Dynamic Parameter Sensitivities: Summary of Computation Methods for Continuous-time Modelica Models

Atiyah Elsheikh

Mathemodica.com , Egypt & Germany , Atiyah.Elsheikh@mathemodica.com

Abstract

Applications of Sensitivity Analysis (SA) encouraged several Modelica platforms to independently provide facilities for externally computing Dynamic Parameter Sensitivities (DPS). FMI specifies an optional function call for evaluating directional derivatives. On the other hand, mathematical foundation for uniform representation of DPS at the Modelica language level has been established. This has resulted in a platform-independent approach demonstrated through example libraries. The paper neutrally hints that already conducted efforts may converge to the integration of language facilities for DPS without neglecting to mention many mathematical difficulties. Surprisingly, many of what could be thought to be algorithmic obstacles have intuitive solutions along a minimalist implementation approach.

Keywords: algorithmic differentiation, parameter sensitivities, sensitivity analysis

1 Introduction

1.1 Motivation to DPS

In (Wiechert et al., 2010) one reads:

> Simulation tools not only perform numerical solutions based on the system equations but also assist the modeler in systems analysis. Doubtlessly the most important systems analysis tool is SA ... SA is required for parameter fitting, statistical regression analysis, experimental design, and metabolic control theory.

Analogously, common guidelines for model-based studies recommend SA to be performed in order to assist the validity of the conclusions, for example demanded from:

1. Impact Assessment Guide, European Commission
2. Guidance on the development, evaluation and application of environmental models, US Environmental Protection Agency

Hence, facilities for SA are crucial for modelers if offered by a modeling language or a simulation platform.

While many methods of SA are based on numerical approaches (Saltelli et al., 2004) their applicability on typically large-scale Modelica models is questionable. As an attractive alternative, analytical derivatives of model outputs w.r.t. model inputs can be exploited. Derivative-based approaches usually lead to superior results in terms of accuracy and computational complexity e.g., derivative-based global SA methods (Kucherenko and Iooss, 2016). In Modelica-based terminologies, Dynamic Parameter Sensitivities (DPS) are sought.

1.2 Applications of DPS

Despite of many applications of SA DPS enable, there are not so many works conducted within the Modelica community exploiting DPS. One possible reason for that (up to the author knowledge) there is no comprehensive literature focusing on general applications of DPS. Instead, their applications are splintered among thousands of books and articles many of which belong to Chemical and Bio-Engineering domains. In conjunction with this paper, (Elsheikh and Kucherenko, 2019) provides a new classification and initial summary of applications of DPS. In this technical report[1], three families of application samples are categorized:

1. *Modeling-oriented applications:*
   Applications benefiting from the presence of DPS at the model level without the need of mathematically sophisticated post processing or leaving the GUI of the simulation platform such as control coefficients (Fell, 1992), local SA, parameter sweeping studies, model simplification and error analysis

2. *statistical-oriented applications:*
   Statistical tools benefiting from DPS for improved efficiency in terms of accuracy and computational complexity such as regression analysis, global SA, uncertainty analysis and identifiability analysis

3. *optimization-oriented applications:*
   high-level optimization problems with objective

[1] The technical report is subject to continuous modification from the author. Interested readers are welcome to contribute.

functions expressed in terms of DPS such as experimental design, optimal design and parameter estimation in conjunction with identifiability analysis

Figure 1 demonstrates an application of a DPS-enabled simulation. The underlying model is capable of additionally computing DPS. Consequently it was possible to apply parameter sweeping studies using generic models within the PSTools library.

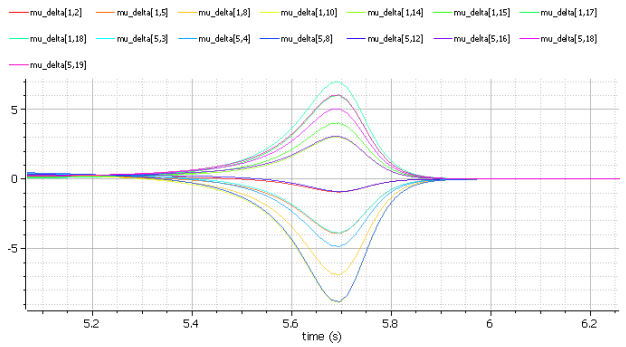

Figure 1. Parameter sweeping study based on one DPS-enabled simulation with two parameters. Instead of running multiple simulations, Taylor series expansion is exploited

1.3 Methodologies for Computing DPS

Assume that a given continuous-time Modelica model equivalently corresponds to an explicit ODE:

$$\dot{x} = f(x,q,t) = 0 \quad , \quad x_0(q,t_0) = x_0(q) \tag{1}$$

where:

- t : time
- $q \in R^{n_q}$: set of model parameters
- $x(q,t) \in R^{n_x}$: set of state variables

The chosen parameters (say $p \in R^{n_p}$) w. r. t. which derivatives are sought are referred to as *active parameters*. This work is summarizing some of the available methods for computing DPS

$$\frac{\partial x}{\partial p}(p,t) \in R^{n_x \times n_p}$$

These methods are:

1. Finite Difference (FD) methods (Section 2): straightforward to implement with Modelica but subject to serious accuracy issues
2. Specialized solvers (Section 3): provided by specific simulation environments usually accessible by external scripting capabilities
3. Equation-based Algorithmic Differentiation (AD) (Section 4): a platform-independent approach allowing DPS to be present at the language level but requires additional implementation efforts

Section 5 discusses a potential enhancement to the Modelica language by allowing the operator *der* to take a second argument as a model parameter. The discussion includes some expected obstacles with a minimalist implementation approach highlighted in Section 6 and an outlook given in Section 7.

