KU LEUVEN

Modelica – Advanced Concepts

Filip Jorissen – KU Leuven

Context

- Modelica Specification:
  "No particular variable needs to be solved for manually. A Modelica tool will have enough information to decide that automatically. Modelica is designed such that available, specialized algorithms can be utilized to enable efficient handling of large models having more than one hundred thousand equations."

Modelica Association, *Modelica – A Unified Object-Oriented Language for Systems Modeling. Language Specification version 3.3*, May 2012

Context

- Building Energy Simulation
  - Slow, linear building dynamics
  - Non-linear HVAC systems
  - Fast, discrete control systems

- Model size
  - 1300 time-depending states
  - > 100k equations
  - Large non-linear algebraic loops
  - Small time constants: ~ 1s

Context

- Modelica Specification:
  "No particular variable needs to be solved for manually. A Modelica tool will have enough information to decide that automatically. Modelica is designed such that available, specialized algorithms can be utilized to enable ~~efficient handling of large models~~ having more than one hundred thousand equations."

Modelica Association, *Modelica – A Unified Object-Oriented Language for Systems Modeling. Language Specification version 3.3*, May 2012

Outline

- Modelica works fine out of the box for small/simple models. However, for more advanced models and for debugging, having some basic solver knowledge is preferable. Otherwise models may fail and/or become slow.

1. How is a Modelica model solved?
2. How can Modelica users exploit this knowledge?
3. Application to large model

How is a Modelica model solved?

KU LEUVEN

Outline

- Given time t, variables $\mathbf{y}(t)$, equations $\mathbf{F}(\mathbf{y},t)$, initial equations $\mathbf{F_0}(\mathbf{y},t)$, initial time t_0

1. Compute $\mathbf{y_0}$ from $\mathbf{F_0}(\mathbf{y_0},t_0) = \mathbf{0}$
2. Set initial values $\mathbf{y} = \mathbf{y_0}$, $t = t_0$
3. Solve $\mathbf{F}(\mathbf{y},t)$
4. Do an integration step
5. Update $\mathbf{y}$ and t
6. Go to 3

Solving model equations

- Modelica simulation models consist of
  - time t
  - n variables **y**(t)
  - m equations **F**(**y**,t)
- Basic requirements:
  - n = m
  - equations are consistent
- Task of Modelica solver: compute values of **y(t)** for multiple time steps t such that the values satisfy **F(y,t)**.
  - Efficiently

Solving model equations

- Two equation types in **F(y,**t**)**
  - Algebraic equation

    ```
    Q_flow = G*dT;
    ```

    - No time derivative
    - Describes the relation between variables within one time step
      - I.e. steady state equations
    - Denoted using vector **z** and equations **H(x,z,t) = 0**
  - Differential equation:

    ```
    C*der(T) = port.Q_flow;
    ```

    - Contains time derivative ‘**der(**$\mathbf{y_i}$**)**’
    - Describes time dynamics of the system
    - Denoted using ‘state’ vector **x**, and equations **der(x) = G(x,z,t)**

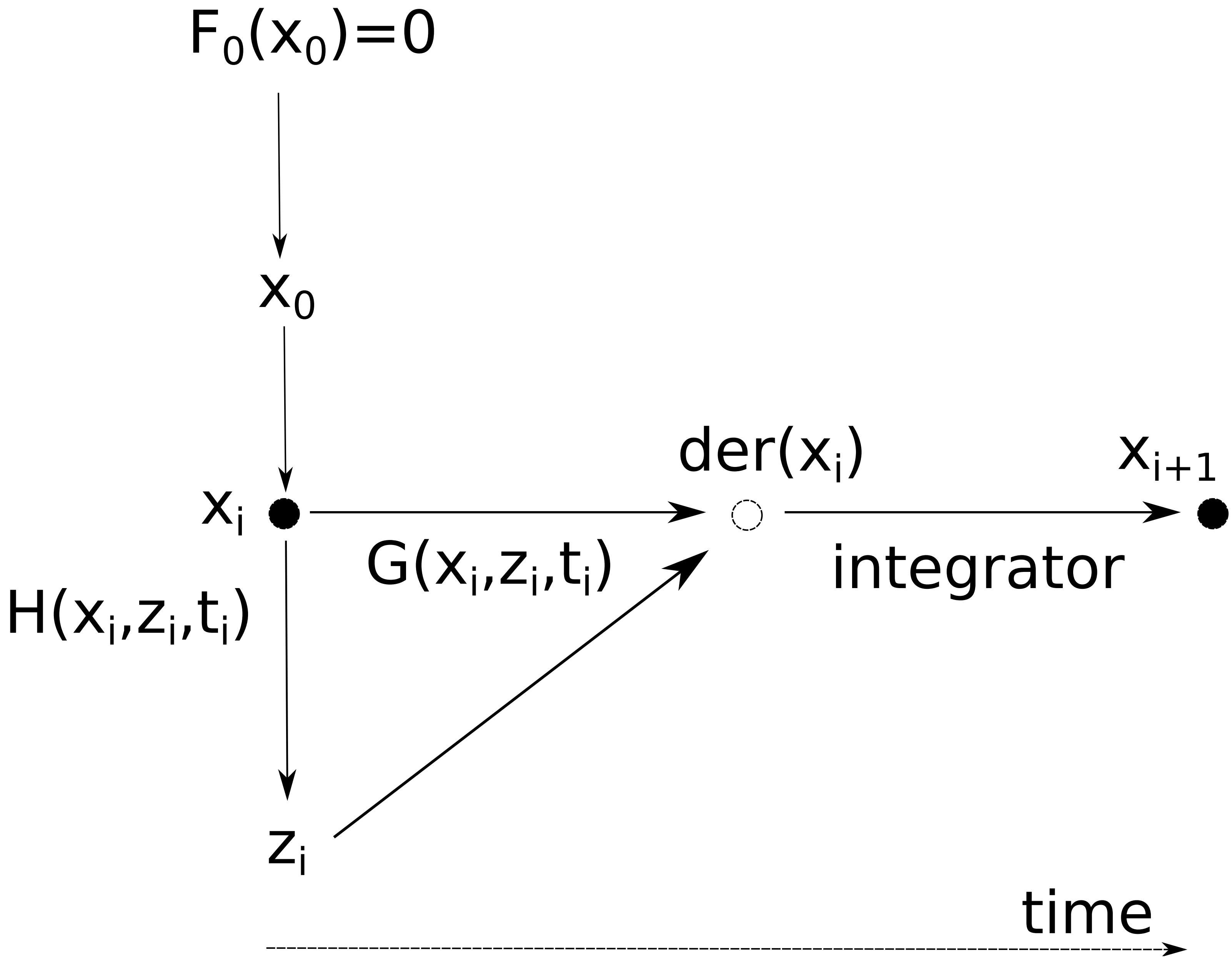
KU LEUVEN

Outline - revised

- Equations:
  - $\mathbf{0 = H(x,z,t)}$
  - $\mathbf{der(x) = G(x,z,t)}$

- Solution algorithm (simplified):
  1. Compute $\mathbf{y_0}$ from $\mathbf{F_0(y_0,}t_0\mathbf{)}$
  2. Set initial values $\mathbf{x = x_0}$, $t = t_0$
  3. Solve **H** towards **z** using known values of **x** and t
  4. Solve **G** towards **der(x)** using known values of **x, z, t**
  5. Compute next **x** and **t** from **der(x)** using time integrator
  6. Go to (3)

