can be better approached by having a reusable internal structure which DSLs can target. This internal is ModelingToolkit.jl: an Intermediate Representation (IR) with a well-defined interface for defining system transformations and compiling to Julia functions for use in numerical libraries. Now a DSL can easily be written by simply defining the translation to ModelingToolkit.jl's primatives and querying for the mathematical quantities one needs.

0.0.1 Basic usage: defining differential equation systems, with performance!

Let's explore the IR itself. ModelingToolkit.jl is friendly to use, and can used as a symbolic DSL in its own right. Let's define and solve the Lorenz differential equation system using ModelingToolkit to generate the functions:

```
using ModelingToolkit

### Define a differential equation system

@parameters t σ ρ β
@variables x(t) y(t) z(t)
@derivatives D'~t

eqs = [D(x) ~ σ*(y-x),
       D(y) ~ x*(ρ-z)-y,
       D(z) ~ x*y - β*z]
de = ODESystem(eqs, t, [x,y,z], [σ,ρ,β])
ode_f = ODEFunction(de)

### Use in DifferentialEquations.jl

using OrdinaryDiffEq
u_0 = ones(3)
tspan = (0.0,100.0)
```

```julia
p = [10.0,28.0,10/3]
prob = ODEProblem(ode_f,u_0,tspan,p)
sol = solve(prob,Tsit5())

using Plots
plot(sol,vars=(1,2,3))
```



0.0.2 ModelingToolkit is a compiler for mathematical systems

At its core, ModelingToolkit is a compiler. It's IR is its type system, and its output are Julia functions (it's a compiler for Julia code to Julia code, written in Julia).

DifferentialEquations.jl wants a function `f(u,p,t)` or `f(du,u,p,t)` for defining an ODE system, so ModelingToolkit.jl builds both. First the out of place version:

```julia
generate_function(de)[1]
```

```
:((u, p, t)->begin
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:61 =#
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:62 =#
          if u isa Array
              #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:63 =#
              return #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl
:55 =# @inbounds(begin
                          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\
utils.jl:55 =#
                          let (x, y, z, σ, ρ, β) = (u[1], u[2], u[3], p[1], p[2], p[3])
                              [σ * (y - x), x * (ρ - z) - y, x * y - β * z]
                          end
                      end)
          else
```

```
            #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:65 =#
            X = #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:54
 =# @inbounds(begin
                        #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\
utils.jl:54 =#
                        let (x, y, z, σ, ρ, β) = (u[1], u[2], u[3], p[1], p[2], p[3])
                            (σ * (y - x), x * (ρ - z) - y, x * y - β * z)
                        end
                    end)
        end
        #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:67 =#
        T = promote_type(map(typeof, X)...)
        #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:68 =#
        map(T, X)
        #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:69 =#
        construct = if u isa ModelingToolkit.StaticArrays.StaticArray
                ModelingToolkit.StaticArrays.similar_type(typeof(u), eltype(X))
            else
                x->begin
                        #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\
utils.jl:69 =#
                        convert(typeof(u), x)
                    end
            end
        #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:70 =#
        construct(X)
    end)
```

and the in-place:

```
generate_function(de)[2]
```

```
:((var"##MTIIPVar#1267", u, p, t)->begin
        #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:75 =#
        #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:76 =#
        #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:76 =#
@inbounds begin
                #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:77
 =#
                #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:56
 =# @inbounds begin
                        #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\
utils.jl:56 =#
                        let (x, y, z, σ, ρ, β) = (u[1], u[2], u[3], p[1], p[2], p[3])
                            var"##MTIIPVar#1267"[1] = σ * (y - x)
                            var"##MTIIPVar#1267"[2] = x * (ρ - z) - y
                            var"##MTIIPVar#1267"[3] = x * y - β * z
                        end
                    end
            end
        #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:79 =#
        nothing
    end)
```

ModelingToolkit.jl can be used to calculate the Jacobian of the differential equation system:

```
jac = calculate_jacobian(de)
```

```
3×3 Array{Expression,2}:
  σ * -1           σ   Constant(0)
```

```
 ρ - z(t)  Constant(-1)    x(t) * -1
     y(t)          x(t)          -1β
```

