Embedded Algorithms through Template-based Generic Programming

Eric Phipps (etphipp@sandia.gov), Roger Pawlowski, Andy Salinger, Sandia National Laboratories

2011 Trilinos User Group Meeting
Nov. 1-3, 2011

SAND 2011-8396C
Outline

• Embedded algorithms
• Template-based generic programming
• Incorporating approach into complex codes
• Computational demonstrations
• Ongoing and future work
What does *embedded* mean?

• We used to call this *intrusive*

• Generally anything that requires more of a simulation code than just running it
  – i.e., not black-box or non-intrusive

• Why do this?
  – By asking for more, improvements can be made
    • Increased efficiency, scalability, robustness
    • Greater understanding through deeper analysis
Examples of embedded algorithms

- Model problem
  \[ f(\dot{x}, x, p) = 0, \quad \dot{x}, x \in \mathbb{R}^n, \quad p \in \mathbb{R}^m, \quad f : \mathbb{R}^{2n+m} \rightarrow \mathbb{R}^n \]

- Direct to steady-state, implicit time-stepping, linear stability analysis
  \[ \left( \alpha \frac{\partial f}{\partial \dot{x}} + \beta \frac{\partial f}{\partial x} \right) \Delta x = -f \]

- Steady-state parameter continuation
  \[ f(x^{(n)}, p^{(n)}) = 0 \]
  \[ g(x^{(n)}, p^{(n)}) = v_x^T(x^{(n)} - x^{(n-1)}) + v_p^T(p^{(n)} - p^{(n-1)}) - \Delta s_n = 0 \]
  \[ \begin{bmatrix} \frac{\partial f}{\partial x} \\ v_x^T \\ \frac{\partial f}{\partial p} \\ v_p^T \end{bmatrix} \begin{bmatrix} \Delta x^{(n)} \\ \Delta p^{(n)} \end{bmatrix} = - \begin{bmatrix} f \\ g \end{bmatrix} \]

- Bifurcation analysis
  \[ f(x, p) = 0, \quad \sigma = -u^T Jv, \quad \frac{\partial \sigma}{\partial x} = -u^T \frac{\partial}{\partial x} (Jv), \quad \frac{\partial \sigma}{\partial p} = -u^T \frac{\partial}{\partial p} (Jv), \]
  \[ \begin{bmatrix} J & a \\ b^T & 0 \end{bmatrix} \begin{bmatrix} v \\ s_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} J^T & b \\ a^T & 0 \end{bmatrix} \begin{bmatrix} u \\ s_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \]
Examples of embedded algorithms

- Steady-state sensitivity analysis

\[ f(x^*, p) = 0, \quad s^* = g(x^*, p) \quad \Rightarrow \quad \frac{ds^*}{dp} = -\frac{\partial g}{\partial x}(x^*, p) \left( \frac{\partial f}{\partial x}(x^*, p) \right)^{-1} \frac{\partial f}{\partial p}(x^*, p) + \frac{\partial g}{\partial p}(x^*, p) \]

- Transient sensitivity analysis

\[ f(\dot{x}, x, p) = 0, \quad \frac{\partial f}{\partial \dot{x}} \frac{\partial \dot{x}}{\partial p} + \frac{\partial f}{\partial x} \frac{\partial x}{\partial p} + \frac{\partial f}{\partial p} = 0 \]
Stochastic Galerkin UQ Methods

- **Steady-state stochastic problem (for simplicity):**
  \[
  \text{Find } u(\xi) \text{ such that } f(u, \xi) = 0, \quad \xi : \Omega \to \Gamma \subset \mathbb{R}^M, \text{ density } \rho
  \]

- **Stochastic Galerkin method (Ghanem and many, many others...):**
  \[
  \hat{u}(\xi) = \sum_{i=0}^{P} u_i \psi_i(\xi) \rightarrow F_i(u_0, \ldots, u_P) = \frac{1}{\langle \psi_i^2 \rangle} \int_{\Gamma} f(\hat{u}(y), y) \psi_i(y) \rho(y) dy = 0, \quad i = 0, \ldots, P
  \]

