

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 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 PDE by finite element methods, we present a mathematical framework that unifies the discretization of PDE with these additional analysis requirements. In particular, Fréchet differentiation on a class of functionals together with a high-performance finite element framework have 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

1. Introduction. Advanced simulation of realistic systems governed by partial differential equations (hence PDE) 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. Efficient solution of the underlying linear systems may be facilitated by additional operators introduced by physics-based preconditioning. Beyond this, sensitivity analysis, optimization and control require even further 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*.

Traditional automatic differentiation (AD) tools [15] 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.

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 [3], provide infrastructure for handling meshes, basis functions, assembly, and interfaces to linear solvers. Other projects, such as Analyza [2] and FFC [20, 21] use a high-level input syntax to generate low-level code for assembling variational forms. Yet other projects, such

*Department of Mathematics and Statistics, Texas Tech University (kevin.long@ttu.edu). Author acknowledges support from NSF award 0830655.

†Department of Mathematics and Statistics, Texas Tech University (robert.c.kirby@ttu.edu). Author acknowledges support from NSF award 0830655.

‡Applied Mathematics and Applications, Sandia National Laboratory (bartv@sandia.gov)- Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000, PO Box 5800, Albuquerque, NM 87122.

as as LifeV [26] and FreeFEM [17] 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 (LifeV).

Our present work, encoded in the open-source project Sundance [23, 24], 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, we also provide 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 [15], we differentiate at more abstract level on abstract 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.

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 receives considerable research activity. The Smart project of Püschel *et al.* [11, 12, 27, 28, 29] 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 [4, 16] 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 2.2 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 2.3.1 2.4.1, 2.5.1 and 3 then present some concluding thoughts and directions for future development in Section 4. A complete code listing 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 PDE on a d -dimensional spatial domain Ω . We will use lower-case italic

symbols such as u, v for functions mapping $\Omega \rightarrow \mathbb{R}$. We will denote arbitrary function spaces with upper-case 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 a function to \mathbb{R} ; we denote functionals by upper-case 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 to 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.*, [8]) 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.* [33]), 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.*, [1], [30]) and field theory (*e.g.*, [5], [7]) 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. The spatial coordinates are represented as usual by x, y, \dots . 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

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

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

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

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

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

1. $\frac{\partial \mathcal{F}}{\partial u_i}$, 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 Equation 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. Software Architecture. Equation 2.3 suggests a natural partitioning of software components into loosely-coupled families:

1. Linear algebra components for matrices, vectors, and solvers.
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, computation of integrals.

In the discussion of software in this paper we will concentrate on the symbolic components, with a brief mention of mechanisms for interoperability between our symbolic components and third-party discretization components.

We require a data structure for symbolic expressions that provides 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 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.

2.2.1. 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. 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; such string evaluations are of practical use for 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.

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.

2.2.2. Interoperability with discretization components. 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 for as much as possible. The appropriate interface between the symbolic and discretization component systems is the *mediator* pattern [14], 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 2.1. 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.