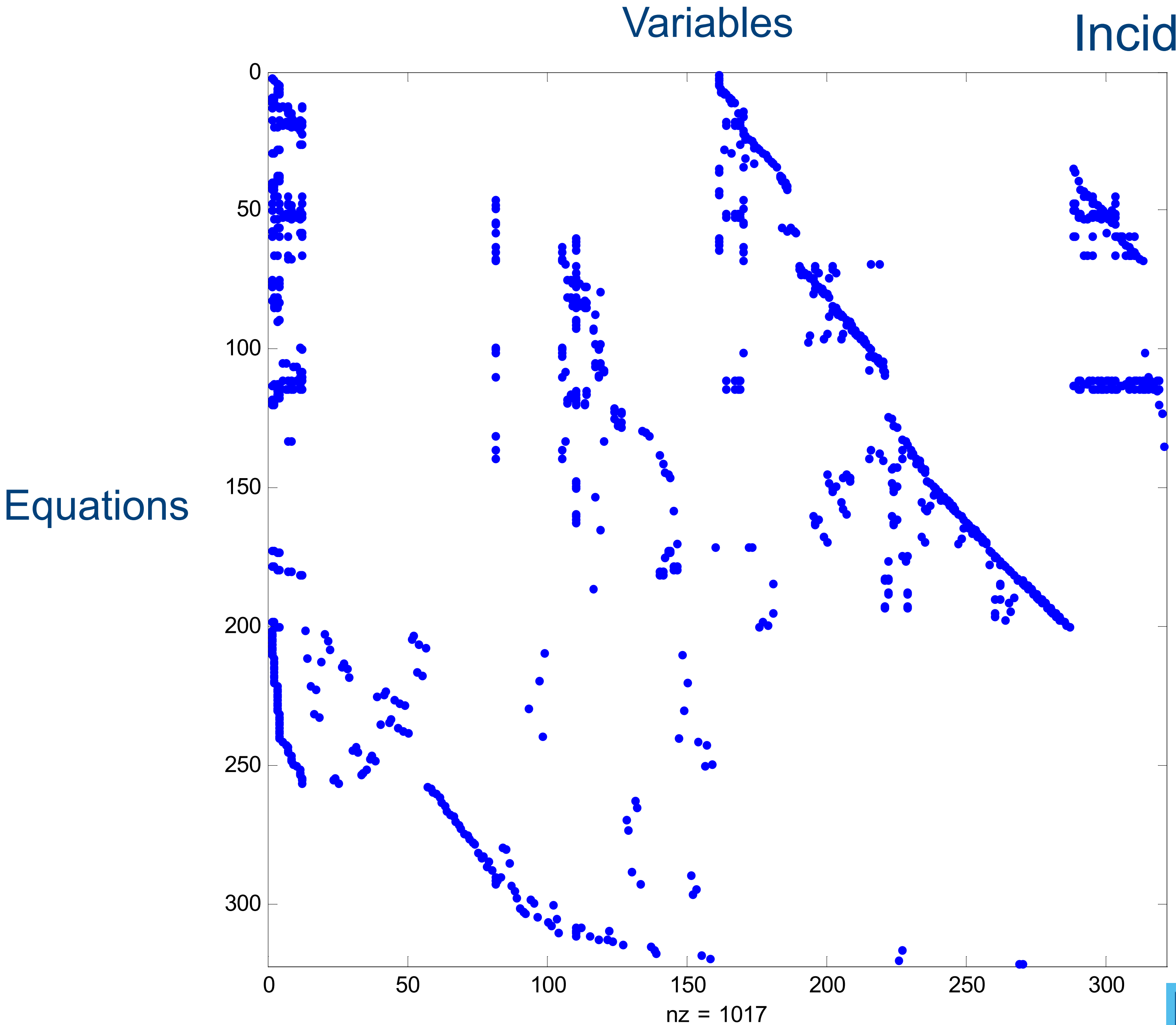
KU LEUVEN


$F_0(x_0)=0$
x_0
x_i
$der(x_i)$
x_{i+1}
$G(x_i,z_i,t_i)$
integrator
$H(x_i,z_i,t_i)$
z_i
time

Solving model equations

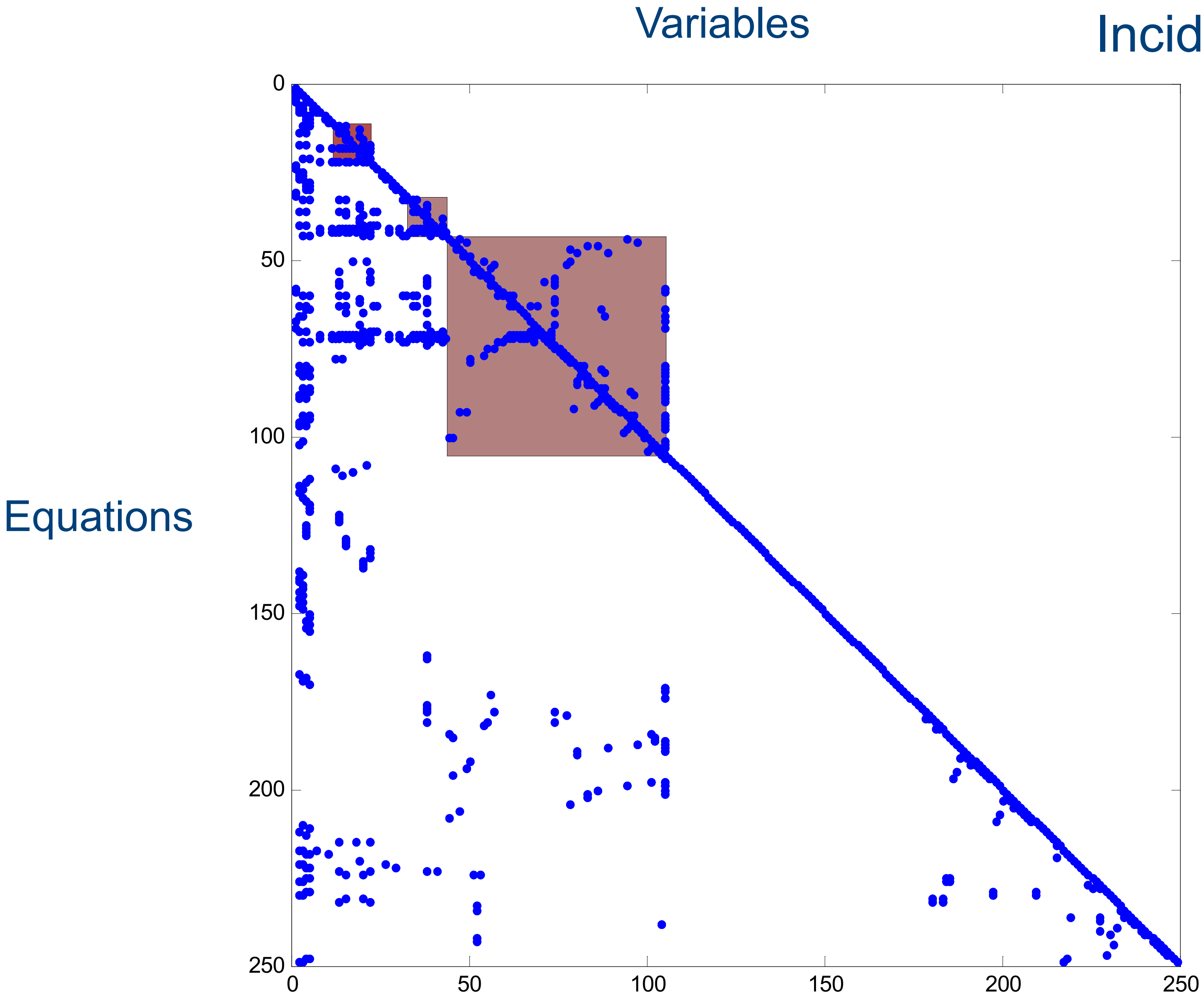
- Solving **F** (**H** and **G)**
  - F is a large system of equations
  - Newton Solver could be used for complete set of equations, but inefficient
  - => Exploit problem structure