- **Method generates new coupled spatial-stochastic nonlinear problem (intrusive)**
  \[
  0 = F(U) = \begin{bmatrix} F_0 \\ F_1 \\ \vdots \\ F_P \end{bmatrix}, \quad U = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_P \end{bmatrix} \cdot \frac{\partial F}{\partial U}
  \]

- **Advantages:**
  - Many fewer stochastic degrees-of-freedom for comparable level of accuracy

- **Challenges:**
  - Computing SG residual and Jacobian entries in large-scale, production simulation codes
  - Solving resulting systems of equations efficiently

Stochastic sparsity
Spatial sparsity
Challenges of embedded algorithms

• Many kinds of quantities required
  – State and parameter derivatives
  – Various forms of second derivatives
  – Polynomial chaos expansions
  – ...

• Incorporating these directly requires significant effort
  – Time consuming, error prone
  – Gets in the way of physics/model development

• Requires code developers to understand requirements of algorithmic approaches
  – Limits embedded algorithm R&D on complex problems
A solution

• Need a framework that
  – Allows simulation code developers to focus on complex physics development
  – Doesn’t make them worry about advanced analysis
  – Allows derivatives and other quantities to be easily extracted
  – Is extensible to future embedded algorithm requirements

• Template-based generic programming
  – Code developers write physics code templated on scalar type
  – Operator overloading libraries provide tools to propagate needed embedded quantities
  – Libraries connect these quantities to embedded solver/analysis tools

• Foundation for this approach lies with Automatic Differentiation (AD)
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>x</th>
<th>$\frac{d}{d.x}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.000</td>
<td>1.000</td>
</tr>
<tr>
<td>7.389</td>
<td>7.389</td>
</tr>
<tr>
<td>0.301</td>
<td>0.500</td>
</tr>
<tr>
<td>0.602</td>
<td>1.301</td>
</tr>
<tr>
<td>7.991</td>
<td>8.690</td>
</tr>
<tr>
<td>0.991</td>
<td>-1.188</td>
</tr>
</tbody>
</table>
Sacado: AD Tools for C++ Codes

- Several modes of Automatic Differentiation (AD)
  - Forward (Jacobians, Jacobian-vector products, ...)
  - Reverse (Gradients, Jacobian-transpose-vector products, ...)
  - Taylor (High-order univariate Taylor series)
  - Modes can be nested for various forms of higher derivatives

- Sacado uses operator overloading-based approach for C++ codes
  - Sacado provides C++ data type for each AD mode
  - Replace scalar type (e.g., double) with AD type in your code
  - Mathematical operations replaced by overloaded versions provided by Sacado
  - Sacado uses expression templates to reduce overhead
Templating for AD

- Sacado AD types are designed for utmost efficiency of overloaded operators
  - Small, simple, highly optimized AD classes for each AD mode
  - Higher order modes are implemented by nesting lower order AD classes
  - Many AD types to incorporate into your code

- Templating to automate process of incorporating sacado AD
  - Replace scalar type with template parameter
  - Instantiate this template code on each AD data type
  - Use metaprogramming techniques to manage templates

<table>
<thead>
<tr>
<th>Data type</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>$f(x)$</td>
</tr>
<tr>
<td>DFad&lt;double&gt;</td>
<td>$f_x V$</td>
</tr>
<tr>
<td>Rad&lt;double&gt;</td>
<td>$f^T_W$</td>
</tr>
<tr>
<td>DFad&lt;DFad&lt;double&gt;</td>
<td>$(f_x V_1)_x V_2$</td>
</tr>
<tr>
<td>Rad&lt;DFad&lt;double&gt;</td>
<td>$(f^T_W)_x V$</td>
</tr>
</tbody>
</table>
#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
    Sacado::Fad::DFad<double> bfad(num_deriv, 1, b); // Second (1) indep. var
    Sacado::Fad::DFad<double> cfad(c); // Passive variable
    Sacado::Fad::DFad<double> rfad; // Result

