

Sundance: High-Level Software for PDE-Constrained Optimization

Kevin Long*, Paul T. Boggs[†] and Bart G. van Bloemen Waanders[‡]

April 5, 2012

Abstract

Sundance is a package in the Trilinos suite designed to provide high-level components for the development of high-performance PDE simulators with built-in capabilities for PDE-constrained optimization. We review the implications of PDE-constrained optimization on simulator design requirements, then survey the architecture of the Sundance problem specification components. These components allow immediate extension of a forward simulator for use in an optimization context. We show examples of the use of these components to develop full-space and reduced-space codes for linear and nonlinear PDE-constrained inverse problems.

1 Introduction

Numerical optimization has become an essential tool for mathematicians, scientists and engineers. Manufacturers seek to maximize efficiency in their production operations. Aerodynamicists try to minimize drag of airplane wings while maximizing lift characteristics. Geophysicists strive to determine material properties in subsurface structures. In each case, a “model” can be identified that drives the underlying state or dynamics of the system. For shape optimization of a wing, the underlying model consists of the compressible fluid flow equations, whereas in the case of determining subsurface material properties, the model consists of the wave equation. These problems can be formulated as a minimization or maximization of a function subject to a model as constraints on its variables. This can be expressed mathematically as:

$$\min_{\boldsymbol{\alpha}} \mathcal{J}(\mathbf{u}(\boldsymbol{\alpha}), \boldsymbol{\alpha}) \quad (1)$$

$$\text{s.t. } \mathbf{c}(\mathbf{u}, \boldsymbol{\alpha}) = \mathbf{0} \quad (2)$$

$$\mathbf{h}(\mathbf{u}, \boldsymbol{\alpha}) \geq \mathbf{0} \quad (3)$$

where $\mathcal{J}(\mathbf{u}(\boldsymbol{\alpha}), \boldsymbol{\alpha})$ is the objective function, $\mathbf{c}(\mathbf{u}(\boldsymbol{\alpha}), \boldsymbol{\alpha}) = \mathbf{0}$ represents the “model” or state equations, $\boldsymbol{\alpha}$ are the design variables, \mathbf{u} are the state variables, and \mathbf{h} are the inequality constraints. We write $\mathbf{u}(\boldsymbol{\alpha})$ to indicate that for any value of the design variables, we can solve for the state variables. Our solution strategies require the knowledge of both the state (model variables) and the design variables, and we therefore specifically include these in the above described, general formulation of the optimization problem. The inequality constraints (3) are often simple bounds on the design variables, but could be used more generally to ensure that certain (nonlinear) functions of the state and/or design variables are appropriately bounded. We refer to this formulation as a constrained optimization problem and a significant body of literature (a subset of which is listed here) [17, 8, 7, 9, 3, 22, 13, 6] deals with appropriate solution methods and algorithms.

This paper is focused on the common and important case in which the constraints $\mathbf{c}(\mathbf{u}, \boldsymbol{\alpha}) = \mathbf{0}$ are partial differential equations, a class of problems known as PDE-constrained optimization (PDECO). The size, complexity, and infinite-dimensional nature of PDECO problems all present significant challenges for general-purpose optimization algorithms and require special attention in the handling of regularization, iterative solvers, preconditioning, globalization, management of inexactness, sensitivity calculations, and parallel implementation, all of which need tailoring to the structure of the

*Department of Mathematics and Statistics, Texas Tech University, Lubbock, Texas, USA. kevin.long@ttu.edu

[†]Sandia National Laboratories, Livermore, California, USA. ptboggs@sandia.gov

[‡]Sandia National Laboratories, Albuquerque, New Mexico, USA. 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)

underlying operators. PDECO methods require not only considerable information from the simulator (or model) but also require the manipulation of various linear algebra objects that need to be embedded in the model.

In this paper we present a software design and code capability that elegantly and efficiently enables PDECO algorithms. To set the stage, in section 2 we examine several approaches to solving the PDECO problem (1) – (3). In section 3 we describe such a software system whose syntax allows a natural expression of PDECO problems and seamless way to connect to an optimizer. In addition, we present the use of multiple Trilinos packages ranging from linear solvers to distributed linear object capabilities as part of our software infrastructure. In section 4 we give a collection of examples that illustrate the power of our system. In Section 5 we discuss the current state and future directions for this work.

2 PDE-Constrained Optimization Overview

In this section, we consider two approaches for solving constrained optimization problems (1) –(3). The first approach solves the PDE $\mathbf{c}(\mathbf{u}, \boldsymbol{\alpha}) = \mathbf{0}$ at each step of the optimization process and thus the iterates are always feasible. (We refer to these methods as “feasibility preserving methods.”) These methods are based on the fact that, in some cases, one only has an existing code for solving the PDE and this code is not easily modified, or only can be only “slightly” modified. The second approach only requires that the PDE constraints be satisfied in the limit as the optimization process converges. We refer to these methods as “all-at-once” methods since we attempt to solve the optimization problem and the PDE constraint at the same time. This approach requires significant intrusion into the PDE solver so that it can be tightly coupled to the optimizer. Not surprisingly, each of these two strategies has variants, as we discuss below.

First, however, we note that our discussion below is predicated on the assumption that the PDECO problem has sufficient smoothness for gradient-based methods to be applicable; if not, then one has little choice but to use a derivative-free method; such methods can only be realistically considered for problems with very few design variables and even in this case, they often converge very slowly. For the remainder of this paper, we will assume that the functions are all sufficiently differentiable.

2.1 Feasibility-Preserving Methods

The methods for solving PDECO that solve the state equation at each iteration have two main variants. The first is the “black-box” procedure in which no modifications to the PDE solver can be made. The code can only evaluate the state variables given an instance of the design variables as input. They completely separate optimization strategies from the model by communicating results through loosely coupled interfaces, e.g., through the file system. Furthermore, sensitivity information is typically unavailable from the underlying dynamics equation and consequently, the objective function gradients are acquired by finite differencing the entire forward simulation. Although less efficient, these methods are nevertheless able to employ some powerful optimization methods, provided the number of design variables is small. In addition, black-box methods have the distinct advantage of having a very simple interface between the optimizer and the PDE simulator, and so are sometimes a practical choice.

As mentioned above, an implicit assumption in (1) – (3) is that for any (reasonable) value of the design variables, the underlying model can be solved for the state variables. Thus, ignoring the inequality constraints to simplify the algorithmic presentation, we can perform a nonlinear elimination on the equality constraints, i.e., solve the equality constraints for $\mathbf{u}(\boldsymbol{\alpha})$ to obtain an unconstrained optimization problem of the form $\min_{\boldsymbol{\alpha}} \mathcal{J}(\boldsymbol{\alpha})$. A local minimizer may be obtained by applying a variant of Newton’s method (see, e.g., [17]) given by

$$\boldsymbol{\alpha}^{k+1} = \boldsymbol{\alpha}^k - \sigma^k (B^k)^{-1} \nabla \mathcal{J}(\boldsymbol{\alpha}^k),$$

where the superscripts denote the iteration number, B is an approximation to the Hessian of the objective function evaluated at the k^{th} iterate, and σ^k is a step length parameter appropriately chosen. The convergence theory for such methods is well developed.

As noted above, the primary disadvantage of this approach is in its inability to consider large number of optimization variables and guarantee accuracy. If there are n design variables, then n forward simulations are required for each finite difference gradient calculation and this must be done at each iteration. For n large, e.g., a design variable at each point of the computational domain, the finite difference method becomes computationally intractable. Furthermore, even when n is small the accuracy of finite difference derivatives may be low because the forward simulations are themselves subject to discretization errors and inexact solves, whose effect is amplified by differencing.

We conclude this part by providing two alternatives to finite differencing that, although much more efficient, are still not as efficient as the all-at-once methods described below. These are the direct sensitivity and adjoint methods. Note that both require some intrusion into the state-equation solver.