Incidence matrix



KU LEUVEN

Incidence matrix



Source: Dymola user manual, Volume 2

Solving model equations

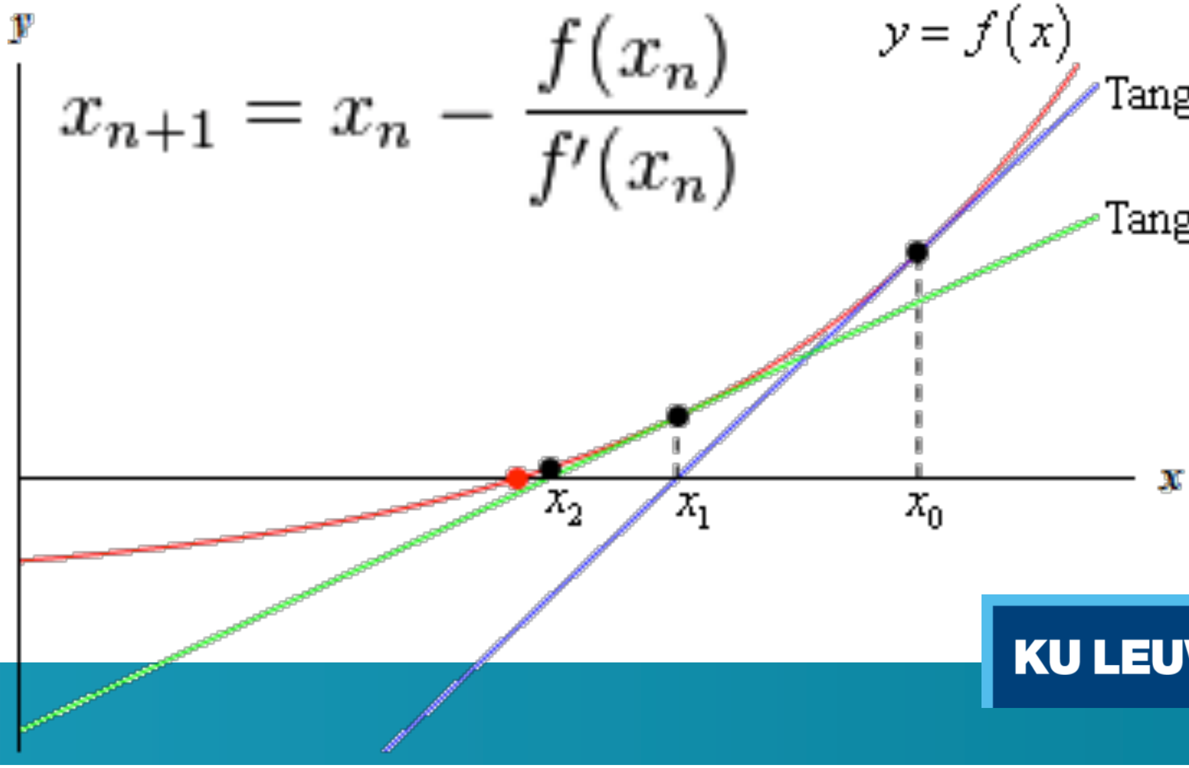
- After reordering and simplification, solving **F** (**H** and **G)** consists of:
  - Alias variables (eliminated)
  - Solve sequential equations (cheap)
  - Solve linear algebraic loops with constant coefficients (analytic solution possible)
  - Solve linear algebraic loops with non-constant coefficients (1 iteration)
  - Solve non-linear algebraic loops (many iterations)
  - Solve mixed algebraic loops (many iterations)
  - …

Solving model equations

- Newton solver:
  - Requires iterations
  - Requires derivative to exist
  - Requires f to be sufficiently smooth
  - etc

Sources: wikipedia, http://tutorial.math.lamar.edu/Classes/CalcI/NewtonsMethod.aspx

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$



KU LEUVEN

Outline - revised

- Equations:
  - $\mathbf{0 = H(x,z,t)}$
  - $\mathbf{der(x) = G(x,z,t)}$

- Solution algorithm (simplified):
  1. Compute $\mathbf{y_0}$ from $\mathbf{F_0(y_0,}t_0\mathbf{)}$
  2. Set initial values $\mathbf{x = x_0}$, $t = t_0$
  3. Solve **H** towards **z** using known values of **x** and t
  4. Solve **G** towards **der(x)** using known values of **x, z, t**
  5. Compute next **x** and **t** from **der(x)** using time integrator
  6. Go to (3)

Time integrator

- Compute $\mathbf{x}_{i+1}$ from $\mathbf{x}_i$ and **der(x)**

- Explicit Euler:
  - Fixed time step Δt
  - $x_{i+1} = x_i + \Delta t * \mathrm{der}(x_i)$
  - Unstable for large Δt
- Implicit Euler
  - Fixed time step Δt
  - $x_{i+1} = x_i + \Delta t * \mathrm{der}(x_{i+1})$
  - Stable for typical problems

Time integrators

- Higher order implicit methods
  - Radau IIa, LSodar, DASSL
  - Polynomial approximations
  - Variable step length such that specified tolerance is attained
  - Often require multiple evaluations of **F()** since multiple support points may be used
  - Implicit method -> requires iterations and therefore multiple evaluations of **F()**
  - Advantage: less steps / larger step size

Time integrators

- Higher order explicit methods
  - Dopri45
    - Variable step

- Dymola default: DASSL
  - Implicit, variables step -> easy to use
  - Fast for small problems
  - Lsodar seems to perform better

