Lecture 3

Simulation of differential-algebraic equations

Erik Frisk
erik.frisk@liu.se

Department of Electrical Engineering
Linköping University

February 23, 2022

Outline

BDF solver

- DASSL is an *implementation/code* to solve systems of differential-algebraic equations of index 0 or 1
- SUNDIALS is another free, more modern, implementation. Written in C and not Fortran.
  - Fairly straightforward (at least on Linux and Mac) to install on your computer if you are used to compiling your own software.
  - Installed on charger.ad.liu.se
- Why am I talking about DASSL?
- Basic principle, one-step and multiple step
- A little on what it is good at
- Some principles on details

Basic principle

For a DAE in the form

$$\begin{aligned} F(t, y, y') &= 0 \\ y(t_0) &= y_0 \\ y'(t_0) &= y'_0 \end{aligned}$$

the DASSL basic principle is not unique, replace the derivative for a difference approximation and solve the resulting system of equations with a Newton method.
Properties of the DASSL implementation:

- Variable step-length
- Variable order
- Efficient implementation, for example measured in the number of evaluations of the function F.

DASSL - first order step

Replace the derivative with a first order BDF, then

$$F(t_{n+1}, y_{n+1}, (y_{n+1} - y_n)/h_{n+1}) = 0$$

A direct Newton method to solve for y_{n+1} then becomes

$$y_{n+1}^{(m+1)} = y_{n+1}^{(m)} - \left(F_y + \frac{1}{h_{n+1}} F_{y'}\right)^{-1} F(t_{n+1}, y_{n+1}^{(m)}, (y_{n+1}^{(m)} - y_n)/h_{n+1})$$

where F_y and $F_{y'}$ is evaluated in $y_{n+1}^{(m)}$.

With a good stop condition on the iterations so this is approximately how it works. I will now briefly look at:

- higher order algorithm
- strategies for choice of order, step-length
- stop conditions for the Newton iterations
- $F_y + \alpha F_{y'}$ – iteration matrix (sometimes referred to as the jacobian)

DASSL - a step of order k

1. Compute a prediction polynomial $\omega^P(t)$ based on k previous time-points
2. Compute a correction polynomial $\omega^C(t)$, equal $\omega^P(t)$ on k equidistant (h_{n+1}) previous time-steps
3. Predict next time-step using prediction polynomial

DASSL - a step of order k, cont'd.

There are y_{n-i} that approximates $y(t_{n-i})$ for $i = 0, \ldots, k$. Wanted: y_{n+1}.

Prediction polynomial - $\omega_{n+1}^P(t)$, order k

$$\omega_{n+1}^P(t_{n-i}) = y_{n-i}, \ i = 0, \ldots, k$$

Prediction for y_{n+1} and y'_{n+1} is then given by

$$y_{n+1}^{(0)} = \omega_{n+1}^P(t_{n+1})$$
$$y'^{(0)}_{n+1} = \omega'^P_{n+1}(t_{n+1})$$

Correction polynomial - $\omega_{n+1}^C(t)$, order k

$$\omega_{n+1}^C(t_{n+1} - ih_{n+1}) = \omega_{n+1}^P(t_{n+1} - ih_{n+1}), \ i = 1, \ldots, k$$
$$F(t_{n+1}, \omega_{n+1}^C(t_{n+1}), \omega'^C_{n+1}(t_{n+1})) = 0$$

i.e. $y_{n+1} = \omega_{n+1}^C(t_{n+1})$

Note: fixed vs. variable step length.

DASSL - a step of order k, cont.

Skips a lot of notation and rather straightforward algebra,

Equations to be solved

$$\omega_{n+1}^C(t_{n+1} - ih_{n+1}) = \omega_{n+1}^P(t_{n+1} - ih_{n+1}), \ i = 1, \ldots, k$$
$$F(t_{n+1}, \omega_{n+1}^C(t_{n+1}), \omega'^C_{n+1}(t_{n+1})) = 0$$

can be written as the solution to the nonlinear equation

$$F(t_{n+1}, y_{n+1}, \alpha y_{n+1} + \beta) = 0$$

i.e. the same form that we had in the one step case (not so surprising)

$$F(t_{n+1}, y_{n+1}, (y_{n+1} - y_n)/h_{n+1}) = 0$$

but the constants α and β depends on the step length h_{n+1}, order k and y_{n-i} in a non-trivial way. Exactly how is not of main importance here.

DASSL - a step of order k, cont.

Now solve

$$F(t, y, \alpha y + \beta) = 0$$

by

$$y^{(m+1)} = y^{(m)} - c \underbrace{(\alpha F_{y'} + F_y)}_{\text{iteration matrix } G}{}^{-1} F(t, y^{(m)}, \alpha y^{(m)} + \beta)$$

Each iteration is solved by $G = LU$ factorization. With $\delta^{(m)} = y^{(m+1)} - y^{(m)}$ and $r^{(m)} = cF$ then

$$Ls^{(m)} = r^{(m)}$$
$$U\delta^{(m)} = s^{(m)}$$

The iteration matrix, strategies

To make an iteration, compute G and LU factorize.

$$y^{(m+1)} = y^{(m)} - cG^{-1}F(t, y^{(m)}, \alpha y^{(m)} + \beta)$$

Why is this important? Mostly to state that computation of G and the corresponding LU factorization is a main part (even dominant for large models) of the computational burden for one integration step.

If $F_{y'}$ and F_y varies slowly over the solution, reuse old already computed $\hat{G}$. As long as $\hat{G}$ is sufficiently close to G we can expect on convergence.

Supervise convergence speed and compute a new G only when convergence speed is not satisfactorily.

DASSL - stop conditions in Newton iterations

An important aspect is when to stop the Newton iterations. To directly look at $\|y^{(m+1)} - y^{(m)}\|$ is not always a good idea. Instead, define

$$d^{(m)} = \frac{\|y^{(m+1)} - y^*\|}{\|y^{(m)} - y^*\|}$$

and assume that we know an upper bound on $d^{(m)} < \rho$, direct usage of the triangle inequality gives a stop condition

$$\|y^{(m+1)} - y^*\| \leq \frac{\rho}{1-\rho}\|y^{(m+1)} - y^{(m)}\|$$

Use, for example, the largest

$$\frac{\|y^{(m+1)} - y^{(m)}\|}{\|y^{(m)} - y^{(m-1)}\|}$$

observed so far as an estimate on ρ.

