Modeling Parameter Sensitivities via Equation-based Algorithmic Differentiation Techniques: The ADMSL.Electrical.Analog Library

Atiyah Elsheikh
Austrian Institute of Technology, Vienna, Austria
Atiyah.Elsheikh@ait.ac.at

Abstract

Parameter sensitivities of mathematical models play a vital rule in many applications of sensitivity analysis. The availability of algorithmic capabilities for representing and computing these quantities is surly advantageous. In this work it is shown how to systematically transform a Modelica library to another library that describes the desired models together with derivatives of model variables w.r.t. model parameters. The produced library remains with the same structure and the underlying models keep the same interface and outlook. The proposed approach relies on novel equation-based algorithmic differentiation techniques that are especially designed for Modelica. The illustration is rather done via a compact library, the open-source ADMSL library, but rich enough to facilitate a lot of representative Modelica language constructs. The ADMSL library is the algorithmically differentiated version of the standard Modelica library subpackage `Modelica.Electrical.Analog.Basic`.

Keywords: algorithmic differentiation; parameter sensitivities; sensitivity analysis, ADMSL, AD of Modelica libraries

1 Motivation to algorithmic differentiation

If you want to draw Bamboos, you should try drawing Bamboos for your whole life, then you might be able to draw Bamboos, an ancient Chinese wisdom. In other words, the earliest modelers have already recognized that there is no model that identically describes the reality up to the tiniest details. This seems to be also the case now days, at least in the field of Systems Biology [27]. However even an elementary model describing a complex system is an essential initial step towards gaining insights and winning additional knowledge of the modeled system. Via e.g. the availability of further experimental data, the model can be better tuned [5]. Nevertheless, instead of trying to approach a true model for one's whole life, many tools of sensitivity analysis, such as model identification, validation and optimization [11, 12] can help the modelers to realize their visions, hopefully in a reasonable amount of time.

Significant quantities, that can assist the implementation of such computational tools, are parameter sensitivities, i.e. the derivatives of model outputs w.r.t. model parameters. Straightforward ways for evaluating these quantities via finite difference methods are not recommended for accuracy reasons [16]. A more reliable but technically difficult approach is to symbolically derive these quantities and then to evaluate them via a numerical integrator, e.g. the IDAS solver within the Sundials Suite [21].

These factors among other typical applications of mathematical derivatives increasingly attract attention at a special domain of scientific computing called Algorithmic Differentiation (AD) of computer programs [19]. Based on the chain rule of Calculus, procedural compiler methods and other established techniques, many automatic differentiation tools (cf. `www.autodiff.org`) are capable of:

- analyzing a large set of computer programs written in many procedural programming languages
- adequately computing new programs additionally representing partial derivatives

The resulting generated programs are typically used for evaluating the derivatives of program outputs w.r.t. program inputs among other directional derivatives.

2 Introduction

AD and the Modelica community

While classical AD techniques and tools are targeting a large scope of developers of mathematical programs, only a tiny part of the Modelica community is explicitly utilizing AD methods, namely tool vendors and developers of Modelica simulation environments. For instance, in [25, 4, 7] the Jacobian is symbolically computed for the task of index reduction and adaptive numerical integration. For the more difficult task of computing parameter sensitivities, some simulation environments such as JModelica [3] and SystemModeler [2] can be used. A translated Modelica model is linked with advanced specialized integrators suite like SUNDIALS capable of evaluating parameter sensitivities [21]. In this context, AD techniques are usually applied at low level C code describing the translated equations.

In contrary, the approach of applying AD techniques directly to the high-level descriptive model represents another attractive option. An example is given by the tool ADModelica [15, 14], which is capable of transforming a given high-level Modelica model into another Modelica model additionally describing parameter sensitivities. Similar works have been also reported with other modeling languages [6, 20]. Nevertheless and even with the availability of open-source developer-oriented compiler tools like OpenModelica [26], the development and maintenance efforts for such a tool attempting to cope with a rich language like Modelica becomes a continuously exhaustive task.

Contribution

For the first time an equation-based modeling-oriented AD approach for differentiating Modelica libraries is demonstrated. The approach provides the basic guidelines for systematically transforming a library into a topologically-identical algorithmically differentiated library in which parameter sensitivities are additionally represented. By reimporting the augmented library into already existing base models and slightly altering the declaration for specifying the required model parameters w.r.t. which derivatives are sought, parameter sensitivities is represented at the model level. In this sense, transformed library components are overloaded with semantics for describing parameter sensitivities. Using an arbitrary Modelica simulation environments, parameter sensitivities are directly simulated within the model.

Due to the speciality of the Modelica language and its significant difference from assignment-based procedural languages for which classical AD techniques were designed, new specialized equation-based AD techniques are demonstrated in this work. These techniques, relying on equation-based compiler notions, utilize Modelica powerful capabilities to provide efficient representation of partial derivatives. Consequently, the inclusion of parameter sensitivities within Modelica libraries can be considered from the early design phase or alternatively, existing libraries can be augmented with additional components for describing the required derivatives. In this sense, AD techniques needs to be applied only once and the resulting library can be used for ever.