Speeding up models

and model robustness

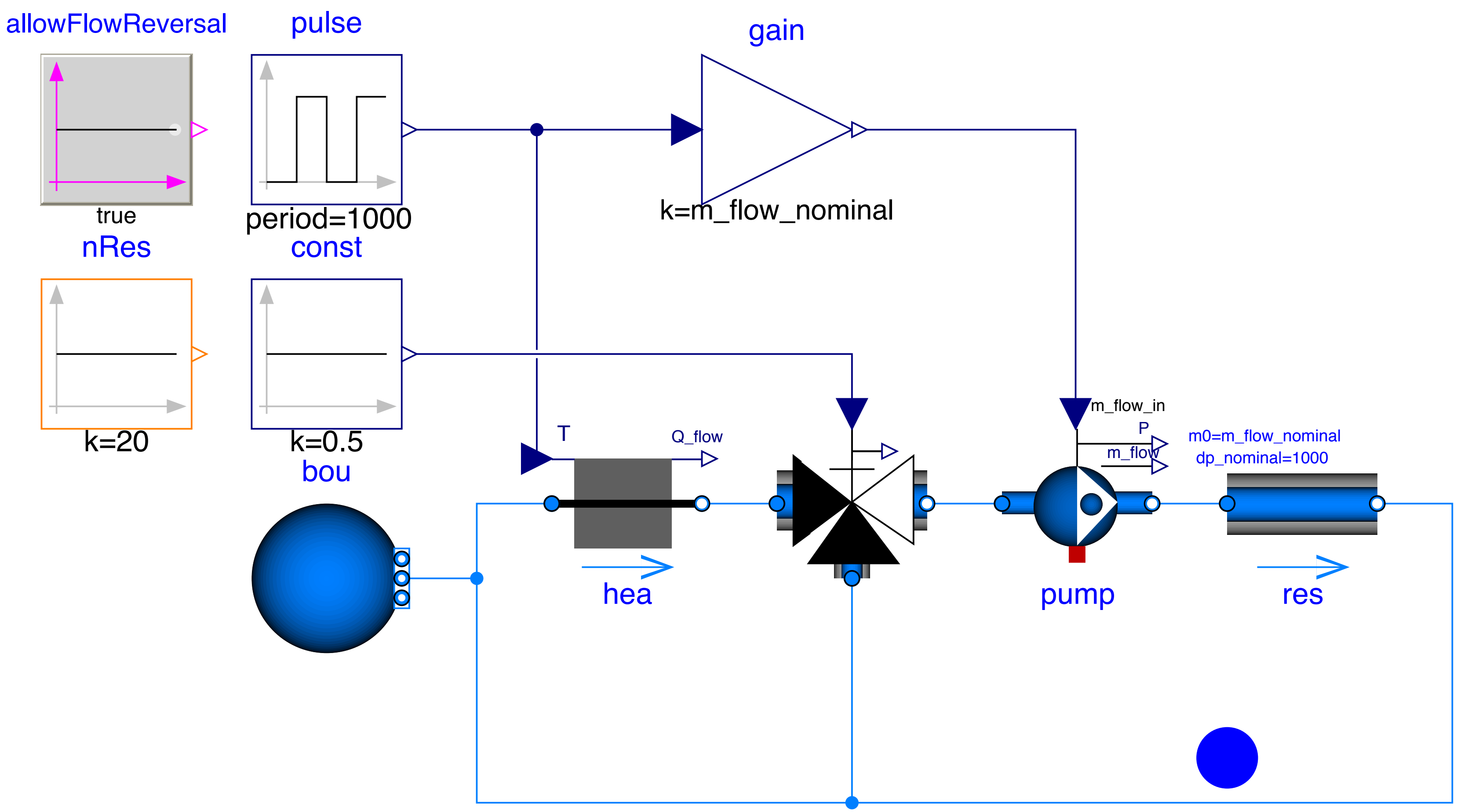
Outline

- Computation time consists of:
  - Time per evaluation of **F()**
    - number of equations
    - algebraic loops
  - Number of evaluations of **F()**
    - integrator choice
    - solver tolerance or fixed step size
  - Overhead for integrator
  - Overhead for storing data

Time per evaluation

- Algebraic loops

Time per evaluation: Algebraic loops



Time per evaluation: Algebraic loops

- For nRes.k = 20:

Sizes nonlinear systems of equations	{6,	21,	46}
Sizes after manipulation	{1,	19,	22}

- Advanced.GenerateBlockTimers = true;

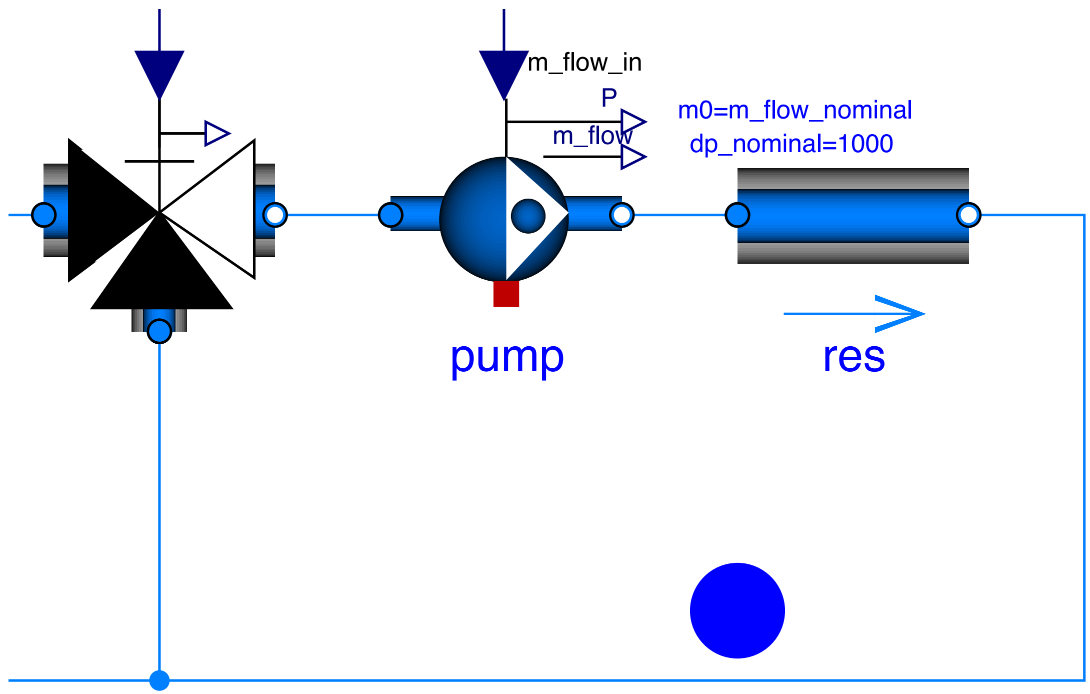
```
Name of block,      Block, CPU[s],
 DynamicsSection:      14, 0.200, ...
   Dynamics 2 eq:      15, 0.000, ...
   Dynamics code:      16, 0.000, ...
   Nonlin sys(1):      17, 0.007, ...
   Dynamics code:      18, 0.000, ...
  Dynamics 20 eq:      19, 0.066, ...
   Dynamics code:      20, 0.002, ...
  Nonlin sys(22):      21, 0.122, ...
   Dynamics code:      22, 0.001, ...
```

