

Automating embedded analysis capabilities and managing software complexity in multiphysics simulation, Part I: Template-based generic programming

Roger P. Pawlowski, Eric T. Phipps* and Andrew G. Salinger
*Sandia National Laboratories***, Albuquerque, NM, USA

Abstract. An approach for incorporating embedded simulation and analysis capabilities in complex simulation codes through template-based generic programming is presented. This approach relies on templating and operator overloading within the C++ language to transform a given calculation into one that can compute a variety of additional quantities that are necessary for many state-of-the-art simulation and analysis algorithms. An approach for incorporating these ideas into complex simulation codes through general graph-based assembly is also presented. These ideas have been implemented within a set of packages in the Trilinos framework and are demonstrated on a simple problem from chemical engineering.

Keywords: Generic programming, templating, operator overloading, automatic differentiation, uncertainty quantification

1. Introduction

As computational algorithms, hardware, and programming languages have advanced over time, computational modeling and simulation is being leveraged to understand, analyze, predict and design increasingly complex physical, biological and engineered systems. Because of this complexity, significant investments must be made, both in terms of manpower and programming environments, to develop simulation capabilities capable of accurately representing the system at hand. At the same time, modern analysis approaches such as stability analysis, sensitivity analysis, optimization and uncertainty quantification require increasingly sophisticated capabilities of those complex simulation tools. Often simulation frameworks are not designed with these kinds of analysis requirements in

mind, which limits the efficiency, robustness, and accuracy of the resulting analysis. For example, sensitivity analysis, stability analysis, and optimization often need accurate estimates of derivatives of output quantities of interest with respect to simulation input data. These derivatives can be estimated in a “black-box” fashion using some form of finite-differencing, however this is often both very inefficient and inaccurate. This then limits the choice of algorithms that can be applied and reduces the number of quantities that can be analyzed. It is more efficient and accurate to compute these derivatives analytically by embedding the calculation within the simulation code itself. However because of the significant investments and complexity in these codes, it is often not practical to add the capabilities required by state-of-the-art analysis approaches after the fact. Instead it is generally more attractive to design the programming environments with these kinds of capabilities in mind from the beginning.

Unfortunately this idea is very difficult to implement in practice. The reality is the people who develop the simulation code capabilities are often different from those who develop the algorithmic analysis capabilities. Each group often has very detailed, in-depth knowledge of their specific domain (e.g., complex physics simulation versus state-of-the-art uncertainty

*Corresponding author: Eric T. Phipps, Department of Optimization and Uncertainty Quantification, Sandia National Laboratories, P.O. Box 5800 MS-1318, Albuquerque, NM 87185, USA. Tel.: +1 505 284 9268; Fax: +1 505 8457442; E-mail: ethipp@sandia.gov.

**Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the US Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

quantification algorithms), and it is usually impossible to be an expert in all of these domains. Furthermore, the requirement to provide the extra information needed by these analysis algorithms (e.g., derivatives) often impedes the development of the simulation code, particularly early in the software development cycle (and thus there is often significant resistance to incorporate these capabilities early). Finally, because of the long term investments in these codes, the analysis requirements can change over the life-cycle of the code and thus it may be impossible to incorporate from the beginning requirements that may arise later.

In this work, we describe an approach for building simulation code capabilities that natively support many types of analysis algorithms. This approach leverages compile-time polymorphism and generic programming through C++ templates to insulate the code developer from the need to worry about the requirements of advanced analysis, yet provides hooks within the simulation code so that these analysis techniques can be added later. The ideas presented here build on operator overloading-based automatic differentiation techniques to transform a simulation code into one that is capable of providing analytic derivatives. However we extend these ideas to compute quantities that are not derivatives such as polynomial chaos expansions, floating point counts, and extended precision calculations. In general any calculation that can be implemented in an operation-by-operation fashion can be supported, and the use of templates significantly simplifies the incorporation of many types of overloaded operators. In addition to the template-based generic programming, we also present an approach for integrating these calculations into higher-level data structures appropriate for large-scale scientific computing based on a seed-compute-extract paradigm. Finally we discuss a graph-based assembly approach for organizing the terms in the equations defining a complex multiphysics simulation and how to support generic programming and the seed-compute-extract paradigm. Numerous challenges beyond those described here arise when applying these techniques to large-scale problems such as discretizations of multi-dimensional partial differential equations. Our approach for applying template-based generic programming to these types of problems is the subject of a subsequent paper [40].

While the concepts presented here are general and can be applied with a variety of software libraries, we base our discussion on a set of tools available within Trilinos [27] that implement these ideas. In particular we describe an automatic differentiation pack-

age called Sacado [43,44] that provides the foundation for our template-based operator overloading approach. Additionally we discuss a package called Stokhos [42] that generalizes the differentiation-based operator overloading in Sacado to a specific class of forward uncertainty quantification approaches reviewed in the next section. We also present a library called Phalanx [35,38] that implements the graph-based multiphysics assembly idea supporting template-based generic programming in complex physics simulation codes.

Embedded analysis can be implemented using alternate techniques. A successful approach used in the FEniCS [32,33], Life/FEEL++ [37,45] and Sundance [34] projects is to parse a symbolic representation of the problem written in a domain specific language (DSL) to generate the embedded quantities of interest. The FEniCS project requires the problem description to be specified in the unified form language [3] and produces sensitivities through low level code generation. The Life/FEEL++ project implements the symbolic front end using C++ expression templates and template metaprogramming techniques. The Sundance project implements the symbolic front end using C++ expression objects (dynamic polymorphism) and traverses the resulting expression tree to perform in-place differentiation. While DSL approaches are quite successful, we feel that our approach provides some advantages in certain contexts. When working in legacy code environments, the template-based approach allows for easy integration with the legacy components since the data interfaces are directly exposed to the applications. In contrast, almost all DSL codes (except Sundance) constrain applications to implement the entire problem in the DSL, thus ruling out any complex operation that cannot be described by the DSL (e.g., nonlinear elimination by a third party library [58]). A second advantage is that extensibility to new analysis types is easily supported. Adding a new unexpected analysis capability such as polynomial chaos expansions to the back end parser in the symbolic approach can take considerable effort and expertise whereas with the template-based approach, the scalar type operations are overloaded regardless of the analysis capability. The primary drawback to TBGP is that the flexibility provided by TBGP requires more effort to setup the initial machinery. The DSL-based codes, on the other hand, typically have strong vertical integration and have a much lower barrier for adoption. A more detailed comparison of our assembly engine technology and DSLs can be found in [35].

This paper is organized as follows. We first provide a brief overview of several common simulation and

analysis algorithms that are employed within modern scientific computing environments in Section 2. These algorithms motivate several derivative-based and non-derivative-based quantities that need to be computed within complex simulation codes. We then review automatic differentiation techniques which provide the foundation for our work in Section 3, focusing in particular on operator overloading approaches in C++. In Section 4 we generalize operator overloading for differentiation to compute more general quantities necessary in some forms of analysis which motivates the general template-based generic programming ideas presented in Section 5. Then in Section 6 we describe a seed–compute–extract paradigm that separates phases of the template-based generic programming approach into those that are specific to the kind of analysis chosen and those that are specific to the physical problem at hand. In Section 7 we then discuss the generic graph-based assembly approach and how it incorporates template-based generic programming. The approach is illustrated through a simple physical problem arising from chemical engineering where C++ code for portions of the problem are provided. Finally in Section 8 we provide a small example demonstrating these ideas for simulating and analyzing the chemical engineering problem from the previous section. To perform these numerical calculations, several other Trilinos packages are leveraged including NOX [39] (nonlinear solver), LOCA [47,49] (stability and bifurcation analysis), Stokhos [42] (uncertainty quantification) and Rythmos [12] (time integration), all four of which are wrapped in a common interface through Piro [48] (analysis tools). We then close with brief concluding remarks.

2. Common solution and embedded analysis algorithms

Before discussing our programming approach in detail, we first motivate the approach by presenting several common solution and analysis algorithms that often require significantly advanced simulation code capabilities. As a model problem we will consider the general finite-dimensional differential-algebraic equation

$$\begin{aligned} f(\dot{x}, x, p) = 0, \quad \dot{x}, x \in \mathbb{R}^n, p \in \mathbb{R}^m, \\ f: \mathbb{R}^{2n+m} \rightarrow \mathbb{R}^n. \end{aligned} \quad (1)$$

Here x is the unknown solution vector, \dot{x} its time derivative, p is a set of model parameters, and f is the set of residual equations defining the model.

2.1. Steady-state nonlinear solves

Given a set of parameter values p , Newton-type methods are often employed to solve the steady-state version of Eq. (1) where $\dot{x} = 0$, due to its quadratic convergence rate. Given an initial guess x_0 , the standard Newton's method involves solving the linear systems

$$J(x_k, p)\Delta x_k = -f(x_k, p) \quad (2)$$

for the update Δx_k where for notational compactness we have dropped the dependence on $\dot{x} = 0$. Here $x_{k+1} = x_k + \Delta x_k$, $k = 0, 1, \dots$, and $J = \partial f / \partial x$ is the Jacobian of f . Thus the basic Newton's method requires evaluating the residual f and Jacobian J at an arbitrary point (x, p) . There are many modifications of this algorithm to provide better global convergence properties (e.g., damping, line searches, and trust regions) as well as approaches that attempt to eliminate the need to compute the Jacobian directly (e.g., Broyden methods). However we note the convergence and robustness of most of these methods are often significantly improved with accurate approximations of the Jacobian.

2.2. Steady-state sensitivity analysis and optimization

Once a steady-state solution x^* has been computed via some nonlinear solver strategy, the sensitivity of the solution to the parameters p can be computed directly through the implicit function theorem:

$$\begin{aligned} \frac{\partial f}{\partial x}(x^*, p) \frac{dx^*}{dp} + \frac{\partial f}{\partial p}(x^*, p) = 0 \\ \implies \frac{dx^*}{dp} = -J^{-1}(x^*, p) \frac{\partial f}{\partial p}(x^*, p). \end{aligned} \quad (3)$$

Often one is only interested in the sensitivity of a set of response functions $s^* = g(x^*, p)$, $g: \mathbb{R}^{n+m} \rightarrow \mathbb{R}^q$, which can be computed by

$$\begin{aligned} \frac{ds^*}{dp} &= \frac{\partial g}{\partial x}(x^*, p) \frac{dx^*}{dp} + \frac{\partial g}{\partial p}(x^*, p) \\ &= -\frac{\partial g}{\partial x}(x^*, p) \left(J^{-1}(x^*, p) \frac{\partial f}{\partial p}(x^*, p) \right) \\ &\quad + \frac{\partial g}{\partial p}(x^*, p). \end{aligned} \quad (4)$$

Thus the sensitivities ds^*/dp can be computed via m linear solves of the Jacobian J with each right-hand

side given by $\partial f / \partial p_i$. If all partial derivatives are computed analytically then the only error arising in the sensitivities comes from the error in solving these equations, which can often be controlled to some degree by the user through the choice of the linear solver algorithm. This is to be contrasted with a finite differencing approach around the nonlinear solver, which entails m nonlinear solves and introduces difficult to control finite difference truncation error. Thus this approach is significantly more efficient, and often more robust. It requires analytic evaluation of not only f and J , but also $\partial f / \partial p$, $\partial g / \partial p$, and $(\partial g / \partial x)V$ for a given matrix V .

