

UNIFIED EMBEDDED PARALLEL FINITE ELEMENT COMPUTATIONS VIA SOFTWARE-BASED FRÉCHET DIFFERENTIATION*

KEVIN LONG[†], ROBERT KIRBY[†], AND BART VAN BLOEMEN WAANDERS[‡]

Abstract. Computational analysis of systems governed by partial differential equations (PDEs) requires not only the calculation of a solution but the extraction of additional information such as the sensitivity of that solution with respect to input parameters or the inversion of the system in an optimization or design loop. Moving beyond the automation of discretization of PDEs by finite element methods, we present a mathematical framework that unifies the discretization of PDEs with these additional analysis requirements. In particular, Fréchet differentiation on a class of functionals together with a high-performance finite element framework has led to a code, called Sundance, that provides high-level programming abstractions for the automatic, efficient evaluation of finite variational forms together with the derived operators required by engineering analysis.

Key words. finite element method, partial differential equations, embedded algorithms

AMS subject classifications. 15A15, 15A09, 15A23

DOI. 10.1137/09076920X

1. Introduction. Advanced simulation of realistic systems governed by partial differential equations (PDEs) can require a significant collection of operators beyond evaluating the residual of the nonlinear algebraic equations for the system solution. As a first example, Newton’s method requires not only the residual evaluation but also the formation or application of the Jacobian matrix. Beyond this, sensitivity analysis, optimization, and control require even more operators that go beyond what is implemented in standard simulation codes. We describe algorithms requiring additional operators beyond a black-box residual evaluation or system matrix as *embedded*. In this paper we concentrate on producing operators for embedded linearization, optimization, and sensitivity analysis, but we point out that the issue of producing auxiliary operators also arises in the contexts of physics-based preconditioning and embedded uncertainty quantification.

Traditional automatic differentiation (AD) tools [16] bridge some of the gap between what is implemented and what modern embedded algorithms require. For example, AD is very effective at constructing code for Jacobian evaluation from code for residual evaluation and finding adjoints or derivatives needed for sensitivity. However, AD tools can only construct operators that are themselves derivatives of operators already implemented in an existing code.

*Received by the editors August 26, 2009; accepted for publication (in revised form) July 21, 2010; published electronically November 17, 2010. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. Copyright is owned by SIAM to the extent not limited by these rights.

<http://www.siam.org/journals/sisc/32-6/76920.html>

[†]Department of Mathematics and Statistics, Texas Tech University, Lubbock, TX 79409 (kevin.long@ttu.edu, robert.c.kirby@ttu.edu). The work of the first and second authors was supported by NSF award 0830655.

[‡]Applied Mathematics and Applications, Sandia National Laboratories, P.O. Box 5800, Albuquerque, NM 87122 (bartv@sandia.gov). Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin company, for the U.S. Department of Energy under contract DE-AC04-94AL85000.

Further, implementing these operators efficiently and correctly typically presents its own difficulties. While the necessary code is typically compact, it requires the programmer to hold together knowledge about meshes, basis functions, numerical integration, and many other techniques. Current research projects aim to simplify this process. Some of these, such as the widely used Deal.II library [5], provide infrastructure for handling meshes, basis functions, assembly, and interfaces to linear solvers. Other projects, such as Analyza [4] and FFC [24, 25] use a high-level input syntax to generate low-level code for assembling variational forms. Yet other projects, such as Life [35] and FreeFEM [19], provide domain-specific language for finite element computation, either by providing a grammar and interpreter for a new language (FreeFEM) or by extending an existing language with library support for variational forms (Life). We also include in this category more recent FEniCS projects such as UFL [1], which is a language specification for finite element methods, and UFC [2], a compiler for UFL similar to `ffc`.

Our present work, encoded in the open-source project Sundance [43, 30, 31], unifies these two perspectives of *differentiation* and *automation* by developing a theory in which formulae for even simple forward operators such as stiffness matrices are obtained through run-time Fréchet differentiation of variational forms. Like many finite element projects described above, our work also provides interfaces to meshes, basis functions, and solvers, but our formalism for obtaining algebraic operators via differentiation appears to be new in the literature. While, mathematically, our version of AD is similar to that used in [16], we differentiate at a more abstract level on representations of functionals to obtain low-level operations rather than writing those low-level operations by traditional means and then differentiating. Also, we require differentiation with respect to variables that themselves may be (derivatives of) functions and rules that can distinguish between spatially variable and constant expressions. These techniques are typically not included in AD packages. While this complicates some of our differentiation rules, it provides a mechanism for automating the evaluation of variational forms. Sundance is a C++ library for symbolically representing, manipulating, and evaluating variational forms, together with necessary lower-level finite element tools.

Sundance is open-source code, freely available as part of the Trilinos suite of mathematical software [20, 21].

Automated evaluation of finite element operators by Sundance or other codes provides a smooth transition from problem specification to production-quality simulators, bypassing the need for intermediate stages of prototyping and optimizing code. When all variational forms of a general class are efficiently evaluated, each form is not implemented, debugged, and optimized as a special case. This increases code correctness and reliability once the internal engine is implemented. Generation and optimization of algorithms from an abstract specification is receiving considerable attention in several areas of numerical computing. Most work in this area has concentrated on lower-level computational kernels. For examples, the Smart project of Püschel and coauthors [12, 13, 36, 37, 38] algebraically finds fast signal processing algorithms and is attached to a domain-specific compiler for these kinds of algorithms. In numerical linear algebra, the Flame project led by van de Geijn (see [6, 17]) demonstrates how correct, high-performance implementations of matrix computations may be derived by formal methods. Like these projects, Sundance uses inherent, domain-specific mathematical structure to automate numerical calculations.

In this paper, we present our mathematical framework for differentiation of variational forms, survey our efficient software implementation of these techniques, and

present examples indicating some of the code's capabilities. Section 2 provides a unifying mathematical presentation of forward simulation, sensitivity analysis, eigenvalue computation, and optimization from our perspective of differentiation. Section 3 provides an overview of the Sundance software architecture and evaluation engine, including some indications of how we minimize the overhead of interpreting variational forms at run-time. We illustrate Sundance's capabilities with a series of examples in sections 2.2.1, 2.3.1, 2.4.1, and 4 and then present some concluding thoughts and directions for future development in section 5. Many of our examples include code segments as needed; a complete code listing for an advection-diffusion equation is included as an appendix.

2. A unified approach to multiple problem types through functional differentiation.

2.1. Functional differentiation as the bridge from symbolic to discrete.

We consider PDEs on a d -dimensional spatial domain Ω . We will use lowercase italic symbols such as u, v for functions mapping $\Omega \rightarrow \mathbb{R}$. We will denote arbitrary function spaces with uppercase italic letters such as U, V . Operators act on functions to produce new functions. We denote operators by calligraphic characters such as \mathcal{F}, \mathcal{G} . Finally, a functional maps one or more functions to \mathbb{R} ; we denote functionals by uppercase letters such as F, G and put arguments to functionals in square brackets. As usual, x, y , and z represent spatial coordinates.

Note that the meaning of a symbol such as $\mathcal{F}(u)$ is somewhat ambiguous. It can stand both for the operator \mathcal{F} acting on the function u , and for the function $\mathcal{F}(u(x))$ that is the result of this operation. This is no different from the familiar useful ambiguity in writing $f(x) = e^x$, where we often switch at will between referring to the *operation* of exponentiation of a real number and the *value* of the exponential of x .

To differentiate operators and functionals with respect to functions, we use the Fréchet derivative $\frac{\partial \mathcal{F}}{\partial u}$ (see, e.g., [10]) defined implicitly through

$$\lim_{\|h\| \rightarrow 0} \frac{\|\mathcal{F}(u+h) - \mathcal{F}(u) - \frac{\partial \mathcal{F}}{\partial u} h\|}{\|h\|} = 0.$$

The Fréchet derivative of an operator is itself an operator, and thus as per the preceding paragraph, when acting on a function, the symbol $\frac{\partial \mathcal{F}}{\partial u}$ can also be considered a function. In the context of a PDE we will encounter operators that may depend not only on a function u but on its spatial derivatives such as $D_x u$. As is often done in elementary presentations of the calculus of variations (e.g., [42]), we will find it useful to imagine u and $D_x u$ as distinct variables, and write $\mathcal{F}(u, D_x u)$ for an operator involving derivatives. Differentiating with respect to a variable that is itself a derivative of a field variable is a notational device commonly used in Lagrangian mechanics (e.g., [3, 39]) and field theory (e.g., [7, 8]), and we will use it throughout this paper. This device can be justified rigorously via the Fréchet derivative.

We now consider some space U of real-valued functions over Ω . For simplicity of presentation we assume for the moment that all differentiability and integrability conditions that may arise will be met. Introduce a discrete N -dimensional subspace $U^h \subset U$ spanned by a basis $\{\phi_i(x)\}_{i=1}^N$, and let $u \in U^h$ be expanded as $u(x) = \sum_{i=1}^N u_i \phi_i(x)$, where $\{u_i\} \subset \mathbb{R}^N$ is a vector of coefficients. When we encounter a functional $F[u] = \int \mathcal{F}(u) d\Omega$, we can ask for the derivative of F with respect to each

expansion coefficient u_i . Formal application of the chain rule gives

$$(2.1) \quad \frac{\partial F}{\partial u_i} = \int \frac{\partial \mathcal{F}}{\partial u} \frac{\partial u}{\partial u_i} d\Omega$$

$$(2.2) \quad = \int \frac{\partial \mathcal{F}}{\partial u} \phi_i(x) d\Omega,$$

where $\frac{\partial \mathcal{F}}{\partial u}$ is a Fréchet derivative.

The derivative of a functional involving u and $D_x u$ with respect to an expansion coefficient is

$$(2.3) \quad \frac{\partial F}{\partial u_i} = \int \frac{\partial \mathcal{F}}{\partial u} \phi_i(x) d\Omega + \int \frac{\partial \mathcal{F}}{\partial (D_x u)} D_x \phi_i(x) d\Omega.$$

Equation (2.3) contains three distinct kinds of mathematical objects, each of which plays a specific role in the structure of a simulation code.

1. $\frac{\partial F}{\partial u_i}$, which is a vector in \mathbb{R}^N . This discrete object is typical of the sort of information to be produced by a simulator's discretization engine for use in a solver or optimizer routine.
2. $\frac{\partial \mathcal{F}}{\partial u}$ and $\frac{\partial \mathcal{F}}{\partial (D_x u)}$, which are Fréchet derivatives acting on an operator \mathcal{F} . The operator \mathcal{F} is a symbolic object, containing by itself no information about the finite-dimensional subspace on which the problem will be discretized. Its derivatives are likewise symbolic objects.
3. Terms such as ϕ_i and $D_x \phi_i$, which are spatial derivatives of a basis function.

