Equation and Object-oriented Modeling

Modeling Course – Automatic Control

Hilding Elmqvist

Mogram AB and Modelon AB

In collaboration with: Martin Otter, Gerhard Hippman, Andrea Neumayr, Oskar Åström

Assistants: Karl Johan Åström and Oskar Åström

Content

- Introduction
- Part 1: Equation Oriented Modeling (Modia)
  - structural and symbolic algorithms
  - DAE index reduction
  - examples/exercises: Yoyo, Drum model
- Part 2: Object-oriented Modeling
  - physical considerations
  - structuring mechanisms – principles of Modia and Modelica
  - multibody models and contact problems (Modia3D and Modiator)
  - examples/exercises: Rattleback, TippeTop
  - performance - benchmark models

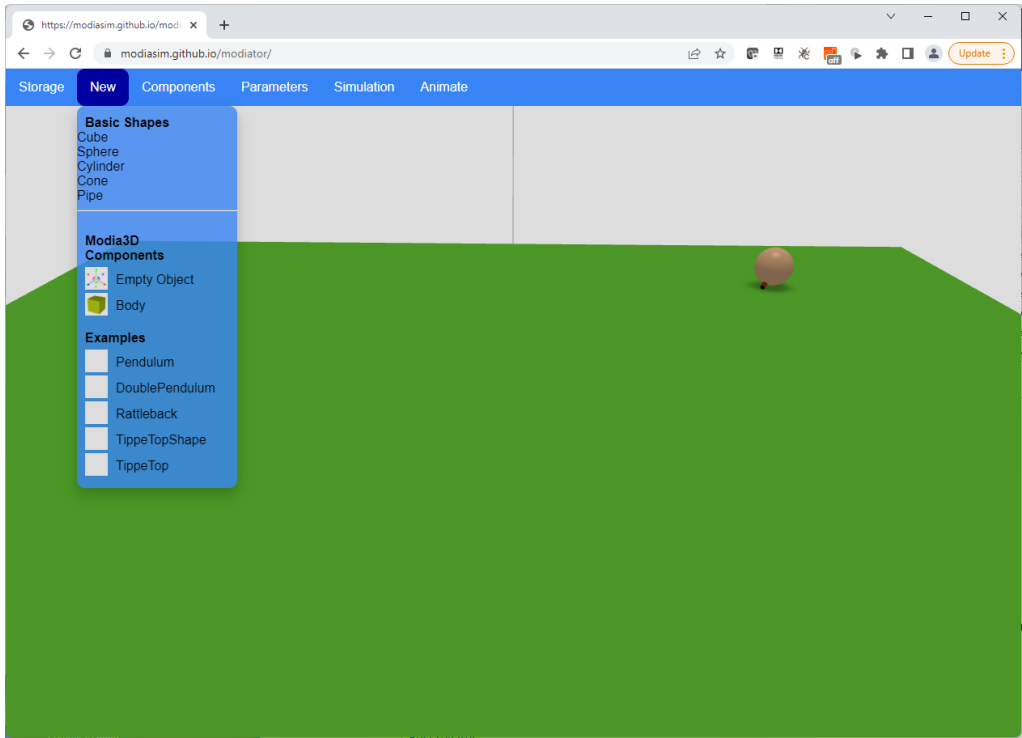
Introduction

- Interesting phenomena:
  - Contact and collision
  - Demo: Yoyo, Rattleback, TippeTop

- Preview of Modiator subset:
  - Go to: **tinyurl.com/modiator**
  - Readme: tinyurl.com/modiator/readme
  - Select New/Examples/…
  - In order to simulate:
    - Download and install Julia version 1.7.2 (https://julialang.org/downloads/)
    - Start Julia

      > using Modia3D

      > @usingModiaPlot
      > ModiatorServer()



Part 1 – Equation Oriented Modeling

Modia - Equation oriented modeling

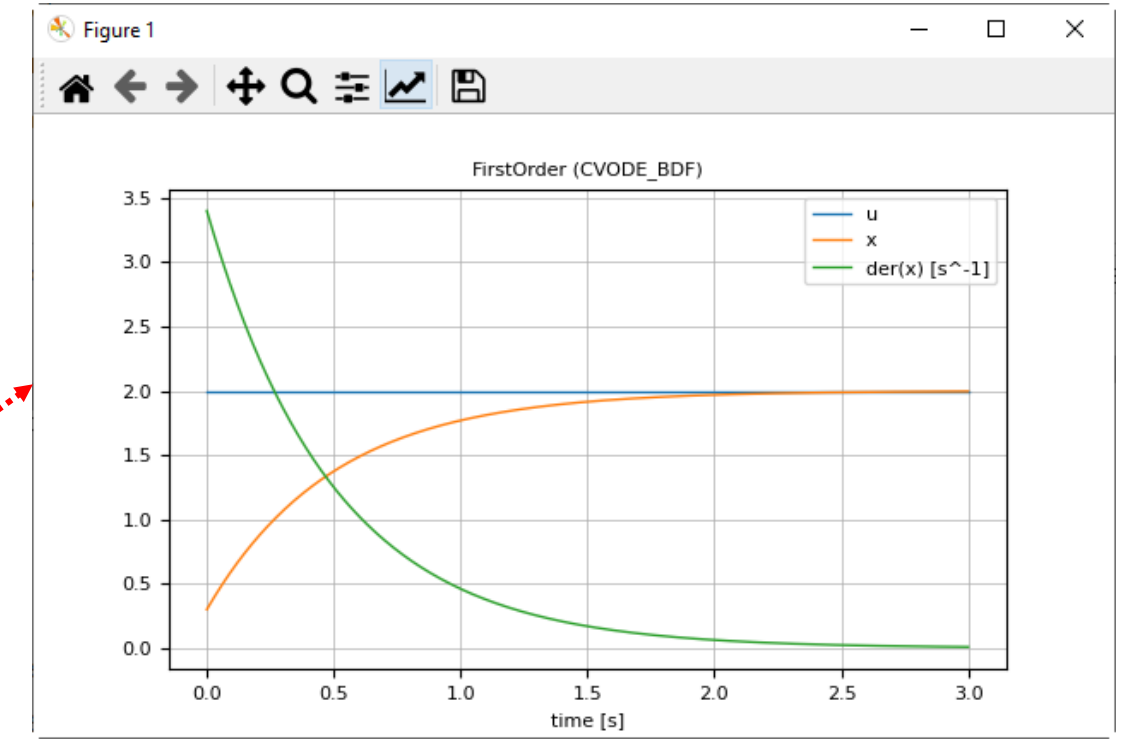
$$T \cdot \frac{dx}{dt} + x = 2$$

```
using Modia
@usingModiaPlot

FirstOrder = Model(
   T = 0.5u"s",
   x = Var(init=0.3),
   equations = :[
      u = 2
      T * der(x) + x = u])

firstOrder = @instantiateModel(FirstOrder)
simulate!(firstOrder, stopTime=3.0)
plot(firstOrder, ("u", "x", "der(x)"))
```

using PyPlot or Makie

$$\frac{dx}{dt} = (2 - x)/T$$

using DifferentialEquations

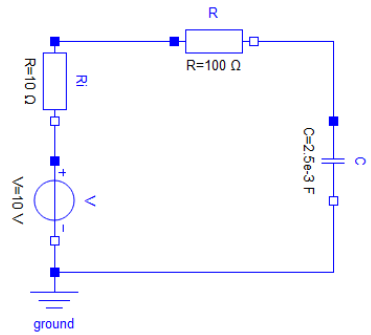


Structural and Symbolic Processing

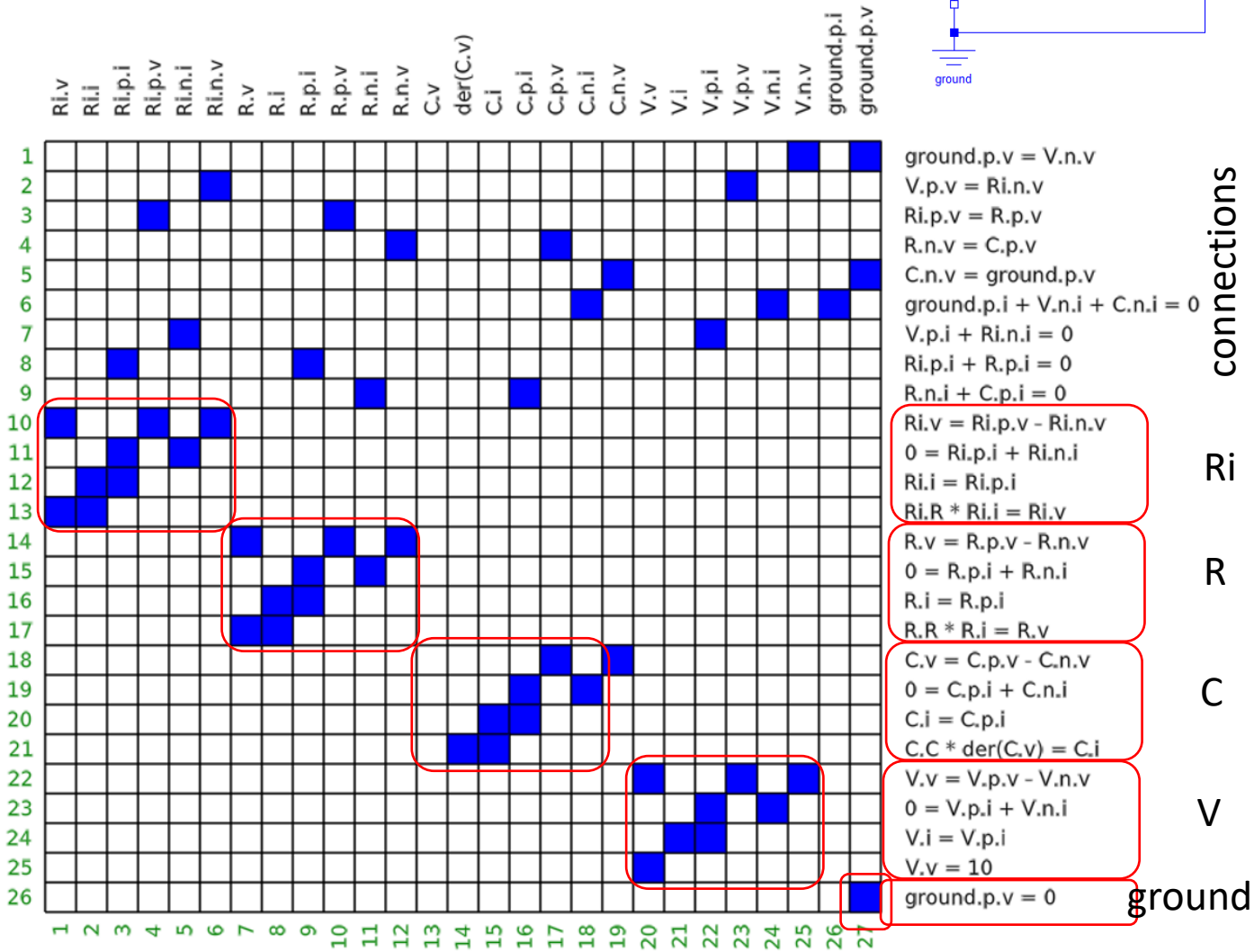
- Build incidence matrix
- Perform alias elimination
- Assign a variable to each equation
- Sort and group equations (BLT)
- Reduce block size by tearing
- Perform Index reduction
- Symbolically solve for the unknowns
- Symbolically differentiate certain equations
- Generate executable code

Incidence Matrix

In a first step, the structural information of a DAE is provided as **incidence matrix**:



- Every **column** corresponds to one **variable**.
- Every **row** corresponds to one **equation**.
- A **cell** is **marked** (here in "blue"), if a **variable** is present in one **equation**.



Representation in Julia in ModiaBase:

```
G = [ [25, 27],
      [6 ,23],
        ...
      [27] ]
```

Every row of vector G consists of a vector of Integers
(example: equation 1 is a function of variables 25 and 27)

Note, **G** is

- a sparse representation of an **incidence matrix** and
- a representation of a **bi-partite graph**.

Alias Elimination

In a first step, Modelica tools usually perform the following simplifications on a DAE:

- Equations of the form **v = 0** are removed and „v“ is replaced by „0“ at all places where „v“ occurs, and these equations are simplified.
- Equations of the form **v1 = v2, v1 = -v2, 0 = v1+v2** etc. are removed, „v1“ is replaced by „v2“ (or „-v2“) at all places where „v1“ occurs (so called alias-variables), and these equations are simplified.

