SAND2006-7054C

Automatic Differentiation of C++ Codes With Sacado

Eric Phipps and David Gay

Sandia National Laboratories

Trilinos User's Group Meeting

November 7, 2006

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company,
for the United States Department of Energy's National Nuclear Security Administration
under contract DE-AC04-94AL85000.

Outline

- Introduction to automatic differentiation
  - Forward mode
  - Reverse mode
  - Taylor polynomial mode
- Software implementations
  - Source Transformation
  - Operator Overloading
- Sacado
  - Forward
  - Reverse
  - Higher derivatives
- Derivatives for nonlinear algorithms
- Differentiating large-scale element-based codes

What Is Automatic Differentiation (AD) ?

- All differentiable computations are composition of simple operations
  - sin(), log(), +, *, /, etc…
- We know the derivatives of these simple operations
- We have the chain rule from calculus
- Systematic application of the chain rule through your computation differentiating each statement line-by-line.

A Simple Example

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

	x	$\frac{d}{dx}$
$x \leftarrow 2$	2.000	1.000
$t_1 \leftarrow e^x$	7.389	7.389
$t_2 \leftarrow \log x$	0.693	0.500
$t_3 \leftarrow x t_2$	1.386	1.693
$t_4 \leftarrow t_1 + t_3$	8.775	9.082
$y \leftarrow \sin t_4$	0.605	-7.233

Analytic derivative evaluated to machine precision

Related Methods

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

Automatic Differentiation

$$\begin{aligned}
x &\leftarrow 2 & \frac{dx}{dx} &\leftarrow 1 \\
t_1 &\leftarrow e^x & \frac{dt_1}{dx} &\leftarrow t_1\frac{dx}{dx} \\
t_2 &\leftarrow \log x & \frac{dt_2}{dx} &\leftarrow \frac{1}{x}\frac{dx}{dx} \\
t_3 &\leftarrow xt_2 & \frac{dt_3}{dx} &\leftarrow t_2\frac{dx}{dx} + x\frac{dt_2}{dx} \\
t_4 &\leftarrow t_1 + t_3 & \frac{dt_4}{dx} &\leftarrow \frac{dt_1}{dx} + \frac{dt_3}{dx} \\
y &\leftarrow \sin t_4 & \frac{dy}{dx} &\leftarrow \cos(t_4)\frac{dt_4}{dx}
\end{aligned}$$

$$\frac{dy}{dx} = 7.233\ 340\ 400\ 802\ 3158$$

Symbolic Differentiation

$$\frac{dy}{dx} = \cos(e^x + x\log x)\cdot (e^x + \log x + 1)$$

$$\begin{aligned}
x &\leftarrow 2 \\
t_1 &\leftarrow e^x \\
t_2 &\leftarrow \log x \\
t_3 &\leftarrow xt_2 \\
t_4 &\leftarrow t_1 + t_3 \\
y &\leftarrow \sin t_4 \\
\hline
s_1 &\leftarrow \cos t_4 \\
s_2 &\leftarrow t_1 + t_2 \\
s_3 &\leftarrow s_2 + 1 \\
\frac{dy}{dx} &\leftarrow s_1 s_3
\end{aligned}$$

$$\frac{dy}{dx} = 7.233\ 340\ 400\ 802\ 3167$$

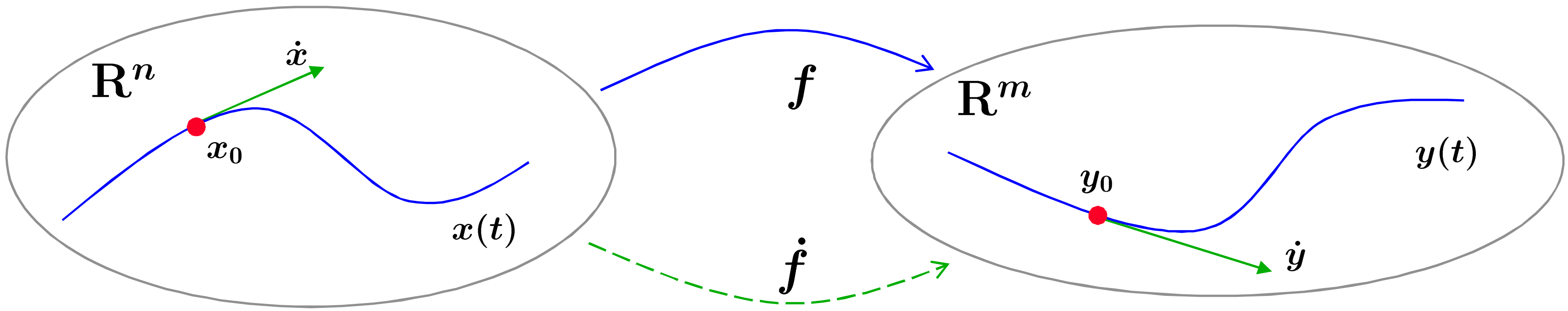
Finite Differencing

$$\frac{dy}{dx} \approx \frac{y(2+\varepsilon) - y(2)}{\varepsilon} \approx 7.233\ 343\ 187$$