Equation (2.3) is the bridge leading from a symbolic specification of a problem as a symbolic operator \mathcal{F} to a discrete vector for use in a solver or optimizer algorithm. The central ideas in this paper are that (1) the discretization of many apparently disparate problem types can be represented in a unified way through functional differentiation as in (2.3), and (2), that this ubiquitous mathematical structure provides a path for connecting high-level symbolic problem representations to high-performance low-level discretization components.

2.2. Illustration in a scalar forward nonlinear PDE. The weak form of a scalar PDE for $u \in V$ in d spatial dimensions will be the requirement that a functional of two arguments

$$(2.4) \quad G[u, v] = \sum_r \int_{\Omega_r} \mathcal{G}_r(\{D_\alpha v\}_\alpha, \{D_\beta u\}_\beta, x) d\mu_r$$

be zero for all v in some subspace \hat{V} . The operators \mathcal{G}_r are homogeneous linear functions of v and its derivatives, but can be arbitrary nonlinear functions of u , its derivatives, and the independent spatial variable $x \in \mathbb{R}^d$. We use the notation $D_\alpha f$ to indicate partial differentiation of f with respect to the combination of spatial variables indicated by the multi-index α . When we use a set $\{D_\alpha u\}_\alpha$ as the argument to \mathcal{G}_r , we mean that \mathcal{G}_r may depend on any one or more members of the set of partial spatial derivatives of u . The summation is over geometric subregions Ω_r , which may include lower-dimension subsets such as portions of the boundary. The integrand \mathcal{G}_r may take different functional forms on different subregions; for example, it will usually have different functional forms on the boundary and on the interior. Finally, note that we may use different measures $d\mu_r$ on different subdomains; this allows, for instance, the common practice of enforcing Dirichlet boundary conditions by fixing values at nodes.

As usual we discretize u on a finite-dimensional subspace V^h and also consider only a finite-dimensional space \hat{V}^h of test functions; we then expand u and v as a linear combination of basis vectors $\phi \in V^h$ and $\psi \in \hat{V}^h$,

$$(2.5) \quad u = \sum_{j=1}^N u_j \phi_j(x),$$

$$(2.6) \quad v = \sum_{i=1}^N v_i \psi_i(x).$$

The requirement that (2.4) hold for all $v \in V$ is met by ensuring that it holds for each of the basis vectors ψ_i . Because each G has been defined as a homogeneous linear functional in v , this condition is met if and only if

$$(2.7) \quad \frac{\partial G}{\partial v_i} = \sum_r \sum_\alpha \int_{\Omega_r} \frac{\partial \mathcal{G}_r}{\partial (D_\alpha v)} D_\alpha \psi_i d\mu_r = 0.$$

Repeating this process for $i = 1$ to N gives N (generally nonlinear) equations in the N unknowns u_j . We now linearize (2.7) with respect to u about some $u^{(0)}$ to obtain a system of linear equations for the full Newton step δu ,

$$(2.8) \quad \frac{\partial G}{\partial v_i} + \frac{\partial^2 G}{\partial v_i \partial u_j} \Big|_{u_j^{(0)}} \delta u_j = 0.$$

In the case of a linear PDE (or one that has already been linearized with an alternative formulation, such as the Oseen approximation to the Navier–Stokes equations [14]), the “linearization” would be done about $u^{(0)} = 0$, and δu is then the solution of the PDE.

Writing the above equation out in full, we have

$$(2.9) \quad \left[\sum_r \sum_\alpha \int_{\Omega_r} \frac{\partial \mathcal{G}_r}{\partial (D_\alpha v)} D_\alpha \psi_i d\mu_r \right] + \sum_j \delta u_j \left[\sum_r \sum_\alpha \sum_\beta \int_{\Omega_r} \frac{\partial^2 \mathcal{G}_r}{\partial (D_\alpha v) \partial (D_\beta u)} D_\alpha \psi_i D_\beta \phi_j d\mu_r \right] = 0.$$

The two bracketed quantities are the load vector f_i and stiffness matrix K_{ij} , respectively.

With this approach, we can compute a stiffness matrix and load vector by quadrature provided that we have computed the first and second order Fréchet derivatives of \mathcal{G}_r . Were we free to expand \mathcal{G}_r algebraically, it would be simple to compute these Fréchet derivatives symbolically, and we could then evaluate the resulting symbolic expressions on quadrature points. We have devised an algorithm and associated data structure that will let us compute these Fréchet derivatives in place, with neither symbolic expansion of operators nor code generation, saving us the combinatorial explosion of expanding \mathcal{G}_r and the overhead and complexity of code generation. The relationship between our approach and code generation is discussed further in section 3.

It should be clear that generalization beyond scalar problems to vector-valued and complex-valued problems, perhaps with mixed discretizations, is immediate.

2.2.1. Example: Galerkin discretization of Burgers' equation. As a concrete example, we show how a Galerkin discretization of Burgers' equation appears in the formulation above. Consider the steady-state Burgers' equation on the one-dimensional domain $\Omega = [0, 1]$,

$$(2.10) \quad uD_x u = cD_{xx}u.$$

We will ignore boundary conditions for the present discussion; in the next section we explain how boundary conditions fit into our framework. The Galerkin weak form of this equation is

$$(2.11) \quad \int_0^1 [vuD_x u + cD_x v D_x u] dx = 0 \quad \forall v \in H_\Omega^1.$$

To cast this into the notation of (2.4), we define

$$(2.12) \quad \mathcal{G} = vuD_x u + cD_x v D_x u.$$

The nonzero derivatives appearing in the linearized weak Burgers' equation are shown in Table 1. The table makes clear the correspondence between differentiation variables and basis combination and between derivative value and coefficient in the linearized, discretized weak form.

TABLE 1

This table shows for the Burgers' equation example the correspondence, defined by (2.12), between functional derivatives, coefficients in weak forms, and basis function combinations in weak forms. Each row shows a particular functional derivative, its compact representation as a multiset, the value of the derivative, the combination of basis function derivatives extracted via the chain rule, and the resulting term in the linearized, discretized weak form.

Derivative	Multiset	Value	Basis combination	Integral
$\frac{\partial \mathcal{G}}{\partial v}$	$\{v\}$	$uD_x u$	ϕ_i	$\int uD_x u \phi_i$
$\frac{\partial \mathcal{G}}{\partial D_x v}$	$\{D_x v\}$	$cD_x u$	$D_x \phi_i$	$\int cD_x u D_x \phi_i$
$\frac{\partial^2 \mathcal{G}}{\partial v \partial u}$	$\{v, u\}$	$D_x u$	$\phi_i \phi_j$	$\int D_x u \phi_i \phi_j$
$\frac{\partial^2 \mathcal{G}}{\partial v \partial D_x u}$	$\{v, D_x u\}$	u	$\phi_i D_x \phi_j$	$\int u \phi_i D_x \phi_j$
$\frac{\partial^2 \mathcal{G}}{\partial D_x v \partial D_x u}$	$\{D_x v, D_x u\}$	c	$D_x \phi_i D_x \phi_j$	$\int cD_x \phi_i D_x \phi_j$

The user-level Sundance code to represent the weak form on the interior is

```
Expr eqn = Integral(interior, c*(dx*u)*(dx*v) + v*u*(dx*u), quad);,
```

where `interior` and `quad` specify the domain of integration and the quadrature scheme to be used, and the other variables are symbolic expressions. Example results are shown in section 2.3.1 in the context of sensitivity analysis.

2.2.2. Representation of Dirichlet boundary conditions. Dirichlet boundary conditions can fit into our framework in several ways, including the symmetrized formulation of Nitsche [32] and a simple generalization of the traditional row-replacement method. The Nitsche method augments the original weak form in a

manner that preserves consistency, coercivity, and symmetry; as far as software is concerned, the additional terms require no special treatment and need not be discussed further in this context.

The most widely used method for imposing Dirichlet boundary conditions is to replace rows in the discrete linear system by equations that force the solution to take on specified boundary values. This is usually done by postprocessing the assembled matrix and vector. Because this postprocessing is done to the discretized system, it has sometimes been said that a symbolic system such as ours cannot be used to specify boundary conditions with the row-replacement method. We will therefore discuss why it is desirable to represent Dirichlet boundary conditions through symbolic expressions and then explain how such a representation can be used to specify a row-replacement method.

First, note that in the context of embedded algorithms it would not be suitable to use a “hybrid” approach in which we specify internal equations with symbolic expressions, discretize them as described above, and then apply Dirichlet boundary conditions by postprocessing the discrete equations according to some other (nonsymbolic) specification. For example, in optimization problems in which design variables appear in the boundary conditions, it is necessary to differentiate the boundary conditions with respect to these design variables. It is therefore desirable to represent the boundary conditions using our differentiation-based formulation.

Fortunately, that representation is rather simple: the only modification to the system described above is to “tag” certain expressions as replacement-style boundary conditions. These expressions are then discretized as any other, but their tagged status then indicates that upon assembly, their row data replaces that produced by any untagged (i.e., nonboundary condition) expressions. The content of the boundary condition rows is produced through our differentiation-based discretization system, just like any other row, enabling differentiation as needed for embedded algorithms. The boundary row “tag” then directs replacement.

Thus, both the row replacement method and Nitsche’s method are available in our formulation. With either method, differentiation is enabled throughout, allowing the use of embedded algorithms on problems with design variables on boundaries.

2.3. Sensitivity analysis. In sensitivity analysis, we seek the derivatives of a field u with respect to a parameter p . When u is determined through a forward problem of the form (2.4), we do implicit differentiation to find

$$(2.13) \quad \sum_r \sum_\beta \int_{\Omega_r} \left[\frac{\partial \mathcal{G}_r}{\partial D_\beta u} D_\beta \left(\frac{\partial u}{\partial p} \right) + \frac{\partial \mathcal{G}_r}{\partial p} \right] d\mu_r = 0 \quad \forall v \in \hat{V}.$$

Differentiating by v_i to obtain discrete equations gives

$$(2.14) \quad \sum_r \sum_\alpha \left[\int_{\Omega_r} \frac{\partial^2 \mathcal{G}_r}{\partial D_\alpha v \partial p} D_\alpha \psi_i d\mu_r \right] + \sum_j \frac{\partial u_j}{\partial p} \left[\sum_r \sum_\alpha \sum_\beta \int_{\Omega_r} \frac{\partial^2 \mathcal{G}_r}{\partial D_\alpha v \partial D_\beta u} D_\alpha \psi_i D_\beta \phi_j d\mu_r \right] = 0.$$

This has the same general structure as the discrete equation for a Newton step; the only change has been in the differentiation variables. Thus, the mathematical framework and software infrastructure outlined above are immediately capable of performing sensitivity analysis given a high-level forward problem specification.

2.3.1. Example: Sensitivity analysis of Burgers' equation. We now show how this automated production of weak sensitivity equations works in the context of the one-dimensional Burgers' equation example from section 2.2.1.