If the number of parameters m is large but the number of response functions q is small, then a better approach is to compute adjoint sensitivities, which are given by transposing Eq. (4):

$$\begin{aligned} \left(\frac{ds^*}{dp} \right)^T &= - \left(\frac{\partial f}{\partial p}(x^*, p) \right)^T \\ &\quad \times \left(J^{-T}(x^*, p) \left(\frac{\partial g}{\partial x}(x^*, p) \right)^T \right) \\ &\quad + \left(\frac{\partial g}{\partial p}(x^*, p) \right)^T. \end{aligned} \quad (5)$$

This involves q solves of J^T where each right-hand side is given by $\partial g_i / \partial x$. It requires analytic evaluation of f , J , $\partial g / \partial x$, $\partial g / \partial p$ and $(\partial f / \partial p)^T W$ for an arbitrary matrix W .

With either the forward (4) or adjoint sensitivities (5), a variety of gradient-based optimization algorithms can be applied for efficiently optimizing the quantity of interest g .

2.3. Stability and bifurcation analysis

After a steady-state solution x^* has been found, the transient stability of the solution can be investigated through linear stability analysis. By considering infinitesimal perturbations away from the steady-state, simple analysis shows that the asymptotic stability can be understood through the generalized eigenvalue problem

$$\lambda Mz + Jz = 0, \quad (6)$$

where $M = \frac{\partial f}{\partial x}(0, x^*, p)$ and $J = \frac{\partial f}{\partial x}(0, x^*, p)$. If all of the eigenvalues λ have a negative real part, then the solution is stable in a neighborhood of the solu-

tion. If any eigenvalue has a positive real part, the solution is unstable and if any has zero real part, higher-order analysis is needed. Thus in addition to the Jacobian J , derivatives with respect to the transient terms are needed. Moreover, for an accurate determination of stability, it is critical to have exact analytic derivatives J and M .

The location of the steady-state and its stability can be tracked as a function of the parameters p through techniques such as pseudo-arclength continuation [30]. Moreover parameter values at which a change in stability occurs, i.e. a bifurcation, can be solved for directly. For example, a stable and unstable steady-state may collide as a parameter is increased yielding no steady-state for parameter values larger than where the bifurcation occurs, commonly called a saddle-node, fold, or turning-point bifurcation. In this case one can show that Eq. (6) has at least one solution with a zero eigenvalue λ , and thus J has a non-trivial null-space [23]. Such a point can be computed directly by solving the following augmented system of equations [19]:

$$f(x, p) = 0, \quad \sigma(x, p) = 0, \quad (7)$$

where

$$\begin{aligned} \sigma &= -u^T Jv, \quad \begin{bmatrix} J & a \\ b^T & 0 \end{bmatrix} \begin{bmatrix} v \\ s_1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \\ \begin{bmatrix} J^T & b \\ a^T & 0 \end{bmatrix} \begin{bmatrix} u \\ s_1 \end{bmatrix} &= \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \end{aligned} \quad (8)$$

and a and b are given vectors. This system has a unique solution at a saddle-node bifurcation for almost all choices of a and b , under suitable conditions [19]. Applying a Newton-type nonlinear solver to Eq. (7) requires calculation of the derivatives $\partial \sigma / \partial x$ and $\partial \sigma / \partial p$. From Eq. (8) it can be shown that

$$\begin{aligned} \frac{\partial \sigma}{\partial x} &= -u^T \frac{\partial}{\partial x}(Jv), \\ \frac{\partial \sigma}{\partial p} &= -u^T \frac{\partial}{\partial p}(Jv). \end{aligned} \quad (9)$$

Thus this method requires calculation of various second derivatives. As with stability analysis, it is critical to have an accurate approximation of the Jacobian J since it appears directly in the equations defining the bifurcation. It is less critical to accurately evaluate the second derivatives appearing in Eq. (9) since they are only used in the Newton solver method, but as discussed above, robustness and convergence are often improved if these derivatives are computed accurately.

2.4. Transient analysis

There are a variety of explicit and implicit time integration strategies for computing numerical solutions to the transient problem (1) with suitable initial conditions. For stiff or differential-algebraic problems, implicit backward differentiation formulas (BDF) [24, 25] are often applied. These methods approximate \dot{x} through the multistep formula

$$h_n \dot{x}_n = \sum_{i=0}^q \alpha_i x_{n-i}, \quad (10)$$

where q is the order of the method, h_n is the time step size at time step n , and x_j, \dot{x}_j represent the solution and its time derivative respectively at time step j . Substituting Eq. (10) in Eq. (1) we obtain the following nonlinear system to solve for x_n :

$$f\left(\frac{1}{h_n} \sum_{i=0}^q \alpha_i x_{n-i}, x_n, p\right) = 0. \quad (11)$$

Solving this system for x_n with a Newton-type method requires solving linear systems of the form

$$\left(\frac{\alpha_0}{h_n} \frac{\partial f}{\partial \dot{x}} + \frac{\partial f}{\partial x}\right) \Delta \mathbf{x}_n = -f. \quad (12)$$

Thus derivatives similar to the eigensystem Eq. (6) must be computed.

2.5. Uncertainty quantification

The last form of analysis we will consider is one where the parameters p appearing in Eq. (1) have uncertainty associated with them. Typically this uncertainty manifests through lack of knowledge as to what the values of the parameters in the system model (1) should be (so-called epistemic uncertainty), or represents some variability in realizations of those parameters (so-called aleatory uncertainty). For analysis purposes, the uncertainty must be represented in some mathematical fashion, such as bounds or intervals in the first case, or random variables with prescribed probability distributions in the second. Then the uncertainty quantification problem is to estimate a representation of the uncertainty of the corresponding solution x to Eq. (1) (either steady or transient). There are a variety of computational approaches for this in

the literature, but recently response surface methods have become popular. These methods compute a response surface approximating the mapping $x(p)$, which can then be used to approximate the uncertainty in x . For example, $x(p)$ can be sampled to compute bounds on x for epistemic analysis or probabilities and statistics in an aleatory analysis.

One response surface approximation method in particular that has seen significant attention in the literature is one based on approximating $x(p)$ in terms of orthogonal polynomials, often called polynomial chaos [17,18,56,57]. For simplicity, consider the steady-state version of Eq. (1) where to represent the uncertainty in the parameters p we replace them with random variables ξ with a known probability measure μ and density function ρ . More precisely, assume (Ω, \mathcal{B}, P) is a given probability space, with sample space Ω , σ -algebra \mathcal{B} representing admissible events, and probability measure $P: \mathcal{B} \rightarrow [0, 1]$, then $\xi: \Omega \rightarrow \mathbb{R}^m$, $\mu = P \circ \xi^{-1}$ is the measure of ξ , and $\rho = d\mu/d\lambda$ where λ is Lebesgue measure on \mathbb{R}^m . Then formally $x(\xi)$ can be written as the following sum:

$$x(\xi) = \sum_{i=0}^{\infty} x_i \psi_i(\xi), \quad (13)$$

where $\{\psi_i: i = 0, 1, 2, \dots\}$ is a family of polynomials orthogonal with respect to the measure P :

$$\begin{aligned} & \int_{\Omega} \psi_i(\xi(\omega)) \psi_j(\xi(\omega)) dP \\ &= \int_{\mathbb{R}^m} \psi_i(\xi) \psi_j(\xi) d\mu \\ &= \int_{\mathbb{R}^m} \psi_i(y) \psi_j(y) \rho(y) dy = \langle \psi_i^2 \rangle \delta_{ij} \end{aligned} \quad (14)$$

and where $\langle \cdot \rangle = \int \cdot \rho dy$. The convergence of this sum is in the $L_P^2(\Omega)$ sense. The coefficients x_i are generalized Fourier coefficients, and thus Eq. (13) is often called a spectral representation of $x(\xi)$.

For computational purposes, the sum (13) must be truncated at some finite order N and the coefficients x_i must be approximated. Owing to the orthogonality relation (14), we have

$$x_i = \frac{1}{\langle \psi_i^2 \rangle} \int_{\mathbb{R}^m} x(y) \psi_i(y) \rho(y) dy \quad (15)$$

which can be approximated by a multi-dimensional

quadrature rule $\{(w^k, y^k): k = 0, \dots, Q\}$ defined by ρ :

$$x_i \approx \frac{1}{\langle \psi_i^2 \rangle} \sum_{k=0}^Q w^k x(y^k) \psi_i(y^k). \quad (16)$$

For each k , $x^k = x(y^k)$ is computed by solving $f(x^k, y^k) = 0$. This is the so-called non-intrusive spectral projection (NISP) [46] method. In high dimensions m , an efficient quadrature rule is difficult to find for a general random vector ξ . When the components are independent, sparse-grid quadrature rules [36] are the most effective. Thus a sequence of $Q + 1$ steady-state solves are necessary for each quadrature point realization y_k of the random parameters ξ .

Another approach for approximating the coefficients x_i , the so-called stochastic Galerkin approach [17,18], is to solve the equations

$$\begin{aligned} F_i &\equiv \frac{1}{\langle \psi_i^2 \rangle} \int_{\mathbb{R}^m} f(\hat{x}(y), y) \psi_i(y) \rho(y) dy \\ &= 0, \quad i = 0, \dots, N, \end{aligned} \quad (17)$$

where $\hat{x}(y) = \sum_{i=0}^N x_i \psi_i(y)$. This generates a new, fully-coupled set of nonlinear equations for all of the x_i :

$$\begin{aligned} F(X) &= 0, \\ X &= [x_0, \dots, x_N]^T, F = [F_0, \dots, F_N]^T. \end{aligned} \quad (18)$$

To solve these equations with Newton's method, calculation of the Jacobian matrix $\partial F / \partial X$ is necessary. From Eq. (17) one can show

$$\begin{aligned} \frac{\partial F_i}{\partial x_j} &= \frac{1}{\langle \psi_i^2 \rangle} \int_{\mathbb{R}^m} \frac{\partial f}{\partial x}(\hat{x}(y), y) \psi_i(y) \psi_j(y) \rho(y) dy \\ &\approx \sum_{k=0}^N J_k \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_i^2 \rangle}, \end{aligned} \quad (19)$$

where

$$J_k = \frac{1}{\langle \psi_k^2 \rangle} \int_{\mathbb{R}^m} \frac{\partial f}{\partial x}(\hat{x}(y), y) \psi_k(y) \rho(y) dy \quad (20)$$

are the polynomial chaos coefficients of the Jacobian operator $\partial f / \partial x$. Unlike the NISP method above, this method is intrusive as it requires the formulation and solution of this new nonlinear system $F(X) = 0$, and

thus requires significant software infrastructure to implement. In particular the simulation code must implement a method for computing the stochastic Galerkin residuals F_i and Jacobians J_k . The primary reason for doing so is that typically the number N of unknown coefficients x_i is significantly smaller than the number Q of quadrature points y_k , which can lead to computational savings in some cases [15].

3. Automatic differentiation through operator overloading

As we have seen above, advanced simulation and analysis algorithms require accurate and efficient evaluation of a variety of first and higher derivatives, as well as evaluation of other quantities such as polynomial chaos expansions. For the reasons discussed in the Introduction, it is not practical to expect a simulation code implementing the evaluation of f in Eq. (1) to provide implementations of all of these calculations. Thus techniques for automatically generating this information are necessary. The conceptual foundation for our approach to automating these calculations lies with automatic differentiation techniques, specifically through operator overloading.