We again use just the equality constrained version to simplify the presentation. Applying the chain rule to the objective function, we have

$$\nabla \mathcal{J} = \frac{\partial \mathcal{J}}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \boldsymbol{\alpha}} + \frac{\partial \mathcal{J}}{\partial \boldsymbol{\alpha}} \quad (4)$$

for the reduced gradient ∇J . Similarly, for $\mathbf{c}(\mathbf{u}(\boldsymbol{\alpha}), \boldsymbol{\alpha}) = \mathbf{0}$ we have

$$\frac{\partial \mathbf{c}}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \boldsymbol{\alpha}} + \frac{\partial \mathbf{c}}{\partial \boldsymbol{\alpha}} = \mathbf{0}. \quad (5)$$

Since $\frac{\partial \mathbf{c}}{\partial \mathbf{u}}$ is invertible, we combine (4) and (5) to obtain

$$\nabla \mathcal{J} = - \frac{\partial \mathcal{J}}{\partial \mathbf{u}} \frac{\partial \mathbf{c}}{\partial \mathbf{u}}^{-1} \frac{\partial \mathbf{c}}{\partial \boldsymbol{\alpha}} + \frac{\partial \mathcal{J}}{\partial \boldsymbol{\alpha}}. \quad (6)$$

The ‘‘direct sensitivity matrix’’ $\frac{\partial \mathbf{u}}{\partial \boldsymbol{\alpha}} = \frac{\partial \mathbf{c}}{\partial \mathbf{u}}^{-1} \frac{\partial \mathbf{c}}{\partial \boldsymbol{\alpha}}$ requires the solution of the state Jacobian against a right hand side with multiple columns equal to the number of optimization variables. Although the direct sensitivity matrix offers more efficient and exact gradient calculations, if the number of optimization variables is sufficiently large this method also becomes computationally intractable. Fortunately, a simple transformation is possible to avoid the computational expense associated with the dependence of the optimization variable. By shifting $\frac{\partial \mathcal{J}}{\partial \mathbf{u}} \frac{\partial \mathbf{c}}{\partial \mathbf{u}}^{-1}$ and taking the transpose, the dependence on the multiple right hand sides is now avoided. This transformation is termed the ‘‘adjoint based sensitivity’’ method and the gradient is calculated as:

$$\nabla \mathcal{J} = \frac{\partial \mathbf{c}}{\partial \mathbf{u}}^{-T} \frac{\partial \mathcal{J}}{\partial \mathbf{u}} \frac{\partial \mathbf{c}}{\partial \boldsymbol{\alpha}} + \frac{\partial \mathcal{J}}{\partial \boldsymbol{\alpha}}. \quad (7)$$

Note that we already have $\frac{\partial \mathbf{c}}{\partial \mathbf{u}}$ since it is needed to solve the the constraint. Indeed, in the case of a linear constraint, it’s just the linear operator. Often, the other derivatives are not hard to obtain. The implementation of adjoint based sensitivities provides the primary prerequisite towards solving PDECO problems with large design variables. In the following section, we discuss all-at-once methods which rely on adjoints and offer additional computational improvements.

2.2 The All-at-Once Approach

The methods described here offer further computational improvements by tightly coupling the convergence of the state and optimization calculations. That is, there is no need to solve the nonlinear PDEs exactly until the optimization converges. The classical way to approach this problem is to introduce Lagrange multiplier fields, $\boldsymbol{\lambda}$, known as the *adjoint states* or *costate* variables, and form a Lagrangian functional \mathcal{L} that incorporates the PDE constraints via an inner product with $\boldsymbol{\lambda}$. In particular, let \mathbf{c} be the PDE constraint and assume that the initial conditions are included. Then we can write the Lagrangian as

$$\mathcal{L}(\boldsymbol{\alpha}, \mathbf{u}, \boldsymbol{\lambda}) = \mathcal{J}(\mathbf{u}, \boldsymbol{\alpha}) + \langle \boldsymbol{\lambda}, \mathbf{c} \rangle.$$

One then requires stationarity of \mathcal{L} with respect to the state variables (\mathbf{u}), decision variables ($\boldsymbol{\alpha}$), and adjoint variables ($\boldsymbol{\lambda}$). Taking variations, the following system of equations is derived, representing the first-order necessary conditions for optimality:

$$\mathcal{L}_{\boldsymbol{\lambda}}(\boldsymbol{\alpha}, \mathbf{u}, \boldsymbol{\lambda}) = \mathbf{c}(\mathbf{u}, \boldsymbol{\alpha}) = \mathbf{0} \quad \text{state equation} \quad (8)$$

$$\mathcal{L}_{\mathbf{u}}(\boldsymbol{\alpha}, \mathbf{u}, \boldsymbol{\lambda}) = \mathcal{J}_{\mathbf{u}}(\hat{\mathbf{u}}, \boldsymbol{\alpha}) + \mathbf{c}_{\mathbf{u}}(\mathbf{u}, \boldsymbol{\alpha}) = \mathbf{0} \quad \text{adjoint equation} \quad (9)$$

$$\mathcal{L}_{\boldsymbol{\alpha}}(\boldsymbol{\alpha}, \mathbf{u}, \boldsymbol{\lambda}) = \mathcal{J}_{\boldsymbol{\alpha}}(\mathbf{u}, \boldsymbol{\alpha}) + \mathbf{c}_{\boldsymbol{\alpha}}(\mathbf{u}, \boldsymbol{\alpha}) = \mathbf{0} \quad \text{decision equation} \quad (10)$$

where the subscripts denote taking variations of the particular functional with respect to the subscript. When appropriately discretized on the current grid level, the dimension of each of \mathbf{u} , $\boldsymbol{\lambda}$ is equal to the number of grid points N_g multiplied by number of time steps (in time-dependent problems) N_t , $\boldsymbol{\alpha}$ is of dimension N_g , and thus the system (8) – (10) is of dimension $N_g N_t + N_g$. This can be very large for problems of interest—for example, in numerical problems presented in [1], the system contains 3.4×10^9 unknowns. The time dimension cannot be “hidden” with the usual time-stepping procedures, since (8) – (10) couples the initial and final value problems through the decision equation. This is not shown here, but the optimality system is thus a boundary value-problem in 4D space–time.

If one considers (8) – (10) as a nonlinear system of equations to be solved by Newton’s method, one will have to solve a linearized system at each iteration. It has been noted that the linearized system corresponds to a constrained optimization problem that is a quadratic approximation to the objective function with linearized constraints. In the optimization literature, this goes by the name sequential quadratic programming, or SQP. See [4] for a discussion of SQP methods in finite dimensions. The system (8) – (10) is known in the optimization literature as the Karush-Kuhn-Tucker (KKT) conditions.

Since the coupled optimality system can be formidable to solve simultaneously, a popular alternative is to eliminate state and adjoint variables and thereby reducing the system to a manageable one in just the decision variable. Methods of this type are known as *reduced space* methods. A *nonlinear elimination* or *nonlinear Gauss-Seidel* variant of a reduced space method proceeds as follows for the KKT system. Given $\boldsymbol{\alpha}$ at some iteration, solve the *state equation* for the state variable \mathbf{u} . Knowing the state then permits solution of the *adjoint equation* for the adjoint variables $\boldsymbol{\lambda}$ and $\hat{\mathbf{p}}$. Finally, with the state and adjoint known, the decision variable $\boldsymbol{\alpha}$ is updated via an appropriate linearization of the *decision equation*. This loop is then repeated until convergence. This procedure is demonstrated as a solution mechanism in our numerical result section. Historically, reduced space methods have been attractive because solving the subsets of equations in sequence exploits the state/adjoint/decision structure of the optimality system and capitalizes on well-established methods and software for solving the state equation. In addition, adjoint PDE solvers are becoming more popular, due to their role in goal-oriented error estimation and efficient sensitivity computation, so they can be exploited as well.

In contrast to reduced space methods, *full space* methods solve for the state, decision, and adjoint variables simultaneously. For large-scale problems, this is typically effected via Newton-Krylov iteration. That is, the linear system arising from the KKT systems at each Newton iteration is solved using a Krylov iterative method. The difficulty of this approach is the complex structure, indefiniteness, and ill-conditioning of the KKT system, which in turn requires effective preconditioning. Similar to the reduced space methods, our software accommodates full space methods with similar ease as shown in the numerical results section.