To produce an easily solvable parametrized problem, we apply the method of manufactured solutions [40, 41] to construct a forcing term that produces a convenient, specified solution. We define a function

$$f(p, x) = p (px (2x^2 - 3x + 1) + 2),$$

where p is a design parameter. With this function as a forcing term in the steady-state Burgers' equation,

$$uu_x = u_{xx} + f(p, x), \quad u(0) = u(1) = 0,$$

we find that the solution is $u(x) = px(1 - x)$. The sensitivity $\frac{\partial u}{\partial p}$ is $x(1 - x)$.

Now consider the sensitivity of solutions to (2.11) to the parameter p . The Fréchet derivatives appearing in the second set of brackets in (2.14) are identical to those computed for the linearized forward problem; as is well known, the matrix in a sensitivity problem is identical to the problem's Jacobian matrix. The derivatives in the first set of brackets are summarized in Table 2. Notice that in this example, the derivative $\{D_x v, p\}$ is identically zero; this fact can be identified in a symbolic preprocessing step so that it is ignored in all numerical calculations.

TABLE 2

This table shows the terms in the first brackets of (2.14) that arise in the Burgers' sensitivity example described in the text.

Derivative	Multiset	Value	Basis combination	Integral
$\frac{\partial^2 g}{\partial v \partial p}$	$\{v, p\}$	$\frac{\partial f}{\partial p}$	ϕ_i	$\int \frac{\partial f}{\partial p} \phi_i$
$\frac{\partial^2 g}{\partial D_x v \partial p}$	$\{D_x v, p\}$	0	$D_x \phi_i$	0

Next we show the user-level C++ code for setting up this problem using objects from the Sundance toolkit. Initialization code and construction of some objects (such as `mesh`) are omitted, but a brief description of the object types is in order. Most Sundance objects are reference-counted handles, so we have transparent memory management and polymorphism at the user level without need for explicit use of pointers. Class `Expr` is a polymorphic handle for symbolic expressions. Objects `u` and `v` are expressions of type `UnknownFunction` and `TestFunction`, respectively. The basis for these expressions has been specified at construction. Objects `x` and `dx` represent the spatial coordinate x and the spatial differential operator $\frac{\partial}{\partial x}$, respectively. The objects `interior`, `leftPoint`, and `rightPoint` are `CellFilter` objects and are used to specify which mesh cells are used in a particular integration. `NonlinearProblem` is an object that is responsible for building Jacobians and residuals given a specification of the weak forms and the mesh. Finally, the objects `solver` and `linSolver` are handles to nonlinear and linear solvers, respectively, from the Trilinos toolkit. More complete documentation for the objects used in this example is distributed with the Sundance source code. Here is the code for the steady-state Burgers' problem.

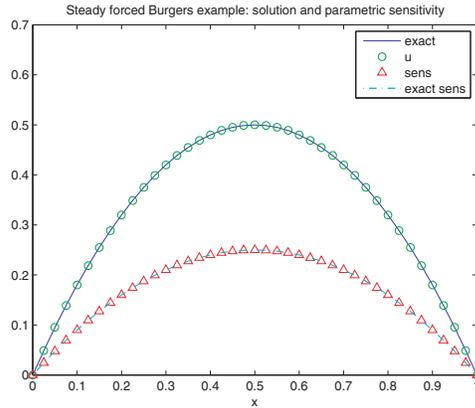


FIG. 1. Solution and sensitivity for steady-state Burgers' equation. The sensitivity is with respect to the parameter p defined in the text. The symbols indicate the numerical results computed by Sundance; the solid and dashed lines indicate the exact curves for the solution and sensitivity.

```

/* Define expressions for parameters */
Expr p = new UnknownParameter("p");
Expr p0 = new Sundance::Parameter(2.0);

/* Define the forcing term */
Expr f = p * (p*x*(2.0*x*x - 3.0*x + 1.0) + 2.0);

/* Define the weak form for the forward problem */
Expr eqn = Integral(interior, (dx*u)*(dx*v) + v*u*(dx*u) - v*f, quad);
/* Define the Dirichlet BC */
Expr bc = EssentialBC(leftPoint+rightPoint, v*u, quad);

/* Create a NonlinearProblem object that manages discretization
 * of the equations on the specified mesh */
NonlinearProblem prob(mesh, eqn, bc, v, u, u0, p, p0, vecType);

/* Use NOX to solve the nonlinear system for the forward problem */
NOX::StatusTest::StatusType status = prob.solve(solver);

/* compute sensitivities */
Expr sens = prob.computeSensitivities(linSolver);

```

Note that the user never explicitly sets up sensitivity equations; rather, the forward problem is created with the design parameters defined as `UnknownParameter` expressions. The same `NonlinearProblem` object supports discretization and solution of both the forward problem and the sensitivity problem. Numerical results are shown in Figure 1.

2.4. PDE-constrained optimization. PDE-constrained optimization methods pose difficult implementation issues for monolithic production codes that from initial conception have not been instrumented to perform efficiently certain nonstandard operations. For instance, the gradient of the objective function can be calculated

using forward sensitivities or adjoint-based sensitivities which require access to the Jacobian and its transpose and the calculation of additional derivatives. Our mathematical framework and software infrastructure completely avoid such low-level details. By applying Fréchet differentiation to a Lagrangian functional the optimality conditions are automatically generated.

To more concretely explain these ideas, we formulate a constrained optimization problem and follow the typical solution strategy of taking variations of a Lagrangian with respect to the state, adjoint, and optimization variables. First, we formulate the minimization of a functional of the form

$$(2.15) \quad F(u, p) = \sum_r \int_{\Omega_r} \mathcal{F}_r(u, p, x) d\mu_r$$

subject to equality constraints written in weak form as

$$(2.16) \quad \lambda^T G(u, p) = \sum_r \int_{\Omega_r} \mathcal{G}_r(u, p, \lambda, x) d\mu_r = 0 \quad \forall \lambda \in \hat{V}.$$

The constraint densities \mathcal{G}_r are assumed to be linear and homogeneous in λ , but can be nonlinear in the state variable u and the design variable p . We form a Lagrangian functional $L = F - \lambda^T G$, with Lagrangian densities $\mathcal{L}_r = \mathcal{F}_r - \mathcal{G}_r$. It is well known [10] that the necessary condition for optimality is the simultaneous solution of the three equations

$$(2.17) \quad \frac{\partial L}{\partial u} = \frac{\partial L}{\partial p} = \frac{\partial L}{\partial \lambda} = 0.$$

In a so-called all-at-once or simultaneous analysis and design (SAND) method [43], we solve these equations simultaneously, typically by means of a Newton or quasi-Newton method. In a reduced space or nested analysis and design (NAND) method, we solve successively the state and adjoint equations, respectively, $\frac{\partial L}{\partial \lambda} = 0$ and $\frac{\partial L}{\partial u} = 0$, while holding the design variables p fixed. The results are then used in calculation of the reduced gradient $\frac{\partial F}{\partial p}$ for use in a gradient-based optimization algorithm such as limited-memory BFGS. In either the SAND or NAND approach, the required calculations still fit within our framework: we represent \mathcal{L}_r symbolically, and then carry out the Fréchet derivatives necessary to form discrete equations. In a SAND calculation, derivatives with respect to all variables are computed simultaneously, whereas in each stage of a NAND calculation two of the variables are held fixed while differentiation is done with respect to the third.

For example, the discrete adjoint equation in a NAND calculation is

$$(2.18) \quad \left[\sum_r \sum_\alpha \int_{\Omega_r} \frac{\partial \mathcal{L}_r}{\partial (D_\alpha u)} D_\alpha \psi_i d\mu_r \right] + \sum_j \lambda_j \left[\sum_r \sum_\alpha \sum_\beta \int_{\Omega_r} \frac{\partial^2 \mathcal{L}_r}{\partial (D_\alpha u) \partial (D_\beta \lambda)} D_\alpha \psi_i D_\beta \phi_j d\mu_r \right] = 0.$$

The state and design equations are obtained similarly, by a permutation of the differentiation variables. In a SAND approach, the discrete, linearized equality-constrained KKT equations are

$$(2.19) \quad \begin{bmatrix} L_{\lambda u} & L_{\lambda \lambda} & L_{\lambda p} \\ L_{u u} & L_{u \lambda} & L_{u p} \\ L_{p u} & L_{p \lambda} & L_{p p} \end{bmatrix} \begin{bmatrix} \delta u \\ \delta \lambda \\ \delta p \end{bmatrix} + \begin{bmatrix} L_u \\ L_\lambda \\ L_p \end{bmatrix} = 0,$$

where the elements of the matrix blocks above are computed through integrations such as

$$(2.20) \quad L_{\lambda_i u_j} = \sum_r \sum_\alpha \sum_\beta \int_{\Omega_r} \left[\frac{\partial^2 \mathcal{L}_r}{\partial(D_\alpha \lambda) \partial(D_\beta u)} D_\alpha \psi_i D_\beta \phi_j d\mu_r \right].$$

Note that because we form discrete problems through differentiation of a Lagrangian, rather than differentiation of a forward solver, our framework leads naturally to the “optimize then discretize” formulation of a discrete optimization problem, in which adjoints and derivatives are taken at the continuous level, then discretized. By contrast, retrofitting an existing forward solver to provide derivatives via automatic differentiation tools (in either forward or backward mode AD) cannot recover the adjoint of the continuous operator. What is done in practice is to use the adjoint of the discretized forward operator, a formulation known as “discretize then optimize.” In general, the adjoint of the discretized forward operator is *not* equal to the discretization of the adjoint operator. The “discretize then optimize” formulation is known [11] to have inferior convergence properties on certain problems compared to “optimize then discretize.” By introducing the adjoint variable explicitly at the continuous level, and by deriving both the forward and adjoint operators through differentiation of a continuous Lagrangian, our approach automatically provides the correct optimized-then-discretized operators.

2.4.1. Example: PDE-constrained optimization problem. To demonstrate this capability we consider a contrived optimization problem in which minimization of a quadratic objective function is constrained by a simple nonlinear PDE. This problem is formulated as

$$\min_{u, \alpha} \frac{1}{2} \int_{\Omega} (u - u^*)^2 d\Omega + \frac{R}{2} \int_{\Omega} \alpha^2 d\Omega$$

subject to

$$\nabla^2 u + 2\pi^2 u + u^2 = \alpha$$

and Dirichlet boundary condition $u(\partial\Omega) = 0$. The Lagrangian for this problem is, after integration by parts,

$$L = \int_{\Omega} \left[\frac{1}{2} (u - u^*)^2 + \frac{R}{2} \alpha^2 + \nabla \lambda \cdot \nabla u - \lambda (2\pi^2 u + u^2) + \lambda \alpha d\Omega \right] + \int_{\partial\Omega} \lambda u d\Omega.$$

We can represent the Lagrangian as a Sundance `Functional` object using the code in Figure 2. Differentiation of the Lagrangian with respect to specified variables is then carried out via calls to certain member functions. It should be re-emphasized that differentiation of the functional object does not produce a new expression graph as when doing symbolic differentiation; rather, it annotates the existing expression graph with instructions for in-place evaluation of the requested derivatives. In this step, either a SAND or a NAND approach can be chosen as directed by the selection of differentiation variables.