In DASSL, max 4 iterations! If $\rho > 0.9$, compute a new G. If the iteration still not converges (4 steps), reduce step length by 1/4. After 10 tries, abort.

DASSL - order selection and step length

Order selection

Estimates, through some clever algebra, error of $y_{n+1} - y(t_{n+1})$ as a function of different orders

c_1 = error term of order $k-1$

c_2 = error term of order k

c_3 = error term of order $k+1$

c_4 = error term of order $k+2$

The basic principle is that $c_1 > c_2 > c_3 > c_4$. Otherwise, lower the order to be more secure.

Step length

After a new order is chosen, estimate the error as if the latest k steps are taken with the same step length such that the error estimate fulfills the tolerance condition

$$ERR = M\|y_{n+1} - y_{n+1}^{(0)}\| \leq 1$$

Automatic Differentiation

An analytical expression for the iteration matrix/system jacobian

$$G = F_y + \alpha\, F_{y'}$$

is good for efficiency and accuracy.

How do we get G?

- Encode not only F but also G and send to DASSL.
- Difference approximations of the derivative.

Automatic Differentiation/Algorithm differentiation

- Automatically separate function F into elementary operations.
- The derivative by direct application of the chain rule
- C/C++/Python/Fortran/Julia/Java/..., mature field with many tools
- Some solvers has direct support for automatic differentiation

SUNDIALS

SUite of Nonlinear and DIfferential/ALgebraic Equation Solvers

Software library consisting of 6 different solvers, written in C.
`https://computing.llnl.gov/casc/sundials`

- CVODE(S)
  Solves IVP for ordinary differential equation (ODE) systems. Includes sensitivity analysis capabilities (forward and adjoint).
- IDA(S)
  Solves IVP for differential-algebraic equation (DAE) systems. Includes sensitivity analysis capabilities (forward and adjoint).
- ARKode
  Solves IVP ODE problems with additive Runge-Kutta methods, including support for IMEX methods.
- KINSOL
  solves nonlinear algebraic systems.

Lots of functionality not available in vanilla Matlab/Python/Julia.

SUNDIALS

- Written in C with interfaces to Fortran (77 and 2003)
- Designed to be incorporated into existing codes
- Nonlinear and linear solvers and all data use is fully encapsulated from the integrators and can be user-supplied
- Parallelism is directly supported
- Support for GPU computations
- Through the ECP, developing a rich infrastructure of support on exascale systems and applications
- Freely available; released under the BSD 3-Clause license (>27,000 downloads in 2019)

See `https://computing.llnl.gov/sites/default/files/SUNDIALS_ECP_Tutorial_2020_Final_0.pdf`
for a fairly recent presentation of Sundials and its capabilities.

Sundials - code and documentation

If you are interested in more details, and how the code is organized and works, see the documentation

`https://sundials.readthedocs.io/en/latest/idas/`

- Fairly easy to install on Linux/Mac (haven't tried on Windows).
- Installed on charger.ad.liu.se if you want to try.
- Included a, non mandatory, exercise for you to make some first simulations.

Outline

Sensitivity analysis

Motivation

Parameters in models may not be known accurately or need to be determined

- sensitivity to changes
- optimization of variables
- parameter identification based on measurement data

We will revisit this problem, already covered in the ODE module, for

$$F(\dot{x}, x, t, p) = 0$$

and you'll see that the sensitivity equations are similar to the ODE case.

Instead we will spend some time we have a performance measure

$$G = \int_0^T g(x, p, t)\, dt$$

and a high-dimensional parameter p.

Sensitivity analysis - forward solution

From the ODE part of the course we know how to compute sensitivity of the solution $x(t, p)$ with respect to parameters p

$$x_p = \frac{dx}{dp}$$

for the ODE

$$\dot{x} = f(t, x, p), \quad x(0) = x_0(p)$$

as the solution to the linear ODE

$$\dot{x}_p = f_x x_p + f_p, \quad x_p(0) = \frac{dx_0(p)}{dp}$$

The corresponding equation for a DAE

$$F(t, \dot{x}, x, p) = 0, \quad x(0) = x_0(p)$$

is derived in the same way (you will do it in an exercise).

Complexity of sensitivity analysis

What happens if the number of parameters is large? We need to solve

$$\dot{x} = f(t, x, p), \quad x(0) = x_0(p)$$

$$\dot{x}_p = f_x x_p + f_p, \quad x_p(0) = \frac{dx_0(p)}{dp}$$

Obtaining forward sensitivities with respect to m parameters is roughly equivalent to solving an DAE system of size $n + m \times n$.

For example, $n = 10$ states and $m = 20$ parameters leads to 210 states in the sensitivity computation.

What if we are interested in the sensitivity of a scalar, with respect to a large number of parameters

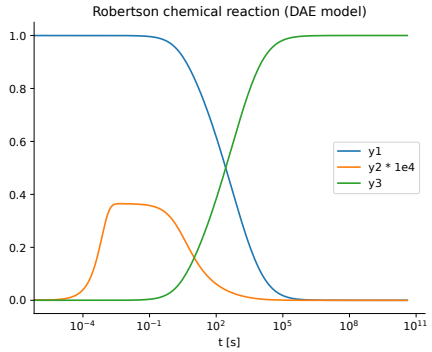
$$G(x, p) = \int_0^T g(x, t, p)\, dt = \int_0^T g_x x_p + g_p\, dt$$

(compare backpropagation in neural networks)

Q: Is there a way to avoid computing all those sensitivities in x_p?