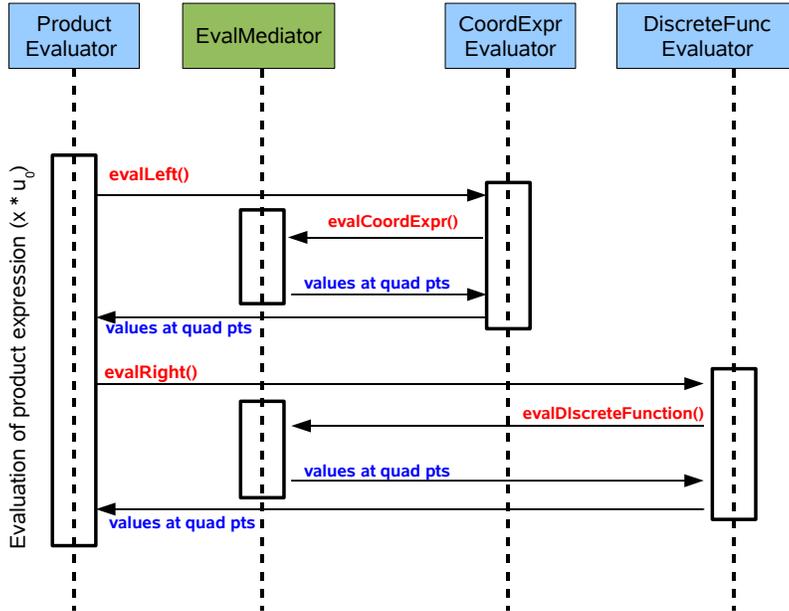


FIG. 2.1. UML sequence diagram showing the evaluation of a product of two framework-dependent expressions through calls to an evaluation mediator. Components in blue boxes are framework-independent. Red text indicates function calls, and blue text indicates data returned through function calls. The returned information, marked “values at quad points,” need not be numerical values; it could be, for instance, a string, or possibly generated code.

With the above overview of mathematical foundations and software architecture in mind, we proceed to show how these principles apply in several types of embedded analysis problems.

2.3. 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 variables

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

is 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 multiindex 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 $\psi \in \hat{V}^h$,

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

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

The requirement that (2.4) holds 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

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

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 ,

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

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 [13]), 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

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

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.

2.3.1. 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 1D domain $\Omega = [0, 1]$,

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

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

$$\int_0^1 [vuD_xu + cD_xvD_xu] dx = 0 \quad \forall v \in H_\Omega^1. \quad (2.11)$$

To cast this into the notation of equation 2.4, we define

$$\mathcal{G} = vuD_xu + cD_xvD_xu \quad (2.12)$$

The nonzero derivatives appearing in the linearized weak Burgers equation are shown in table 2.1. The table makes clear the correspondence between differentiation variables and basis combination and between derivative value and coefficient in the linearized, discretized weak form.

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

TABLE 2.1

This table shows for the Burgers equation example the correspondence, defined by equation 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.

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 2.4.1 below in the context of sensitivity analysis.

2.3.2. Representation of Dirichlet boundary conditions. Dirichlet boundary conditions are usually described conceptually in terms of a restriction of the space of trial functions, but in practice they are often handled in a somewhat *ad hoc* manner by simply replacing rows in the resulting linear system with a trivial equation that sets values at the specified boundary nodes. This procedure gives correct results (though can affect conditioning and hence solver scalability) and is easy to program for forward problems; however, in an optimization or sensitivity problem, when Dirichlet boundary conditions depend on a design variable in a nontrivial way the code must be modified to supply the correct boundary conditions. Therefore, it is essential to present Dirichlet boundary conditions so that they can be implemented automatically in terms of Fréchet derivatives; by doing so, correct boundary conditions for optimization and sensitivity problems fall into place immediately.

Dirichlet boundary conditions can fit into our framework in several ways, including the symmetrized formulation of Nitsche [25] 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.

Handling Dirichlet boundary conditions by row replacement does impact the software design and user interface. At the user level, a simulation developer simply “tags” certain expressions for replacement by creating them with `EssentialBC` functions rather than `Integral` functions. Thus, Dirichlet boundary conditions are encompassed by our differentiation-based approach to computing discrete equations; the only difference from any other kind of expression is that Dirichlet terms must be tagged as such so that the replacement procedure can be carried out.

2.4. 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

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

Differentiating by v_i to obtain discrete equations gives

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

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 is immediately capable of performing sensitivity analysis given a high-level forward problem specification.

2.4.1. Example: Sensitivity Analysis of Burgers' Equation. We now show how this automated production of weak sensitivity equations works in the context of the 1D Burgers equation example from section 2.3.1.

To produce an easily solvable parametrized problem, we apply the method of manufactured solutions [31, 32] 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 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 equation 2.11 to the parameter p . The Fréchet derivatives appearing in the second set of brackets in equation 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 2.2. 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.