2.3 Discussion

Numerical evidence suggests that for steady-state PDE-constrained optimization problems, full-space methods can outperform reduced space methods by a wide margin. For optimization of systems governed by time-dependent PDEs, the answer is not as clear. The nonlinearities within each time step of a time-dependent PDE solve are usually much milder than for the corresponding stationary PDEs, so amortizing the nonlinear PDE solve over the optimization iterations is less advantageous. Moreover, time dependence results in large storage requirements for full-space methods, since the full space optimality system becomes a boundary value problem in the space–time cylinder. For such problems, reduced space methods are often preferable.

In their survey of approaches to PDECO, van Bloemen Waanders *et al.* [21] laid out a hierarchy of methods ranging from a black-box approaches to the all-at-once approaches describe above. They found that for more than ~5-10 design variables the more intrusive algorithms become more efficient than black-box by many orders of magnitude. Despite this clear performance advantage, intrusive algorithms present several difficulties for the prospective user. First, one must compute certain operators not usually available from off-the-shelf simulators. Second, the PDE solver and the optimizer must interact directly, often in ways more complex than the simple master-slave relationship used in a black-box method. Given the advantages of the more intrusive approaches for very large problems, we now consider the software implications and introduce Sundance.

3 Sundance

In their survey of PDECO methods, [21] a high level software vision was outlined for specifying and solving PDECO problems; a draft version of that software was described in [21] and in [2]. The outgrowth of that work was a full-featured finite-element toolkit called Sundance, designed from the ground up with the intention that it be used in the context of

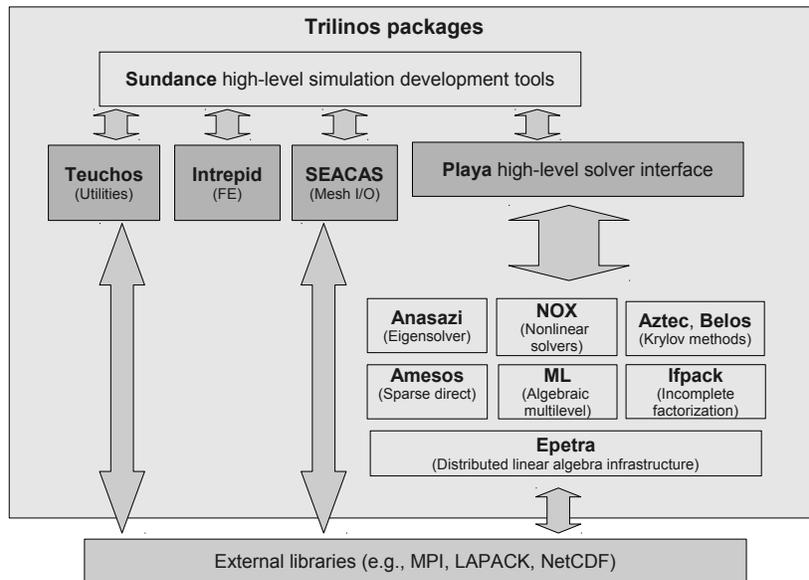


Figure 1: Schematic of relationship between Sundance and other Trilinos packages. The grey box delimits the “universe” of Trilinos packages. Sundance is a single Trilinos package, and is shown at the top. The four packages shown in dark grey (Playa, Teuchos, Intrepid, and SEACAS) are those Trilinos packages with which Sundance interacts directly. Interaction with other packages, for example, the NOX nonlinear solvers and the Epetra distributed linear algebra infrastructure, is indirect and mediated by the Playa high-level solver interface. Interaction with third parties is indirect and mediated by Teuchos, SEACAS, and Epetra.

embedded algorithms for PDECO and uncertainty quantification. Sundance is implemented in C++, with 3D capabilities, fully parallel, and is built upon tools and solver components of the Trilinos library [10]. An overview of Sundance can be found in the foundational paper [15].

There are a number of similar efforts that produce high-performance simulators from high-level specification or PDEs. See [14] and [15] for literature surveys.

3.1 Interoperation with other Trilinos packages

In its role as a simulation development toolkit, the Sundance package must interoperate with many different Trilinos packages that provide services for concrete parallel linear algebra representations, linear and nonlinear solvers, and preconditioners. As Trilinos is a growing and changing system, we expect that the packages used by Sundance will change; for example, as Epetra is phased out in favor of Tpetra, Sundance will have to be at least partially templated. To keep the interaction manageable and extensible, Sundance is interfaced directly with only a small number of Trilinos packages: the Teuchos utilities, the Intrepid low level finite element package, the SEACAS mesh I/O utilities, and the Playa high-level linear and nonlinear algebra objects. The current state of the interface is shown in figure 3.1. Centrally important to Sundance’s interoperation with other Trilinos packages is one package, Playa [11], which provides Sundance with a single point of contact for the various vector, operator, and solver types available through Trilinos.

3.2 Differentiation as a unifying principle

The central idea behind the design of Sundance is the realization that differentiation makes plain the association between coefficient expressions and basis functions. Differentiation thereby enables the binding of computational kernels for coefficients with computational kernels for basis functions and integration. This is the case even in the context of a linear forward PDE, for which one does not ordinarily consider derivative computation to be necessary; in Sundance, automatic differentiation is used in the discretization of every PDE. This system enables runtime coordination of very efficient matrix and vector assembly starting from a high-level specification of the problem’s weak form, using in-place automatic

differentiation to compute the required derivatives. By “in-place” automatic differentiation we mean that derivatives are evaluated concurrently with function values during traversal of an untransformed expression graph; this should not be confused with symbolic differentiation or with automatic differentiation by source transformation. This concept and a high-level view of its implementation are described more fully in [15]. In that same paper, performance results are presented which indicate that the runtime assembly algorithms used by Sundance in fact often outperform both hand-tuned simulators and simulators based on code generation.

Most pertinent to this paper is the central, and unifying, role of differentiation in the design of Sundance. Because Sundance uses differentiation to process every weak form, the tools necessary for computation of gradients and Hessians are built into the core design. In the present paper we will emphasize the user-level features through which problem specification and derivative specification are used to set up a PDECO algorithm. Because the same functionality for the forward model setup can be used to differentiate a Lagrangian in PDECO, a focus on the design of the forward problem in the next section is sufficient to explain the optimization capabilities. The core design will be demonstrated on optimization in several numerical examples.

3.3 Overview of object architecture

The entire Sundance toolkit contains many classes and nonmember functions, about two dozen of which might commonly appear in user-level code. To impose some organization on that collection we will first group the user-level objects into three categories:

- **Problem specification building block objects** - these are objects out of which problem specifications are assembled. Examples include objects to deal with the meshing (`Mesh`), general utility objects (`Expr`), objects to identify subsets of the computational domain (`CellFilter`), and a family of finite element basis functions (`BasisFamily`). We can further subdivide these objects into those relating to symbolic geometry, discrete geometry, discretization specification, and symbolic expressions.
- **Problem specification objects** - these are objects that encapsulate a problem along with instructions for its discretization. Problem specification objects produce algorithm interface objects or perhaps other problem specification objects.
- **Algorithm interface objects** - these are objects that interact directly with solvers or optimizers. These are actually objects from the Playa package, not the Sundance package, but we include them in the discussion to illustrate the role of the problem specification objects as producers of objects that interact directly with algorithms.

These categories are shown graphically in Figure 2. In subsequent diagrams of program flow, we will refer back to this categorization.

The problem specification building blocks have subclasses. Sundance uses the reference-counted handle idiom ([11] for discussion of this idiom in the context of Playa) to provide polymorphism and safe memory management along with value syntax. The actual relationship between a handle (`Expr`), a pointer to a base class (`ExprBase`), and a derived class (`CoordExpr`) is as shown in the left side of figure 3. Logically, however, what matters is the relationship between the handle and the derived class; the presence of the base class is an implementation detail irrelevant to the user. For simplicity, we omit the actual base class and regard the handle as playing the role of a base of the inheritance diagram, as shown in the right side of figure 3. In some cases, there may be intermediate derived types between a base class and a final derived type; these are also invisible to an end user so we will omit such intermediaries from this discussion.

All of the building block objects have two or more subclasses; the object whose subclasses play the largest role in our examples is class `Expr`, whose subclasses represent different types of mathematical expressions, for example, test functions, products, or coordinate functions. A listing of several of the user-level subtypes is shown in figure 4 in the form of a UML inheritance diagram.