Robertson chemical reaction model – common test case

$$\begin{aligned} \dot{y}_1 + p_1 y_1 - p_2 y_2 y_3 &= 0 \\ \dot{y}_2 - p_1 y_1 + p_2 y_2 y_3 + p_3 y_2^2 &= 0 \\ y_1 + y_2 + y_3 - 1 &= 0 \end{aligned}$$



with the quadrature

$$G = \int_0^T y_3(t)\, dt$$

Then, the question how to efficiently compute

$$dG/dp = \begin{pmatrix} 1.484 \cdot 10^6 & -5.932 & 9.899 \cdot 10^{-4} \end{pmatrix}$$

without computing the 9 sensitivities

$$\frac{dy_i}{dp_j}, \quad i \in \{1,2,3\}, j \in \{1,2,3\}$$

Adjoint sensitivity analysis

Our problem today: Assume the low-index DAE

$$F(t, \dot{x}, x, p) = 0, \quad x(0) = x_0(p)$$

and we want to compute the sensitivity dG/dp of the objective function

$$G(x, p) = \int_0^T g(x, t, p)\, dt$$

Then

$$\frac{d}{dp} G(x, p) = \int_0^T g_x x_p + g_p\, dt$$

But, as mentioned the x_p scales badly ($n \times m$) with size of p and the system order. How to compute this without going through x_p?

Adjoint sensitivity analysis

Introduce lagrange multipliers λ as

$$I(x, p) = G(x, p) - \int_0^T \lambda^T F(x, \dot{x}, t, p)\, dt$$

Since $F(x, \dot{x}, t, p) = 0$

$$\frac{d}{dp} G(x, p) = \frac{d}{dp} I(x, p) = \int_0^T g_x x_p + g_p - \lambda^T (F_{\dot{x}} \dot{x}_p + F_x x_p + F_p)\, dt$$

The problem is, as stated before, x_p.

Reminder: Integration by parts

$$\int_0^T f(t) g'(t)\, dt = f(t) g(t) \big|_{t=0}^T - \int_0^T f'(t) g(t)\, dt$$

Now, first look at the term with $\dot{x}_p$:

$$\int_0^T \lambda^T F_{\dot{x}} \dot{x}_p\, dt = \lambda^T F_{\dot{x}} x_p \Big|_{t=0}^T - \int_0^T \frac{d}{dt} \left(\lambda^T F_{\dot{x}} \right) x_p\, dt$$

Adjoint sensitivity analysis

Collect all integrands with x_p in a separate term. Can we get rid of those?

$$\frac{d}{dp} G(x, p) = \int_0^T (g_p - \lambda^T F_p)\, dt - \int_0^T [-g_x + \lambda^T F_x - \frac{d}{dt} \left(\lambda^T F_{\dot{x}} \right)] x_p\, dt - \lambda^T F_{\dot{x}} x_p \Big|_0^T$$

The function λ is free to choose, why not choose λ as the solution to the adjoint differential equation

$$\frac{d}{dt} \left(\lambda^T F_{\dot{x}} \right) - \lambda^T F_x + g_x = 0$$

- Linear, time-varying!
- Has size n!!

Adjoint sensitivity

We have

$$\frac{d}{dp}G(x,p) = \int_0^T (g_p - \lambda^T F_p)\,dt - \lambda^T(T)F_{\dot{x}}(T)x_p(T) + \lambda^T(0)F_{\dot{x}}(0)x_p(0)$$

with

$$\frac{d}{dt}\left(\lambda^T F_{\dot{x}}\right) - \lambda^T F_x + g_x = 0$$

What about the boundary values? Involves $x_p(0)$: easy, we know that one. But $x_p(T)$ is not easy and it was exactly to *avoid* computing x_p we did the adjoint approach.

A solution: Since any adjoint solution λ is ok, choose $\lambda(T)$ such that

$$\lambda^T(T)F_{\dot{x}}(T) = 0 \tag{1}$$

and solve the adjoint sensitivity equations *backwards* in time.

Note: (1) only works in general for low-index problems, for high index models you need to dig deeper.

Adjoint sensitivity – summary

1) do a forward sweep and solve the low-index DAE

$$F(\dot{x}, x, t, p) = 0, \quad p = p_0$$

Solution is typically efficiently stored using checkpointing.

2), solve the adjoint DAE

$$\frac{d}{dt}\left(\lambda^T F_{\dot{x}}\right) - \lambda^T F_x + g_x = 0, \quad \lambda^T(T)F_{\dot{x}} = 0$$

backwards (from T)

3) In the integration, compute the sensitivity by the quadrature

$$\frac{d}{dp}G(x,p) = \int_0^T (g_p - \lambda^T F_p)\,dt + \lambda^T(0)F_{\dot{x}}(0)x_p(0)$$

Adjoint sensitivity

Some comments of the adjoint DAE

$$\frac{d}{dt}\left(\lambda^T F_{\dot{x}}\right) - \lambda^T F_x + g_x = 0, \quad \lambda^T(T)F_{\dot{x}} = 0$$

- What about index – same (high index problems more difficult)
- What about stability of the adjoint system?
  (original problem stable, then adjoint is also stable)
- How to efficiently and accurately include the forward solution $x(t)$, checkpointing

Cao, Yang, et al. "*Adjoint sensitivity analysis for differential-algebraic equations: The adjoint DAE system and its numerical solution.*" SIAM journal on scientific computing 24.3 (2003): 1076-1089.

Recommendation: Section 1, 2.1, 4.1-4.2 in a first reading.