For example, the following **equations** are **removed**

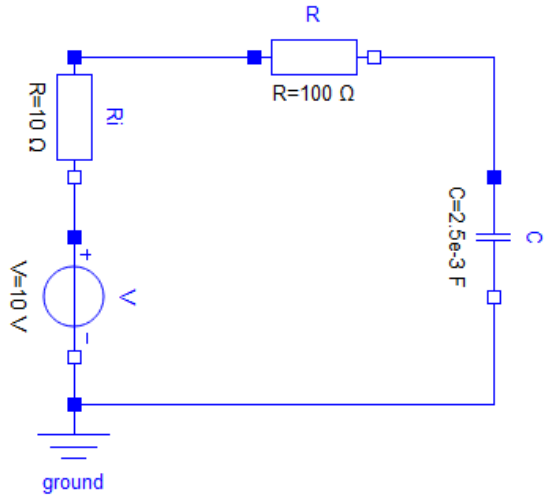
```
ground.p.v = 0
V.n.v      = ground.p.v
C.n.v      = ground.p.v
C.v        = C.p.v - C.n.v
R.n.v      = C.p.v
```

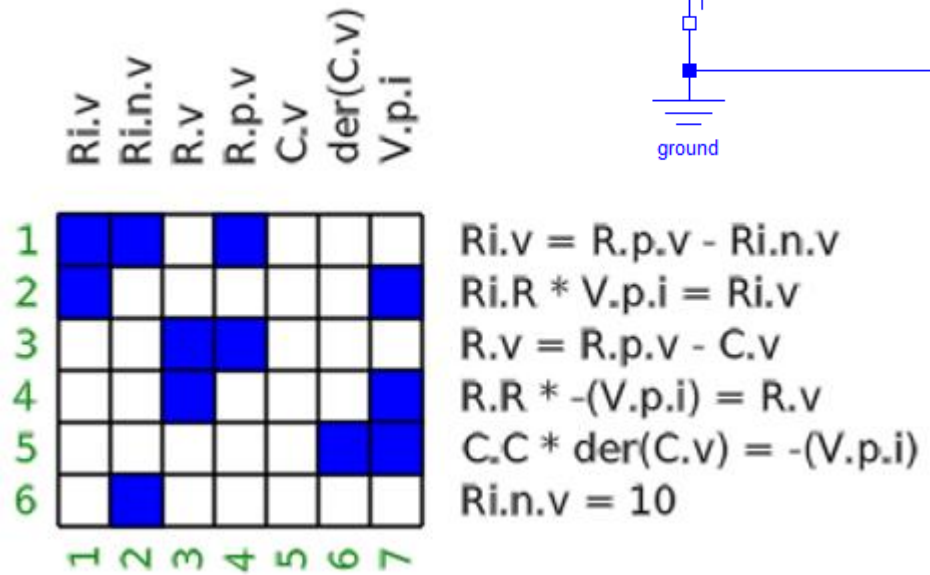
and all **variables**, with exception of C.v, are **removed** (C.v is kept and not C.p.v, because der(C.v) appears). The following information is kept internally:

```
ground.p.v := 0
V.n.v      := 0
C.n.v      := 0
C.p.v      := C.v
R.n.v      := C.v
```

`expr1 = expr2` means right hand side expression is equal to left hand side expression.
`v := expr` means to evaluate right hand side expression and assign result to left hand side variable.

With these simplifications, the incidence matrix of the low pass filter is changed to:





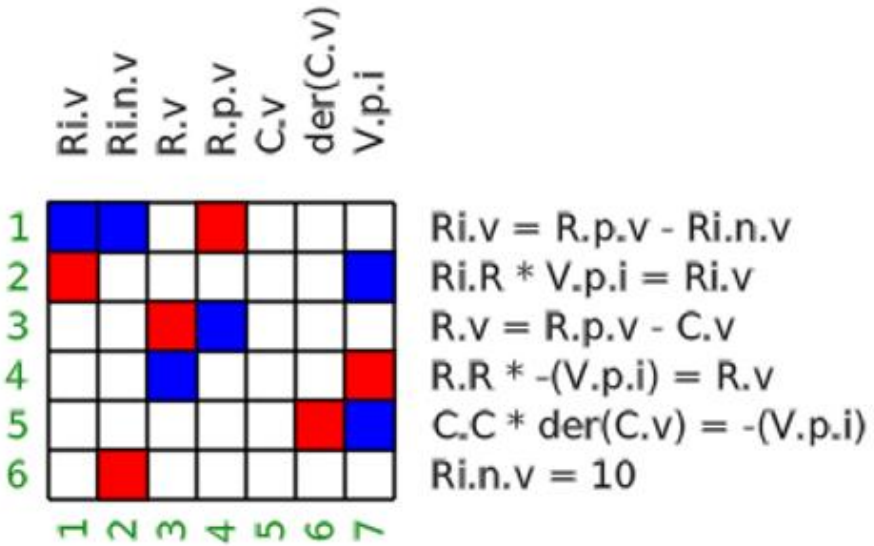
Ri.v = R.p.v - Ri.n.v
Ri.R * V.p.i = Ri.v
R.v = R.p.v - C.v
R.R * -(V.p.i) = R.v
C.C * der(C.v) = -(V.p.i)
Ri.n.v = 10

```
G = [ [1,2,4],
      [1,7],
      [3,4],
      [3,7],
      [6,7],
      [2] ]
```

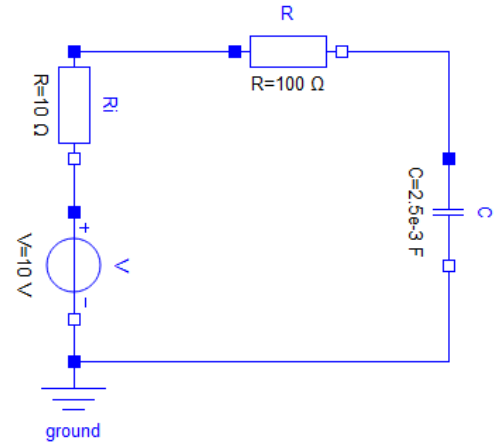
Eliminated variables:

```
R.i          := -(V.p.i)
ground.p.v   := 0
R.p.i        := -(V.p.i)
R.n.v        := C.v
V.n.i        := -(V.p.i)
V.n.v        := 0
V.p.v        := Ri.n.v
Ri.p.i       := V.p.i
C.n.v        := 0
C.p.v        := C.v
Ri.p.v       := R.p.v
C.n.i        := V.p.i
V.i          := V.p.i
R.n.i        := V.p.i
C.p.i        := -(V.p.i)
ground.p.i   := 0
C.i          := -(V.p.i)
Ri.i         := V.p.i
V.v          := Ri.n.v
Ri.n.i       := -(V.p.i)
```

Assignment/matching



- „**Red**“ marks show the assigned variables.
- „**Blue**“ marks show if a variable is part of the respective equation



Julia code:

```
# G is defined on slide 10
vActive    = fill(true,7)
vActive[5] = false # state C.v is known
assign = matching(G, 7, vActive)
```

resulting in:

```
assign = [2,6,3,1,0,5,4]
```

(meaning:
variable 1 is solved from equation 2,
variable 2 is solved from equation 6,
...)

A very compact and efficient **assignment algorithm** is described in:

Pantelides C.C: *The consistent initialization of differential-algebraic systems*.
SIAM Journal of Scientific and Statistical Computing, No. 9, S. 213-231, 1988.

Open source Julia implementation as function **matching**(...) in ModiaBase/src/BLTandPantelides.jl:

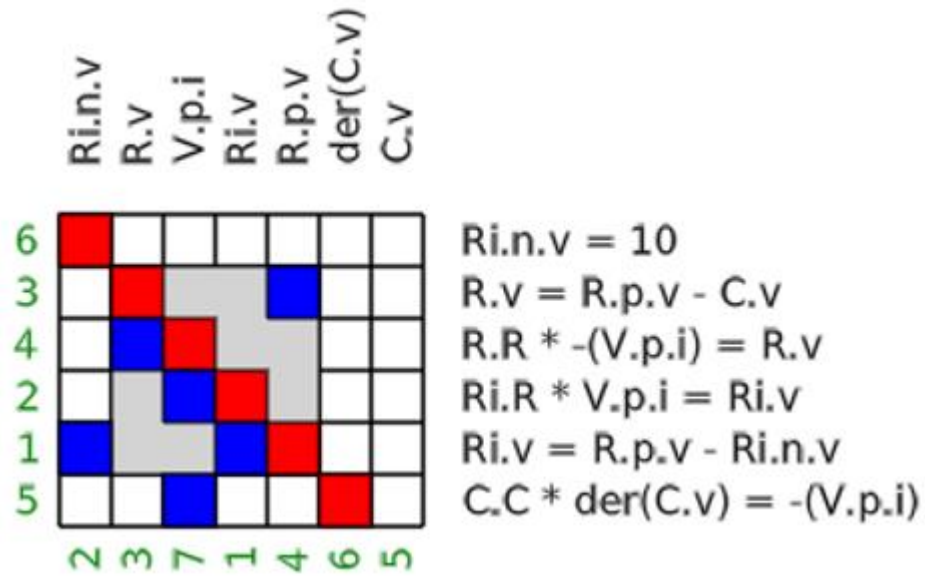
```
"""
    function matching(G, M, vActive=fill(true, M))
Find maximum matching in bipartite graph

* `G`: bipartite graph
* `M`: number of V-nodes
* `vActive`: set to false has the same effect as deleting V-node and corresponding edges
* `return assign`: assign[j] contains the E-node to which V-node j is assigned or 0 if V-node j not assigned

Reference:
Pantelides, C.: The consistent initialization of differential-algebraic systems. SIAM Journal
of Scientific and Statistical Computing, 9(2), pp. 213-231 (1988).
"""
function matching(G, M, vActive=fill(true, M))
    assign::Array{Int,1} = fill(0, M)
    eColour::Array{Bool,1} = fill(false, length(G))
    vColour::Array{Bool,1} = fill(false, M)
    vPassive::Array{Int,1} = [if va; 0 else 1 end for va in vActive]
    for i in 1:length(G)
        fill!(eColour, false)
        fill!(vColour, false)
        pathFound = augmentPath!(G, i, assign, vColour, eColour, vPassive)
    end
    return assign
end
```

function augmentPath!(...) is shown on the next slide

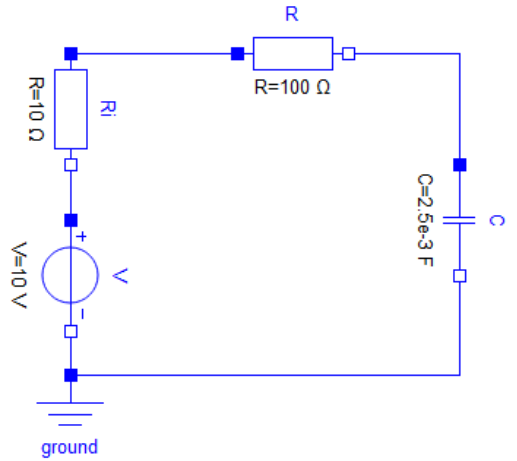
```julia
function augmentPath!(G, i, assign, vColour, eColour, vPassive)
    # returns pathFound
    # assign: assign[j] contains the E-node to which V-node j is assigned or 0 if V-node j not assigned
    # i: E-node
    # vPassive: set to != 0 has the same effect as deleting V-node and corresponding edges
    # j: V-node

    if log
        println("augmentPath: equation $i")
    end

    pathFound = false
    eColour[i] = true

    # If a V-node j exists such that edge (i-j) exists and assign[j] == 0
    for j in G[i]
        if vPassive[j] == 0 && assign[j] == 0
            pathFound = true
            assign[j] = i
            return pathFound
        end
    end

    # For every j such that edge (i-j) exists and j is uncoloured
    for j in G[i]
        if vPassive[j] == 0 && !vColour[j]
            vColour[j] = true
            k = assign[j]
            pathFound = augmentPath!(G, k, assign, vColour, eColour, vPassive)

            if pathFound
                assign[j] = i
                return pathFound
            end
        end
    end
    return pathFound
end
```

Convention in Julia:
function augmentPath!(...)
The „!“ indicates that elements of input arrays are changed,
e.g. here: assign, vColour, eColour

BLT transformation



- „**Red**“ marks show the assigned variables.
  „**Blue**“ marks show if a variable is part of the respective equation
- „Grey“ area marks an algebraic loop.



Julia code:

```
# G is defined on slide 10
# assign is defined on slide 20
blt = BLT(G, assign)
```

resulting in:

```
blt = [ [6],
        [3,4,2,1],
        [5] ]
```

(meaning:
The second BLT block consists of equations 3,4,2,1 and forms an algebraic loop)

A very compact and efficient algorithm is from:

Tarjan R.E.: *Depth First Search and Linear Graph Algorithms*.
SIAM Journal of Comp., Vol. 1, S. 146-160, 1972.

