Implementation Of Exact Sensitivities In A Circuit Simulator Using Automatic Differentiation

C. E. Christoffersen
E-mail: c.christoffersen@ieee.org

Department of Electrical Engineering, Lakehead University,
Thunder Bay, Ontario, Canada P7B 5E1
Outline

- Introduction
- Motivation
- Automatic Differentiation
- Circuit Analysis
- Implementation
- Case Study
- Conclusions
- Future Work
Introduction

Given a circuit function ($\Phi$) and a circuit parameter ($h$),

$$D^\Phi_h = \frac{\partial \Phi}{\partial h}$$

is the Sensitivity of $\Phi$ with respect to $h$. 
Introduction

Given a circuit function ($\Phi$) and a circuit parameter ($h$),

$$D_\Phi^h = \frac{\partial \Phi}{\partial h}$$

is the \textbf{Sensitivity} of $\Phi$ with respect to $h$.

Examples of circuit functions ($\Phi$):

- Nodal voltage
- Branch current
- Filter Bandwidth
- Amplifier distortion (IP3)
Introduction

Given a circuit function \((\Phi)\) and a circuit parameter \((h)\),

\[
D_{h}^{\Phi} = \frac{\partial \Phi}{\partial h}
\]

is the Sensitivity of \(\Phi\) with respect to \(h\).

Examples of circuit parameters \((h)\):

- MOS transistor channel length
- Device temperature
- Reverse saturation current in a diode
- Width of transmission line
# Introduction

Circuit analysis techniques → Circuit functions

<table>
<thead>
<tr>
<th>Method</th>
<th>DC</th>
<th>AC</th>
<th>Harmonic Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bias point</td>
<td>bias point</td>
<td>linearised</td>
<td>frequency-domain</td>
</tr>
<tr>
<td>Linearised</td>
<td>constant</td>
<td>sinusoidal</td>
<td>arbitrary</td>
</tr>
<tr>
<td>Time-domain</td>
<td></td>
<td></td>
<td>quasi-periodic</td>
</tr>
<tr>
<td>Frequency-domain</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quasi-periodic</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Introduction

Circuit analysis techniques → Circuit functions

<table>
<thead>
<tr>
<th>DC</th>
<th>AC</th>
<th>Transient</th>
<th>Harmonic Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>bias point</td>
<td>linearised</td>
<td>time-domain</td>
<td>frequency-domain</td>
</tr>
<tr>
<td>constant</td>
<td>sinusoidal</td>
<td>arbitrary</td>
<td>quasi-periodic</td>
</tr>
</tbody>
</table>

The derivatives of the individual device equations are always required for sensitivity evaluation.
Motivation

- Numerical differences
  - Uncertainty in increment size
  - Inaccuracy (high order derivatives)
  - Decrease in convergence rate

- Manual coding or symbolic differentiation
  - Unwieldy formulae
  - Repeated evaluation of common expressions
  - Many device types → Tedious maintainance
Automatic Differentiation

\[ f = (x + y) \sin x \cos y \]
\[ \frac{\partial f}{\partial x} = \sin x \cos y + (x + y) \cos x \cos y \]
Automatic Differentiation

\[ f = (x + y) \sin x \cos y \]
\[ \frac{\partial f}{\partial x} = \sin x \cos y + (x + y) \cos x \cos y \]

<table>
<thead>
<tr>
<th>Code list</th>
<th>Tangent code list</th>
<th>Linearisation on ( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 = x )</td>
<td>( \nabla t_3 = \nabla t_1 + \nabla t_2 )</td>
<td>( \nabla t_1 = 1 )</td>
</tr>
<tr>
<td>( t_2 = y )</td>
<td>( \nabla t_4 = \nabla t_1 \cos t_1 )</td>
<td>( \nabla t_2 = 0 )</td>
</tr>
<tr>
<td>( t_3 = t_1 + t_2 )</td>
<td>( \nabla t_5 = -\nabla t_2 \sin t_2 )</td>
<td>1</td>
</tr>
<tr>
<td>( t_4 = \sin t_1 )</td>
<td>( \nabla t_6 = \nabla t_3 t_4 + t_3 \nabla t_4 )</td>
<td>( \cos x )</td>
</tr>
<tr>
<td>( t_5 = \cos t_2 )</td>
<td>( \nabla t_7 = \nabla t_6 t_5 + t_6 \nabla t_5 )</td>
<td>0</td>
</tr>
<tr>
<td>( t_6 = t_3 t_4 )</td>
<td></td>
<td>( \sin x + (x + y) \cos x )</td>
</tr>
<tr>
<td>( t_7 = t_6 t_5 )</td>
<td></td>
<td>( (\sin x + (x + y) \cos x) \cos y )</td>
</tr>
</tbody>
</table>
Automatic Differentiation (AD)

- Only the function has to be coded
- Source code is differentiated (instead of function expression)
- Numerically exact
- No more than 5 times the operations needed to evaluate the function (scalar gradient)
- Many AD libraries exist (http://www.autodiff.org)
Automatic Differentiation (AD)

- Only the function has to be coded
- Source code is differentiated (instead of function expression)
- Numerically exact
- No more than 5 times the operations needed to evaluate the function (scalar gradient)
- Many AD libraries exist (http://www.autodiff.org)

We focus in C++ operator-overloading AD libraries.
F<double> x, y, f;
x = 1;
x.diff(0, 2);
y = 2;
y.diff(1, 2);
f = (x + y) * sin(x) * cos(y);
double fval = f.x();
double dfdx = f.d(0);
double dfdy = f.d(1);
Circuit Analysis