Repetition: Solving a differential equation backwards

Consider the DAE with an end value condition

$$F(\dot{x}(t), x(t), t) = 0, \quad x(T) = x_T,\ \dot{x}(T) = \dot{x}_T$$

To solve backwards, define

$$y(\tau) = x(T - \tau), \quad t = T - \tau$$

Then,

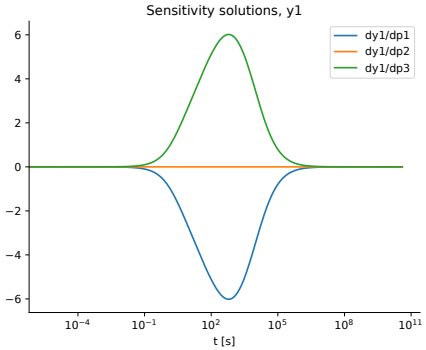
$$\dot{x}(t) = -\dot{y}(T - t) = -\dot{y}(\tau)$$
$$x(t) = y(T - t) = y(\tau)$$

and the DAE in reverse time variables becomes

$$F(-\dot{y}(\tau),\ y(\tau),\ T - \tau) = 0, \quad y(0) = x_T,\ \dot{y}(0) = \dot{x}_T$$

Return to the Robertson chemical reaction model

$$\dot{y}_1 + p_1 y_1 - p_2 y_2 y_3 = 0$$
$$\dot{y}_2 - p_1 y_1 + p_2 y_2 y_3 + p_3 y_2^2 = 0$$
$$y_1 + y_2 + y_3 - 1 = 0$$



with the quadrature

$$G = \int_0^T y_3(t)\, dt$$

If we compute

$$dG/dp = \begin{pmatrix} 1.484 \cdot 10^6 & -5.932 & 9.899 \cdot 10^{-4} \end{pmatrix}$$

using both forward and adjoint techniques, we get the same result.

Outline

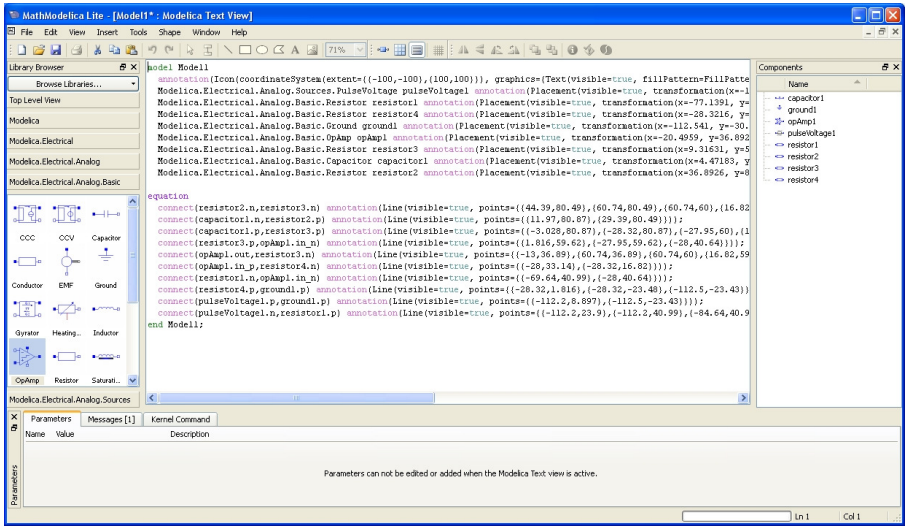
Modelica

Modelica

Modelica is a language for specifying mathematical models for a large class of systems.

- equation based and non-causal, equations not assignments
- declarative, not an algorithm (like Simulink)
- object oriented
- multi domain
- continuous and discrete (not time) models, hybrid systems
- mainly developed for simulation
- high index problem more “more common than not”
- Language resources `http://modelica.org/`

Commercial model editors



Modelica standard library v3.2.2 (2016-04-03)

Blocks	Continuous, discrete and logical input/output blocks (Continuous, Discrete, Logical, Math, Nonlinear, Routing, Sources, Tables)
Constants	Mathematical and physical constants (pi, eps, h, ...)
Electrical	Electric and electronic components (Analog, Digital, Machines, MultiPhase)
Icons	Icon definitions Math Mathematical functions for scalars and matrices (such as sin, cos, solve, eigenValues, singular values)
Mechanics	Mechanical components (Rotational, Translational, MultiBody)
Media	Media models for liquids and gases (about 1250 media, including high precision water model)
SIunits	SI-unit type definitions (such as Voltage, Torque)
StateGraph	Hierarchical state machines (similiar power as Statecharts)
Thermal	Thermal components (FluidHeatFlow, HeatTransfer)
Utilities	Utility functions especially for scripting (Files, Streams, Strings, System)

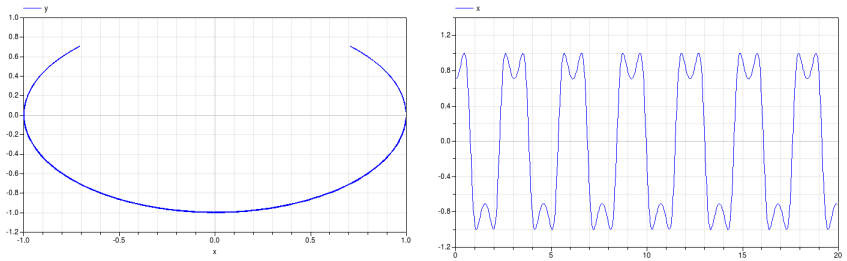
Modelica example

Pendulum model of index 3

```
model Pendulum
  import [Skipped some imports]
  parameter Real phi0 = 45*pi/180;
  parameter Real L = 1, M = 1, g=9.82;

  Real x(start=L*cos(phi0)), dx(start=0);
  Real y(start=L*sin(phi0)), dy(start=0);
  Real lambda;
equation
  der(x) = dx;
  der(y) = dy;
  M*der(dx) = lambda*x;
  M*der(dy) = lambda*y-M*g;
  0 = x*x+y*y-L*L;
end Pendulum;
```


Simulation result – pendulum example



Example with components from the standard library

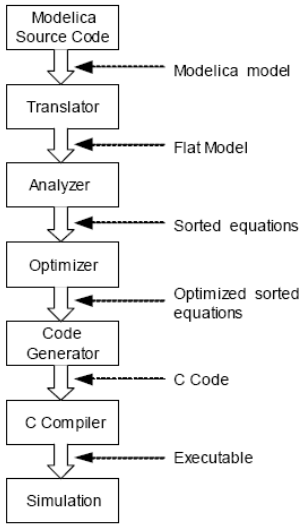
Electrical circuit

```
class SimpleCircuit
  Resistor R1(R=10), R2(R=100);
  Capacitor C(C=0.01);
  Inductor L(L=0.1);
  VsourceAC AC;
  Ground G;
equation
  connect(AC.p, R1.p);
  connect(R1.n, C.p);
  connect(C.n, AC.n);
  connect(R1.p, R2.p);
  connect(R2.n, L.p);
  connect(L.n, C.n);
  connect(AC.n, G.P);
end SimpleCircuit;
```