This example:
97% of computation time
spent solving algebraic loops

KU LEUVEN

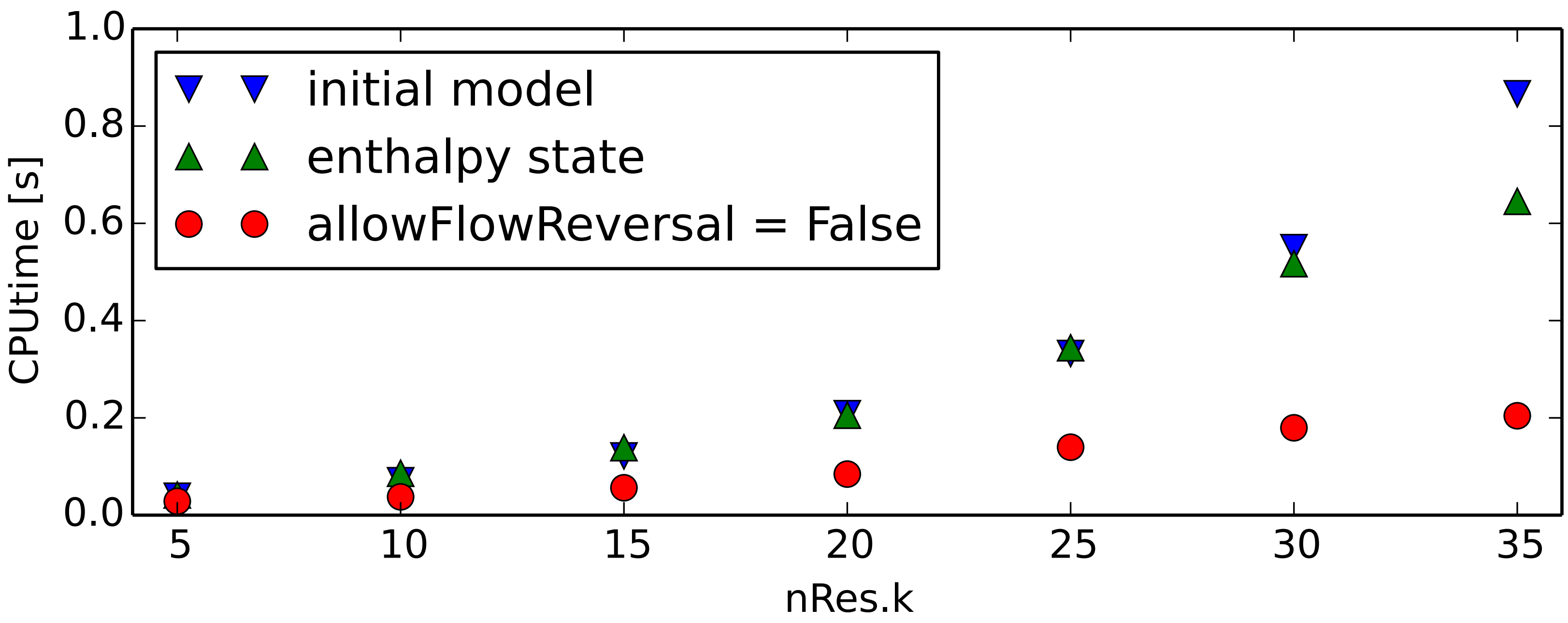
Time per evaluation: Algebraic loops

- Algebraic loop solving for enthalpy
  - Add states
  - allowFlowReversal = false



Time per evaluation: Algebraic loops

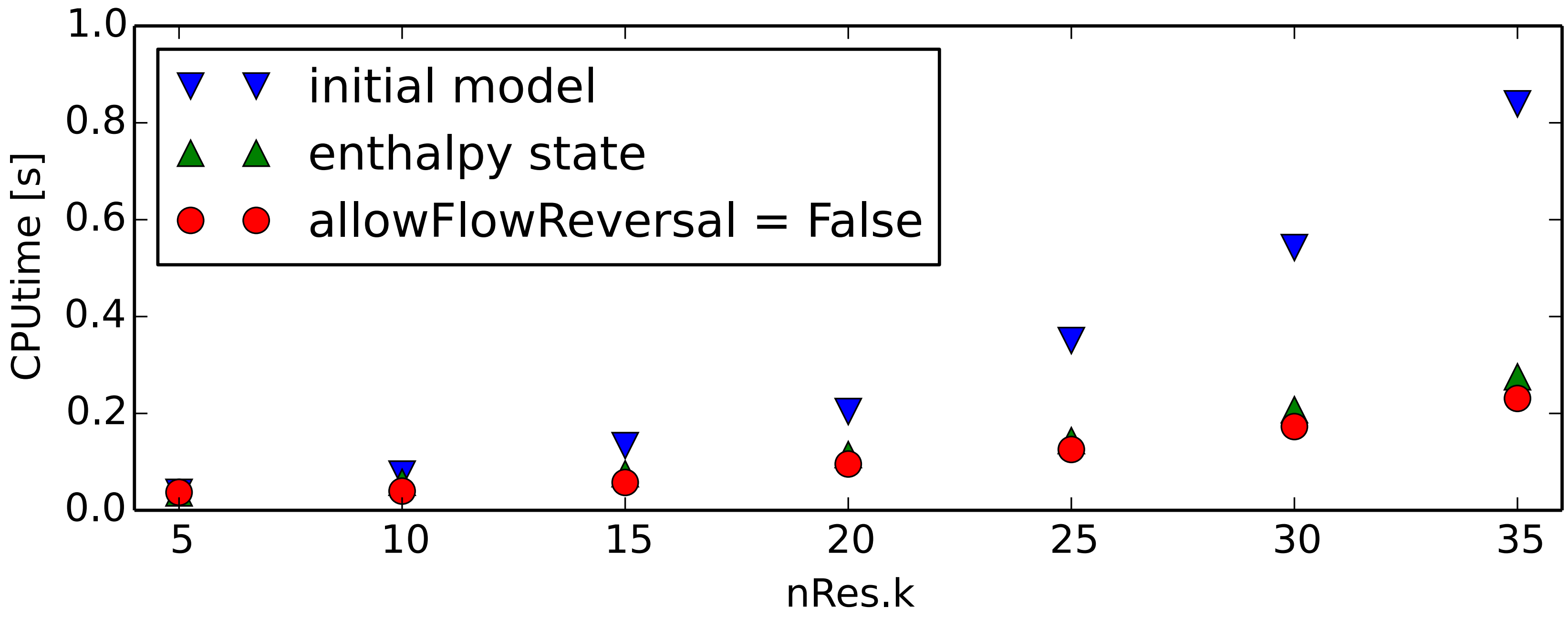
- Algebraic loop solving for enthalpy
  - Add states
  - allowFlowReversal = false