Even such a simple nonlinear problem is analytically intractable, so rather than choose a target u^* and then attempt to solve a nonlinear PDE, we again use the method of manufactured solutions to produce the target u^* that yields a specified solution u . From the assumed solution $u = \sin \pi x \sin \pi y$ we derive, successively,

$$(2.21) \quad \alpha = \sin^2 \pi x \sin^2 \pi y,$$

$$(2.22) \quad \lambda = -R \sin^2 \pi x \sin^2 \pi y,$$

```

Expr mismatch = u-uStar;
Expr fit = Integral(interior, 0.5*mismatch*mismatch, quad);
Expr reg = Integral(interior, 0.5*R*alpha*alpha, quad);

Expr g = 2.0*pi*pi*u + u*u;

Expr constraint = Integral(interior, (grad*u)*(grad*lambda)
                          - lambda*g + lambda*alpha, quad);

Expr lagrangian = fit + reg + constraint;

Expr bc = EssentialBC(top+bottom+left+right, lambda*u, quad);

Functional L(mesh, lagrangian, bc, vecType);

```

FIG. 2. Code listing for setting up a model PDE-constrained optimization problem using Sundance.

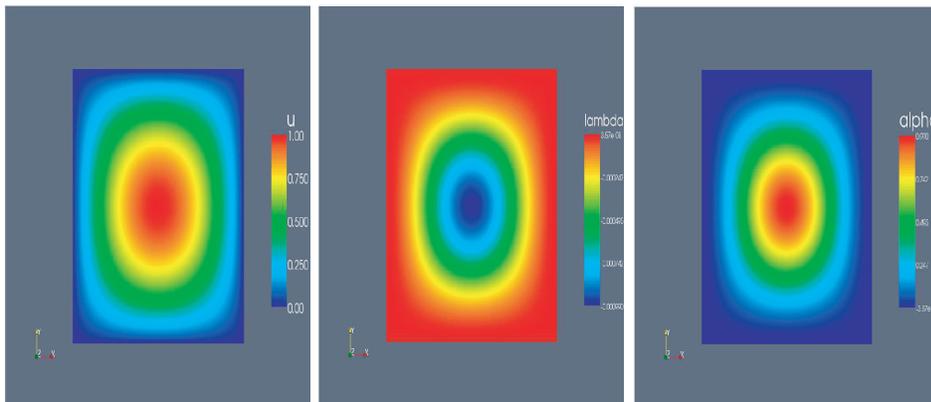


FIG. 3. State, adjoint, and optimization solutions.

and, finally,

$$(2.23) \quad u^* = 2\pi^2 R \cos^2 \pi y \sin^2 \pi x + \sin \pi y (\sin \pi x + 2\pi^2 R \cos^2 \pi x \sin \pi y - 2\pi^2 R \sin^2 \pi x \sin \pi y + 2R \sin^3 \pi x \sin^2 \pi y).$$

Figure 3 shows the state, adjoint, and inversion solutions. As expected, the state and adjoint responses are qualitatively similar though opposite in sign. The solution of the optimization problem, i.e., the optimal design variable α , is shown in the far right window pane.

3. Software architecture. The implementation of our formulation in interoperable software is simplified by noticing that (2.3) suggests a natural partitioning of software components into the following three loosely coupled families:

1. Linear algebra components for matrices, vectors, and solvers. These appear explicitly on the left-hand side of (2.3). In nonlinear problems, vectors representing previous solution iterates will appear implicitly in the evaluation of expressions such as \mathcal{F} .

2. Symbolic components for representation of expressions such as \mathcal{F} and evaluation of its derivatives. We usually refer to these objects as “symbolic” expressions; however, this is something of a misnomer because in the context of discretization many expression types must often be annotated with non-symbolic information such as basis type. A better description is “annotated symbolic expressions” or “quasi-symbolic expressions.”
3. Discretization components for tasks such as evaluation of basis functions and computation of integrals on meshes.

In the discussion of software in this paper we will concentrate on a high-level view of the symbolic components, with a brief mention of mechanisms for interoperability between our symbolic components and third-party discretization components. A detailed explanation of the symbolic representation and evaluation system will follow in a subsequent paper.

To be used in our context, a symbolic system must provide several key capabilities.

1. It must be possible to compute numerically the value of an expression and its Fréchet derivatives at specified spatial points, e.g., quadrature points or nodes. Such computations must be done in-place in a scalable way on a static expression graph; that is, no symbolic manipulations of the graph should be done other than certain trivial constant-time modifications.
2. This numerical evaluation of expression values should be done as efficiently as possible.
3. Functions appearing in an expression must be annotated with an abstract specification of their finite element basis. This enables the automated association of the signature of a Fréchet derivative, i.e., a multiset of functions and spatial derivatives, with a combination of basis functions. If, for example, the function v is expanded in a basis $\{\psi\}$ and the function u in a basis $\{\phi\}$, the association

$$\frac{\partial^2 \mathcal{F}}{\partial v \partial (D_x u)} \rightarrow \psi D_x \phi$$

can be made automatically. It is this association that allows automatic binding of coefficients to elements.

4. It must be possible to specify differentiation with respect to arbitrary combinations of variables.
5. Some expression types must be able to refer to discrete objects such as vectors and mesh cells. However, this dependence on the discrete world must be cleanly partitioned through a mediation interface.

3.1. Evaluation of symbolic expressions. A factor for both performance and flexibility is to distinguish between expression *representation* and expression *evaluation*, by which we mean that the components used to represent an expression graph may not be those used to evaluate it. We use `Evaluator` components to do the actual evaluation. In simple cases, these form a graph that structurally mirrors the expression to be evaluated, but when possible, expression nodes can be aggregated for more efficient evaluation (for example, an expression input in the form $x \cdot x \cdot x$ can be recognized as equivalent to x^3 , so that a one-node power evaluator would be used in place of a tree of product evaluators. Furthermore, it is possible to provide multiple evaluation mechanisms for a given expression. For example, in addition to the default numerical evaluation of an expression, one can construct an evaluator which produces

string representations of the expression and its derivatives; we have in fact done so for practical use during debugging.

Another useful alternative evaluation mechanism is to produce not numerical results but low-level code for computing numerical values of an expression and its derivative. Thus, while the default mode of operation for Sundance components is numerical evaluation of interpreted expressions, these same components could be used to generate code. We therefore do not make a conceptual distinction between our approach to finite element software and other approaches based on code generation, because the possibility of code generation is already built into our design. However, as our performance results will indicate, it seems that replacing our current implementation with generated code does not seem necessary to achieve excellent performance.

Other applications of nonstandard evaluators would be to tune evaluation to hardware architecture, for example, a multithreaded evaluator that distributes subexpression evaluations among multiple cores.

3.1.1. Mixing spatial and functional differentiation. An issue that arises is the interplay of derivatives with respect to the spatial coordinates and derivatives with respect to functions. We have already seen that derivatives such as $\frac{\partial \mathcal{F}}{\partial D_x u}$ may appear in expressions. The complication presented by such derivatives is that they can appear implicitly in cases where the “variable” $D_x u$ does not appear explicitly. For example, the expression

$$\mathcal{F} = D_x [e^{2x} u]$$

has no *explicit* dependence on $D_x u$, but, of course, after computing the spatial derivative appearing in \mathcal{F} to find

$$\mathcal{F} = 2e^{2x} u + e^{2x} D_x u,$$

it is easily seen that

$$\frac{\partial \mathcal{F}}{\partial D_x u} = e^{2x}.$$

However, with complicated expressions it is prohibitively expensive to compute all spatial derivatives—which involves expanding the expression graph—before carrying out functional differentiation.

The solution is to develop in-place functional differentiation rules for expressions involving spatial derivatives. To illustrate the idea, consider the spatial chain rule applied to an expression g that depends on a coordinate x and a function u ,

$$(3.1) \quad D_x g = \frac{\partial g}{\partial x} + \frac{\partial g}{\partial u} D_x u.$$

Note that $\frac{\partial g}{\partial x}$ is nonzero only if g depends *explicitly* on x . Differentiation of (3.1) with respect to $D_x u$ and u results in

$$(3.2) \quad \frac{\partial}{\partial D_x u} (D_x g) = \frac{\partial g}{\partial u},$$

$$(3.3) \quad \frac{\partial}{\partial u} (D_x g) = \frac{\partial^2 g}{\partial u \partial x} + \frac{\partial^2 g}{\partial u^2} D_x u.$$

These rules can be applied in-place concurrently with the evaluation of $D_x g$ with no need to expand the expression graph.

Extension to multiple spatial variables, multiple functions, and higher order derivatives presents notational difficulties, but is conceptually no more difficult than the simple case considered here. The general case will be discussed in detail in our upcoming paper on data flow analysis for in-place Fréchet differentiation.

3.2. Interoperability with discretization components. As foreshadowed above, a challenge in the design of a symbolic expression evaluation system is that some expressions depend explicitly on input from the discrete form of the problem. For example, evaluation of a function u at a linearization point u_0 requires interpolation using a vector and a set of basis functions. Use of a coordinate function such as x in an integral requires its evaluation at transformed quadrature points, which must be obtained from a mesh component.

A guiding principle has been that the symbolic core should interact with other components, e.g., meshes and basis functions, loosely through abstract interfaces rather than through hardwired coupling; this lets us use others' software components for those tasks. We have provided reference implementations for selected components, but the design is intended to use external component libraries as much as possible. The appropriate interface between the symbolic and discretization component systems is the *mediator* pattern [15], which provides a single point of contact between the two component families. The handful of expression subtypes that need discrete information (discrete functions, coordinate expressions, and cell-based expressions such as cell normals arising in boundary conditions and cell diameters arising in stabilization terms) access that information through calls to virtual functions of an `AbstractEvaluationMediator`. Allowing use of discretization components with our symbolic system is then merely a matter of writing an evaluation mediator subclass in which these virtual functions are implemented.

A use case of the mediator is shown in Figure 4. Here, a product of a coordinate expression x and a discrete function u_0 is evaluated. The product evaluator calls the evaluators for the two subexpressions, and their evaluators make appropriate calls to the evaluation mediator.

3.3. Interoperability with solver components. Direct interaction between the symbolic system and solver components is not needed; indirect interaction occurs via discrete function evaluation mediated by the `AbstractEvaluationMediator`. Construction of matrices and vectors, and their use in solvers, occurs within the discretization framework and requires no interaction with the symbolic components.