According to Naumann [24], to meister AD concepts, AD users should be first able to manually differentiate their own compact programs before applying AD tools. In this work, the presented techniques are intuitive enough to be systematically applied on manual basis even on sophisticated libraries with complex mathematical models, e.g. ADGenKinetics [10]. In this way, a larger part of the Modelica community can possess the art of self-handing AD techniques along their modeling activities. The presented approach describes the basic steps subject to automation and consideration within realistic Modelica-based AD tools. A comprehensive algorithmic specification of the demonstrated techniques is given in [13].

3 General scheme

The key idea of the presented approach is illustrated via the diagram presented in Figure 1. Without loss of generality, a model M with two connected components C_1 and C_2 is given. The components are mathematically described by Differential Algebraic Equations (DAEs) via the functions f_1 and f_2, respectively. x_i and p_i for $i \in \{1,2\}$ correspond to systems variables and model parameters, respectively. The function g describes a causal or an acausal connection relation between C_1 and C_2.

The algorithmically differentiated components C_1' and C_2' extend the components C_1 and C_2 with the underlying Sensitivity Equation Systems (SESs). A SES is obtained by *forward differentiation* of

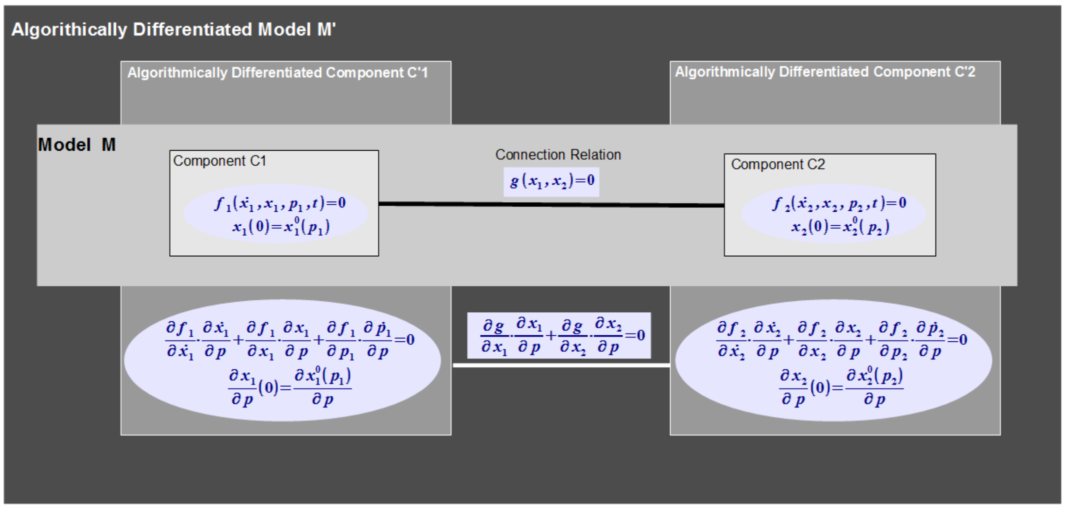
Algorithically Differentiated Model M'
Algorithmically Differentiated Component C'1
Algorithmically Differentiated Component C'2
Model M
Component C1
$f_1(\dot{x}_1, x_1, p_1, t)=0$
$x_1(0)=x_1^0(p_1)$
Connection Relation
$g(x_1, x_2)=0$
Component C2
$f_2(\dot{x}_2, x_2, p_2, t)=0$
$x_2(0)=x_2^0(p_2)$
$\frac{\partial f_1}{\partial \dot{x}_1}\cdot\frac{\partial \dot{x}_1}{\partial p}+\frac{\partial f_1}{\partial x_1}\cdot\frac{\partial x_1}{\partial p}+\frac{\partial f_1}{\partial p_1}\cdot\frac{\partial p_1}{\partial p}=0$
$\frac{\partial x_1}{\partial p}(0)=\frac{\partial x_1^0(p_1)}{\partial p}$
$\frac{\partial g}{\partial x_1}\cdot\frac{\partial x_1}{\partial p}+\frac{\partial g}{\partial x_2}\cdot\frac{\partial x_2}{\partial p}=0$
$\frac{\partial f_2}{\partial \dot{x}_2}\cdot\frac{\partial \dot{x}_2}{\partial p}+\frac{\partial f_2}{\partial x_2}\cdot\frac{\partial x_2}{\partial p}+\frac{\partial f_2}{\partial p_2}\cdot\frac{\partial p_2}{\partial p}=0$
$\frac{\partial x_2}{\partial p}(0)=\frac{\partial x_2^0(p_2)}{\partial p}$

Figure 1: A general block diagram of a model with two connected components and their algorithmically differentiated copies

the original DAE system w.r.t. the model parameters $p = (p_1\ p_2)^T$. The component C_1' additionally declares the *input Jacobian* of the parameters $\partial p_1/\partial p$, i.e. the derivatives of the local *active parameters* within C_1 w.r.t. the parameters of the whole model p. In general, the model parameters are of course not known within the separate component C_1'. Therefore, input Jacobian J_p is declared w.r.t. arbitrary model parameters p that are first defined at the model level. An example is given in the next Section. Analogously C_2' specifies the active local parameters within C_2 via their input Jacobian w.r.t. arbitrary unknown parameters.