Having established a categorization of objects and a shorthand for discussion of handled inheritance hierarchies, we can now outline how these objects are typically used to construct a sequence of objects leading ultimately to Playa objects that can be used in a solver or optimizer. Figure 5 shows the construction of a term in a weak form from its components that specify the region of integration, the integrand, and the method of quadrature. Each of these is represented by one of the building block objects: `CellFilter` for the region of integration, `Expr` for the integrand, and `QuadratureFamily` for the method of quadrature. The result is an `Expr` representing the term in a weak form; because this is an `Expr`, it may be added to other weak forms. Once the weak form and boundary condition expressions are put together, a user

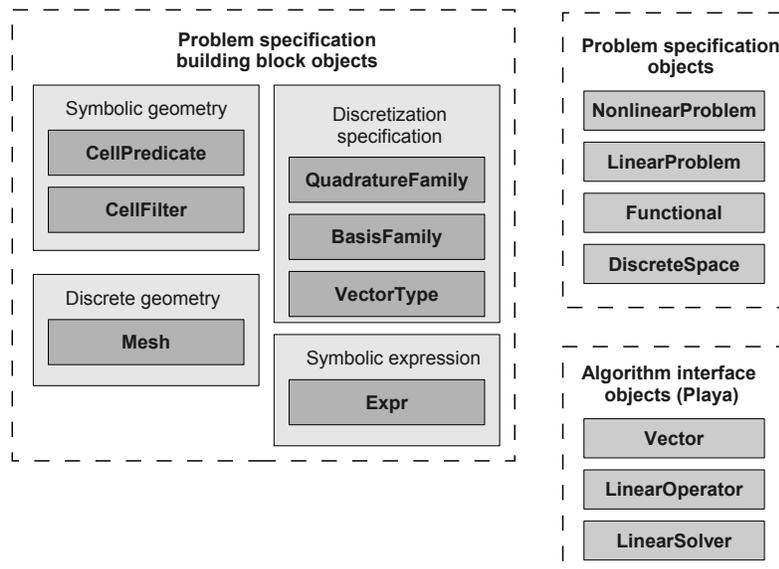


Figure 2: Classification of user-level Sundance classes into the categories of problem specification building blocks, problem specifications objects, and algorithm interface objects. The problem specification building blocks are further subdivided into those relating to symbolic geometry, discrete geometry, discretization specification, and symbolic expressions.

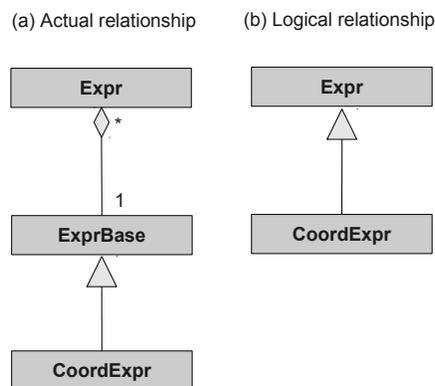


Figure 3: UML object diagram showing (a) the actual relationship between a handle class, a polymorphic base class, and a subclass, and (b) the logical relationship with the implementation detail of the base class suppressed.

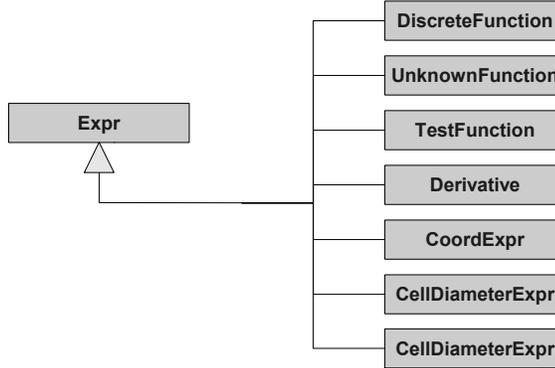


Figure 4: UML object diagram showing several (logical) subclasses of the Expr class.

can construct a problem specification of appropriate type. Figure 6 shows the construction of a `NonlinearProblem` object from a specification of the problem’s mesh, weak form, essential boundary conditions, list of test functions, list of unknown functions, expression for initial guess, and specification of the type of low-level linear algebra representation to be used. Finally, figure 6 shows the production of Playa operator and vector objects as requested by a solver algorithm.

In some cases, a problem specification object does not produce Playa algorithm interface objects directly, but instead produces two or more problem specification objects. For example, in figure 7 building blocks are used to construct a `Functional` object. Member functions of the `Functional` then produce the `NonlinearProblem` or `LinearProblem` objects resulting from computing variations with respect to certain specified functions (these functions being represented as `Expr` objects). This example is of particular importance in optimization, where the functional represents a Lagrangian and the nonlinear and linear problems it produces are the state and adjoint equations, respectively.

3.4 Example: constructing a nonlinear forward problem

To place the software objects in the setting of a concrete problem, we show an example of setting up a simple forward problem. Let V^h be a space of piecewise linear functions on some meshing of $[0, 1]$, and consider the problem of finding $u \in V^h$ such that

$$\int_0^1 v' u' + v x e^u - v g(x) dx = 0 \quad \forall v \in V^h \tag{11}$$

with boundary condition $u(0) = u(1) = 0$. This is a variant of Bratu’s problem [5] and of Toomre’s problem [20]. To produce an exactly solvable problem, we use the method of manufactured solutions [18, 19] (MMS). Choosing the solution $u(x) = x(1 - x)$ gives the forcing function $g(x) = x e^{x(1-x)} + 2$.

The weak form requires a region of integration, and the boundary conditions require some specification of where they are to be applied. The `CellFilter` class is used to identify geometric regions; during discretization, a cell filter acts to select certain cells according to some criterion, for example, all cells of a specified dimension. Cell filters can also be specified in terms of predicate functions; in the code fragment below, the zero-dimensional cells are further filtered by predicate functions that select the points at $x = 0.0$ and $x = 1.0$.

```

CellFilter omega = new MaximalCellFilter();
CellFilter pts = new DimensionalCellFilter(0);
CellFilter left = pts.subset(new CoordinateValueCellPredicate(0, 0.0));
CellFilter right = pts.subset(new CoordinateValueCellPredicate(0, 1.0));
  
```

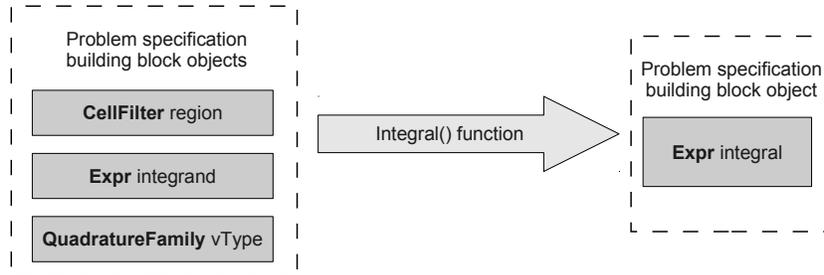


Figure 5: Diagram indicating how the `Integral` non-member function accepts a symbolic geometry object (type `CellFilter`), a symbolic integrand (type `Expr`), and a specification of quadrature method (type `QuadratureFamily`) to produce an object representation (type `Expr`) of a weak form.

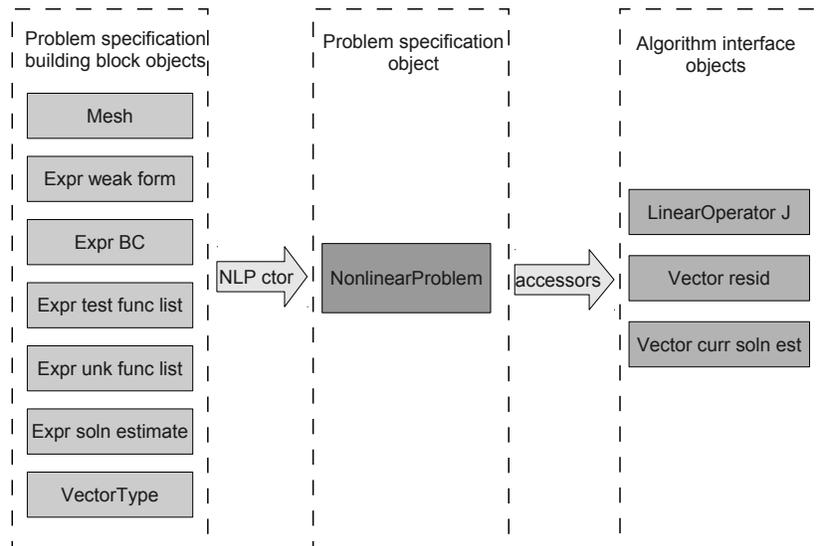


Figure 6: Example of program flow from building block components, to problem specification object for a nonlinear PDE, to Playa linear algebra objects that can be used in a nonlinear solver.