4. Numerical results.

4.1. A simple example. We first introduce a simple example to cover the fundamental functionality of Sundance. We consider the advection-diffusion equation on the unit square Ω . The bottom, top, left, and right sides of the square are called, respectively, Γ_1 , Γ_2 , Γ_3 , and Γ_4 . The equation and boundary conditions are

$$(4.1) \quad V \cdot \nabla r - k \cdot \Delta r = 0 \quad \text{on } \Omega,$$

$$(4.2) \quad r = 0 \quad \text{on } \Gamma_1,$$

$$(4.3) \quad r = x \quad \text{on } \Gamma_2,$$

$$(4.4) \quad r = 0 \quad \text{on } \Gamma_3,$$

$$(4.5) \quad r = y \quad \text{on } \Gamma_4,$$

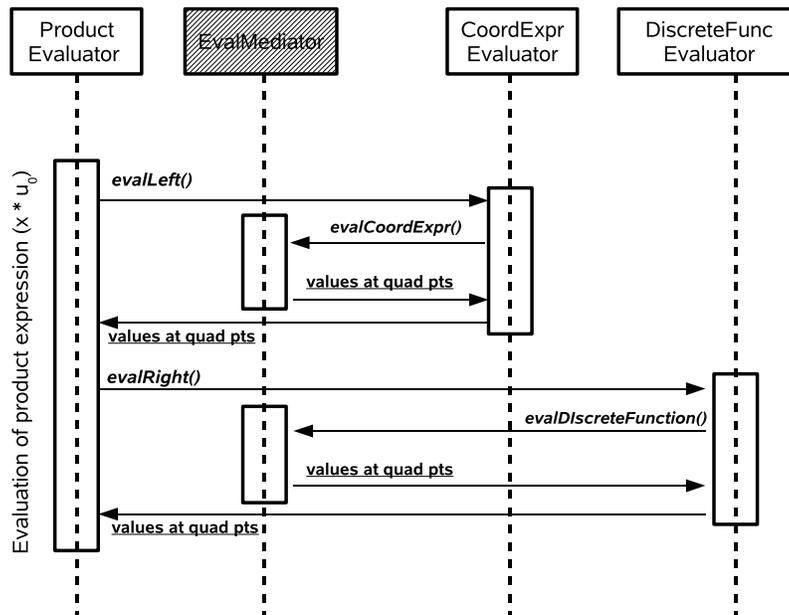


FIG. 4. Unified Modeling Language (UML) sequence diagram showing the evaluation of a product of two framework-dependent expressions through calls to an evaluation mediator. The component in shaded box is framework-dependent; others are framework-independent. Italicized text indicates function calls, and underlined text indicates data returned through function calls. The returned information marked “values at quad points” need not be numerical values; it could be, for instance, a string or possibly generated code.

where r represents concentration, k is the diffusivity, and V is the velocity field, which in this case is set to potential flow:

$$(4.6) \quad \Delta u = 0 \quad \text{on } \Omega,$$

$$(4.7) \quad u = \frac{1}{2}x^2 \quad \text{on } \Gamma_1,$$

$$(4.8) \quad u = \frac{1}{2}(x^2 - 1) \quad \text{on } \Gamma_2,$$

$$(4.9) \quad u = -\frac{1}{2}y^2 \quad \text{on } \Gamma_3,$$

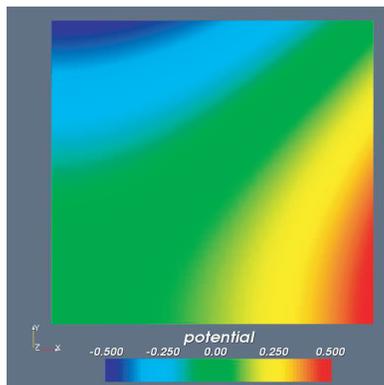
$$(4.10) \quad u = 1 - \frac{1}{2}y^2 \quad \text{on } \Gamma_4.$$

In weak form the advection-diffusion equation is written as

$$(4.11) \quad \int_{\Omega} \nabla s \cdot \nabla r + \int_{\Omega} s \cdot V \cdot \nabla r = 0 \quad \text{on } \Omega,$$

where $V = \nabla u$ and s is the Lagrange polynomial test function. This equation is represented very compactly in terms of expression objects as

```
Expr adEqn = Integral(Omega, (grad*s)*(grad*r), quad2)
+ Integral(Omega, s*V*(grad*r), quad4);,
```

FIG. 5. *Advection-diffusion solution.*

where the `quad2` and `quad4` arguments are objects that specify the quadrature rule used to evaluate each integral. Boundary conditions are specified using similar objects. The internal mesher is used to create a finite element domain of 50×50 simplicial elements in two dimensions. Figure 5 shows the final concentration solution. The complete Sundance code is included in the appendix, which includes basic boilerplate code to enable boundary conditions, meshing, test and trial function definitions, quadrature rules, interface for the linear solver, and postprocessing.

4.2. Single-processor timing results. While the run-time of simulations is typically determined largely by the linear and nonlinear solvers, the extra overhead of interpreting variational forms during matrix assembly could conceivably introduce a new bottleneck into the computation.

Timings of Sundance code shown here were carried out using version 10.0 of Trilinos.

4.3. Comparison to DOLFIN. Here, we compare the performance of Sundance to another high-level finite element method tool, DOLFIN [29], for assembling linear systems for the Poisson and Stokes operators. All of our DOLFIN experiments use code for element matrix computation generated offline by `ffc` rather than the just-in-time compiler strategy available in PyDOLFIN, so the DOLFIN timings include no overhead for the high-level representation of variational forms. Both codes assemble stiffness matrices into an Epetra matrix, so the runs are normalized with respect to the linear algebra back end. The timings in both cases include the initialization of the sparse matrix and evaluation and insertion of all local element matrices into an Epetra matrix. Additionally, the Sundance timings include the overhead of interpreting the variational forms. In both libraries, times to load and initialize a mesh are omitted. It is our goal to assess the total time for matrix assembly, which will indicate whether Sundance's run-time interpretation of forms presents a problem, rather than report detailed profiling of the lower-level components. All runs were done on a MacPro with dual quad-core 2.8GHz processors and 32GB of RAM. Both Sundance and DOLFIN were compiled with versions of the GNU compilers using options recommended by the developers.

DOLFIN depends on the `ffc` project to generate code variational forms, and that tool offers some alternate approaches. For the constant coefficient bilinear forms we consider, the tensor contraction approach developed in [24, 25], which precomputes all

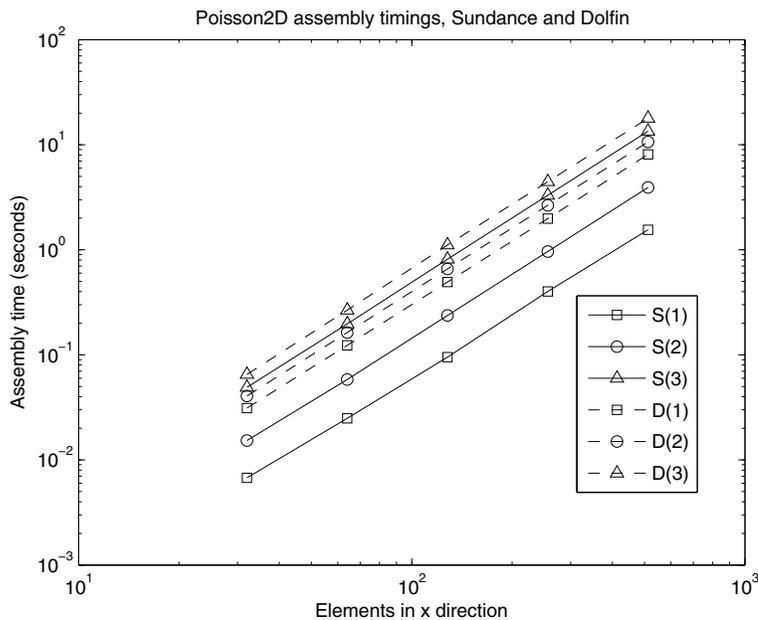


FIG. 6. Timing results for Sundance and Dofin assembly comparisons using the Poisson operator.

integrals on the reference element, is preferred over the quadrature mode developed in [33]. The results shown were obtained using the tensor contraction approach. No Ferrari-based optimizations [23, 27, 28, 26] were used in the `ffc` code. For the forms we consider, these would improve the DOLFIN results by only a marginal amount [33].

In two dimensions, a unit square was divided into an $N \times N$ square mesh, which was then divided into right triangles to produce a three-line mesh. In three dimensions, meshes of a unit cube with the reported numbers of vertices and tetrahedra were generated using Cubit [34]. At this point, Sundance does not rely on an outside element library and only provides Lagrange elements up to order three on triangles and two on tetrahedra, while DOLFIN is capable of using higher order elements through `ffc`'s interface to the FIAT project [22]. The Poisson equation had Dirichlet boundary conditions on faces of constant x value and Neumann conditions on the remaining faces. The Stokes simulations we performed had Dirichlet boundary conditions on velocity over the entire boundary.

Figure 6 and Table 3 show times required to construct the Poisson global stiffness matrices in each library. In all cases, the DOLFIN code actually takes somewhere between a factor of 1.3 and 6 longer than the Sundance code. Figure 7 and Table 4 indicate similar results for the Stokes equations, with the added issue that the DOLFIN runs seemed to run out of memory on the finest meshes. It is interesting that, even including symbolic overhead, Sundance outperforms the DOLFIN programs. We believe this is because Sundance makes very careful use of level 3 BLAS to process batches of cells during the assembly process. It may also have to do with discrete math/bandwidth issues in how global degrees of freedom are ordered. We plan to report on the implementation details of the Sundance assembly engine in a later publication.

TABLE 3

Timings for assembling the Poisson matrix in DOLFIN and Sundance using linear and quadratic elements. In both cases, Sundance requires less time.

Poisson assembly timings, 3D					
Vertices	Tets	$p = 1$		$p = 2$	
		Sundance	Dolfin	Sundance	Dolfin
142	495	0.003626	0.0192	0.01544	0.0278
874	3960	0.02283	0.152	0.129	0.228
6091	31680	0.1761	1.23	1.04	1.90
45397	253440	1.449	10.0	8.617	15.5

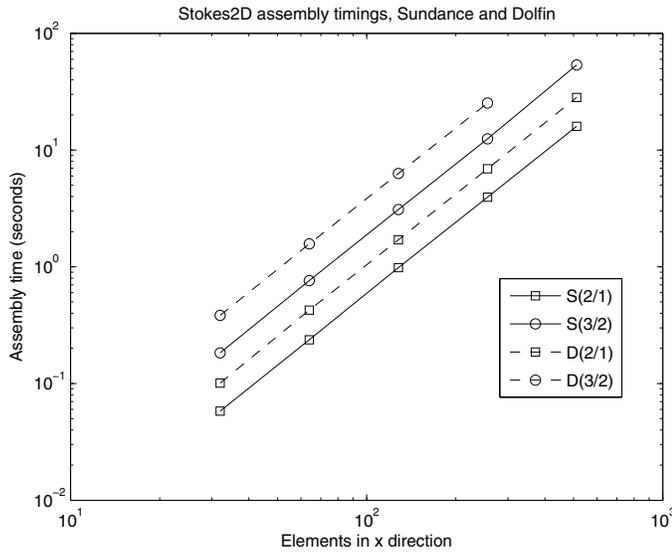


FIG. 7. Timing results for Sundance and Dolfin assembly comparisons using the Stokes operator.