An algorithmically differentiated model M' is obtained by connecting the components C_1' and C_2'. The connection relation is similarly described by forward differentiation of the equation system g w.r.t. p. The key idea is based on the chain rule of Calculus by which partial derivatives are propagated among components and hence parameter sensitivities of the whole model are inherently present.

There are many structural similarities between the model M and its AD version M' [17]:

1. Both models have the same interface in a GUI-editor, the SES remains hidden from the user perspective
2. The Jacobian of the differentiated equation system w.r.t. one single parameter has an identical structure and sparsity pattern of the Jacobian of the original system [23]
3. It can be proven that the structural indices of both models are equal

These characteristics can be utilized for simplifying the compilation phases within a Modelica compiler targeting AD of Modelica libraries and models. The next section provides a simple example illustrating the whole paradigm in Figure 1.

4 Illustrative example: The declaration part

4.1 The ADMSL library

Based on the techniques presented, an example illustrated on a subpackage of the Modelica standard library (MSL) `Modelica.Electrical.Analog.Basic`. The whole illustration is provided via the open-source `ADMSL` library [1]. The ADMSL library serves as an experimentation platform for:

1. Illustrating the basic steps for performing AD of Modelica libraries
2. Identifying the best capabilities of the Modelica language relevant for expressing parameter sensitivities at the library components level
3. Recognizing current limitations from expressibility perspective with which automatic code generation becomes less systematic

4.2 Code generation rules

In Naumann [24], basic code generation rules for performing AD of procedural languages have been presented. Analogously, in this paper simple code generation rules, however, for the Modelica language are demonstrated. These Modelica-specialized rules serve as main guidelines for performing AD of Modelica libraries. Note that the presented rules are distinguishably different than those provided for classical languages. For a complete understanding of the following subsections, the reader is recommended to consult the subpackage `Modelica.Electrical.Analog`.

Declaration rule 1: structure duplication

When performing AD of a Modelica library, all packages and subpackages are simply duplicated. For instance, the subpackage `ADMSL.Electrical.Analog` corresponds to the subpackage `Modelica.Electrical.Analog`. Within each package, model components with the same names are placed. Each model component extends the original components. For instance, the corresponding AD version of the connector `Pin` becomes:

Listing 1: AD version of `Pin`

```
connector Pin
  extends Modelica.Electrical.Analog.
          Interfaces.Pin;
  ...
end Pin;
```

For components declaring these `Pin`s, the corresponding AD version, e.g. of the `OnePort` component, should declare the corresponding AD versions of `Pin`s. This is done by redeclaring replaceable `Pin`s:

Listing 2: AD version of `OnePort`

```
partial model OnePort
  extends
    Modelica.Electrical.Analog.
    Interfaces.OnePort(
      redeclare ADMSL.Electrical.Analog.
                Interfaces.PositivePin p,
      redeclare ADMSL.Electrical.Analog.
                Interfaces.NegativePin n);
  ...
end OnePort;
```

The above code assumes that the electrical pins `p` and `n` are declared as `replaceable`. However, this is not the case in the latest Modelica library version 3.2. Alternatively, as a temporary solution, the original `TwoPort` model has been slightly modified and placed under `Analog.Interfaces.Bases` package as shown in Appendix A. Similarly, for models (e.g. `Capacitor`) extending partial models (e.g. `OnePort`), their AD versions need to extend the AD versions of these partial models, for example:

Listing 3: AD version of `Capacitor`

```
model Capacitor
  extends Bases.Capacitor(
    redeclare replaceable class
      OnePort = ADMSL.Electrical.Analog.
                  Interfaces.OnePort);
  ...
end Capacitor;
```

Declaration rule 2: duplication of data segments

Any new component needs to reference the global number of active parameters w.r.t. which derivatives are sought. This is done by extending the component `ADMSL.Interfaces.GradientInfo` declaring global gradient related information:

Listing 4: Declaring the number of gradients

```
outer parameter Integer NG
  "dimension of gradient";
```

The parameter is declared as `outer` utilizing implicit connection mechanisms. It gets first initialized only by explicit declaration defining the same parameter as `inner` at the top level model. Accordingly, for all model components a *derivative object* (i.e. the gradient) is additionally declared for each real variable or parameter within the corresponding component, e.g. the AD version of theconnector `Pin`:

Listing 5: AD version of `Pin`

```
connector Pin
  extends ... // as before
  extends ADMSL.Interfaces.GradientInfo;
    Real g_v[NG]
      "gradient of the voltage";
    flow Real g_i[NG]
      "gradient of the current";
end Pin;
```

The idea is simple, potential derivative objects are associated with variables while flow derivative objects are associated with flow variables. Similarly, in the model capacitor:

Listing 6: Declaration part of `Capacitor`

```
model Capacitor
  extends ...; // as before
  parameter Real g_C[NG] = zeros(NG);
  ...
end Capacitor;
```

Here, a derivative object initialized to the zero vector is associated with the parameter C. Each entry of a gradient represents the derivative w.r.t. a parameter specified via the input Jacobian at the top model. So far the main rules for altering the declaration part has been demonstrated. In the next section, further rules for deriving the sensitivity equations are illustrated.

