SAND2007-7474P

Automatic Differentiation of C++ Codes With Sacado

Eric Phipps and David Gay

Sandia National Laboratories

Software Engineering Seminar Series

November 13, 2007

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 via tangent propagation
- Sacado Trilinos package
  - Operator Overloading
- Differentiating large-scale element-based codes
  - Sacado::FEApp example FEM application
- Complications/advanced concepts
  - Explicit template instantiation
  - Conditionals/non-differentiability
  - Expression templates

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

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

	x	$\frac{d}{dx}$
$x \leftarrow 2$	2.000	1.000
$t \leftarrow e^x$	7.389	7.389
$u \leftarrow \log x$	0.301	0.500
$v \leftarrow xu$	0.602	1.301
$w \leftarrow t + v$	7.991	8.690
$y \leftarrow \sin w$	0.991	-1.188

Sandia National Laboratories

Related Methods

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

Automatic Differentiation

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

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

Symbolic Differentiation

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

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

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

Finite Differencing

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

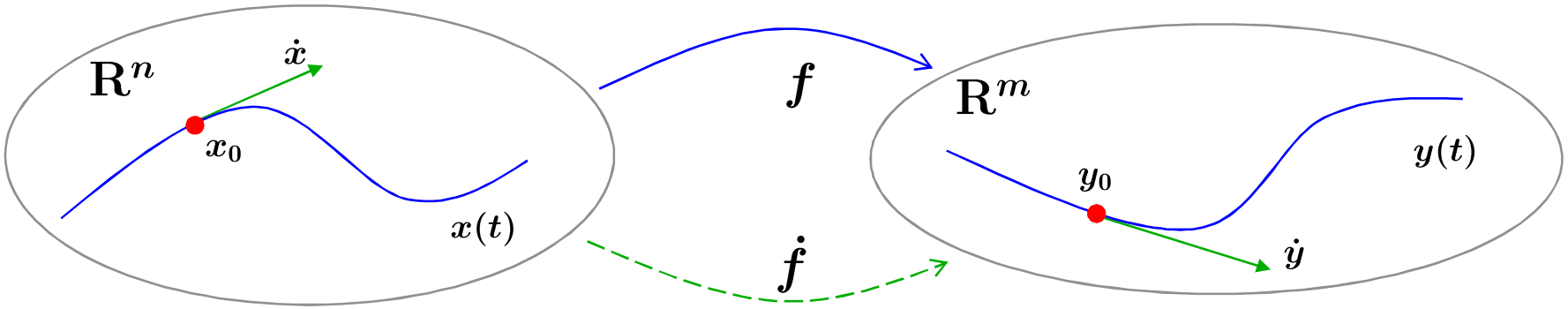
Sandia National Laboratories

Why is this important?

- 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
- 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

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

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, $\dot{x}_1$, $\dot{x}_2$:

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

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 $v_1, \ldots, v_p$ simultaneously

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

Jacobian matrix product

- Forward mode AD:

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

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

Other AD Modes

- Reverse mode (gradient propagation)

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

  - Gradients of scalar valued functions
  - Jacobian-transpose matrix-vector products
  - Computational cost (matrix W has q columns) $\approx (1.5 + 2.5q)\text{time}(f)$

- Taylor polynomial mode (univariate truncated Taylor series propagation)
  - Extension of tangent propagation to higher degree
  - Given $d+1$ coefficients $x_0, \ldots, x_d \in \mathrm{R}^n$

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

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

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

- Modes can be combined for various higher derivatives

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)

ADIFOR* Example

```
subroutine func(x, y)
C
      double precision x(2), y(2), 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
```

```
subroutine g_func(g_p_, x, g_x, ldg_x, y, g_y, ldg_y)
C
C       Initializations removed for clarity
C
      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--------
C
C       continues
C
```

*ADIFOR 2.0D

www-unix.mcs.anl.gov/autodiff/ADIFOR/

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;
}
```

```
template <typename T>
void func(const T x[], T y[]) {
  T 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 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;
}
```

```
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 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 sin(const Tangent& a) {
  Tangent c;
  c.val = sin(a.val);
  double t = cos(a.val);
  for (int i=0; i<Tangent::N; i++)
    c.dot[i] = t * a.dot[i];
  return c;
}
```

Sacado: AD Tools for C++ Codes

- Sacado provides several modes of Automatic Differentiation (AD)
  - Forward (Jacobians, Jacobian-vector products, …)
  - Reverse (Gradients, Jacobian-transpose-vector products, …)
  - Taylor (High-order univariate Taylor series)
- Sacado implements 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"
  - Very successful in Charon application code
  - `Sacado::`FEApp example demonstrates approach
- Sacado provides other useful utilities
  - Scalar flop counting (Ross Bartlett)
  - Scalar parameter library
  - Template utilities

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-examples, --enable-examples` — Enables examples
    - `nox/examples/epetra/LOCA_Sacado_FEApp` — Continuation example using `Sacado::FEApp` 1D finite element application
- 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 (not much there yet)
- Doxygen documentation (not all that useful)
- Examples are best way to learn how to use Sacado

Using Sacado

- As always: `#include "Sacado.hpp"`
- All relevant classes/functions 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:
  - `Sacado::ADvar<ScalarT>` (`Sacado_trad.h`)
- Taylor polynomial classes:
  - `Sacado::Taylor::DTaylor<ScalarT>`

How to use Sacado

- Template code to be differentiated: `double -> ScalarT`