3.1. Introduction to automatic differentiation

Automatic differentiation (AD) is a well-known set of techniques for transforming a given computation implemented in a programming language, into one that computes derivatives of that computation. For an in-depth introduction to AD techniques we refer the reader to [21] and the references contained within. Briefly, automatic differentiation works by recognizing that any computation implemented in a programming language must be a composition of simple arithmetic operations (addition, subtraction, multiplication, and division) and standard mathematical functions (sine, exponential, etc.). All of these have known formulas for computing their derivatives, and thus computer code for evaluating the derivative of the computation can be generated by combining these formulas with the chain rule of basic differential calculus. Automatic differentiation is just a method for automating this process, and the end result is computer code that computes the derivative of the calculation at a given set of inputs. Most realistic computations also involve branches, e.g., conditionals and loops, which can be handled by a variety of mechanisms. The simplest is

to only compute the derivative of the branch that was evaluated at the given set of inputs upon which the original calculation was based (this means the choice of branches within the derivative code is only based on the values of the original calculation and not its derivatives). For more detail on approaches for dealing with branches, discontinuities, and iteration, see [21].

There are different modes of automatic differentiation that accumulate derivatives in different orders, namely the forward and reverse modes, as well as modes to compute higher order derivatives (e.g., multivariate tensors and univariate Taylor series). For example, if the calculation to be differentiated is represented mathematically as $y = f(x)$ where $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$, then given a point $x_0 \in \mathbb{R}^n$ and an arbitrary matrix $V \in \mathbb{R}^{n \times p}$, the forward mode of AD computes the derivative (see the discussion below on seed matrices as to why this form is chosen)

$$\frac{dy}{dx}V = \frac{df}{dx}(x_0)V = \frac{d}{dz}f(x_0 + Vz) \Big|_{z=0}. \quad (21)$$

Forward AD implements this calculation by computing derivatives with respect to z at each stage of the calculation using rules such as those from Table 1, propagated forward starting from the initialization $dx/dz = V$. Owing to the directional derivative nature of formula (21), this method is often called tangent propagation. One can then easily show [21] that the cost of the resulting derivative calculation is bounded by

$$\text{cost}\left(\frac{dy}{dx}V\right) \sim (1 + 1.5p) \text{cost}(f), \quad (22)$$

where $\text{cost}(f)$ is the cost to evaluate the function f . Since all of the columns of V are usually propagated

simultaneously, the cost of forward mode AD seen in practice is often significantly less [8].

Similarly, given an arbitrary matrix $W \in \mathbb{R}^{m \times q}$, reverse AD computes

$$W^T \frac{dy}{dx} = W^T \frac{df}{dx}(x_0) = \frac{d}{dx}(W^T f(x)) \Big|_{x=x_0} \quad (23)$$

by computing derivatives of $z = W^T y$ with respect to each intermediate quantity using rules such as those from Table 2, propagated backward starting from the initialization $dz/dy = W^T$. This mode is often referred to as adjoint or gradient propagation. One can show the cost is given by

$$\text{cost}\left(W^T \frac{dy}{dx}\right) \sim 4q \text{cost}(f). \quad (24)$$

Thus for a scalar valued calculation ($m = 1$), its gradient ($q = 1$) can be computed in the cost of about four function evaluations, regardless of the number of independent variables. Note that since reverse AD requires propagating derivatives backwards through the calculation, some facility for reversing the code evaluation is necessary. Typically this is accomplished by building a data structure storing the computational graph of the calculation generated through a forward pass. This data structure stores the values of all of the intermediate operations, some mechanism for computing the derivative of each operation (either encoding some functional representation of the operation or a partial derivative computed in the forward pass), and connectivity representing the arguments of each operation. This data structure can then be traversed in reverse order to accu-

Table 1

Forward AD differentiation rules for several scalar intermediate operations

Operation	Forward AD rule
$c = a \pm b$	$\dot{c} = \dot{a} \pm \dot{b}$
$c = ab$	$\dot{c} = a\dot{b} + \dot{a}b$
$c = a/b$	$\dot{c} = (\dot{a} - c\dot{b})/b$
$c = a^r$	$\dot{c} = ra^{r-1}\dot{a}$
$c = \sin(a)$	$\dot{c} = \cos(a)\dot{a}$
$c = \cos(a)$	$\dot{c} = -\sin(a)\dot{a}$
$c = \exp(a)$	$\dot{c} = c\dot{a}$
$c = \log(a)$	$\dot{c} = \dot{a}/a$

Note: Overdots denote differentiation with respect to a single independent variable z .

Table 2

Reverse AD rules for several scalar intermediate operations

Operation	Reverse AD rule
$c = a + b$	$\bar{a} = \bar{c} \quad \bar{b} = \bar{c}$
$c = a - b$	$\bar{a} = \bar{c} \quad \bar{b} = -\bar{c}$
$c = ab$	$\bar{a} = \bar{c}b \quad \bar{b} = \bar{c}a$
$c = a/b$	$\bar{a} = \bar{c}/b \quad \bar{b} = -\bar{c}c/b$
$c = a^r$	$\bar{a} = \bar{c}ra^{r-1}$
$c = \sin(a)$	$\bar{a} = \bar{c}\cos(a)$
$c = \cos(a)$	$\bar{a} = -\bar{c}\sin(a)$
$c = \exp(a)$	$\bar{a} = \bar{c}c$
$c = \log(a)$	$\bar{a} = \bar{c}/a$

Notes: Overbars over each intermediate variable denote differentiation of a single dependent variable z with respect to that intermediate variable, e.g., $\bar{a} = dz/da$. For examples putting these rules together to differentiate a function, we refer the reader to [21].

mulate the derivatives. However because of the overhead associated with traversing this data structure, often the cost of reverse mode AD is somewhat higher than the bound (24) above [8].

The matrices V and W are referred to as seed matrices, and the user is allowed to choose them to suit their purposes. For example if the whole Jacobian matrix dy/dx is desired, forward or reverse mode can be used with the corresponding seed matrix set to the identity. Similarly, the seed matrix can be set to a single vector to compute Jacobian-vector or Jacobian-transpose-vector products with significantly less cost than computing the whole matrix. Finally, the forward and reverse modes can be applied recursively to compute a variety of higher derivatives.

3.2. Automatic differentiation software tools

There are a number of software tools available that implement the various modes of automatic differentiation discussed above in a variety of programming languages. Most tools can be put into two categories: those based on source transformation and those based on operator overloading. Source transformation tools operate by reading the source code to be differentiated, parsing it, building an internal representation of the calculation, applying the AD rules discussed above, generating derivative code and writing this code out to disk. The derivative code can then be compiled and linked into an application using standard compiler tools. It is up to the user to write additional code that calls the differentiated routines and incorporates the derivatives in whatever way necessary. This approach has been most successful for languages such as Fortran through tools such as ADIFOR [10], Tapenade [26], and OpenAD [51], but has also been used for C (e.g., ADIC [11]) and recently some C++ (e.g., OpenAD and TAC++ [55]). This approach is ideal in the sense that it provides the automatic differentiation tool a global view of the code to be differentiated, allowing optimization of the resulting derivative calculation.

However the source transformation approach has seen less use with the C++ language, primarily due to the complexity of parsing and supporting all of the features of ANSI C++. Instead the simpler approach of leveraging the native classing, operator overloading, and templating language features of C++ is more popular. This approach is library based, whereby new data types are created to store both function values and derivative values, and overloaded versions of the arithmetic and math functions are provided that implement

the derivatives of those operations operating on these new data types using rules such as those from Tables 1 and 2. Then the floating point data type in a given calculation (e.g., `float` or `double`) is replaced by the corresponding AD data type. The compiler then applies its operator overloading resolution rules to replace the original function calls with the differentiated versions. Templating can be used (although is not required) to automate the necessary type replacement. As with source transformation, the user must provide additional code to call the differentiated calculation, as well as code to initialize the AD data types and extract the resulting derivatives.

There are numerous tools available implementing this approach, including ADOL-C [22], FAD [4,5], FADBAD [9] and Sacado [43]. All of the automatic differentiation modes can be implemented this way, and it can be applied to any language that supports derived types and operator overloading (including Python and Matlab). However the disadvantage of this approach is that the resulting derivative calculation can be less efficient than the one resulting from source transformation. This is partly due to the fact that operator overloading-based tools can only manipulate individual operations and thus do not have an opportunity to optimize the whole derivative calculation, but is mainly due to the extra overhead operator overloading introduces. In essence, each operation in the original calculation is replaced by a function call, which adds overhead. Moreover, C++ requires that the overloaded operators return copies of the object created within them representing the result of the differentiated operation, which is then copied into the object to store the result in the calling code. This again adds overhead. Some of this overhead can be automatically eliminated by the compiler through aggressive function inlining and optimization. However more recent AD tools such as FAD and Sacado employ expression templates to trick the compiler into eliminating virtually all of this overhead.

3.3. Expression templates

Expression templates are a general set of techniques that make operator overloading more efficient [53]. Expression templates have been used in numerous contexts such as the Blitz++ vector library [54], the LLANO [31] and Blaze [29] dense linear algebra libraries, and the Playa sparse linear algebra package [28], however for concreteness we will describe them in the context of AD. Essentially they work by creating a new data type for each expression in the calculation.

This data type is a tree that encodes the structure of the expression, the types of operations in the expression, and their arguments. Instead of differentiating each individual operation directly, each overloaded operator appearing in an expression adds a node to the tree to represent the operation by combining the expression trees for its arguments. Thus the arguments and return type of the operator are expressions instead of AD data types. An overload of the assignment operator must then be provided for the AD data type with an arbitrary expression as the right-hand side which then recursively differentiates the entire expression. The type of each operation in the expression can be specified by a template parameter, and each node in the tree must only store a reference to the expressions representing its arguments. Thus the creation of the expression template involves no runtime calculation, and the entire expression object can usually be eliminated through aggressive compiler optimizations. This eliminates much of the overhead of operator overloading by eliminating the creation and copying of temporary objects and fusing the differentiation of all of the operations within an expression. This latter optimization allows a single loop to be created iterating over all of the components of the derivative array, instead of having separate loops for each operation. Moreover, if the length of this array (i.e., the number of independent variables) is known at runtime, this loop can be unrolled by the compiler.

FAD [4,5] was the first AD library we are aware of that implemented expression templates, and these ideas were later incorporated into the Sacado [43] AD library in Trilinos. Sacado provides further refinements of the expression template ideas in FAD to make them more efficient [41]. In particular, Sacado implements an expression level reverse mode where each operation is differentiated by the reverse mode AD approach. This reduces the number of floating point operations by recognizing that an expression often has many inputs, but only one output. The expression-level reverse mode computes the gradient of this output with respect to the expression inputs, and then applies the chain rule to obtain the derivatives with respect to the independent variables. Furthermore, Sacado can cache the value of each operation in the expression tree for use in the subsequent derivative evaluation. This trades a small amount of data storage required of the expression for a reduction in the number of floating point operations. This is particularly important when nesting AD types for higher derivatives or for other kinds of non-derivative analysis (such as polynomial chaos expansions) discussed below. These opti-

mizations together with modern optimizing compilers significantly improve the performance of the forward AD tools [41], virtually eliminating the overhead typically associated with operator overloading (see Fig. 1). In addition to forward mode AD, Sacado also provides reverse-mode AD tools called RAD [8,16]. Both the forward and reverse mode AD data types can be combined to compute a variety of higher derivatives through a general template mechanism discussed below.