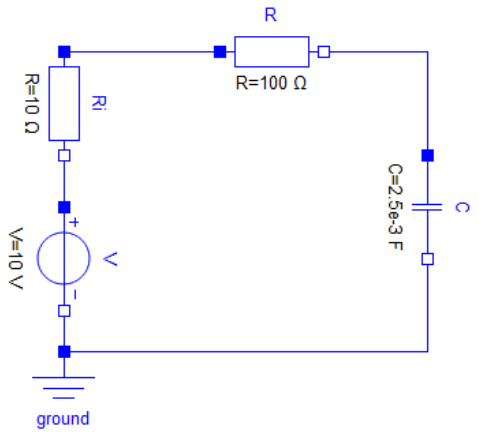
Open source Julia implementation as function **BLT**(..) in ModiaBase/src/BLTandPantelides.jl

```
"""
    function BLT(G, assign)

Find Block Lower Triangular structure for a bipartite graph `G` with assignment `assign`

* `G`: bipartite graph
* `assign`: assign[j] contains the E-node to which V-node j is assigned or 0 if V-node j not assigned
* `return components`: cell array of components. Each component is a list of indices to E-nodes
"""
function BLT(G, assign)
    nextnode::Int = 0
    stack = []
    components = []
    lowlink = fill(0, length(G))
    number = fill(0, length(G))

    for v in 1:length(G)
        if number[v] == 0
            nextnode = strongConnect!(G, assign, v, nextnode, stack, components, lowlink, number)
        end
    end
    return components
end
```

Function strongConnect!(...) is shown on the next slide.

```julia
function strongConnect!(G, assign, v, nextnode, stack, components, lowlink, number)
    # println("strongConnect: ", v)

    if v == 0
        return nextnode
    end

    nextnode += 1
    lowlink[v] = number[v] = nextnode
    push!(stack, v)

    for w in [assign[j] for j in G[v]] # for w in the adjacency list of v
        if w > 0   # Is assigned
            if number[w] == 0 # if not yet numbered
                nextnode = strongConnect!(G, assign, w, nextnode, stack, components, lowlink, number)
                lowlink[v] = min(lowlink[v], lowlink[w])
            else
                if number[w] < number[v]
                    # (v, w) is a frond or cross-link
                    # if w is on the stack of points. Always valid since otherwise number[w]=notOnStack (a big number)
                    lowlink[v] = min(lowlink[v], number[w])
                end
            end
        end
    end

    if lowlink[v] == number[v]
        # v is the root of a component
        # start a new strongly connected component
        comp = []
        repeat = true
        while repeat
            # delete w from point stack and put w in the current component
            # println("delete w from point stack and put w in the current component")
            w = pop!(stack)
            number[w] = notOnStack
            push!(comp, w)
            repeat = w != v
        end
        push!(components, comp)
    end
    return nextnode
end
```

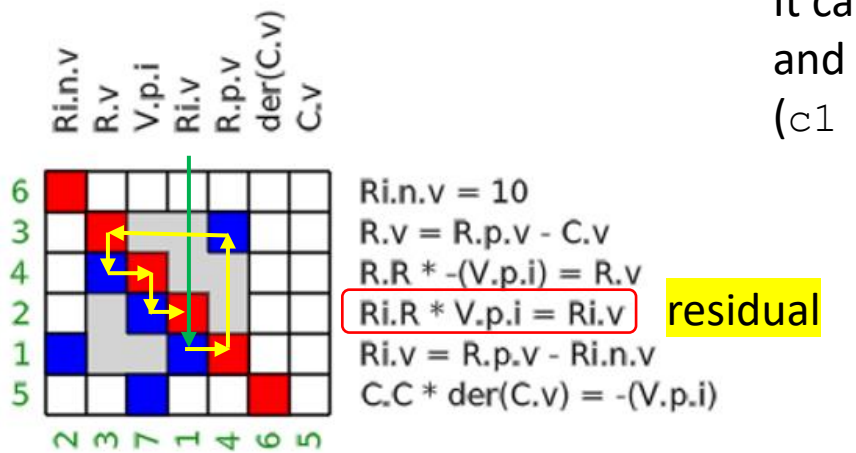
Tearing



Reduces the dimension of BLT block 2 from size 4 to size 1:

```
iteration variable: Ri.v

R.p.v    := Ri.n.v - Ri.v    # Ri.n.v already computed
R.v      := R.p.v - C.v      # C.v is a state (known)
V.p.i    := - R.v/R.R
residual:= Ri.R*V.p.i - Ri.v
```

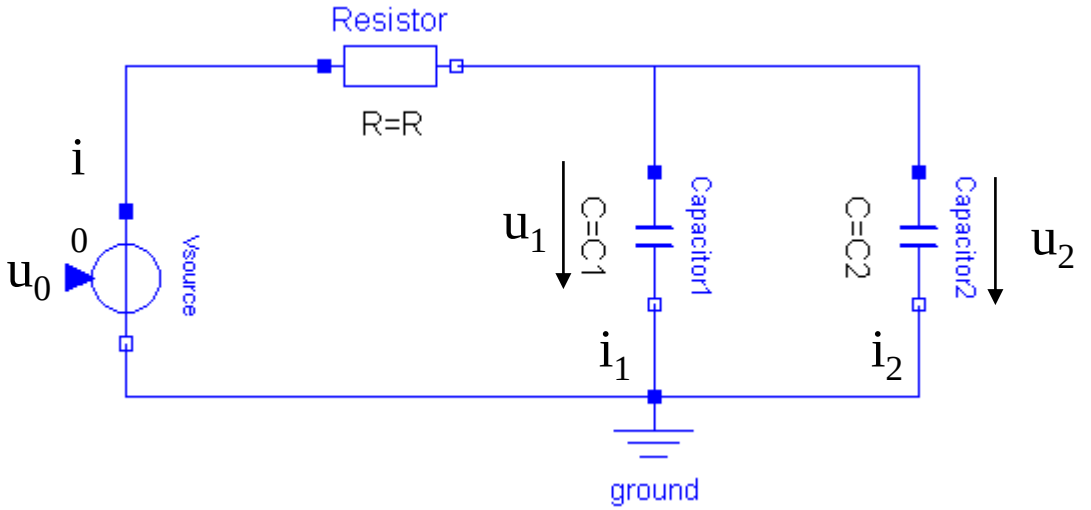
BLT



residual

It can be determined that this equation system is linear in the unknowns and it can be transformed to one linear equation in one unknown (`c1 * Ri.v = c2`) that can be easily solved.

Index Reduction



singular DAE

$$\mathbf{0} = \mathbf{f}(\dot{\mathbf{x}}, \mathbf{x}, \boldsymbol{w}, t)$$

$$\begin{aligned}
0 &= C_1 \cdot \dot{u}_1 - i_1 \\
0 &= C_2 \cdot \dot{u}_2 - i_2 \\
0 &= u_1 - u_2 \\
0 &= i_0 - i_1 - i_2 \\
0 &= u_0 - R \cdot i_0 - u_1
\end{aligned}$$

given: $u_0(t)$, C_1, C_2, R

$$\mathbf{x} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \qquad \mathbf{z} = \begin{bmatrix} \dot{\mathbf{x}} \\ \boldsymbol{w} \end{bmatrix}$$

$$\mathbf{w} = \begin{bmatrix} i_0 \\ i_1 \\ i_2 \end{bmatrix} \qquad \mathbf{z} = \begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \\ i_0 \\ i_1 \\ i_2 \end{bmatrix}$$

$$\frac{\partial \mathbf{f}}{\partial \mathbf{z}} = \begin{bmatrix} C_1 & 0 & 0 & -1 & 0 \\ 0 & C_2 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 \\ 0 & 0 & -R & 0 & 0 \end{bmatrix}$$

singular

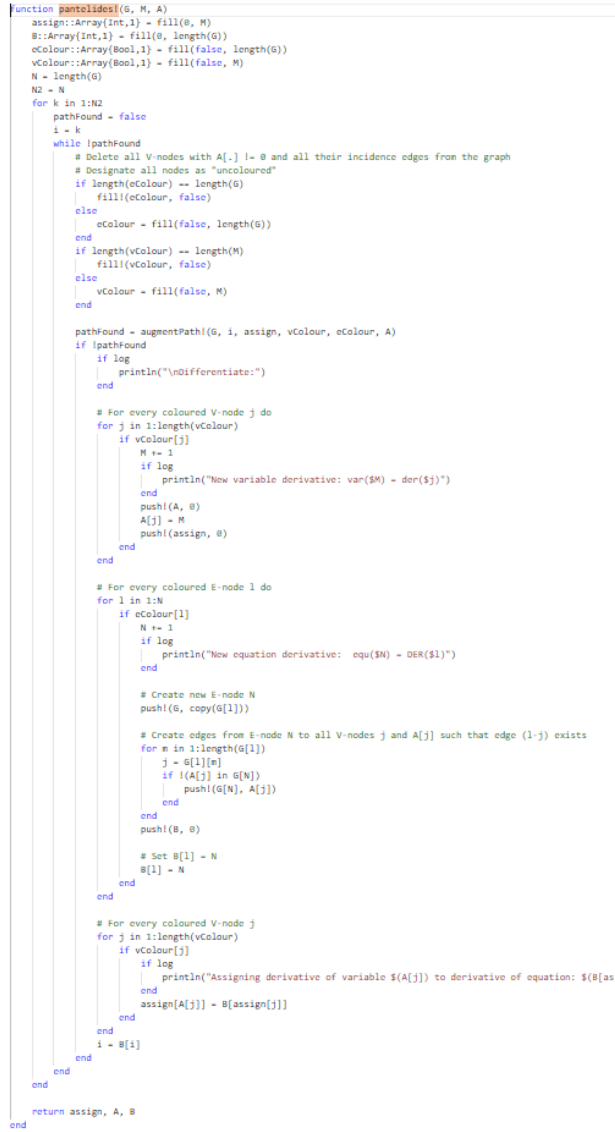
General method to solve such singular DAEs automatically

1. **Determine equations that need to be differentiated** by a structural algorithm
   (open source implementation: function pantelides!(..) in ModiaBase)

2. **Differentiate** the identified equations **analytically**
   (open source implementation: function derivative(..) in ModiaBase)

3. **Select states** and **sort the** original and differentiated **equations**
   The system (with the original and differentiated equations) is a regular DAE **after selecting the "real" states** and the methods from lecture 6 can be used to transform this DAE to ODE form.
   (open source implementation: function getSortedAndSolvedAST(..) in ModiaBase)

4. **Generate Code**
   This might include code to **numerically** solve linear and/or nonlinear equation systems.
   (open source implementations:
   function generate_getDerivatives! (..) in Julia package ModiaLang generates a Julia function getDerivatives!(..) that can be
   utilized by Julia package DifferentialEquations which provides a huge amount of integrators to simulate this function)



```
function pantelides!(G, M, A)
    assign::Array{Int,1} = fill(0, M)
    B::Array{Int,1} = fill(0, length(G))
    eColour::Array{Bool,1} = fill(false, length(G))
    vColour::Array{Bool,1} = fill(false, M)
    N = length(G)
    N2 = N
    for k in 1:N2
        pathFound = false
        i = k
        while !pathFound
            # Delete all V-nodes with A[.] != 0 and all their incidence edges from the graph
            # Designate all nodes as "uncoloured"
            if length(eColour) == length(G)
                fill!(eColour, false)
            else
                eColour = fill(false, length(G))
            end
            if length(vColour) == length(M)
                fill!(vColour, false)
            else
                vColour = fill(false, M)
            end

            pathFound = augmentPath!(G, i, assign, vColour, eColour, A)
            if !pathFound
                if log
                    println("\nDifferentiate:")
                end

                # For every coloured V-node j do
                for j in 1:length(vColour)
                    if vColour[j]
                        M += 1
                        if log
                            println("New variable derivative: var($M) = der($j)")
                        end
                        push!(A, 0)
                        A[j] = M
                        push!(assign, 0)
                    end
                end

                # For every coloured E-node l do
                for l in 1:N
                    if eColour[l]
                        N += 1
                        if log
                            println("New equation derivative:  equ($N) = DER($l)")
                        end

                        # Create new E-node N
                        push!(G, copy(G[l]))

                        # Create edges from E-node N to all V-nodes j and A[j] such that edge (l-j) exists
                        for m in 1:length(G[l])
                            j = G[l][m]
                            if !(A[j] in G[N])
                                push!(G[N], A[j])
                            end
                        end
                        push!(B, 0)

                        # Set B[l] = N
                        B[l] = N
                    end
                end

                # For every coloured V-node j
                for j in 1:length(vColour)
                    if vColour[j]
                        if log
                            println("Assigning derivative of variable $(A[j]) to derivative of equation: $(B[as
                        end
                        assign[A[j]] = B[assign[j]]
                    end
                end
                i = B[i]
            end
        end
    end

    return assign, A, B
end
```

Exercise: Understanding the Basic Algorithms

> using ModiaBase

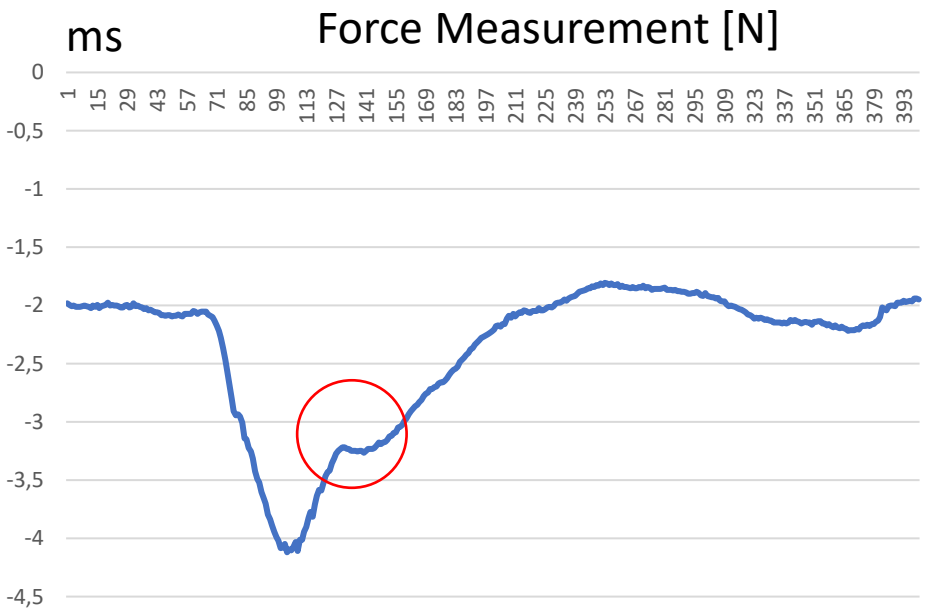
> include("$(ModiaBase.path)/test/runtests.jl")

```
@testset "Test ModiaBase" begin
    include("TestSymbolic.jl")
    include("TestBLTandPantelides.jl")
    include("TestDifferentiate.jl")
    include("TestTearing.jl")
    include("TestLinearIntegerEquations.jl")
    include("TestLinearEquations.jl")
end
```

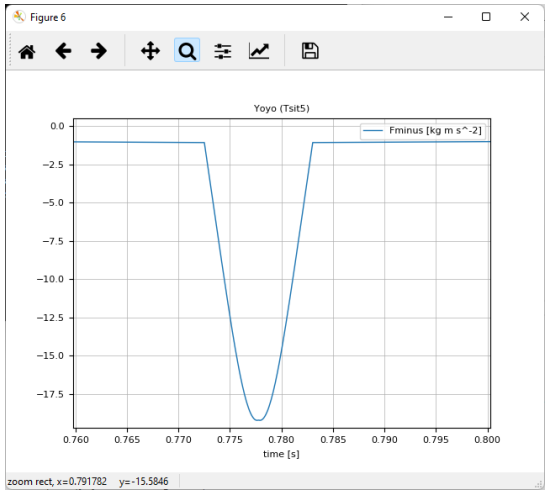
Example - Yoyo

- Determine force when yoyo changes direction
  - Assumption: no sliding, no losses, no elasticity in string
- Constraint between translation and rotation
- Varying radius