2 Finite Difference Methods

2.1 Implementation

A straightforward way to compute DPS is realizable by a first-order FD method. Assuming that simulation of Equation (7) leads to the approximated solution:

$$x_i(p,t_k) = x_{i,k}(p) \tag{2}$$

$$\text{for} \quad i \in \{1,2,...,n_x\} \quad \& \quad k = 0,1,2,..$$

DPS are evaluated by post-processing additional n_p simulations each with one slightly modified active parameter:

$$\frac{\partial x_i}{\partial p_j}(t_k,p) \quad \approx \quad \frac{x_{i,k}(p+e_j\delta_j) - x_{i,k}(p)}{\delta_j} \tag{3}$$

where:

$$\delta_j = \xi_j \cdot p_j, \quad \xi_j : \text{perturbation factor}$$

and $e_j \in R^{n_p}$: the j-th unit vector. However, utilizing Modelica capabilities, DPS can be implemented via one simulation as follows:

```
model FDModel
 parameter Real zeta = 0.01 "perturbation";
 parameter Real p1=3.0,p2=0.3,p3=...;

 MyModel M (p1=p1,p2=p2,...);
 MyModel M1(p1=p1*zeta+p1,
            p2=p2,
            ...);
 MyModel M2(p1=p1,
            p2=p2*zeta+p2,
            ...);
 ...
 Real dxdp[nx,np] "DPS";
 ...
equation
 dxdp[1,1] = (M1.x1-M.x1) / (p1*zeta);
 dxdp[1,2] = (M2.x1-M.x1) / (p2*zeta);
 ...
 dxdp[2,1] = (M1.x2-M.x2) / (p1*zeta);
 dxdp[2,2] = (M2.x2-M.x2) / (p2*zeta);
 ...
end FDModel;
```

2.2 Performance

Theoretically, for a Modelica model with say n_x nontrivial equations and n_p active parameters, the previous implementation results in a model with $n_p(n_x+1)$ nontrivial equations. The current computation paradigms for Modelica simulation environments (at least for the published

DOI 10.3384/ecp19157795

ones) imply that the above model translates to one single block of an ODE / a DAE system of equations. This causes performance drawbacks when considering large number of active parameters. One may rather attempt to consider fewer numbers of active parameters and additional models. However, by extending BLT-based speed-up techniques (Cellier, 1991) to ODEs / DAEs level rather than only algebraic equations, advanced compiler strategies may allow independent simulation of smaller blocks of ODEs / DAEs, even in parallel. Hence, run-time performance of FD-methods should not be the main concern but the accuracy. As mathematically justified in (Elsheikh and Wiechert, 2012), the study shows that FD is non-reliable in the context of large-scale nonlinear dynamics. Hence FD, in the way implemented in this subsection, should not be the chosen approach for computing DPS.

2.3 Applying FD within the PSTools Library

Employing FD for computing DPS can still be exploited only as a first experimental step, in particular if advanced FD strategies and more accurate formulas are employed. This is also useful to validate other approaches for computing DPS. The PSTools library provides modelers capabilities for computing DPS of arbitrary Modelica models. Given a Modelica model *M*, the first step is to parameterize the model under consideration as follows[2]:

```
model ParM
  extends PSTools.Utilities.Parameterized(
    NP = 2,
    _P = {0.4, 0.5 / 3600},
    PNAME = {"p1", "p2"},
    NX = 4,
    _X = {_M.x1, _M.x2},
    XNAME = {"x1", "x2"});
protected
    M _M(p1 = _P[1], p2 = _P[2]);
end ParM;
```

In this way, the active parameters and the significant variable of the model *M* are specified. Alternatively, exploiting enumeration classes is also possible:

```
model ParM
  extends
    PSTools.Utilities.EnumParameterized(
      redeclare type ActivePars =
                        enumeration(p1,p2),
      _P = {0.4, 0.5 / 3600},
      redeclare type SignificantVars =
                        enumeration(x1,x2));
protected
  M _M(p1 = _P[ActivePars.p1],
       p2 = _P[ActivePars.p2]);

equation

  _X[SignificantVars.x1]  = _M.x1;
  _X[SignificantVars.x2]  = _M.x2;

end ParM;
```

[2]other styles other than declaring the model under consideration in a protected section are also realizable