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

    // Compute function and derivative with AD
    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
AD to TBGP

• Benefits of templating
  – Developers only develop, maintain, test one templated code base
  – Developers don’t have to worry about what the scalar type really is
  – Easy to incorporate new scalar types

• Templates provide a deep interface into code
  – Can use this interface for more than derivatives
  – Any calculation that can be implemented in an operation-by-operation fashion will work
    • i.e., any calculation who’s data can be encoded in an object that looks like a scalar where operations on that scalar can be written in closed form

• We call this extension Template-Based Generic Programming (TBGP)
  – Extended precision
  – Floating point counts
  – Logical sparsity
  – Uncertainty propagation
    • Intrusive stochastic Galerkin/polynomial chaos
    • Simultaneous ensemble propagation
Intrusive polynomial chaos through TBGP

\[ f(u, \xi) = 0, \quad \hat{u}(\xi) = \sum_{i=0}^{P} u_i \psi_i(\xi) \]

\[ \rightarrow F_i(u_0, \ldots, u_P) = \frac{1}{\langle \psi_i^2 \rangle} \int_\Gamma f(\hat{u}(y), y) \psi_i(y) \rho(y) dy = 0, \quad i = 0, \ldots, P \]

- By orthogonality of the basis polynomials

\[ (\psi_i, \psi_j) = \langle \psi_i \psi_j \rangle = \int_\Gamma \psi_i(y) \psi_j(y) \rho(y) dy = \langle \psi_i^2 \rangle \delta_{ij} \]

- The \( F_i \) are just the first \( P + 1 \) coefficients of the polynomial chaos expansion

\[ f(\hat{u}(y), y) = \sum_{i=0}^{\infty} F_i \psi_i(y) \]

- Basic idea is to compute such a truncated polynomial chaos expansion for each intermediate operation in the calculation of \( f(u, y) \)

Given \( a(y) = \sum_{i=0}^{P} a_i \psi_i(y) \), \( b = \sum_{i=0}^{P} b_i \psi_i(y) \), find \( c(y) = \sum_{i=0}^{P} c_i \psi_i(y) \)

such that \( \int_\Gamma (c(y) - \phi(a(y), b(y))) \psi_i(y) \rho(y) dy = 0, \quad i = 0, \ldots, P \)
Projections of intermediate operations

• Addition/subtraction

\[ c = a \pm b \Rightarrow c_i = a_i \pm b_i \]

• Multiplication

\[ c = a \times b \Rightarrow \sum_i c_i \psi_i = \sum_i \sum_j a_i b_j \psi_i \psi_j \rightarrow c_k = \sum_i \sum_j a_i b_j \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_k^2 \rangle} \]

• Division

\[ c = a / b \Rightarrow \sum_i \sum_j c_i b_j \psi_i \psi_j = \sum_i a_i \psi_i \rightarrow \sum_i \sum_j c_i b_j \langle \psi_i \psi_j \psi_k \rangle = a_k \langle \psi_k^2 \rangle \]

• Several approaches for transcendental operations
  – Taylor series and line integration (Fortran UQ Toolkit by Najm, Debusschere, Ghanem, Knio)
  – Tensor product and sparse-grid quadrature (Pecos/Dakota)
  – New work by Kevin Long on using the AGM method
Intrusive PCE Data Types

• By creating a new data type storing PC coefficients, and overloaded operators using these formulas, we can “automatically” propagate PC expansions (these live in Stokhos package)

\[
\text{OrthogPoly\textless double\textgreater : } \quad x(\xi) = \sum_{i=0}^{P} x_i \psi_i(\xi) \rightarrow f(x(\xi)) \approx \sum_{i=0}^{P} f_i \psi_i(\xi)
\]