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

TABLE 2.2

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

The user-level Sundance code for setting up this problem is shown here.

```

/* Define expressions for parameters */
Expr p = new UnknownParameter("p");
Expr p0 = new SundanceCore::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 TSF NonlinearOperator object */

```

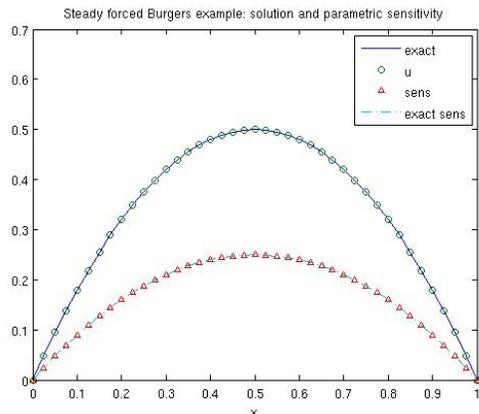


FIG. 2.2. Solution and sensitivity for steady Burgers equation. The sensitivity is with respect to the parameter p defined in the text. The symbols indicate the numerical results computed by Sundance; the solid and dashed lines indicate the exact curves for the solution and sensitivity.

```
NonlinearProblem prob(mesh, eqn, bc, v, u, u0, p, p0, vecType);
```

```
/* 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 2.2.

2.5. PDE-Constrained Optimization. PDE-constrained optimization methods pose difficult implementation issues for monolithic production codes that from initial conception have not been instrumented to efficiently access certain linear algebraic objects. For instance, the gradient of the objective function can be calculated using forward sensitivities or adjoint-based sensitivities which require access to the Jacobian, transposes 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 an optimization problem constrained with simple dynamics 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 as:

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

subject to equality constraints written in weak form as

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

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 [8] that the necessary condition for optimality is the simultaneous solution of the three equations

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

In a so-called ‘‘all-at-once’’ or simultaneous analysis and design (SAND) method [34] 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, 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

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

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

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

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

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

Note that because we form discrete problems using expressions that have already been differentiated, our framework leads naturally to the ‘‘optimize then discretize’’ formulation of a discrete optimization problem. The ‘‘discretize then optimize’’ formulation has been more commonly used because it requires fewer modifications to a forward solver, but is known [10] to have inferior convergence properties compared to ‘‘optimize then discretize’’ on certain problems.

2.5.1. Example: PDE-constrained optimization problem. To demonstrate this capability we consider a contrived optimization problem in which a least squares objective function is constrained by simple dynamics. 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 conditions $u(\partial\Omega) = 0$. The Lagrangian for this problem is, after integration by parts,

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

The Sundance code for this problem is:

```
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);

Even such a simple 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,

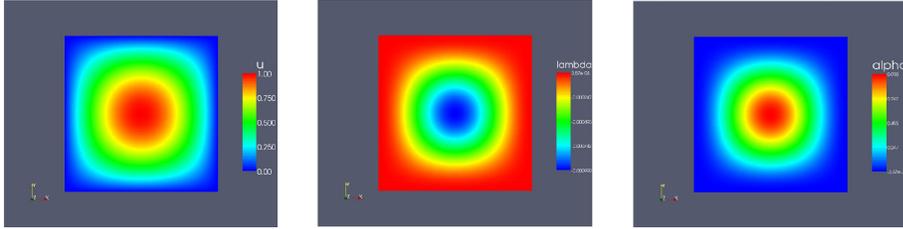
$$\alpha = \sin^2 \pi x \sin^2 \pi y \quad (2.21)$$

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

and finally

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

Figure 2.3 shows the state, adjoint and inversion solutions. As expected, the adjoint demonstrates similar in nature but inverted dynamics as the forward equation. The opposite dynamics of the adjoint is a result of different signs for the diffusive operators from the integration by parts process. The solution of the optimization problem is shown in the far right window pane.

