Analytic Sensitivities in Large-scale Production Applications via Automatic Differentiation with Sacado

Eric Phipps
Roscoe Bartlett, David Gay
Optimization & Uncertainty Quantification Department
Sandia National Laboratories
Albuquerque, NM USA

Parallel Processing for Scientific Computing
March 13, 2008
Analytic Derivatives Enable Robust Simulation and Design Capabilities

• We need analytic first & higher derivatives for predictive simulations
  – Computational design, optimization and parameter estimation
  – Stability analysis
  – Uncertainty quantification
  – Verification and validation

• Analytic derivatives improve robustness and efficiency
  – Very hard to make finite differences accurate

• Infeasible to expect application developers to code analytic derivatives
  – Time consuming, error prone, and difficult to verify
  – Thousands of possible parameters in a large code
  – Developers must understand what derivatives are needed

• Automatic differentiation solves these problems
What is Automatic Differentiation (AD)?

- Technique to compute analytic derivatives without hand-coding the derivative computation
- How does it work -- freshman calculus
  - Computations are composition of simple operations (+, *, sin(), etc...) with known derivatives
  - Derivatives computed line-by-line, combined via chain rule
- Derivatives accurate as original computation
  - No finite-difference truncation errors
- Provides analytic derivatives without the time and effort of hand-coding them

\[
y = \sin(e^x + x \log x), \quad x = 2
\]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>(x)</th>
<th>(\frac{d}{dx})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x \leftarrow 2)</td>
<td>(\frac{dx}{dx}) \leftarrow 1</td>
<td>2.000</td>
<td>1.000</td>
</tr>
<tr>
<td>(t \leftarrow e^x)</td>
<td>(\frac{dt}{dx}) \leftarrow t \frac{dx}{dx}</td>
<td>7.389</td>
<td>7.389</td>
</tr>
<tr>
<td>(u \leftarrow \log x)</td>
<td>(\frac{du}{dx}) \leftarrow \frac{1}{x} \frac{dx}{dx}</td>
<td>0.301</td>
<td>0.500</td>
</tr>
<tr>
<td>(v \leftarrow xu)</td>
<td>(\frac{dv}{dx}) \leftarrow u \frac{dx}{dx} + x \frac{du}{dx}</td>
<td>0.602</td>
<td>1.301</td>
</tr>
<tr>
<td>(w \leftarrow t + v)</td>
<td>(\frac{dw}{dx}) \leftarrow \frac{dt}{dx} + \frac{dv}{dx}</td>
<td>7.991</td>
<td>8.690</td>
</tr>
<tr>
<td>(y \leftarrow \sin w)</td>
<td>(\frac{dy}{dx}) \leftarrow \cos(w) \frac{dw}{dx}</td>
<td>0.991</td>
<td>-1.188</td>
</tr>
</tbody>
</table>
Automatic Differentiation Projects

• Many AD projects around the world, e.g.,
  – ADIFOR/ADIC (ANL, Rice) -- Fortran 77, C
  – OpenAD (ANL, Rice, Aachen) -- Fortran 77/95, C/C++ (maybe)
  – ADOL-C (TU-Desden) -- C/C++
  – TFAD<> -- C/C++

• Most source transformation tools limited to Fortran

• Most operator overloading based tools are slow
  – TFAD<> shows how to do this efficiently

• Many AD projects are geared towards “black-box” solutions

• We need efficient OO tools optimized for Sandia’s large-scale, parallel, C++ applications
Sandia Physics Simulation Codes

- **Element-based**
  - Finite element, finite volume, finite difference, network, etc…

- **Large-scale**
  - Billions of unknowns

- **Parallel**
  - MPI-based SPMD
  - Distributed memory

- **C++**
  - Object oriented
  - Some coupling to legacy Fortran libraries

- **We need AD techniques that fit these requirements**
Sacado: AD Tools for C++ Applications

• Trilinos package: www.trilinos.sandia.gov

• Implements several modes of AD
  – Forward (Jacobians, Jacobian-vector products, …)
  – Reverse (Gradients, Jacobian-transpose-vector products, …)
  – Taylor (High-order univariate Taylor series)

• AD via operator overloading and C++ templating
  – Expression templates for OO efficiency
  – Application code templating for easy incorporation