4. Generalized operator overloading for embedded analysis

The operator overloading approaches discussed above provide a relatively simple mechanism for generating analytic derivatives of arbitrarily complex calculations. This idea can be extended in a rather straight forward manner to compute other kinds of quantities that are not derivatives. For example, several packages are available that compute floating point operations in higher than double precision (e.g., [20]). In C++ these packages work by implementing a new data type that stores the floating point value for each operation, and overloads all of the same operations discussed above to compute these operations in extended precision. Similarly, Sacado (and likely other AD packages) provides a simple floating point operation counting class that, along with its overloaded operators, can count the number of floating point operations (as well as other operations such as copies and assignments) in a given calculation.

Finally, operator overloading can be used to generate information needed for the uncertainty quantification methods discussed in Section 2.5. For example, the stochastic Galerkin method requires the evaluation of the stochastic Galerkin residual equations (17). This calculation can be implemented by creating a data type storing the polynomial chaos coefficients for a given intermediate quantity used in the evaluation of the residual function f . Then overloaded operators can be written that compute these coefficients for each type of intermediate operation. For example, assume a and b are two intermediate quantities in the calculation for f and assume by way of induction that their polynomial chaos expansions

$$a(\xi) = \sum_{i=0}^N a_i \psi_i(\xi), \quad b(\xi) = \sum_{i=0}^N b_i \psi_i(\xi) \quad (25)$$

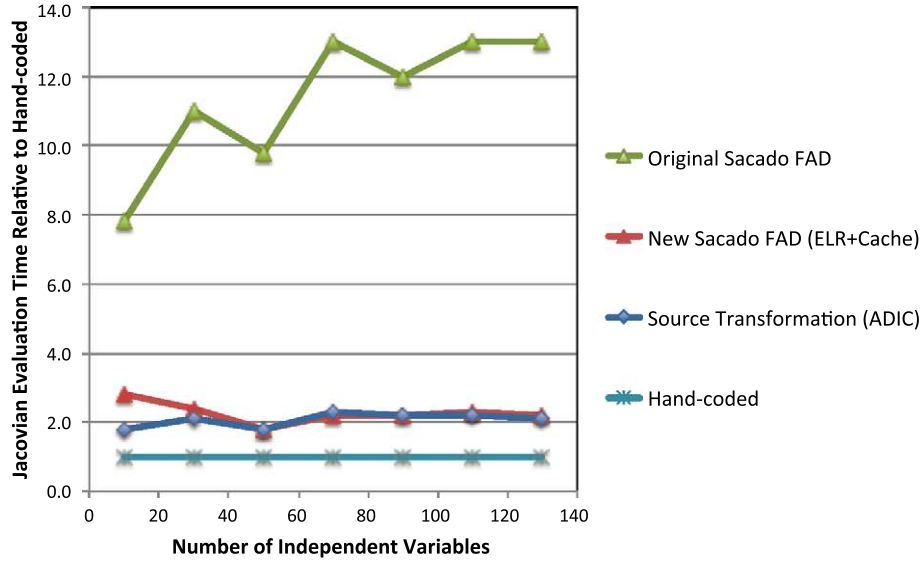


Fig. 1. Comparison of Jacobian evaluation times for a simple vector PDE using several approaches: the original Sacado FAD AD tools in Sacado based on the original FAD library, the new Sacado FAD tools incorporating expression-level reverse mode and caching, source transformation with ADIC [11] and a hand-coded, optimized Jacobian evaluation. Times are relative to the hand-coded approach and all cases were implemented using recent GNU C/C++ compilers (4.6.3) with standard `-O3` optimization flags. In all cases the Jacobian values agree to machine precision. The comparison to source transformation demonstrates that for a sufficiently large number of independent variables, the overhead traditionally associated with operator overloading can be eliminated by the new expression template implementations. (Colors are visible in the online version of the article; <http://dx.doi.org/10.3233/SPR-2012-0350>.)

have already been computed. We wish to compute the chaos expansion

$$c(\xi) = \sum_{i=0}^N c_i \psi_i(\xi), \quad (26)$$

where $c = \varphi(a, b)$ is given by a standard elementary operation of a and b . If $c = a \pm b$ then clearly $c_i = a_i \pm b_i$, $i = 0, \dots, N$. If $c = ab$, then by orthogonality of the basis polynomials we have

$$c_k = \sum_{i,j=0}^N a_i b_j \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_k^2 \rangle}, \quad k = 0, \dots, N. \quad (27)$$

Similarly if $c = a/b$, then we obtain the following linear system of equations to solve for the coefficients c_i :

$$\sum_{i,j=0}^N c_i b_j \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_k^2 \rangle} = a_k, \quad k = 0, \dots, N. \quad (28)$$

Computing stochastic Galerkin projections of transcendental operations is more challenging, and several approaches have been investigated [14]. A relatively simple and robust approach is quadrature inspired by

non-intrusive spectral projection:

$$c_k \approx \frac{1}{\langle \psi_k^2 \rangle} \sum_{j=0}^Q w^j \varphi(a(y^j), b(y^j)) \psi_k(y^j), \quad k = 0, \dots, N. \quad (29)$$

Sacado, together with the stochastic Galerkin package Stokhos [42] implement these approaches to provide an operator overloading mechanism for generating stochastic Galerkin residual values.

5. Template-based generic programming and compile-time polymorphism

As we have seen, a myriad of analysis capabilities can be incorporated into complex simulation codes through the creation of specialized types, which we will call scalar types, and corresponding overloaded operators. Applying each of these to the code requires replacing the basic floating point type with a new scalar type. Since this conversion must happen for many different types, we advocate a templating approach where the scalar type is replaced by a template parameter. When this template code is instantiated on the origi-

nal floating point type, the original code is obtained (and thus no performance is lost in the original calculation), and when it is instantiated on each new scalar type, the corresponding overloaded evaluation is obtained. This simplifies extending the AD library to support new modes since only a new scalar data type and corresponding overloaded operators must be provided and the template code can be easily instantiated on this new type. Furthermore the code developer only needs to develop, test, and maintain one templated code base. Finally the implementation of the Sacado scalar data types and overloaded operators are themselves templated on the scalar type to allow nesting of types. For example, by replacing the floating point type in a forward mode AD evaluation with the forward mode AD data type, a second derivative of the form

$$\frac{d}{dx} \left(\frac{df}{dx} V_1 \right) V_2 \quad (30)$$

is obtained. Similarly, by nesting forward AD inside reverse AD, one can compute

$$W^T \frac{d}{dx} \left(\frac{df}{dx} V \right). \quad (31)$$

Finally, by nesting the stochastic Galerkin scalar type inside the forward AD type, one can obtain the polynomial chaos expansion of the Jacobian from Eq. (20).

Once the relevant portions of an application code have been templated in this manner, a very deep and powerful interface into the computation is provided which can be leveraged for the several analysis tasks discussed above. By providing a template version of the code, the application developer is essentially creating a generic version of their calculation that can be transformed into a new calculation through template instantiation. Clearly this template instantiation occurs at compile-time and the corresponding overloaded operators for the chosen scalar type are for the most part incorporated automatically through the compiler's normal template deduction and overload resolution mechanisms. Thus this approach introduces compile-time polymorphism in each of the mathematical operations invoked in a given calculation through template-based generic programming.

6. Seed–compute–extract paradigm

As was seen in Section 3, the forward mode of automatic differentiation requires seeding the AD objects

representing the independent variables to initialize the AD calculation. Once seeded, the derivative calculation proceeds by evaluating the calculation with the scalar type template parameter replaced by the forward AD type, starting from the seeded independent variables. When the calculation arrives at the dependent variables, the AD objects store the values and derivatives of the dependent variables with respect to the seeded independent variables (Eq. (21)). Here the function values and derivative values must be extracted and communicated to the analysis algorithm requesting them. This general paradigm of seeding the objects representing the independent variables, evaluating the code on these objects by instantiating the template code on the corresponding types, and extracting the results is general and applies to all of the analysis types discussed above.

From a generic programming perspective, the only difference between all of these different evaluations is the seeding and extraction phase, and these are the only points in the calculation where each scalar type must be explicitly referenced. Moreover, the seeding and extraction is for the most part independent of the details of the actual calculation that is being transformed, and only depends on the identification of independent and dependent variables (i.e., which quantities in the program these correspond to and how many of them there are). We have found it useful to create templated seeding and extraction classes explicitly specialized for each different type (see, for example, [52] for a discussion on template specialization). For a given simulation code, these classes only need to be written once and will suffice for all of the different problems that code simulates, and localizes the scalar type-specific parts of the calculation to one or two classes (for each scalar type). And since these classes are templated, they can be incorporated into the general templated residual evaluation where the compiler will automatically choose the correct specialization during its template instantiation process. This structure dramatically simplifies both the addition of new problems or physics and new scalar types for new types of analysis. In the first case, only the compute parts of the evaluation need to be modified or extended and this will generally be independent of the scalar types. In the latter case, only new seed/extract classes need to be provided which is independent of the physics or problem dependent parts of the evaluation. These seed and compute classes provide the linkage between the simulation code evaluation of an analysis quantity (residual, Jacobian, etc.) and the high-level data structures

needed for the analysis algorithm (e.g., parallel vector and matrix data structures). Since these classes are explicitly specialized (and thus are not template code), these high-level data structures themselves do not need to be templated.

From a software engineering perspective, the actual implementation of the seed–compute–extract paradigm can be rather complex. In Section 7 we will discuss these issues in greater detail with respect to the Trilinos tools.

7. Handling complexity

The template-based generic programming concepts described above allow for a rich analysis environment. However the software engineering complexity related to implementing these concepts grows quickly. A Trilinos package called Phalanx [38] was implemented to simplify the use of template-based generic programming. While the main use of this package focuses on partial differential equation (PDE) assembly, it was written to support any type of evaluation and is not limited to cell-based discretizations.

In this section we will address three central complexity issues that arise when evaluating quantities. The first issue deals with the complexity introduced when supporting a myriad of analysis algorithms. Each of the algorithms described in Section 2 requires the evaluation of specific object types such as residuals, Jacobians, and parameter sensitivities. Providing a flexible and extensible framework for object evaluation is essential. The second issue is complexity associated with the algorithm used to construct multiphysics models. In this case, the ability to robustly switch between different equation sets and constitutive models is critical. The final issue is complexity associated with data management. Control of the allocation and placement of data on hardware is critical for efficient assembly routines. A flexible system must be implemented to allow “power users” to control data allocation when possible for specific architectures/hardware.

7.1. Evaluation types

The seed–compute–extract paradigm of Section 6 is a powerful tool for separation of algorithmic analysis requirements from the implementation of the physics models. However templating directly on the scalar type presents a number of difficulties. Instead, the code base is templated on an *evaluation type*. An evaluation type

is a simple C++ struct that corresponds directly to an object that is required by the analysis algorithms in Section 2. For example, residual, Jacobian, parameter sensitivities, stochastic residual, stochastic Jacobian, and Hessian are all evaluation types.

By templating on the evaluation type, any method associated with the evaluation process can be specialized for the particular evaluation type using template specialization. This is invaluable when writing the seed and extract methods. In a previous implementation, the code was templated on the scalar type and any specializations based on the evaluation type were done at runtime using dynamic polymorphism with abstract base classes. This added to the complexity of the code by requiring that a number of objects with pure virtual interfaces be built and passed through the code whenever a specialization was needed. By moving to static polymorphism, the code is much cleaner to read and debug. Additionally the multiplicity is handled at compile time, reducing the overhead of virtual function lookups if the specialization is nested in a low level loop in the assembly kernel.

