IMPLEMENTATION OF EXACT SENSITIVITIES IN A CIRCUIT SIMULATOR USING AUTOMATIC DIFFERENTIATION

Carlos E. Christoffersen
Department of Electrical Engineering,
Lakehead University Thunder Bay, Ontario, Canada P7B 5E1
E-mail: c.christoffersen@ieee.org

Keywords— **circuit simulation, sensitivity analysis, automatic differentiation, object-oriented design**

Abstract— **Sensitivities are very important in electronic circuit analysis and design. This paper presents a general circuit simulation programme that calculates analytical sensitivities with respect to any parameter using automatic differentiation. A novel aspect of this implementation is that C++ templates are used along with automatic differentiation libraries to produce (at compilation time) different versions of the model evaluation functions, each optimised for a specific purpose. This results in a good compromise between the time to develop and maintain device models and execution efficiency. Algorithms and software design aspects of the circuit simulator are described. Sensitivities of a bipolar self-biasing current source are used to demonstrate the approach.**

I. INTRODUCTION

Optimisation algorithms are widely used for electronic circuit design. Some of these algorithms require the partial derivatives of certain network functions with respect circuit parameters (sensitivities) [11]. Other reasons to calculate sensitivities [5] are to understand how the parameters in a circuit affect the response and to compare circuits with the same nominal output. The simplest sensitivity (D_h^Φ) definition is just the partial derivative of a circuit function (Φ) with respect to a circuit parameter (h),

$$D_h^\Phi = \frac{\partial \Phi}{\partial h}.$$

The sensitivity calculation for different types of circuit analyses has been studied extensively (some examples are [5, 11, 12], among many others). In order to calculate any type of sensitivity, the derivatives of the device model model equations are required. Manually coding these derivatives is often a tedious and error-prone task due to the complexity of device models. A common alternative is to use numerical differences to approximate the derivatives. This approach is often unreliable because the size of the optimum numerical increment for each independent variable is not always known. Inaccuracy (especially for higher-order derivatives) and a decrease in the convergence rate of numerical methods due to numerical differences has been reported in [1–4, 8]. To the knowledge of the author all implementations in available circuit simulators involve numerical approximations of the derivatives at the device level mainly because it is not practical to implement all the required derivative functions at the nonlinear device models. The two remaining alternatives to calculate the device model derivatives are to use symbolic derivatives generated by tools such as Maple and Mathematica or to use automatic differentiation (AD). AD is clearly the best alternative since the function to be differentiated must be implemented as a programme code, often with many intermediate variables and subroutine calls [2, 8]. The main idea behind AD is that instead of differentiating formulae, the actual source code that calculates the functions is differentiated. References [2, 3, 8] provide illustrative examples on how the chain rule is used to propagate derivatives.

In [2,3,6,7] AD was used to compute derivatives used to solve the nonlinear equations in a circuit simulation. References [6, 7] describe general circuit simulators. In [6] it is mentioned that AD can also be used for sensitivity calculation, but design details and calculation results for sensitivities are not presented. References [2] and [3] describe tools for the analysis of electric power systems. In [2] AD is also used for sensitivity calculation.

Several AD tools have been developed in the last 10 years (see http://www.autodiff.org). Some of these tools create a very efficient code to calculate derivatives but they are not freely available or require human intervention. In this work we will concentrate on C++ AD tools that operate at compile [9] or run time [8]. The implementation of AD with this type of library frequently results in some overhead compared with the original code because additional steps are required [8,9]. In this paper we present a design approach that avoids this overhead and allows the computation of the exact sensitivities without increasing the complexity of nonlinear model implementation. This is achieved by the use of C++ expression templates.

The simulator programme described in this paper (named *Carrot*) is under active development and is available by contacting the author. Currently DC, DC sensitivities and transient analysis are supported. In the following sections we describe how sensitivities are calculated. Then the details of the implementation are given. This is followed by an example were sensitivities obtained with Carrot are compared with numerical sensitivities obtained using the well-known Spice simulator.