(a) numeric Jacobian

Time per evaluation: Algebraic loops

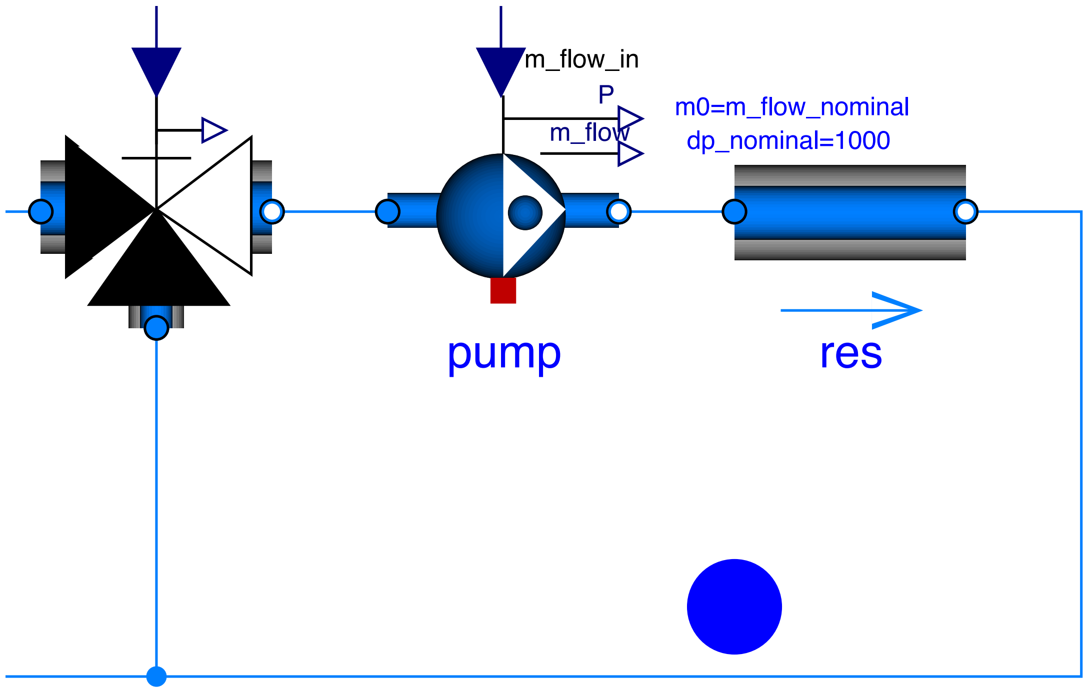
- Algebraic loop solving for enthalpy
  - Add states
  - allowFlowReversal = false