Once a model under consideration is a subclass of *PSTools.Utilities.Parameterized*, it is straightforward to employ advanced capabilities within the PSTools library for computing DPS as follows:

```
model FDParM
  PSTools.PS.FD.CD2
    PS(redeclare replaceable model
         ParModel =  ParM);

  Real g_x1_p1, g_x1_p2, g_x2_p1, g_x2_p2;

equation

  g_x1_p1 = PS.g_x[1,1];
  g_x1_p2 = PS.g_x[1,2];
  g_x2_p1 = PS.g_x[2,1];
  g_x2_p2 = PS.g_x[2,2];

end FDParM;
```

In this way DPS are computed using a central difference formula of order two. Figure 2 demonstrates the trajectories of DPS for some chosen significant variables w.r.t. some active parameters for a Modelica model in the Bio-engineering domain with 10 significant variables and 9 active parameters resulting in 90 trajectories for DPS.

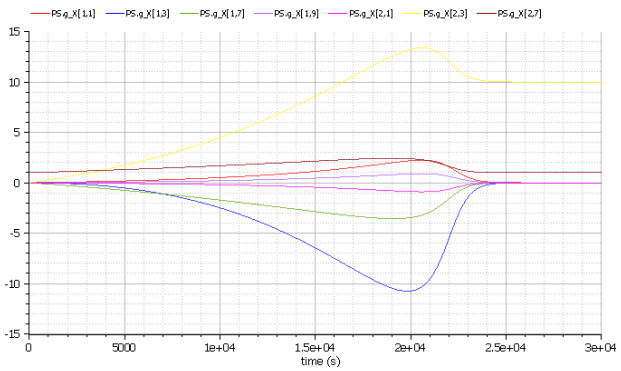

Figure 2. Some trajectories of DPS of a Model from Bio-Engineering domain

3 Specialized Solvers

3.1 Approach

There are specialized numerical solvers, e.g. Sundials package (Hindmarsh et al., 2005) or DSPACK (Petzold et al., 2006) that can be employed for computing DPS. This is done by numerical integration of the sensitivity system composed of the ODE (1) together with sensitivity subsystems:

$$\dot{x}_p = f_x x_p + f_p \quad , \quad x_p(t_0) = \partial x_0(p)/\partial p \qquad (4)$$

obtained by differentiating ODE (1) w.r.t. p. OpenModelica and Wolfram SystemModeler exploit such solvers with the help of external scripting capabilities. Dymola can be coupled with the DSPack solver as shown in (Wolf et al., 2008). FMI2.0 specifies a

functionality for evaluating partial derivatives of an FMU (*fmi2GetDirectionalDerivative*). Due to apparent implementation overhead this functionality is optional.

A common way to compute the required partial derivatives f_x and f_p is to employ classical AD techniques (Naumann, 2012) (cf. `www.autodiff.org`). The simpler way to accomplish that is via a computational approach for forward differentiation, cf. next Subsection. Alternatively, advanced heuristics based on FD schemes can be utilized. Instead of direct numerical integration of a system of size $n_x(n_p+1)$, couple of methods exist with improved performance efficiency (Maly and Petzold, 1996; Feehery et al., 1997). These methods attempt to exploit the linearized form of a sensitivity system and the structure of its extended Jacobian.

3.2 Computation of Partial Derivatives

Given the equation system f from (1), an AD algorithm in forward mode is capable of computing the s.c. tangent linear model y corresponding to directional derivatives of f w.r.t. a user-defined input. Formally, y is computed as:

$$y(u,s) = \frac{\partial f}{\partial u} \cdot s \tag{5}$$

where $u \in \{x,p\}$ and $s \in R^{|u|}$ is a seed vector. Several works related to evaluation of the Jacobian (i.e. $f_x(t) \in R^{n_x \times n_x}$) via AD in Modelica compilers have been reported (Olsson et al., 2005; Andersson et al., 2010; Braun et al., 2011). Given that $x = (x_1, x_2, \ldots, x_{n_x})^T$, the term f_x is accumulated by letting s ranges over the Cartesian basis of R^{n_x} as follows:

$$\begin{aligned} f_x = \left[y(x,e_i) \right]_{i=1,2,\ldots,n_x} &= \left[\frac{\partial f}{\partial x} \cdot e_i \right]_{i=1,2,\ldots,n_x} \\ &= \left[\frac{\partial f}{\partial x_i} \right]_{i=1,2,\ldots,n_x} \end{aligned} \tag{6}$$

where e_i is the i-th unit vector in R^n and each iteration i computes the i-th column in f_x. If f is expressed as a functional unit within a program P, AD produces another program P' that includes evaluation of f together with y. This implies that the program P' needs to get repeatedly evaluated for n_x (n_p) times in order to compute f_x (f_p) with potentially common intermediate computations being repeatedly performed.