II. CIRCUIT EQUATION

In the following derivation we adopt the current source approach described in [10]. Individual circuit components are represented internally as a combination of voltage-controlled current sources (VCCS) and independent current sources (ICS). This approach al-

lows the description of any circuit without any loss of generality. Let the circuit be represented by its nodal equations:

$$Gu(t) + C\frac{du(t)}{dt} + \frac{dQ(u(t))}{dt} + I(u(t)) = S(t), \quad (1)$$

here $u(t)$ is the vector of nodal voltages, G is a matrix of conductances, C is the matrix representing the linear charge terms, $Q(u(t))$ and $I(u(t))$ are nonlinear vector functions corresponding to the nonlinear devices and $S(t)$ is a vector that represents the sources.

In this paper we consider the DC sensitivity calculation. Thus, Eq. (1) is simplified to

$$Gu + I(u) = S. \quad (2)$$

Eq. (2) is solved using Newton's method. The J_j matrix represents the Jacobian matrix of $I(u_j)$, where j is the iteration number. The value of u_{j+1} is obtained by solving the following linear system:

$$[G + J_j]u_{j+1} = S - I(u_j) + J_j u_j. \quad (3)$$

Given some initial guess (u_0), Eq. (3) is iterated until convergence is achieved. Note that after the last iteration the decomposed $[G + J_j]$ matrix is available.

As mentioned before any circuit device can be decomposed into a set of VCCSs and ICSs. The ICSs contribute to the S vector in Eq. (3). It is convenient to divide the VCCSs into two categories: linear and nonlinear. The linear VCCSs contribute to the G matrix in Eq. (3) and the nonlinear VCCSs contribute to the J_j matrix and the $I(u_j)$ vector. The contributions of the nonlinear VCCSs must be recalculated at each Newton iteration.

III. CIRCUIT SENSITIVITIES

In the following derivation we assume that the parameter of interest (Φ) in the circuit can be obtained as a linear combination of the elements in the u vector, *i.e.*,

$$\Phi = d^T u,$$

were d is a column vector. The adjoint method [5] allows the efficient calculation of the sensitivities of Φ with respect to a parameter h. First, the vector of adjoint voltages is calculated by solving the following linear system:

$$[G + J]^T u_a = d.$$

Note that that, as mentioned before, the $[G + J]^T$ matrix does not need to be decomposed since we have its transposed matrix already decomposed from the solution of Eq. (2). The sensitivity ($\partial\Phi/\partial h$) is now calculated:

$$\frac{\partial \Phi}{\partial h} = u_a^T \left[\frac{\partial S}{\partial h} - \frac{\partial G}{\partial h} u - \frac{\partial I}{\partial h} \right]. \quad (4)$$

As expected, the calculation of the sensitivity with respect to a parameter h requires the derivatives of all the sources, conductances and nonlinear currents in the circuit with respect to h. The derivatives in Eq. (4) are normally approximated using numerical differences.