FIG. 2.3. *State, adjoint, and optimization solutions*

3. Numerical Results.

3.1. A Simple Example. We first introduce a simple example to cover the fundamental functionality of Sundance.

$$V \cdot \nabla r - k \cdot \Delta r = 0 \quad \in \Omega \quad (3.1)$$

$$r = 0 \quad \text{on } \Gamma_1 \quad (3.2)$$

$$r = x \quad \text{on } \Gamma_2 \quad (3.3)$$

$$r = y \quad \text{on } \Gamma_3 \quad (3.4)$$

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

$$\Delta u = 0 \quad \in \Omega \quad (3.5)$$

$$u = \frac{1}{2}(x^2 - 1.0) \quad \text{on } \Gamma_1 \quad (3.6)$$

$$u = -\frac{1}{2}y^2 \quad \text{on } \Gamma_2 \quad (3.7)$$

$$u = \frac{1}{2}(1.0 - y^2) \quad \text{on } \Gamma_3 \quad (3.8)$$

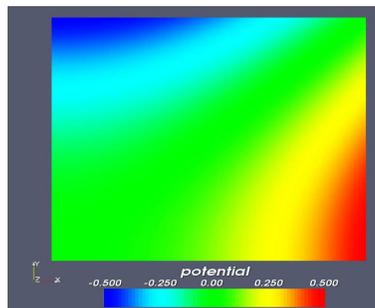
In weak form the advection-diffusion is written as:

$$\int_{\Omega} \nabla s \cdot \nabla r + \int_{\Omega} s \cdot V \cdot \nabla r = 0 \quad \in \Omega \quad (3.9)$$

where $V = \nabla u$ and s is the Lagrange polynomial test function. The dynamics is defined in one line this is represented verbatim as:

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

The internal mesher is used to create a finite element domain of $50 * 50$ simplicial elements in 2D. Figure 3.1 shows the final concentration solution. The complete

FIG. 3.1. *Advection Diffusion Solution*

Sundance code is included in the Appendix which includes basic boiler plate code to enable boundary conditions, meshing, test and trial function definitions, quadrature rules, interface for linear solver, and post processing.

3.2. 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. Here, we compare the performance of Sundance to another high-level finite element method tool, DOLFIN [22], to 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. Sundance and DOLFIN runs assemble stiffness matrices into an Epetra matrix, so the runs are normalized with respect to linear algebra backend. 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 we report 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.

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-lines mesh. In three dimensions, meshes of a unit cube with the reported numbers of vertices and tetrahedra were generated using cubit [9]. 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 [19]. The Poisson equation had Dirichlet boundary conditions on faces of constant x value and Neumann conditions on the remaining. The Stokes simulations we performed had Dirichlet boundary conditions on velocity over the entire boundary.

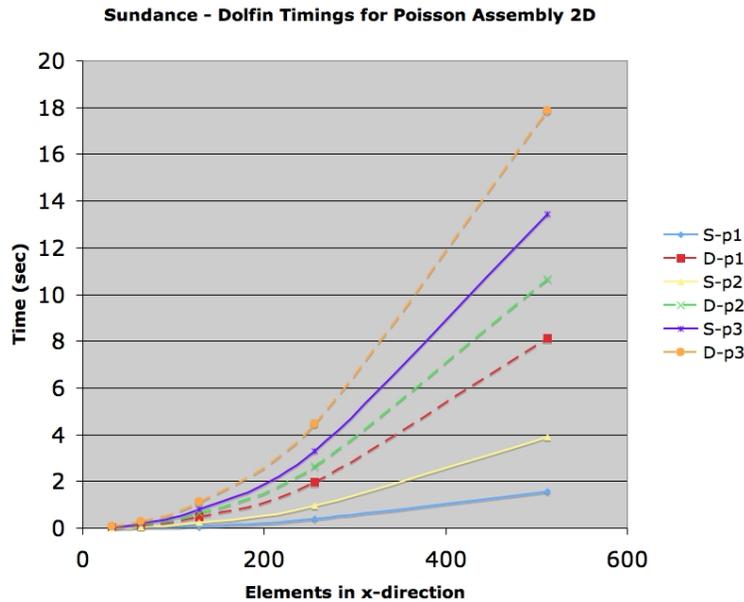


FIG. 3.2. Timing results for Sundance and Dolfin assembly comparisons using the Poisson operator

Figure 3.2 and table 3.2 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 3.3 and table 3.2 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.