5 Illustrative example: The equation part

5.1 Modeling SES in gradient format

Assume that a given Modelica component C is described more or less DAE system (without discrete variables):

$$F(\dot{x},x,p,t)=0 \quad , \qquad x(t_0)=x_0(p) \qquad (1)$$

where $x(t) \in R^n$ and $p \in R^m$ represent state variables and parameters, respectively. Additionally, assume that $F : R^{2n+m+1} \to R^n$ is continuously differentiable w.r.t. $\dot{x}, x$ and p. Required is an additional component C' that augments C with additional equations for describing the time-dependent parameter sensitivities $(\partial x/\partial p)(t) \in R^{n\times m}$. As mentioned before, C' needs to include the *sensitivity equation subsystems* (SESub):

$$F_{\dot{x}}\dot{s}_i + F_x s_i + F_{p_i} = 0 \quad , \qquad s_i(t_0) = \frac{\partial x_0(p)}{\partial p_i} \qquad (2)$$
$$\text{where} \qquad s_i = \frac{\partial x}{\partial p_i} \quad \text{for} \quad i=1,2,\ldots,m$$

SESub is obtained by differentiating all equations w.r.t. desired parameters. This is a large equation system of dimension $m \cdot n$. Explicit listing all these equations makes C' not compactly implemented.

To overcome this drawback, Modelica array capabilities can be utilized if the equation system 2 is implemented in gradient format. Within a model component corresponding to a DAE of the form (1), assuming that F_i corresponds to the i-th equation in F let F_x be defined as follows:

$$F_x = \left[\frac{\partial F_i}{\partial x_j} : i,j = 1,2,\ldots,n\right] \in R^{n\times n}$$

and let $F_{\dot{x}}, F_p, \dot{x}_p$ and x_p be analogously defined. Furthermore, let the input Jacobian J_p be defined as follows:

$$J_p = \left[\frac{\partial p_i}{\partial p_j} : i,j = 1,2,\ldots,m\right] \in R^{m\times m}$$

A typical case is to set $J_p = I_m$, the identity matrix, for a set of independent parameters. Then the corresponding differentiated equation w.r.t. a parameter p_j is derived from Equation (2) as follows:

$$\sum_{l=1}^{m}\left[\sum_{k=1}^{n}[F_{\dot{x}}(i,k)\,\dot{x}_p(k,l)] + \sum_{k=1}^{n}[F_x(i,k)\,x_p(k,l)] + \sum_{k=1}^{m}[F_p(i,k)\,J_p(k,l)]\right]J_p(l,j) = 0 \qquad (3)$$

Using Modelica array capabilities and assuming that the given set of parameters is independent (i.e. $\partial p_i/\partial p_j = 0$ for $i \neq j$), Equation (3) can be rewritten in a gradient format comprising m equations:

$$\sum_{k=1}^{n}[F_{\dot{x}}(i,k)\,\dot{x}_p(k,:)] + \sum_{k=1}^{n}[F_x(i,k)\,x_p(k,:)] + F_p(i,:) = 0 \qquad (4)$$

for $i = 1,2,\ldots,n$ where $A(i,:)$ represents the i-th row of a matrix A.

5.2 Code generation rules

Forward differentiation rule 3: deriving sensitivity equations

Using Equation (4), SES of simple model components can be easily implemented. For instance the equation part of the component `ADMSL.Analog.Basic.Capacitor` becomes:

Listing 7: Equation part of `Capacitor`

```
model Capacitor
  extends ...;
  ...
equation
  g_i[1:NG] = g_C[1:NG] * der(v) +
              C * der(g_v[1:NG]);
end Capacitor;
```

Deriving SES for simple mathematical formulas like the previous one is straightforward. For long complex formulas, common computer algebra packages can be used. However, it is recommendable to employ the equation-based AD techniques illustrated in Section 6.