First Experiment

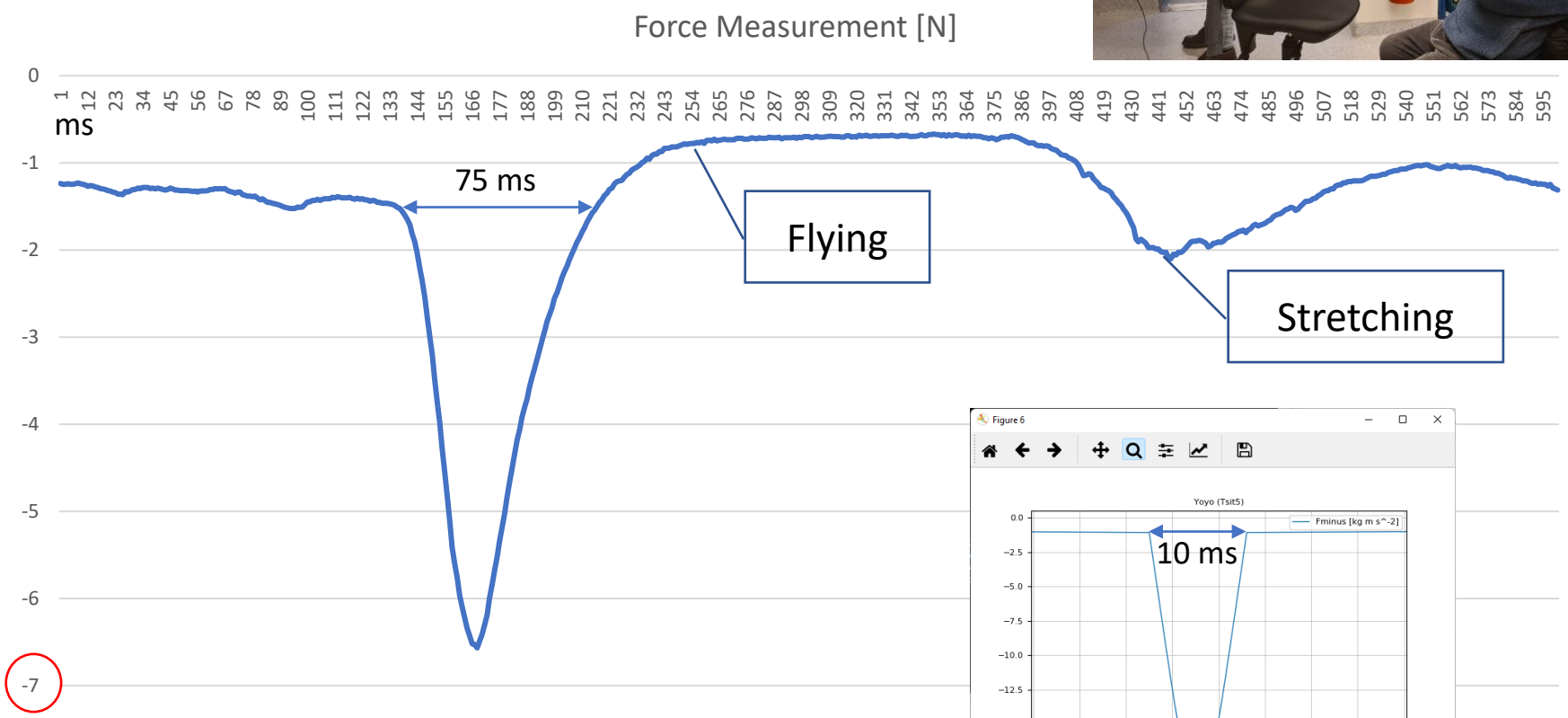


Force (model) [N]

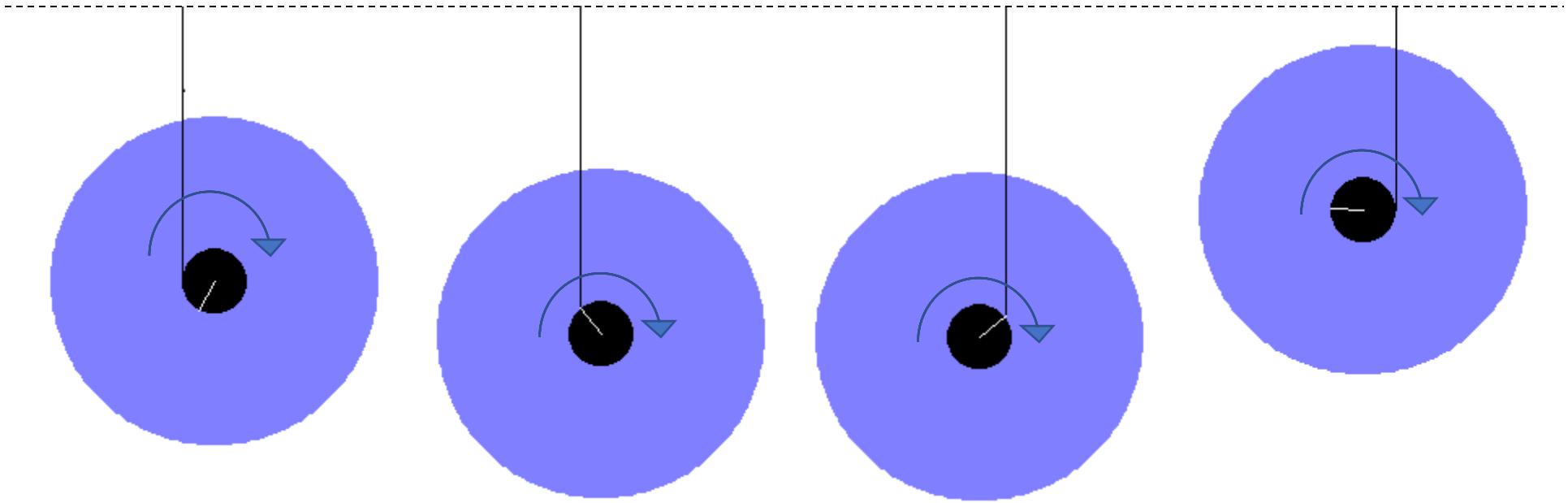


Improved Experiment

Experiment setup:
Julian Salt Ducaju



Constraint modeling - Yoyo



Coupling between position and orientation
When changing translational direction

Yoyo Model

```
Yoyo = Model(
    x = Var(), # Vertical position of Yoyo center
    theta = Var(init=100),
    w = Var(init=0u"1/s"),
    r0=0.004u"m",
    m=0.050u"kg",
    J=9e-6u"kg*m*m",
    g=9.81u"m/s/s",
    pi = Base.MathConstants.pi,
    stringThickness = 0.0005u"m",

equations = :[
    v = der(x)
    w = der(theta)
    m*der(v) = F - m*g
    r = r0 + stringThickness/2/pi*(if theta>0; theta else -theta end)
    J*der(w) = tau
    tau = F*y
    x = if positive(pi/2-abs(theta)); -r*cos(theta) elseif positive(theta-pi/2); (theta - pi/2)*r else (-theta
          - pi/2)*r end
    y = if positive(pi/2-abs(theta)); -r*sin(theta) elseif positive(theta-pi/2); -r              else r end
    Fminus = -F
    ]
)
```

Symbolically differentiated equations

First derivative

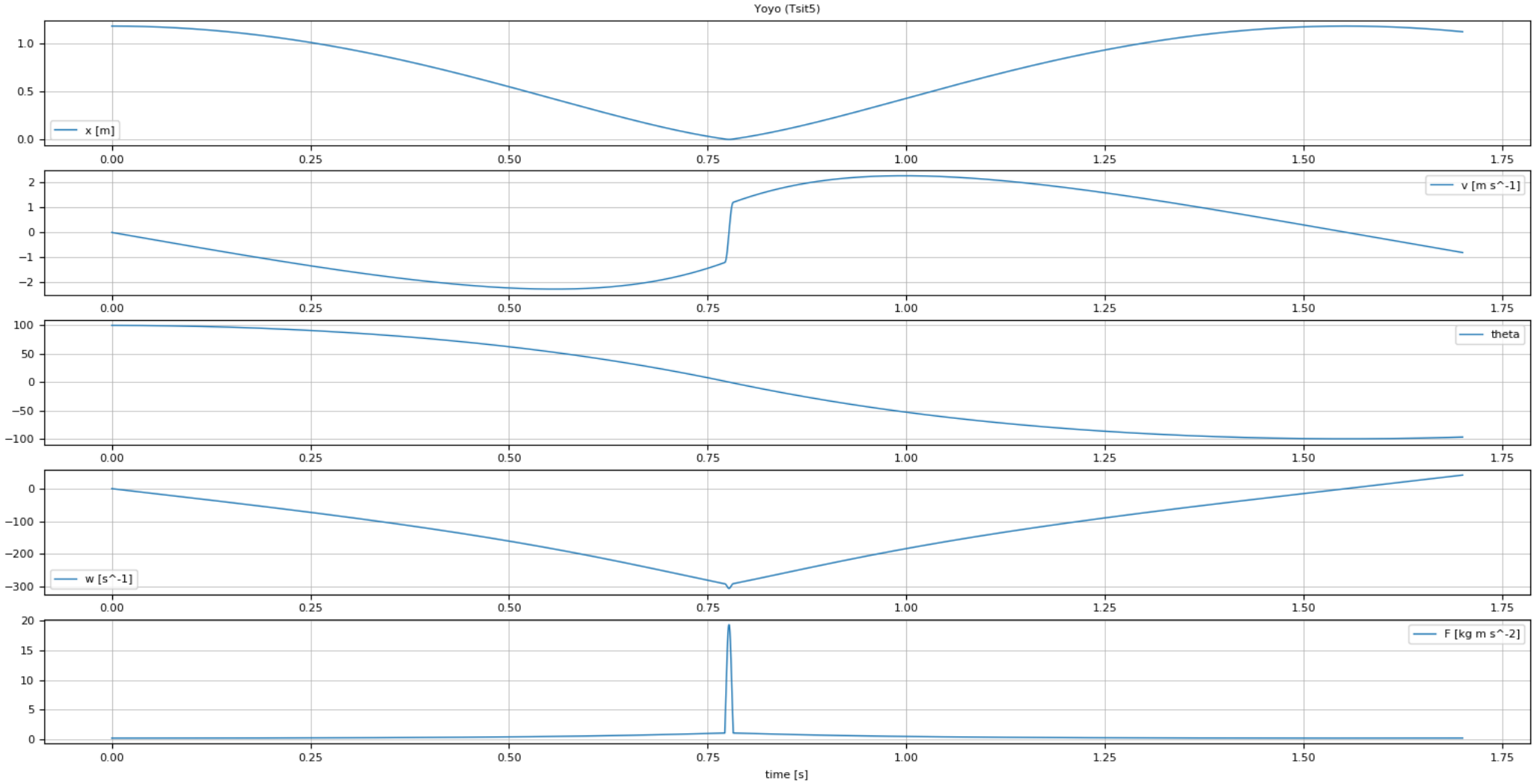
Second derivative

```
v = der(x)
  der(v) = der(der(x))
w = der(theta)
  der(w) = der(der(theta))

r = r0 + ((stringThickness / 2) / pi) * if theta > 0; theta else -theta end
  der(r) = +(((stringThickness / 2) / pi) * if theta > 0; der(theta) else -(der(theta)) end)
    der(der(r)) = +(((stringThickness / 2) / pi) * if theta > 0; der(der(theta)) else -(der(der(theta))) end)

x = if positive(pi / 2 - abs(theta)); -r * cos(theta) elseif positive(theta - pi / 2); (theta - pi / 2) * r else (-theta - pi / 2) * r end
  der(x) = if positive(pi / 2 - abs(theta)); -(der(r)) * cos(theta) + -r * (-(sin(theta)) * der(theta)) elseif positive(theta - pi / 2);
        der(theta) * r + (theta - pi / 2) * der(r) else -(der(theta)) * r + (-theta - pi / 2) * der(r) end
    der(der(x)) = if positive(pi / 2 - abs(theta))
            (-(der(der(r))) * cos(theta) + -(der(r)) * (-(sin(theta)) * der(theta))) +
                (-(der(r)) * (-(sin(theta)) * der(theta)) + -r * (-(cos(theta) * der(theta)) * der(theta) + -(sin(theta)) * der(der(theta))))
        elseif positive(theta - pi / 2)
            (der(der(theta)) * r + der(theta) * der(r)) + (der(theta) * der(r) + (theta - pi / 2) * der(der(r)))
        else
            (-(der(der(theta))) * r + -(der(theta)) * der(r)) + (-(der(theta)) * der(r) + (-theta - pi / 2) * der(der(r)))
        end
```

Simulation

```
module YoyoModule

using Modia
@usingModiaPlot

# Insert Yoyo model

yoyo = @instantiateModel(Yoyo, log=true, logCode=true)
simulate!(yoyo, Tsit5(), stopTime = 1.7u"s", log=true, logStates=true, interval=0.0001)
plot(yoyo, ["x", "v", "theta", "w", "F"], figure = 1)

end
```

Yoyo simulation results



Suggested Exercises

- Include flexibility in string
- Include loss

Constraints in Thermodynamical Systems

- Drum model – Åström, Bell
  - Simon:
  - https://www.sciencedirect.com/science/article/pii/S1474667017537349
  - Matlab
  - https://lucris.lub.lu.se/ws/portalfiles/portal/4879107/8727216.pdf
  - https://www.sciencedirect.com/science/article/abs/pii/S000510989900017
- Volume constraint
- Equations differentiated and manipulated

SIMPLE DRUM-BOILER MODELS

K. J. Åström* and R. D. Bell**

*Department of Automatic Control, Lund Institute of Technology, S-221 00 Lund, Sweden

**School of Mathematics and Physics, Macquarie University, North Ryde, New South Wales 2113, Australia

Abstract

This paper describes a simple nonlinear models for a drum-boiler. The models are derived from first principles. They can be characterized by a few physical parameters that are easily obtained from construction data. The models also require steam tables for a limited operating range, which can be approximated by polynomials. The models have been validated against experimental data. A complete simulation program is provided.