Poisson assembly timings, 3D					
vertices	tets	$p = 1$		$p = 2$	
		Sundance	Dolfin	Sundance	Dolfin
142	495	0.003626	0.0194	0.01544	0.03146
874	3960	0.02283	0.1536	0.129	0.2607
6091	31680	0.1761	1.246	1.04	2.149
45397	253440	1.449	10.11	8.617	17.44

3.3. Parallelism. 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 impact on weak scalability. The table below shows assembly times for a model convection-diffusion-reaction problem on up to 256 processors of ASC RedStorm at

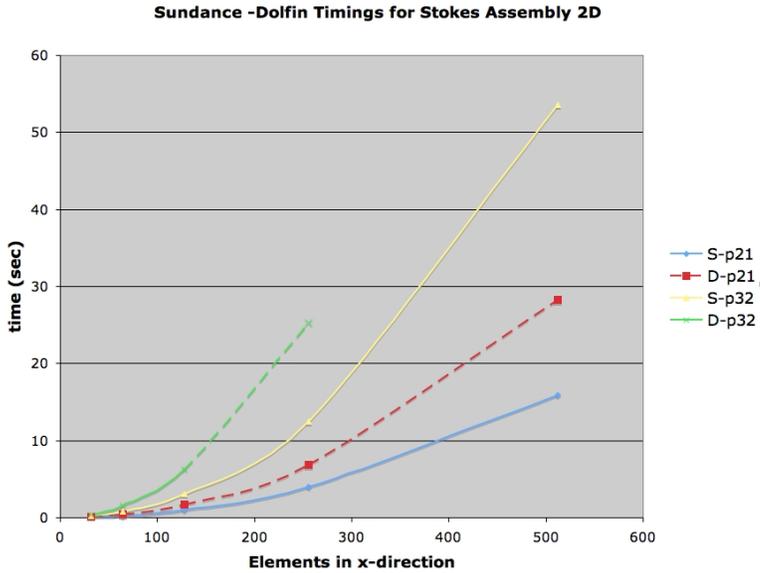


FIG. 3.3. Timing results for Sundance and Dolfin assembly comparisons using the Stokes operator

Stokes assembly timings, 3D			
verts	tets	$p = 2; 1$	
		Sundance	Dolfin
142	495	0.07216	0.3362
874	3960	0.6677	2.793
6091	31680	5.521	22.57
45397	253440	45.97	crash

Sandia National Laboratories.

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

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 [18].

3.4. Thermal-fluid coupling. As we have just seen, the run-time interpretation of variational forms does not seem to 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 on trial, test, or discrete functions uniformly.

We apply this concept to a nonlinear coupled system, the problem of Benard convection [6]. In this problem, a Newtonian fluid is initially stationary, but heated from the bottom. Because of thermal effects, 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 only alter the momentum balance through buoyant forces. The model is

$$\begin{aligned} -\Delta u + u \cdot \nabla u - \nabla p - \frac{Ra}{Pr} c \hat{\mathbf{j}} &= 0 \\ \nabla \cdot u &= 0 \\ -\frac{1}{Pr} \Delta c + u \cdot \nabla T &= 0. \end{aligned} \tag{3.10}$$

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

$$A[u, v] - B[p, v] + B[w, u] + C[u, u, v] + D[c, v] + E[u, c, q] = 0, \tag{3.11}$$

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

$$\begin{aligned} A[u, v] &= \int_{\Omega} \nabla u : \nabla v \, dx \\ B[p, v] &= \int_{\Omega} p \nabla \cdot v \, dx \\ C[w, u, v] &= \int_{\Omega} w \cdot \nabla u \cdot v \, dx \\ D[c, v] &= \frac{Ra}{Pr} \int_{\Omega} c v_x \, dx \\ E[u, c, q] &= \int_{\Omega} \nabla c \cdot \nabla q + (u \cdot \nabla c) q \, dx \end{aligned} \tag{3.12}$$

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 omitted for clarity of presentation.

While the full nonlinear system expressed by (3.11) may be directly defined and differentiated for a Newton-type method in Sundance, typically a more robust (though 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

$$A[u^{i+1}, v] - B[p^{i+1}, v] + B[w, u^{i+1}] + C[u^i, u^{i+1}, v] + D[c^i, v] = 0, \quad (3.13)$$

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

$$K[u^{i+1}, c^{i+1}, q] = 0, \quad (3.14)$$

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 together the Navier-Stokes terms and buoyant forcing term, shown in Figure 3.4. Then, to form the Gauss-Seidel 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`. 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 3.4.