Outline

Simulation of Modelica models

The path from a Modelica model to C-code looks something like this



Representation of a general DAE

A DAE has the following components

$$F(x(t), \dot{x}(t), y(t), u(t), t, \theta, c(t)) = 0$$

- $x(t)$ vector of dynamic variables
- $y(t)$ vector of static variables
- $u(t)$ vector of given input/known signals
- θ model parameters
- $c(t)$ Event functions (more about these later)

I have skipped discrete variables, see book by P. Fritzons for more details.

Translator

- Parse the model file into an AST (Abstract Syntax Tree), a computer representation of the model
- Flatten the model, i.e., resolve all inheritance and component equations
- Change `if-expressions` to `if-equations`

```
if (a>5.0)
  val = 10.0;
else
  val = 20.0;
end if;
```

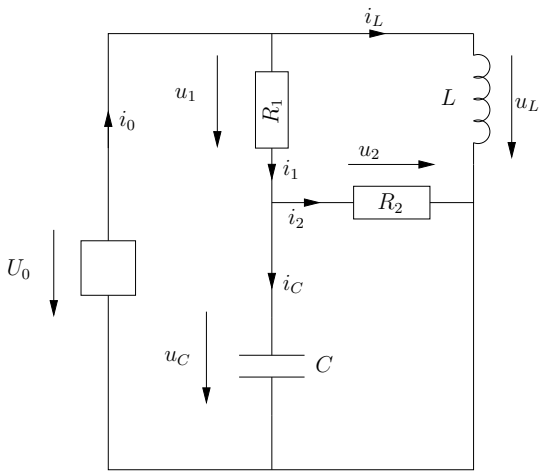
to

```
val = (if a>5.0 then 10.0 else 20.0);
```

- Change dot-notation to underbar (”.” → ”_”)

$$R1.p.i \rightarrow R1\_p\_i$$

Now we have the flattened model




```
model Circuit
  Resistor R1;
  Resistor R2;
  Capacitor C;
  Inductor L;
  Ground G;
  SineVoltage src;
equation
  connect(G.p, src.n);
  connect(src.p, R1.p);
  connect(src.p, L.p);
  connect(R1.n, R2.p);
  connect(R1.n,C.p);
  connect(L.n,R2.n);
  connect(L.n, C.n);
  connect(C.n,G.p);
end Circuit;
```


Flattened model

```
R1.R * R1.i = R1.v;
R1.v = R1.p.v - R1.n.v;
0.0 = R1.p.i + R1.n.i;
R1.i = R1.p.i;
R2.R * R2.i = R2.v;
R2.v = R2.p.v - R2.n.v;
0.0 = R2.p.i + R2.n.i;
R2.i = R2.p.i;
C.i = C.C * der(C.v);
C.v = C.p.v - C.n.v;
0.0 = C.p.i + C.n.i;
C.i = C.p.i;
L.L * der(L.i) = L.v;
L.v = L.p.v - L.n.v;
0.0 = L.p.i + L.n.i;
L.i = L.p.i;
G.p.v = 0.0;
src.signalSource.y = sin();
src.v = src.signalSource.y;
src.v = src.p.v - src.n.v;
0.0 = src.p.i + src.n.i;
src.i = src.p.i;
L.n.i + R2.n.i + C.n.i + G.p.i
+ src.n.i = 0.0;
L.n.v = R2.n.v;
R2.n.v = C.n.v;
C.n.v = G.p.v;
G.p.v = src.n.v;
R1.n.i + R2.p.i + C.p.i = 0.0;
R1.n.v = R2.p.v;
R2.p.v = C.p.v;
src.p.i + R1.p.i + L.p.i = 0.0;
src.p.v = R1.p.v;
R1.p.v = L.p.v;
```


Analyzer and optimizer

Main objectives

1. Transform the model to state space form (or index 1) through index reduction
2. Optimize the computation of the model (evaluation of F)

Steps

- Basic model analysis, find simple, e.g., linear, equation
- Simplify model by eliminating trivial (and some simple) equations by algebraic manipulation
- Compute index and perform index reduction
- Find algebraic loops (strong components) with respect to the most differentiated variables

Analyzer and optimizer, cont.

Three main problems I will not address further today

- compute index
- perform index reduction and transform the model equations from a high index formulation to ODE/index-1 DAE
- find consistent initial values

All these are strongly connected and the main content of the next lecture.

From now on, it is assumed the model is index 1.

Code generation

Simplified; it is assumed that the model is in the form

$$F(x', x, t) = 0, \quad c_i(t), \quad i = 1, \ldots, n_c$$

- The residual function $F(x', x, t)$ and event functions can now be sent directly to the numerical DAE solver
- This is not exactly how OpenModelica operates, it goes one step further and takes the model into an ODE. More on this later.
- On the course web site there is a (rather incomplete) document that shows by example how OpenModelica transfers a simple model to C-code. Recommended for the interested.

Outline

Flat model with index 1 to ODE, OpenModelica

$$F(x', x, y, t) = 0$$

If the model is low index, then it is possible to solve for the highest differentiated variables x' and y as

$$x' = f(t, x, y)$$
$$y = G(t, x)$$

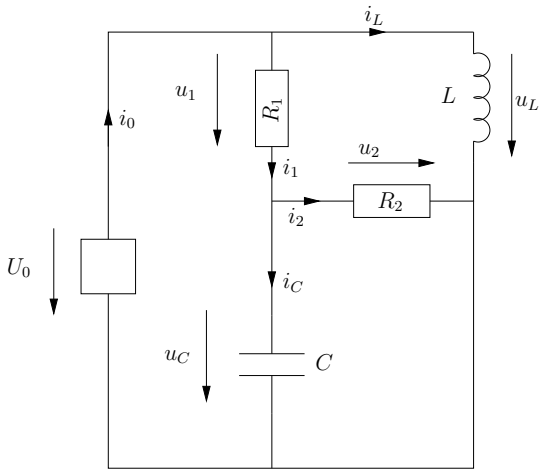
To generate code there is a need to find a computational scheme for the functions f and G, preferably as a pure substitution chain.