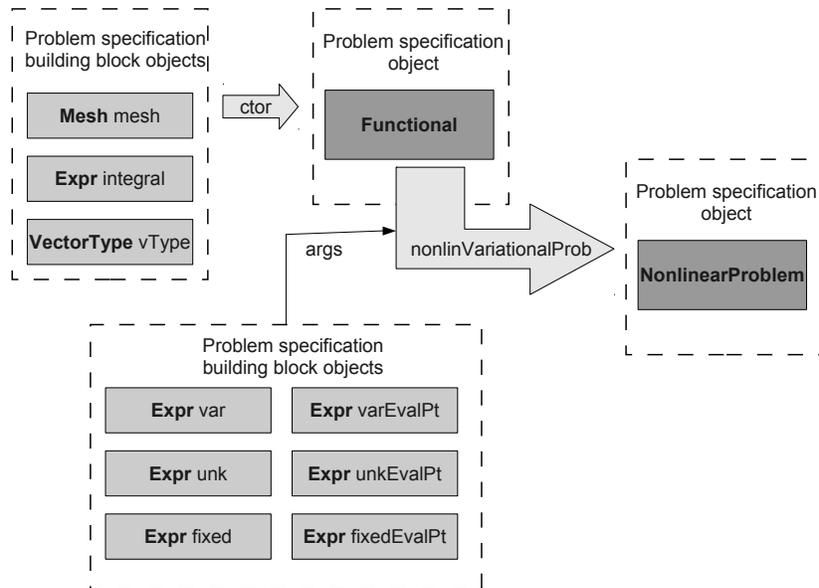


Figure 7: Example of program flow from building block components to construction of a functional, followed by taking variations with respect to specified functions to obtain a nonlinear problem.

The symbolic objects composing the integrand must also be constructed. In the next code fragment, we define test and unknown functions v and u and specify that they use first-order Lagrange basis functions. Expressions for the coordinate function x and the differentiation operator are also defined; the argument “0” specifies the first coordinate direction. With these components ready, the integrand can be formed using overloaded operators. Finally, a quadrature rule must be chosen; to give reasonable accuracy on the nonlinear term we use fourth-order Gauss-Legendre quadrature.

```
BasisFamily basis = new Lagrange(1);

Expr v = new TestFunction(basis);
Expr u = new UnknownFunction(basis);

Expr x = new CoordExpr(0);
Expr dx = new Derivative(0);

QuadratureFamily quad = new GaussianQuadrature(4);

Expr weakForm = Integral(omega, (dx*v)*(dx*u) + v*x*exp(u) - g*v, quad);
```

The boundary conditions are set up in a similar manner. There are a number of ways to specify Dirichlet boundary conditions with Sundance objects. Nitsche’s method may be used, in which case, the appropriate expressions are formed and added to the weak form. Here we use the simple method of “replacing” the rows associated with boundary degrees of freedom by equations that impose the boundary conditions. That these expressions are to replace the weak form for the specified rows is indicated by using them in an `EssentialBC` function rather than an `Integral` function; otherwise, the specification of weak forms and replacement BCs is identical. Multiplication by a test function is used to allow the user to indicate which rows are to be replaced. The code to define the Dirichlet boundary conditions is shown here.

```
Expr bc = EssentialBC(left + right, v*u, quad);
```

We’ve now specified the weak form and boundary conditions in what one might call quasi-symbolic form: symbolic expressions annotated by specification of basis functions and quadrature rules. To be ready to produce discrete objects, a mesh must be defined and a low-level linear algebra representation must be chosen. Meshes are obtained through an abstract `MeshSource` interface, subclasses of which might do on-the-fly building of simple meshes or reading from mesh files. For a simple one-dimensional mesh we build on the fly with

```

MeshType meshType = new BasicSimplicialMeshType();
/* Mesh the interval [0,1] with 16 elements */
int nx = 16;
MeshSource mesher = new PartitionedLineMesher(0.0, 1.0, nx, meshType);
Mesh mesh = mesher.getMesh();

```

The selection of a subclass of `Playa:VectorType` controls what type of linear algebra objects will be built; here, we choose `Epetra`.

```

VectorType<double> vecType = new EpetraVectorType(); // Use Epetra objects

```

The next code fragment shows the construction of the problem's discrete space and the discrete function that represents the initial guess for the solution.

```

DiscreteSpace discSpace(mesh, basis, vecType);
Expr u0 = new DiscreteFunction(discSpace, 0.0);

```

Everything needed to define the problem is now in place, so we construct a `NonlinearProblem` object.

```

NonlinearProblem prob(mesh, weakForm, bc, v, u, u0, vecType);

```

The nonlinear problem class provides member functions to compute the problem's Jacobian and residual, and also to obtain a vector representation of the current state. One could use those functions to write an adapter allowing the use of `NonlinearProblem` with a user's desired nonlinear solver library. Because Trilinos already provides a full-featured nonlinear solver package, `NOX`, `NonlinearProblem` also has a `solve()` member function that accepts a `NOX` solver object as an argument and carries out the solve. The result is written into the discrete function `u0` that was used in the construction of the `NonlinearProblem`. Definition of the `NOX` solver object and the solution of the problem is shown in this code fragment.

```

ParameterXMLFileReader reader("nox-amesos.xml");
ParameterList noxParams = reader.getParameters();
NOXSolver nonlinSolver(noxParams);

prob.solve(nonlinSolver);

```

Results are shown in figure 8.

3.5 The objective function interface

Once the functional has been defined, the procedure of setting up and using the problems required for a reduced-space formulation of a nonlinear PDECO problem is largely independent of the specific form of the functional and can be neatly encapsulated in a further set of driver objects. These are the `LinearPDEConstrainedObj` and `NonlinearPDEConstrainedObj` objects, which represent differentiable objective functions in the reduced space. These objects manage internally the sequence of solving the state and adjoint equations and then computing the gradient. These two classes implement a very lightweight objective function interface, `Playa:ObjectiveBase` that can be adapted for use with nonlinear solvers such as `NOX` or gradient-based optimization libraries such as `MOOCHO`.

3.5.1 Independent or sequential constraints

In some problems, certain of the constraint equations and their associated state variables may either be completely decoupled, or perhaps coupled in a sequential way. An example of a set of uncoupled constraints and states arises in multifrequency inversion, where the responses at different frequencies are mutually independent. Sequential coupling

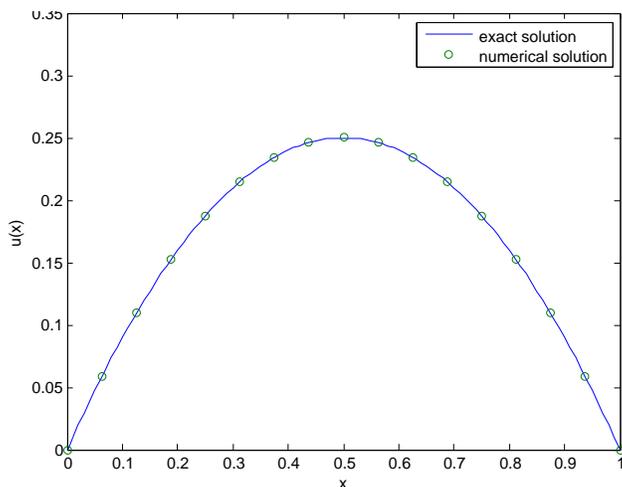


Figure 8: Comparison of exact and numerical solutions to the nonlinear forward problem.

arises, for example, in passive advective transport where the equation for the velocity is independent of the concentration. In such cases, it is often more efficient to solve the smaller systems. To enable this performance optimization, the constructors for the linear and nonlinear `PDEConstrainedObj` objects allows specification of sequences of state and adjoint variables. This is taken as a directive that the state equations are to be solved in the specified order, and the adjoint equations in the reverse order.