It will automatically generate functions for using this Jacobian within the stiff ODE solvers for faster solving:

```
jac_expr = generate_jacobian(de)
```

```
(:((u, p, t)->begin
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:61 =#
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:62 =#
          if u isa Array
              #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:63 =#
              return #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl
:55 =# @inbounds(begin
                          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\
utils.jl:55 =#
                          let (x, y, z, σ, ρ, β) = (u[1], u[2], u[3], p[1], p[2], p[3])
                              [σ * -1, ρ - z, y, σ, -1, x, 0, x * -1, -1β]
                          end
                      end)
          else
              #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:65 =#
              X = #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:54
 =# @inbounds(begin
                          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\
utils.jl:54 =#
                          let (x, y, z, σ, ρ, β) = (u[1], u[2], u[3], p[1], p[2], p[3])
                              (σ * -1, ρ - z, y, σ, -1, x, 0, x * -1, -1β)
                          end
                      end)
          end
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:67 =#
          T = promote_type(map(typeof, X)...)
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:68 =#
          map(T, X)
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:69 =#
          construct = if u isa ModelingToolkit.StaticArrays.StaticArray
                  ModelingToolkit.StaticArrays.similar_type(typeof(u), eltype(X))
              else
                  x->begin
                          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\
utils.jl:69 =#
                          convert(typeof(u), x)
                      end
              end
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:70 =#
          construct(X)
      end), :((var"##MTIIPVar#1269", u, p, t)->begin
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:75 =#
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:76 =#
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:76 =#
@inbounds begin
                  #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:77
 =#
                  #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:56
 =# @inbounds begin
                          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\
utils.jl:56 =#
                          let (x, y, z, σ, ρ, β) = (u[1], u[2], u[3], p[1], p[2], p[3])
```

```
                                var"##MTIIPVar#1269"[1] = σ * -1
                                var"##MTIIPVar#1269"[2] = ρ - z
                                var"##MTIIPVar#1269"[3] = y
                                var"##MTIIPVar#1269"[4] = σ
                                var"##MTIIPVar#1269"[5] = -1
                                var"##MTIIPVar#1269"[6] = x
                                var"##MTIIPVar#1269"[7] = 0
                                var"##MTIIPVar#1269"[8] = x * -1
                                var"##MTIIPVar#1269"[9] = -1β
                            end
                        end
                end
            #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:79 =#
            nothing
        end))
```