```
class foo {
public:
  foo(double x) : x_(x) {}
  double bar(double y) { ... }
private:
  double x_;
};

double my_func(double a, double b) { ... }
```



```
template <typename ScalarT>
class foo {
public:
  foo(const ScalarT& x) : x_(x) {}
  ScalarT bar(const ScalarT& y) { ... }
private:
  ScalarT x_;
};

template <typename ScalarT>
ScalarT my_func(const ScalarT& a, const ScalarT& b) { ... }
```

- Replace independent/dependent variables with AD variables
- Initialize seed matrix
  - Derivative array of i'th independent variable is i'th row of seed matrix
- Evaluate function on AD variables
  - Instantiates template classes/functions
- Extract derivatives
  - Forward: Derivative components of dependent variables
  - Reverse: Derivative components of independent variables

sacado/example/dfad_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);
```

Differentiating Element-Based Codes

- Global residual computation (ignoring boundary computations):

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

- Jacobian computation:

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

- Jacobian-transpose product computation:

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

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

Sacado FEApp Example Application

- General 1D finite element application
  - Simple enough to be easily understood
  - Demonstrate complexity seen in real applications
- Currently implements two "physics"
  - Heat equation with nonlinear source

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + f(u), \quad f(u) = \alpha u^2 \text{ or } f(u) = \alpha u^3$$

$$u(0,t) = u_0, u(1,t) = u_1$$

  - Brusselator

$$\frac{\partial u}{\partial t} = D_1 \frac{\partial^2 u}{\partial x^2} + \alpha - (\beta + 1)u + vu^2$$

$$\frac{\partial v}{\partial t} = D_2 \frac{\partial^2 v}{\partial x^2} + \beta u - vu^2$$

$$u(0,t) = u(1,t) = \alpha$$

$$v(0,t) = v(1,t) = \beta/\alpha$$

- Source lives in Sacado
  - sacado/example/FEApp
- Drivers live in other package directories, e.g.,
  - nox/example/epetra/LOCA_Sacado_FEApp

FEApp::Application

```
namespace FEApp {
  class Application {
  public:

    //! Constructor
    Application(const std::vector<double>& coords, const Teuchos::RCP<const Epetra_Comm>& comm,
                const Teuchos::RCP<Teuchos::ParameterList>& params, bool is_transient);

    //! Compute global residual
    void computeGlobalResidual(const Epetra_Vector* xdot, const Epetra_Vector& x,
                               const Sacado::ScalarParameterVector* p, Epetra_Vector& f);

    //! Compute global Jacobian
    void computeGlobalJacobian(double alpha, double beta, const Epetra_Vector* xdot, const Epetra_Vector& x,
                               const Sacado::ScalarParameterVector* p, Epetra_Vector* f, Epetra_CrsMatrix& jac);

  protected:

    bool transient;                                 //! Is problem transient
    Teuchos::RCP<FEApp::AbstractDiscretization> disc;      //! Element discretization
    std::vector< Teuchos::RCP<FEApp::NodeBC> > bc;         //! Boundary conditions
    Teuchos::RCP<const FEApp::AbstractQuadrature> quad;    //! Quadrature rule
    FEApp::AbstractPDE_TemplateManager<ValidTypes> pdeTM;  //! PDE equations
    Teuchos::RCP<Epetra_Vector> initial_x;                 //! Initial solution vector
    Teuchos::RCP<Epetra_Import> importer;                  //! Importer for overlapped data
    Teuchos::RCP<Epetra_Export> exporter;                  //! Exporter for overlapped data
    Teuchos::RCP<Epetra_Vector> overlapped_x;              //! Overlapped solution vecto
    Teuchos::RCP<Epetra_Vector> overlapped_xdot;           //! Overlapped time derivative vecto
    Teuchos::RCP<Epetra_Vector> overlapped_f;              //! Overlapped residual vector
    Teuchos::RCP<Epetra_CrsMatrix> overlapped_jac;         //! Overlapped Jacobian matrix
    Teuchos::RCP<Sacado::ScalarParameterLibrary> paramLib; //! Parameter library

  };
}
```

FEApp::Application::computeGlobalResidual

```
void FEApp::Application::computeGlobalResidual(const Epetra_Vector* xdot, const Epetra_Vector& x,
                                               const Sacado::ScalarParameterVector* p, Epetra_Vector& f) {
  // Scatter x, xdot to the overlapped distrbution
  overlapped_x->Import(x, *importer, Insert);
  if (transient) overlapped_xdot->Import(*xdot, *importer, Insert);

  // Set parameters
  if (p != NULL)
    for (unsigned int i=0; i<p->size(); ++i)
      (*p)[i].family->setRealValueForAllTypes((*p)[i].baseValue);

  // Zero out overlapped residual
  overlapped_f->PutScalar(0.0);

  // Create residual init/post op
  Teuchos::RCP<FEApp::ResidualOp> op =
    Teuchos::rcp(new FEApp::ResidualOp(overlapped_xdot, overlapped_x, overlapped_f));

  // Get template PDE instantiation
  Teuchos::RCP< FEApp::AbstractPDE<ResidualOp::fill_type> > pde = pdeTM.getAsObject<ResidualOp::fill_type>();

  // Do global fill
  FEApp::GlobalFill<ResidualOp::fill_type> globalFill(disc->getMesh(), quad, pde, bc, transient);
  globalFill.computeGlobalFill(*op);

  // Assemble global residual
  f.Export(*overlapped_f, *exporter, Add);
}
```