In principle, these relationships could be deduced automatically from the symbolic problem specification. In the current implementation of Sundance it is the user’s responsibility to provide the correct dependency ordering; automation of this step is planned in a future version.

4 Model Problems

To illustrate the use of Sundance components to program and solve PDE-constrained optimization problems, we develop several simple model problems. To keep the focus on the software objects used to set up the problems, we use very simple solution algorithms.

We show results to verify accuracy, but because we have made no attempt to tune the solvers or optimizers, and because of the difficulty of programming these problems without Sundance, we present no timing results. For timings of Sundance on forward problems compared to several other codes and for parallel scalability results, we refer the reader to [15].

4.1 Linear source inversion

Our first model problem is source inversion for the Poisson equation on a washer-shaped domain. The state, adjoint, and full KKT equations are all linear. The goal of the problem is to select a source in order to match a specified target function. We use the method of manufactured solutions [18, 19] to construct a problem yielding an exact solution with simple form. With Tikhonov regularization on the design variable α , the optimization problem is

$$\min_{u, \alpha} f(u, \alpha) = \frac{1}{2} \int_{\Omega} (u - u^*)^2 d\Omega + \frac{R}{2} \int_{\Omega} (\nabla \alpha)^2 d\Omega \quad (12)$$

$$\text{subject to } \begin{cases} \nabla^2 u = \alpha + g & \text{in } \Omega \\ u = 0 & \text{on } \Gamma_{\text{inner}} \\ u = \cos(\theta) \sin(\theta) & \text{on } \Gamma_{\text{outer}} \\ \frac{\partial u}{\partial n} = 0 & \text{on remaining surfaces} \end{cases} \quad (13)$$

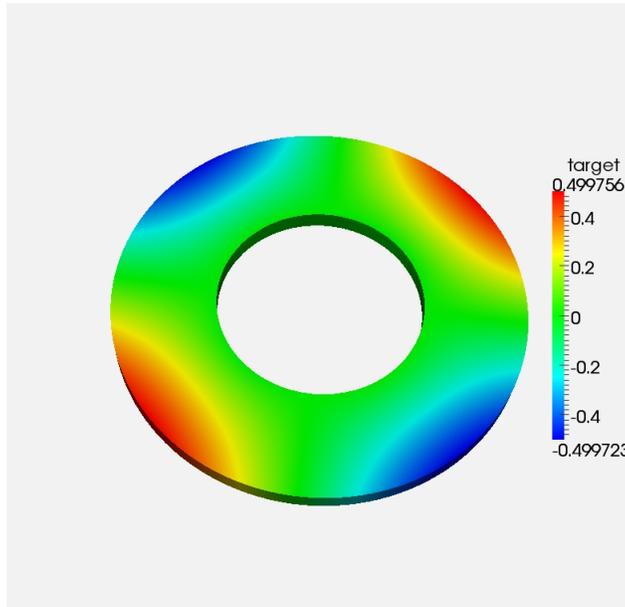


Figure 9: Target function u^* for the Poisson source inversion problem on a washer-shaped region. The outer ring of the washer is labeled Γ_{outer} , the inner ring is Γ_{inner} .

Here R is a regularization parameter, and u^* and g are functions to be defined below. Introducing a multiplier variable λ , the problem's Lagrangian is

$$L = f + \int_{\Omega} [\nabla \lambda \cdot \nabla u + \lambda (\alpha + g)] d\Omega. \quad (14)$$

The manufactured target function u^* and exact solution u are, in cylindrical coordinates,

$$u^*(r, \theta, z) = [2r - 1 + ((4 - 9r)R)/r^2] \cos \theta \sin \theta \quad (15)$$

and

$$u(r, \theta, z) = (2r - 1) \cos(\theta) \sin(\theta). \quad (16)$$

With these the forcing function g is

$$g = (1/(600r^2))(-100r^6 + 360r^5 - 150 \log(r)r^4 + (-273 + \log(1024))r^4 - 3600r + \log(1024) + 2392) \cos \theta \sin \theta. \quad (17)$$

The domain and the target function are shown in figure 9.

4.1.1 Programming the functional

The bulk of the work for the programmer is in setting up the `Functional` object for the Lagrangian.

```

/* Choose the vector type */
VectorType<double> vecType = new EpetraVectorType();

/* Create the mesh */
MeshType meshType = new BasicSimplicialMeshType();
MeshSource meshSrc = new ExodusMeshReader("concCylinder3D-0", meshType);
Mesh mesh = meshSrc.getMesh();

/* Create the symbolic geometry. Mesh labels are assumed to have been
 * assigned by the mesher */
CellFilter interior = new MaximalCellFilter();
CellFilter faces = new DimensionalCellFilter(2);
CellFilter inner = faces.labeledSubset(2);

```

```

CellFilter outer = faces.labeledSubset(1);

/* Create the unknown functions */
BasisFamily basis = new Lagrange(1);
Expr u = new UnknownFunction(basis, "u");
Expr lambda = new UnknownFunction(basis, "lambda");
Expr alpha = new UnknownFunction(basis, "alpha");

/* Set up the symbolic expressions */
Expr dx = new Derivative(0);
Expr dy = new Derivative(1);
Expr dz = new Derivative(2);
Expr grad = List(dx, dy, dz);

Expr x = new CoordExpr(0);
Expr y = new CoordExpr(1);
Expr z = new CoordExpr(2);

Expr r = sqrt(x*x + y*y);
Expr cosT = x/r;
Expr sinT = y/r;

const double pi = 4.0*atan(1.0);
double R = 1.0e-2;

Expr target = uExact + R*(4.0-9.0*r)/r/r*cosT*sinT;
Expr g = (2392 - 3600*r + 360*pow(r,5) - 100*pow(r,6)
  + pow(r,4)*(-273 + log(1024)) + log(1024) - 150*pow(r,4)*log(r))
  / (600.*pow(r,2));

/* Create the Lagrangian */
QuadratureFamily quad = new GaussianQuadrature(4);

Expr f = Integral(interior, 0.5*pow(u-target, 2.0), quad, watch);
Expr reg = Integral(interior, 0.5*R*(grad*alpha)*(grad*alpha), quad);

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

Expr BC = EssentialBC(outer, lambda*(u-cosT*sinT), quad)
  + EssentialBC(inner, lambda*u, quad);

Functional L(mesh, f+reg+constraint, BC, vecType);

```

With the Lagrangian encapsulated as a `Functional` object, we can set up our choice of reduced-space or full-space formulations.

4.1.2 Solution by the full-space method

A `LinearProblem` representation of the full KKT equations is obtained by taking variations of the Lagrangian.

```

DiscreteSpace discSpace(mesh, List(basis, basis, basis), vecType);
Expr w0 = new DiscreteFunction(discSpace, 0.0);

Expr dum;

LinearProblem KKT_Prob = L.linearVariationalProb(List(lambda,u,alpha),
  w0, List(lambda, u, alpha), dum, dum);

```

At this point, the system's matrix and right-hand side can be obtained as `Playa LinearOperator` and `Vector` objects through member functions of `LinearProblem`. Alternatively, the `solve()` member function of `LinearProblem` can be called, with a `Playa LinearSolver` argument to specify the solve algorithm to be used.

```
LinearSolver<double> linSolver
  = LinearSolverBuilder::createSolver("amesos.xml");
Expr solnFS = prob.solve(linSolver);
```

The `solve()` function returns the solution in the form of a `DiscreteFunction`.