1. Introduction

There are many models of drum-boilers in the literature. See the reference list. The models described in this paper are derived from first principles. They are characterized by a few parameters only which can be obtained from first principles. The models are validated by comparison with extensive plant data.

A key feature of a drum-boiler is that there is a very efficient energy and mass transfer between all parts that are in contact with the steam. The mechanism responsible for the heat transfer is boiling and condensation. A consequence of this is that it is a very good approximation to assume that all water steam and metal is in thermal equilibrium. This means that the total energy can be represented by a global energy balance. The validity of this approximation has been shown by many modeling exercises.

The paper is organized as follows. A first order model is presented in Section 2. This model is obtained from a global energy balance for the total plant. The model has one state variable which is chosen as the drum pressure. This model has the same structure as the model presented in Åström and Eklund (1972). The parameters are, however, obtained from first principles. To model the drum water level it is necessary to account for the shrink and swell phenomena. This is done in Section 3. A third order model is obtained. This model has drum pressure, water volume and steam quality in the risers as state variables. The model exhibits a complex behaviour in spite of being of low order. Simulation of step responses are presented in Section 4.

2. A First Order Model

Because of the efficient heat and mass transfer due to boiling and condensation all parts of the system which are in contact with the steam will be in thermal equilibria. It is therefore natural to describe the plant with global mass and energy balances as was done in Åström and Eklund (1972). The global energy balance can be written as

$$\frac{d}{dt}[\varrho_s h_s V_{st} + \varrho_w h_w V_{wt} + mc_p T] = P + q_{fw} h_{fw} - q_s h_s \quad (1)$$

where ϱ denotes specific density, h enthalphy, V volume and q mass flow. The indices s, w and fw refers to steam, water and feedwater respectively. The total mass of the metal tubes is m, the specific heat is c_p and the average metal temperature is T.

The input power from the fuel is denoted by P. The total steam volume is given by

$$V_{st} = V_{drum} - V_w + a_m V_r \quad (2)$$

where V_{drum} is the drum volume, V_w the volume of water in the drum, V_r the riser volume and a_m the average steam-water volume ratio. The total water volume is

$$V_{wt} = V_w + V_{dc} + (1 - a_m)V_r \quad (3)$$

The right hand side of equation (1) represents the energy flow to the system from fuel and feedwater and the energy flow from the system via the steam. Since all parts are in thermal equilibria the state of the system can be represented by one variable which we choose as the steam pressure. Using steam tables the variables ϱ_s, ϱ_w, h_s and h_w can then be expressed as functions of steam pressure. Similarly T can be expressed as a function of pressure by assuming that T is equal to the saturation temperature of steam which corresponds to p.

This model represents the dynamics due to input power well. When the feedwater flow or the steam flow is changed it is, however, necessary to also take into account that the water and steam masses are also changing. This can be accounted for with a global massbalance.

$$\frac{d}{dt}[\varrho_s V_{st} + \varrho_w V_{wt}] = q_{fw} - q_s \quad (4)$$

The dynamics which describe how the drum pressure is influenced by input power, feedwater flow and steam flow is well captured by equations (1) and (4).

The derivative of the total water volume (dV_{wt}/dt) can be eliminated between equations (1) and (4). Multiplication of (4) by h_w and subtracting from (1) gives

$$h_c \frac{d}{dt}(\varrho_s V_{st}) + \left[\varrho_s V_{st}\frac{dh_s}{dt} + \varrho_w V_{wt}\frac{dh_w}{dt} + mc_p\frac{dT}{dt}\right] = P - q_{fw}(h_w - h_{fw}) - q_s h_c \quad (5)$$

The condensation enthalpy $h_c = h_s - h_w$ has also been introduced. If the boiler is provided with a good level control system the total water volume (V_{wt}) and the total steam volume (V_{st}) do not change much. Equation (5) can then be simplified to

$$e_{11}\frac{dp}{dt} = P - q_{fw}(h_w - h_{fw}) - q_s h_c \quad (6)$$

where

$$e_{11} = h_c V_{st}\frac{d\varrho_s}{dp} + \varrho_s V_{st}\frac{dh_s}{dp} + \varrho_w V_{wt}\frac{dh_w}{dp} + mc_p\frac{dT_s}{dp}$$

Apart from steam table data it is thus sufficient to know total steam and water volumes and total metal mass. The model (6) is identical to the model in Åström and Eklund (1972). Notice however that in this case the parameters are obtained from construction data. Also notice that the term

$$q_c = -\frac{1}{h_c}\left[\varrho_s V_{st}\frac{dh_s}{dt} + \varrho_w V_{wt}\frac{dh_w}{dt} + mc_p\frac{dT_s}{dt}\right]$$

Equations

```
# Total energy balance.
Ed = rs*hs*Vst + rw*hw*Vwt + m*cp*T
der(Ed) = P + qfw*hfw - qs*hs # Eq. 1

# Total steam volume.
Vst = Vdrum - Vw + am * Vr # Eq. 2

# Total water volume.
Vwt = Vw + Vdc + (1 - am) * Vr # Eq. 3

# Total mass balance.
Md = rs*Vst + rw*Vwt
der(Md) = qfw - qs # Eq. 4

# Mass balance in risers.
Mr = rs*am*Vr + rw*(1 - am)*Vr
der(Mr) = qdc - qr # Eq. 8

# Energy balance in risers.
Er = rs*hs*am*Vr + rw*hw*(1 - am)*Vr
der(Er) = P + qdc*hw - xr*qr*hs - (1-xr)*qr*hw # Eq. 9

# Average steam-water volume ratio in the risers.
am = rw/(rw - rs) * (1 - rs/(rw - rs)/xr * log(1 + (rw - rs) / rs * xr)) # Eq. 11

# Drum water level.
dl = (Vw + am * Vr)/adrum # Eq. 12

# Momentum balance for downcomers.
am*Vr*(rw - rs) = k*qdc^2/2 # Eq. 13

# Drum level
lv = Vw/adrum
lr = am*Vr/adrum
dl = lr + lw
```

Differentiated equations

```
Ed = rs * hs * Vst + rw * hw * Vwt + m * cp * ts
  der(Ed) = (der(rs) * hs * Vst + rs * der(hs) * Vst + rs * hs * der(Vst)) + (der(rw) * hw * Vwt + rw * der(hw) * Vwt + rw * hw * der(Vwt)) + m * cp * der(ts)
Vst = (Vdrum - Vw) + am * Vr
  der(Vst) = -(der(Vw)) + der(am) * Vr
Vwt = Vw + Vdc + (1 - am) * Vr
  der(Vwt) = der(Vw) + -(der(am)) * Vr
Md = rs * Vst + rw * Vwt
  der(Md) = (der(rs) * Vst + rs * der(Vst)) + (der(rw) * Vwt + rw * der(Vwt))
Mr = rs * am * Vr + rw * (1 - am) * Vr
  der(Mr) = (der(rs) * am * Vr + rs * der(am) * Vr) + (der(rw) * (1 - am) * Vr + rw * -(der(am)) * Vr)
Er = rs * hs * am * Vr + rw * hw * (1 - am) * Vr
  der(Er) = (der(rs) * hs * am * Vr + rs * der(hs) * am * Vr + rs * hs * der(am) * Vr) + (der(rw) * hw * (1 - am) * Vr + rw * der(hw) * (1 - am) * Vr + rw * hw * -(der(am)) * Vr)

hs = aOl + (all + a21 * (p - 10)) * (p - 10)
  der(hs) = +((+(a21 * der(p)) * (p - 10) + (all + a21 * (p - 10)) * der(p)))
rs = a02 + (a12 + a22 * (p - 10)) * (p - 10)
  der(rs) = +((+(a22 * der(p)) * (p - 10) + (a12 + a22 * (p - 10)) * der(p)))
hw = a03 + (a13 + a23 * (p - 10)) * (p - 10)
  der(hw) = +((+(a23 * der(p)) * (p - 10) + (a13 + a23 * (p - 10)) * der(p)))
rw = a04 + (a14 + a24 * (p - 10)) * (p - 10)
  der(rw) = +((+(a24 * der(p)) * (p - 10) + (a14 + a24 * (p - 10)) * der(p)))
ts = a05 + (a15 + a25 * (p - 10)) * (p - 10)
  der(ts) = +((+(a25 * der(p)) * (p - 10) + (a15 + a25 * (p - 10)) * der(p)))
```

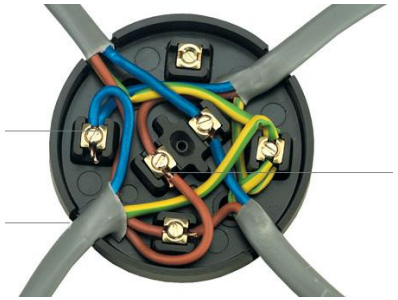
Suggested Exercise

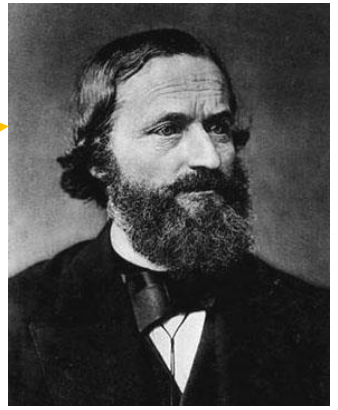
- Make the model complete
- Replicate the published simulation results

Part 2 - Object-oriented Modeling

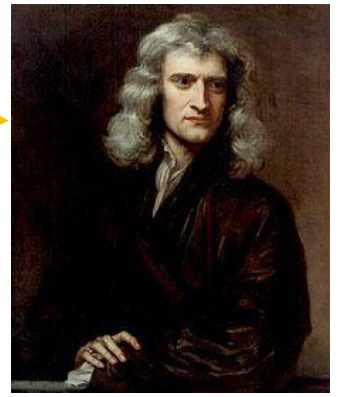
The Roots - Ideal Connection semantic

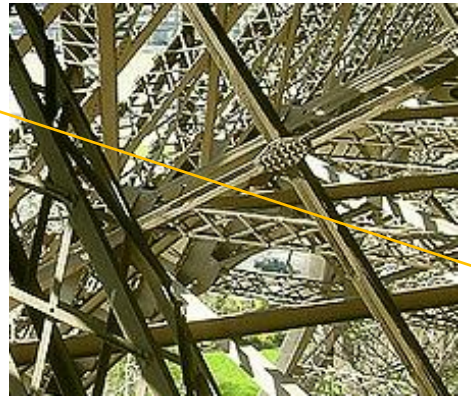
- Electrical: Kirchhoff's current law, 1845
  - **Sum of currents** at junction is zero



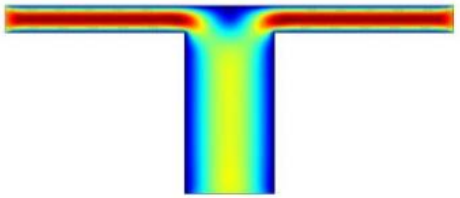


- Mechanics: Newton´s (1687) and Euler's (about 1737) second laws:
  - The **vector sum of the forces** on an object is equal to the mass of that object multiplied by the acceleration vector of the object.
  - The rate of change of angular momentum about a point that is fixed in an inertial reference frame, is equal to the **sum of torques** acting on that body about that point.
  - Neglect mass and moment of inertia at junction → **Sum of forces** are zero **and sum of torques** are zero





- Fluid systems:
  - Consider a small volume at junction →
  - Mass balance: **Sum of mass flow rates** are zero
  - Energy balance: **Sum of energy flow rates** are zero



History – The roots of Modelica

- Maxwell (1873) introduced Force-Voltage Analogy
  - Effort and flow variables
  - Mass ≈ inductance
  - **Series** connection of electrical component **correspond** to **parallel** connection of mechanical components and vice versa
  - Paynter (1960): Bond graphs
- Firestone (1933) introduced Force-Current Analogy
  - Across (relative quantities) and Through variables
  - Mass ≈ Capacitor (Mass has reference to ground)
  - Kirchhoff's current law, etc – sum of through variables are zero
- Trent (1955): Isomorphism between Oriented Linear Graphs and Lumped Physical Systems

Modia/Modia3D

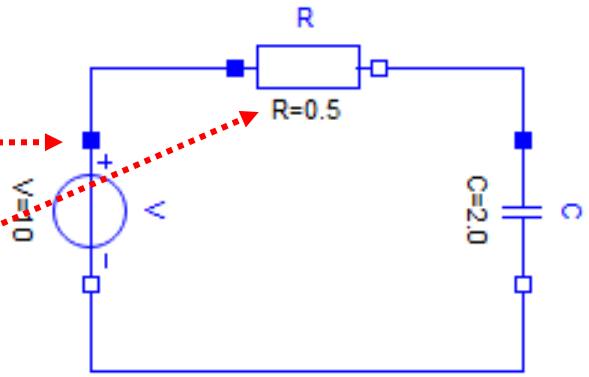
- Modia Evolution
  - Experimentation in Julia started in 2016 => Modia 0.3
  - New design - Started autumn 2020 => Modia 0.5
- Essentials
  - Simplification and unification of constructs
  - Coupling between Modia acausal models and Modia3D (multibody)
  - Focus on scalability
  - Use of DifferentialEquations.jl (generating ODE instead of DAE)

