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 (Φ) and a circuit parameter (h),

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

is the Sensitivity of Φ with respect to h.

Introduction

Given a circuit function (Φ) and a circuit parameter (h),

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

is the Sensitivity of Φ with respect to h.

Examples of circuit functions (Φ):

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

Introduction

Given a circuit function (Φ) and a circuit parameter (h),

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

is the Sensitivity of Φ 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

DC	bias point	constant
AC	linearised	sinusoidal
Transient	time-domain	arbitrary
Harmonic Balance	frequency-domain	quasi-periodic

Introduction

Circuit analysis techniques → Circuit functions

DC	bias point	constant
AC	linearised	sinusoidal
Transient	time-domain	arbitrary
Harmonic Balance	frequency-domain	quasi-periodic

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

$$
\begin{aligned}
f &= (x+y)\sin x\cos y \\
\frac{\partial f}{\partial x} &= \sin x\cos y + (x+y)\cos x\cos y
\end{aligned}
$$

Automatic Differentiation

$$\begin{aligned} f &= (x+y)\sin x\cos y \\ \frac{\partial f}{\partial x} &= \sin x\cos y + (x+y)\cos x\cos y \end{aligned}$$

Code list	Tangent code list	Linearisation on x
$t_1 = x$		$\nabla t_1 = 1$
$t_2 = y$		$\nabla t_2 = 0$
$t_3 = t_1 + t_2$	$\nabla t_3 = \nabla t_1 + \nabla t_2$	1
$t_4 = \sin t_1$	$\nabla t_4 = \nabla t_1 \cos t_1$	$\cos x$
$t_5 = \cos t_2$	$\nabla t_5 = -\nabla t_2 \sin t_2$	0
$t_6 = t_3 t_4$	$\nabla t_6 = \nabla t_3\, t_4 + t_3 \nabla t_4$	$\sin x + (x+y)\cos x$
$t_7 = t_6 t_5$	$\nabla t_7 = \nabla t_6\, t_5 + t_6 \nabla t_5$	$(\sin x + (x+y)\cos x)\cos y$

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.