Forward differentiation rule 4: handling blocks within flow control

Modelica, as many other languages, provides classical language constructs for flow control such as `for`, `while`, `if`,..etc. In classical AD concepts, assignment blocks within typical control flow constructs (e.g. for, if, ...etc.) are directly differentiated as an independent block. Although such constructs are interpreted differently in an equation-based context, the generation of SES within such blocks are similarly intuitive. For example, the equation part of the model `Analog.Examples.Utilities.NonlinearResistor` is implemented as follows:

Listing 8: Equation part of `NonlinearResistor`

```
model NonlinearResistor
...
equation
  i =
    if (v < -Ve) then
      Gb*(v + Ve) - Ga*Ve
    else if (v > Ve) then
      Gb*(v - Ve) + Ga*Ve
    else
      Ga*v;
end NonlinearResistor;
```

The implementation of its AD version becomes:

Listing 9: Equation part of `NonlinearResistor`

```
model NonlinearResistor
...
equation
  g_i[:] = if (v < -Ve) then
    g_Gb[:] * (v + Ve) +
    Gb * (g_v[:] + g_Ve[:])
    - (g_Ga[:] * Ve + Ga * g_Ve[:]);
 else if (v > Ve) then
    ...
 else
   ...;
end NonlinearResistor;
```

6 Equation-based AD

So far, the demonstrated components have short equations that can be easily differentiated. In this section, an equation-based AD technique, especially designed for the Modelica language, is shown. In the first ever algorithmically differentiated library, ADGenKinetics, the following equation corresponding to the convenience kinetics of chemical reaction rates

$$v = \prod_a \frac{K_{A_a} + [A_a]}{K_{A_a}} \cdot \prod_b \frac{K_{I_b}}{K_{I_b} + [I_b]} \cdot \frac{V_{max}^{fwd} \prod_i \frac{[S_i]}{K_{mS_i}} - V_{max}^{bwd} \prod_j \frac{[P_j]}{K_{mP_j}}}{\prod_i \left(1 + \frac{[S_i]}{K_{mS_i}}\right) + \prod_j \left(1 + \frac{[P_j]}{K_{mP_j}}\right) - 1} \qquad (5)$$

has been easily differentiated using the demonstrated technique, consult [10] for more details about this equation and the ADGenKinetics library. In this paper, the technique is illustrated on a more simple equation within the model `Analog.Basic.Conductor` which implements conductance as:

$$G_{actual} = \frac{G}{\left(1 + \alpha \left(T_{hp} - T_{ref}\right)\right)} \qquad (6)$$

While this formula can be used within a Calculus exam, the technique is applicable on formulas like (5).

6.1 Classical AD techniques

The fundamental terminologies and concepts of classical AD techniques for assignment-based procedural languages have been largely developed decades ago Application of classical AD techniques is not best suitable to be applied on equation-based languages due to [15, 13]:

- Classical AD techniques are mainly designed for explicit assignments and not implicit equations
- Excessive number of expressions evaluations is performed
- Excessive storage for temporary variables and their gradient computations is needed

In order to overcome these drawbacks, an equation-based technique relying on fundamental notions of equation-based languages has been designed.

6.2 Equation-based AD technique

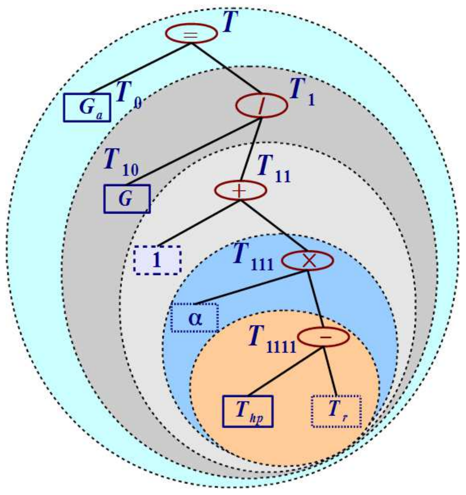

Figure 2: The AST of Equation 6 and the enumeration of expression subtrees

Preprocessing: Intermediate elementary equations

The key idea of enabling differentiation of a complex equation is to decompose it into *elementary equations*. Each equation is composed of a unitary operator (e.g. $+,-$), a binary operator (e.g. $+,-,*,\prod$) or an intrinsic function (e.g. $\sin, \cos$). By constructing the abstract syntax tree (AST) of the equation, elementary equations can be computed. In Figure 2, non-leaf nodes correspond to operands. Leaf nodes correspond to identities. Identities are distinguished according to their variability, e.g. constants, parameters and variables. Each subtree T_k is intuitively indexed with a binary number k according to its position in the tree. Each node with index k has childern of index $2k$ and $2k+1$, if any. Intermediate equations and their derivatives can be easily computed via a left-right-node (LRN) traversal as follows:

$$
\begin{aligned}
T_{1111} &= T_{hp} - T_{ref} \\
\Longrightarrow T'_{1111} &= T'_{hp} - T'_{ref} \\
T_{111} &= \alpha\, T_{1111} \\
\Longrightarrow T'_{111} &= \alpha'\, T_{1111} + \alpha\, T'_{1111} \\
1/T_{11} &= 1 + T_{111} \\
\Longrightarrow T'_{11} &= -T'_{111}\, T_{11}\, T_{11} \\
T_1 &= G\, T_{11} \\
\Longrightarrow T'_1 &= G'\, T_{11} + G\, T'_{11} \\
\Longrightarrow G'_a &= T'_1
\end{aligned}
$$

where

$$
T'_k = \left(\frac{\partial T_k}{\partial p_1}, \frac{\partial T_k}{\partial p_2}, \ldots, \frac{\partial T_k}{\partial p_m} \right)
$$

Here, only intermediate equations corresponding to non-leaf nodes are considered. Moreover, Modelica capabilities are utilized for providing specialized treatment of the division operator, the most expensive arithmetic operator. This is done in a way that the let derivatives don't include further divisions.

