A Software Engineering Approach to Automatic Differentiation of C++ Applications with Sacado

Eric Phipps
David Gay, Ross Bartlett

Optimization & Uncertainty Quantification Department
Sandia National Laboratories
Albuquerque, NM  USA
etphipp@sandia.gov
AD R&D Dictated by Sandia’s Computational Environment

- Object oriented C++ simulation codes
  - Operator overloading AD approach
  - Sacado (part of Trilinos): Forward, Reverse & Taylor modes

- Operator overloading introduces runtime overhead
  - Many highly specialized AD types
  - Compiler optimization (e.g., expression templates, expression-level reverse mode)
  - C++ templates

- Templating introduces developer overhead
  - Only apply AD to hard nonlinear parts (e.g., element level)
  - Manually differentiate remaining pieces
  - Incorporate derivatives into solvers via interfaces

- Targeting implicit element-based codes
  - We know the combinatorial structure
Differentiating Element-Based Simulation Codes

- Global residual computation (ignoring boundary computations):
  \[ f(\dot{u}, u, p, t) = \sum_{i=1}^{N} Q_i^T e_{k_i}(P_i \dot{u}, P_i u, p, t) \]

- Time-step Jacobian computation:
  \[ \alpha \frac{\partial f}{\partial \dot{u}} + \beta \frac{\partial f}{\partial u} = \sum_{i=1}^{N} Q_i^T \left( \alpha \frac{\partial e_{k_i}}{\partial \dot{u}_i} + \beta \frac{\partial e_{k_i}}{\partial u_i} \right) P_i \]

- Parameter derivative computation:
  \[ \frac{\partial f}{\partial p} = \sum_{i=1}^{N} Q_i^T \frac{\partial e_{k_i}}{\partial p} \]

- Hybrid symbolic/AD procedure
  - Only use AD for element derivatives
  - Element computations are narrow & shallow
  - Avoids differentiating through MPI
  - Demonstrated through Sandia’s Charon application code
Manually Incorporating Derivatives for Transient Sensitivity Analysis

- Spatially discretized PDE:
  \[ f(\dot{u}, u, p, t) = 0 \]

- Temporal discretization (Backward Euler):
  \[ f \left( \frac{u_n - u_{n-1}}{\Delta t}, u_n, p, t_n \right) = 0 \]

- Forward sensitivity problem:
  \[ \frac{\partial f}{\partial \dot{u}} \left( \frac{\partial \dot{u}}{\partial p} \right) \frac{\partial f}{\partial u} \left( \frac{\partial u}{\partial p} \right) + \frac{\partial f}{\partial p} = 0 \]
  \[ \frac{1}{\Delta t} \frac{\partial f}{\partial \dot{u}} \left( \frac{\partial u_n}{\partial p} - \frac{\partial u_{n-1}}{\partial p} \right) + \frac{\partial f}{\partial u} \left( \frac{\partial u_n}{\partial p} \right) + \frac{\partial f}{\partial p} = 0 \]

- Rythmos time integration package
  - Implicit BDF time integration method
  - Staggered corrector sensitivity method (Barton)
  - Derivatives provided through interfaces
QASPR
Qualification of electronic devices in hostile environments

Si interstitial (I) (+2, +1, 0, –1, –2)

Vacancy (V) (+2, +1, 0, –1, –2)

VV (+1, 0, –1, –2)

B

I

(B, 0, –)

C

I

(B, 0, –)

VP (0, –)

VB (+, 0)

VO (0, –)

B, B (0, –)

B, O (+, 0)

B, C

Defect reactions

Stockpile BJT

PDE semiconductor device simulation

Annihilation

Annihilation

Vacancy (V) (+2, +1, 0, –1, –2)

Annihilation

VV (+1, 0, –1, –2)

VB (+, 0)

VP (0, –)

VO (0, –)

No irradiation: $I_B = -0.05 \, \mu A$

Experiment

Defect annealing

Base current (µA)

Time (s)
Transient Sensitivity Analysis of a Radiation Damaged Bipolar Junction Transistor

- Bipolar Junction Transistor
- Pseudo 1D strip (9x0.1 micron)
- 39 PDEs, linear 4-node finite elements, ~100k unknowns
- 126 parameters

Sensitivities show dominant physics

Comparison to FD:
- Sensitivities at all time points
- More accurate
- More robust
- 14x faster!
Concluding Remarks

• Extending our capabilities
  – C++ templating enables easy incorporation of new AD types
    • Second derivatives, Taylor polynomials, polynomial chaos expansions,…
  – Solver algorithm R&D
    • Transient adjoint sensitivities (Rythmos), optimization (MOOCHO, Aristos)

• AD recognized as vehicle supporting transformation to predictive science
  – Computational design, optimization and parameter estimation
  – Stability analysis
  – Uncertainty quantification
  – Verification and validation