By templating on an evaluation type, a scalar type can then be used in multiple evaluation types. For example, a Jacobian evaluation and a parameter sensitivity evaluation can both use a Sacado forward AD (FAD) object, `Sacado::Fad::DFad<double>`. The difference is that the seed method and extract method will operate on different objects. In the case of a Jacobian evaluation, the FAD objects for the degrees of freedom will be seeded while for a parameter sensitivity evaluation, the FAD object for the parameter of interest will be seeded. In the extract phase, a Jacobian sensitivity evaluation will extract FAD derivative values into a Jacobian matrix, while the parameter sensitivity evaluation will extract FAD derivative values into a vector (or multi-vector for multiple parameters). If the code were specialized based on the scalar type, then we could not differentiate the above calculation without falling back on runtime polymorphism or partial specialization.

The definition of evaluation types and all supported scalar types is handled through a single struct in Phalanx. It is called a “Traits” object but instead of implementing a true traits mechanism (see, for example, [2]) it is in effect a type glob to simplify implementation requirements (future work may include transitioning this into a true traits mechanism). This class defines each evaluation type and binds a default scalar type to the evaluation type by declaring a typedef for `ScalarT`. The following example defines the scalar

types and evaluation types for implementing a residual, a Jacobian, a Jacobian-vector product and a Hessian.

```
struct UserTraits : public PHX::TraitsBase {
// Scalar Types
typedef double RealType;
typedef Sacado::Fad::DFad<RealType> FadType;
typedef Sacado::Fad::DFad<
  Sacado::Fad::DFad<RealType> > FadFadType;

// Evaluation Types with default scalar type
struct Residual { typedef RealType ScalarT; };
struct Jacobian { typedef FadType ScalarT; };
struct Tangent { typedef FadType ScalarT; };
struct Hessian { typedef FadFadType ScalarT; };
.
.
.
}
```

An evaluation is usually associated with one scalar type, but can use multiple scalar types. For example a Jacobian calculation normally uses a Sacado FAD object to compute the derivatives. However if a user knows that a certain piece of physics is insensitive to the degree of freedom, then they can attempt to speed up the code by replacing the FAD data type with a data type that will not carry out the derivative computations.

The construction of objects required for an evaluation type is simplified by using static polymorphism. We automate the building of objects by implementing compile time loops over evaluation types using template metaprogramming techniques [1] implemented in the Sacado and Boost [13] libraries. The user then only needs to list the evaluation types they would like to build in the Phalanx Traits struct.

```
struct UserTraits : public PHX::TraitsBase {
.
.
.
typedef Sacado::mpl::vector<Residual, Jacobian,
  Tangent, Hessian> EvalTypes;
.
.
.
}
```

7.2. Multiphysics complexity

Multiphysics simulation software is plagued by complexity stemming from the multitude of models that must be supported in simulating complex physics systems. Such software environments must support the ability to flexibly adapt the physics models as a simulation transitions to different regimes/problems.

This might require the modification/addition/deletion of equations, constitutive laws and material models. Supporting a multiplicity in physics models often results in complex algorithms and rigid software. The Phalanx library implements a new design approach that automates the assembly of an evaluation type (e.g., residuals and Jacobians) and eliminates the inherent complexity in physics models. A thorough description and analysis of the design is beyond the scope of this paper but can be found in [35]. This section will briefly summarize the design principles and focus on how the template-based generic programming concepts are applied to the framework.

7.2.1. The concept: Data centric assembly

The basic idea is to shift the evaluation of an analysis quantity from an algorithmic focus (i.e., the properly-ordered sequence of steps to assemble the quantity of interest) to the concept of data and their low-level dependencies. In essence, we are decomposing a complex problem into a number of simpler problems with managed dependencies. By exposing data dependencies, an algorithm can be constructed automatically from the simpler pieces.

This concept is best explained by example. Here we choose a simple Continuous Stirred Tank Reactor (CSTR) model of a first-order, exothermic, irreversible chemical reaction $A \rightarrow B$ in a tank with ideal mixing and an Arrhenius temperature dependence of the reaction rate [50]. Note that this is a trivial example for demonstration purposes. In general, balance of either mass or energy requires

$$[\text{accumulation}] = [\text{in}] - [\text{out}] + [\text{generation}] - [\text{consumption}] \quad (32)$$

for both the quantity of chemical species A and B , and the temperature of the fluid. If we consider an inlet flow of pure A with concentration c_{A_f} and temperature T_f at a rate λF mixed with a recycle flow with rate of $(1 - \lambda)F$ in a reactor of volume V surrounded by a cooling jacket of temperature T_c this becomes

$$\begin{aligned} V \frac{dc_A}{dt} &= [F(\lambda c_{A_f} + (1 - \lambda)c_A)] \\ &\quad - [Fc_A] + [0] - [Vr_{A \rightarrow B}], \\ V \rho C_p \frac{dT}{dt} &= [\rho C_p F(\lambda T_f + (1 - \lambda)T)] \quad (33) \\ &\quad - [\rho C_p FT] + [V(-\Delta H)r_{A \rightarrow B}] \\ &\quad - [hA(T - T_c)], \end{aligned}$$

where c_A is the concentration of A , $r_{A \rightarrow B}$ is the reaction rate, T is the temperature in the tank, ρ is fluid density, C_p is the specific heat, ΔH is the heat of reaction, h is the heat transfer coefficient to the cooling jacket, and A is the heat transfer area. For a first-order reaction,

$$r_{A \rightarrow B} = k c_A, \quad (34)$$

where k is the reaction rate constant. Typically k is adjusted for the temperature of the medium through the Arrhenius law

$$k = k_0 \exp\left(-\frac{E}{RT}\right), \quad (35)$$

where k_0 is the pre-exponential factor, E is the activation energy of the reaction, and R is the universal gas constant.

The CSTR model, consisting of two nonlinear ordinary differential equations, can now be defined in the analysis nomenclature of Section 2. The solution vector, x , is defined as

$$x = \begin{bmatrix} c_A \\ T \end{bmatrix}, \quad (36)$$

and the residual vector, f , is defined as

$$f = \begin{bmatrix} f_A \\ f_T \end{bmatrix}, \quad (37)$$

where

$$\begin{aligned} f_A &= V \frac{dc_A}{dt} - F(\lambda c_{A_f} + (1 - \lambda)c_A) \\ &\quad + F c_A - 0 + V r_{A \rightarrow B}, \\ f_T &= V \rho C_p \frac{dT}{dt} - \rho C_p F (\lambda T_f + (1 - \lambda)T) \\ &\quad + \rho C_p F T - V (-\Delta H) r_{A \rightarrow B} \\ &\quad + h A (T - T_c). \end{aligned} \quad (38)$$

The next step is to decompose Eq. (33) into a system of simple pieces. These simple pieces are called “evaluators”. In [35], the term “expression” was used instead of evaluator, however in the context of template-based generic programming, expression templates (an entirely different concept) are used quite often and so to avoid confusion between expressions and expression templates, the term evaluator will be used in this paper and is used in the Phalanx code base.

An evaluator is an atomic unit templated on an evaluation type that contains the following functionality:

- (1) Declares both (1) the fields it evaluates, called *evaluated fields* and (2) the fields it directly depends on to perform the evaluation, called *dependent fields*. A field is a quantity of interest that is computed and stored for use in the assembly process. For example, we might want to compute and store the reaction rate, $r_{A \rightarrow B}$, in Eq. (33). A discussion of fields can be found in Section 7.3.
- (2) Binds the fields required for the calculation. When fields are available for read/write, a method can be called on an evaluator to allow the evaluator to resolve memory for the fields that it writes and reads.
- (3) Calculates the values for the evaluated fields. When all dependent fields that an evaluator requires have been evaluated, an evaluator can compute the values of the evaluated fields using the values of the dependent fields.

By directly providing the functionality enumerated above, we can automatically construct algorithms (the properly-ordered sequence of steps to calculate all required quantities/expressions). This can be done by providing a few more abstractions:

- (1) A *registry* where all evaluators that may be required for the calculation are placed. This registry allows registration of evaluators and provides a way to obtain fully constructed fields via a *tag*. Tags will be discussed in Section 7.3. In the Phalanx library, the registry is called a Field Manager and is implemented in the `PHX::FieldManager` object.
- (2) A *memory allocator* used to allocate space to store field data. This will also be discussed in Section 7.3.
- (3) A mechanism to build the directed acyclic graph (DAG) representing the dependency among evaluators. Once the user defines one or more fields that are desired, the required evaluators can be resolved from the registry and form the “root” nodes in a graph. Each node may then be queried to determine its dependencies, which are placed on the out-edges of the node. This process is repeated recursively until the “bottom” of the graph is reached where there are no further out-edges.
- (4) A *scheduler* that traverses the graph to execute it. The execution graph may be obtained by inverting the dependency graph constructed in step 3.

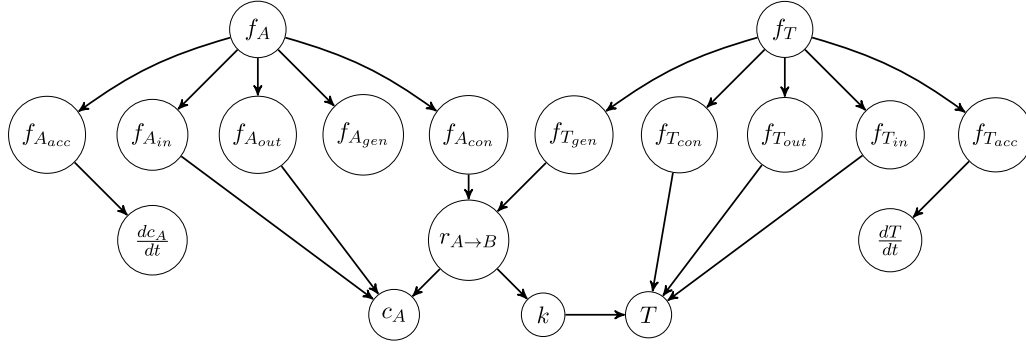


Fig. 2. Example expression dependency graph for the CSTR system (33).

A node is scheduled for execution once all of its data-dependents (execution-parents) have completed. This allows significant flexibility in the ordering in which nodes are executed, and facilitates task-based parallelism (see [35] for more details).

An example decomposition of Eq. (33) is shown in Fig. 2. This choice in decomposition follows naturally from separation of terms in the balance equation (32). The residuals are defined as

$$\begin{aligned}
 f &= \begin{bmatrix} f_A \\ f_T \end{bmatrix} \\
 &= \begin{bmatrix} f_{A_{acc}} - f_{A_{in}} + f_{A_{out}} - f_{A_{gen}} + f_{A_{cons}} \\ f_{T_{acc}} - f_{T_{in}} + f_{T_{out}} - f_{T_{gen}} + f_{T_{cons}} \end{bmatrix},
 \end{aligned} \tag{39}$$

where the evaluators for fields are defined as

$$\begin{aligned}
 f_{A_{acc}} &= V \frac{dc_A}{dt}, \\
 f_{A_{in}} &= F(\lambda c_{A_f} + (1 - \lambda)c_A), \\
 f_{A_{out}} &= Fc_A, \\
 f_{A_{gen}} &= 0, \\
 f_{A_{cons}} &= Vr_{A \rightarrow B}, \\
 f_{T_{acc}} &= V\rho C_p \frac{dT}{dt}, \\
 f_{T_{in}} &= \rho C_p F(\lambda T_f + (1 - \lambda)T), \\
 f_{T_{out}} &= \rho C_p FT, \\
 f_{T_{gen}} &= V(-\Delta H)r_{A \rightarrow B}, \\
 f_{T_{cons}} &= hA(T - T_c).
 \end{aligned} \tag{40}$$

Note that while the accumulation terms are discretized in time using BDF methods as described in Section 2.4, we recommend this calculation be performed in the high-level time integration package, not within the simulation code itself. Thus the time derivative terms (dc_A/dt and dT/dt) are treated as independent variables in the evaluation of the CSTR residual equations. Each node in this graph represents an evaluator and lists the field(s) that it evaluates. Out edges from a node point to other evaluators that evaluate fields that the origin node is directly dependent on. Note that the dependency graph nodes are evaluators and not fields. This is intentional so that evaluators can evaluate *multiple* fields. This is done to support the incorporation of third party libraries that, for efficiency, compute multiple fields (e.g., the source terms for the mass and heat balances) in a single function call.