Sandia National Laboratories

Tangent Propagation

$$y = f(x),\ f : \mathrm{R}^n \rightarrow \mathrm{R}^m$$



- Tangents

$$y(t) = f(x(t)) \implies \dot{y} \equiv \left.\frac{dy}{dt}\right|_{t=t_0} = \frac{\partial f}{\partial x}\dot{x}$$

- For each intermediate operation

$$c = \varphi(a,b) \implies \dot{c} = \frac{\partial \varphi}{\partial a}\dot{a} + \frac{\partial \varphi}{\partial b}\dot{b}$$

- Tangents map forward through evaluation

Operation	Tangent Rule
$c = a + b$	$\dot{c} = \dot{a} + \dot{b}$
$c = a - b$	$\dot{c} = \dot{a} - \dot{b}$
$c = ab$	$\dot{c} = a\dot{b} + \dot{a}b$
$c = a/b$	$\dot{c} = (\dot{a} - c\dot{b})/b$
$c = a^b$	$\dot{c} = c(\dot{b}\log(a) + \dot{a}b/a)$
$c = \sin(a)$	$\dot{c} = \cos(a)\dot{a}$
$c = \log(a)$	$\dot{c} = \dot{a}/a$

Sandia National Laboratories

A Simple Tangent Example

$$y_1 = \sin(e^{x_1} + x_1 x_2)$$
$$y_2 = \frac{y_1}{y_1 + x_1^2}$$

$$\begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \end{bmatrix} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix}$$

Given x_1, x_2, $\dot{x}_1$, $\dot{x}_2$:

$$\begin{aligned}
s_1 &\leftarrow e^{x_1} & \dot{s}_1 &\leftarrow s_1 \dot{x}_1 \\
s_2 &\leftarrow x_1 x_2 & \dot{s}_2 &\leftarrow x_1 \dot{x}_2 + \dot{x}_1 x_2 \\
s_3 &\leftarrow s_1 + s_2 & \dot{s}_3 &\leftarrow \dot{s}_1 + \dot{s}_2 \\
y_1 &\leftarrow \sin(s_3) & \dot{y}_1 &\leftarrow \cos(s_3) \dot{s}_3 \\
s_4 &\leftarrow x_1^2 & \dot{s}_4 &\leftarrow 2 x_1 \dot{x}_1 \\
s_5 &\leftarrow y_1 + s_4 & \dot{s}_5 &\leftarrow \dot{y}_1 + \dot{s}_4 \\
y_2 &\leftarrow y_1 / s_5 & \dot{y}_2 &\leftarrow (\dot{y}_1 - y_2 \dot{s}_5)/s_5
\end{aligned}$$

Return y_1, y_2, $\dot{y}_1$, $\dot{y}_2$

Forward Mode AD via Tangent Propagation

- Choice of space curve $x(t)$ is arbitrary
- Tangent $\dot{y}$ depends only on x_0, $\dot{x}$
- Given x_0 and v:

$$y(t) = f(x_0 + vt) \implies \dot{y} = \frac{\partial f}{\partial x_0} v$$

Jacobian vector product

- Propagate p vectors $v_1, \dots, v_p$ simultaneously

$$[\dot{y}_1 \dots \dot{y}_p] = \frac{\partial f}{\partial x_0} [v_1 \dots v_p] = \frac{\partial f}{\partial x_0} V$$

Jacobian matrix product

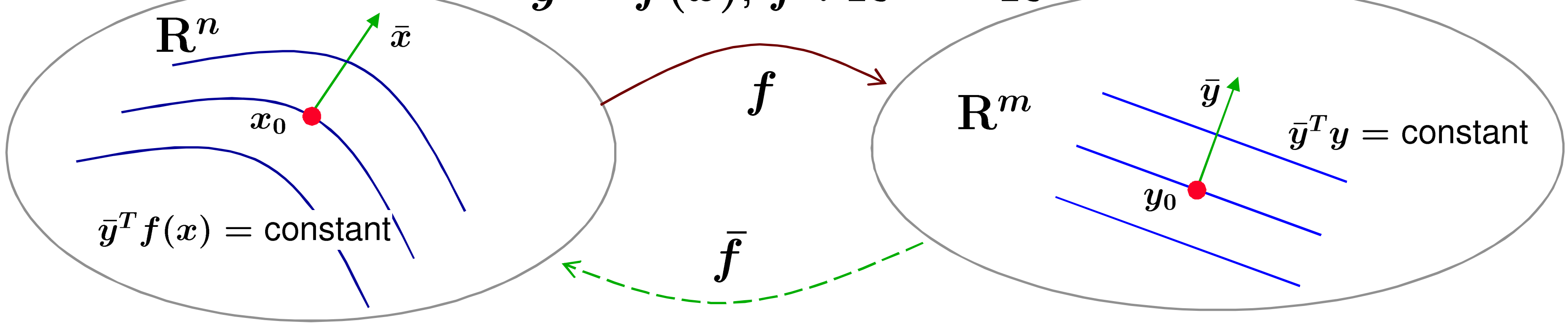
- Forward mode AD:

$$(x, V) \rightarrow \left(f(x), \frac{\partial f}{\partial x} V \right)$$

- V is called the seed matrix. Setting equal to identity matrix yields full Jacobian
- Computational cost $\approx (1 + 1.5p)$time(f)
- Jacobian-vector products, directional derivatives, Jacobians for $m \geq n$