Processing: Accumulation

The previous intermediate equations though represent desired partial derivatives, however it requires a lot of storage for T'_k. To overcome this drawback one can rather iteratively accumulate the partial derivatives, within the same LRN traversal of the AST. On a manual basis, this corresponds to an iterative process of copy and paste of the intermediate equations one after another. In this way, the *accumulated intermediate equations* become:

$$
\begin{aligned}
T'^{*}_{1111} &= T'_{hp} - T'_{ref} \\
T'^{*}_{111} &= \alpha'\, T_{1111} + \alpha\, (T'_{hp} - T'_{ref}) \\
T'^{*}_{11} &= -(\alpha'\, T_{1111} + \alpha\, (T'_{hp} - T'_{ref}))\, T_{11}\, T_{11} \\
T'^{*}_{1} &= G'\, T_{11} \\
&+ G\, (-(\alpha'\, T_{1111} + \alpha\, (T'_{hp} - T'_{ref}))\, T_{11}\, T_{11}) \\
\Longrightarrow G'_a &\equiv T'^{*}_1
\end{aligned}
$$

Now, it is enough to declare only one derivative object for G_a.

Postprocessing: Common subexpressions

The accumulated intermediate equations contain multiplicative term that are going to be evaluated m times. Within the same LRN-traversal, these multiplicative terms (e.g. $\alpha\, T_{11}\, T_{11}$) can be rather stored in an a local variable. These terms are stored in local variables `adl_*` within the `Conductor` model:

Listing 10: The AD version of the `Conductor` model

```
model Conductor
  extends ...;
  ...
protected
  Real T_1111;
  Real T_111;
  Real D_11;
  Real adl_11_1;
  Real adl_11_2;
  Real adl_11_3;
  Real adl_1_1;
  Real adl_1_2;
equation
  T_1111 = T_heatPort - T_ref;
  T_111 = alpha * T_1111;
  1/D_11 = 1 + T_111;
  adl_11_1 = - D_11 * D_11;
  adl_11_2 = adl_11_1 * alpha;
  adl_11_3 = T_111 * adl_11_1;
  adl_1_1 = G * adl_11_3;
  adl_1_2 = G * adl_11_2;
  g_G_actual[:] = g_G[:] * D_11 +
    adl_1_1 * g_alpha[:]  + adl_1_2
      * (g_T_heatPort[:]-g_T_ref[:]);
  g_i[:] = g_G_actual[:] * v
    + G_actual * g_v[:];
  g_LossPower[:] = g_v[:] * i
    + v * g_i[:];
end Conductor;
```

This is an optional step, particularly, if the used Modelica compiler is capable of recognizing common subexpressions and treating them adequately. However this step still can be used for coming up with an efficient compact representation of the partial derivatives.

7 Illustrative Example: The top model

Having performed equation-based AD of the `Electrical.Analog` sub-package, then parameter sensitivities of the electrical circuit in `Analog.Basic.Example.ChuaCircuit` are implicitly represented by simple slight modification:

- reimporting the AD version of the library and
- specifying the input Jacobian

Listing 11: The AD version of the `ChuaCircuit` model

```
model ChuaCircuit
  "Chua's circuit, ns, V, A"
  // import Modelica.Electrical.
             Analog.Basic;
  import ADMSL.Electrical.Analog.Basic;
  // import Modelica.Electrical.Analog.
  //         Examples.Utilities;
  import ADMSL.Electrical.Analog.
          Examples.Utilities;
  import Modelica.Icons;
  extends Icons.Example;
  import ADMSL.Utilities.*;
  inner parameter Integer NG = 8;
  Basic.Inductor L(L=18,
                 g_L=unitVector(1,NG));
  Basic.Resistor Ro(R=12.5e-3,
                 g_R=unitVector(2,NG));
  Basic.Conductor G(G=0.565,
                 g_G=unitVector(3,NG));
  Basic.Capacitor C1(C=10, v(start=4),
                 g_C=unitVector(4,NG));
  Basic.Capacitor C2(C=100,
                g_C=unitVector(5,NG));
  Utilities.NonlinearResistor Nr(
    Ga(min=-1) = -0.757576,
      g_Ga = unitVector(6,NG),
    Gb(min=-1) = -0.409091,
      g_Gb = unitVector(7,NG),
    Ve=1,
      g_Ve = unitVector(8,NG));
  Basic.Ground Gnd
equation
  // same as before
  ...
end ChuaCircuit;
```

Figures 3 and 4 show some results of the parameter sensitivities.