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

Sandia National Laboratories

FEApp::Application::computeGlobalJacobian

```
void FEApp::Application::computeGlobalJacobian(double alpha, double beta, const Epetra_Vector* xdot,
                                               const Epetra_Vector& x, const Sacado::ScalarParameterVector* p,
                                               Epetra_Vector* f, Epetra_CrsMatrix& jac) {
  // Scatter x, xdot to the overlapped distrbution
  overlapped_x->Import(x, *importer, Insert);
  if (transient) overlapped_xdot->Import(*xdot, *importer, Insert);

  // Set parameters
  if (p != NULL)
    for (unsigned int i=0; i<p->size(); ++i)
      (*p)[i].family->setRealValueForAllTypes((*p)[i].baseValue);

  // Zero out overlapped residual, Jacobian
  Teuchos::RCP<Epetra_Vector> overlapped_ff;
  if (f != NULL) { overlapped_ff = overlapped_f; overlapped_ff->PutScalar(0.0); }
  overlapped_jac->PutScalar(0.0);

  // Create Jacobian init/post op
  Teuchos::RCP<FEApp::JacobianOp> op =
    Teuchos::rcp(new FEApp::JacobianOp(alpha, beta, overlapped_xdot, overlapped_x, overlapped_ff,
                                       overlapped_jac));

  // Get template PDE instantiation
  Teuchos::RCP< FEApp::AbstractPDE<JacobianOp::fill_type> > pde = pdeTM.getAsObject<JacobianOp::fill_type>();

  // Do global fill
  FEApp::GlobalFill<JacobianOp::fill_type> globalFill(disc->getMesh(), quad, pde, bc, transient);
  globalFill.computeGlobalFill(*op);

  // Assemble global residual, Jacobian
  if (f != NULL) f->Export(*overlapped_f, *exporter, Add);
  jac.Export(*overlapped_jac, *exporter, Add);
  jac.FillComplete(true);
}
```

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

FEApp::GlobalFill

```
namespace FEApp {
  template <typename ScalarT>
  class GlobalFill {
  public:

    //! Constructor
    GlobalFill(const Teuchos::RCP<const FEApp::Mesh>& elementMesh,
               const Teuchos::RCP<const FEApp::AbstractQuadrature>& quadRule,
               const Teuchos::RCP< FEApp::AbstractPDE<ScalarT> >& pdeEquations,
               const std::vector< Teuchos::RCP<FEApp::NodeBC> >& nodeBCs,
               bool is_transient);

    //! Compute global fill
    void computeGlobalFill(FEApp::AbstractInitPostOp<ScalarT>& initPostOp);

  protected:

    Teuchos::RCP<const FEApp::Mesh> mesh;              //! Element mesh
    Teuchos::RCP<const FEApp::AbstractQuadrature> quad; //! Quadrature rule
    Teuchos::RCP< FEApp::AbstractPDE<ScalarT> > pde;    //! PDE Equations
    std::vector< Teuchos::RCP<FEApp::NodeBC> > bc;      //! Node boundary conditions
    bool transient;                          //! Are we transient?
    unsigned int nnode;                                 //! Number of nodes per element
    unsigned int neqn;                                                //! Number of PDE equations
    unsigned int ndof;                                                //! Number of element-level DOF

    std::vector<ScalarT> elem_x;                        //! Element solution variables
    std::vector<ScalarT>* elem_xdot;                    //! Element time derivative variables
    std::vector<ScalarT> elem_f;                        //! Element residual variables
    std::vector<ScalarT> node_x;                        //! Node solution variables
    std::vector<ScalarT>* node_xdot;                    //! Node time derivative variables
    std::vector<ScalarT> node_f;                        //! Node residual variables
  };
}
```

FEApp::GlobalFill::computeGlobalFill

```
template <typename ScalarT>
void FEApp::GlobalFill<ScalarT>::computeGlobalFill(FEApp::AbstractInitPostOp<ScalarT>& initPostOp)
{
  // Loop over elements
  Teuchos::RCP<const FEApp::AbstractElement> e;
  for (FEApp::Mesh::const_iterator eit=mesh->begin(); eit!=mesh->end(); ++eit) {
    e = *eit;

    // Zero out element residual
    for (unsigned int i=0; i<ndof; i++)
      elem_f[i] = 0.0;

    initPostOp.elementInit(*e, neqn, elem_xdot, elem_x); // Initialize element solution

    pde->evaluateElementResidual(*quad, *e, elem_xdot, elem_x, elem_f); // Compute element residual

    initPostOp.elementPost(*e, neqn, elem_f); // Post-process element residual
  }

  // Loop over boundary conditions
  for (std::size_t i=0; i<bc.size(); i++) {
    if (bc[i]->isOwned() || bc[i]->isShared()) {
      // Zero out node residual
      for (unsigned int j=0; j<neqn; j++)
        node_f[j] = 0.0;

      initPostOp.nodeInit(*bc[i], neqn, node_xdot, node_x); // Initialize node solution

      bc[i]->getStrategy<ScalarT>()->evaluateResidual(node_xdot, node_x, node_f); // Compute node residual

      initPostOp.nodePost(*bc[i], neqn, node_f); // Post-process node residual
    }
  }
}
```