4.1.3 Programming an adjoint gradient method

Much of the code for this problem is identical to that for the SAND case shown above. The principal difference is that instead of setting up a `LinearProblem` object that encapsulates the KKT equations, we set up a `Functional` object representing the Lagrangian. The linear problem is then produced by taking variations as in the initial example above; in this example that step is done automatically by the `LinearPDEConstrainedObj` object.

```
DiscreteSpace ds2(mesh, basis, vecType);

Expr u0 = new DiscreteFunction(ds2, 0.0);
Expr lambda0 = new DiscreteFunction(ds2, 0.0);
Expr alpha0 = new DiscreteFunction(ds2, 0.0);

RCP<PDEConstrainedObjBase> obj
  = rcp(new LinearPDEConstrainedObj(L, u, u0, lambda, lambda0, alpha, alpha0,
    linSolver, verb));
```

The `linSolver` argument is used to specify the solve algorithm used for the state and adjoint equations. This objective function object can then be used in a gradient-based optimizer.

Notice a subtle but significant difference from the specification of the full-space method: there, a single discrete function `w0` was defined on the full space `discSpace`, but here we create three discrete functions `u0`, `lambda0`, `alpha0` each on the reduced space.

4.1.4 Numerical results

We solved this problem using both the full space and reduced space formulations. The same code for the Lagrangian was used for both cases. We used the KLU sparse direct solver from the Amesos package for all linear solves arising in either formulation. The mesh used had 15255 elements and 3829 nodes. In the reduced space calculations, we used a limited-memory BFGS [16, 17] (LM-BFGS) algorithm with line search. In all calculations, the initial estimate of the Hessian was the identity and the initial estimate of the design variable was zero. Stopping tolerances were 10^{-7} in objective function value, 10^{-6} in gradient norm, and 10^{-4} in step. When all three tolerances have been attained the problem is considered to have converged.

| R | Reduced space | | | Full space | |
|--------|----------------------------|---------------|---------------|----------------------------|---------------|
| | $\ u - u_{\text{exact}}\ $ | $\ u - u^*\ $ | LM-BFGS iters | $\ u - u_{\text{exact}}\ $ | $\ u - u^*\ $ |
| 1 | 0.00848782 | 0.797512 | 47 | 0.0040405 | 0.797476 |
| 0.01 | 0.00403758 | 0.00673811 | 171 | 0.00403347 | 0.00674269 |
| 0.0001 | 0.00364934 | 0.00361768 | 124 | 0.00364973 | 0.00361806 |

Table 1: Error norms and iteration counts for the linear source inversion problem with several different regularization parameters, and for both the reduced space and full space formulations. The error norms are L^2 .

Some results are shown in table 1. For a sample of three different regularization parameters R , we have computed L^2 norms of the error in the solution and the mismatch from the target, and for the reduced-space method we have recorded the number of LM-BFGS iterations needed to reach the specified tolerance. As expected, as the regularization parameter is reduced the target is matched more closely.

It is of course possible to improve on this optimization procedure in many ways; however, the focus of this paper is on the software infrastructure needed to enable setting up either the full KKT system needed for a full-space approach or the sequence of systems needed for a reduced-space approach.

4.2 Nonlinear source inversion

In this example we consider least-squares estimation of the source term in a nonlinear boundary value problem.

$$\min_{u,\alpha} f(u,\alpha) = \frac{1}{2} \int_0^\pi (u - u^*)^2 d\Omega + \frac{R}{2} \int_0^\pi \alpha^2 d\Omega \quad (18)$$

$$\text{subject to } \begin{cases} \nabla^2 u = \sin(u) + \alpha + g & \text{in } \Omega \\ u = 0 & \text{at } x = 0, x = \pi \end{cases} \quad (19)$$

The Lagrangian is

$$L = f + \int_0^\pi [\nabla \lambda \cdot \nabla u + \lambda \sin(u) + \lambda \alpha + \lambda g] dx = 0. \quad (20)$$

With the method of manufactured solutions we can construct an exactly solvable problem with

$$u(x) = \sin(x) \quad (21)$$

$$\lambda(x) = R \sin^2(x) \quad (22)$$

$$\alpha(x) = -\sin^2(x) \quad (23)$$

$$u^* = -R \cos(2x) + R \sin^2(x) + \sin(\sin(x)) + \sin(x) \quad (24)$$

$$g(x) = \sin^2(x) - \sin(x) - \sin(\sin(x)). \quad (25)$$

4.2.1 Programming the functional

Here is the code to create the Lagrangian Functional object.

```

/* Create the mesh object */
int nx = 512;
const double pi = 4.0*atan(1.0);
MeshType meshType = new BasicSimplicialMeshType();
MeshSource mesher = new PartitionedLineMesher(0.0, pi, nx, meshType);

Mesh mesh = mesher.getMesh();

/* Define the symbolic geometry */
CellFilter interior = new MaximalCellFilter();
CellFilter bdry = new BoundaryCellFilter();

/* Discretization specifiers */
QuadratureFamily quad = new GaussianQuadrature(4);
BasisFamily basis = new Lagrange(1);

/* Define the unknown functions */
Expr u = new UnknownFunction(basis, "u");
Expr lambda = new UnknownFunction(basis, "lambda");
Expr alpha = new UnknownFunction(basis, "alpha");

DiscreteSpace discSpace(mesh, List(basis, basis, basis), vecType);
Expr w0 = new DiscreteFunction(discSpace, 0.0);

/* Regularization constant (logR is a loop variable) */
double R = pow(10.0, logR);

/* Write the target and forcing function */
Expr dx = new Derivative(0);
Expr x = new CoordExpr(0);

```

```

Expr uExact = sin(x);
Expr sx = sin(x);
Expr cx = cos(x);
Expr ssx = sin(sx);
Expr sx2 = sx*sx;
Expr cx2 = cx*cx;

Expr g = sx2 - sx - ssx;
Expr target = 2.0*R*(sx2-cx2) + R*sx2*ssx + sx;

/* we can now define the objective and constraint */
Expr fit = Integral(interior, 0.5*pow(u-target, 2.0), quad);
Expr reg = Integral(interior, 0.5*R*(alpha*alpha), quad);
Expr constraint = Integral(interior,
    (grad*u)*(grad*lambda) + alpha*lambda + g*lambda + lambda*sin(u), quad);
Expr constraintBC = EssentialBC(bdry, lambda*u, quad);

/* Write the Lagrangian */
Expr L = fit + reg + constraint;
Functional Lagrangian(mesh, L, constraintBC, vecType);

```

The next step is to set up either a full-space or reduced-space solve.

4.2.2 Solution by the full-space method

As in the linear source inversion problem, obtaining the full KKT system is a matter of taking variations of the Lagrangian. The difference is that a `NonlinearProblem` is produced.

```

LinearSolver<double> linSolver
    = LinearSolverBuilder::createSolver("amesos.xml");

ParameterXMLFileReader reader("nox-amesos.xml");
ParameterList noxParams = reader.getParameters();
NOXSolver nonlinSolver(noxParams);

Expr dum;
NonlinearProblem prob = Lagrangian.nonlinearVariationalProb(List(lambda,u,alpha),
    w0, List(lambda, u, alpha), w0, dum, dum);

prob.solve(nonlinSolver);

```

The solution is written into the discrete function `w0`.

4.2.3 Programming an adjoint gradient method

The Lagrangian is used to construct a `NonlinearPDEConstrainedObj` object.

```

DiscreteSpace ds2(mesh, basis, vecType);

Expr u0 = new DiscreteFunction(ds2, 0.0);
Expr lambda0 = new DiscreteFunction(ds2, 0.0);
Expr alpha0 = new DiscreteFunction(ds2, 0.0);

RCP<PDEConstrainedObjBase> obj
    = rcp(new NonlinearPDEConstrainedObj(
        Lagrangian, u, u0,
        lambda, lambda0,
        alpha, alpha0,
        nonlinSolver, linSolver));

```

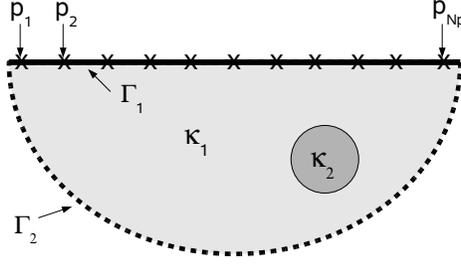


Figure 10: Geometry of frequency-domain conductivity inversion. The data were generated by a forward model having the inclusion region shown with conductivity κ_2 . Samples were taken at the probe locations p_1 to p_{Np} , over a range of frequencies. Boundary conditions are a unit sinusoidal load on Γ_1 and insulation on Γ_2 .