It can even do fancy linear algebra. Stiff ODE solvers need to perform an LU-factorization which is their most expensive part. But ModelingToolkit.jl can skip this operation and instead generate the analytical solution to a matrix factorization, and build a Julia function for directly computing the factorization, which is then optimized in LLVM compiler passes.

```
ModelingToolkit.generate_factorized_W(de)[1]
```

```
(:((u, p, gam, t)->begin
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:61 =#
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:62 =#
          if u isa Array
              #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:63 =#
              return #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl
:55 =# @inbounds(begin
                              #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\
utils.jl:55 =#
                              let (x, y, z, σ, ρ, β) = (u[1], u[2], u[3], p[1], p[2], p[3])
                                  [σ * -1 * gam + -1, gam * (ρ - z) * inv(σ * -1 * gam + -1),
 gam * y * inv(σ * -1 * gam + -1), gam *
σ, (gam * -1 + -1) - gam * (ρ - z) * inv(σ * -1 * gam + -1) * gam * σ, (gam * x - gam * y
 * inv(σ * -1 * gam + -1) * gam * σ) * in
v((gam * -1 + -1) - gam * (ρ - z) * inv(σ * -1 * gam + -1) * gam * σ), 0, x * -1 * gam -
0, ((-1 * β * gam + -1) - 0) - (gam * x -
 gam * y * inv(σ * -1 * gam + -1) * gam * σ) * inv((gam * -1 + -1) - gam * (ρ - z) * inv(
σ * -1 * gam + -1) * gam * σ) * (x * -1 *
 gam - 0)]
                              end
                          end)
          else
              #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:65 =#
              X = #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:54
 =# @inbounds(begin
                              #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\
utils.jl:54 =#
                              let (x, y, z, σ, ρ, β) = (u[1], u[2], u[3], p[1], p[2], p[3])
                                  (σ * -1 * gam + -1, gam * (ρ - z) * inv(σ * -1 * gam + -1),
 gam * y * inv(σ * -1 * gam + -1), gam *
σ, (gam * -1 + -1) - gam * (ρ - z) * inv(σ * -1 * gam + -1) * gam * σ, (gam * x - gam * y
 * inv(σ * -1 * gam + -1) * gam * σ) * in
v((gam * -1 + -1) - gam * (ρ - z) * inv(σ * -1 * gam + -1) * gam * σ), 0, x * -1 * gam -
0, ((-1 * β * gam + -1) - 0) - (gam * x -
 gam * y * inv(σ * -1 * gam + -1) * gam * σ) * inv((gam * -1 + -1) - gam * (ρ - z) * inv(
σ * -1 * gam + -1) * gam * σ) * (x * -1 *
```

```
 gam - 0))
                         end
                     end)
         end
         #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:67 =#
         T = promote_type(map(typeof, X)...)
         #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:68 =#
         map(T, X)
         #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:69 =#
         construct = (x->begin
                     #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\systems
\diffeqs\diffeqsystem.jl:202 =#
                     #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\systems
\diffeqs\diffeqsystem.jl:203 =#
                     A = SMatrix{(3, 3)...}(x)
                     #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\systems
\diffeqs\diffeqsystem.jl:204 =#
                     StaticArrays.LU(LowerTriangular(SMatrix{(3, 3)...}(
UnitLowerTriangular(A))), UpperTriangular(A), SVector(ntu
ple((n->begin
                                          #= C:\Users\accou\.julia\packages\
ModelingToolkit\xi418\src\systems\diffeqs\diffeqsystem
.jl:204 =#
                                          n
                                      end), max((3, 3)...))))
                 end)
         #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:70 =#
         construct(X)
     end), :((var"##MTIIPVar#1271", u, p, gam, t)->begin
         #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:75 =#
         #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:76 =#
         #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:76 =#
@inbounds begin
                 #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:77
 =#
                 #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:56
 =# @inbounds begin
                         #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\
utils.jl:56 =#
                         let (x, y, z, σ, ρ, β) = (u[1], u[2], u[3], p[1], p[2], p[3])
                             var"##MTIIPVar#1271"[1] = σ * -1 * gam + -1
                             var"##MTIIPVar#1271"[2] = gam * (ρ - z) * inv(σ * -1 * gam
+ -1)
                             var"##MTIIPVar#1271"[3] = gam * y * inv(σ * -1 * gam + -1)
                             var"##MTIIPVar#1271"[4] = gam * σ
                             var"##MTIIPVar#1271"[5] = (gam * -1 + -1) - gam * (ρ - z) *
 inv(σ * -1 * gam + -1) * gam * σ
                             var"##MTIIPVar#1271"[6] = (gam * x - gam * y * inv(σ * -1 *
 gam + -1) * gam * σ) * inv((gam * -1 + -
1) - gam * (ρ - z) * inv(σ * -1 * gam + -1) * gam * σ)
                             var"##MTIIPVar#1271"[7] = 0
                             var"##MTIIPVar#1271"[8] = x * -1 * gam - 0
                             var"##MTIIPVar#1271"[9] = ((-1 * β * gam + -1) - 0) - (gam
* x - gam * y * inv(σ * -1 * gam + -1) *
gam * σ) * inv((gam * -1 + -1) - gam * (ρ - z) * inv(σ * -1 * gam + -1) * gam * σ) * (x *
 -1 * gam - 0)
                         end
                     end
             end
```

```
        #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:79 =#
        nothing
    end))
```