After that, OpenModelica simulates (with DASSL) x as

$$\dot{x} = f(t, x, G(x))$$

Remember the initial examples from the first DAE lecture, substitution chains not always possible.

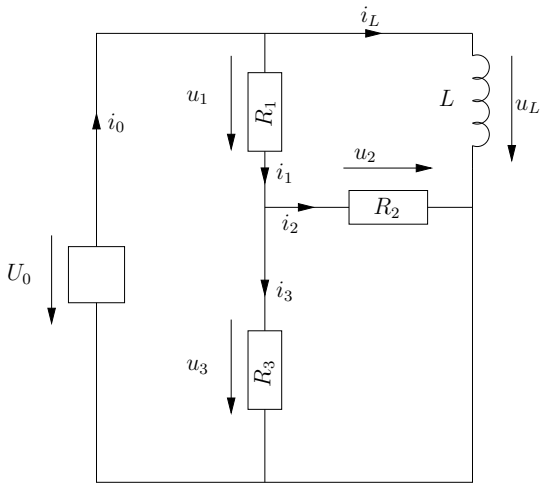
Remember simple circuit model



$$e_1 : u_0 = f(t)$$
$$e_2 : u_1 = R_1 i_1$$
$$e_3 : u_2 = R_2 i_2$$
$$e_4 : i_C = C\frac{du_c}{dt}$$
$$e_5 : u_L = L\frac{di_L}{dt}$$
$$e_6 : i_0 = i_1 + i_L$$
$$e_7 : i_1 = i_2 + i_C$$
$$e_8 : u_0 = u_1 + u_C$$
$$e_9 : u_L = u_1 + u_2$$
$$e_{10} : u_C = u_2$$

$x_1 = (u_c, i_L)$, $x_2 = (u_2, i_2, u_0, u_1, u_L, i_1, i_C, i_0)$

Remember the simple circuit model



$$u_0 = f(t)$$
$$u_1 = R_1 i_1$$
$$u_2 = R_2 i_2$$
$$u_3 = R_3 i_3$$
$$u_L = L\frac{di_L}{dt}$$
$$i_0 = i_1 + i_L$$
$$i_1 = i_2 + i_3$$
$$u_0 = u_1 + u_3$$
$$u_L = u_1 + u_2$$
$$u_3 = u_2$$

$x_1 = i_L$, $x_2 = (u_L, u_2, i_2, u_0, u_1, u_L, i_1, i_C, i_0)$

Remember the simple circuit model, cont.

$$\frac{di_L}{dt} = \frac{1}{L}u_L$$

$$u_0 := f(t)$$

Solve for $\{u_1, u_2, u_3, i_1, i_2, i_3\}$ in (6 unknowns, 6 equations)

$$u_1 = R_1 i_1$$
$$u_2 = R_2 i_2$$
$$u_3 = R_3 i_3$$
$$i_1 = i_2 + i_3$$
$$u_0 = u_1 + u_3$$
$$u_3 = u_2$$

$$i_0 := i_1 + i_L$$
$$u_L := u_1 + u_2$$

Example: BLT form of dynamic part of model

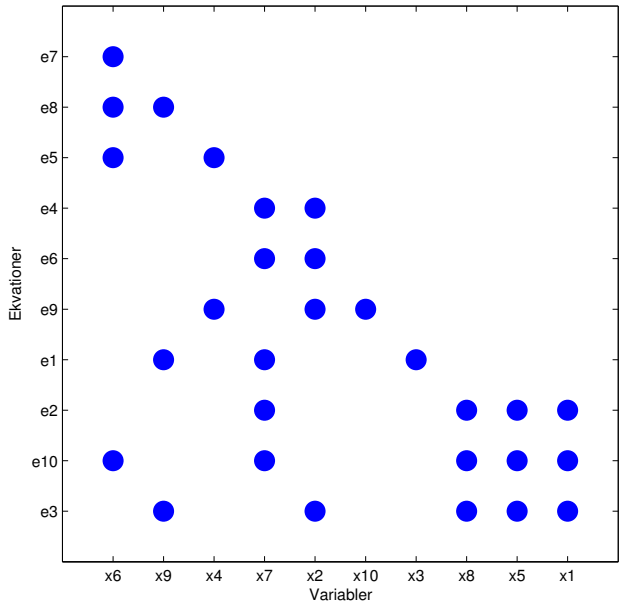
An electrical circuit

```
R2.v      = R2.R*L.i
R1.p.v    = AC.VA*sin(2*AC.f*AC.PI*time)
L.v       = R1.p.v-R2.v
R1.v      = R1.p.v-C.v
C.i       = R1.v/R1.R
der(L.i)  = L.v/L.L
der(C.v)  = C.i/C.C
```

Highest differentiated variables: $(R2.v, R1.p.v, L.v, R1.v, C.i, L.i', C.v')$

$R2.v$	$R1.p.v$	$L.v$	$R1.v$	$C.i$	$L.i'$	$C.v'$
1						
	1					
1	1	1				
	1		1			
			1	1		
		1			1	
				1		1

Algebraic loops and Tarjans algorithm/BLT form



Algebraic and dynamic loops

An algebraic loop can, in its simplest form, look like

$$x_1 = f_1(x_2)$$
$$x_2 = f_2(x_1)$$

Here x_1 and x_2 has to be computed simultaneously using either analytical tools (not feasible in the general case) or a numerical solver. Two simple numerical methods are

- Newton iterations
- Fix point iterations

Tearing

With fix point iterations (and contraction conditions) the loop

$$x_1 = f_1(x_2)$$
$$x_2 = f_2(x_1)$$

can be solved through the iteration

```
NEWx2 = INITx2
repeat
  x2 := NEWx2
  x1 := f1(x2)
  NEWx2 := f2(x1)
until converged(NEWx2-x2)
```

Here it was "clear" where to *tear* the loop due to the given causality. For a general loop

$$f_i(x_1, \ldots, x_n) = 0, \quad i = 1, \ldots, n$$

this is not as clear.