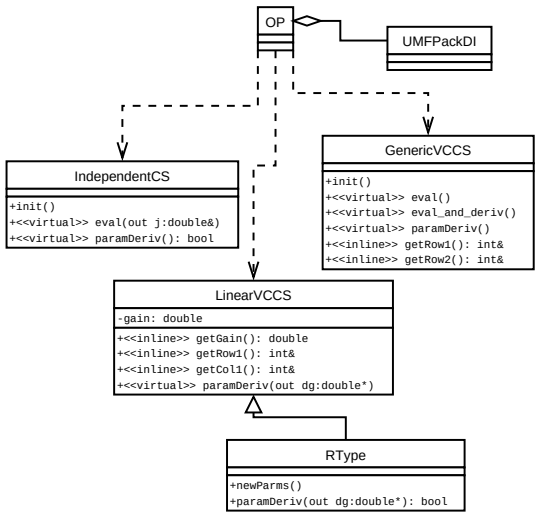

Fig. 1. UML Class Diagram Showing the Dependencies seen by the **OP** Class

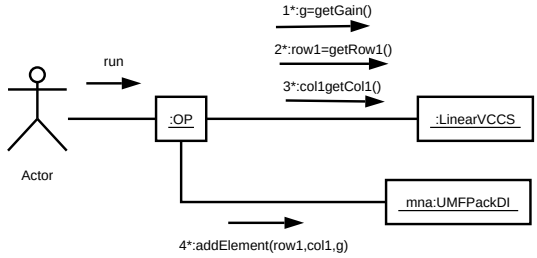

Fig. 2. UML Collaboration Diagram Showing the use of the **LinearVCCS** Class

IV. IMPLEMENTATION

Nonlinear and linear VCCSs, and ICSs are represented in the simulator by the **GenericVCCS**, **LinearVCCS** and **IndependentCS** classes, respectively. These classes provide the necessary methods (*i.e.*, interface) to build the respective parts of Eqs. (3) and (4). **OP** (operating point) is a class that implements the DC analysis using the Newton method as discussed before. The UML class diagram [13] in Fig. 1 depicts the dependences between these classes. By use of polymorphism [7] the **OP** class is independent of any particular device model. Note that *inline* functions are used where possible for efficiency. Virtual functions are necessary in functions that are different for each device model. The $[G + J_j]$ matrix is stored by the **UMFPackDI** class. This class encapsulates function calls to the UMFPack library [14].

In the collaboration diagram in Fig. 2 it is shown how the **LinearVCCS** is used by the **OP** to fill the $[G + J_j]$ matrix. To simplify the diagram only some of the messages sent to **LinearVCCS** are shown.

We now focus on the evaluation of derivatives. Many of the virtual methods of the classes in Fig. 1 require the evaluation of derivatives. These derivatives are obtained by means of AD tools. There are several AD tools available for the C++ language. The tools can be divided in three categories, with the first two being the most common:

1. Using source transformation: this type of tool gen-

erates the source code of the derivative function from the source code of the function.
2. Using operator overloading: in this type of tools the operators used for mathematical operations are overloaded to store the calculations required to evaluate a function (in addition to their usual behaviour). The derivatives are obtained by calling a special function that uses the stored operations to evaluate the derivatives. The overloading and derivative calculations can be made at compilation time using C++ templates [9] or at run time [8].
3. Other methods.

Source transformation tools potentially produce the most efficient code. They have the advantage that the automatically-generated derivative source code can be analysed and fine-tuned for performance. Operator overloading tools on the other hand offer the convenience that only the function source file needs to be maintained. They usually require the source code describing the function to use some special variable type instead of "double". For example, the Adol-C library [8] has a type called "adouble" and the FADBAD++ library [9] defines a special "F<>" template type. Operator overloading introduces an overhead in the evaluation of the function. The overhead becomes more important as the number of independent variables is increased. The design described in this paper attempts to minimise the overhead caused by the use of operator overloading. In particular, the FADBAD++ library is used, but it would be very simple to replace this by any other AD tool based on operator overloading.

Consider nonlinear VCCS contributions. We need the derivatives of the currents with respect to the controlling voltages (J_j) for Newton iterations in Eq. (3) but we also need the derivatives of the currents with respect to the device parameters ($\frac{\partial I}{\partial h}$) in Eq. (4). To avoid unnecessary overhead due to operator overloading, it would be convenient to have two versions of the source code that calculates the current in a nonlinear VCCS. The first version treats the controlling voltages as the input variables (subject to operator overloading overhead) and the device parameters as constants. This version of the code is then used to calculate the contribution to J_j. The second version is complementary to the first and is used to calculate the contributions to $\frac{\partial I}{\partial h}$. A third version of the function code using regular doubles in all variables would also be useful in the case that no derivative information is required. The main idea in this paper is to implement the function code using a template class in C++ and instantiate the template to produce the three different versions of the function code. In the following paragraphs we analyse the implementation of each type of current source individually.

A. NONLINEAR VCCS

To illustrate the implementation of a nonlinear VCCSs, consider the DC model for a bipolar transistor shown in Fig. 3. The model is composed of 8 VCCSs. Four of them are linear: the collector and emitter re-