0.0.3 Solving Nonlinear systems

ModelingToolkit.jl is not just for differential equations. It can be used for any mathematical target that is representable by its IR. For example, let's solve a rootfinding problem `F(x)=0`. What we do is define a nonlinear system and generate a function for use in NLsolve.jl

```
@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])
nlsys_func = generate_function(ns)
```

```
(:((u, p)->begin
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:61 =#
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:62 =#
          if u isa Array
              #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:63 =#
              return #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl
:55 =# @inbounds(begin
                          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\
utils.jl:55 =#
                          let (x, y, z, σ, ρ, β) = (u[1], u[2], u[3], p[1], p[2], p[3])
                              [(*)(σ, (-)(y, x)), (-)((*)(x, (-)(ρ, z)), y), (-)((*)(x, y
), (*)(β, z))]
                          end
                      end)
          else
              #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:65 =#
              X = #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:54
 =# @inbounds(begin
                          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\
utils.jl:54 =#
                          let (x, y, z, σ, ρ, β) = (u[1], u[2], u[3], p[1], p[2], p[3])
                              ((*)(σ, (-)(y, x)), (-)((*)(x, (-)(ρ, z)), y), (-)((*)(x, y
), (*)(β, z)))
                          end
                      end)
          end
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:67 =#
          T = promote_type(map(typeof, X)...)
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:68 =#
          map(T, X)
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:69 =#
          construct = if u isa ModelingToolkit.StaticArrays.StaticArray
                  ModelingToolkit.StaticArrays.similar_type(typeof(u), eltype(X))
              else
                  x->begin
                          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\
utils.jl:69 =#
                          convert(typeof(u), x)
```

```
                    end
                end
            #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:70 =#
            construct(X)
        end), :((var"##MTIIPVar#1275", u, p)->begin
            #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:75 =#
            #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:76 =#
            #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:76 =#
@inbounds begin
                  #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:77
 =#
                  #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:56
 =# @inbounds begin
                          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\
utils.jl:56 =#
                          let (x, y, z, σ, ρ, β) = (u[1], u[2], u[3], p[1], p[2], p[3])
                              var"##MTIIPVar#1275"[1] = (*)(σ, (-)(y, x))
                              var"##MTIIPVar#1275"[2] = (-)((*)(x, (-)(ρ, z)), y)
                              var"##MTIIPVar#1275"[3] = (-)((*)(x, y), (*)(β, z))
                          end
                      end
              end
          #= C:\Users\accou\.julia\packages\ModelingToolkit\xi418\src\utils.jl:79 =#
          nothing
      end))
```

We can then tell ModelingToolkit.jl to compile this function for use in NLsolve.jl, and then numerically solve the rootfinding problem:

```
nl_f = @eval eval(nlsys_func[2])
# Make a closure over the parameters for for NLsolve.jl
f2 = (du,u) -> nl_f(du,u,(10.0,26.0,2.33))

using NLsolve
nlsolve(f2,ones(3))

Results of Nonlinear Solver Algorithm
 * Algorithm: Trust-region with dogleg and autoscaling
 * Starting Point: [1.0, 1.0, 1.0]
 * Zero: [2.2228042242798023e-10, 2.222804224296743e-10, -9.990339458476605e-11]
 * Inf-norm of residuals: 0.000000
 * Iterations: 3
 * Convergence: true
   * |x - x'| < 0.0e+00: false
   * |f(x)| < 1.0e-08: true
 * Function Calls (f): 4
 * Jacobian Calls (df/dx): 4
```

0.0.4 Library of transformations on mathematical systems

The reason for using ModelingToolkit is not just for defining performant Julia functions for solving systems, but also for performing mathematical transformations which may be required in order to numerically solve the system. For example, let's solve a third order ODE. The way this is done is by transforming the third order ODE into a first order ODE, and then solving the resulting ODE. This transformation is given by the `ode_order_lowering` function.

```
@derivatives D3'''~t
@derivatives D2''~t
@variables u(t), x(t)
eqs = [D3(u) ~ 2(D2(u)) + D(u) + D(x) + 1
       D2(x) ~ D(x) + 2]
de = ODESystem(eqs, t, [u,x], [])
de1 = ode_order_lowering(de)
```

```
ODESystem(ModelingToolkit.DiffEq[ModelingToolkit.DiffEq(u_tt, 1, ((2 * u_tt(t) + u_t(t))
+ x_t(t)) + 1), ModelingToolkit.DiffEq(x_
t, 1, x_t(t) + 2), ModelingToolkit.DiffEq(u_t, 1, u_tt(t)), ModelingToolkit.DiffEq(u, 1,
u_t(t)), ModelingToolkit.DiffEq(x, 1, x_t
(t))], t, Variable[u, x, u_tt, u_t, x_t], Variable[], Base.RefValue{Array{Expression,2}}(
Array{Expression}(undef,0,0)), Base.RefVa
lue{Array{Expression,2}}(Array{Expression}(undef,0,0)), Base.RefValue{Array{Expression
,2}}(Array{Expression}(undef,0,0)))
```

```
de1.eqs
```

```
5-element Array{ModelingToolkit.DiffEq,1}:
 ModelingToolkit.DiffEq(u_tt, 1, ((2 * u_tt(t) + u_t(t)) + x_t(t)) + 1)
 ModelingToolkit.DiffEq(x_t, 1, x_t(t) + 2)
 ModelingToolkit.DiffEq(u_t, 1, u_tt(t))
 ModelingToolkit.DiffEq(u, 1, u_t(t))
 ModelingToolkit.DiffEq(x, 1, x_t(t))
```