Object-oriented Modeling in Modia

```
Pin = Model(v = potential, i = flow)
```

```
OnePort = Model(
   p = Pin,
   n = Pin,
   equations = :[
      0 = p.i + n.i
      v = p.v - n.v
      i = p.i ])
```

```
Resistor  = OnePort | Model(R = missing,
                    equations = :[ R*i = v ] )

Capacitor = OnePort | Model(C = missing,
                    v = Var(init=0.0u"V"),
                    equations = :[ C*der(v) = i ] )

ConstantVoltage = OnePort | Model(V = missing,
                    equations = :[ v = V ] )
```



Simple electrical circuit
(Modelica schematic)

Parameterize: **merge** with Map

```
Circuit = Model(
   R = Resistor  | Map(R=0.5u"Ω"),
   C = Capacitor | Map(C=2.0u"F"),
   V = ConstantVoltage | Map(V=10.0u"V"),
   equations = :[connect(V.p, R.p)
                 connect(R.n, C.p)
                 connect(C.n, V.n)])
```

Units

Inherit: hierarchically **merge** models

Unification - Dictionaries and merge

Modia (low level) language elements

- **Model** - hierarchical dictionary of variables and components and array of equations
- **Var** - dictionary of variable attributes (min, max, etc)
  - Domain specific attributes
  - Predefined: flow = Var(flow=true), …
- **Map** - hierarchical dictionary to set and modify parameters

- | (merge) - Overloading of *bitwise or*
  - Inheritance - Merge between models (including concatenation of equations arrays)
  - Setting parameters (numeric and generic)
  - Modifying parameters (hierarchically)

```
function Base.:|(x, y)
  result = deepcopy(x)
  for (key, value) in y
    if typeof(value) <: AbstractDict && key in keys(result)
      value = result[key] | value
    elseif key in keys(result) && key == :equations
      equa = copy(result[key])
      push!(equa.args, value.args...)
      value = equa
    end
    result[key] = value
  end
  return result
end
```

Recursion

Typo in paper

Merging for Circuit.R

```
potential = Var(
  potential = true
)

flow = Var(
  flow = true
)

Pin = Model(
  v = potential,
  i = flow
)

OnePort = Model(
  p = Pin,
  n = Pin,
  equations = :[
    0 = p.i + n.i
    v = p.v - n.v
    i = p.i
  ]
)
```

Expanded

```
OnePort = Model(
  p = Model(
    v = Var(
      potential = true,
    ),
    i = Var(
      flow = true,
    ),
  ),
  n = Model(
    v = Var(
      potential = true,
    ),
    i = Var(
      flow = true,
    ),
  ),
  equations = :[
    0 = p.i + n.i
    v = p.v - n.v
    i = p.i
  ]
)
```

Merge

```
Resistor  =
OnePort |
Model(
  R = missing,
  equations = :[
    R * i = v
  ]
)
```

Expanded

```
Resistor = Model(
  p = Model(
    v = Var(
      potential = true,
    ),
    i = Var(
      flow = true,
    ),
  ),
  n = Model(
    v = Var(
      potential = true,
    ),
    i = Var(
      flow = true,
    ),
  ),
  equations = :[
    0 = p.i + n.i
    v = p.v - n.v
    i = p.i
    R * i = v
  ]
  R = missing,
)
```

Merge

```
R = Resistor  |
Map(R=0.5u"Ω")
```

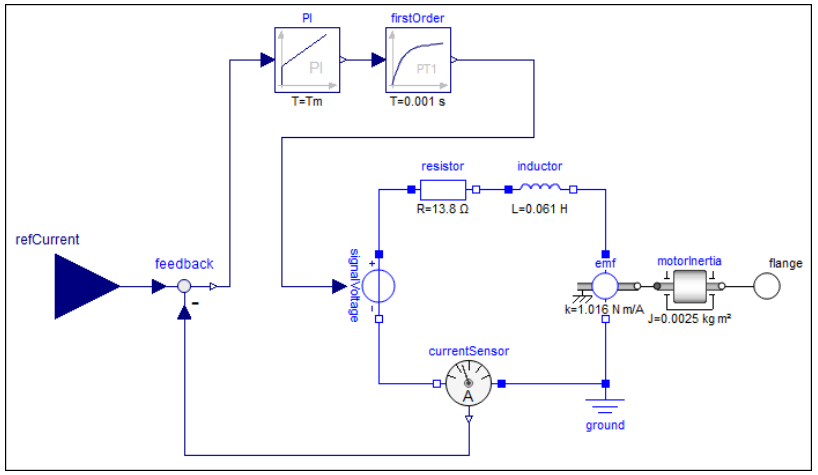
Expanded

```
Circuit.R = Model(
  p = Model(
    v = Var(
      potential = true,
    ),
    i = Var(
      flow = true,
    ),
  ),
  n = Model(
    v = Var(
      potential = true,
    ),
    i = Var(
      flow = true,
    ),
  ),
  equations = :[
    0 = p.i + n.i
    v = p.v - n.v
    i = p.i
    R * i = v
  ]
  R = 0.5 Ω,
)
```

Modia Example: Servo system

- From: https://github.com/ModiaSim/Modia.jl/blob/main/examples/ServoSystem.jl

```
using Modia
@usingModiaPlot
include("$(Modia.modelsPath)/Blocks.jl")
include("$(Modia.modelsPath)/Electric.jl")
include("$(Modia.modelsPath)/Rotational.jl")

ControlledMotor = Model(
  refCurrent = input,
  flange = Flange,
  feedback      = Feedback,
  PI            = PI | Map(k=30, T=1.0u"s"),
  firstOrder    = FirstOrder | Map(k=1.0, T=0.001u"s"),
  signalVoltage = UnitlessSignalVoltage,
  resistor      = Resistor | Map(R=13.8u"Ω"),
  inductor      = Inductor | Map(L=0.061u"H"),
  emf           = EMF | Map(k=1.016u"N*m/A"),
  ground        = Ground,
  currentSensor = UnitlessCurrentSensor,
  motorInertia  = Inertia | Map(J=0.0025u"kg*m^2"),

  connect = :[
    (refCurrent, feedback.u1)
    (feedback.y, PI.u)
    (PI.y, firstOrder.u)
    (firstOrder.y, signalVoltage.v)

    (signalVoltage.p, resistor.p)
    (resistor.n, inductor.p)
    (inductor.n, emf.p)
    (emf.n, ground.p, currentSensor.p)
    (currentSensor.n, signalVoltage.n)
    (currentSensor.i, feedback.u2)

    (emf.flange, motorInertia.flange_a)
    (motorInertia.flange_b, flange) ]
)
```



Suggested Exercises

- Run examples:
  - include("$(Modia.path)/examples/runexamples.jl")
- Select some examples with logging, see:

  ? @instantiateModel

  ? simulate!
- For example:
  - include("$(Modia.path)/examples/ServoSystem.jl")

Upcoming Modia Syntax

- Using macro **@define**
- Allows pre-transformation of AST
- More Modelica like
- Easier to comprehend

Upcoming Modia

```
@define RCCircuit = Model(
   r = Resistor(R=2),
   c = Capacitor(C=5),
   source = ConstantVoltage(V=10),
   equations = :[
      connect(source.p, r.p)
      connect(r.n, c.p)
      connect(source.n, c.n)
   ]
)

@define RCCircuits = Model(
   rc1 = RCCircuit(r(R=4), c(C=10)),
   rc2 = RCCircuit(c.C=10, c.v.init=5, source.V=-10)
)
```

Usual constructor syntax

Nested modifier

Modia with merge

```
RCCircuit = Model(
  r = Resistor | Map(R = 2)
  c = Capacitor | Map(C = 5)
  source = ConstantVoltage | Map(V = 10)
  equations = :[
    connect(source.p, r.p);
    connect(r.n, c.p);
    connect(source.n, c.n)
  ]
)

RCCircuits = Model(
  rc1 = RCCircuit | Map(r = Map(R = 4)) |
        Map(c = Map(C = 10))
  rc2 = RCCircuit | Map(c = Map(C = 10)) |
        Map(c = Map(v = Map(init = 5))) |
        Map(source = Map(V = -10))
)
```

Modifier with dot-notation

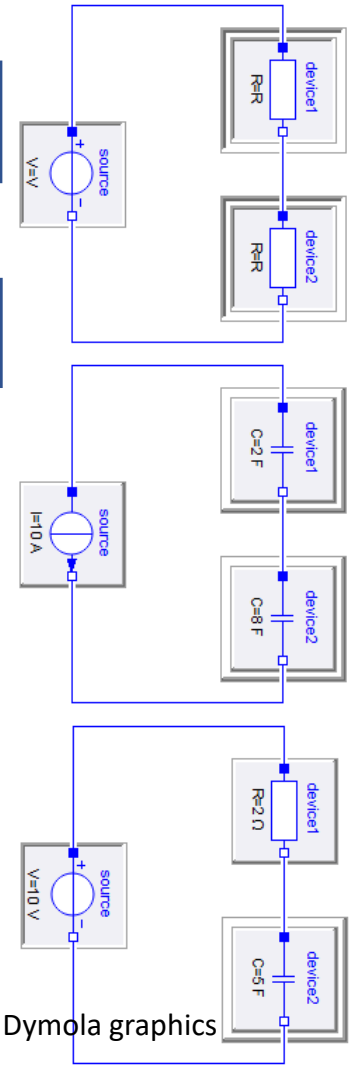
Upcoming Modia Syntax - generics

Modia

```
@define Circuit = Model(
   DeviceType = Resistor,
   device1 = DeviceType(),
   device2 = DeviceType(),
   source = ConstantVoltage(),
   equations = :[
      connect(source.p, device1.p)
      connect(device1.n, device2.p)
      connect(source.n, device2.n)
   ]
)
```

Replaceable model

All components are replaceable



Courtesy of Dymola graphics

```
@define MixedCircuits = Model(
  c = Circuit(DeviceType = Capacitor(C=2), device2(C=8),
              source=ConstantCurrent(I=10)),

  rc = Circuit(DeviceType = Capacitor(), device1=Resistor(R=2),
               device2(C=5), source(V=10))

)
```

Modelica

```
model Circuit
  replaceable model DeviceType = Resistor constrainedby OnePort;
  replaceable DeviceType device1 constrainedby OnePort;
  DeviceType device2;
  replaceable ConstantVoltage source constrainedby OnePort;
  Ground ground;
equation
  connect(source.p, device1.p);
  connect(device1.n, device2.p);
  connect(source.n, device2.n);
  connect(source.n, ground.p);
end Circuit;
```

```
model MixedCircuits
  Circuit c(redeclare model DeviceType = Capacitor(C=2), device2(C=8),
            redeclare ConstantCurrent source(I=10));

  Circuit rc(redeclare model DeviceType = Capacitor,
             redeclare Resistor device1(R=2),
             device2(C=5), source(V=10));
end MixedCircuits;
```

Domain Specific Algorithms – 3D Mechanics

KUKA YouBot robots
Animation: three.js (open source) or DLR Visualization (freeware/commercial)

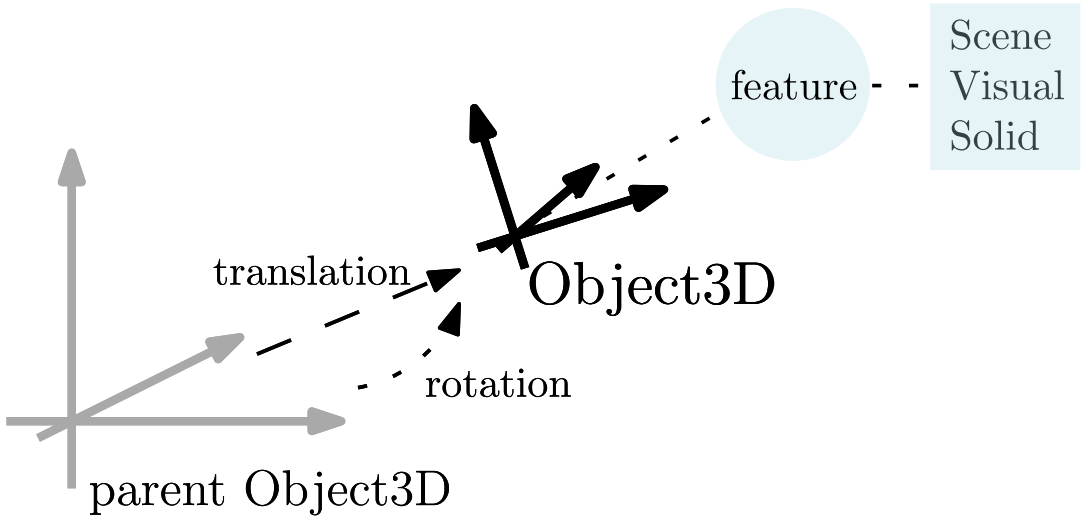
3D Mechanics with Modia3D

```
Lx = 0.5, angleY = pi/2, file = "base_frame.obj"
Object3D(
        parent = ...,
        translation = [-Lx/2,0,0],
        rotation = [0,angleY,0],
        feature = Solid(shape = FileMesh(filename = file))))
```