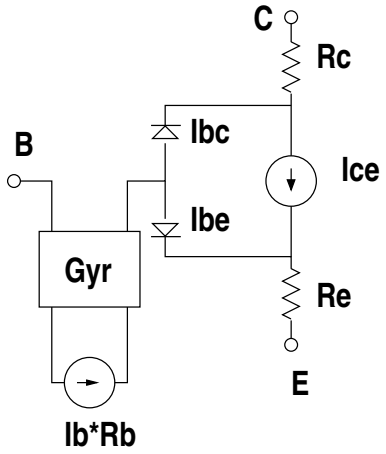

Fig. 3. DC Bipolar Transistor Model

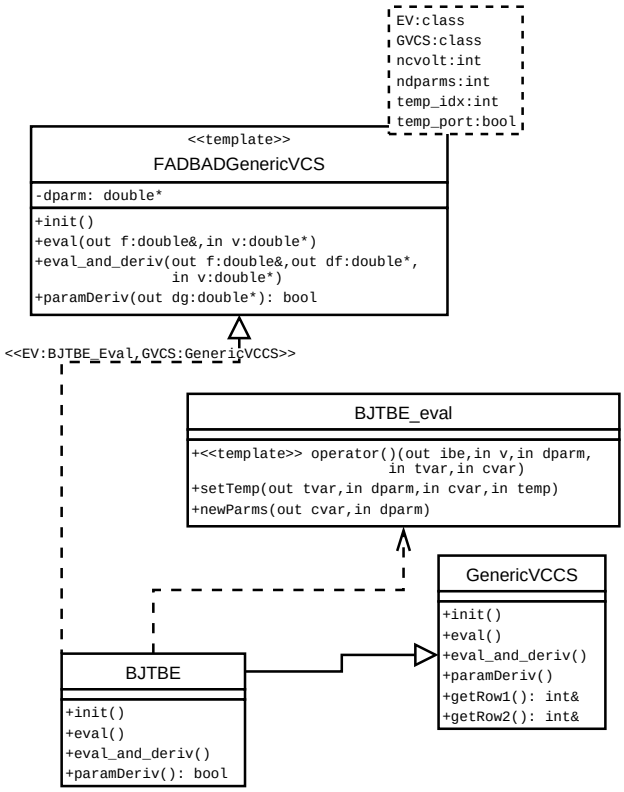

Fig. 4. UML Diagram for the BJTBE Class

sistors and the two VCCSs that form the gyrator [10]. Each of the nonlinear VCCSs is implemented by a class derived from **GenericVCCS**. A UML class diagram in Fig. 4 shows the implementation of the **BJTBE** class. This class models the diode connection between the internal base and emitter terminals in Fig. 3. The *eval()* method in **GenericVCCS** is a function that returns the output current of the VCCS given the values of the controlling voltages. The *eval_and_deriv()* method in addition to the output current calculates the derivatives of the output current with respect to the controlling voltages. The *paramDeriv()* method calculates the derivatives of the output current with respect to all the device parameters. There are other methods provided by this class but they are not essential for this discussion. As shown in Fig. 4, **BJTBE** is a class

derived from **GenericVCCS** and implements the discussed methods using the equations of the base-emitter diode. It is not necessary to manually code **BJTBE** because this class is actually a template instantiation of **FADBADGenericVCS**. As shown in the diagram, one of the arguments of this template class is the base class from which the VCCS is derived (**GenericVCCS** in this example) and another argument (**EV**) is a class that contains the actual device equations in template form. The rest of the template parameters are regular variables to indicate the number of controlling voltages, for example.

It is important to remark that the only dependence with the FADBAD++ library is inside the **FADBADGenericVCS** class. If this class were re-written for another AD library (based on operator overloading), any existing device models would continue to work.

The equations of the base-emitter diode are contained in the class named **BJTBE_eval** in the diagram. The *()* operator is overloaded to calculate the output current. There are two additional methods. The first one (*setTemp()*) is used to recalculate internal model variables if the temperature parameter is changed. The second (*newParms()*) is used to recalculate internal model variables when any other parameter different from temperature is changed. This is to avoid redundant calculations. The reason for special treatment to the temperature parameter is to allow electrothermal simulations where the temperature of a device is changing according to the power dissipated by the circuit and its environment. In an electrothermal simulation, the temperature of the device is treated as another controlling voltage [15].