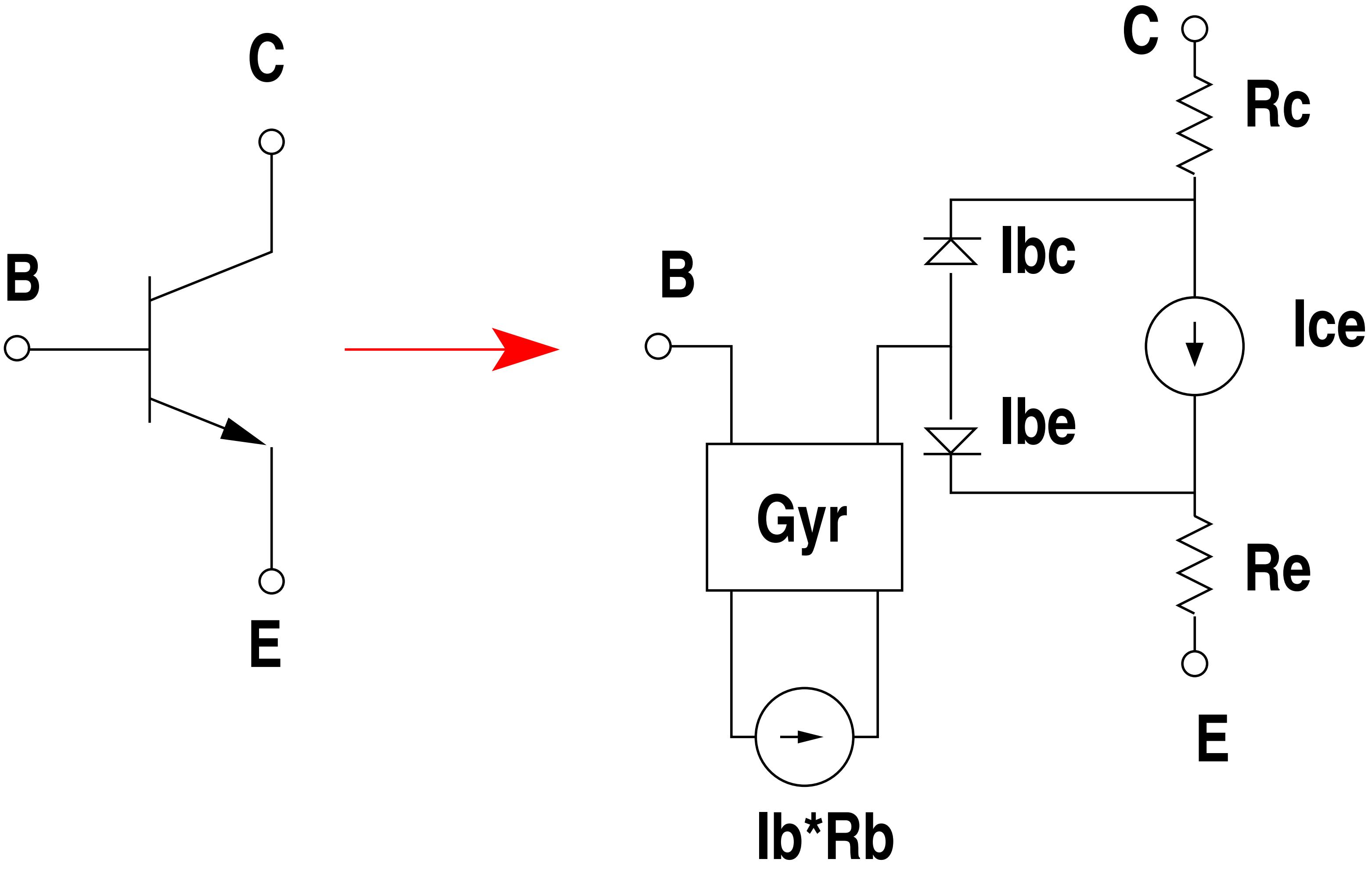
Automatic Differentiation: FADBAD++

http://www2.imm.dtu.dk/˜ km/FADBAD/

```
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

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

$$\boxed{I(u) \rightarrow J(u), \tfrac{\partial I}{\partial h}}$$

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++

	Normalised Time
x, y, z: `double`	1
x, y, z: `F<double>`	8.08
x: `F<double>` — y, z: `double`	4.42

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++

	Normalised Time
x, y, z: `double`	1
x, y, z: `F<double>`	8.08
x: `F<double>` — y, z: `double`	4.42