A possibility to reduce such excessive evaluations is to exploit sparsity patterns of object-oriented Modelica models. In this way, many directional derivatives can be simultaneously computed by compressing many unit vectors into the seed vector s (Braun et al., 2012). However, even with this technique, it is not a wonder that straightforward symbolic differentiation outperforms AD w.r.t. runtime performance as reported in (Åkesson et al., 2012). In the case of $n_p \gg n_x$, excessive computational efforts can be avoided by following a backward AD approach as demonstrated in (Braun et al., 2017). (Hannemann-Tamas et al., 2012) shows another approach for computing first and second-order DPS using adjoint sensitivity analysis applied to a flattened Modelica model.

4 Equation-based AD for Modelica

So far demonstrated reliable approaches for computing DPS are limited to external solutions (FMI) or non-unified platform-dependent additional services. In contrary, a platform-independent approach to realize DPS is to exploit equation-based AD technique with which DPS is modeled using Modelica syntax. Given a Modelica model (or a library) DPS are modeled by systematically extending every base component by another component additionally including entities and equations for the sensitivity system. Then, a top-level model is slightly changed by specifying the set of active parameters.

4.1 Example: The ADGenKinetics Library

The ADGenKinetics library (Elsheikh, 2012) an algorithmically differentiated library capable of modeling the dynamics of biochemical reaction networks together with DPS, the major connection mechanism:

```
connector ChemicalPort
   "reaction connector"
  Units.Concentration c "concentration";
  flow Units.VolumetricReactionRate r
                        "reaction rate";
end ChemicalPort;
```

is extended within another separate package *Derivatives* as follows:

```
connector ChemicalPort
  extends Interfaces.ChemicalPort;
  outer parameter Integer NG
          "dimension of the gradients";
  Real g_c[NG] "gradients of c";
  flow Real g_r[NG] "gradients of r";
end ChemicalPort;
```

The parameter *NG* specifies the number of active parameters that is first to be specified at a top-level model. The array *g_c* (*g_r*) is a derivative object describing the derivatives of the concentration (the reaction rate) w.r.t. active parameters. Similarly a component providing basic interfaces for arbitrary chemical substances:

```
partial model BasicNode
  "Basic declarations of any Metabolite"
  extends
      Interfaces.dynamic.NodeConnections;
  parameter Units.Concentration c_0 = 0
        "initial concentration";
  Units.Concentration c(start = c_0)
        "substance concentration";
  Units.VolumetricReactionRate r_net
        "net reaction rate";
equation
  r_net = rc.r;
  rc.c = c;
```


DOI 10.3384/ecp19157795

```
  mc.c = c;
end BasicNode;
```

is extended as follows:

```
partial model BasicNode
  "Basic declarations of any Metabolite"
  extends Derivatives.Interfaces.
            dynamic.NodeConnections;
  extends NodeElements.dynamic.BasicNode;
  outer parameter Integer NG
                "# of gradients";
  parameter Real g_c_0[NG] = zeros(NG)
                "gradients of c_0";
  Real g_c[NG](start = g_c_0)
                "gradients of c";
  Real g_r_net[NG](start = zeros(NG))
                "gradients of r_net";
equation
  g_r_net[:] = rc.g_r[:];
  rc.g_c[:] = g_c[:];
  mc.g_c[:] = g_c[:];
end BasicNode;
```

The extended equations are obtained by differentiating the original equations w.r.t. arbitrary parameters. Note that the differentiated component is now extending the differentiated model of *NodeConnections*. To each variable and parameter, a corresponding derivative object is associated, e.g. *g_c_0*. In this way, derivatives w.r.t. start values can be also represented. Obtaining derivative formulas as well as the naming conventions of intermediate variables are comprehensively explained in (Elsheikh, 2015). The algorithm is also employed to manually derive partial derivatives of complex formulas.

4.2 A Top-level Model

A top-level model is slightly modified from:

```
model Spirallusdyn "An abstraction TCA
    cycle"
  import NodeElements.dynamic.*;
  import Reactions.convenience.dynamic.*;
  ..
  Node A;
  RevKinetic v1
    (NS = 1, NP = 1,
     Vfwdmax = 3.0, Vbwdmax = 1.0,
     ...);
  ...
equation
  ...
  connect(A.rc, v1.rc_S[1]);
  connect(v1.rc_P[1], B.rc);
  ...
end Spirallusdyn;
%
```

to:

```
model spirallusdynAll "Parameter
    sensitivities"
 import Derivatives.NodeElements.dynamic.*;
 import Derivatives.Reactions.
           convenience.dynamic.*;
 import Derivatives.Functions.*;
 // instead of
 // import NodeElements.dynamic.*;
 // ...

 // additional declaration
 inner parameter Integer NG = 24;
 ...
 // the exisiting components
 Node A;
 RevKinetic v1
    (NS = 1, NP = 1,
    Vfwdmax = 3.0,
        // declare the 4-th active
            parameter
        g_Vfwdmax = unitVector(4, NG),
    Vbwdmax = 1.0,
        // declare the 5-th active
            parameter
        g_Vbwdmax = unitVector(5, NG),
    ...);
  ...
equation
  // connection equations remain the same
  ...
  connect(A.rc, v1.rc_S[1]);
  connect(v1.rc_P[1], B.rc);
  ...
end  spirallusdynAll;
```

in order to additionally simulate DPS, cf. (Elsheikh, 2012) Section 5 for simulation results. Imported differentiated types are employed and input Jacobian is additionally declared. Methodological details on the equation-based AD method are demonstrated in (Elsheikh, 2014) illustrated on a simple part of the MSL, the ADMSL library. Regardless of being platform-independent uniform approach, the main advantage of this method is that DPS are already a part of the model, profitable to many applications highlighted in (Elsheikh and Kucherenko, 2019).