Tearing

For a general algebraic loop

$$f_i(x_1, \ldots, x_n) = 0, \quad i = 1, \ldots, n$$

it is generally a difficult problem to choose how to rewrite as a fix point iteration on the best way; requires knowledge about the analytical expressions, inversion properties, contraction properties etc.

In the course literature, there is a structural method for choosing tearing variables

Good choice of tearing variables is often based on physical insight and therefore it is possible to introduce such information in the model definition (not discussed further here)

Tearing, example 1

Below is a system of equations, which is an algebraic loop.

$$\begin{aligned}
c: &\quad u_3 = R_3 i_3 \\
d: &\quad u_1 = u_0 - u_3 \quad (u_0 \text{ known}) \\
a: &\quad i_1 = 1/R_1 \; u_1 \\
e: &\quad u_2 = u_3 \\
b: &\quad i_2 = 1/R_2 \; u_2 \\
f: &\quad i_1 - i_2 - i_3 = 0
\end{aligned}$$

Choose i_3 as tearing variable in equation f.
Then we get an equation to solve numerically

$$f: \; g(i_3, i_1(i_3), i_2(i_3)) = \tilde{g}(i_3) = 0$$

For example by Newton iterations

$$i_3^{(k+1)} = i_3^{(k)} - \tilde{g}'(i_3^{(k)})^{-1} \tilde{g}(i_3^{(k)})$$

Tearing, example 2

Instead, choose i_1 in equation a as a tearing variable

$$\begin{aligned} a: &\quad u_1 - R_1 i_1 = 0 \\ b: &\quad u_2 - R_2 i_2 = 0 \\ c: &\quad u_3 - R_3 i_3 = 0 \\ d: &\quad u_1 + u_3 - u_0 = 0 \\ e: &\quad u_2 - u_3 = 0 \\ f: &\quad i_1 - i_2 - i_3 = 0 \end{aligned}$$

Then we do not get anywhere. No new variable can be computed even though i_1 is assumed known.

Tearing, example 3

Now, choose a tearing variable, u_2 in equation b, then it all clears out

$$\begin{aligned} e: &\quad u_3 = u_2 \\ c: &\quad i_3 = 1/R_3\ u_3 \\ d: &\quad u_1 = u_0 - u_3 \\ f: &\quad i_2 = i_1 - i_3 \end{aligned}$$

and the following two equations has to be solved, for example by Newton iterations for variables i_1 and u_2

$$\begin{aligned} a: &\quad u_1 - R_1 i_1 = 0 \\ b: &\quad u_2 - R_2 i_2 = 0 \end{aligned}$$

Tearing, conclusions

- Tearing is used to tear algebraic loops apart into smaller parts
- The objective is to rewrite a large system of equations into smaller systems of equations that can be solved numerically
- Choice of tearing variables and equations is complex, NP-hard to find a minimal set of tearing variables.

A simple heuristic

1. Choose an equation e that contains the most unknown variables
2. In equation e, for each variable v compute how many variables that can be computed by a direct substitution chain of v is assumed known.
3. Choose the variable that maximizes the number of new computed variables as a tearing variable and e as iteration equation.

Simple heuristic on the example

$$\begin{aligned} c: &\quad u_3 = R_3 i_3 \\ d: &\quad u_1 = u_0 - u_3 \\ a: &\quad i_1 = 1/R_1\ u_1 \\ e: &\quad u_2 = u_3 \\ b: &\quad i_2 = 1/R_2\ u_2 \\ f: &\quad i_1 - i_2 - i_3 = 0 \end{aligned}$$

Equation with most variables is f with variables (i_1, i_2, and i_3.
Assume known:

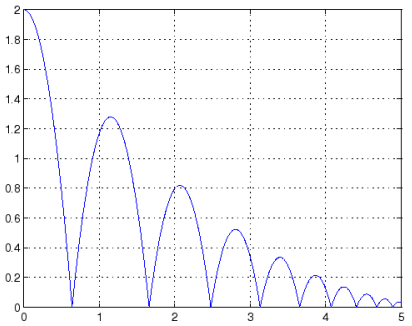
- i_1: u_1 (1 variable)
- i_2: u_2, u_3, i_3, i_1, u_1 (5 variables)
- i_3: u_3, u_1, i_1, i_2, u_2 (5 variables)

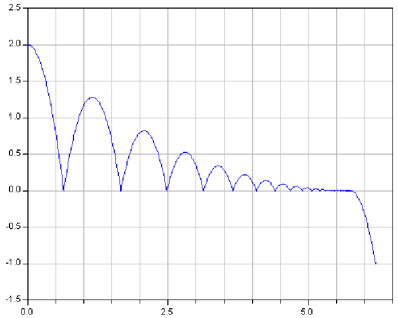
Conclusion: Choose i_2 or i_3 as tearing variable.