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 have not actually. 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 need for stabilized methods in the temperature equation.

Figure 3.7 shows the temperature computed for $Ra = 5 \times 10^5$ and $Pr = 1$ on a 128×128 mesh subdivided into right triangles. We performed several nonlinear Gauss-Seidel iterations before starting a full Newton solve.

4. 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 of differentiating particular kinds of functionals defined over finite element spaces. 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.

```

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. 3.4. 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 *DiscreteFunction*

```

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. 3.5. Polymorphic implementation of temperature equation, where the *flow* variable may be passed as a *DiscreteFunction* or *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. 3.6. Function call to form fully coupled convection equations.

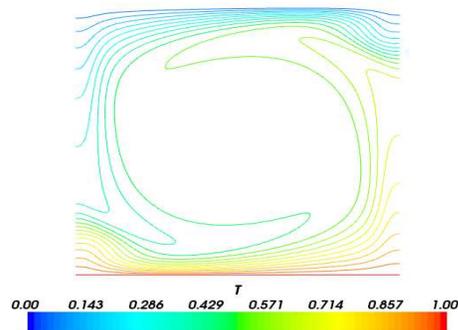


FIG. 3.7. Solution of Benard convection on a 128×128 mesh subdivided into triangles with $Ra = 5 \times 10^5$ and $Pr = 1$.

In the future, we will develop future papers documenting how the assembly engine achieves such good performance as well how the data flow for Sundance’s automatic differentiation works. Besides this, 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 to simplify the implementation of physics-based preconditioners.

REFERENCES

- [1] V. I. ARNOLD, *Mathematical Methods of Classical Mechanics*, Springer-Verlag, 1989.
- [2] BABAK BAGHERI AND L. RIDGWAY SCOTT, *About analysa*, Tech. Report TR-2004-09, The University of Chicago Department of Computer Science, 2004.
- [3] W. BANGERTH, R. HARTMANN, AND G. KANSCHAT, *deal.II — a general-purpose object-oriented finite element library*, ACM Trans. Math. Softw., 33.
- [4] PAOLO BIENTINESI, ENRIQUE S. QUINTANA-ORTÍ, AND ROBERT A. VAN DE GEIJN, *Representing linear algebra algorithms in code: The FLAME application programming interfaces*, ACM Transactions on Mathematical Software, 31 (2005), pp. 27–59.
- [5] 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 Science Publications, 1992.
- [6] EBERHARD BODENSCHATZ, WERNER PESCH, AND GUENTER AHLERS, *Recent developments in rayleigh-bnard convection*, Annual Review of Fluid Mechanics, 32 (2000), pp. 709–778.
- [7] P. M. CHAIKIN AND T. C. LUBENSKY, *Principles of Condensed-Matter Physics*, Cambridge University Press, 1995.
- [8] E. W. CHENEY, *Analysis for Applied Mathematics*, Springer, 2001.