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

FEApp::JacobianOp

```
namespace FEApp {
  class JacobianOp : public FEApp::AbstractInitPostOp< Sacado::Fad::DFad<double> > {
  public:
    //! Constructor
    JacobianOp(double alpha, double beta, const Teuchos::RCP<const Epetra_Vector>& overlapped_xdot,
                          const Teuchos::RCP<const Epetra_Vector>& overlapped_x,
                          const Teuchos::RCP<Epetra_Vector>& overlapped_f,
                          const Teuchos::RCP<Epetra_CrsMatrix>& overlapped_jac);

    //! Evaluate element init operator
    virtual void elementInit(const FEApp::AbstractElement& e, unsigned int neqn,
                                        std::vector< Sacado::Fad::DFad<double> >* elem_xdot,
                                        std::vector< Sacado::Fad::DFad<double> >& elem_x);

    //! Evaluate element post operator
    virtual void elementPost(const FEApp::AbstractElement& e, unsigned int neqn,
                                        std::vector< Sacado::Fad::DFad<double> >& elem_f);

    //! Evaulate node init operator
    virtual void nodeInit(const FEApp::NodeBC& bc, unsigned int neqn,
                                      std::vector< Sacado::Fad::DFad<double> >* node_xdot,
                                      std::vector< Sacado::Fad::DFad<double> >& node_x);

    //! Evaluate node post operator
    virtual void nodePost(const FEApp::NodeBC& bc, unsigned int neqn,
                                      std::vector< Sacado::Fad::DFad<double> >& node_f);
  protected:
    double m_coeff;                    //! Coefficient of mass matrix
    double j_coeff;                   //! Coefficient of Jacobian matrix
    Teuchos::RCP<const Epetra_Vector> xdot; //! Time derivative vector (may be null)
    Teuchos::RCP<const Epetra_Vector> x;    //! Solution vector
    Teuchos::RCP<Epetra_Vector> f;          //! Residual vector
    Teuchos::RCP<Epetra_CrsMatrix> jac;     //! Jacobian matrix
  };
}
```

FEApp::JacobianOp::elementInit

```
void FEApp::JacobianOp::elementInit(const FEApp::AbstractElement& e, unsigned int neqn,
                                    std::vector< Sacado::Fad::DFad<double> >* elem_xdot,
                                    std::vector< Sacado::Fad::DFad<double> >& elem_x) {

  unsigned int node_GID;          // Global node ID
  unsigned int firstDOF;          // Local ID of first DOF
  unsigned int nnode = e.numNodes(); // Number of nodes
  unsigned int ndof = nnode*neqn;    // Number of dof

  // Copy element solution
  for (unsigned int i=0; i<nnode; i++) {

    node_GID = e.nodeGID(i);
    firstDOF = x->Map().LID(node_GID*neqn);

    for (unsigned int j=0; j<neqn; j++) {

      elem_x[neqn*i+j] = Sacado::Fad::DFad<double>(ndof, (*x)[firstDOF+j]);
      elem_x[neqn*i+j].fastAccessDx(neqn*i+j) = j_coeff;

      if (elem_xdot != NULL) {
        (*elem_xdot)[neqn*i+j] = Sacado::Fad::DFad<double>(ndof, (*xdot)[firstDOF+j]);
        (*elem_xdot)[neqn*i+j].fastAccessDx(neqn*i+j) = m_coeff;
      }

    }

  }

}
```

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

Sandia National Laboratories

FEApp::HeatNonlinearSourcePDE

```
namespace FEApp {
  template <typename ScalarT>
  class HeatNonlinearSourcePDE : public FEApp::AbstractPDE<ScalarT> {
  public:

    //! Constructor
    HeatNonlinearSourcePDE(const Teuchos::RCP< const FEApp::AbstractSourceFunction<ScalarT> >& src_func);

    //! Initialize PDE
    virtual void init(unsigned int numQuadPoints, unsigned int numNodes);

    //! Evaluate discretized PDE element-level residual
    virtual void
    evaluateElementResidual(const FEApp::AbstractQuadrature& quadRule,
                            const FEApp::AbstractElement& element,
                            const std::vector<ScalarT>* dot,
                            const std::vector<ScalarT>& solution,
                            std::vector<ScalarT>& residual);

  protected:

    Teuchos::RCP< const FEApp::AbstractSourceFunction<ScalarT> > source; //! Source function
    unsigned int num_qp;                                                 //! Number of quad points
    unsigned int num_nodes;                                              //! Number of nodes
    std::vector< std::vector<double> > phi;                              //! Shape function values
    std::vector< std::vector<double> > dphi;                             //! Shape function derivative
    std::vector<double> jac;                                             //! Element transformation Jacobian
    std::vector<ScalarT> u;                                              //! Discretized solution
    std::vector<ScalarT> du;                                             //! Discretized solution derivative
    std::vector<ScalarT> udot;                                           //! Discretized time derivative
    std::vector<ScalarT> f;                                              //! Source function values
  };
}
```

FEApp::HeatNonlinearSourcePDE:: evaluateElementResidual

```
template <typename ScalarT> void FEApp::HeatNonlinearSourcePDE<ScalarT>::
evaluateElementResidual(const FEApp::AbstractQuadrature& quadRule, const FEApp::AbstractElement& element,
                        const std::vector<ScalarT>* dot, const std::vector<ScalarT>& solution,
                        std::vector<ScalarT>& residual) {

  const std::vector<double>& xi = quadRule.quadPoints();          // Quadrature points
  const std::vector<double>& w = quadRule.weights();              // Weights

  element.evaluateShapes(xi, phi);                                // Evaluate shape function
  element.evaluateShapeDerivs(xi, dphi);                          // Evaluate shape function derivative
  element.evaluateJacobian(xi, jac);                              // Evaluate element Jacobian

  // Compute u, du, udot
  for (unsigned int qp=0; qp<num_qp; qp++) {
    u[qp] = 0.0; du[qp] = 0.0; udot[qp] = 0.0;
    for (unsigned int node=0; node<num_nodes; node++) {
      u[qp] += solution[node] * phi[qp][node];
      du[qp] += solution[node] * dphi[qp][node];
      if (dot != NULL) udot[qp] += (*dot)[node] * phi[qp][node];
    }
  }

  source->evaluate(u, f); // Evaluate source function

  // Evaluate residual
  for (unsigned int node=0; node<num_nodes; node++) {
    residual[node] = 0.0;
    for (unsigned int qp=0; qp<num_qp; qp++) {
      residual[node] += w[qp]*jac[qp]*(-(1.0/(jac[qp]*jac[qp]))*du[qp]*dphi[qp][node] +
                                       phi[qp][node]*(f[qp] - udot[qp]));
    }
  }
}
```