The main work of a person that develops a new device model for the simulation programme is to implement the classes that contain the equations such as **BJTBE_eval**. The source code of **BJTBE_eval** is provided in Fig. 5. The controlling voltages, transistor parameters and internal variables are passed as arrays. Each individual transistor parameter is assigned a position in the *dparm* array with a corresponding *enum* name such as *EG*. This greatly simplifies parameter handling in the other classes in Fig. 4. Each of the template functions is instantiated with different argument types to implement the methods of the **BJTBE** class. In addition to the **BJTBE** class (one controlling voltage), the **TBJTBE** class is defined with 2 controlling voltages: the internal base-emitter voltage and the device thermal port voltage. The two classes are defined by instantiating the **FADBADGenericVCS** template class with different arguments. It is important to remark that template instantiations are performed at compilation time so there is no computational performance loss at run time.

The implementation of nonlinear VCCSs presented here results in a very compact code with a reasonable run-time performance and full derivative capabilities not available in any other circuit simulator programme. Another advantage of this approach is that the model equations can be tested and debugged by instantiating

```
// Implements the BE junction currents of a NPN BJT
#ifndef BJTBE_H
#define BJTBE_H 1
class BJTBE_eval {
public:
  // This are locally-defined variables
  enum {T_ISoBF, T_ISE, VT, MAX_TVAR_INDEX};
  enum {dummy, MAX_CVAR_INDEX};

  BJTBE_eval(int *iparm) {} // iparm not needed here

  // Main evaluation function. Only voltage
  // dependence considered here.
  template <typename currDouble, typename voltDouble,
            typename paramDouble,
            typename varDouble, typename cvarDouble>
  inline void operator()(currDouble& ibe, voltDouble* v,
                         paramDouble* dparm,
                         varDouble* tvar, cvarDouble* cvar)
  {
    // Limit the maximum value of the exponential function
    // (consider a special exp() function).
    ibe = dparm[AREA]
      * ( tvar[T_ISoBF] * (safe_exp(v[0]
          / dparm[NF] / tvar[VT]) - 1.)
          + tvar[T_ISE] * (safe_exp(v[0]
          / dparm[NE] / tvar[VT]) - 1.)) ;
  }
  // Adjust variables to a new temperature.
  template <typename tvarDouble, typename varDouble,
            typename tempDouble>
  inline void setTemp(tvarDouble* tvar,
                      varDouble* dparm, varDouble* cvar,
                      tempDouble& T)
  {
    // Parameter calculation
    tvarDouble tn = T / dparm[TNOM];
    tvar[VT] = kBoltzman * T / eCharge;
    tvar[T_ISoBF] = dparm[IS] * exp((tn-1.)
      * dparm[EG] / tvar[VT])
      * pow(tn, dparm[XTI] - dparm[XTB]) / dparm[BF];
    tvar[T_ISE] = dparm[ISE] * exp((tn-1.) * dparm[EG]
      / dparm[NE] / tvar[VT])
      * pow(tn, dparm[XTI] / dparm[NE] - dparm[XTB]);
  }
  // For efficiency, we may want to pre-calculate
  // some expressions that do not change with voltage or
  // temperature.  Adjust variables to a new parameter set.
  template <typename varDouble>
  inline void newParms(varDouble* cvar, varDouble* dparm)
  {
    // Do nothing.
  }
};

// No thermal port
typedef FADBADGenericVCS<BJTBE_eval, GenericVCCS, 1,
                         MAX_DPARM_INDEX, TEMP, false> BJTBE;
// With thermal port
typedef FADBADGenericVCS<BJTBE_eval, GenericVCCS, 2,
                         MAX_DPARM_INDEX, true> TBJTBE;

#endif
```

Fig. 5. Complete Source Code of the BJTBE Class

the template function with *double* type variables. This is much easier than debugging a function with special AD variable types.

B. LINEAR VCCS

Linear VCCSs are implemented in a similar way as illustrated in Fig. 6 for the **RType** class which is used to represent a regular resistor. The main difference with nonlinear VCCSs is that for regular DC analysis no derivatives are necessary and derivatives are only needed for sensitivity analysis (the contributions to $\frac{\partial G}{\partial h}$). The **RI_eval** class only is required to implement one function (the *()* operator) to calculate the conductance of the VCCS given the parameter values.