Gradient Propagation

$$y = f(x),\ f : \mathrm{R}^n \to \mathrm{R}^m$$



- Gradients

$$z = \bar{y}^T y = \bar{y}^T f(x) \implies \bar{x} \equiv \left(\frac{\partial z}{\partial x}\right)^T = \left(\frac{\partial f}{\partial x}\right)^T \bar{y}$$

- For each intermediate operation

$$c = \varphi(a,b) \implies \begin{aligned} \bar{a} &= \frac{\partial z}{\partial a} = \frac{\partial z}{\partial c}\frac{\partial c}{\partial a} = \bar{c}\frac{\partial \varphi}{\partial a}, \\ \bar{b} &= \frac{\partial z}{\partial b} = \frac{\partial z}{\partial c}\frac{\partial c}{\partial b} = \bar{c}\frac{\partial \varphi}{\partial b} \end{aligned}$$

- Gradients map backward through evaluation

Operation	Gradient Rule
$c = a + b$	$\bar{a} = \bar{c}, \quad \bar{b} = \bar{c}$
$c = a - b$	$\bar{a} = \bar{c}, \quad \bar{b} = -\bar{c}$
$c = ab$	$\bar{a} = \bar{c}b, \quad \bar{b} = \bar{c}a$
$c = a/b$	$\bar{a} = \bar{c}/b, \quad \bar{b} = -\bar{c}c/b$
$c = a^b$	$\bar{a} = \bar{c}c\log(a), \ \bar{b} = \bar{c}cb/a$
$c = \sin(a)$	$\bar{a} = \bar{c}\cos(a)$
$c = \log(a)$	$\bar{a} = \bar{c}/a$

Sandia National Laboratories

A Simple Gradient Example

$$y_1 = \sin(e^{x_1} + x_1 x_2)$$
$$y_2 = \frac{y_1}{y_1 + x_1^2}$$

$$\begin{bmatrix} \bar{x}_1 \\ \bar{x}_2 \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \end{bmatrix}^T \begin{bmatrix} \bar{y}_1 \\ \bar{y}_2 \end{bmatrix}$$

$$c = \varphi(a, b) \implies \begin{aligned} \bar{a} &= \bar{c}\frac{\partial \varphi}{\partial a}, \\ \bar{b} &= \bar{c}\frac{\partial \varphi}{\partial b} \end{aligned}$$

Given x_1, x_2, $\bar{y}_1$, $\bar{y}_2$:

$s_1 \leftarrow e^{x_1}$

$s_2 \leftarrow x_1 x_2$

$s_3 \leftarrow s_1 + s_2$

$y_1 \leftarrow \sin(s_3)$

$s_4 \leftarrow x_1^2$

$s_5 \leftarrow y_1 + s_4$

$y_2 \leftarrow y_1 / s_5$

$\bar{y}_1 \leftarrow \bar{y}_1 + \bar{y}_2 / s_5, \quad \bar{s}_5 \leftarrow -y_2 \bar{y}_2 / s_5$

$\bar{y}_1 \leftarrow \bar{y}_1 + \bar{s}_5, \quad \bar{s}_4 \leftarrow \bar{s}_5$

$\bar{x}_1 \leftarrow 2\bar{s}_4 x_1$

$\bar{s}_3 \leftarrow \bar{y}_1 \cos(s_3)$

$\bar{s}_1 \leftarrow \bar{s}_3, \quad \bar{s}_2 \leftarrow \bar{s}_3$

$\bar{x}_1 \leftarrow \bar{x}_1 + \bar{s}_2 x_2, \quad \bar{x}_2 \leftarrow \bar{s}_2 x_1$

$\bar{x}_1 \leftarrow \bar{x}_1 + \bar{s}_1 s_1$

Return y_1, y_2, $\bar{x}_1$, $\bar{x}_2$

Reverse Mode AD via Gradient Propagation

- Choice of normal $\bar{y}$ is arbitrary
- Gradient $\bar{x}$ depends only on $x_0, \bar{y}$
- Given x_0 and w:

$$\bar{y} = w, y = f(x) \implies \bar{x} = \left(\frac{\partial f}{\partial x}\right)^T w$$