- [9] B. W. CLARK ET AL., *Cubit geometry and mesh generation toolkit*, 2008. <http://cubit.sandia.gov/>.
- [10] S. SCOTT COLLIS AND MATTHIAS HEINKENSCHLOSS, *Analysis of the streamline upwind/ Petrov galerkin method applied to the solution of optimal control problems*, Tech. Report TR02-01, Rice University Computational and Applied Mathematics, 2002.
- [11] SEBASTIAN EGNER AND MARKUS PÜSCHEL, *Automatic generation of fast discrete signal transforms*, IEEE Transactions on Signal Processing, 49 (2001), pp. 1992–2002.
- [12] ———, *Symmetry-based matrix factorization*, Journal of Symbolic Computation, special issue on "Computer Algebra and Signal Processing", 37 (2004), pp. 157–186.
- [13] H. ELMAN, D. SILVESTER, AND A. WATHEN, *Finite Elements and Fast Linear Solvers*, Oxford Science Publications, 2005.
- [14] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON, AND JOHN VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [15] ANDREAS GRIEWANK AND ANDREA WALTHER, *Evaluating derivatives*, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second ed., 2008. Principles and techniques of algorithmic differentiation.
- [16] JOHN A. GUNNELS, FRED G. GUSTAVSON, GREG M. HENRY, AND ROBERT A. VAN DE GELIN, *FLAME: Formal Linear Algebra Methods Environment*, ACM Transactions on Mathematical Software, 27 (2001), pp. 422–455.
- [17] F. HECHT, O. PIRONNEAU, A. LE HYARIC, AND K. OHTSUKA, *FreeFEM++ manual*, 2005.
- [18] 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*, Tech. Report SAND2002-2729, Sandia National Laboratories, 2003.
- [19] R. C. KIRBY, *FIAT: A new paradigm for computing finite element basis functions*, ACM Trans. Math. Software, 30 (2004), pp. 502–516.
- [20] R. C. KIRBY AND A. LOGG, *A compiler for variational forms*, ACM Transactions on Mathematical Software, 32 (2006), pp. 417–444.
- [21] ———, *Efficient compilation of a class of variational forms*, ACM Transactions on Mathematical Software, 33 (2007).
- [22] ANDERS LOGG ET AL., *The FEniCS dolfin project*, 2007. <http://www.fenics.org/wiki/DOLFIN>.
- [23] KEVIN LONG, *Chapter contribution in "Large-Scale PDE-Constrained Optimization, L. Biegler, O. Ghattas, M. Heinkenschloss, and B. van Bloemen Waanders editors"*, vol. 30 of Lecture Notes in Computational Science and Engineering, Springer-Verlag, 2003.
- [24] ———, *Sundance*, 2003. <http://www.math.ttu.edu/~klong/Sundance/html/>.
- [25] J. NITSCHKE, *ber ein variationsprinzip zur lsung von dirichlet-problemen bei verwendung von teilrumen, die keinen randbedingungen unterworfen sind.*, Abhandlungen aus dem Mathematischen Seminar der Universitt Hamburg, 36 (1971), pp. 9–15.
- [26] CHRISTOPHE PRUD'HOMME, *A domain specific embedded language in c++ for automatic differentiation, projection, integration and variational formulations*, Scientific Programming, 14 (2006), pp. 81–110.
- [27] MARKUS PÜSCHEL, *Decomposing monomial representations of solvable groups*, Journal of Symbolic Computation, 34 (2002), pp. 561–596.
- [28] MARKUS PÜSCHEL AND JOSÉ M. F. MOURA, *Algebraic signal processing theory: Cooley-Tukey type algorithms for DCTs and DSTs*, IEEE Transactions on Signal Processing, (2008). to appear.
- [29] MARKUS PÜSCHEL, JOSÉ M. F. MOURA, JEREMY JOHNSON, DAVID PADUA, MANUELA VELOSO, BRYAN W. SINGER, JIANXIN XIONG, FRANZ FRANCHETTI, ACA GAČIĆ, YEVGEN VORONENKO, KANG CHEN, ROBERT W. JOHNSON, AND NICK RIZZOLO, *SPIRAL: Code generation for DSP transforms*, Proceedings of the IEEE, 93 (2005). special issue on "Program Generation, Optimization, and Adaptation".
- [30] N. RASBAND, *Dynamics*, Wiley, 1989.
- [31] P. J. ROACHE, *Verification and Validation in Computational Science and Engineering*, Hermosa, Albuquerque, NM, 1998.
- [32] ———, *Code Verification by the Method of Manufactured Solutions*, Transactions of the ASME, 124 (2002), pp. 4–10.
- [33] H. SAGAN, *An Introduction to the Calculus of Variations*, Dover, 1993.
- [34] B. VAN BLOEMEN WAANDERS, R. BARTLETT, K. LONG, P. BOGGS, AND A. SALINGER, *Large scale non-linear programming for PDE constrained optimization*, Tech. Report SAND2002-3198, Sandia National Laboratories, 2002.

5. Appendix.

```

// 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.getParameters();
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();
}

```