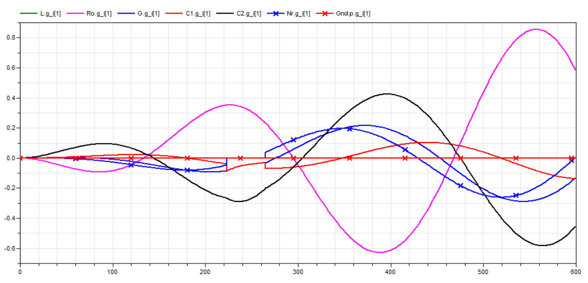

Figure 3: The sensitivities of the current at all components (i.e. `L.i,Ro.i,G.i,C1.i,C2.i,Nr.i,Gr.i`) w.r.t. the inductance `L.L`

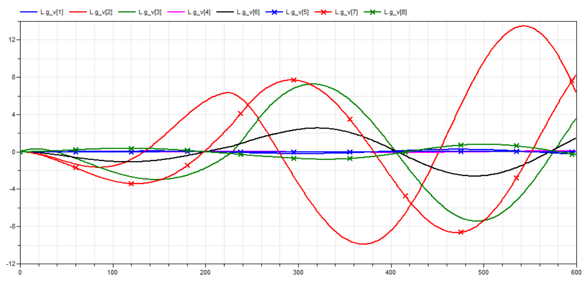

Figure 4: The sensitivities of `L.v` w.r.t. all parameters (i.e. `L.L,Ro.R,G.G,C1.C,..` etc.)

8 Summary and outlook

In this paper, an equation-based methodology for AD of Modelica libraries has been comprehensively demonstrated. The new equation-based AD techniques have been designed for the Modelica language. With few simple code generation rules, parameter sensitivities of base models are additionally described. These rules make use of the art of AD to let modelers themselves be capable of manually differentiating large-set of models. Once a library is already differentiated for once, it can be used forever. The whole work serves as an experimentation platform for the implementation of AD tool. Already a lot of the functionalities are implemented within the tool ADModelica [14].

Further ongoing works are running on several dimensions:

- Ensuring that all used terminology is conform to the Modelica specification
- choosing the most proper constructs allowing further extension of this library to be compilable with arbitrary simulation environments

- Implementing unit tests not only for single components but also to ensure the correctness of intermediate components, processed and postprocessed classes
- Implementing further useful sensitivity-related functionalities, e.g. scaled sensitivities, finite difference functions, etc.
- Utilizing Modelica capabilities for operator overloading

Finally, there are further two issues that are worth to mention. First, although normal numerical solvers not especially designed for sensitivity analysis are used, the expensive computation of parameter sensitivities can be reduced by following a numerical integration approach based on Dicknson [8]. Smaller subsets of parameter sensitivities are considered one after another rather than considering the whole SES [9]. The structure of the proposed gradient-oriented code easily suggests that. Moreover, this approach is parallelizable in a scalable manner [22].

Second, the presented approach in this work is restricted to continuous-time based components by which large set of already existing libraries can be considered. However, theorems and concepts for handling discontinuous functions and hybrid systems already exist [18], out of which further extensions to this work can be considered.

A Some modified components in the `Modelica.Electrical.Analog` library

`ADMSL.Electrical.Analog.Interfaces.Bases.OnePort`

The `TwoPin` model with the MSL is slightly modified by letting the declared connectors become `replaceable` as follows:

Listing 12: Implementation of `OnePort`

```
partial model OnePort
  "Component with two electrical pins"
  Modelica.SIunits.Voltage v
    "Voltage drop between the two pins";
  Modelica.SIunits.Current i
    "Current flowing from pin p to pin n";
  replaceable Modelica.Electrical.Analog.
    Interfaces.PositivePin p
      constrainedby
        Modelica.Electrical.Analog.
          Interfaces.PositivePin;
  replaceable Modelica.Electrical.Analog.
    Interfaces.NegativePin n
      constrainedby
        Modelica.Electrical.Analog.
          Interfaces.NegativePin;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end OnePort;
```

`ADMSL.Electrical.Analog.Interfaces.Bases.TwoPin`

The `TwoPin` model with the MSL is slightly modified by letting the declared connectors become `replaceable` as follows:

Listing 13: Implementation of two pins

```
partial model TwoPin
  "Component with two electrical pins"
  Modelica.SIunits.Voltage v
    "Voltage drop between the two pins";
  replaceable Modelica.Electrical.Analog.
    Interfaces.PositivePin p
      constrainedby
        Modelica.Electrical.Analog.
        Interfaces.PositivePin;
  replaceable Modelica.Electrical.Analog.
    Interfaces.NegativePin n
      constrainedby
        Modelica.Electrical.Analog.
        Interfaces.NegativePin;
equation
  v = p.v - n.v;
end TwoPin;
```

References

[1] The ADMSL library. https://github.com/AIT-CES-LAB/ADMSL.