• Nesting with traditional AD types enables PC expansions of derivatives

\[
\text{DFad\textless OrthogPoly\textless double\textgreater > : } \quad x(\xi) = \sum_{i=0}^{P} x_i \psi_i(\xi) \rightarrow \frac{\partial f}{\partial x}(x(\xi)) \approx \sum_{i=0}^{P} J_i \psi_i(\xi)
\]
Applying TBGP to PDEs

• Sacado overloaded operators are designed for small, dense operations
  – Avoids performance issues of sparse arrays
  – Eliminates need for row/column compression
  – Avoids issues with MPI

• PDEs don’t generate small dense computations
  – But discretizations do generate sparse combinations of small, dense computations

• Apply Sacado at PDE “element-fill” level
  – Template element-fill routines
  – Manually gather/scatter data to/from global data structures
    • Highly dependent on AD type used
    • Make it appear templated through template specialization
Templated Element Fill

Field Manager

Scatter (Extract)

PDE Terms

Properties

Source Terms

FE Interpolation

Compute Derivs

Get Coordinates

Gather (Seed)

Legend:

Global Data Structures

Local Data Structures

Generic Template Type used for Compute Phase

<EvalT>

Template Specializations for Seed and Extract phases:

Residual

Jacobian

Tangent

Hessian

Adjoint

PCE

Shape Opt
Trilinos Tools for PDEs Supporting TBGP

• **Intrepid**: Tools for discretizations of PDEs
  – Basis functions, quadrature rules, ...
  – All Intrepid classes/functions templated on scalar type
    • Derivatives w.r.t. DOFs
    • Derivatives w.r.t. coordinates

• **Phalanx**: Local field evaluation kernels
  – Organize consistent evaluation of “terms” in PDEs
  – Explicitly manages fields/evaluators for different scalar types

• **Shards**
  – Templated multi-dimensional array

• **Stokhos**
  – PCE classes, overloaded operators
  – Simultaneous ensemble propagation classes, overloaded operators
  – Tools and data structures for forming, solving embedded SG systems

• **Sacado**
  – Parameter library – tools to manage model parameters
  – Template manager – tools to manage instantiations of a template class on multiple scalar types
  – MPL – simple implementation of some metaprogramming constructs
• These ideas provide tools to implement calculations needed for embedded analysis algorithms
  – Tools to implement ModelEvaluator OutArgs
  – Connect to high level nonlinear analysis algorithms

• Examples of how to put these ideas together
  – Trilinos/packages/FEApp – simple 1D finite element code demonstrating TBGP
  – Albany – real PDE code

• These ideas really do work for complex physics
Rapid Physics Development

Albany/LCM – Thermo-Elasto-Plasticity
– J. Ostein et al

Albany/QCAD – Quantum Device Modeling
– R. Muller et al

Charon/MHD – Magnetic Island Coalescence
– Shadid, Pawlowski, Cyr

Drekar/CASL – Thermal-Hydraulics
– Pawlowski, Shadid, Smith, Cyr
Partially Embedded Optimization

- Shape optimization of a sliding electromagnetic contact
  - Salinger et al
  - Coupled electrostatics, heat conduction
  - Minimize increase in temperature
  - Analytic derivatives w.r.t. mesh coordinates
  - Finite differences of mesh coordinates w.r.t. shape parameters (FD around Cubit)
  - Dakota gradient-based optimization
Transient Sensitivities of Radiation Damage in Semiconductor Devices

Comparison to FD:
- Sensitivities at all time points
- More accurate
- More robust
- 14x faster!
Embedded UQ R&D in Albany

Navier-Stokes

Thermal-Electrostatics

Linear Problem

Nonlinear Problem

Enabling embedded UQ R&D on complex problems
Simultaneous propagation leads to greater performance

Set of N hypothetical chemical species:

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

Steady-state mass transfer equations:

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

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

- Sacado AD C++ operator overloading library (Trilinos)
- Charon implicit finite element code