Modia3D constructor

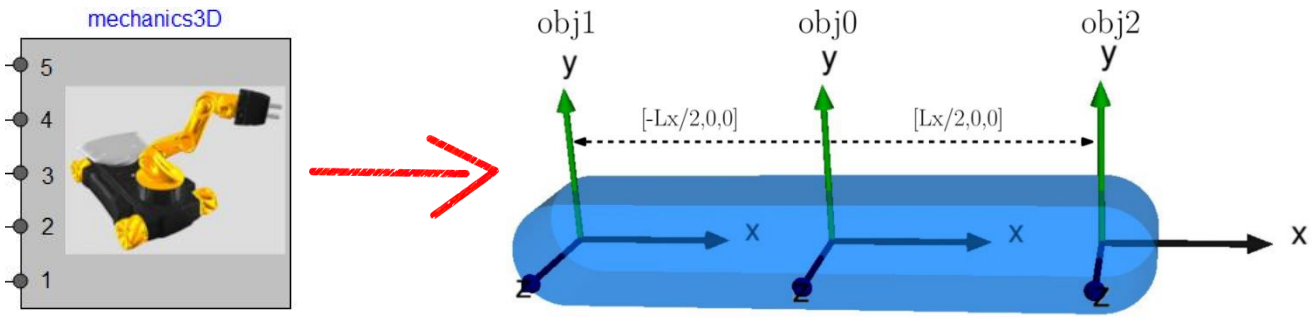
key

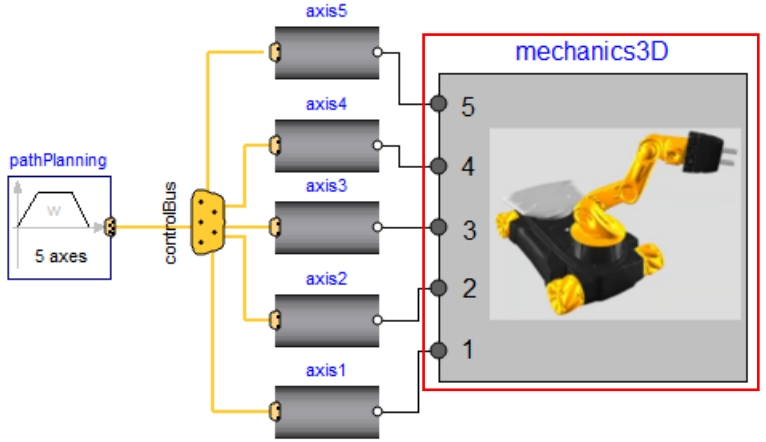
value



3D Mechanics Components

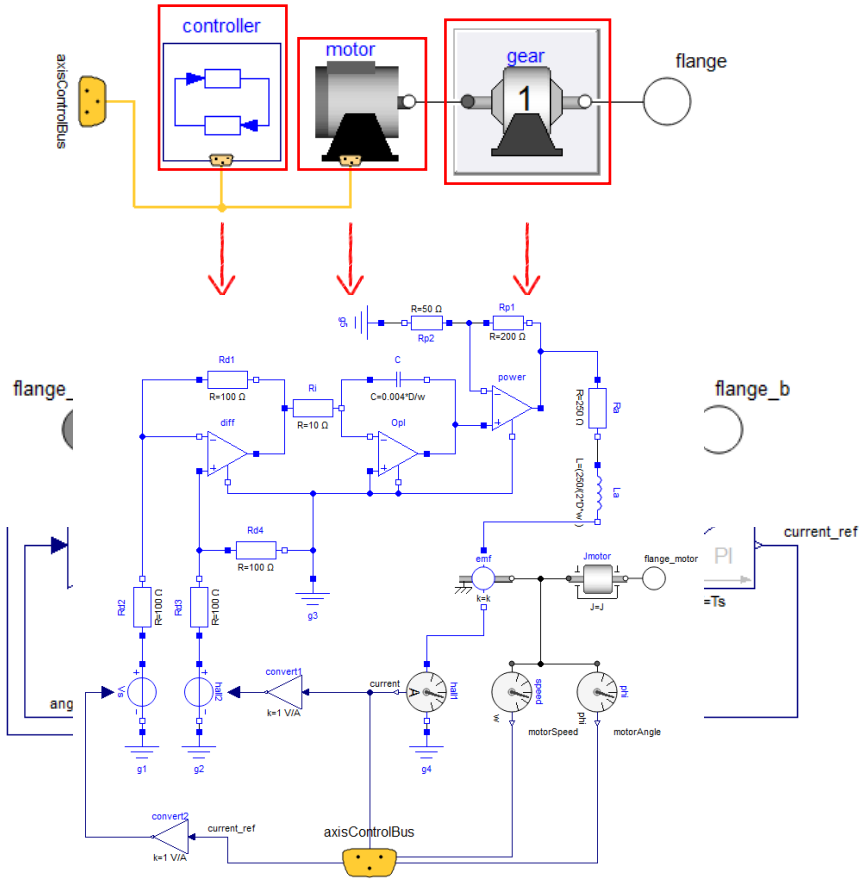
User's View

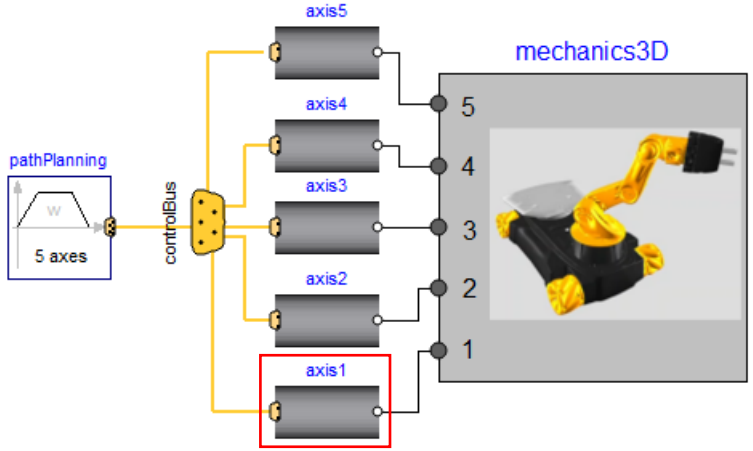





```
Link = Model(
    obj0 = Object3D(feature = Solid(shape = FileMesh(filename = :file)),
    obj1 = Object3D(parent = :obj0, translation = :[-Lx/2,0,0]),
    obj2 = Object3D(parent = :obj0, translation = :[ Lx/2,0,0]),
)
```

Equation Based Components



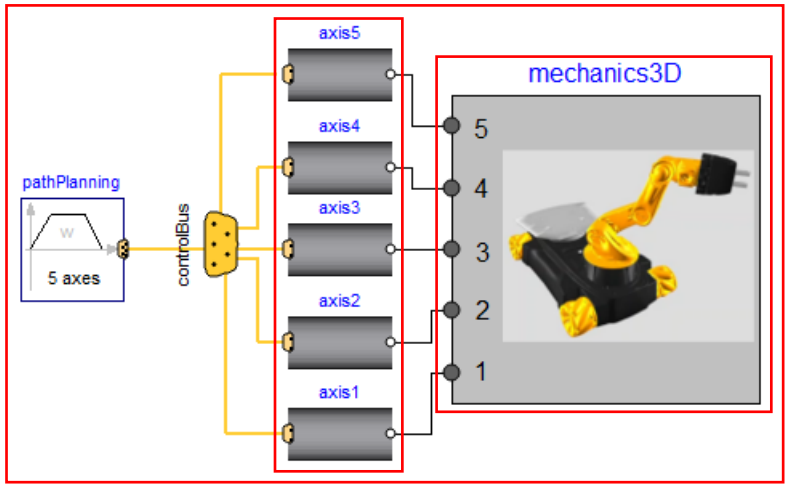



```
Axis = Model(
    motor = Motor,
    gear  = Gear,
    controller = Controller,
    ...,
    connect = :[(motor.flange, gear.flange_a),
                ...]
)
```

Model automatically converted from Modelica to Modia

Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.Components.AxisType1

Equation Based + 3D Mechanics




```
YouBot = Model(
    # Modia - 3D mechanics
    world = Object3D(feature = Scene()),
    base  = Base,
    link1 = Link | linkParameters1,
    link2 = Link | linkParameters2,
    rev1  = RevoluteWithFlange(obj1 = :(link1.obj2), obj2 = :(link2.obj1)),
    ...,

    # Modia - equations
    axis1 = Axis | axisParameters1,
    ...,

    # combine equations with 3D mechanics
    connect = :[(axis1.flange, rev1.flange)
                (axis2.flange, rev2.flange) ...]
)

youbot = @instantiatedModel(buildModia3D(YouBot))
```

domain specific function

Rattleback

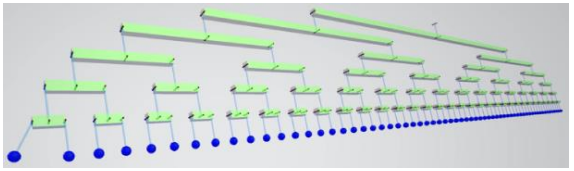
```
RattleBac = Model3D(
    world = Object3D(feature=Scene(provideAnimationHistory=true, enableContactDetection=true,
        gravityField=UniformGravityField(n=[0, 0, -1]))),
    ground = Object3D(parent=:world, translation=[0, 0, -0.015],
        feature=Solid(solidMaterial="DryWood", collision=true, shape=Box(lengthX=0.2, lengthY=0.2, lengthZ=0.01),
            visualMaterial=VisualMaterial(color=[100, 200, 50]))),
    ellipsoid = Object3D(translation=[0, 0, 0], rotation=[0, 0, 0],
        feature=Solid(solidMaterial="DryWood", collision=true, shape=Ellipsoid(lengthX=0.1, lengthY=0.02, lengthZ=0.02),
        massProperties=MassProperties(mass=0.014660765716752368, centerOfMass=[0, 0, 0],
            Ixx=6.39886e-7, Iyy=0.00000757014, Izz=0.0000076236, Ixy=-6.10996e-7),
        visualMaterial=VisualMaterial(color=[100, 150, 255]))),
    free = FreeMotion(obj1=:world, obj2=:ellipsoid,
        r=Var(init=Modia.SVector{3,Float64}(0, 0, 0.001)), w=Var(init=Modia.SVector{3,Float64}(0, 0.01, -5))),
 )
```

TippeTop

```
Unnamed = Model3D(
    world = Object3D(feature=Scene(provideAnimationHistory=true, enableContactDetection=true,
        gravityField=UniformGravityField(n=[0, 0, -1]))),
    ground = Object3D(parent=:world, translation=[0, 0, -0.01],
        feature=Solid(solidMaterial="BilliardTable", collision=true, shape=Box(lengthX=1, lengthY=1, lengthZ=0.02),
            visualMaterial=VisualMaterial(color=[100, 200, 50]))),
    ball = Object3D(feature=Solid(solidMaterial="BilliardBall", collision=true,
        shape=Sphere(diameter=0.04),
            massProperties=MassProperties(mass=0.07, centerOfMass=[0, 0, -0.01], Ixx=0.0000117, Iyy=0.0000117, Izz=0.0000187),
            visualMaterial=VisualMaterial(color=[255, 200, 150]))),
    rod = Object3D(parent=:ball, translation=[0, 0, 0.0225],
        feature=Solid(solidMaterial="BilliardBall", collision=false, shape=Cylinder(diameter=0.005, length=0.005),
            massProperties=MassProperties(mass=0.005),
            visualMaterial=VisualMaterial(color=[200, 50, 0]))),
    tip = Object3D(parent=:ball, translation=[0, 0, 0.025],
        feature=Solid(solidMaterial="BilliardBall", collision=true, shape=Sphere(diameter=0.005),
            massProperties=MassProperties(mass=0.0001, Ixx=1e-10, Iyy=1e-10, Izz=1e-10),
            visualMaterial=VisualMaterial(color=[0, 0, 0]))),
    free = FreeMotion(obj1=:world, obj2=:ball,
        r=Var(init=Modia.SVector{3,Float64}(0, 0, 0.02)), rot=Var(init=Modia.SVector{3,Float64}(0.05, 0, 0)),
        w=Var(init=Modia.SVector{3,Float64}(5, 0.01, 550))),
)
```

Benchmark

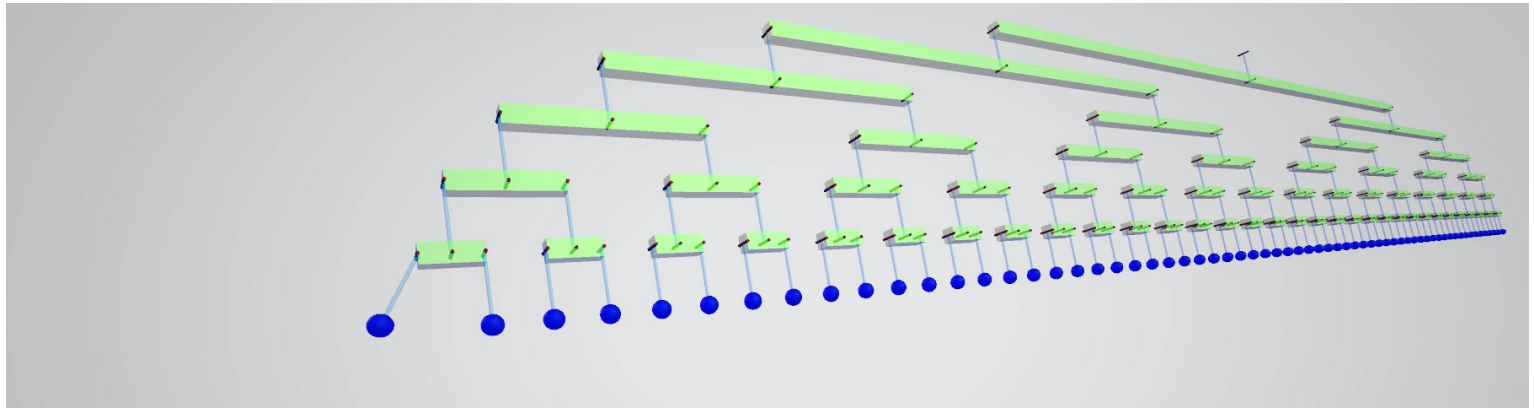
Compare large Modia models with equivalent Modelica models and various Modelica tools.