TABLE 4

Sundance significantly outperforms DOLFIN in assembling the Taylor–Hood discretization of the Stokes operator on a range of tetrahedral meshes. Additionally, on the finest mesh, DOLFIN reported an out-of-memory error.

Stokes assembly timings, 3D				
Verts	Tets	$p = 2; 1$		
		Sundance	Dolfin	
142	495	0.07216	0.143	
874	3960	0.6677	1.24	
6091	31680	5.521	10.2	
45397	253440	45.97	crash	

4.3.1. Comparison to low-level C loops. In the next set of experiments we examine the efficiency of the evaluation engine in comparison to low-level C code for the same expressions. We compare the following three methods of evaluating a sequence of univariate polynomials written in Horner’s form:

1. executing the evaluation tree built by the Sundance high-level symbolic objects;

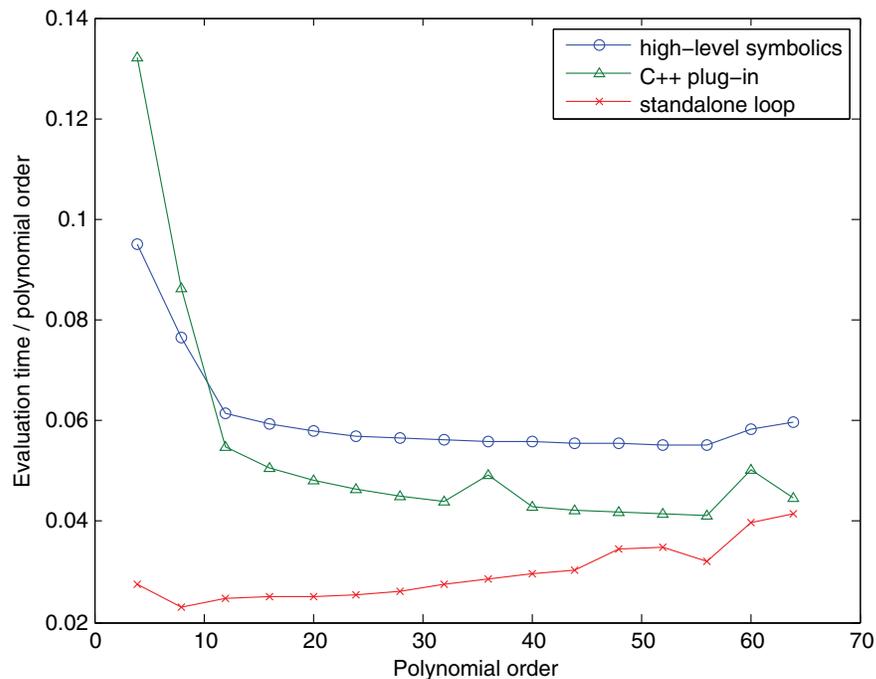


FIG. 8. *Timing comparison between methods of polynomial evaluation.*

2. computation by a standalone C loop, using no Sundance objects;
3. computation by a “plug-in” class that embeds raw C or C++ code in a Sundance expression.

Results are shown in Figure 8. Evidently, evaluation through the expression tree is 2–5 times slower than through a standalone C loop.

It is perhaps surprising that for a small polynomial order, evaluation by the C++ plug-in is less efficient than evaluation through high-level symbolic expressions. However, the plug-in interface is designed for evaluation of multivariable arguments, so there is additional overhead for a simple single-variable argument.

4.3.2. Comparison of automated versus manual linearization. In our final set of timing experiments we examine the efficiency of automating the full Newton linearization of expressions. Given an expression object, Sundance can evaluate its Fréchet derivatives in-place without doing symbolic transformations. Alternatively, one can do the differentiation by hand, then construct an expression object for the linearized form. The test cases we consider are

$$(4.12) \quad u^3 \nabla u \cdot \nabla v,$$

which arises in the radiation diffusion equation, and the nonlinear advection term

$$(4.13) \quad \mathbf{v} \cdot ((\mathbf{u} \cdot \nabla) \mathbf{u}),$$

which arises in the Euler and Navier–Stokes equations of fluid flow. Linearizing by hand gives

$$(4.14) \quad u_0^3 \nabla u_0 \cdot \nabla v + u_0^3 \nabla w \cdot \nabla v + 3u_0^2 w \nabla u_0 \cdot \nabla v$$

and

$$(4.15) \quad \mathbf{v} \cdot ((\mathbf{u}_0 \cdot \nabla) \mathbf{u}_0 + (\mathbf{u}_0 \cdot \nabla) \mathbf{w} + (\mathbf{w} \cdot \nabla) \mathbf{u}_0),$$

where w and \mathbf{w} are the Newton steps.

Results are shown in Figures 9 and 10. Each curve in Figure 9 is a ratio of wall clock times: ratio of the time for a calculation with the hand-linearized expression (4.14) to the time for the same calculation using in-place derivative evaluation on expression (4.12). Figure 10 shows the same results for (4.13) and (4.15). The timings shown are for the complete matrix/vector assembly (circles), evaluation of expressions (triangles), and matrix/vector fill (squares). Timings were averaged over 100 calculations at each of a range of two-dimensional mesh sizes; we expect no significant changes in timing ratios as mesh size is varied. The matrix/vector fill timing ratios were included as a baseline check: both modes of linearization compute and insert exactly the same matrix and vector entries, so the timing ratios for matrix/vector fill should be 1. Other than some noise, that appears to be the case.

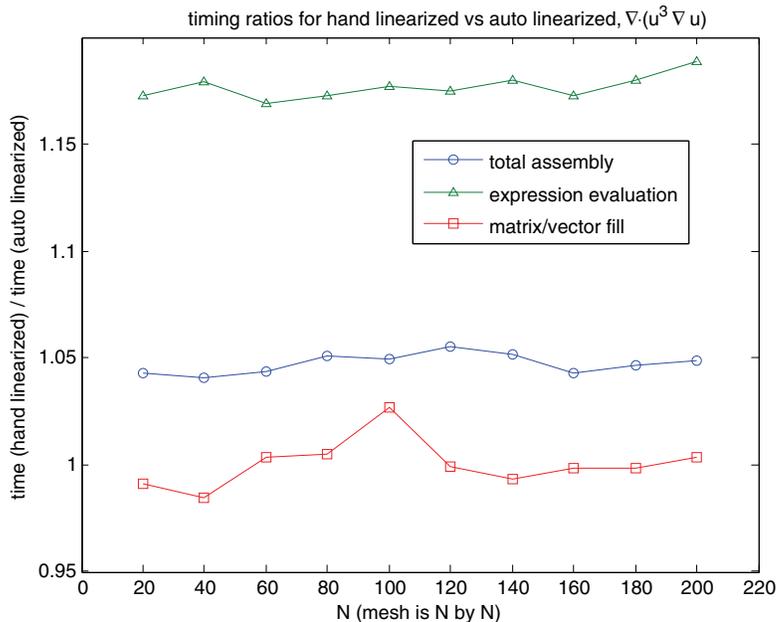


FIG. 9. Timing results for computations involving expressions (4.12) and (4.14).

In both test cases, evaluation of the hand-linearized expression is slightly (10–20%) more expensive than automated in-place derivative evaluation. Because evaluation of the coefficient expressions is only part of the total assembly work (which also includes local integrations and matrix/vector fill), this difference yields a narrower (3–5%) advantage in total assembly time. These results are not unexpected, as the in-place differentiation does the same calculations with less expression traversal overhead.

4.4. Parallel timings. A design requirement is that efficient parallel computation should be transparent to the simulation developer. The novel feature of Sundance, the differentiation-based intrusion, requires no communication and so should have no

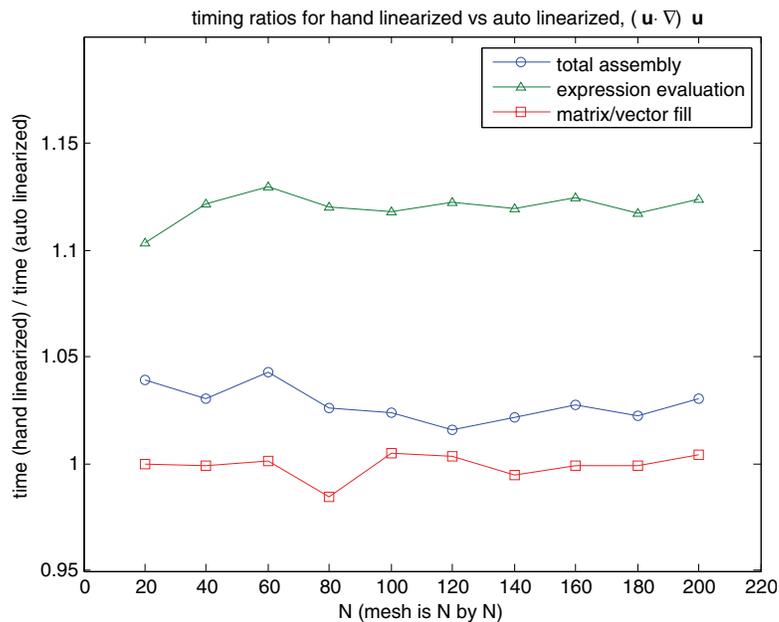


FIG. 10. Timing results for computations involving expressions (4.13) and (4.15).

TABLE 5

Sundance assembly demonstrates perfect weak scaling, where the problem size increases along with the number of processors to give a constant amount of work per processor.

Processors	4	16	32	128	256
Assembly time (s)	54.5	54.7	54.3	54.4	54.4

impact on weak scalability. Table 5 shows assembly times for a model convection-diffusion-reaction problem on up to 256 processors of ASC RedStorm at Sandia National Laboratories. As expected, we see weak scalability on a multiprocessor architecture.

The scalability of the *solve* is another issue and depends critically on problem formulation, boundary condition formulation, and preconditioner in addition to the distributed matrix and vector implementation. An advantage of our approach is that it provides the flexibility needed to simplify the development of algorithms for scalable simulations. To provide low-level parallel services, we defer to a library such as Trilinos [20].

4.5. Thermal-fluid coupling. As we have just seen, the run-time interpretation of variational forms does not adversely impact Sundance's performance. Moreover, defining variational forms at run-time provides opportunities for code reuse. Sundance variational forms may be defined without regard to the degree of polynomial basis; efficiency is obtained without special-purpose code for each polynomial degree. Besides this, the same functions defining variational forms may be reused in a variety of ways, which may be useful in the context of nonlinear coupled problems. Namely, we may use the object polymorphism of the `Expr` class to define variational forms that can work uniformly on trial, test, or discrete functions. Because of the many parameters needed to specify iteration strategies (fixed point versus Newton, contin-

uation in dimensional parameters, preconditioning, inexact iterations, and adapting tolerances), it is difficult to present timing results in this context. However, the results of the previous sections indicate that the actual work of constructing matrices will be performed efficiently in any of these contexts.