Note that the decomposition in Fig. 2 is not unique but is chosen by the application developers. All equations could be implemented in a single model evaluator, or can be split across multiple evaluators as shown here. The advantages and disadvantages of the graph decomposition process with respect to template based generic programming are discussed in the next section. A separate discussion of general advantages irrespective of template based generic programming can be found in [35].

7.2.2. Benefits of graph-based assembly

The use of a graph-based approach has a number of benefits. First, the evaluation of terms will be consistent. If a field is dependent on another field, it will not be calculated until all of its dependent fields are first calculated. For a simple problem such as described by Eq. (33) it is not difficult. The logical order is clear and can be implemented by visual inspection. However as a code allows for multiple equations and closure models to be added, subtracted, or swapped, the complex-

ity makes visual inspection and ordering nearly impossible for all combinations and thus results in fragile code. This graph-based approach eliminates the logical complexity.

The use of fields and graph-based evaluation also provides a mechanism for code reuse. As different models are inserted into the graph and the dependency tree changes, the order is automatically updated and the surrounding nodes can be reused. Changing dependencies on one node does not require a code change in any of the surrounding nodes. For example, a simpler model of the reaction constant k can be used that does not depend on the temperature T . This can easily be accomplished by using a different evaluator for k without changing any other parts of the code.¹ Thus the use of evaluators isolates model code.

The dependency graph can generate very efficient code. Due to the complexity of ever changing equations and closure models, some applications ignore the complexity and resort to recomputing the same quantities over and over if needed in separate equation sets and/or operators, resulting in highly inefficient code. By using the dependency graph, a field is guaranteed to be computed once and used by all other evaluators that depend on it. For example, in Fig. 2, $r_{A \rightarrow B}$ is computed once and is used by both the A -consumption and T -generation evaluators. Additionally once $r_{A \rightarrow B}$ is used by all dependent evaluators, the scheduler could even recycle the memory for $r_{A \rightarrow B}$ to be used for other fields not yet processed in the DAG.

In terms of template-based generic programming, each evaluator is a separate C++ object templated on the evaluation type. As such, it not only becomes a point of variability to switch models, it can also be a point of variability in evaluation types by using template specialization on a single model. The user can implement different algorithms based on the evaluation type for each model. For example, in some cases having an exact Jacobian can hinder convergence rates. The `Jacobian` evaluation type could be specialized to drop certain dependencies from the AD scalar types.

The ability to apply template specialization to evaluators for a particular evaluation type is critical in the seed–compute–extract paradigm in Section 6. For each of the evaluation types, only the seed and extract eval-

¹Note however we assume the dependencies of any given evaluator are static throughout the computational simulation. Thus branching conditions are allowed within the implementation of any given evaluator as long as they do not change the structure of the graph. This latter case could be supported by allowing dynamic reconfiguring of the graph, however our current tools do not support this.

uators need be specialized. All other evaluators in the graph are used in the compute phase and have a *single implementation templated on the scalar type*.

7.3. Data structures and memory management

A critical aspect to efficient assembly algorithms is the management of data. This section is devoted to discussing a number of issues that impacted the design of Phalanx.

The allocation of memory for storing field data in the Phalanx library is completely controlled by the user. A templated parameter in the Traits class is used to specify a memory allocator object that can be written by the user. Once the dependency graph is constructed, the registry contains a list of all fields that the registry requires memory for. The registry can then call the allocator to dynamically allocate storage for the field data. An important aspect is that the allocator could allocate a single contiguous array for all field data members (including mixed data types) in an attempt to fit all field data for a complete graph traversal into cache. Phalanx contains two allocator implementations. A simple allocator that uses the C++ `new` command and an advanced allocator that allocates one single contiguous block of memory for all fields of all data types. The advanced allocator is templated on an alignment type so that the arrays can be correctly aligned for a base scalar type. When a block of the contiguous array is assigned to a field, a special reinterpret cast is called on the block to correctly call the scalar type constructors. This functionality along with the reference counting guarantees that the constructors and destructors of the field scalar type are called correctly. However for flexibility, users are free to specify their own allocator. This is useful, for example, when users have their own mesh database for storing field data.

The field interface is a simple handle layer. A field consists of a tag used for identification in the registry and a reference-counted smart pointer (RCP) [2,7] for accessing the field data. Two implementations of fields are available in Phalanx, a simple `PHX::Field` object and the more advanced multidimensional array `PHX::MDField` object. While the memory management relies on RCPs, in the future the array will optionally support out-of-core memory for running on GPGPUs using the multidimensional array under development in the Kokkos package [6].

The tag on a field is used to provide a unique signature for identifying fields in the registry. It is con-

structured from a name (a `std::string`), a scalar type, and a data layout. The scalar type information is returned in a `std::type_info` object to support runtime comparisons. The data layout object specifies the data signature for a multidimensional array. It contains the rank of the array, the names of the ordinals in the array, the sizes of the ordinals in the array, and an optional name. The unique signature is formed from a combination of these three objects so fields can be differentiated by name, scalar type, or data layout.

7.4. An example evaluator

Now that the data structures and algorithms have been discussed, an example evaluator for the node representing the reaction rate $r_{A \rightarrow B}$, will be shown. The header file, excluding the include guards and (optional) explicit template instantiation declarations, is shown in Fig. 3.

Three fields are declared. One for the reaction rate values, $r_{A \rightarrow B}$, and two for the dependent fields, k and c_A , required to perform the calculation in Eq. (34).

A set of macros can be used to hide the class declaration boilerplate. This substantially cuts down on the boilerplate and additionally hides most of the templating from users. The same header from Fig. 3 is shown in Fig. 4 using the macros.

The implementation of the three methods for this evaluator class follows. The constructor, the post registration setup and the evaluate methods are shown in Fig. 5 using the macro definitions. The constructor is used to create the tag for each field. This consists of a string name and the associated data layout. The data layout is passed in by the user through a parameter list. The data layout contains a description of the multidimensional array as described in Section 7.3. In the constructor, the evaluator is told how to advertise its capabilities. It declares what fields it evaluates and what fields it requires to complete the evaluation. The post registration setup function is used for binding the memory allocated for each field. The `field_manager` object is the field manager and contains the registry with pointers for the memory for each field.

```

#include "Phalanx_ConfigDefs.hpp"
#include "Phalanx_Evaluator_WithBaseImpl.hpp"
#include "Phalanx_Evaluator_Derived.hpp"
#include "Phalanx_MDField.hpp"

template<typename EvalT, typename Traits>
class FirstOrderReaction : public PHX::EvaluatorWithBaseImpl<Traits>,
public PHX::EvaluatorDerived<EvalT, Traits> {

public:
    FirstOrderReaction(const Teuchos::ParameterList& p);
    void postRegistrationSetup(typename Traits::SetupData d,
        PHX::FieldManager<Traits>& vm);
    void evaluateFields(typename Traits::EvalData d);

private:

    typedef typename EvalT::ScalarT ScalarT;

    PHX::MDField<ScalarT> k; // reaction rate constant
    PHX::MDField<ScalarT> c; // concentration
    PHX::MDField<ScalarT> r; // total reaction rate
}

```

Fig. 3. Header file source for implementing the reaction rate $r_{A \rightarrow B}$. Note that the include guards and (optional) explicit template instantiation machinery are not shown for clarity.

```

#include "Phalanx_ConfigDefs.hpp"
#include "Phalanx_Evaluator_WithBaseImpl.hpp"
#include "Phalanx_Evaluator_Derived.hpp"
#include "Phalanx_MDField.hpp"

PHX_EVALUATOR_CLASS(FirstOrderReaction)

    PHX::MDField<ScalarT> k; // reaction rate constant
    PHX::MDField<ScalarT> c; // concentration
    PHX::MDField<ScalarT> r; // total reaction rate

PHX_EVALUATOR_CLASS_END

```

Fig. 4. Header file source for implementing the reaction rate $r_{A \rightarrow B}$ using macro definition for class declaration boilerplate.

```

PHX_EVALUATOR_CTOR(FirstOrderReaction, params)
{
    Teuchos::RCP<PHX::DataLayout> scalar =
        params.get<Teuchos::RCP<PHX::DataLayout>>("Scalar Layout");

    k = PHX::MDField<ScalarT>("k", scalar);
    c = PHX::MDField<ScalarT>("c", scalar);
    r = PHX::MDField<ScalarT>("r", scalar);

    this->addDependentField(k);
    this->addDependentField(c);
    this->addEvaluatedField(r);
}

PHX_POST_REGISTRATION_SETUP(FirstOrderReaction, user_data, field_manager)
{
    this->utils.setFieldData(k, field_manager);
    this->utils.setFieldData(c, field_manager);
    this->utils.setFieldData(r, field_manager);
}

PHX_EVALUATE_FIELDS(FirstOrderReaction, user_data)
{
    r(0) = k(0) * c(0);
}

```

Fig. 5. Evaluator implementation.

The evaluate routine is the actual implementation of the algorithm. While this is a very simple expression, note that there is no limit on how much work a sin-

gle operator could do. For example, a single evaluator could implement multiple complete equations instead of this single term.

The `PHX::MDField` object is a multidimensional array that wraps the memory associated with storing the values of a field. In the case of our simple demonstration problem the fields are single scalar values (i.e., a rank one array of size one). This notation is slightly clunky for single scalars since one would prefer writing $r = k * c$ instead of $r(0) = k(0) * c(0)$. However, most use cases require MDFields that are larger arrays than single scalars (such as for finite element calculations). In such cases, multidimensional array support is crucial for performance. Note that one could recover the simple syntax by implementing expression templates for the MDField objects.

8. Numerical examples

We now provide simple numerical examples that demonstrate the power of this template-based generic programming approach. While these ideas can be applied to very complex multiphysics systems, for pedagogical purposes we consider the simple CSTR equations (33). Using the ideas discussed above, Phalanx evaluators can be written for each of the terms appearing in Eq. (33), along the lines of those shown in Figs 3–5. These evaluators, along with seed and extract evaluators for each evaluation type and interfaces connecting these calculations to high-level simulation and analysis algorithms in Trilinos, provide the software implementation necessary for the numerical examples shown below. We present these examples in terms of the non-dimensional version [50] of Eq. (33):

$$\begin{aligned} \frac{dx}{dt} &= -x + D(1-x) \exp\left(\frac{y}{1+y/\gamma}\right), \\ \frac{dy}{dt} &= -y + BD(1-x) \exp\left(\frac{y}{1+y/\gamma}\right) \\ &\quad - \beta(y - y_c). \end{aligned} \quad (41)$$

These equations exhibit a variety of interesting dynamical phenomena including multiple stable steady-states, stable and unstable oscillations, and bifurcations depending on the values of the parameters D , B , γ , β and y_c . As in [50], we consider the case $\gamma \rightarrow \infty$ and $y_c = 0$.