Jacobian-transpose vector product

- Propagate p vectors $w_1, \ldots, w_p$ simultaneously

$$[\bar{x}_1 \ldots \bar{x}_p] = \left(\frac{\partial f}{\partial x}\right)^T [w_1 \ldots w_p] = \left(\frac{\partial f}{\partial x}\right) W$$

Jacobian-transpose matrix product

- Reverse mode AD:

$$(x, W) \to \left(f(x), \left(\frac{\partial f}{\partial x}\right)^T W\right)$$

- W is called the seed matrix. Setting equal to identity matrix yields full Jacobian
- Computational cost $\approx (1.5 + 2.5p)\text{time}(f)$ $\quad m = p = 1 \implies$ cost ≈ 4 time(f)
- Jacobian-transpose products, gradients, Jacobians for $n > m$

Sandia National Laboratories

Taylor Polynomial Propagation

$$y = f(x),\ f : \mathrm{R}^n \to \mathrm{R}^m$$

- Extension of tangent propagation to higher degree

- Given $d+1$ coefficients $x_0, \ldots, x_d \in \mathrm{R}^n$

$$x(t) \equiv \sum_{k=0}^{d} x_k t^k$$

$$y(t) \equiv f(x(t)) = \sum_{k=0}^{d} y_k t^k + O(t^{d+1})$$

$$y_k \equiv \frac{1}{k!} \frac{d^k y}{dt^k}\bigg|_{t=0} = y_k(x_0, \ldots, x_k)$$

Operation	Taylor Rule
$c = a + b$	$c_k = a_k + b_k$
$c = a - b$	$c_k = a_k - b_k$
$c = ab$	$c_k = \sum_{j=0}^{k} a_j b_{k-j}$
$c = a/b$	$c_k = \frac{1}{b_0}\left(a_k - \sum_{j=1}^{k} b_j c_{k-j}\right)$
$c = \exp(a)$	$c_k = \frac{1}{k}\sum_{j=1}^{k} j c_{k-j} a_j$
$c = \log(a)$	$c_k = \frac{1}{k a_0}\left(k a_k - \sum_{j=1}^{k-1} j a_{k-j} c_j\right)$
$s = \sin(a)$ $c = \cos(a)$	$s_k = \frac{1}{k}\sum_{j=1}^{k} j a_j c_{k-j}$ $c_k = -\frac{1}{k}\sum_{j=1}^{k} j a_j s_{k-j}$

- Computational cost $\approx O(d^2)\text{time}(f)$

Sandia National Laboratories

Software Implementations

- Source transformation
  - Preprocessor reads code to be differentiated, uses AD to generate derivative code, writes-out differentiated code in original source language which is then compiled using a standard compiler
  - Resulting derivative computation is usually very efficient
  - Works well for simple languages (FORTRAN, some C)
  - ADIFOR, ADIC out of Argonne
  - Extremely difficult for C++ (no existing tool)
- Operator overloading
  - New data types are created for forward, reverse, Taylor modes, and intrinsic operations/elementary operations are overloaded to compute derivatives as a side-effect
  - Generally easy to incorporate into C++ codes
  - Generally slower than source transformation due to function call overhead
  - Requires changing data types from floats/doubles to AD types
    - C++ templates greatly help
  - ADOL-C (slow), FAD/TFAD (fast)

Sandia National Laboratories

ADIFOR* Example

```
      subroutine func(x, y)
C
      double precision x(2), y(2)
      double precision u, v, w
C
      u = exp(x(1))
      v = x(1)*x(2)
      w = u+v
      y(1) = sin(w)
C
      u = x(1)**2
      v = y(1) + u
      y(2) = y(1)/v
C
      return
      end
```

*ADIFOR 2.0D
www-unix.mcs.anl.gov/autodiff/ADIFOR/

```
      subroutine g_func(g_p_, x, g_x, ldg_x, y, g_y, ldg_y)

C     Initializations removed for clarity…

      d2_v = exp(x(1))
      d1_p =  d2_v
      do g_i_ = 1, g_p_
        g_u(g_i_) = d1_p * g_x(g_i_, 1)
      enddo
      u = d2_v
C--------
      do g_i_ = 1, g_p_
        g_v(g_i_) = x(1) * g_x(g_i_, 2) + x(2) * g_x(g_i_, 1)
      enddo
      v = x(1) * x(2)
C--------
      do g_i_ = 1, g_p_
        g_w(g_i_) = g_v(g_i_) + g_u(g_i_)
      enddo
      w = u + v
C--------
      d2_v = sin(w)
      d1_p = cos(w)
      do g_i_ = 1, g_p_
        g_y(g_i_, 1) = d1_p * g_w(g_i_)
      enddo
      y(1) = d2_v

C    continues…
```