This has generated a system of 5 first order ODE systems which can now be used in the ODE solvers.

0.0.5 Linear Algebra... for free?

Let's take a look at how to extend ModelingToolkit.jl in new directions. Let's define a Jacobian just by using the derivative primatives by hand:

```
@parameters t σ ρ β
@variables x(t) y(t) z(t)
@derivatives D'~t Dx'~x Dy'~y Dz'~z
eqs = [D(x) ~ σ*(y-x),
       D(y) ~ x*(ρ-z)-y,
       D(z) ~ x*y - β*z]
J = [Dx(eqs[1].rhs) Dy(eqs[1].rhs) Dz(eqs[1].rhs)
 Dx(eqs[2].rhs) Dy(eqs[2].rhs) Dz(eqs[2].rhs)
 Dx(eqs[3].rhs) Dy(eqs[3].rhs) Dz(eqs[3].rhs)]
```

```
3×3 Array{Operation,2}:
        derivative(σ * (y(t) - x(t)), x(t))  ...         derivative(σ * (y(t) - x(t)), z(
t))
 derivative(x(t) * (ρ - z(t)) - y(t), x(t))     derivative(x(t) * (ρ - z(t)) - y(t), z(t)
)
   derivative(x(t) * y(t) - β * z(t), x(t))       derivative(x(t) * y(t) - β * z(t), z(t)
)
```

Notice that this writes the derivatives in a "lazy" manner. If we want to actually compute the derivatives, we can expand out those expressions:

```
J = expand_derivatives.(J)
```

```
3×3 Array{Expression,2}:
   σ * -1              σ  Constant(0)
 ρ - z(t)  Constant(-1)    x(t) * -1
     y(t)          x(t)          -1β
```

Here's the magic of ModelingToolkit.jl: **Julia treats ModelingToolkit expressions like a Number, and so generic numerical functions are directly usable on ModelingToolkit expressions!** Let's compute the LU-factorization of this Jacobian we defined using Julia's Base linear algebra library.

```
using LinearAlgebra
luJ = lu(J,Val(false))
```

```
LU{Expression,Array{Expression,2}}
L factor:
3×3 Array{Expression,2}:
                  Constant(1)  ...  Constant(0)
 (ρ - z(t)) * inv(σ * -1)     identity(0)
       y(t) * inv(σ * -1)     Constant(1)
U factor:
3×3 Array{Expression,2}:
      σ * -1  ...
                                        Constant(0)
 identity(0)
     x(t) * -1 - ((ρ - z(t)) * inv(σ * -1)) * 0
 identity(0)     (-1β - (y(t) * inv(σ * -1)) * 0) - ((x(t) - (y(t) * inv(σ * -1)) * σ) *
 inv(-1 - ((ρ - z(t)) * inv(σ * -1)) * σ))
 * (x(t) * -1 - ((ρ - z(t)) * inv(σ * -1)) * 0)
```

```
luJ.L
```

```
3×3 Array{Expression,2}:
                  Constant(1)  ...  Constant(0)
 (ρ - z(t)) * inv(σ * -1)     identity(0)
       y(t) * inv(σ * -1)     Constant(1)
```