Example: 2 controlling voltages ($J(u)$), 30 device parameters ($\frac{\partial I}{\partial h}$)

Implementation

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

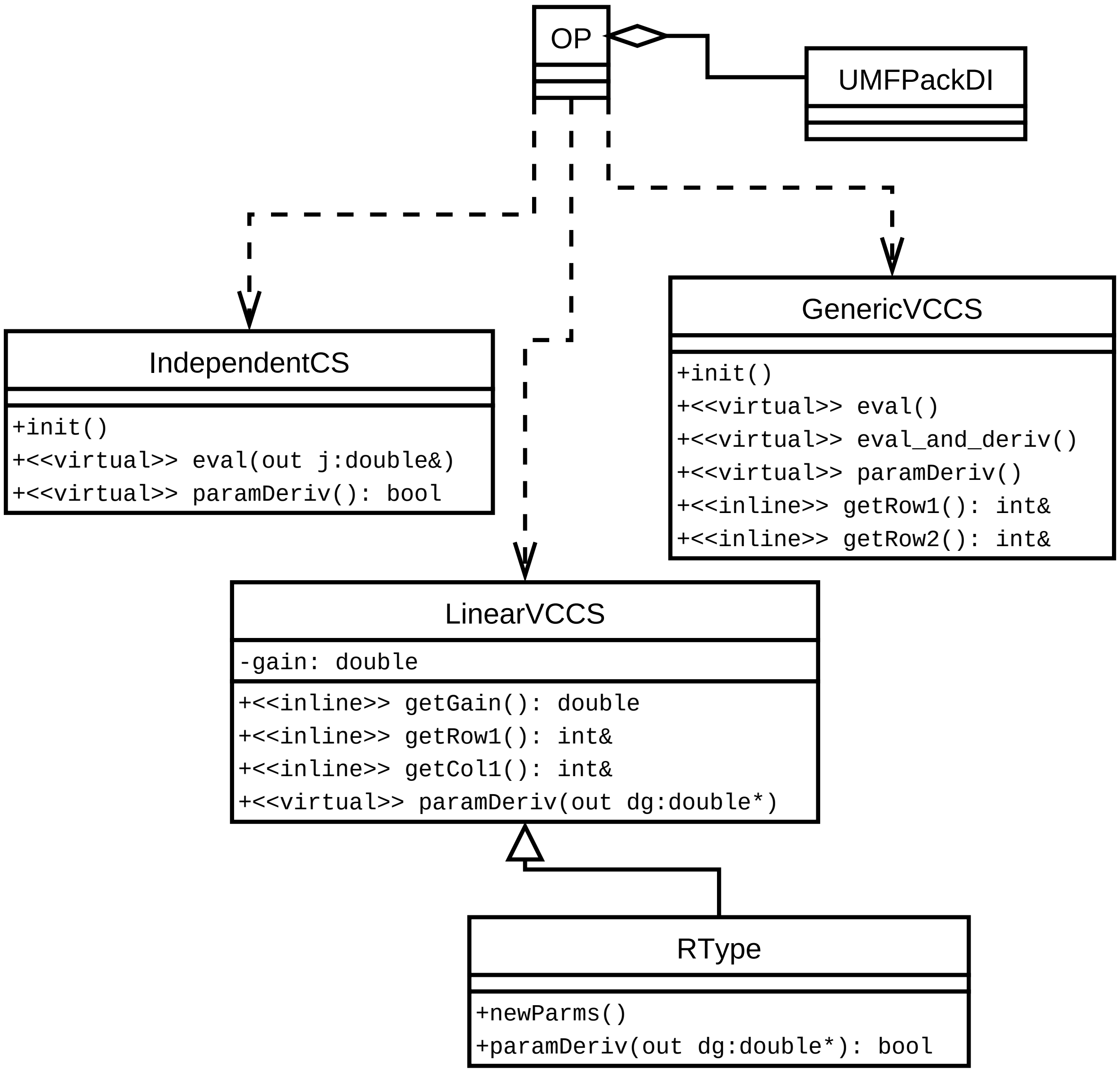
Implementation

- 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 $\rightarrow$ Minimise overhead $\rightarrow$ 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



OP

UMFPackDI

GenericVCCS

```
+init()
+<<virtual>> eval()
+<<virtual>> eval_and_deriv()
+<<virtual>> paramDeriv()
+<<inline>> getRow1(): int&
+<<inline>> getRow2(): int&
```

IndependentCS