We first consider the locus of steady-state solutions as a function of D for fixed B and β . Such a curve is shown in Fig. 6(a), which was computed using the LOCA [49] and NOX [39] packages in Trilinos via pseudo-arclength continuation. This curve illuminates

values of D where multiple steady-states exist and the existence of turning-point bifurcations. The parameter values where these bifurcations occur can be solved for using the bifurcation equations (7), (8) and tracked as a function of a second parameter (also through pseudo-arclength continuation). Such a calculation using LOCA is displayed in Fig. 6(b). For both of these calculations, the template-based generic programming ideas discussed above are leveraged for analytic evaluation of the Jacobian and parameter derivatives needed by these algorithms as discussed in Section 2.3. Note that the implementation of these equations is capable of providing the analytic second derivatives needed by the turning point algorithm (9), and LOCA is capable of using them, however the interface used to connect this problem to LOCA currently does not support them and thus they were estimated by first-order finite differencing of the first derivatives.

For other choices of the model parameters, the system exhibits oscillations. An example of this is shown in Fig. 6(c) using the Rythmos time integration package [12] in Trilinos. In this case, analytic derivatives with respect to the transient terms are provided through the template-based generic programming approach and the Sacado automatic differentiation package.

As a final example, we consider the case when D , B and β are uncertain as represented by the following independent random variables:

$$\begin{aligned} D &\sim \text{uniform on } [0.03, 0.05], \\ B &\sim \text{uniform on } [7, 8], \\ \beta &\sim \text{uniform on } [0.05, 1.05]. \end{aligned} \quad (42)$$

We then estimate a fourth order polynomial chaos expansion of the steady-state solution to Eq. (41) using the intrusive stochastic Galerkin approach from Section 2.5. In this case the polynomials ψ are tensor products of one-dimensional Legendre polynomials. The stochastic Galerkin residual equations (17) and corresponding Jacobians (20) are evaluated using the specialized overloaded operators discussed above using the Sacado and Stokhos packages. These are then assembled into the stochastic Galerkin linear systems needed to solve the equations using Newton's method as provided by NOX [39]. Once the solution is computed, it can be employed for a variety of analysis purposes. For example, in Fig. 6(d) and (e) we plot an estimation of the probability density function for x and y , computed by Monte Carlo sampling of the polynomial chaos response surface $x(\xi)$, $y(\xi)$.

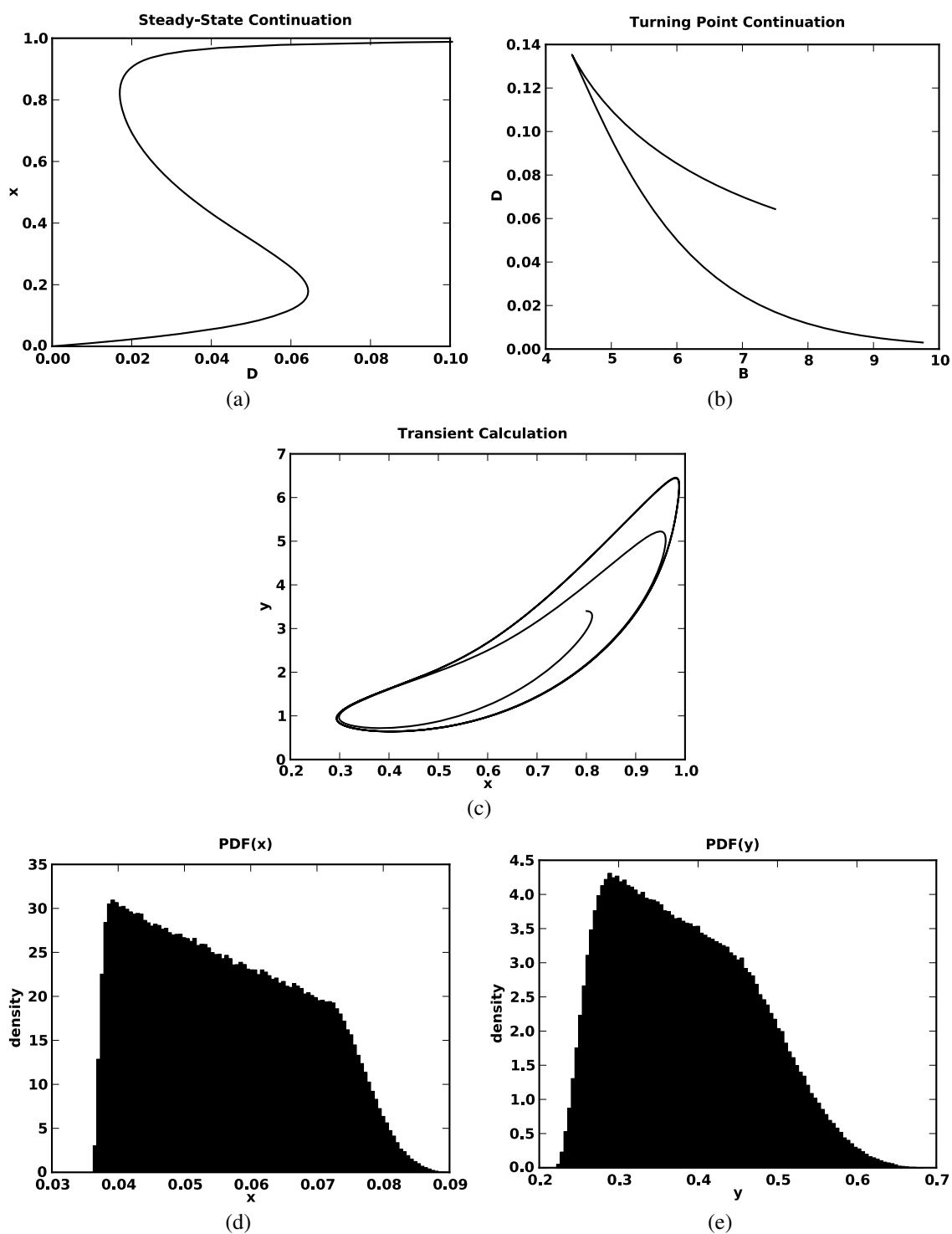


Fig. 6. Numerical examples of several analysis approaches applied to the CSTR equations. (a) Continuation. (b) Turning-point continuation. (c) Transient. (d) x density. (e) y density.

The purpose of these calculations is not to demonstrate any new numerical results or new algorithmic approaches, rather to demonstrate how a single templated code base can be leveraged for a variety of analysis algorithms that require advanced capabilities of the simulation code. While all of these calculations are very intrusive to the simulation code, the use of template-based generic programming for the most part makes these calculations transparent to the code developer and provides hooks for incorporation of future intrusive analysis algorithms.

9. Concluding remarks

In this paper we described a template-based generic programming approach for incorporating advanced analysis algorithms into complex simulation codes. This approach relies on operator overloading techniques in the C++ language inspired by automatic differentiation to transform a base residual calculation into one that can compute a variety of derivative and non-derivative quantities useful for advanced simulation and analysis. The approach also leverages the C++ template facilities to automate the conversion from the base scalar floating point type to the multitude of scalar types needed for operator overloading. Additionally we described a graph-based assembly approach for creating complex simulation tools, and how this approach builds on template-based generic programming. While these techniques are general and could be implemented in a variety of software tools, we described several tools within the Trilinos software framework that implement these ideas, namely the Sacado automatic differentiation, Stokhos stochastic Galerkin, and Phalanx graph-based assembly packages. Together with advanced solver and analysis tools in Trilinos such as NOX, LOCA and Rythmos, as well as other simulation tools such as geometry specification, mesh database and discretization tools also available in Trilinos, simulation tools for complex physical, biological and engineered processes can be developed that support a wide variety of state-of-the-art simulation and analysis algorithms. These tools and techniques have already been applied to more interesting and challenging problems than the simple example described here, which is the subject of a subsequent paper [40]. Together these papers demonstrate the most compelling aspect of the overarching template-based generic programming approach: while it is always possible to develop a customized software tool for a given

kind of simulation or analysis (e.g., transient, stability, or uncertainty analysis), the approach allows a *single* software implementation of the underlying equations to be leveraged for all of these analyses.

We admit that the approach described here is complex and requires a fair degree of sophistication on the part of the programmer in terms of familiarity with templating, polymorphism and operator overloading. However the beauty of the template-based generic programming approach is that it for the most part allows the programmer to ignore, and perhaps not even be aware of, the embedded analysis that ultimately might be performed on the simulation or the details of the specific scalar types and their corresponding overloaded operators.

Acknowledgements

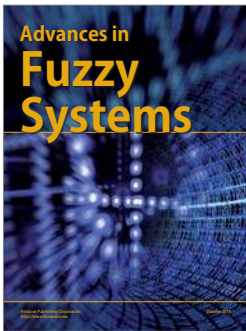
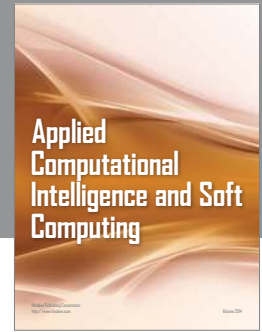
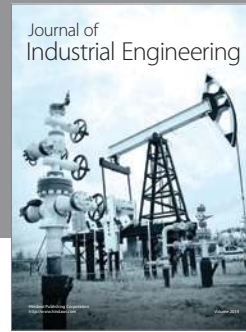
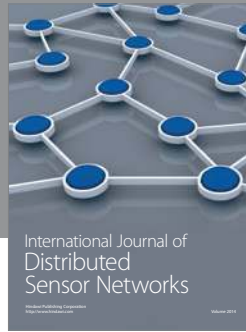
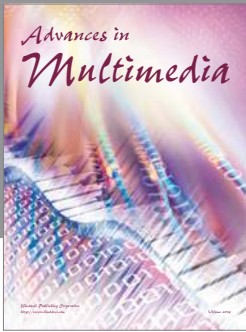
This work was funded by the US Department of Energy through the NNSA Advanced Scientific Computing and Office of Science Advanced Scientific Computing Research programs.

References

- [1] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley, Reading, MA, 2004.
- [2] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, C++ In-Depth Series, Addison-Wesley, Reading, MA, 2001.
- [3] M.S. Alnæs, UFL: a finite element form language, in: *Automated Solution of Differential Equations by the Finite Element Method*, A. Logg, K.-A. Mardal, G.N. Wells et al., eds, Lecture Notes in Computational Science and Engineering, Vol. 84, Springer, 2012, pp. 303–308 (Chapter 17).
- [4] P. Aubert and N. Di Césaré, Expression templates and forward mode automatic differentiation, in: *Automatic Differentiation of Algorithms: From Simulation to Optimization*, G. Corliss, C. Faure, A. Griewank, L. Hascoët and U. Naumann, eds, Computer and Information Science, Chapter 37, Springer, New York, NY, 2002, pp. 311–315.
- [5] P. Aubert, N. Di Césaré and O. Pironneau, Automatic differentiation in C++ using expression templates and application to a flow control problem, *Comput. Visual. Sci.* **3** (2001), 197–208.
- [6] C.G. Baker, M.A. Heroux and H.C. Edwards, Core kernels package, 2011, available at: <http://trilinos.sandia.gov/packages/kokkos/>.
- [7] R.A. Bartlett, Teuchos C++ memory management classes, idoms, and related topics: the complete reference (a comprehensive strategy for safe and efficient memory management in C++ for high performance computing), Technical Report SAND2010-2234, Sandia National Laboratories, May 2010.