5 Whispering about *der(x,p)*

5.1 *der(x,p)* !?

The established methodology for equation-based AD of Modelica models / libraries inherits an implicit but intuitive proposal: *der(x,p)*. This allows the operator *der* to take a second argument as a parameter to represent DPS. The equation-based AD approach can be viewed as an equivalent prototype implementation of such an enhancement. More or less, *der(x,p)* may provide all equivalent activities a modeler currently needs to explicitly implement DPS at model level. For instance, a demonstration may look as follows:

```
model MyModel
  MyComp1  C1;
  MyComp2  C2;
  Real dxdp;
  Real dydp;
  Real dydq;
equation
  dxdp = der(C1.x,C2.p);
  dydp = der(C2.y,C2.p);
  dydq = der(C2.y,C1.q);
end MyModel;
```

The above model is equivalently realizable by equation-based AD at the model level.

5.2 What speaks for *der(x,p)?*

The demonstrated efforts for computing DPS of Modelica models reveal that they unintentionally converge to provide DPS-capabilities at the language level:

1. If a Modelica simulation environment will eventually support (or are already supporting) DPS for FMI, then it makes sense also to consider DPS at Modelica level due to potentially duplicated efforts
2. Why should a Modelica modeler need to externally employ FMI with additional overhead for reasons FMI have not been apparently established for?
3. Representing DPS at model level is intuitive and realizable for significant subset of Modelica models, cf. Section 6
4. Regardless of a significant set of applications of DPS where its advantageous to have DPS at the model level (Elsheikh and Kucherenko, 2019), DPS can be physically part of the model (Fell, 1992)

The last argument is referring to the s.c. control coefficients corresponding to scaled DPS allowing comparative studies among parameters generally applicable to arbitrary physical domains.

5.3 What speaks against *der(x,p)?*

While the demonstrated algorithmically differentiated libraries correspond to continuous-time physical models, probably most of Modelica applications cover discrete phenomena. It should be clear how to treat *der(x,p)* if x is not continuous. Formally speaking: assume that the given Modelica model corresponds to an explicit hybrid ODE:

$$\dot{x} = f(x,z,m,q,t) = 0 \quad , \quad x_0(p,t_0) = x_0(p) \qquad (7)$$

where x, p and t as given in Equation (1) and

- $m(q,t) \in R^{n_m}$ set of discrete variables with constant values between events
- $z(q,t) \in R^{n_z}$ set of event indicators where $z_i(t_j)$ changes sign implies that for a small $\varepsilon > 0$:

$$m_k(t_j - \varepsilon) \neq m_k(t_j + \varepsilon),$$

$$\lim_{t \to t_j^+} x_s(t_j) \neq \lim_{t \to t_j^-} x_s(t_j) \text{ or}$$

$$\lim_{t \to t_j^+} \dot{x}_r(t_j) \neq \lim_{t \to t_j^-} \dot{x}_r(t_j)$$

  for some $k \in \{1,2,\ldots,n_m\}$, $s \in \{1,\ldots,n_x\}$ and $r \in \{1,\ldots,n_x\}$

Similarly, assume that p are the active parameters. Some justifiable questions, among others, on the validity of this study on hybrid systems would include:

1. When does a unique solution for the trajectories of sensitivities exist?
2. How to handle sensitivities when x jumps between events?
3. What if a parameter p_k is the time of event, i.e. $\partial p_k / \partial t_j = 1$ with some $z_i(t_j) = 0$
4. Generally, what if z is a function of parameters, i.e. $z = z(p,t)$?
5. How to handle the case where a parameter may even influence the presence of an event?
6. How to treat the influence of parameter values on event indicators z or even discrete variables m?

Definitely some of the above questions need to be adequately addressed. On the other hand, there are plenty of studies on SA of hybrid systems in literature e.g. (Galán et al., 1999; Barton and Lee, 2002; Saccon et al., 2014) some of which could be inspiring for addressing these questions.

6 Evaluation, Compilation and Simulation of Sensitivity Systems

Two AD approaches for generating a sensitivity system out of *der(x,p)* specification can be imagined. The classical one is to generate the required derivatives needed by specialized solvers for DPS, possibly using classical AD tools as demonstrated in Section 3.2. Another naive but attractive alternative is to self generate the sensitivity system using symbolic differentiation capabilities, if not equation-based AD. Afterwards the generated sensitivity system is subject to the standard optimization techniques of Modelica compilers (Murota, 1987; Cellier, 1991). In many cases, compilation complexity does not need to be overwhelmed due to the dimension of sensitivity system, as revealed in Section 6.2.