```
+init()
+<<virtual>> eval(out j:double&)
+<<virtual>> paramDeriv(): bool
```

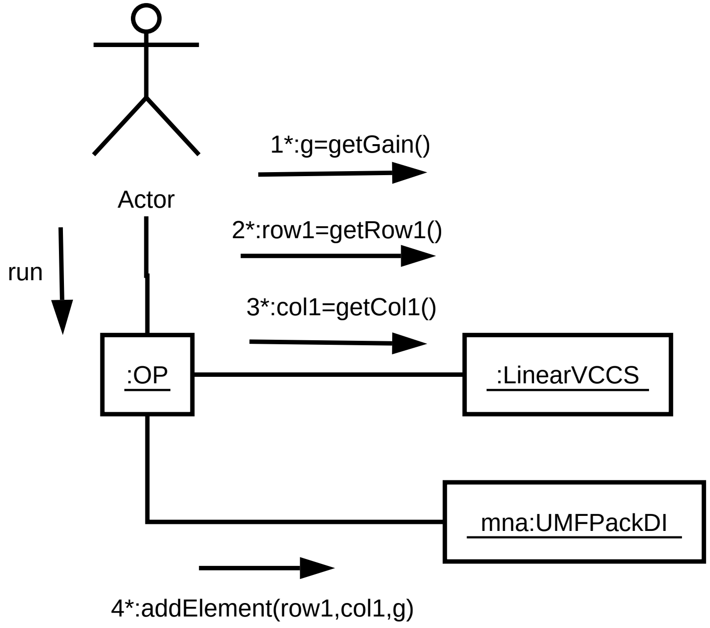
LinearVCCS

```
-gain: double
```

```
+<<inline>> getGain(): double
+<<inline>> getRow1(): int&
+<<inline>> getCol1(): int&
+<<virtual>> paramDeriv(out dg:double*)
```

RType