and the inverse?

```
invJ = inv(luJ)
```

```
3×3 Array{Expression,2}:
 (σ * -1) \ ((true - 0 * (((-1β - (y(t) * inv(σ * -1)) * 0) - ((x(t) - (y(t) * inv(σ *
-1)) * σ) * inv(-1 - ((ρ - z(t)) * inv(σ *
-1)) * σ)) * (x(t) * -1 - ((ρ - z(t)) * inv(σ * -1)) * 0)) \ ((0 - (y(t) * inv(σ * -1)) *
 true) - ((x(t) - (y(t) * inv(σ * -1)) *
σ) * inv(-1 - ((ρ - z(t)) * inv(σ * -1)) * σ)) * (0 - ((ρ - z(t)) * inv(σ * -1)) * true))
)) - σ * ((-1 - ((ρ - z(t)) * inv(σ * -1)
) * σ) \ ((0 - ((ρ - z(t)) * inv(σ * -1)) * true) - (x(t) * -1 - ((ρ - z(t)) * inv(σ *
-1)) * 0) * (((-1β - (y(t) * inv(σ * -1)) *
 0) - ((x(t) - (y(t) * inv(σ * -1)) * σ) * inv(-1 - ((ρ - z(t)) * inv(σ * -1)) * σ)) * (
x(t) * -1 - ((ρ - z(t)) * inv(σ * -1)) * 0
)) \ ((0 - (y(t) * inv(σ * -1)) * true) - ((x(t) - (y(t) * inv(σ * -1)) * σ) * inv(-1 -
((ρ - z(t)) * inv(σ * -1)) * σ)) * (0 - ((
ρ - z(t)) * inv(σ * -1)) * true)))))  ...  (σ * -1) \ ((0 - 0 * (((-1β - (y(t) * inv(σ *
 -1)) * 0) - ((x(t) - (y(t) * inv(σ * -1))
* σ) * inv(-1 - ((ρ - z(t)) * inv(σ * -1)) * σ)) * (x(t) * -1 - ((ρ - z(t)) * inv(σ * -1)
) * 0)) \ ((true - (y(t) * inv(σ * -1)) *
 0) - ((x(t) - (y(t) * inv(σ * -1)) * σ) * inv(-1 - ((ρ - z(t)) * inv(σ * -1)) * σ)) *
(0 - ((ρ - z(t)) * inv(σ * -1)) * 0)))) - σ
```

```
 * ((-1 - ((ρ - z(t)) * inv(σ * -1)) * σ) \ ((0 - ((ρ - z(t)) * inv(σ * -1)) * 0) - (x(t)
 * -1 - ((ρ - z(t)) * inv(σ * -1)) * 0) *
 (((-1β - (y(t) * inv(σ * -1)) * 0) - ((x(t) - (y(t) * inv(σ * -1)) * σ) * inv(-1 - ((ρ -
 z(t)) * inv(σ * -1)) * σ)) * (x(t) * -1
- ((ρ - z(t)) * inv(σ * -1)) * 0)) \ ((true - (y(t) * inv(σ * -1)) * 0) - ((x(t) - (y(t)
* inv(σ * -1)) * σ) * inv(-1 - ((ρ - z(t)
) * inv(σ * -1)) * σ)) * (0 - ((ρ - z(t)) * inv(σ * -1)) * 0))))))
```

```
            (-1 - ((ρ - z(t)) * inv(σ * -
1)) * σ) \ ((0 - ((ρ - z(t)) * inv(σ * -1)) * true) - (x(t) * -1 - ((ρ - z(t)) * inv(σ *
 -1)) * 0) * (((-1β - (y(t) * inv(σ * -1))
 * 0) - ((x(t) - (y(t) * inv(σ * -1)) * σ) * inv(-1 - ((ρ - z(t)) * inv(σ * -1)) * σ)) *
 (x(t) * -1 - ((ρ - z(t)) * inv(σ * -1)) *
 0)) \ ((0 - (y(t) * inv(σ * -1)) * true) - ((x(t) - (y(t) * inv(σ * -1)) * σ) * inv(-1 -
 ((ρ - z(t)) * inv(σ * -1)) * σ)) * (0 -
((ρ - z(t)) * inv(σ * -1)) * true))))
```

```
      (-1 - ((ρ - z(t)) * inv(σ * -1)) * σ) \ ((0 - ((ρ - z(t)) * inv(σ * -1)) * 0) - (x
(t) * -1 - ((ρ - z(t)) * inv(σ * -1)) * 0)
 * (((-1β - (y(t) * inv(σ * -1)) * 0) - ((x(t) - (y(t) * inv(σ * -1)) * σ) * inv(-1 - ((
ρ - z(t)) * inv(σ * -1)) * σ)) * (x(t) * -
1 - ((ρ - z(t)) * inv(σ * -1)) * 0)) \ ((true - (y(t) * inv(σ * -1)) * 0) - ((x(t) - (y(t
) * inv(σ * -1)) * σ) * inv(-1 - ((ρ - z(
t)) * inv(σ * -1)) * σ)) * (0 - ((ρ - z(t)) * inv(σ * -1)) * 0))))
```

```
               ((-1β - (y(t) * inv(σ * -1
)) * 0) - ((x(t) - (y(t) * inv(σ * -1)) * σ) * inv(-1 - ((ρ - z(t)) * inv(σ * -1)) * σ))
 * (x(t) * -1 - ((ρ - z(t)) * inv(σ * -1))
 * 0)) \ ((0 - (y(t) * inv(σ * -1)) * true) - ((x(t) - (y(t) * inv(σ * -1)) * σ) * inv(-1
 - ((ρ - z(t)) * inv(σ * -1)) * σ)) * (0
- ((ρ - z(t)) * inv(σ * -1)) * true))
```

```
      ((-1β - (y(t) * inv(σ * -1)) * 0) - ((x(t) - (y(t) * inv(σ * -1)) * σ) * inv(-1 -
((ρ - z(t)) * inv(σ * -1)) * σ)) * (x(t) *
 -1 - ((ρ - z(t)) * inv(σ * -1)) * 0)) \ ((true - (y(t) * inv(σ * -1)) * 0) - ((x(t) - (y
(t) * inv(σ * -1)) * σ) * inv(-1 - ((ρ -
z(t)) * inv(σ * -1)) * σ)) * (0 - ((ρ - z(t)) * inv(σ * -1)) * 0))
```