$$u_e = \sum_{j=1}^{n} a_j^e(t)\phi_j^e(x)$$

$$\frac{\partial u_e}{\partial x} = \sum_{j=1}^{n} a_j^e(t)\frac{d\phi_j^e}{dx}$$

$$\frac{\partial u_e}{\partial t} = \sum_{j=1}^{n} \frac{da_j^e}{dt}\phi_j^e(x)$$

$$F_i^e = \int_e \left(-\frac{\partial u^e}{\partial x}\frac{d\phi_i^e}{dx} + f(u^e) - \frac{\partial u^e}{\partial t} \right) dx$$

FEApp::CubicSourceFunction

```
namespace FEApp {
  template <typename ScalarT>
  class CubicSourceFunction : public FEApp::AbstractSourceFunction<ScalarT> {
  public:
    //! Constructor
    CubicSourceFunction(const ScalarT& factor, const Teuchos::RCP<Sacado::ScalarParameterLibrary>& paramLib) :
      alpha(factor)
    {
      // Add nonlinear factor to parameter library
      std::string name = "Cubic Source Function Nonlinear Factor";
      if (!paramLib->isParameter(name))
        paramLib->addParameterFamily(name, true, false);
      if (!paramLib->template isParameterForType<ScalarT>(name)) {
        Teuchos::RCP< CubicNonlinearFactorParameter<ScalarT> > tmp =
          Teuchos::rcp(new CubicNonlinearFactorParameter<ScalarT>(Teuchos::rcp(this,false)));
        paramLib->template addEntry<ScalarT>(name, tmp);
      }
    }

    //! Evaluate source function
    virtual void evaluate(const std::vector<ScalarT>& solution, std::vector<ScalarT>& value) const {
      for (unsigned int i=0; i<solution.size(); i++)
        value[i] = alpha*solution[i]*solution[i]*solution[i];
    }

    //! Set nonlinear factor
    void setFactor(const ScalarT& val, bool mark_constant) { alpha = val; }

    //! Get nonlinear factor
    const ScalarT& getFactor() const { return alpha; }
  protected:
    ScalarT alpha; //! Factor
  };
}
```

$$f(u^e) = \alpha u_e^3$$

FEApp::JacobianOp::elementPost

```
void FEApp::JacobianOp::elementPost(const FEApp::AbstractElement& e, unsigned int neqn,
                                    std::vector< Sacado::Fad::DFad<double> >& elem_f) {
  unsigned int nnode = e.numNodes(); // Number of nodes

  // Loop over nodes in element
  for (unsigned int node_row=0; node_row<nnode; node_row++) {

    // Loop over equations per node
    for (unsigned int eq_row=0; eq_row<neqn; eq_row++) {
      unsigned int lrow = neqn*node_row+eq_row // Local row
      int row = static_cast<int>(e.nodeGID(node_row)*neqn + eq_row); // Global row

      if (f != Teuchos::null) f->SumIntoGlobalValue(row, 0, elem_f[lrow].val()); // Sum residual

      // Check derivative array is nonzero
      if (elem_f[lrow].hasFastAccess()) {

        // Loop over nodes in element
        for (unsigned int node_col=0; node_col<nnode; node_col++){

          // Loop over equations per node
          for (unsigned int eq_col=0; eq_col<neqn; eq_col++) {
                    unsigned int lcol = neqn*node_col+eq_col; // Local column
                    int col = static_cast<int>(e.nodeGID(node_col)*neqn + eq_col); // Global column

                    jac->SumIntoGlobalValues(row, 1, &(elem_f[lrow].fastAccessDx(lcol)), &col); // Sum Jacobian

          } // column equation
        } // column node
      } // has fast access
    } // row equation
  } // row node
}
```

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

FEApp::CubicNonlinearFactorParameter

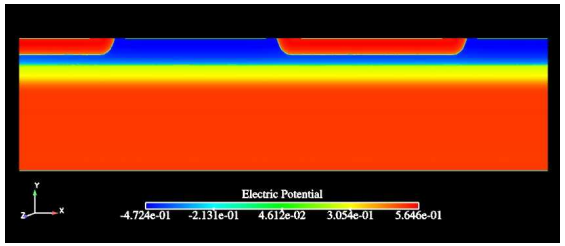
```
namespace FEApp {
  template <typename ScalarT>
  class CubicNonlinearFactorParameter : public Sacado::ScalarParameterEntry<ScalarT> {
  public:

    //! Constructor
    CubicNonlinearFactorParameter(const Teuchos::RCP< CubicSourceFunction<ScalarT> >& s) : srcFunc(s) {}

    //! Destructor
    virtual ~CubicNonlinearFactorParameter() {}

    //! Set real parameter value
    virtual void setRealValue(double value) { setValueAsConstant(ScalarT(value)); }

    //! Set parameter this object represents to \em value
    virtual void setValueAsConstant(const ScalarT& value) { srcFunc->setFactor(value, true); }

    //! Set parameter this object represents to \em value
    virtual void setValueAsIndependent(const ScalarT& value) { srcFunc->setFactor(value, false); }

    //! Get parameter value this object represents
    virtual const ScalarT& getValue() const { return srcFunc->getFactor(); }

  protected:

    Teuchos::RCP< CubicSourceFunction<ScalarT> > srcFunc; //! Pointer to source function

  };
}
```