(b) analytic Jacobian

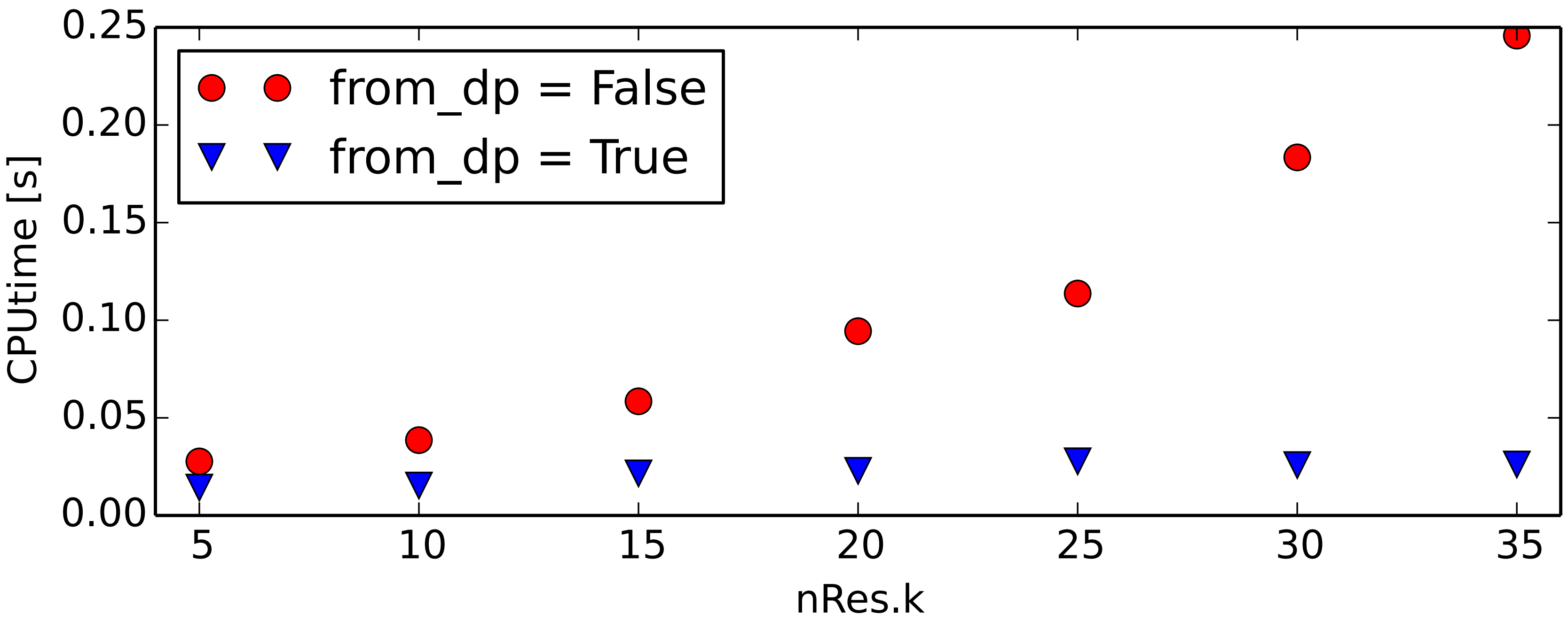
Time per evaluation: Algebraic loops

- Algebraic loop solving for <u>mass flow rate / pressure</u>



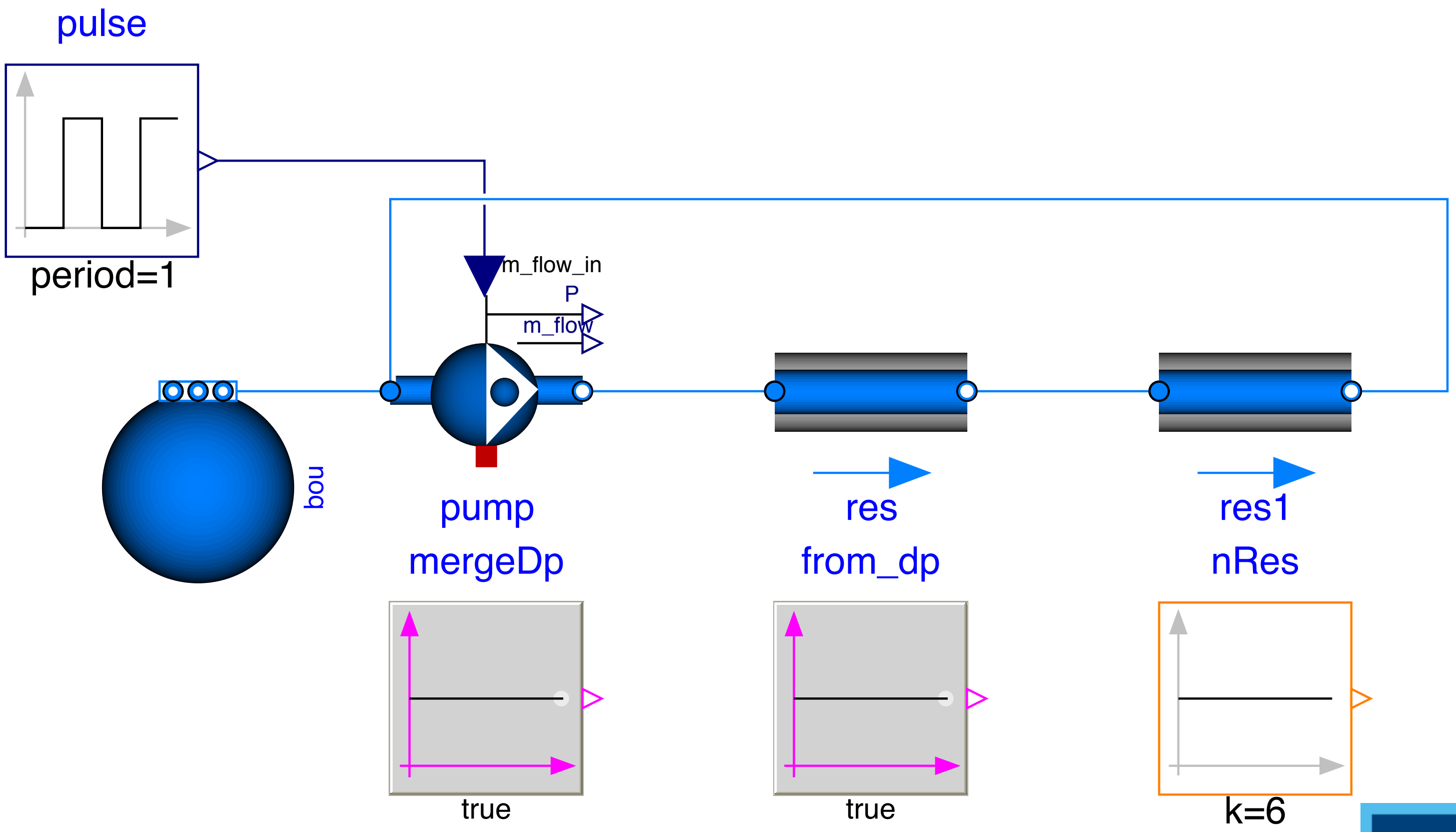
Time per evaluation: Algebraic loops

- Algebraic loop solving for <u>mass flow rate / pressure</u>



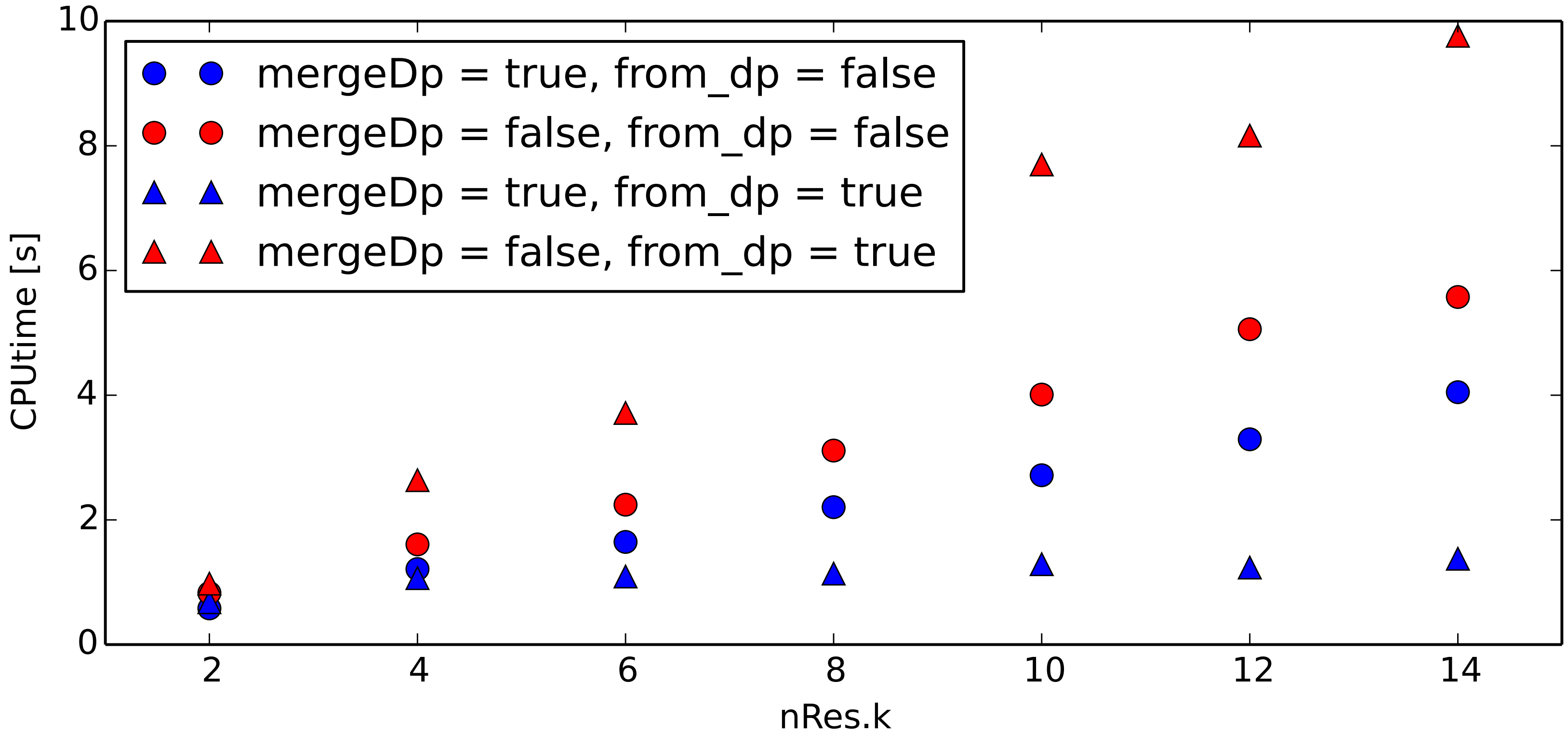
Time per evaluation: Algebraic loops

- Algebraic loop solving for mass flow rate / pressure
  - Parallel and series connections