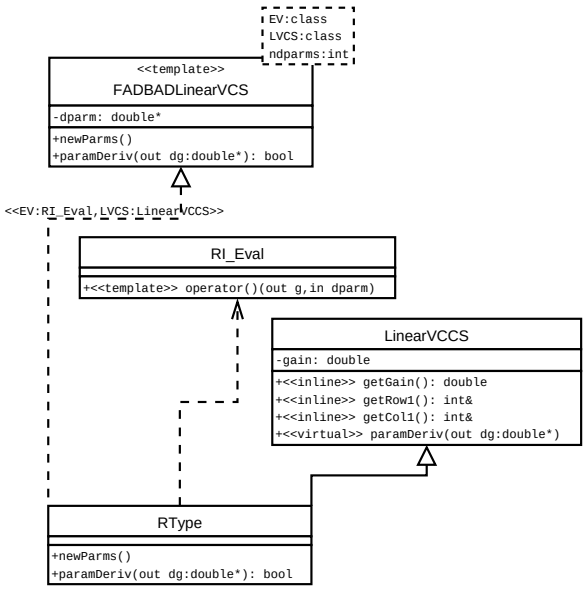

Fig. 6. UML Diagram for the RType Class

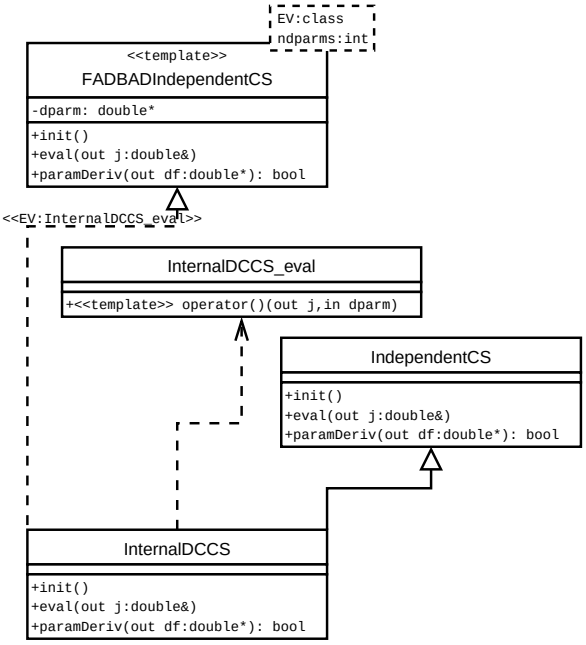

Fig. 7. UML Diagram for the InternalDCCS Class

The *getGain()* function in **LinearVCCS** is declared *inline* for optimal performance.

C. ICS

The **InternalDCCS** is a class used to represent a ICS that is part of an ideal voltage supply model. The UML diagram of the class is shown in Fig. 7. A similar design technique as in the previous cases is used.

V. CASE STUDY AND DISCUSSION

Fig. 8 shows the schematic diagram of a self-biasing current source [16]. This circuit produces an load current (I_L) that is almost independent of the supply voltage. The sensitivity of the load current with respect to a parameter (h) in the circuit can be calculated from

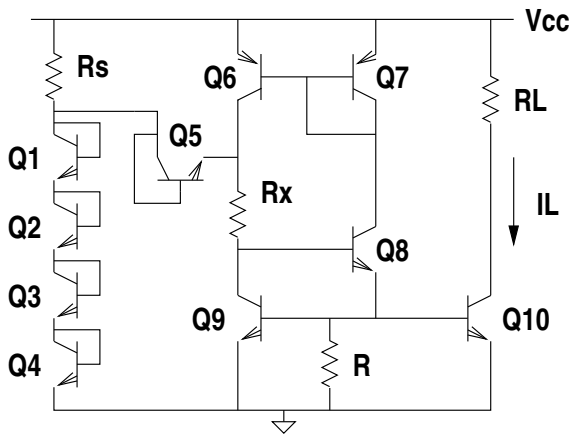

Fig. 8. Current Source Schematic

TABLE I: Comparison of Results

	Carrot	Spice	(increment)
I_L (μA)	135.18	135.18	-
$\partial I_L/\partial V_{CC}$ (μA/V)	6.902	6.900	(1 mV)
$\partial I_L/\partial T$ (μA/K)	-0.320	-0.320	(0.1 K)
$\partial I_L/\partial BF$ (nA)	2.041	2.000	(0.1)
$\partial I_L/\partial RB$ (nA/Ω)	0.6736	0.6800	(5 Ω)