6.1 Equation-based AD

In the following, an advanced equation-based AD approach following a forward-differentiation scheme is demonstrated which

1. evaluates a sensitivity system in one single shot
2. overcomes the repetitive computations drawback
3. requires no additional memory for derivative objects of intermediate computations

This is the same approach that has been applied to compute partial derivatives of library components demonstrated in Section 4. Technically speaking w.r.t. the above mentioned first argument and partially the second argument, symbolic differentiation results in similar conclusions. The same applies to classical AD if the seed

DOI 10.3384/ecp19157795

vector s from Eq. (5) becomes a seed matrix and assigned to the identity matrix I_{n_x} for computing the Jacobian. However both methods were not satisfiable for computing partial derivatives at Modelica language level (Elsheikh et al., 2008).

On the other hand, equation-based AD combines the advantages of both approaches by employing algorithmic capabilities borrowed from equation-based compilation techniques for simplifying common sub-expressions. In (Elsheikh, 2015) it is shown that equation-based AD does not associate derivative objects for intermediate computation resulting in efficient memory management of large equation systems of complex formulas. The proposed approach directly derives the sensitivity system by computing the rhs of Eq. (4) in one single shot as follows:

$$f_x\, S_1 + f_p\, S_2 \tag{8}$$

$$\text{where } S_1 = \frac{\partial x}{\partial v} \in R^{n_x \times q} \quad , \quad S_2 = \frac{\partial p}{\partial w} \in R^{n_p \times q}$$

S_1 & S_2 are seed matrices (vectors if $q = 1$) that are set using the auxiliary vectors $v, w \in R^q$ in the following way:

1. only f_x required: let $v = x \in R^{n_x}$, $w = \phi_{n_x} \in R^{n_x}$ where ϕ_{n_x} is a dummy vector with which $\partial \dot{x} / \partial \phi_{n_x} = 0 \in R^{n_x \times n_x}$ and $\partial p / \partial \phi_{n_x} = 0 \in R^{n_p \times n_x}$

2. only F_p required: let $w = p \in R^m$, $u = v = \phi_m \in R^m$

3. sensitivity subsystem required: set $v = w = p \in R^{n_p}$

Despite the large size of sensitivity systems, they can be compactly represented. Assuming that $x = (x_1, x_2, \ldots, x_{n_x})^T$, $p = (p_1, p_2, \ldots, p_{n_p})^T$ and $f = (f_1, f_2, .., f_{n_x})^T$ with

$$f_i : R^{n_x + n_p + 1} \rightarrow R$$

$$\text{s.t. } f_i \equiv f_i(x, p, t) = e_i^T \cdot f(x, p, t) \ , \ i \in \{1, 2, \ldots, n_x\}$$

corresponds to the i-th equation in f, the corresponding differentiated equation w.r.t. a parameter p_j is derived from Equation (8) as follows:

$$\frac{\partial \dot{x}_i}{\partial p_j}(x, p, t) = \sum_{k=1}^{n} \frac{\partial f_i}{\partial x_k} \frac{\partial x_k}{\partial p_j} + \sum_{k=1}^{m} \frac{\partial f_i}{\partial p_k} \frac{\partial p_k}{\partial p_j} \tag{9}$$

By assigning a gradient object (i.e. array) to each partial derivative, the sensitivity subsystem is compactly formulated with n_x equations as:

$$\frac{\partial \dot{x}_i}{\partial p}(x, p, t) = \sum_{k=1}^{n} \frac{\partial f_i}{\partial x_k} \frac{\partial x_k}{\partial p} + \frac{\partial f_i}{\partial p} \tag{10}$$

6.2 Compilation of Sensitivity Systems

When computing the sensitivity system of a Modelica model, an approach is to generate the sensitivity equation after optimizing the flat representation of the model. However the other way around might not be significantly less efficient. Naive generation of the sensitivity system of a flat equation system is also subject to optimization techniques of Modelica code. For instance, the differentiation of simple equations from connections leads to simple equations subject to removal at the optimized equation set. Another significant remark is demonstrated in Figure 3, the computational graph of an equation system describing the motion of a free pendulum in the Cartesian space (cf. (Fritzson, 2011)). The computational graph of the whole

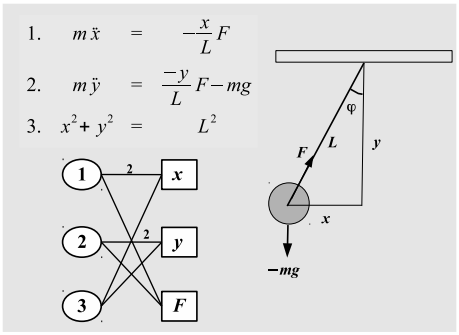

Figure 3. Pendulum equation system

sensitivity system is shown in Figure 4.