We apply this concept to a nonlinear coupled system, the problem of Benard convection [18, 9]. In this problem, a Newtonian fluid is initially stationary, but is heated from the bottom. The density of the fluid decreases with increasing temperature. At a critical value of a certain parameter, the fluid starts to overturn. Fluid flow transports heat, which in turn changes the distribution of buoyant forces.

In nondimensional form, the steady state of this system is governed by a coupling of the Navier–Stokes equations and heat transport. Let $u = (u_x, u_y)$ denote the velocity vector, p the fluid pressure, and c the temperature of the fluid. The parameter Ra is called the Rayleigh number and measures the ratio of energy from buoyant forces to viscous dissipation and heat condition. The parameter Pr is called the Prandtl number and measures the ratio of viscosity to heat conduction. The model uses the Boussinesq approximation, in which density differences are assumed to alter the momentum balance only through buoyancy forces. The model is

$$(4.16) \quad \begin{aligned} -\Delta u + u \cdot \nabla u - \nabla p - \frac{Ra}{Pr} c \hat{\mathbf{j}} &= 0, \\ \nabla \cdot u &= 0, \\ -\frac{1}{Pr} \Delta c + u \cdot \nabla c &= 0. \end{aligned}$$

No-flow boundary conditions are assumed on the boundary of a box. The temperature is set to 1 on the bottom and 0 on the top of the box, and no-flux boundary conditions are imposed on the temperature on the sides.

This problem may be written in the variational form of finding u, p, c in the appropriate spaces (including the Dirichlet boundary conditions) such that

$$(4.17) \quad A[u, v] - B[p, v] + B[w, u] + C[u, u, v] + D[c, v] + E[u, c, q] = 0$$

for all test functions $v = (v_x, v_y)$, w , and q , where the variational forms are

$$(4.18) \quad \begin{aligned} A[u, v] &= \int_{\Omega} \nabla u : \nabla v \, dx, \\ B[p, v] &= \int_{\Omega} p \nabla \cdot v \, dx, \\ C[w, u, v] &= \int_{\Omega} w \cdot \nabla u \cdot v \, dx, \\ D[c, v] &= \frac{Ra}{Pr} \int_{\Omega} c v_x \, dx, \\ E[u, c, q] &= \int_{\Omega} \nabla c \cdot \nabla q + (u \cdot \nabla c) q \, dx. \end{aligned}$$

This standard variational form is suitable for inf-sup stable discretizations such as Taylor–Hood. Convective stabilization such as streamline diffusion is also possible, but is omitted for clarity of presentation.

While the full nonlinear system expressed by (4.17) may be directly defined and differentiated for a Newton-type method in Sundance, typically a more robust (though

```

Expr flowEquation( Expr flow , Expr lagFlow
                  Expr varFlow , Expr temp ,
                  Expr rayleigh, Expr inv_prandtl .
                  QuadratureFamily quad )
{
  CellFilter interior = new MaximalCellFilter();
  /* Create differential operators */
  Expr dx = new Derivative(0); Expr dy = new Derivative(1);
  Expr grad = List(dx, dy);

  Expr ux = flow[0]; Expr uy = flow[1]; Expr u = List( ux , uy );
  Expr lagU = List( lagFlow[0] , lagFlow[1] );
  Expr vx = varFlow[0]; Expr vy = varFlow[1];
  Expr p = flow[2]; Expr q = varFlow[2];
  Expr temp0 = temp;
  return Integral(interior,
                 (grad*vx)*(grad*ux) + (grad*vy)*(grad*uy)
                 + vx*(lagU*grad)*ux + vy*(lagU*grad)*uy
                 - p*(dx*vx+dy*vy) - q*(dx*ux+dy*uy)
                 - temp0*rayleigh*inv_prandtl*vy,quad);
}

```

FIG. 11. Flow equations for convection. The Expr lagFlow argument can be equal to flow to create nonlinear coupling, or as a DiscreteFunction to lag the convective velocity. Additionally, the temp argument may be either an UnknownFunction or a DiscreteFunction.

more slowly converging) iteration is required to reach the ball of convergence for Newton. One such possible strategy is a fixed-point iteration. Start with initial guesses u^0, p^0 , and T^0 . Then, define u^{i+1} and p^{i+1} by the solution of

$$(4.19) \quad A[u^{i+1}, v] - B[p^{i+1}, v] + B[w, u^{i+1}] + C[u^i, u^{i+1}, v] + D[c^i, v] = 0$$

for all test functions v and w , which is a linear Oseen-type equation with a forcing term. Note that the previous iteration of temperature is used, and the convective velocity is lagged so that this is a linear system. Then, c^{i+1} is defined as the solution of

$$(4.20) \quad K[u^{i+1}, c^{i+1}, q] = 0,$$

which is solving the temperature equation with a fixed velocity u^{i+1} .

We implemented both Newton and fixed-point iterations for P_2/P_1 Taylor–Hood elements in Sundance, using the same functions defining variational forms in each case. Using the run-time polymorphism of Expr, we wrote a function flowEquation that groups the Navier–Stokes terms and buoyant forcing term, shown in Figure 11. Then, to implement the iteration strategy, we formed two separate equations. The first calls flowEquation, the actual UnknownFunction flow variables for flow, and the previous iterate stored in a DiscreteFunction for lagFlow and for the temperature. The second equation does the analogous thing in tempEquation, as shown in Figure 12. This allows us to form two linear problems and alternately solve them. After enough iterations, we used these same functions to form the fully coupled system. If ux, uy, p, T are the UnknownFunction objects, the fully coupled equations are obtained through the code in Figure 13.

```

Expr tempEquation( Expr temp , Expr varTemp , Expr flow ,
                  Expr inv_prandtl ,
                  QuadratureFamily quad )
{
  CellFilter interior = new MaximalCellFilter();
  Expr dx = new Derivative(0); Expr dy = new Derivative(1);
  Expr grad = List(dx, dy);

  return Integral( interior ,
                  inv_prandtl * (grad*temp)*(grad*varTemp)
                  + (flow[0]*(dx*temp)+flow[1]*(dy*temp))*varTemp ,
                  quad );
}

```

FIG. 12. *Polymorphic implementation of temperature equation, where the flow variable may be passed as a DiscreteFunction or an UnknownFunction variable to enable full coupling or fixed-point strategies, respectively.*

```

Expr fullEqn = flowEquation( List( ux , uy , p ) ,
                           List( ux , uy , p ) ,
                           List( vx , vy , q ) ,
                           T , rayleigh, inv_prandtl , quad )
+ tempEquation( T , w , List( ux , uy , p ) , inv_prandtl , quad );

```

FIG. 13. *Function call to form fully coupled convection equations.*

Benard convection creates many interesting numerical problems. We have already alluded to the difficulty in finding an initial guess for a full Newton method. Moreover, early in the iterations, the solutions change very little, which can deceive solvers into thinking they have converged when they actually have not. A more robust solution strategy (which could also be implemented in Sundance) would be solving a series of time-dependent problems until a steady state has been reached. Besides difficulties in the algebraic solvers, large Rayleigh numbers can lead to large fluid velocities, which imply a high effective Peclet number and the need for stabilized methods in the temperature equation.

Finally, our iteration technique is designed to illustrate the ease with which Sundance supports various strategies such as fixed points and Newton methods. The abstract differentiation techniques work uniformly in both cases to enable assembly of system operators for the various linear and nonlinear problems. Other solution strategies are also possible, such as using a continuation loop on the Rayleigh number and solving each system by Newton iteration.

5. Conclusions and future work. The technology incorporated in Sundance represents concrete mathematical and computational contributions to the finite element community. We have shown how the diverse analysis calculations such as sensitivity and optimization are actually instances of the same mathematical structure. This mathematical insight drives a powerful, high-performance code; once abstractions for these functionals and their high-level derivatives exist in code, they may be unified with more standard low-level finite element tools to produce a very powerful general-purpose code that enables basic simulation as well as analysis calculations

essential for engineering practice. The performance numbers included here indicate that we have provided a very efficient platform for doing these calculations, despite the seeming disadvantage of run-time interpretation of variational forms.

In the future, we will develop additional papers documenting how the assembly engine achieves such good performance as well as how the data flow for Sundance's automatic differentiation works. In addition, we will further improve the symbolic engine to recognize composite differential operators (divergences, gradients, and curls) rather than atomic partial derivatives. This will not only improve the top-level user experience but allow for additional internal reasoning about problem structure. Beyond this, we are in the process of improving Sundance's discretization support to include more general finite element spaces such as Raviart–Thomas and Nédélec elements, an aspect in which Sundance lags behind other codes such as DOLFIN and Deal.II. Finally, the ability to generate new operators for embedded algorithms opens up possibilities for simplifying the implementation of physics-based preconditioners.

Appendix. The following is the complete source code for the advection-diffusion shown in section 4.1. Explanation of the classes and functions used can be found in the documentation bundled with the Sundance source code.

```
// Sundance AD.cpp for Advection-Diffusion with Potential flow

#include ‘‘Sundance.hpp’’

CELL_PREDICATE(LeftPointTest, {return fabs(x[0]) < 1.0e-10;})
CELL_PREDICATE(BottomPointTest, {return fabs(x[1]) < 1.0e-10;})
CELL_PREDICATE(RightPointTest, {return fabs(x[0]-1.0) < 1.0e-10;})
CELL_PREDICATE(TopPointTest, {return fabs(x[1]-1.0) < 1.0e-10;})

int main(int argc, char** argv)
{
    try
    {
        Sundance::init(&argc, &argv);
        int np = MPIComm::world().getNProc();

        /* linear algebra using Epetra */
        VectorType<double> vecType = new EpetraVectorType();

        /* Create a mesh */
        int n = 50;
        MeshType meshType = new BasicSimplicialMeshType();
        MeshSource mesher = new PartitionedRectangleMesher(0.0, 1.0, n, np,0.0,
            1.0, n, meshType);
        Mesh mesh = mesher.getMesh();

        /* Create a cell filter to identify maximal cells in the interior (Omega)
           of the domain */
        CellFilter Omega = new MaximalCellFilter();
        CellFilter edges = new DimensionalCellFilter(1);
        CellFilter left = edges.subset(new LeftPointTest());
        CellFilter right = edges.subset(new RightPointTest());
        CellFilter top = edges.subset(new TopPointTest());
        CellFilter bottom = edges.subset(new BottomPointTest());
```

```

/* Create unknown & test functions, discretized with first-order Lagrange
   interpolants */
int order = 2;
Expr u = new UnknownFunction(new Lagrange(order), "u");
Expr v = new TestFunction(new Lagrange(order), "v");

/* Create differential operator and coordinate functions */
Expr dx = new Derivative(0);
Expr dy = new Derivative(1);
Expr grad = List(dx, dy);
Expr x = new CoordExpr(0);
Expr y = new CoordExpr(1);

/* Quadrature rule for doing the integrations */
QuadratureFamily quad2 = new GaussianQuadrature(2);
QuadratureFamily quad4 = new GaussianQuadrature(4);

/* Define the weak form for the potential flow equation */
Expr flowEqn = Integral(Omega, (grad*v)*(grad*u), quad2);

/* Define the Dirichlet BC */
Expr flowBC = EssentialBC(bottom, v*(u-0.5*x*x), quad4)
  + EssentialBC(top, v*(u - 0.5*(x*x - 1.0)), quad4)
  + EssentialBC(left, v*(u + 0.5*y*y), quad4)
  + EssentialBC(right, v*(u - 0.5*(1.0-y*y)), quad4);

/* Set up the linear problem! */
LinearProblem flowProb(mesh, flowEqn, flowBC, v, u, vecType);

ParameterXMLFileReader reader(searchForFile("bigstab.xml"));
ParameterList solverParams = reader.getParameterList();
cerr << "params = " << solverParams << endl;
LinearSolver<double> solver
  = LinearSolverBuilder::createSolver(solverParams);

/* Solve the problem */
Expr u0 = flowProb.solve(solver);

/* Set up and solve the advection-diffusion equation for r */
Expr r = new UnknownFunction(new Lagrange(order), "u");
Expr s = new TestFunction(new Lagrange(order), "v");

Expr V = grad*u0;
Expr adEqn = Integral(Omega, (grad*s)*(grad*r), quad2)
  + Integral(Omega, s*V*(grad*r), quad4);

Expr adBC = EssentialBC(bottom, s*r, quad4)
  + EssentialBC(top, s*(r-x), quad4)
  + EssentialBC(left, s*r, quad4)
  + EssentialBC(right, s*(r-y), quad4);

LinearProblem adProb(mesh, adEqn, adBC, s, r, vecType);
Expr r0 = adProb.solve(solver);