the nodal voltages as follows,

$$\frac{\partial I_L}{\partial h} = \frac{1}{R_L}\frac{\partial V_{CC}}{\partial h} - \frac{1}{R_L}\frac{\partial V_{C10}}{\partial h},$$

where V_{C10} is the collector voltage of $Q10$ and R_L is assumed to be constant. Table I shows the values of I_L and the sensitivities with respect to the supply voltage and other circuit parameters evaluated using the proposed simulator programme and Spice. The corresponding Spice netlist with all circuit parameters is given in Fig. 9. It can be observed that the sensitivity with respect to the supply voltage is small as expected. The sensitivity with respect to temperature was included in the results because its exact value is very cumbersome to obtain analytically if the derivatives for each model are coded manually. Also included are sensitivities with respect to some NPN bipolar transistor model parameters. The sensitivities in Spice were numerically calculated using the ".op" analysis. The choice of the increment for numerical derivatives is critical to obtain an accurate derivative with respect to some parameters such as the base resistance (RB). An increment value of 10 Ω is too large, while an increment value of 1 Ω would cause too much rounding error. These problems can be somewhat alleviated by combining the adjoint method with numerical derivatives, as implemented in most circuit simulators. These problems are completely eliminated with the use of AD.

VI. CONCLUSIONS

Software design aspects of a general circuit simulator that is capable of calculating exact sensitivities using AD have been presented for the first time. The implementation and maintainance of new circuit device

```
*** Self-Biased Current Source ***
.options temp=26.85
vcc 1 0 6.0V
* Start-up branch
rs 1 2 7k res1
q8 2 2 3 mynpn
q9 3 3 4 mynpn
q10 4 4 5 mynpn
q11 5 5 0 mynpn
q7 2 2 7 mynpn
* PNP current mirror
q1 7 6 1 mypnp
q2 6 6 1 mypnp
* NPN current source
rx 7 8 7k res1
q4 6 8 9 mynpn
q5 8 9 0 mynpn
q6 11 9 0 mynpn
r1 9 0 7k res1
rl1 1 11 1k
.model res1 r (tc1=1.5e-3 tnom=26.85)
.model mynpn npn (bf=200 rb=100 re=1 rc=1
+ vaf=80 ikf=3e-3 ise=1e-15 ne=1.5
+ rbm=50 tnom=26.85)
.model mypnp pnp (bf=50 rb=200 re=1 rc=10
+ vaf=50 ikf=.5e-3 ise=1e-15 ne=1.5
+ rbm=100 tnom=26.85)
.op
.end
```

Fig. 9. Spice Netlist of the Current Source

models is greatly simplified with the use of AD techniques. In the case of nonlinear VCCSs C++ templates are used to automatically create three versions of the device equations: the first is optimised to calculate the output current, the second is optimised for the calculation of the output current and derivatives respect to input voltages; and the third is optimised for the calculation of the derivatives of the current with respect to the device parameters.

The design is not strongly dependent on a particular AD tool. It is possible to use different AD libraries with minor changes to the source code. This will be useful as improved AD tools will be available in the future. The programme was demonstrated with the calculation of the sensitivity of a self-biasing current source with respect to the supply voltage, circuit temperature and some bipolar transistor model parameters. The design approach presented here could be easily extended to the calculation of higher-order sensitivities since the AD technique allows the evaluation of derivatives of any order. Future work will address the application of this approach to the calculation of sensitivities in other types of circuit analyses such as transient or harmonic balance.

ACKNOWLEDGEMENT

The author would like to acknowledge the support of Science and Engineering Research Canada (NSERC).

REFERENCES

[1] V. Rizzoli, A. Lipparini, A. Costanzo, F. Mastri, C. Ceccetti, A. Neri and D. Masotti, "State-of-the-Art Harmonic-Balance Simulation of Forced Nonlinear Microwave Circuits by the Piecewise Technique," *IEEE Trans. on Microwave Theory and Tech.*, Vol. 40, No. 1, Jan 1992.