Time per evaluation: Algebraic loops

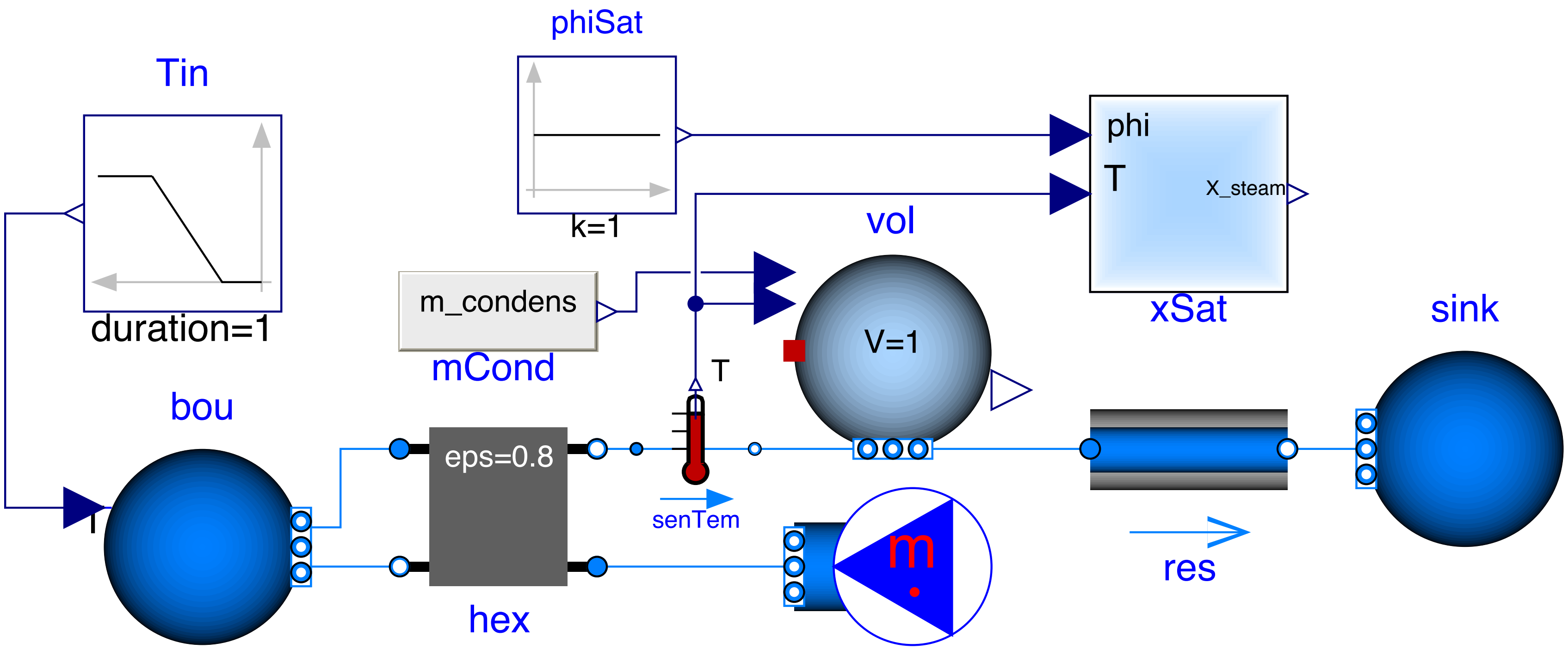
- Algebraic loop solving for mass flow rate / pressure
  - Parallel and series connections



KU LEUVEN

Time per evaluation: Algebraic loops

- Avoiding algebraic loops



Time per evaluation: Inefficient code

- Obsolete variables
- Inlining functions
- Evaluating model parameters
- Duplicate code
- Parameter divisions

- See paper for practical examples:
  - Jorissen, F., Wetter, M., & Helsen, L. (2015). Simulation Speed Analysis and Improvements of Modelica Models for Building Energy Simulation. In 11th International Modelica Conference (pp. 59–69). Paris, France. http://doi.org/10.3384/ecp1511859

Number of evaluations

Number of evaluations

- What determines number of evaluations of **F()**?
  - Integrator tolerance determines step size of integrator
  - Fast dynamics require a smaller step size before the tolerance criterion is met
  - Badly tuned PID controller can lead to excitation of short time scales
  - Number of events
  - Jacobian computation!

Application to large building model

Application to large building model

- Large models: previously illustrated examples can be applied

- Example:

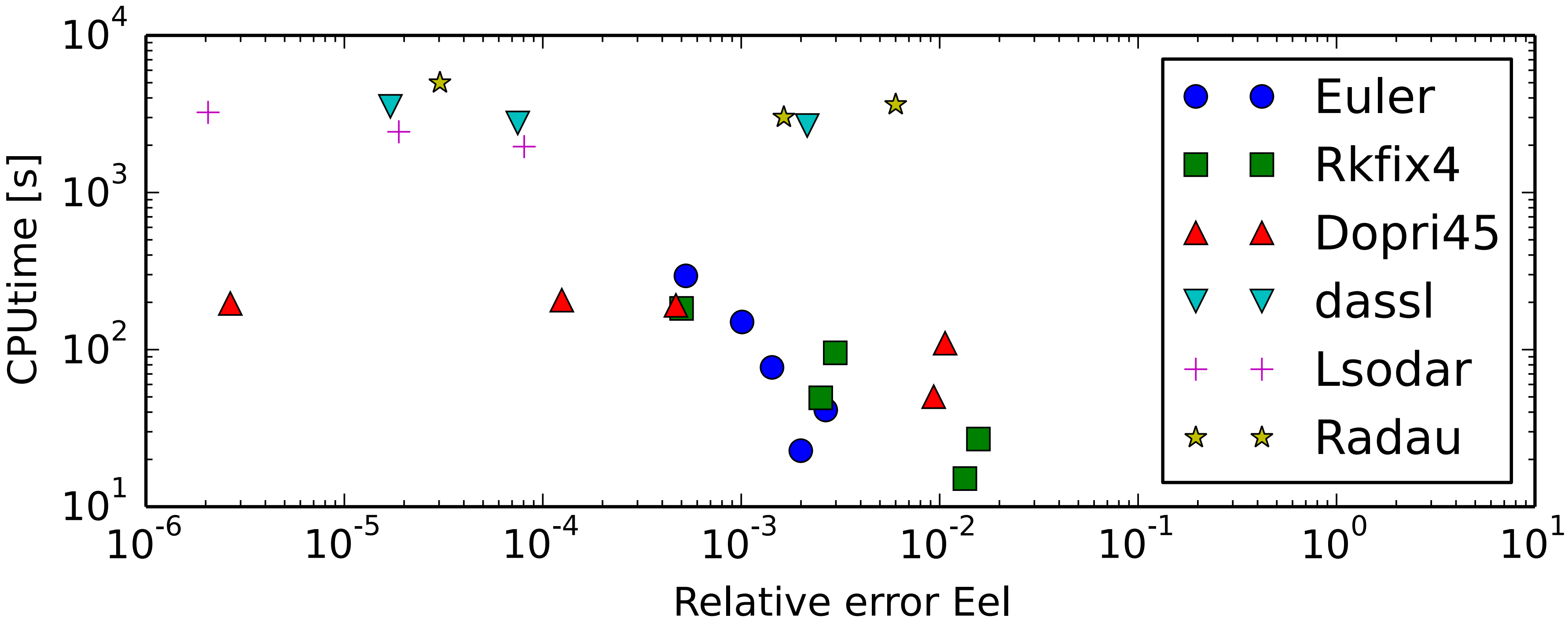
Application to large building model

- A second large gain can be obtained by adapting the model to work with explicit integrators
    1. Remove all fast time constants
    2. Use explicit Euler integration

KU LEUVEN

Application to large building model

- Time constants > 30 s
  - Euler integration 100 times faster than DASSL



KU LEUVEN

Conclusion

- Detailed solver and model analysis has led to 4000 times faster simulations in example case

- These speed improvements were obtained through:
  - Individual model changes (inlining functions, etc)
  - Reconfiguration of groups of models (avoiding algebraic loops, etc)
  - Design decisions for global model (time constant / integrator choice)

- Modelica hides solver complexity from users, but this leads to unexploited speed optimization potential and may cause the solver to fail

KU LEUVEN