(Naive) Operator Overloading Example

```
void func(const double x[], double y[]) {
  double u, v, w;
  u = exp(x[0]);
  v = x[0]*x[1];
  w = u+v;
  y[0] = sin(w);

  u = x[0]*x[0];
  v = y[0] + u;
  y[1] = y[0]/v;
}

void func(const Tangent x[], Tangent y[]) {
  Tangent u, v, w;
  u = exp(x[0]);
  v = x[0]*x[1];
  w = u+v;
  y[0] = sin(w);

  u = x[0]*x[0];
  v = y[0] + u;
  y[1] = y[0]/v;
}
```

```
class Tangent {
public:
  static const int N = 2;
  double val;
  double dot[N];
};

Tangent operator+(const Tangent& a, const Tangent& b) {
  Tangent c;
  c.val = a.val + b.val;
  for (int i=0; i<Tangent::N; i++)
    c.dot[i] = a.dot[i] + b.dot[i];
  return c;
}
Tangent operator*(const Tangent& a, const Tangent& b) {
  Tangent c;
  c.val = a.val * b.val;
  for (int i=0; i<Tangent::N; i++)
    c.dot[i] = a.val * b.dot[i] + a.dot[i]*b.val;
  return c;
}
Tangent exp(const Tangent& a) {
  Tangent c;
  c.val = exp(a.val);
  for (int i=0; i<Tangent::N; i++)
    c.dot[i] = c.val * a.dot[i];
  return c;
}
```

Introducing Sacado
(The Package Formerly Known as ADTools)

- New Trilinos package for automatic differentiation of C++ codes
- Loosely translated as “I have derived” in Spanish
- Developers: Dave Gay, Eric Phipps (with contributions from Ross Bartlett)
- Not in Trilinos 7, will be released with Trilinos 8 (Spring ‘07)
- Forward AD
  – Based on expression template-based public domain Fad package
- Reverse AD
  – Dave Gay’s Rad package
- Univariate Taylor polynomials
- Application support utilities
  – Template metaprogramming
- Coming soon
  – Ross’ ScalarFlopCounter
  – Multi-variate Taylor polynomials
  – Stochastic polynomial chaos expansions
  – Teuchos BLAS/LAPACK wrapper specializations
- Depends on Teuchos only (and currently only through tests)

Sandia National Laboratories

The Usual Suspects

- Configure options
  - `--enable-sacado` — Enables Sacado at Trilinos top-level
  - `--enable-sacado-tests, --enable-tests` — Enables unit, regression, and performance tests
    - `--with-cppunit-prefix=[path]` — Path to CppUnit for unit tests
    - `--with-adolc=[path]` — Enables Taylor polynomial unit tests with ADOL-C
    - `--enable-sacado-alltests` — Enables additional tests that take a VERY LONG time to compile
  - `--enable-sacado-examples, --enable-examples` — Enables examples
    - `--enabled-sacado-fem-example` — Enables a 1D FEM example application (additional dependencies on Epetra, NOX, LOCA, MOOCHO, Rythmos, …)
- Mailing lists
  - Sacado-announce@software.sandia.gov
  - Sacado-checkins@software.sandia.gov
  - Sacado-developers@software.sandia.gov
  - Sacado-regression@software.sandia.gov
  - Sacado-users@software.sandia.gov
- Bugzilla: http://software.sandia.gov/bugzilla
- Bonsai: http://software.sandia.gov/bonsai/cvsqueryform.cgi
- Web: http://software.sandai.gov/Trilinos/packages/sacado
- Doxygen documentation

Using Sacado

- As always: `#include "Sacado.hpp"`

- All classes are templated on the Scalar type

- Forward AD classes:
  - `Sacado::Fad::DFad<ScalarT>`: Derivative array is allocated dynamically
  - `Sacado::Fad::SFad<ScalarT>`: Derivative array is allocated statically and dimension must be known at compile time
  - `Sacado::Fad::SLFad<ScalarT>`: Like SFad except allocated length may be greater than "used" length

- Reverse mode AD classes:
  - `ADvar<ScalarT> (Sacado_trad.h)`

- Taylor polynomial classes:
  - `Sacado::Taylor::DTaylor<ScalarT>`

Sandia National Laboratories

sacado/example/dfad_example.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

  // Fad objects
  Sacado::Fad::DFad<double> afad(num_deriv, 0, a); // First (0) indep. var.  Derivative array is [1.0 0.0]
  Sacado::Fad::DFad<double> bfad(num_deriv, 1, b); // Second (1) indep. var.  Derivative array is [0.0 1.0]
  Sacado::Fad::DFad<double> cfad(c);               // Passive variable
  Sacado::Fad::DFad<double> rfad;                  // Result

  double r = func(a, b, c);                        // Compute function
  rfad = func(afad, bfad, cfad);                   // Compute function and derivative with AD