• AD approach tailored Sandia computational environment
  – Operator-overloading based AD tools & C++ templating
  – AD applied selectively as a software engineering tool
  – Software integration with solvers
  – Requires close collaboration with application/solver developers
Points of Contact

• Sacado available through Trilinos (starting with version 8.0)
  – trilinos.sandia.gov
  – Coming to www.autodiff.org soon

• Sacado developers
  – Eric Phipps (etphipp@sandia.gov)
  – David Gay (dmgay@sandia.gov)

• Examples are best way to learn Sacado
  – Simple examples for basic Sacado use
  – Sacado FEApp example 1D finite element application for how to build this into a large-scale application
Simple Sacado Example

```cpp
// The function to differentiate

double func(double a, double b, double c) {
    double r = c*std::log(b+1.)/std::sin(a);
    return r;
}

int main(int argc, char **argv) {
    double a = std::atan(1.0);          // pi/4
    double b = 2.0;
    double c = 3.0;

    // Compute function
    double r = func(a, b, c);
```
```cpp
#include "Sacado.hpp"

// The function to differentiate
template<typename ScalarT>
ScalarT func(const ScalarT& a, const ScalarT& b, const ScalarT& c) {
  ScalarT r = c*std::log(b+1.)/std::sin(a);
  return r;
}

int main(int argc, char **argv) {
  double a = std::atan(1.0); // pi/4
  double b = 2.0;
  double c = 3.0;

  int num_deriv = 2; // Number of independent variables
  Sacado::Fad::DFad<double> afad(num_deriv, 0, a); // First (0) indep. var
  Sacado::Fad::DFad<double> bfad(num_deriv, 1, b); // Second (1) indep. var
  Sacado::Fad::DFad<double> cfad(c); // Passive variable

  // Compute function
  double r = func(a, b, c);

  // Compute function and derivative with AD
  Sacado::Fad::DFad<double> rfad = func(afad, bfad, cfad);

  // Extract value and derivatives
  double r_ad = rfad.val(); // r
  double drda_ad = rfad.dx(0); // dr/da
  double drdb_ad = rfad.dx(1); // dr/db
  ```
Steady-state mass transfer equations:

\[ 2X_j = X_{j-1} + X_{j+1}, \ j = 2, \ldots, N - 1 \]

Set of \( N \) hypothetical chemical species:

\[ \mathbf{u} \cdot \nabla Y_j + \nabla^2 Y_j = \dot{\omega}_j, \ j = 1, \ldots, N - 1 \]

\[ \sum_{j=1}^{N} Y_j = 1 \]

Forward mode AD
- ✓ Faster than FD
- ✓ Better scalability in number of PDEs
- ✓ Analytic Derivative

Reverse mode AD
- ✓ Scalable adjoint/gradient

\[ J^T w = \nabla (w^T f(x)) \]

DOF per element = 4*N

Scalability of This Approach

Scalability of the element-level derivative computation:

Graphs showing the comparison of Jacobian and Adjoint evaluations for both Forward and Reverse modes, indicating scalability trends with respect to DOF per element.
Charon Drift-Diffusion Formulation with Defects

Current Conservation for e- and h+

\[
\frac{\partial n}{\partial t} - \nabla \cdot J_n = -R_n(\psi, n, p, Y_1, \ldots, Y_N), \quad J_n = -n\mu_n \nabla \psi + D_n \nabla n
\]

\[
\frac{\partial p}{\partial t} + \nabla \cdot J_p = -R_p(\psi, n, p, Y_1, \ldots, Y_N), \quad J_p = -p\mu_p \nabla \psi - D_p \nabla p
\]

Defect Continuity

\[
\frac{\partial Y_i}{\partial t} + \nabla \cdot J_{Y_i} = -R_{Y_i}(\psi, n, p, Y_1, \ldots, Y_N), \quad J_{Y_i} = -\mu_i Y_i \nabla \psi - D_i \nabla Y_i
\]

Electric potential

\[-\nabla (\varepsilon \nabla \psi(x)) = -q \left( p(x) - n(x) + N^+_D(x) - N^-_A(x) \right) - \sum_{i=1}^{N} q_i Y_i(x)\]

Recombination/generation source terms

\[R_X\]

Include electron capture and hole capture by defect species and reactions between various defect species

Electron emission/capture

\[Z^i \leftrightarrow Z^{i+1} + e^-\]

\[R_{[Z^i \rightarrow Z^{i+1} + e^-]} \propto \sigma_i Z^i \rightarrow Z^{i+1} + e^- \] \[Z^i \exp \left( \frac{\Delta E_{[Z^i \rightarrow Z^{i+1} + e^-]}}{kT} \right)\]

Cross section

Activation Energy