[2] A. Ibsais and V. Ajjarapu, "The role of automatic differentiation in power system analysis," *IEEE Trans. on Power Systems*, Vol. 10, No. 2, May 1997, pp. 592–597.

[3] E. Solodovnik, G. Cokkinides, R. Dougal and A. P. Sakis Meliopoulos, "Nonlinear power system component modeling using symbolically assisted computations," *2001 IEEE Power Engineering Soc. Summer Meeting Dig.*, Vol.3, pp. 1439–1444.

[4] J. E. Mehner, A. Schaporin, V. Kolchuzhin, W. Doetzel, T. Gessner, "Parametric model extraction for MEMS based on variational finite element techniques," *13th Int. Conf. on Solid-State Sensors, Actuators and Microsystems Dig.*, Vol.1, 2005 pp. 776–779.

[5] J. Vlach and K. Singhal, *Computer Methods for Circuit Analysis and Design*, Van Nostrand Reinhold, 1983.

[6] B. Melville. P. Feldmann and S. Moinian, "A C++ based environment for analog circuit simulation," *IEEE 1992 Int. Conf. on Computer Design Digest*, Oct. 1992, pp. 516–519.

[7] C. E. Christoffersen, U. A. Mughal and M. B. Steer, "Object oriented microwave circuit simulation," *Int. Journal of RF and Microwave Computer-Aided Engineering*, Vol. 10, Issue 3, 2000, pp. 164–182.

[8] A. Griewank, D. Juedes and J. Utke, "Adol-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++," *ACM TOMS*, Vol. 22(2), pp. 131–167, June 1996.

[9] C. Bendtsen and O. Stauning, "FADBAD, a flexible C++ package for automatic differentiation," *Department of Mathematical Modelling*, Technical University of Denmark, 1996.

[10] M. Valtonen, P. Heikkilä, A. Kankkunen, K. Mannersalo, R. Niutanen, P. Stenius, T. Veijola and J. Virtanen, "APLAC - A new approach to circuit simulation by object orientation," *10th European Conference on Circuit Theory and Design Dig.*, 1991.

[11] J. W. Bandler, S. H. Chen, S. Daijavad and K. Madsen, "Efficient optimization with integrated gradient approximations," *IEEE Trans. on Microwave Theory and Techniques*, Vol. 36, No. 2, Feb. 1988.

[12] J. W. Bandler, Q. Zhang, J. Song and R. M. Biernacki, "Fast gradient based yield optimization of nonlinear circuits," *IEEE Trans. on Microwave Theory and Techniques*, Vol. 38, No. 11, Feb. 1990.

[13] S. W. Ambler, "UML 2.0 Style," Cambridge University Press, 2005.

[14] T. A. Davis, "A column pre-ordering strategy for the unsymmetric-pattern multifrontal method," *ACM Trans. Math. Software*, Vol 30, No. 2, 2004, pp. 165–195.

[15] W. Batty, C. E. Christoffersen, A. B. Yakovlev, J. F. Whitaker, M. Ozkar, S. Ortiz, A. Mortazawi, R. Reano, K. Yang, L. P. B. Katehi, C. M. Snowden and M. B. Steer, "Global Coupled EM-Electrical-Thermal Simulation and Experimental Validation for a Spatial Power Combining MMIC Array," *IEEE Trans. on Microwave Theory and Techniques*, Vol. 50, No. 12, December 2002, pp. 2820–2833.

[16] P. L. Gray, P. J. Hurst, S. H. Lewis and R. Meyer, "Analysis and design of analog integrated circuits," Wiley, fouth edition, 2001.

AUTHOR BIOGRAPHY

CARLOS E. CHRISTOFFERSEN received the Electronic Engineer degree at the National University of Rosario, Argentina in 1993. From 1993 to 1995 he was research fellow of the National Research Council of Argentina (CONICET). He received an M.S. degree and a Ph.D. degree in Electrical Engineering in 1998 and 2000, respectively, from North Carolina State University. Currently he is an Assistant Professor in the Department of Electrical Engineering at Lakehead University, Thunder Bay, Canada. He is a member of the IEEE. His current research interests include programming techniques applied to scientific problems and analogue and RF circuit computer-aided design including electromagnetic and thermal interactions.