Recursively defined mobile with given tree depth:

- 3D mechanics (solid boxes, cylinders, spheres, and revolute joints)
- Rotational damper element in every joint

tree depth = 8:

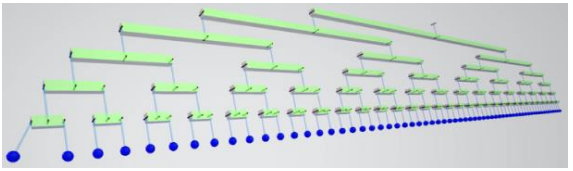


depth	*#states*	*#unknowns*	*#solids*	*#joints*
1	2	603	4	1
2	8	1140	11	4
3	20	3590	25	10
4	44	7800	53	22
5	92	16300	109	46
6	188	33300	221	94
7	380	67000	445	190
8	764	135000	893	373
9	1532	271000	1790	766
10	3068	543000	3580	1534

```
mobile = @instantiatedModel(buildModia3D(Mobile, method=2))
```

various ways to generate 3D mechanics code

Time-to-start-simulation



- reading the model
- symbolic transformations
- generation/compilation of code
- start of code

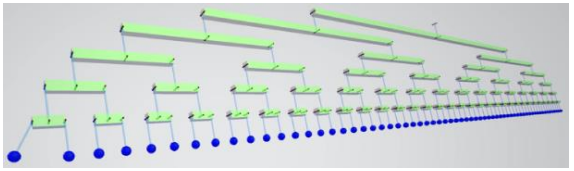
$$\ddot{\boldsymbol{q}} := \boldsymbol{M}(\boldsymbol{q}) \backslash (\boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}}) - \boldsymbol{f}) \qquad \boldsymbol{f} := \boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}})$$

(restriction: No 1D inertias)

depth	#unknowns	Modia method=3	Modia method=2	Commercial Modelica tool	OpenModelica
6	33 300	1.7 s	x 1.3	x 14.0	x 112
7	67 000	3.7 s	x 1.8	x 10.0	x 57
8	135 000	8.8 s	x 2.5	x 7.3	x 90
9	271 000	23.9 s	x 4.9	x 4.9	
10	543 000	73.8 s	x 9.0		

„Time-to-start-simulation“ about 1-2 orders of magnitude faster with Modia as with some Modelica tools because the 3D mechanics Modia code are 3 (Julia) function calls instead of huge number of 3D mechanics equations as in Modelica.

Simulation time



		DAE-solver		*ODE-solver*		
		IDA		**CVODE**		
depth	**#unknowns**	**Modia method=3**	**Modia method=2**	**Modia method=2**	**Commercial Modelica tool**	**OpenModelica**
6	33 300	0.1 s	x 1.3	x 30	x 30	x 240
7	67 000	0.3 s	x 1.6	x 80	x 80	x 110
8	135 000	1.4 s	x 1.5	x 130	x 230	x 120
9	271 000	8.8 s	x 1.7			
10	543 000	57.0 s	x 1.3			

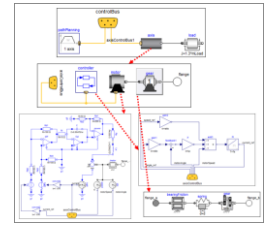
ODE-Solver:
Simulation times **similar**.

DAE-Solver:
Modia simulation time about **100 times faster** as some Modelica tools (either ODE or DAE solver)

Why is Modia with IDA so much faster?
If DAE solver (and not at an event), linear equation system in code is not solved in model but by DAE solver.
For depth = 8:
IDA/CVODE: An leq system of size 764 is built/factorized from time-to-time
CVODE: At every model evaluation, an leq system of size 373 is built/factorized/solved (!)

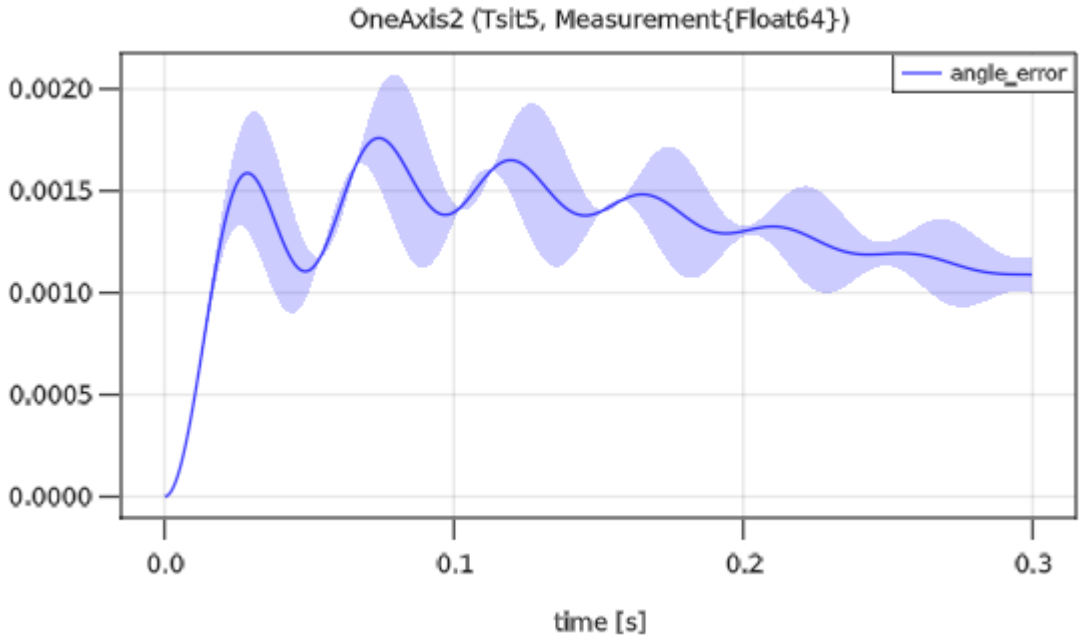
→ *Modelica tools could use the same technique to get a speed-up of 100*

Simulate with uncertainties

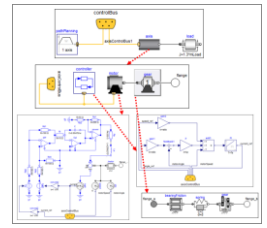


```
using Modia, Measurements

OneAxis2 = OneAxis | Model(
  load = Map(J = 19.5 ± 4.0),
  axis = Map(c = 8.0  ± 0.8),
  angle_error = :(
    axis.axisControlBus.angle_ref - load.phi))

oneAxis2 = @instantiateModel(OneAxis2,
            FloatType = Measurement{Float64})
```

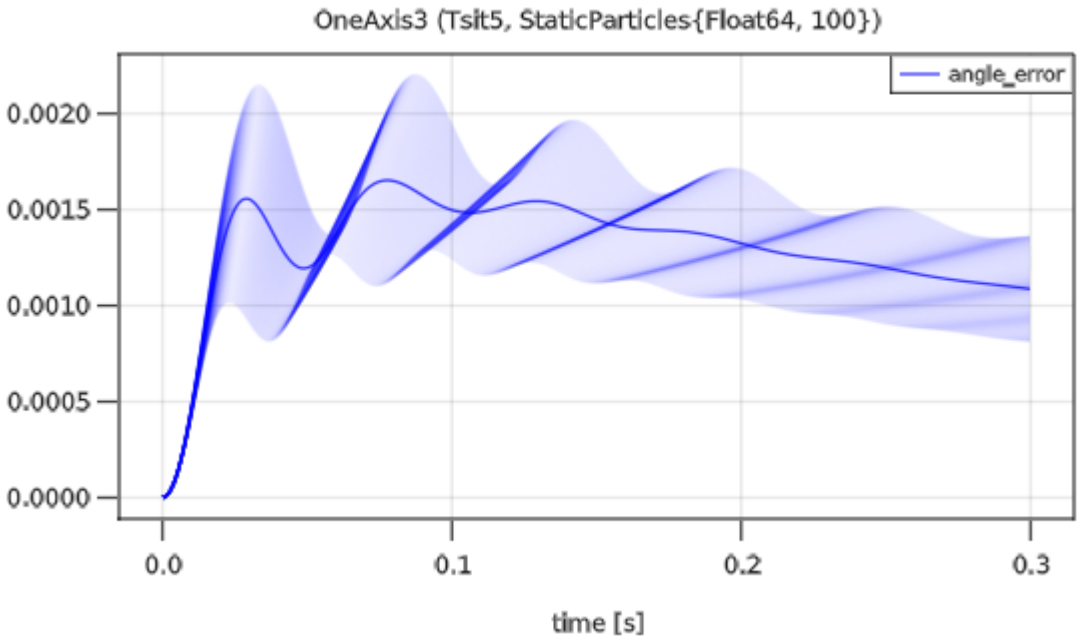
Measurements.jl: Linear error propagation



Monte-Carlo simulation



```
using Modia
using MonteCarloMeasurements, Distributions

uniform(vmin,vmax) = StaticParticles(100,
  Distributions.Uniform(vmin,vmax))

OneAxis3 = OneAxis | Model(
  load = Map(J = uniform(11.5, 27.5)),
  axis = Map(c = uniform(6.6, 9.6)),
  angle_error = :(
    axis.axisControlBus.angle_ref - load.phi)
  )

oneAxis3 = @instantiateModel(OneAxis3,
  FloatType = StaticParticles{Float64,100})
```

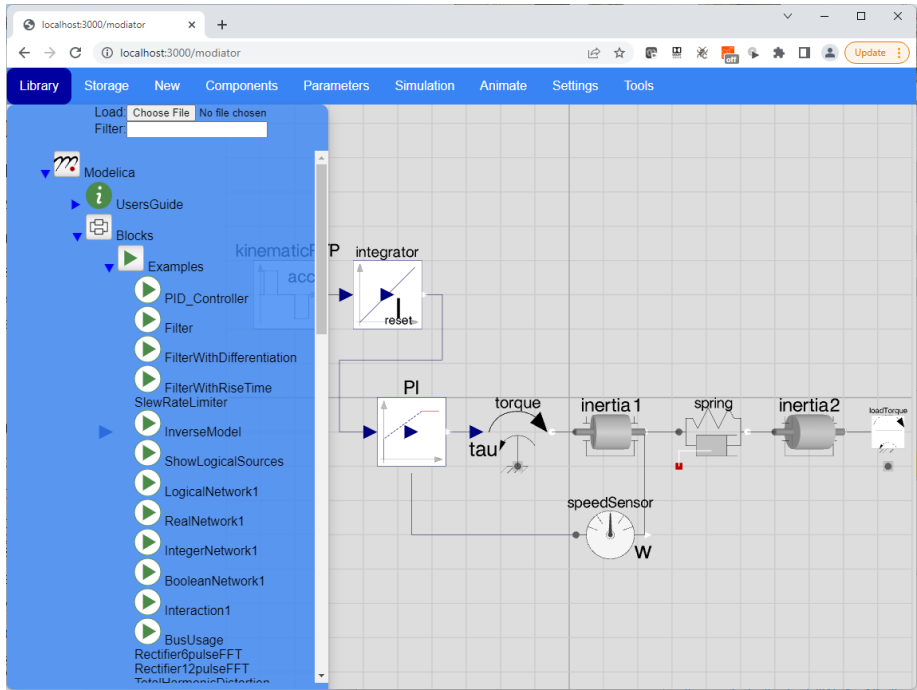


Conclusions

- Shown the power of equation-oriented modeling
- Introduced the needed structural and symbolic algorithms
- Shown the power of object-oriented modeling
- Shown contact handling and speed benefits with domain-specific algorithms
- Laid the foundation for Modelica systems modeling

- You are invited to join the Modia/Modiator projects

Full Modiator (with diagram editor)



Further details

Otter, Elmqvist (2017): *Transformation of Differential Algebraic Array Equations to Index One Form*. Modelica'2017 Conference. Section 4.6.
https://dx.doi.org/10.3384/ecp17132565

For the „incremental cycle detection“ the “Simple algorithm N” from the following reference is used in ModiaBase:

Bender, Fineman, Gilbert, Tarjan (2016):
A New Approach to Incremental Cycle Detection and Related Problems.
ACM Transactions on Algorithms, Volume 12, Issue 2, Feb. 2016.
https://dl.acm.org/citation.cfm?id=2756553

Open source Julia implementation as function **tearEquations!** (..) in ModiaBase

Literature:

Pantelides C.C. (1988): The consistent initialization of differential-algebraic systems. SIAM Journal of Scientific and Statistical Computing, No. 9, S. 213-231.

Mattsson S.E. and Soederlind G. (1992): Index Reduction in Differential-Algebraic Equations Using Dummy Derivatives. SIAM Journal on Scientific Computing. Vol. 14, S. 677-692.

Models

- Yoyo
  - https://www.lehman.edu/faculty/anchordoqui/chapter21.pdf
- Rattleback
  - https://www.diva-portal.org/smash/get/diva2:1138485/FULLTEXT01.pdf
- TippeTop
  - https://downloads.hindawi.com/journals/amp/2021/5552369.pdf
  - https://www.diva-portal.org/smash/get/diva2:602220/FULLTEXT01.pdf
- Drum model
  - http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=830A12C104D0AF3DEFD54A6C971EC0D0?doi=10.1.1.41.8308&rep=rep1&type=pdf