  // 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
```

sacado/example/sfad_example.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

  // Fad objects
  Sacado::Fad::SFad<double,2> afad(num_deriv, 0, a); // First (0) indep. var.  Derivative array is [1.0 0.0]
  Sacado::Fad::SFad<double,2> bfad(num_deriv, 1, b); // Second (1) indep. var.  Derivative array is [0.0 1.0]
  Sacado::Fad::SFad<double,2> cfad(c);               // Passive variable
  Sacado::Fad::SFad<double,2> rfad;                  // Result

  double r = func(a, b, c);                          // Compute function
  rfad = func(afad, bfad, cfad);                     // Compute function and derivative with AD

  // 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
```

sacado/example/trad_example.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

  // Rad objects
  Sacado::Rad::ADvar<double> arad = a;
  Sacado::Rad::ADvar<double> brad = b;
  Sacado::Rad::ADvar<double> crad = c;              // Passive variable
  Sacado::Rad::ADvar<double> rrad;                  // Result
  double r = func(a, b, c);                         // Compute function
  rrad = func(arad, brad, crad);                    // Compute function and derivative with AD
  Sacado::Rad::ADVar<double>::Gradcomp();           // Compute gradients

  // Extract value and derivatives
  double r_ad = rrad.val();     // r
  double drda_ad = arad.adj();  // dr/da
  double drdb_ad = brad.adj();  // dr/db
```

Computing Higher Derivatives

- AD classes are templated, so AD classes can be nested to compute higher derivatives
  - Forward-forward: $y = f(x) \xrightarrow{\text{for}} \frac{\partial y}{\partial x} v_1 \xrightarrow{\text{for}} \frac{\partial}{\partial x}\left(\frac{\partial y}{\partial x} v_1\right) v_2$
  - Reverse-forward: $y = f(x) \xrightarrow{\text{rev}} w^T \frac{\partial y}{\partial x} \xrightarrow{\text{for}} \frac{\partial}{\partial x}\left(w^T \frac{\partial y}{\partial x}\right) v$
  - Forward-Taylor: $y_0 = f(x_0) \xrightarrow{\text{for}} \frac{\partial y_0}{\partial x_0} v \xrightarrow{\text{tay}} \frac{\partial y_k}{\partial x_0} v$
  - Reverse-Taylor: $y_0 = f(x_0) \xrightarrow{\text{rev}} w^T \frac{\partial y_0}{\partial x_0} \xrightarrow{\text{tay}} w^T \frac{\partial y_k}{\partial x_0}$
  - Etc…

Forward or Reverse?

- Forward
  - Number of independent variables <= number of dependent variables
  - Square Jacobians for Newton's method
  - Sensitivities with small numbers of parameters
  - Algorithm naturally calls for Jacobian-vector/matrix products
    - (Block) Matrix-free Newton-Krylov

- Reverse
  - Number of independent variables > number of dependent variables + 40
  - Gradients of scalar valued functions
  - Sensitivities with large numbers of parameters
  - Algorithm naturally calls for Jacobian-transpose-vector/matrix products
    - (Block) Matrix-free solves of transpose matrix
    - Optimization

Differentiating Element-Based Codes

- Global residual computation (ignoring boundary computations):

$$f(x) = \sum_{i=1}^{N} Q_i^T e_{k_i}(P_i x)$$

- Jacobian computation:

$$\frac{\partial f}{\partial x} = \sum_{i=1}^{N} Q_i^T J_{k_i} P_i, \quad J_{k_i} = \frac{\partial e_{k_i}}{\partial x_i}, \quad x_i = P_i x$$

- Jacobian-transpose product computation:

$$w^T \frac{\partial f}{\partial x} = \sum_{i=1}^{N} (Q_i w)^T J_{k_i} P_i$$

- Hybrid symbolic/AD procedure
  - Element-level derivatives computed via AD