```
+newParms()
+paramDeriv(out dg:double*): bool
```



Actor

run

1*:g=getGain()

2*:row1=getRow1()

3*:col1=getCol1()

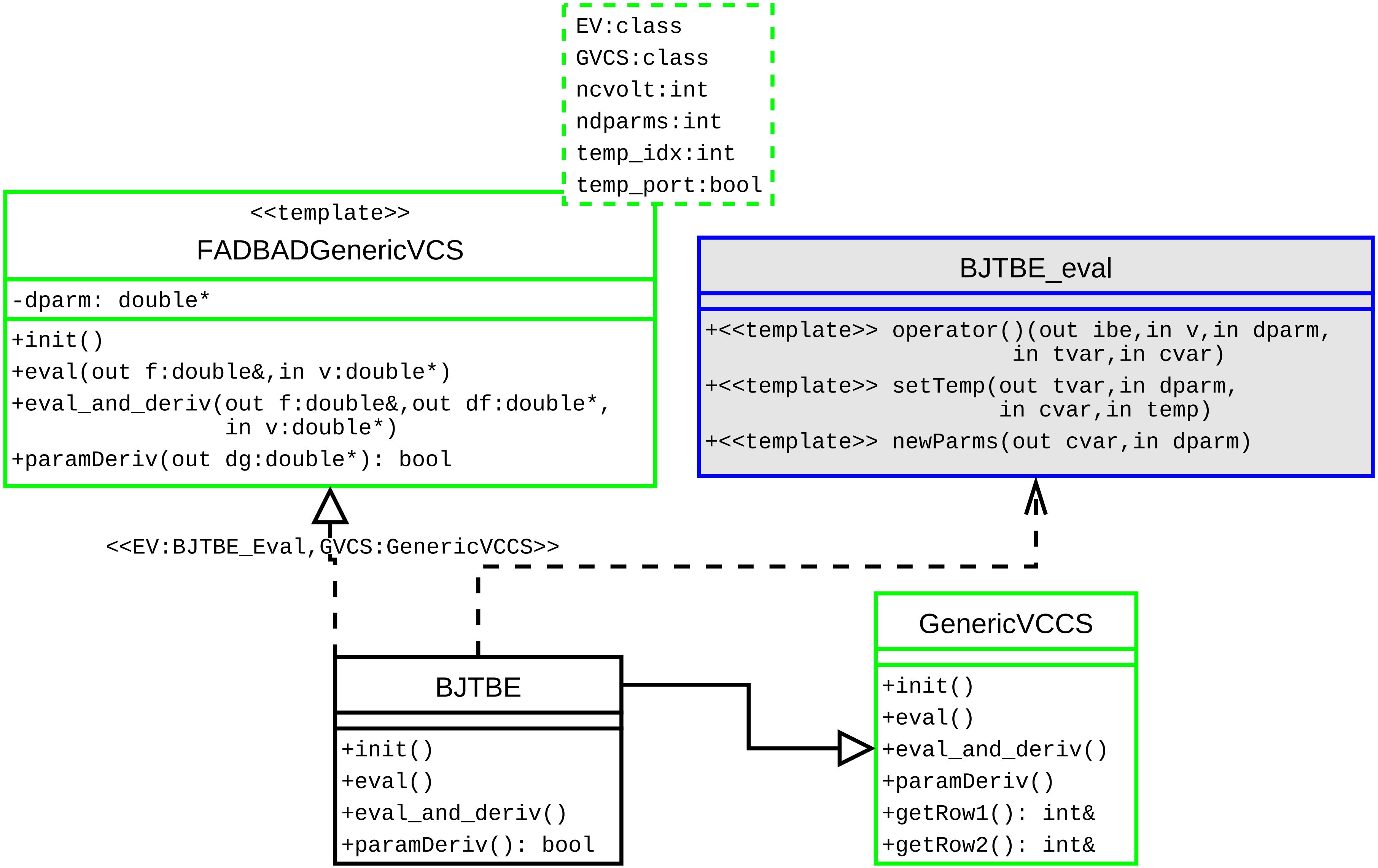
:OP

:LinearVCCS

mna:UMFPackDI

4*:addElement(row1,col1,g)

Implementation



EV:class
GVCS:class
ncvolt:int
ndparms:int
temp_idx:int
temp_port:bool

<<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

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)

<<EV:BJTBE_Eval,GVCS:GenericVCCS>>

BJTBE

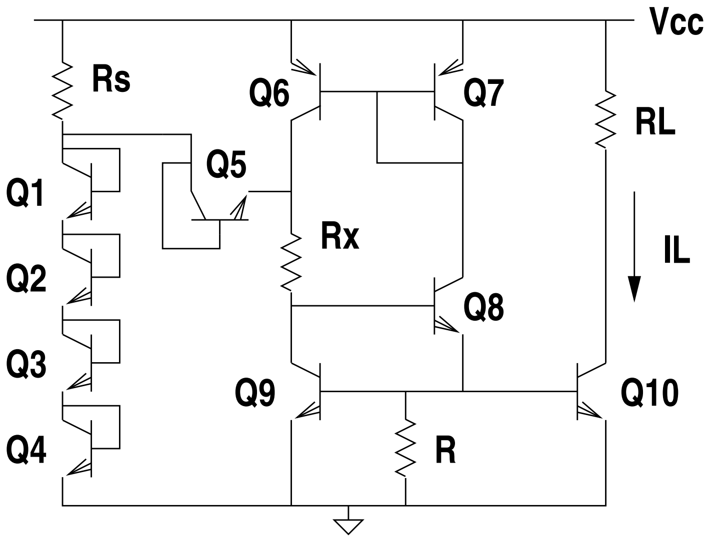
+init()
+eval()
+eval_and_deriv()
+paramDeriv(): bool

GenericVCCS

+init()
+eval()
+eval_and_deriv()
+paramDeriv()
+getRow1(): int&
+getRow2(): int&

Case Study

	Carrot	Spice	Nominal	Increment
I_L (μA)	135.18	135.18	-	-
$\partial I_L/\partial V_{CC}$ (μA/V)	6.902	6.900	6 V	1 mV
$\partial I_L/\partial T$ (μA/K)	-0.320	-0.320	300 K	0.1 K
$\partial I_L/\partial BF$ (nA)	2.041	2.000	200	0.1
$\partial I_L/\partial RB$ (nA/Ω)	0.6736	0.6800	100 Ω	5 Ω



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