Thus ModelingToolkit.jl can utilize existing numerical code on symbolic codes
Let's follow this thread a little deeper.

0.0.6 Automatically convert numerical codes to symbolic

Let's take someone's code written to numerically solve the Lorenz equation:

```
function lorenz(du,u,p,t)
 du[1] = p[1]*(u[2]-u[1])
 du[2] = u[1]*(p[2]-u[3]) - u[2]
 du[3] = u[1]*u[2] - p[3]*u[3]
```

```
end
```

```
lorenz (generic function with 1 method)
```

Since ModelingToolkit can trace generic numerical functions in Julia, let's trace it with Operations. When we do this, it'll spit out a symbolic representation of their numerical code:

```
u = [x,y,z]
du = similar(u)
p = [σ,ρ,β]
lorenz(du,u,p,t)
du
```

```
3-element Array{Operation,1}:
        σ * (y(t) - x(t))
 x(t) * (ρ - z(t)) - y(t)
   x(t) * y(t) - β * z(t)
```

We can then perform symbolic manipulations on their numerical code, and build a new numerical code that optimizes/fixes their original function!

```
J = [Dx(du[1]) Dy(du[1]) Dz(du[1])
     Dx(du[2]) Dy(du[2]) Dz(du[2])
     Dx(du[3]) Dy(du[3]) Dz(du[3])]
J = expand_derivatives.(J)
```

```
3×3 Array{Expression,2}:
   σ * -1              σ  Constant(0)
 ρ - z(t)  Constant(-1)    x(t) * -1
     y(t)          x(t)          -1β
```