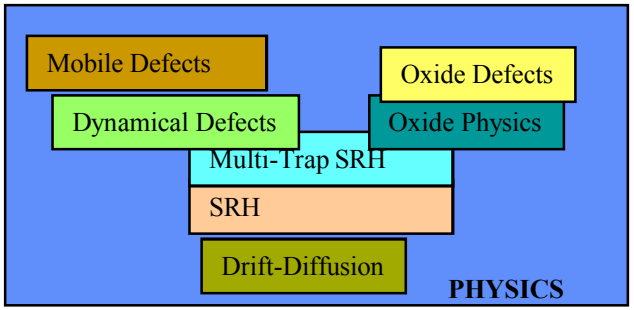
Sandia National Laboratories

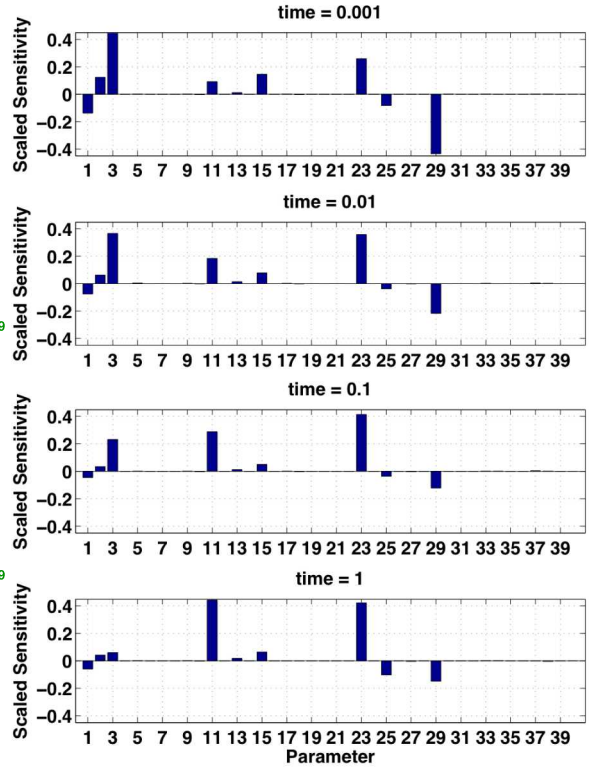
Derivative Calculations

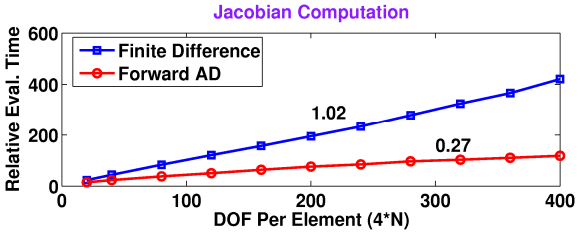
- This approach makes it easy to add new derivative calculations
  - Most of the work is creating new init/post process operators
- Sacado::FEApp has 3:
  - Residual, Jacobian, parameter derivatives
- Charon has 10:
  - Residual, Jacobian (Fad, FD), Adjoint (Rad, Fad, FD), scalar parameter derivs, distributed parameter derivs, 2 types of second derivatives
- Template manager/iterator help insulated code from number of AD types

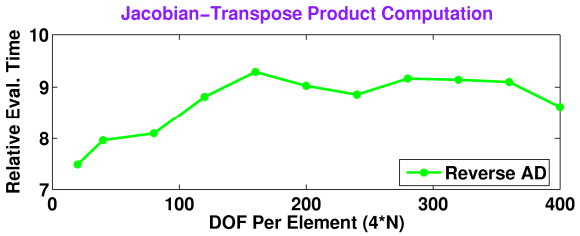
Impacts of AD in Charon
(~114k lines of code, significant portion templated)