• Designed for use in large-scale C++ codes
  – Apply AD at “element-level”
  – Manually integrate derivatives into parallel data structures and solvers
  – Sacado::FEApp example demonstrates approach
// 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
```

Simple Sacado Example
Differentiating Element-Based Production Applications

- Global residual computation (ignoring boundary computations):

\[ f(\dot{x}, x, t, p) = \sum_{i=1}^{N} Q_i^T e_{k_i} (P_i \dot{x}, P_i x, t, p) \]

- Time-step Jacobian computation:

\[ \alpha \frac{\partial f}{\partial \dot{x}} + \beta \frac{\partial f}{\partial x} = \sum_{i=1}^{N} Q_i^T \left( \alpha \frac{\partial e_{k_i}}{\partial \dot{x}_i} + \beta \frac{\partial e_{k_i}}{\partial x_i} \right) P_i, \quad \dot{x}_i = P_i \dot{x}, \quad x_i = P_i x \]

- 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
Integrating AD with Solver Capabilities

Nonlinear Solvers

- NOX
- LOCA
- Rythmos

Abstract application interface

- Thyra::ModelEvaluator

Concrete application interface

- Application::ModelEvaluator

Method implementation

- Evaluate Jacobian
  - Loop over elements
  - gather element solution
  - initialize element AD types
  - evaluate templated element residual
  - extract derivative values
  - sum into global derivative matrix
  - end loop
Scalability of This Approach

Set of \( N \) hypothetical chemical species:
\[ 2X_j \rightleftharpoons X_{j-1} + X_{j+1}, \quad j = 2, \ldots, N - 1 \]

Steady-state mass transfer equations:
\[ \mathbf{u} \cdot \nabla Y_j + \nabla^2 Y_j = \dot{\omega}_j, \quad j = 1, \ldots, N - 1 \]
\[ \sum_{j=1}^{N} Y_j = 1 \]

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

Reverse mode AD
✓ Scalable adjoint/gradient
\[ J^T w = \nabla (w^T f(x)) \]
Defect reactions:

- **Si interstitial (I)** (+2, +1, 0, –1, –2)
  - Annihilation

- **Vacancy (V)** (+2, +1, 0, –1, –2)
  - Annihilation

- **B**
  - (+, 0, –)

- **B**
  - (+, 0, –)

- **C**
  - (+, 0, –)

- **VP** (0, –)

- **VB** (+, 0)

- **VO** (0, –)

- **B₁I**
  - (0, –)

- **B₁O** (+, 0)

- **B₁C**

- **B₁B** (0, –)

**QASPR**

Qualification of electronic devices in hostile environments

**Stockpile BJT**

**PDE semiconductor device simulation**
Charon Drift-Diffusion Formulation with Defects

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

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

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

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

\[R_{X} \quad \text{Include electron capture and hole capture by defect species and reactions between various defect species}\]

\[R_{Z^i \rightarrow Z^{i+1} + e^-} \propto \sigma_{Z^i \rightarrow Z^{i+1} + e^-} Z^i \exp \left(\frac{\Delta E_{Z^i \rightarrow Z^{i+1} + e^-}}{kT}\right)\]
Forward Sensitivity Analysis with Rythmos

- Discretized PDE system:

\[
f(\dot{x}, x, p, t) = 0 \\
\hat{g}(p, t) = g(\dot{x}(t), x(t), p, t)
\]

- Forward sensitivity problem

\[
\frac{\partial f}{\partial \dot{x}} \left( \frac{\partial \dot{x}}{\partial p} \right) + \frac{\partial f}{\partial x} \left( \frac{\partial x}{\partial p} \right) + \frac{\partial f}{\partial p} = 0 \\
\frac{\partial \hat{g}}{\partial p} = \frac{\partial g}{\partial \dot{x}} \frac{\partial \dot{x}}{\partial p} + \frac{\partial g}{\partial x} \frac{\partial x}{\partial p} + \frac{\partial g}{\partial p}
\]

- Rythmos time integration package
  - Todd Coffey, Ross Bartlett (SNL)
  - Implicit BDF time integration method
  - Variable order, step size
  - Staggered corrector forward sensitivity method
Sensitivity Analysis of a Pseudo-1D BJT

• 9x0.1 micron pseudo-1D simulation
• 1046x1 quad cells, linear finite elements + SUPG
• 2 carriers + 35 defect species
• 108,030 unknowns on 32 processors
• 84 carrier-defect reactions
• 126 parameters for sensitivity analysis
• AD Jacobian, parameter deriv’s
• Base current provides observation function

Radiation Pulse

Transient Base Current
# Transient Base Current Sensitivities

## Scaled Sensitivities

![Graph showing scaled sensitivities for different times and parameters.](image1)

## Unscaled Sensitivities

![Graph showing unscaled sensitivities for different times and parameters.](image2)

## Table

<table>
<thead>
<tr>
<th>#</th>
<th>Reaction</th>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>$V^- \rightarrow e^- + V^-$</td>
<td>activation energy</td>
<td>0.09</td>
</tr>
<tr>
<td>16</td>
<td>$e^- + V^0 \rightarrow V^-$</td>
<td>cross-section</td>
<td>2.40E-14</td>
</tr>
<tr>
<td>46</td>
<td>$e^- + PV^0 \rightarrow PV^0$</td>
<td>cross-section</td>
<td>1.50E-15</td>
</tr>
</tbody>
</table>
Comparison to Black-Box Finite Differences

• Run-times:
  – Forward simulation: 105 min.
  – Direct sensitivities: 931 min.
  – Black-box, first-order FD: ~13,000 min.
• Direct approach more efficient
  – 14x speed-up
• Direct approach more accurate
  – 1-2 correct digits w/FD
  – FD requires tighter tolerances to achieve higher accuracy
• Direct approach more robust
  – Accuracy solely dictated by time-integration error
Summary

• Templating key to AD approach
  – Simple, fast Sacado AD tools
  – Apply at “element” level
  – Hooks for future program transformation

• Vertical integration of Trilinos technologies provide remarkable capabilities
  – Efficient, accurate, robust sensitivities
  – Foundation for transient optimization
  – Requires all levels to be effective
Auxiliary Slides
Difficulties With This Approach

• Template code introduces excessive compiler overhead
  – Explicit template instantiation
  – Preprocessor macros make this easy

• Real codes always call other libraries
  – BLAS/LAPACK
  – CHEMKIN
  – Linear/nonlinear solvers
  – Template interfaces (partial template specialization) are general solution

• Operator overloading overhead
  – Residual/derivative fills are not dominant cost