Outline

- An overview of a solver
- Adjoint sensitivity analysis of DAEs
- Modelica
  - Snapshot of the language
  - Simulation of Modelica models
  - Algebraic and dynamical loops/tearing
  - Event functions
- Structural index - introduction and definition

Event/crossing functions





- Here it is clear that the zero-crossing is very important for the ball not to fall through the floor
- Important in many simulation problems that certain bounds not are crossed

Event/crossing functions

Modelica

```
class BouncingBall
  constant Real g=10
  parameter Real c=0.9, radius=0.1
  Real y(start=1), velocity(start=0), x(start=1)
equation
  der(x)=0;
  der(y) = velocity;
  der(velocity)=-g;
  when height<=radius then
    reinit(velocity,-c*pre(velocity))
  end when;
end BouncingBall;
```


Automatic generation of event functions

if-expression

```
y = if x>z then a else b;
```

if-equation

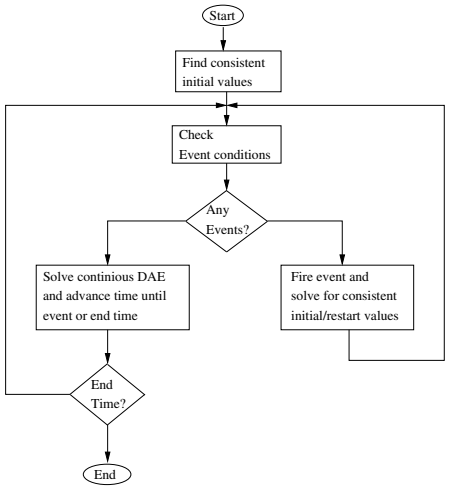
```
if x>z then
  y=a
else
  y=b
end if;
```

- Conditional expressions are translated into if-equations
- direct to generate event functions. In the example above we get

$$c_i(vars) = x - z$$

- Here it is clear that the modelling can help the compiler to, by itself, figure out important conditions that has to be monitored.

Event/crossing functions

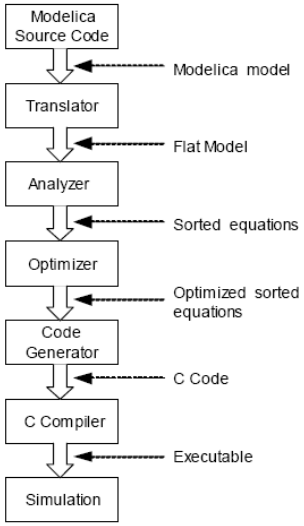


- Fits well into the framework of DASSL and other standard solvers are structured

Zeno effect

- You may run into problems of the solution has infinitely many event detections
- Is sufficient that they are dense enough with respect to the numerical precision for this to be a problem
- For the bouncing ball, this of course happens after a while
- No general solution, logic has to be built into the model to handle the case where the ball starts to roll rather than bounce.

Conclusion



Steps I have not yet addressed concerns the Analyzer/Optimizer steps

Earlier discussion has assumed an ODE (or low index) form

Remains to:

- compute index
- reduce index
- determine consistent initial conditions

Common, and general, solution to the above three problems has a lot in common and will be the main theme for next lecture.

Reduce to ODE or DAE with index 1?

- Standard solvers can handle fully implicit idex 1 DAE models, why?
  - well written code, efficient
  - is it really the best?
- Implicit solvers not efficient for non-stiff problems
- Large model often exhibits stiff dynamics
- Dymola/Openmodelica reduces model to ODE (as far as I know), is not DAE index 1 sufficient?
- "Cellier vs. Petzold"

Outline

Structural index

An important step in the procedure to transfer the model to C code is to perform index reduction. Index reduction requires that you know the index of the model. As we know it is a difficult problem in general to determine index; a method based on model structure is typically used.

Structural index can be defined in many ways. One way, for the DAE

$$A\dot{x} + Bx = 0$$

the structural index is the real index the DAE has for *almost all* A and B with the same structure.

- simple to generalize to non-linear systems
- can be computed with Pantelides algorithm, which will be used also for other purposes

Structural index - introductory example

Let $x = (x_1, x_2) \in \mathbb{R}^2$ and consider the index 1 model

$$\begin{aligned} \dot{x}_1 &= x_1 + x_2 + u \\ 0 &= -2x_1 + x_2 \end{aligned}$$

	$\dot{x}_1$	x_2
e_1	X	X
e_2		X

The highest differentiated variables are $x_{hd} = (\dot{x}_1, x_2)$

DAE has index 1 for almost all coefficients in front of the x variables, only when coefficients in front of $\dot{x}_1$ in e_1 or x_2 in e_2 is 0 we have a problem.

Conclusions: we can from the table on the right determine that this model has (structural-)index 1.

$$F(\dot{x}_1, x_1, x_2) = 0$$

has low index (locally) if

$$\left.\frac{\partial F(\dot{x}_1, x_1, x_2)}{\partial x_{hd}}\right|_{\dot{x}_1 = x_1'^{,*}, x_1 = x_1^*, x_2 = x_2^*}$$

has full rank

Structural index - introductory example

Let $x = (x_1, x_2, x_3) \in \mathbb{R}^3$

$$\begin{aligned} \dot{x}_1 &= x_1 + x_2 + x_3 + u \\ 0 &= -2x_1 + x_2 \\ 0 &= x_1 + x_2 + u \end{aligned}$$

	$\dot{x}_1$	x_2	x_3
e_1	X	X	X
e_2		X	
e_3		X	

From the table on the right we see that *regardless* of which coefficient we have for the variables, the DAE has index > 1. The DAE has a unique solution since

$$|\lambda E - A| = \lambda + 3 \neq 0, \quad (x_1, x_2, x_3) = (-\frac{1}{3}u, -\frac{2}{3}u, -\frac{1}{3}\dot{u})$$

Turns out you can determine structural index only by looking at the tables on the right. This is also direct to automatically do for large scale models in general purpose simulation environments.

Let ν and ν_{str} be the index and the structural index respectively for

$$F(t, y', y) = 0$$

What holds?

$$\nu < \nu_{str}, \qquad \nu \leq \nu_{str}$$
$$\nu_{str} < \nu, \qquad \nu_{str} \leq \nu$$

What is the consequence of this for a method that relies on a structural algorithm for index reduction?

Consider the linear DAE

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \dot{x} + x = 0$$

This DAE can be shown to have index 1 but structural index larger than 1 (exercise, what is the structural index?)

Well known that structural index can be less than the index, but the example above shows that it can also be the other way around.

Lecture 3

Simulation of differential-algebraic equations

Erik Frisk
erik.frisk@liu.se

Department of Electrical Engineering
Linköping University

February 23, 2022