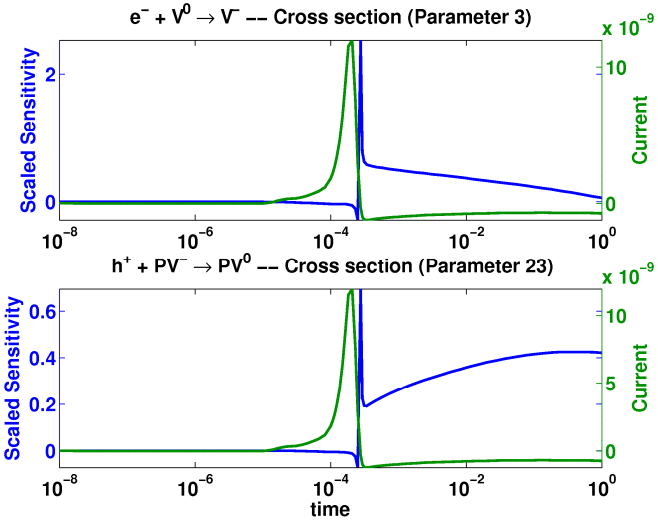


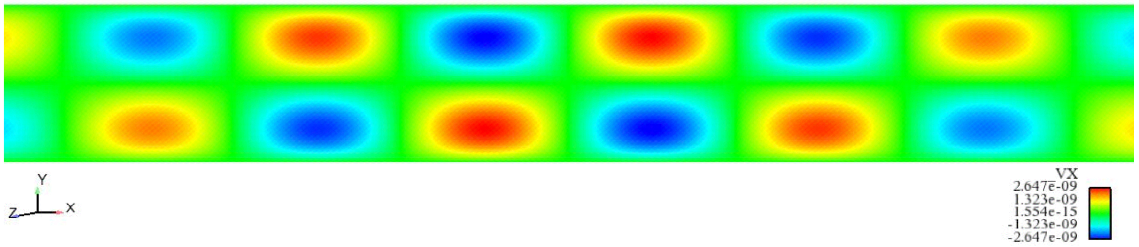


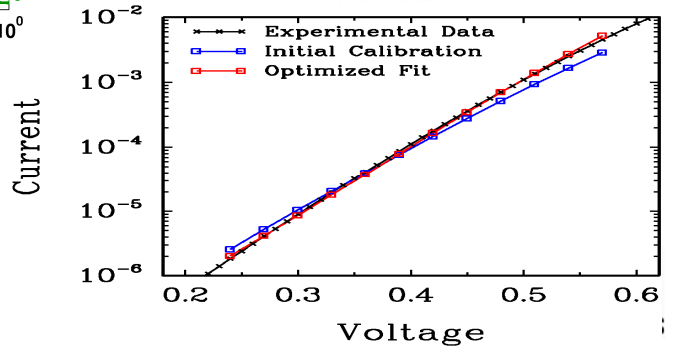












Complications

- Excessive compile times due to application templating
  - Application source files move to headers
  - Small changes cause long compile times
  - Explicit template instantiation (demonstrated in FEApp)
- Interfacing template and non-template code
  - Many places where non-template code must call template code
  - Difficult to add new AD types
  - Sacado template manager/iterator (demonstrated in FEApp)
- Parameter derivatives
  - Application codes don't provide a parameter interface
  - Sacado parameter library (demonstrated in FEApp)
- Interfaces to other derivative methods (e.g., source transformation)
  - Used in Charon (ADIFOR differentiated CHEMKIN)
  - Example coming soon for BLAS/LAPACK

Explicit Template Instantiation

FEApp_TemplateTypes.hpp

```
// Include all of our AD types
#include "Sacado_Fad_DFad.hpp"

// Typedef AD types to standard names
typedef double RealType;
typedef Sacado::Fad::DFad<double> FadType;

// Define which types we are using
#define REAL_ACTIVE 1
#define FAD_ACTIVE 1

// Define macro for explicit template instantiation
#if REAL_ACTIVE
#define INSTANTIATE_TEMPLATE_CLASS_REAL(name) template class name<double>;
#else
#define INSTANTIATE_TEMPLATE_CLASS_REAL(name)
#endif

#if FAD_ACTIVE
#define INSTANTIATE_TEMPLATE_CLASS_FAD(name) template class name<FadType>;
#else
#define INSTANTIATE_TEMPLATE_CLASS_FAD(name)
#endif

#define INSTANTIATE_TEMPLATE_CLASS(name) \
  INSTANTIATE_TEMPLATE_CLASS_REAL(name)  \
  INSTANTIATE_TEMPLATE_CLASS_FAD(name)
```

FEApp_GlobalFill.hpp

```
#include "FEApp_TemplateTypes.hpp"

namespace FEApp {

  template <typename ScalarT>
  class GlobalFill {
  public:

    // . . .

  };

}

// Include implementation
#ifndef SACADO_ETI
#include "FEApp_GlobalFillImpl.hpp"
#endif
```

FEApp_GlobalFill.cpp

```
#include "FEApp_TemplateTypes.hpp"

#ifdef SACADO_ETI

#include "FEApp_GlobalFill.hpp"
#include "FEApp_GlobalFillImpl.hpp"

INSTANTIATE_TEMPLATE_CLASS(FEApp::GlobalFill)

#endif
```

More Complications

- Branching/conditionals
  - For derivative, branch chosen based on value of argument
  - Piecewise derivative
  - Always obtain correct derivative for branch that was evaluated

- Removing portions of computation for derivative calculation

```
inline double ADValue(double x) { return x; }
inline double ADValue(const Sacado::Fad::DFad<double>& x) { return x.val(); }

double my_func(double a, double b) {
  // ...
}

  ScalarT a = ...
  ScalarT b = ...

  double c = my_func(ADValue(a), ADValue(b)); // This will not be differentiated

  ScalarT d = ...
```

More Complications

- Points of non-differentiability
  - Usually signaled by NaN's in derivative
  - First remove unnecessary points of non-differentiability:

```
template <typename ScalarT>
ScalarT vec_norm(ScalarT x[]) {
  return std::sqrt(x[0]*x[0] + x[1]*x[1] + x[2]*x[2]);
}

  ScalarT x[3];

  // ...

  ScalarT norm_x = vec_norm(x);

  ScalarT a = norm_x*norm_x;   // Problem when x = 0
```

  - Then use conditionals if necesssary:

```
 ScalarT a = ...

  ScalarT b;
 if (a = 0.0)
   b = 0.0;
 else
   b = std::sqrt(a);
```

  - We can try to improve support for this in Sacado

Implementing Operator Overloading Efficiently: Expression Templates

- Naïve operator overloading:

```
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 sin(const Tangent& a) {
  Tangent c;
  c.val = sin(a.val);
  double t = cos(a.val);
  for (int i=0; i<Tangent::N; i++)
    c.dot[i] = t * a.dot[i];
  return c;
}
```

  - Each operation returns a copy (bad)
  - Each operation implements a loop (bad)

- Template meta-programming (Abrahams & Gurtovoy, 2005)
  - View templates as a compile-time functional language operating on types and integral values (bool's, int's, etc…)
  - Turing complete (any computation can be implemented)
  - Computations occur at compile time: No run-time cost

- AD via expression templates:
  - Each expression represents a new type with a new derivative rule built at compile time
  - Derivative of expression computed at assignment (at = sign)