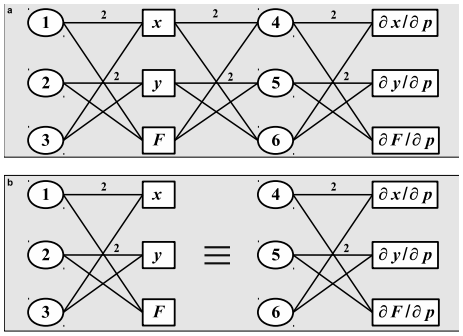

Figure 4. Computational graphs of pendulum equation sensitivity system

Obviously both computational graphs of the equation system and its sensitivity subsystems are isomorphic. This argument is generally valid as one of the mathematical foundlings proven in (Elsheikh and Wiechert, 2018). Consequently one can utilize already existing compiler capabilities (Mattsson, 1995; Maffezzoni et al., 1996), including index reduction (Pantelides, 1988; Mattsson and Söderlind, 1993), to transform the sensitivity system into a solvable optimized format. In many cases, there is no need to apply Modelica compiler techniques to the sensitivity subsystems. For example, the structural index (Leitold and Hangos, 2001) of both systems are identical. The equations within a sensitivity subsystem selected

for differentiation towards index reductions are those corresponding to the equations within the original system selected for differentiation towards index reduction.

6.3 Simulation of Sensitivity Systems

Similarly, numerical integration of sensitivity equations with standard solvers can be slow for high-dimensional systems. By exploiting the structure of sensitivity systems, speed up can be achieved by exploiting the Modelica-friendly method described in (Dickinson and Gelinas, 1976). The original ODE / DAE system is numerically integrated together with a sensitivity subsystem corresponding to only one single active parameter. This computation is repeated for each active parameter. A generalization of this approach is to allow larger sensitivity subsystems corresponding to several active parameters. Hardware-oriented heuristics can be employed for selecting the optimal number of active parameters. Moreover, this approach is salable w.r.t. to parallelization (Elsheikh, 2015).

7 Summary and Outlook

This work summarizes some conducted efforts in the Modelica community to provide a functionality for computing DPS. In particular, equation-based AD allowing DPS to be present at the model level is highlighted. The study hints that these efforts may potentially converge to the integration of DPS facilities for Modelica. One possibility is to allow the operator *der* to have a second argument as a model parameter. From one side, this provides a uniform platform-independent solution for representing DPS. From the other side, many applications based on DPS are realizable. In order to promote the presence of DPS at Modelica level, a new Modelica library called PSTools is currently under development from which the example in Figure 1 is taken.

Representation of DPS at model level has been demonstrated in few application domains that are rather describing systems governed by continuous-time equation systems. While excessive triggering of events are not that common in the field of Systems Biology in comparison for instance with the field of Power Electronics, an absolute justification of *der(x,p)* demands a proper treatment of DPS in the presence of discontinuities and events. The mathematical foundation for DPS of hybrid systems have been extensively studied in literature which could be mapped to a proper Modelica-based treatment.

Acknowledgement

I would like to thank and acknowledge

- Jan Peter Axelsson (Vascaia AB, Stockholm, Sweden) for the model on which Figures 1 and 2 are based
- Hans Olsson (Dessault Systems, Lund, Sweden) for valuable feedback regarding *der(x,p)*

References

Åkesson, J., Braun, W., Lindholm, P., and Bachmann, B. (2012). Generation of sparse Jacobians for the function mock-up interface 2.0. In *Modelica'2012*, Munich, Germany.

Andersson, J., Houska, B., and Diehl, M. (2010). Towards a computer algebra system with automatic differentiation for use with object-oriented modelling languages. In *EOOLT'2010*, Oslo, Norway.

Barton, P. I. and Lee, C. K. (2002). Modeling, simulation, sensitivity analysis, and optimization of hybrid systems. *ACM Transactions on Modling and Computer Simulation (TOMACS)*, 12(4):256–289.

Braun, W., Kulshreshtha, K., Franke, R., Walther, A., and Bachmann, B. (2017). Towards adjoint and directional derivatives in FMI utilizing ADOL-C within OpenModelica. In *Modelica'2017*, Prague, Czech Republic.

Braun, W., Ochel, L., and Bachmann, B. (2011). Symbolically derived Jacobians using automatic differentiation - enhancement of the OpenModelica compiler. In *Modelica'2011*, Dresden, Germany.

Braun, W., Yances, S. G., Link, K., and Bachmann, B. (2012). Fast simulation of fluid models with colored jacobians. In *Modelica'2012*, Munich, Germany.

Cellier, F. E. (1991). *Continuous System Modeling*. Springer Verlag.

Dickinson, R. and Gelinas, R. (1976). Sensitivity analysis of ordinary differential equation systems - a direct method. *Journal of Computational Physics*, 21:123–143.