  - Exactly the same as how you would do this “manually”
  - Avoids parallelization issues

AD for Element-Level Derivatives

- Element computations are
  - Narrow and shallow
  - Dense
- Template application’s element code via C++ templates
  - App developers code and maintain single templated code base
  - Easy to add new AD types
- Ideas developed within Charon
  - Large scale finite element PDE semiconductor device simulation

Performance

Set of N hypothetical chemical species:

$$2X_j \rightleftharpoons X_{j-1} + X_{j+1}, \quad j = 2, \dots, N-1$$

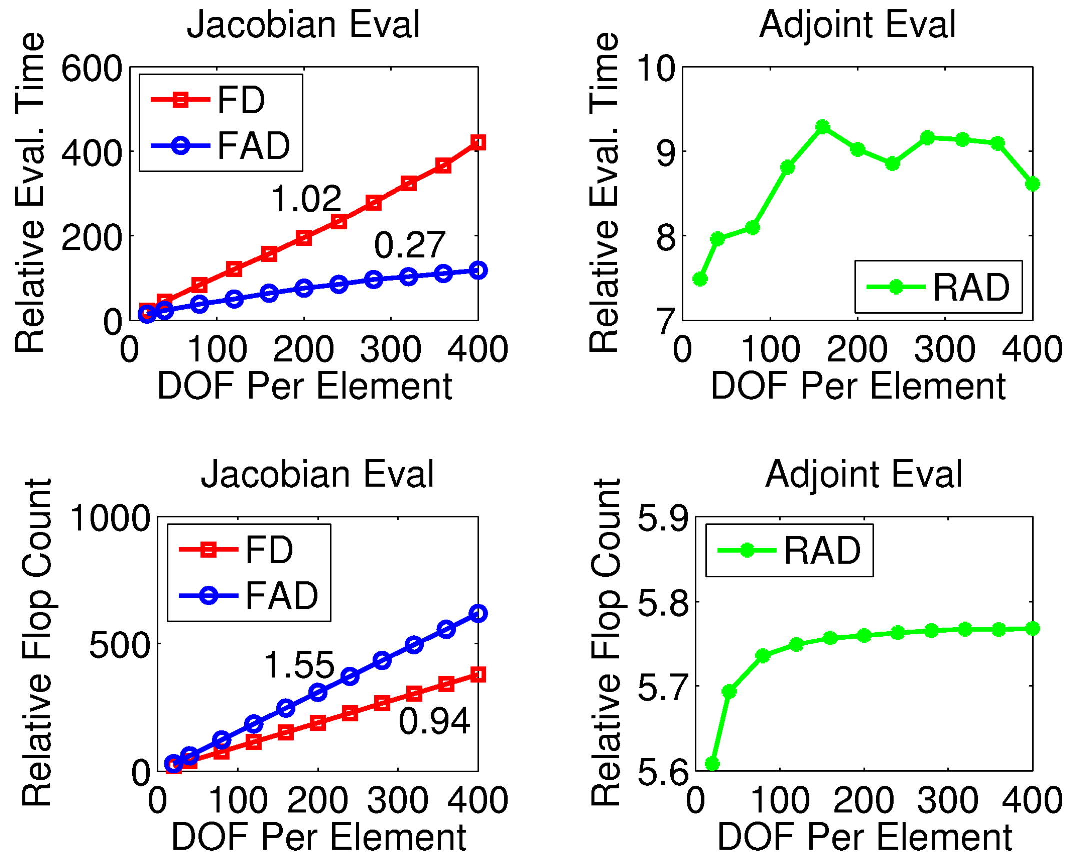
Steady-state mass transfer equations:

$$\mathbf{u} \cdot \nabla Y_j + \nabla^2 Y_j = \dot{\omega}_j, \quad j = 1, \dots, N-1$$

$$\sum_{j=1}^{N} Y_j = 1$$

- Forward mode AD
  - Faster than FD
  - Better scalability in number of PDEs
  - Analytic derivative
  - Provides Jacobian for all Charon physics
- Reverse mode AD
  - Scalable adjoint/gradient $J^T w = \nabla(w^T f(x))$

Scalability of the element-level derivative computation



DOF per element = 4*N

Sandia National Laboratories

How to use Sacado

- Template code to be differentiated
- Replace independent variables with AD variables
- Initialize seed matrix
- Evaluate function on AD variables
- Extract derivatives

Best Practices

- Don't differentiate your global function with AD
- Only use AD for the hard, nonlinear parts
- Do as much up-front as possible
- Never differentiate solvers with AD…instead use AD for the derivative of the solution

$$f(x,p) = 0 \implies \left(\frac{\partial f}{\partial x}\right)\frac{dx}{dp} + \frac{\partial f}{\partial p} = 0 \implies \frac{dx}{dp} = \left(\frac{\partial f}{\partial x}\right)^{-1}\frac{\partial f}{\partial p}$$

- Always put Fad inside of Rad, not the other way around

Auxiliary Slides

Why is this important?

- Speed
  - Filling a sparse Jacobian is significantly faster using forward mode AD than finite differences (FD)
  - Adjoints can be computed in a time independent of the number of independent variables using reverse mode AD
  - Higher derivatives enable more efficient algorithms
- Accuracy
  - AD produces analytic derivatives that are accurate to machine precision
  - Generally impossible to get accurate higher derivatives using FD
- Robustness
  - Analytic derivatives improve robustness of algorithms
  - Coupling of analysis algorithms requires higher derivatives
- Code maintenance
  - Hand-writing derivative code is tedious, time consuming, error prone
  - Code developers will never hand code higher derivatives
  - Using AD, code developers only need to code residuals
  - Any derivative required by an analysis tool can be computed using AD

RAD: Specialized overloading for *rf*

- Reverse mode AD tool developed by David Gay
- AD data type `ADvar`
- Forward sweep: evaluate residual fill on `Advar` type
  - Computes values, partials
  - Store values, partials, and connectivity in "tape"
- Reverse sweep: `ADcontext::Gradcomp();`
  - Accumulates adjoints
  - Reclaims "tape" memory for future gradient computations
- Block memory allocation for efficiency.
- Templated to support higher derivatives (e.g., Hessians)

$$c = \varphi(a,b) \implies \frac{\partial y_j}{\partial a} \mathrel{+}= \frac{\partial y_j}{\partial c}\frac{\partial \varphi}{\partial a}, \quad \frac{\partial y_j}{\partial b} \mathrel{+}= \frac{\partial y_j}{\partial c}\frac{\partial \varphi}{\partial b}$$