Expression Template Operator Overloading

```
void func(const Tangent x[], Tangent y[]) {
  y[0] = sin(exp(x[0]) + x[0]*x[1]);
  //…
}
```

```
Expr< SinExpr< Expr< PlusExpr<
 Expr< ExpExpr<TangentExpr> >,
 Expr< MultExpr< Expr<TangentExpr>,
                 Expr<TangentExpr> > >
               > > > >
```

```
y[0].val = sin(exp(x[0]) + x[0]*x[1]);
for (int i=0; i<N; i++) {
  y[0].dot[i] = cos(exp(x[0]) + x[0]*x[1])*
     (exp(x[0])*x[0].dot[i] +
      x[0]*x[1].dot[i] + x[1]*x[0].dot[i]);
```

Sacado forward AD classes, based on public domain Fad/TFad package

```
template <class E1> Expr {};

template <class E1, E2> class PlusExpr {};
template <class E1, E2> class Expr< PlusExpr<E1,E2> > {
  double val() const { return e1.val() + e2.val(); }
  double dx(int i) const { return e1.dx(i) + e2.dx(i); }
  const Expr<E1>& e1;
  const Expr<E2>& e2;
};

template<class E1, class E2> Expr< PlusExpr<E1,E2> >
operator+(const Expr<E1>& a, const Expr<E2>& b) {
  return Expr< PlusExpr<E1,E2> >(a,b);
}

template <class E1> class SinExpr {};
template <class E1> class Expr< SinExpr<E1> > {
  double val() const { return sin(e1.val())] }
  double dx(int i) const { return cos(e1.val())*e1.dx(i); }
  const Expr<E1>& e1;
};

template<class E1> Expr< SinExpr<E1> > sin(const Expr<E1>& a) {
  return Expr< SinExpr<E1> >(a);
}

class TangentExpr {};
class Expr<TangentExpr> : {
public:
  double val() const { return val; }
  double dx(int i) const { return dot[i]; }
  template <class E> Expr<Tangent>& operator=(const Expr<E>& e) {
    val = e.val();
    for (int i=0; i<N; i++)
      dot[i] = e.dx(i);
  }
};
class Tangent : public Expr<TangentExpr> {};
```

Performance

- Implementing this effectively requires a good optimizing compiler

fad_expr.exe: 10 derivative components through a simple expression						
	GCC 4.1.2 -O3		Intel 10.0 -O3		PGI 6.2-5 -O3 -fastsse	
	Time (s)	Slow down	Time (s)	Slow down	Time (s)	Slow down
Analytic	1.30e-07	1.00	5.00e-12	1.00	1.02e-07	1.00
SFad	9.50e-07	7.32	1.28e-07	2.57e+04	2.11e-06	20.7
DFad	8.38e-07	6.46	1.57e-07	3.15e+04	2.20e-06	21.6
ELR SFad	1.79e-07	1.38	1.09e-07	2.18e+04	2.38e-06	23.3
ELR DFad	1.96e-07	1.51	1.70e-07	3.40e+04	2.61e-06	25.5
fad_lj_grad.exe: Gradient of Leonard-Jones potential						
Analytic	6.83e-09	1.00	1.20e-08	1.00	4.68e-08	1.00
SFad	6.86e-08	10.1	4.67e-08	3.88	1.73e-06	36.9
DFad	4.99e-07	73.1	3.86e-07	32.1	3.07e-06	65.7
ELR SFad	5.20e-08	7.62	9.48e-08	7.88	3.46e-06	74.0
ELR DFad	4.80e-07	70.3	5.01e-07	41.6	4.87e-06	104.0

- Tests live in sacado/test/performance
- Not completely indicative of “real-world” performance

Complications Introduced by Expression Templates

- Template functions
  - An expression can always be converted to a Fad type, but
  - Compilers implement very few automatic conversions for template function arguments

```
template <typename ScalarT>
class MyClass {
public:
  // ...

  ScalarT my_func(const ScalarT& a, const ScalarT& b) {
    // ...
  }
};

template <typename ScalarT>
ScalarT my_func(const ScalarT& a, const ScalarT& b) {
  // ...
}

  ScalarT a = ...                  // Initialize a
  ScalarT b = ...                  // Initialize b

  MyClass<ScalarT> my_class;
  ScalarT c = my_class.my_func(a+b,a); // Will work just fine

  ScalarT d = my_func(a+b,a);          // Won't compile
  ScalarT e = my_func<ScalarT>(a+b,a); // Will work
  ScalarT f = my_func(ScalarT(a+b),a); // Will work
```

- Understanding compiler errors like these can be difficult

How Sacado relates to other packages

- Many Trilinos packages need derivatives
  - NOX (nonlinear solves)
  - LOCA (stability analysis)
  - Rythmos (time integration)
  - MOOCHO, Aristos (optimization)
- Sacado does not provide these derivatives directly
  - Sacado is not a black-box AD solution
- Sacado provides low level AD capabilities
  - Application codes use Sacado to build derivatives these packages need

Best Practices

- Don't differentiate your global function with AD
- Only use AD for the hard, nonlinear parts
- Never differentiate iterative solvers with AD…instead use AD for the derivative of the solution

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

- Prefer template classes over template functions
  - Methods of a template class are not template functions
  - Compiler implements very few conversions for template functions

Where Sacado is going in the future

- Documentation
  - Website, tutorials, papers, etc…
- Performance improvements
  - Expression level reverse-mode (`Sacado::ELRFad`)
- Leveraging AD technology for intrusive uncertainty quantification
  - Polynomial chaos expansions via operator overloading
- Impacting more applications
  - Using Sacado is more about software engineering than AD