Current source approach:
DC Analysis

\[ Gu + I(u) = S \]

\( u \): vector of nodal voltages

\( G \): matrix of conductances (linear devices)

\( I(u) \): vector function (nonlinear devices)

\( S \): source vector

Newton Method → Linear system:

\[ [G + J(u_j)]u_{j+1} = S - I(u_j) + J(u_j)u_j \]

\( j \): iteration index

\( J_j \): Jacobian matrix of \( I(u_j) \). Recalculated at every iteration
DC Analysis Sensitivity

$$\Phi = d^T u$$

$d$: column vector

$$[G + J(u)]^T u_a = d$$

$u_a$: adjoint voltages vector

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

$I(u) \rightarrow J(u), \frac{\partial I}{\partial h}$
Implementation: Performance Issues

\[ f(x, y, z) = y\sqrt{x} + (\sin y + \cos z) \sin \sqrt{x} \]

Time to evaluate \( f \) and \( \partial f / \partial x \) with FADBAD++

<table>
<thead>
<tr>
<th>Normalised Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x, y, z: \text{double} )</td>
</tr>
<tr>
<td>( x, y, z: \text{F&lt;double&gt;} )</td>
</tr>
<tr>
<td>( x: \text{F&lt;double&gt;} \rightarrow y, z: \text{double} )</td>
</tr>
</tbody>
</table>
Implementation: Performance Issues

\[ f(x, y, z) = y\sqrt{x} + (\sin y + \cos z) \sin \sqrt{x} \]

Time to evaluate \( f \) and \( \partial f / \partial x \) with FADBAD++

<table>
<thead>
<tr>
<th>Normalised Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x, y, z: \text{double} )</td>
</tr>
<tr>
<td>( x, y, z: \text{F&lt;double&gt;} )</td>
</tr>
<tr>
<td>( x: \text{F&lt;double&gt;} \rightarrow y, z: \text{double} )</td>
</tr>
</tbody>
</table>

Example: 2 controlling voltages \( (J(u)) \), 30 device parameters \( (\partial I / \partial h) \)
Implementation

Require:
- Nonlinear currents \( I(u) \)
- Derivatives respect to nodal voltages \( J(u) \)
- Derivatives respect to parameters \( \partial I / \partial h \)
Implementation

- Require:
  - Nonlinear currents \( I(u) \)
  - Derivatives respect to nodal voltages \( J(u) \)
  - Derivatives respect to parameters \( \partial I/\partial h \)

- But not all derivatives required every time → Minimise overhead → Need three versions of functions
Implementation

Require:
- Nonlinear currents \( I(u) \)
- Derivatives respect to nodal voltages \( J(u) \)
- Derivatives respect to parameters \( \partial I/\partial h \)

But not all derivatives required every time → Minimise overhead → Need three versions of functions

Each device type handled by a different class:
- **LinearVCCS**: contribute to \( G \)
- **GenericVCCS**: contribute to \( I(u) \)
- **IndependentCS**: contribute to \( S \)
Implementation Of Exact Sensitivities In A Circuit Simulator Using Automatic Differentiation – p. 14
Implementation

::<template>>

FADBADGenericVCS
-dparm: double*
+init()
+eval(out f:double&, in v:double*)
+eval_and_deriv(out f:double&, out df:double*, in v:double*)
+paramDeriv(out dg:double*): bool

<<EV:BJTBE_Eval,GVCS:GenericVCCS>>

BJTBE
+init()
+eval()
+eval_and_deriv()
+paramDeriv(): bool

<<template>>

BJTBE_eval
+<<template>> operator()(out ibe, in v, in dparm, in tvar, in cvar)
+<<template>> setTemp(out tvar, in dparm, in cvar, in temp)
+<<template>> newParms(out cvar, in dparm)

GenericVCCS
+init()
+eval()
+eval_and_deriv()
+paramDeriv()
+getRow1(): int&
+getRow2(): int&
## Case Study

<table>
<thead>
<tr>
<th></th>
<th>Carrot</th>
<th>Spice</th>
<th>Nominal</th>
<th>Increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_L$ ($\mu$A)</td>
<td>135.18</td>
<td>135.18</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$\partial I_L / \partial V_{CC}$ ($\mu$A/V)</td>
<td>6.902</td>
<td>6.900</td>
<td>6 V</td>
<td>1 mV</td>
</tr>
<tr>
<td>$\partial I_L / \partial T$ ($\mu$A/K)</td>
<td>-0.320</td>
<td>-0.320</td>
<td>300 K</td>
<td>0.1 K</td>
</tr>
<tr>
<td>$\partial I_L / \partial BF$ (nA)</td>
<td>2.041</td>
<td>2.000</td>
<td>200</td>
<td>0.1</td>
</tr>
<tr>
<td>$\partial I_L / \partial RB$ (nA/$\Omega$)</td>
<td>0.6736</td>
<td>0.6800</td>
<td>100 $\Omega$</td>
<td>5 $\Omega$</td>
</tr>
</tbody>
</table>

---

![Circuit Diagram](image-url)
Conclusions

- Reduced overhead of AD library using C++ templates
- Device model code independent of AD library
- Transparent to model developer: simpler implementation of new models
Future Work

- Sensitivities in other simulation methods (transient, etc.)
- Higher-order sensitivities
- More device models
- Other AD libraries