```
void ADcontext::Gradcomp() {
   Derp *d = Derp::LastDerp;
   d->c->aval = 1;
   for(; d; d = d->next)
      d->a->aval += d->da * d->c->aval;
      // … (arrange to recycle memory)
```

- Significantly less overhead than other approaches
  - Computes and stores partials on forward sweep
  - Only memory get/put, +, * on reverse sweep
- ADOL-C
  - Stores representation of each operation (name) and computes partial on reverse sweep
  - Allows reuse of tape

Sandia National Laboratories

RAD Performance

Mesh quality metric from Pat Knupp

$$\tau = det(AW^{-1})$$

$$h = 0.5(\tau + \sqrt{\tau^2 + 4\delta^2})$$

$$\mu_1 = \frac{\|AW^{-1} - I\|_F^2}{h^{2/3}}$$

A = element coordinate differences
W = ideal element shape (fixed).

Relative Times (*f* + r*f*) for *f* = μ_1	
Handcoded gradient	1.07
RAD	9.14
nlc	1.00
ADOL-C new tape	55.0
ADOL-C old tape	15.4

Templating Application Codes for AD

- Our interface to an application code is a templated elemental residual fill
- Templating makes it easy to interchange AD data types
- Developers only need to code residuals

fillResidual.h
```
void fillResidual(double *x, double *f);
```

fillResidual.C
```
void fillResidual(double *x, double *f) {
  // Fill elemental residual
  …
}
```

main.C
```
#include "fillResidual.h"

int main() {
  double *x, *f;
  …
  // Fill residual
  fillResidual(x,f)
  …
}
```

fillResidual.h
```
template <typename ScalarT>
void fillResidual(ScalarT *x, ScalarT *f);

#include "fillResidualImpl.h"
```

fillResidualImpl.h
```
template <typename ScalarT>
void fillResidual(ScalarT *x, ScalarT *f) {
  // Fill elemental residual
  …
}
```

main.C
```
#include "fillResidual.h"

int main() {
  double *x, *f;
  Fad<double> *x_fad, *f_fad;
  …
  // Fill residual
  fillResidual<double>(x,f)

  // Fill Jacobian
  fillResidual< Fad<double> >(x_fad, f_fad);
  …
}
```

Sandia National Laboratories

Templating Application Codes for AD

- Developed reusable "template infrastructure" components:
  - Macros that explicitly instantiate template classes/functions preserving the original layout of the application code/framework into separate translation units
  - Sequence containers and iterators storing all instantiations of template classes/functions providing access to instantiations, independent of the number and types of instantiations
  - Adapators interfacing templated AD code to non-templated derivative code (e.g., source transformation of FORTRAN)
- Eliminates a significant obstacle to widespread use of AD at Sandia
  - Easier to create/manage templated code
  - Easy to add AD types for new derivative computations

fillResidual.h

```
template <typename ScalarT>
void fillResidual(ScalarT *x, ScalarT *f);
```

fillResidual.C

```
template <typename ScalarT>
void fillResidual(ScalarT *x, ScalarT *f) {
  // Fill elemental residual
  …
}
template void fillResidual<double>(…);
template void fillReisudal< Fad<double> >(…);
```

main.C

```
#include "fillResidual.h"

int main() {
  double *x, *f;
  Fad<double> *x_fad, *f_fad;
  …
  // Fill residual
  fillResidual<double>(x,f)

  // Fill Jacobian
  fillResidual< Fad<double> >(x_fad, f_fad);
  …
}
```

Sandia National Laboratories

AD With Expression Templates

Expression: $d = a + b * c$

Traditional AD code

$$
* \left\{
\begin{array}{l}
t_0 = b_0 * c_0 \\
\textbf{for } i = 1 : n \\
\quad t_i = b_i * c_0 + b_0 * c_i \\
\textbf{end}
\end{array}
\right.
$$

$$
+ \left\{
\begin{array}{l}
u_0 = a_0 + t_0 \\
\textbf{for } i = 1 : n \\
\quad u_i = a_i + t_i \\
\textbf{end}
\end{array}
\right.
$$

$$
= \left\{
\begin{array}{l}
d_0 = u_0 \\
\textbf{for } i = 1 : n \\
\quad d_i = u_i \\
\textbf{end}
\end{array}
\right.
$$

Expression template code

$$
\begin{array}{l}
d_0 = a_0 + b_0 * c_0 \\
\textbf{for } i = 1 : n \\
\quad d_i = a_i + b_i * c_0 + b_0 * c_i \\
\textbf{end}
\end{array}
$$

Uses templates, lots of compiler optimization.

Requires a very good optimizing compiler!