```

```

FieldWriter w = new VTKWriter("AD-2D");
w.addMesh(mesh);
w.addField("potential", new ExprFieldWrapper(u0[0]));
w.addField("potential2", new ExprFieldWrapper(u0[1]));
w.addField("concentration", new ExprFieldWrapper(r0[0]));
w.write();

}
catch(exception& e)
Sundance::handleException(e);
Sundance::finalize();
}

```

REFERENCES

- [1] M. S. ALNAES AND A. LOGG, *UFL Specification and User Manual 0.1*, <http://www.fenics.org/pub/documents/ufl/ufl-user-manual/ufl-user-manual.pdf> (April 2009).
- [2] M. S. ALNAES, A. LOGG, K.-A. MARDAL, O. SKAVHAUG, AND H. P. LANGTANGEN, *Unified framework for finite element assembly*, *Int. J. Comput. Sci. Eng.*, 4 (2009), pp. 231–244.
- [3] V. I. ARNOLD, *Mathematical Methods of Classical Mechanics*, Springer-Verlag, New York, 1989.
- [4] B. BAGHERI AND L. RIDGWAY SCOTT, *About Analysa*, Technical report TR-2004-09, Department of Computer Science, University of Chicago, Chicago, IL, 2004.
- [5] W. BANGERTH, R. HARTMANN, AND G. KANSCHAT, *deal. II—a general-purpose object-oriented finite element library*, *ACM Trans. Math. Software*, 33 (2007), article 24.
- [6] P. BIENTINESI, E. S. QUINTANA-ORTÍ, AND R. A. VAN DE GEIJN, *Representing linear algebra algorithms in code: The FLAME application programming interfaces*, *ACM Trans. Math. Software*, 31 (2005), pp. 27–59.
- [7] J. J. BINNEY, N. J. DOWRICK, A. J. FISHER, AND M. E. J. NEWMAN, *The Theory of Critical Phenomena: An Introduction to the Renormalization Group*, Oxford University Press, New York, 1992.
- [8] P. M. CHAIKIN AND T. C. LUBENSKY, *Principles of Condensed-Matter Physics*, Cambridge University Press, Cambridge, UK, 1995.
- [9] S. CHANDRASEKHAR, *Hydrodynamic and Hydromagnetic Stability*, Dover, New York, 1981.
- [10] E. W. CHENEY, *Analysis for Applied Mathematics*, Springer-Verlag, New York, 2001.
- [11] S. S. COLLIS AND M. HEINKENSCHLOSS, *Analysis of the Streamline Upwind/Petrov Galerkin Method Applied to the Solution of Optimal Control Problems*, Technical report TR02-01, Department of Computational and Applied Mathematics, Rice University, Houston, TX, 2002.
- [12] S. EGNER AND M. PÜSCHEL, *Automatic generation of fast discrete signal transforms*, *IEEE Trans. Signal Process.*, 49 (2001), pp. 1992–2002.
- [13] S. EGNER AND M. PÜSCHEL, *Symmetry-based matrix factorization*, *J. Symbolic Comput.*, 37 (2004), pp. 157–186.
- [14] H. ELMAN, D. SILVESTER, AND A. WATHEN, *Finite Elements and Fast Linear Solvers*, Oxford University Press, New York, 2005.
- [15] E. GAMMA, R. HELM, R. JOHNSON, AND J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.
- [16] A. GRIEWANK, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, SIAM, Philadelphia, 2000.
- [17] J. A. GUNNELS, F. G. GUSTAVSON, G. M. HENRY, AND R. A. VAN DE GEIJN, *FLAME: Formal linear algebra methods environment*, *ACM Trans. Math. Software*, 27 (2001), pp. 422–455.
- [18] E. GUYON, J.-P. HULIN, L. PETIT, AND C. D. MITESCU, *Physical Hydrodynamics*, Oxford University Press, Oxford, UK, 2001.
- [19] F. HECHT, O. PIRONNEAU, A. L. HYARIC, AND K. OHTSUKA, *FreeFEM++ Manual*, 2005; available online from <http://www.freefem.org>.
- [20] M. HEROUX, R. BARTLETT, V. HOWLE, R. HEOKSTRA, J. HU, T. KOLDA, R. LEHOUCQ, K. LONG, R. PAWLOWSKI, E. PHIPPS, A. SALINGER, H. THORNIQUIST, R. TUMINARO, J. WILLENBRING, AND A. WILLIAMS, *An Overview of Trilinos*, Technical report SAND2003-2927, Sandia National Laboratories, Albuquerque, NM, 2003.
- [21] M. A. HEROUX ET AL., *The Trilinos Project*, [http://trilinos.sandia.gov/\(2010\)](http://trilinos.sandia.gov/(2010)).

- [22] R. C. KIRBY, *FIAT: A new paradigm for computing finite element basis functions*, ACM Trans. Math. Software, 30 (2004), pp. 502–516.
- [23] R. C. KIRBY, M. KNEPLEY, A. LOGG, AND L. R. SCOTT, *Optimizing the evaluation of finite element matrices*, SIAM J. Sci. Comput., 27 (2005), pp. 741–758.
- [24] R. C. KIRBY AND A. LOGG, *A compiler for variational forms*, ACM Trans. Math. Software, 32 (2006), pp. 417–444.
- [25] R. C. KIRBY AND A. LOGG, *Efficient compilation of a class of variational forms*, ACM Trans. Math. Software, 33 (2007), article 17.
- [26] R. C. KIRBY AND A. LOGG, *Benchmarking domain-specific compiler optimizations for variational forms*, ACM Trans. Math. Software, 35 (2009), article 10.
- [27] R. C. KIRBY, A. LOGG, L. R. SCOTT, AND A. R. TERREL, *Topological optimization of the evaluation of finite element matrices*, SIAM J. Sci. Comput., 28 (2006), pp. 224–240.
- [28] R. C. KIRBY AND L. R. SCOTT, *Geometric optimization of the evaluation of finite element matrices*, SIAM J. Sci. Comput., 29 (2007), pp. 827–841.
- [29] A. LOGG AND G. N. WELLS, *DOLFIN: Automated finite element computing*, ACM Trans. Math. Software, 37 (2010), pp. 1–28.
- [30] K. LONG, *Efficient discretization and differentiation of partial differential equations through automatic functional differentiation*, in AD 2004—Fourth International Workshop on Automatic Differentiation, B. Norris and P. Hovland, eds., Atlas Conferences, Toronto, 2004.
- [31] K. LONG, *Sundance 2.0 Tutorial*, Technical report SAND2004-4793, Sandia National Laboratories, Albuquerque, NM, 2004.
- [32] J. NITSCHKE, *Über ein Variationsprinzip zur Lösung von Dirichlet-Problemen bei Verwendung von Teilräumen, die keinen Randbedingungen unterworfen sind*, Abh. Math. Sem. Univ. Hamburg, 36 (1971), pp. 9–15.
- [33] K. B. ØLGAARD AND G. N. WELLS, *Optimizations for quadrature representations of finite element tensors through automated code generation*, ACM Trans. Math. Software, 37 (2010), pp. 1–23.
- [34] S. J. OWEN, *Cubit 10.2 Documentation*, Technical report SAND2006-7156P, Sandia National Laboratories, Albuquerque, NM, 2006.
- [35] C. PRUD'HOMME, *A domain specific embedded language in C++ for automatic differentiation, projection, integration and variational formulations*, Sci. Program., 14 (2006), pp. 81–110.
- [36] M. PÜSCHEL, *Decomposing monomial representations of solvable groups*, J. Symbolic Comput., 34 (2002), pp. 561–596.
- [37] M. PÜSCHEL AND J. M. F. MOURA, *Algebraic signal processing theory: Cooley-Tukey type algorithms for DCTs and DSTs*, IEEE Trans. Signal Process., 56 (2008), pp. 3572–3585.
- [38] M. PÜSCHEL, J. M. F. MOURA, J. JOHNSON, D. PADUA, M. VELOSO, B. W. SINGER, J. XIONG, F. FRANCHETTI, A. GAČIĆ, Y. VORONENKO, K. CHEN, R. W. JOHNSON, AND N. RIZZOLO, *SPIRAL: Code generation for DSP transforms*, Proc. IEEE, 93 (2005), pp. 232–275.
- [39] S. N. RASBAND, *Dynamics*, John Wiley & Sons, New York, 1989.
- [40] P. J. ROACHE, *Verification of codes and calculations*, AIAA Journal, 36 (1998), pp. 696–702.
- [41] P. J. ROACHE, *Code verification by the method of manufactured solutions*, J. Fluids Eng., 124 (2002), pp. 4–10.
- [42] H. SAGAN, *Introduction to the Calculus of Variations*, Dover, New York, 1993.
- [43] B. VAN BLOEMEN WAANDERS, R. BARTLETT, K. LONG, P. BOGGS, AND A. SALINGER, *Large Scale Nonlinear Programming for PDE Constrained Optimization*, Technical report SAND2002-3198, Sandia National Laboratories, Albuquerque, NM, 2002.