DOF per element = 4*N
Simultaneous propagation of UQ sample points

- “Non-intrusive” polynomial chaos
- Simultaneous calculation of residuals & Jacobians
  - Sacado overloaded operators
- Simultaneous solution of block diagonal linear systems
  - Reuse preconditioner
  - Krylov basis recycling (Belos)
- Simple stochastic PDE
  - Albany implicit PDE code (Salinger et al)

<table>
<thead>
<tr>
<th># of uncertain parameters</th>
<th>Non-Intrusive</th>
<th>Embedded</th>
<th>Speed-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Solve Time</td>
<td>Residual + Jacobian Time</td>
<td>Solve Time</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>41</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>200</td>
<td>54</td>
</tr>
<tr>
<td>6</td>
<td>267</td>
<td>546</td>
<td>146</td>
</tr>
<tr>
<td>8</td>
<td>495</td>
<td>1094</td>
<td>315</td>
</tr>
</tbody>
</table>
Ongoing and Future Work

• Incorporating Sacado types in Tpetra
  – Indirect serialization appears to be a challenge

• Incorporating Sacado types in Kokkos MDArray
  – Expression templates?
  – Dynamic memory allocation?
  – Threading within overloaded operators?

• Rearranging embedded UQ algorithms for emerging multicore architectures
Exploit large stochastic blocks for multicore shared-memory parallelism

- Rearrange for an outer-spatial, inner-stochastic, ordering
  - Obtain very large, nearly dense blocks
  - Use sparse outer layout for distributed memory parallelism
  - Use dense inner blocks for on-node shared memory parallelism

- Requires heterogeneous multicore parallelism in complete forward uncertainty propagation calculation
  - Application fill
  - Iterative solver matrix-vector productions
  - Preconditioning

- FY12-14 SNL LDRD
Concluding Remarks

• Enable embedded algorithms through
  – Application code templating
  – Operator overloading

• Numerous advantages
  – Single templated code base to develop, test, maintain
  – Developers for the most part don’t need to worry about embedded algorithms
  – Provides hooks for current and future embedded algorithms

• Main disadvantage is dealing with templates
  – Templates are becoming ubiquitous in Trilinos
  – Template metaprogramming ideas are becoming much more common
  – C++ Template Metaprogramming by D. Abrahams and A. Gurtovoy
  – Some recent work by Argonne OpenAD group to automatically transform code to use Sacado
    • But doesn’t work with templates!
Multicore and AD-based SG propagation through application code

- Quadrature approach for an arbitrary intermediate operation:

\[ a(y) = \sum_{i=0}^{P} a_i \psi_i(y), \quad b(y) = \sum_{i=0}^{P} b_i \psi_i(y), \quad c(y) = \sum_{i=0}^{P} c_i \psi_i(y), \]

\[ c = \phi(a, b) \implies c_i = \frac{1}{\langle \psi_i^2 \rangle} \int_\Gamma \phi(a(y), b(y)) \psi_i(y) \rho(y) dy \approx \sum_{j=0}^{Q} w_j \phi(a(y_j), b(y_j)) \psi_i(y_j) \]

- 2 dense mat-vecs, for-loop, and dense mat-vec:

\[ \Psi = [\psi_i(y_j)] \in \mathbb{R}^{(P+1) \times (Q+1)}, \quad \bar{a} = [a_i] \in \mathbb{R}^{P+1}, \quad \bar{b} = [b_i] \in \mathbb{R}^{P+1}, \quad \bar{c} = [c_i] \in \mathbb{R}^{P+1}, \]

\[ A = [a(y_j)] \in \mathbb{R}^{Q+1}, \quad B = [b(y_j)] \in \mathbb{R}^{Q+1}, \]

\[ \implies A = \Psi^T \bar{a}, \quad B = \Psi^T \bar{b}, \quad \Phi = [w_j \phi(A_j, B_j)] \in \mathbb{R}^{Q+1}, \quad \bar{c} = \Psi \Phi \]

- Each scalar operation is replaced by dense matrix-vector products and easily parallelized for loops
  - Great opportunity for multicore parallelization

- Challenge: Designing overloaded operators that function effectively on GPUs