Elsheikh, A. (2012). ADGenKinetics: An algorithmically differentiated library for biochemical networks modeling via simplified kinetics formats. In *Modelica'2012*, Munich, Germany.

Elsheikh, A. (2014). Modeling parameter sensitivities using equation-based algorithmic differentiation techniques: The ADMSL.Electrical.Analog library. In *Modelica'2014*, Lund, Sweden.

Elsheikh, A. (2015). An equation-based algorithmic differentiation technique for differential algebraic equations. *Journal of Computational and applied Mathematics*, 281:135 – 151.

Elsheikh, A. and Kucherenko, S. (2019). Dynamic parameter sensitivities: Summary of applications – version 1.0. Technical report. to appear online.

Elsheikh, A., Noack, S., and Wiechert, W. (2008). Sensitivity analysis of Modelica applications via automatic differentiation. In *Modelica'2008*, Bielefeld, Germany.

Elsheikh, A. and Wiechert, W. (2012). Accuracy of parameter sensitivities of DAE systems using finite difference methods. In *MATHMOD 2012*, Vienna, Austria.

Elsheikh, A. and Wiechert, W. (2018). The structural index of sensitivity equation systems. *Mathematical and Computer Modelling of Dynamical Systems*, 24(6):553–572.

Feehery, W. F., Tolsma, J. E., and Barton, P. I. (1997). Efficient sensitivity analysis of large-scale differential-algebraic systems. *Applied Numerical Mathematics*, 25(1):41–54.

Fell, D. (1992). Metabolic control analysis: a survey of its theoretical and experimental development. *Biochemocal Journal*.

Fritzson, P. (2011). *Introduction to Modeling and Simulation of Technical and Physical Systems with Modelica*. Wiley-IEEE Press, 1 edition.

Galán, S., Feehery, W. F., and Barton, P. I. (1999). Parametric sensitivity functions for hybrid discrete/continuous systems. *Applied Numerical Mathematics*, 31(1):17 – 47.

Hannemann-Tamas, R., Tillack, J., Schmitz, M., Förster, M., Wyes, J., Nöh, K., on Lieres, E., Naumann, U., Wiechert, W., and Marquardt, W. (2012). First and second order parameter sensitivities of a metabolically and isotopically non-stationary biochemical network model. In *Modelica'2012*, Munich, Germany.

Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., and Woodward, C. S. (2005). Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, 31(3):363–396.

Kucherenko, S. and Iooss, B. (2016). *Derivative-Based Global Sensitivity Measures*, pages 1–24. Handbook of Uncertainty Quantification, Springer International Publishing, Cham.

Leitold, A. and Hangos, K. M. (2001). Structural solvability analysis of dynamic process models. *Computers & Chemical Engineering*, 25:1633–1646.

Maffezzoni, C., Girelli, R., and Lluka, P. (1996). Generating efficient computational procedures from declarative models. *Simulation Practice and Theory*.

Maly, T. and Petzold, L. R. (1996). Numerical methods and software for sensitivity analysis of differential-algebraic systems. *Applied Numerical Mathematics: Transactions of IMACS*, 20(1–2):57–79.

Mattsson, S. E. (1995). Simulation of object-oriented continuous time models. *Mathematics and Computers in Simulation*, 39(5 – 6):513 – 518.

Mattsson, S. E. and Söderlind, G. (1993). Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal on Scientific Computing*, 14(3):677 – 692.

Murota, K. (1987). *Systems Analysis by Graphs and Matroids*. Springer, Berlin.

Naumann, U. (2012). *The Art of Differentiating Computer Programs, an Introduction to Algorithmic Differentiation*. SIAM.

Olsson, H., Tummescheit, H., and Elmqvist, H. (2005). Using automatic differentiation for partial derivatives of functions in Modelica. In *Modelica'2005*, Hamburg, Germany.

Pantelides, C. C. (1988). The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231.

Petzold, L., Li, S. T., Cao, Y., and Serban, R. (2006). Sensitivity Analysis of Differential-Algebraic Equations and Partial Differential Equations. *Computers & Chemical Engineering*, 30:1553–1559.

Saccon, A., van de Wouw, N., and Nijmeijer, H. (2014). Sensitivity analysis of hybrid systems with state jumps with application to trajectory tracking. In *53rd IEEE Conference on Decision and Control*, pages 3065–3070.

Saltelli, A., Tarantola, S., Campolongo, F., and Ratto, M. (2004). *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models*. Halsted Press, New York, NY, USA.

Wiechert, W., Noack, S., and Elsheikh, A. (2010). Modeling languages for biochemical network simulation: Reaction vs equation based approaches. *Advances in Biochemical Engineering / Biotechnology*, 121:109 – 138.

Wolf, S., Haase, J., Clauß, C., Jöckel, M., and Lösch, J. (2008). Methods of sensitivity calculation applied to a multi-axial test rig for elastomer bushings. In *Modelica'2008*, Bielefeld, Germany.

DOI
10.3384/ecp19157795