0.0.7 Automated Sparsity Detection

In many cases one has to speed up large modeling frameworks by taking into account sparsity. While ModelingToolkit.jl can be used to compute Jacobians, we can write a standard Julia function in order to get a spase matrix of expressions which automatically detects and utilizes the sparsity of their function.

```
using SparseArrays
function SparseArrays.SparseMatrixCSC(M::Matrix{T}) where {T<:ModelingToolkit.Expression}
    idxs = findall(!iszero, M)
    I = [i[1] for i in idxs]
    J = [i[2] for i in idxs]
    V = [M[i] for i in idxs]
    return SparseArrays.sparse(I, J, V, size(M)...)
end
sJ = SparseMatrixCSC(J)
```

```
3×3 SparseMatrixCSC{Expression,Int64} with 8 stored entries:
  [1, 1]  =  σ * -1
  [2, 1]  =  ρ - z(t)
  [3, 1]  =  y(t)
  [1, 2]  =  σ
  [2, 2]  =  Constant(-1)
  [3, 2]  =  x(t)
  [2, 3]  =  x(t) * -1
  [3, 3]  =  -1β
```

0.0.8 Dependent Variables, Functions, Chain Rule

"Variables" are overloaded. When you are solving a differential equation, the variable `u(t)` is actually a function of time. In order to handle these kinds of variables in a mathematically correct and extensible manner, the ModelingToolkit IR actually treats variables as functions, and constant variables are simply 0-ary functions (`t()`).

We can utilize this idea to have parameters that are also functions. For example, we can have a parameter σ which acts as a function of 1 argument, and then utilize this function within our differential equations:

```
@parameters σ(..)
eqs = [D(x) ~ σ(t-1)*(y-x),
       D(y) ~ x*(σ(t^2)-z)-y,
       D(z) ~ x*y - β*z]
```

```
3-element Array{Equation,1}:
 Equation(derivative(x(t), t), σ(t - 1) * (y(t) - x(t)))
 Equation(derivative(y(t), t), x(t) * (σ(t ^ 2) - z(t)) - y(t))
 Equation(derivative(z(t), t), x(t) * y(t) - β * z(t))
```

Notice that when we calculate the derivative with respect to `t`, the chain rule is automatically handled:

```
@derivatives D_t'~t
D_t(x*(σ(t^2)-z)-y)
expand_derivatives(D_t(x*(σ(t^2)-z)-y))
```

```
(σ(t ^ 2) - z(t)) * derivative(x(t), t) + x(t) * (derivative(σ(t ^ 2), t) + -1 *
derivative(z(t), t)) + -1 * derivative(y(t), t)
```

0.0.9 Hackability: Extend directly from the language

ModelingToolkit.jl is written in Julia, and thus it can be directly extended from Julia itself. Let's define a normal Julia function and call it with a variable:

```
_f(x) = 2x + x^2
_f(x)
```

```
2 * x(t) + x(t) ^ 2
```

Recall that when we do that, it will automatically trace this function and then build a symbolic expression. But what if we wanted our function to be a primative in the symbolic framework? This can be done by registering the function.

```
f(x) = 2x + x^2
@register f(x)
```

```
f (generic function with 2 methods)
```

Now this function is a new primitive:

```
f(x)
```

```
f(x(t))
```

and we can now define derivatives of our function:

```julia
function ModelingToolkit.derivative(::typeof(f), args::NTuple{1,Any}, ::Val{1})
    2 + 2args[1]
end
expand_derivatives(Dx(f(x)))
```

```
2 + 2 * x(t)
```

0.1 Appendix

This tutorial is part of the DiffEqTutorials.jl repository, found at: https://github.com/JuliaDiffEq/DiffEq

To locally run this tutorial, do the following commands:

```
using DiffEqTutorials
DiffEqTutorials.weave_file("ode_extras","01-ModelingToolkit.jmd")
```

Computer Information:

```
Julia Version 1.3.0
Commit 46ce4d7933 (2019-11-26 06:09 UTC)
Platform Info:
  OS: Windows (x86_64-w64-mingw32)
  CPU: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.1 (ORCJIT, skylake)
Environment:
  JULIA_EDITOR = "C:\Users\accou\AppData\Local\atom\app-1.42.0\atom.exe"  -a
  JULIA_NUM_THREADS = 4
```

Package Information:

```
Status `~\.julia\dev\DiffEqTutorials\Project.toml`
```