- [8] R.A. Bartlett, D.M. Gay and E.T. Phipps, Automatic differentiation of C++ codes for large-scale scientific computing, in: *Computational Science – ICCS’2006*, V.N. Alexandrov, G.D. van Albada, P.M.A. Sloot and J. Dongarra, eds, Lecture Notes in Computer Science, Vol. 3994, Springer, Heidelberg, 2006, pp. 525–532.
- [9] C. Bendtsen and O. Stauning, FADBAD, a flexible C++ package for automatic differentiation, Technical Report IMM-REP-1996-17, Department of Mathematical Modelling, Technical Univ. Denmark, Lyngby, 1996.
- [10] C.H. Bischof, A. Carle, P. Khademi and A. Mauer, ADIFOR 2.0: automatic differentiation of Fortran 77 programs, *IEEE Comput. Sci. Eng.* **3**(3) (1996), 18–32.
- [11] C.H. Bischof, L. Roh and A. Mauer, ADIC – An extensible automatic differentiation tool for ANSI-C, *Software Pract. Exp.* **27**(12) (1997), 1427–1456.
- [12] T.S. Coffey and R.A. Bartlett, Rythmos time integration package, 2011, available at: <http://trilinos.sandia.gov/packages/rythmos/>.
- [13] B. Dawes and D. Abrahams, Boost C++ Libraries, 2011, available at: <http://www.boost.org>.
- [14] B. Debusschere, H. Najm, P. Pebay, O. Knio, R. Ghanem and O.L. Maitre, Numerical challenges in the use of polynomial chaos representations for stochastic processes, *SIAM J. Sci. Comput.* **26**(2) (2004), 698–719.
- [15] H. Elman, C. Miller, E.T. Phipps and R.S. Tuminaro, Assessment of collocation and Galerkin approaches to linear diffusion equations with random data, *Int. J. Uncertainty Quant.* **1**(1) (2011), 19–33.
- [16] D.M. Gay, Semiautomatic differentiation for efficient gradient computations, in: *Automatic Differentiation: Applications, Theory, and Tools*, H.M. Bücker, G. Corliss, P. Hovland, U. Naumann and B. Norris, eds, Lecture Notes in Computational Science and Engineering, Vol. 50, Springer, 2006, pp. 147–158.
- [17] R. Ghanem and P.D. Spanos, Polynomial chaos in stochastic finite elements, *J. Appl. Mech.* **57** (1990), 197–202.
- [18] R.G. Ghanem and P.D. Spanos, *Stochastic Finite Elements: A Spectral Approach*, Springer, New York, 1991.
- [19] W.J.F. Govaerts, *Numerical Methods for Bifurcations of Dynamical Equilibria*, SIAM, Philadelphia, PA, 2000.
- [20] T. Granlund, GNU multiple precision library, 2011, available at: <http://gmplib.org/>.
- [21] A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Frontiers in Appl. Math., Vol. 19, SIAM, Philadelphia, PA, 2000.
- [22] A. Griewank, D. Juedes and J. Utke, Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++, *ACM Trans. Math. Softw.* **22**(2) (1996), 131–167.
- [23] J. Guckenheimer and P. Holmes, *Nonlinear Oscillations, Dynamical Systems, and Bifurcations of Vector Fields*, Applied Mathematical Sciences, Vol. 42, Springer, New York, 1983.
- [24] E. Hairer, S. Nørsett and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*, Springer Series in Computational Mathematics, Vol. 8, Springer, Berlin, 1991.
- [25] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, Springer Series in Computational Mathematics, Vol. 14, Springer, Berlin, 1991.
- [26] L. Hascoët and V. Pascual, TAPENADE 2.1 user’s guide, Rapport technique 300, INRIA, Sophia Antipolis, 2004, available at: <http://www.inria.fr/rrrt/rt-0300.html>.
- [27] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams and K. Stanley, An overview of the Trilinos package, *ACM Trans. Math. Softw.* **31**(3) (2005), 397–423.
- [28] V.E. Howle, R.C. Kirby, K.R. Long, B. Brennan and K. Kennedy, Playa: High-performance programmable linear algebra, *Sci. Program.* **20** (2012), 257–273.
- [29] K. Iglberger, G. Hager, J. Treibig and U. Rude, Expression templates revisited: a performance analysis of current methodologies, *SIAM J. Sci. Comput.* **34**(2) (2012), C42–C69.
- [30] H. Keller, Constructive methods for bifurcation and nonlinear eigenvalue problems, in: *Computing Methods in Applied Sciences and Engineering, 1977, I*, R. Glowinski, J. Lions and I. Laboria, eds, Lecture Notes in Math., Vol. 704, Springer, Berlin/Heidelberg, 1979, pp. 241–251.
- [31] R.C. Kirby, A new look at expression templates for matrix computation, *Comput. Sci. Eng.* **5**(3) (2003), 66–70.
- [32] A. Logg, Automating the finite element method, *Arch. Comput. Methods Eng.* **14**(2) (2007), 93–138.
- [33] A. Logg, K.-A. Mardal, G.N. Wells et al., *Automated Solution of Differential Equations by the Finite Element Method*, Springer, Berlin, 2012.
- [34] K. Long, R. Kirby and B. van Bloemen Waanders, Unified embedded parallel finite element computations via software-based Frechet differentiation, *SIAM J. Sci. Comput.* **32**(6) (2010), 3323–3351.
- [35] P.K. Notz, R.P. Pawlowski and J.C. Sutherland, Graph-based software design for managing complexity and enabling concurrency in multiphysics PDE software, *ACM Trans. Math. Softw.* **39** (2012), to appear.
- [36] E. Novak and K. Ritter, High dimensional integration of smooth functions over cubes, *Num. Math.* **75** (1996), 59–97.
- [37] C. Prud’homme, Life: overview of a unified C++ implementation of the finite and spectral element methods in 1D, 2D and 3D, in: *Applied Parallel Computing. State of the Art in Scientific Computing*, B. Kågström, E. Elmroth, J. Dongarra and J. Wasniewski, eds, Lecture Notes in Computer Science, Vol. 4699, Springer, Berlin, 2007, pp. 712–721.
- [38] R.P. Pawlowski, Phalanx multiphysics assembly package, 2010, available at: <http://trilinos.sandia.gov/packages/phalanx/>.
- [39] R.P. Pawlowski and T.G. Kolda, NOX object-oriented nonlinear solver package, 2011, available at: <http://trilinos.sandia.gov/packages/nox/>.
- [40] R.P. Pawlowski, E.T. Phipps, A.G. Salinger, S.J. Owen, C. Siefert and M.L. Staten, Automating embedded analysis capabilities and managing software complexity in multiphysics simulation, Part II: Application to partial differential equations, *Sci. Program.* **20** (2012), 327–345.
- [41] E. Phipps and R. Pawlowski, Efficient expression templates for operator overloading-based automatic differentiation, in: *Recent Advances in Algorithmic Differentiation*, S. Forth, P. Hovland, E. Phipps, J. Utke and A. Walther, eds, Lecture Notes in Computational Science and Engineering, Vol. 87, Springer, Berlin, 2012, pp. 309–319.

- [42] E.T. Phipps, Stokhos stochastic Galerkin uncertainty quantification methods, 2011, available at: <http://trilinos.sandia.gov/packages/stokhos/>.
- [43] E.T. Phipps, R.A. Bartlett, D.M. Gay and R.J. Hoekstra, Large-scale transient sensitivity analysis of a radiation-damaged bipolar junction transistor via automatic differentiation, in: *Advances in Automatic Differentiation*, C.H. Bischof, H.M. Bücker, P.D. Hovland, U. Naumann and J. Utke, eds, Springer, 2008, pp. 351–362.
- [44] E.T. Phipps and D.M. Gay, Sacado automatic differentiation package, 2011, available at: <http://trilinos.sandia.gov/packages/sacado/>.
- [45] C. Prud'Homme, A domain specific embedded language in C++ for automatic differentiation, projection, integration and variational formulations, *Sci. Program.* **14**(2) (2006), 81–110.
- [46] M. Reagan, H. Najm, R. Ghanem and O. Knio, Uncertainty quantification in reacting-flow simulations through non-intrusive spectral projection, *Combust. Flame* **132**(3) (2003), 545–555.
- [47] A. Salinger, E. Burroughs, R. Pawlowski, E. Phipps and L. Romero, Bifurcation tracking algorithms and software for large scale applications, *Int. J. Bifurcat. Chaos* **15**(3) (2005), 1015–1032.
- [48] A.G. Salinger, E.T. Phipps, R.A. Bartlett, R.P. Pawlowski and T.S. Coffey, Piro embedded analysis tools package, 2011, available at: <http://trilinos.sandia.gov/packages/piro/>.
- [49] A.G. Salinger, E.T. Phipps and R.P. Pawlowski, LOCA: library of continuation algorithms, 2011, available at: <http://trilinos.sandia.gov/packages/nox/>.
- [50] A. Uppal and W.H. Ray, On the dynamic behavior of continuous stirred tank reactors, *Chem. Eng. Sci.* **29** (1974), 967–985.
- [51] J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill and C. Wunsch, OpenAD/F: a modular, open-source tool for automatic differentiation of Fortran codes, *ACM Trans. Math. Softw.* **34**(4) (2008), 18:1–18:36.
- [52] D. Vandevoorde and N.M. Josuttis, *C++ Templates, The Complete Guide*, Addison-Wesley, Reading, MA, 2003.
- [53] T. Veldhuizen, Expression templates, *C++ Report* **7**(5) (1995), 26–31.
- [54] T. Veldhuizen, Arrays in blitz++, in: *Computing in Object-Oriented Parallel Environments*, D. Caromel, R. Oldehoeft and M. Tholburn, eds, Lecture Notes in Computer Science, Vol. 1505, Springer, Berlin, 1998, p. 501.
- [55] M. Voßbeck, R. Giering and T. Kaminski, Development and first applications of TAC++, in: *Advances in Automatic Differentiation*, C.H. Bischof, H.M. Bücker, P.D. Hovland, U. Naumann and J. Utke, eds, Springer, Berlin, 2008, pp. 187–197.
- [56] N. Wiener, The homogeneous chaos, *Am. J. Math.* **60** (1938), 897–936.
- [57] D. Xiu and G. Karniadakis, The Wiener–Askey polynomial chaos for stochastic differential equations, *SIAM J. Sci. Comput.* **24**(2) (2002), 619–644.
- [58] D.P. Young, W.P. Huffman, R.G. Melvin, C.L. Hilmes and F.T. Johnson, Nonlinear elimination in aerodynamic analysis and design, in: *Large-Scale PDE-Constrained Optimization*, L.T. Biegler, O. Ghattas, M. Heinkenschloss and B. van Bloemen Waanders, eds, Lecture Notes in Computational Science and Engineering, Vol. 30, Springer, Berlin, 2003, pp. 17–43.



Hindawi

Submit your manuscripts at
<http://www.hindawi.com>