[2] Wolfram SystemModeler. http://www.wolfram.com/system-modeler/.

[3] J. Åkesson, K.-E. Årzén, M. Gäfvert, T. Bergdahl, and H. Tummescheit. Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem. *Computers and Chemical Engineering*, 34(11):1737–1749, 2010.

[4] J. Andersson, B. Houska, and M. Diehl. Towards a computer algebra system with automatic differentiation for use with object-oriented modelling languages. In *EOOLT'2010: The 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, Oslo, Norway, Oct. 2010.

[5] I. Bauer, H. G. Bock, S. Körkel, and J. P. Schlöder. Numerical methods for optimum experimental design in DAE systems. *Journal of Computational and Applied Mathematics*, 120:1 – 25, 2000.

[6] C. H. Bischof, H. M. Bücker, W. Marquardt, M. Petera, and J. Wyes. Transforming equation-based models in process engineering. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, Lect. Notes in Comp. Sc. and Eng., pages 189–198. Springer, 2005.

[7] W. Braun, L. Ochel, and B. Bachmann. Symbolically derived Jacobians using automatic differentiation - enhancement of the OpenModelica compiler. In *Modelica'2011: The 8th International Modelica Conference*, Dresden, Germany, Mar. 2011.

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

[9] A. Elsheikh. *Modelica-based computational tools for sensitivity analysis via automatic differentiation*. Dissertation, RWTH Aachen university, Aachen, Germany, 2011.

[10] A. Elsheikh. ADGenKinetics: An algorithmically differentiated library for biochemical networks modeling via simplified kinetics formats. In *Modelica'2012: The 9th International Modelica Conference*, number 076 in Linköping Electronic Conference Proceedings, pages 915 – 926, Munich, Germany, Sep. 2012.

[11] A. Elsheikh. Assisting identifiability analysis of large-scale dynamical models with decision trees: DecTrees and InteractiveMenus. In *EuroSim'2013: The 8th EUROSIM Congress on Modelling and Simulation*, pages 300 – 305, Cardiff, Wales, UK, Sep. 2013.

[12] A. Elsheikh. Derivative-based hybrid heuristics for continuous-time simulation-optimization. *Simulation Modelling Practice and Theory*, 2013. In Press.

[13] A. Elsheikh. An equation-based algorithmic differentiation technique for differential algebraic equations. *Computational and applied Mathematics*, 201x. Submitted. Available online as a technical report.

[14] A. Elsheikh, S. Noack, and W. Wiechert. Sensitivity analysis of Modelica applications via automatic differentiation. In *Modelica'2008: The 6th International Modelica Conference*, volume 2, pages 669–675, Bielefeld, Germany, March 2008.

[15] A. Elsheikh and W. Wiechert. Automatic sensitivity analysis of DAE-systems generated from equation-based modeling languages. In C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, pages 235–246. Springer, 2008.

[16] A. Elsheikh and W. Wiechert. Accuracy of parameter sensitivities of DAE systems using finite difference methods. *Mathematical Modelling, IFAC Papers Online*, 7(1):136 – 142, Feb. 2012. Presented in MATHMOD'2012, The 7th Vienna International Conference on Mathematical Modelling, Vienna, Austria.

[17] A. Elsheikh and W. Wiechert. A structure-preserving approach for sensitivity analysis of higher index differential algebraic equations. *Mathematics and Computers in Simulation*, 201x. Submitted.

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

[19] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.

[20] A. K. Gupta and S. A. Forth. An AD-enabled optimization toolxbox in LabVIEW$^{\text{TM}}$. In S. Forth, P. Hovland, E. Phipps, J. Utke, and A. Walther, editors, *Recent Advances in Algorithmic Differentiation*, volume 87 of *Lecture Notes in Computational Science and Engineering*, pages 285–295. Springer-Verlag Berlin Heidelberg, 2012.

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

[22] X. Ke. Tools for sensitivity analysis of modelica models. Master's thesis, Siegen University, Germany, 2009.

[23] S. Li, L. Petzold, and W. Zhu. Sensitivity analysis of differential-algebraic equations: A comparison of methods on a special problem. *Applied Numerical Mathematics: Transactions of IMACS*, 32(2):161–174, Feb. 2000.

[24] U. Naumann. *The art of Differentiating Computer Programs, an Introduction to Algorithmic Differentiation*. SIAM, 2012.

[25] H. Olsson, H. Tummescheit, and H. Elmqvist. Using automatic differentiation for partial derivatives of functions in Modelica. In *Modelica'2005: The 4th International Modelica Conference*, Hamburg, Germany, 2005.

[26] M. Sjölund and P. Fritzson. An openmodelica java external function interface supporting metaprogramming. In *Modelica'2009: The 7th International Modelica Conference*, Como, Italy, 2009.

[27] W. Wiechert and R. Takors. *Validation of Metabolic Models: Concepts, Tools, and Problems*. Taylor & Francis, 2004.