The nonlinear and linear solver arguments specify the solvers to be used for the state and adjoint equations respectively. As in the linear case, three discrete functions are used, each defined on the reduced space.

4.2.4 Numerical results

The reduced space optimizer used the same algorithm and tolerances as in the linear source inversion example above. Nonlinear solves of the full KKT system and of the state equation were done with NOX's implementation of Newton's method with line search. Tolerance for the nonlinear solves was 10^{-10} . The Amesos KLU solver was used for all linear solves.

| R | Reduced space | | | Full space | |
|--------|----------------------------|---------------|---------------|----------------------------|---------------|
| | $\ u - u_{\text{exact}}\ $ | $\ u - u^*\ $ | LM-BFGS iters | $\ u - u_{\text{exact}}\ $ | $\ u - u^*\ $ |
| 1 | 0.0494592 | 3.08711 | 6 | 0.0494581 | 3.08711 |
| 0.01 | 0.00180347 | 0.0295964 | 14 | 0.00180349 | 0.0295963 |
| 0.0001 | 0.000101083 | 0.000325266 | 14 | 2.53442e-05 | 0.0002902 |

Table 2: Error norms and iteration counts for the nonlinear source inversion problem with several different regularization parameters, and for both the reduced space and full space formulations. The error norms are L^2 .

4.3 Frequency-domain conductivity inversion

Our final model problem is frequency-domain inversion of a material's conductivity parameter κ . The underlying physics might be, for example, heat conduction; a slight change in problem setup would give a model appropriate to eddy current inversion. The problem's geometry is sketched in figure 4.3. The time-domain model is assumed to be

$$\nabla \cdot [\kappa \nabla \phi] = \frac{\partial \phi}{\partial t} \quad (26)$$

where ϕ is some scalar field, with insulating boundary conditions on all surfaces except for a surface Γ_1 where a sinusoidal load is imposed,

$$\frac{\partial \phi}{\partial n} = e^{-i\omega t} \text{ on } \Gamma_1. \quad (27)$$

Assuming κ to be independent of ϕ and writing $\phi = u(x) e^{-i\omega t}$, we have

$$\nabla \cdot [\kappa \nabla u] + i\omega u = 0 \quad (28)$$

$$\frac{\partial u}{\partial n} = 1 \quad \text{on } \Gamma_1 \quad (29)$$

$$\frac{\partial u}{\partial n} = 0 \quad \text{on } \Gamma \setminus \Gamma_1. \quad (30)$$

To ensure positive conductivity we introduce an auxiliary design variable α and write $\kappa = e^\alpha$. Now, signals at different frequencies penetrate to different skin depths (see, e.g., [12]) so one usually carries out a frequency sweep, taking data at N_f frequencies $\omega_1, \omega_2, \dots, \omega_{N_f}$. We assume an array of N_p discrete probe locations, and suppose that measurements $u_f^*(p_s)$ have been taken at frequencies ω_f and probe locations p_s . The magnitudes of the signals differ by several orders of magnitude over the frequency range, so in the objective function we will use relative misfits rather than absolute misfits. With only $N_p \times N_f$ measurements the problem is clearly ill-posed and demands regularization. We use a mollified total variation diminishing (TVD) regularization,

$$R \sqrt{\epsilon^2 + h^2 (\nabla \alpha)^2} \quad (31)$$

where R is a regularization coefficient, ϵ is a constant that smooths the singularity, and h is the local cell diameter. Having specified the fitting objective, regularization, and constraints, we can pose the PDECO problem.

$$\min_{u, \alpha} F(u, \alpha) = \frac{1}{2} \sum_{s=1}^{N_p} \sum_{f=1}^{N_f} \left(\frac{u_f(p_s) - u_f^*(p_s)}{u_f^*(p_s)} \right)^2 + R \int_{\Omega} \sqrt{\epsilon^2 + (\nabla \alpha)^2} d\Omega \quad (32)$$

$$\text{subject to, for } f = 1 \text{ to } N_f, \begin{cases} \nabla \cdot [e^\alpha \nabla u_f] + i\omega_f u_f = 0 & \text{in } \Omega \\ \frac{\partial u_f}{\partial n} = 1 & \text{on } \Gamma_1 \\ \frac{\partial u_f}{\partial n} = 0 & \text{on } \Gamma_2 \end{cases} \quad (33)$$

Note that the N_f constraints are decoupled and can be solved independently.

4.3.1 Programming the conductivity inversion problem

For this problem we show only the code for setting up the Lagrangian. As in the previous examples, once the Lagrangian has been constructed the problem is ready for solution by either a full space or reduced space method. Some interesting features of this problem are the use of complex-valued expressions and the decoupling of the states at different frequencies.

```

Expr I = new ComplexExpr(0.0, 1.0); // \sqrt{-1}

Array<Expr> u(nFreq); // Initialization code omitted
Array<Expr> lambda(nFreq); // Initialization code omitted

Expr fit = 0.0;
Expr constraint = 0.0;
Expr constraintBC;

/* set up equation for each frequency */
double R = 0.1;
for (int f=0; f<nFreq; f++)
{
    /* Sum the squared residuals at the probes */
    for (int p=0; p<probes.size(); p++)
    {
        fit = fit + Integral(probes[p],
            0.5*pow((u[f].imag()-p_i[f][p])/p_i[f][p], 2.0)
            + 0.5*pow((u[f].real()-p_r[f][p])/p_r[f][p], 2.0), quad);
    }
    /* Write the PDE as a constraint */
    constraint = constraint
        + Integral(interior, exp(kappa)*(grad*lambda[f])*(grad*u[f])
            - I*omega[f]*lambda[f]*u[f], quad)
        - Integral(top, lambda[f].real(), quad);
}

```

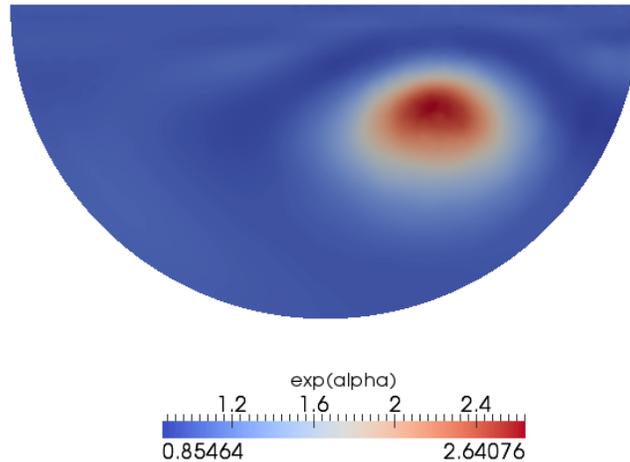


Figure 11: Contours of recovered conductivity. As seen by comparison with figure 11 the location of the region of enhanced conductivity is determined accurately, though the conductivity there is underestimated.

```

/* regularize for smoothness in the conductivity */
Expr h = new CellDiameterExpr();
Expr reg = Integral(interior, R*sqrt(1.0+h*h*(grad*kappa)*(grad*kappa)), quad);

Expr L_eqn = fit + reg + constraint;
Expr L_BC = constraintBC;

```

Once the Lagrangian has been constructed it is used to set up a `NonlinearPDEConstrainedObj` objective function objects as in the nonlinear source inversion example, which is then used in an optimization loop. In a reduced-space approach the state equations at different frequencies decouple and can be solved independently; this is managed automatically by the objective function object. Because of the complexity of preconditioning this problem we make no attempt to solve this system with a full space method.

4.3.2 Numerical results

In figure 11 is shown the recovered conductivity profile in a domain having a region of enhanced conductivity. Twenty-one probes spaced evenly along the top measured the response at ten frequencies. The frequencies were chosen to give a range of skin depths $\frac{1}{11}, \frac{2}{11}, \dots, \frac{10}{11}$. The regularization constant was $R = 1$ and the mollification constant was $\epsilon = 1$. The LM-BFGS algorithm converged after 80 iterations. The location of the enhanced conductivity inclusion is determined accurately, though its magnitude is underestimated ($\kappa \approx 2.5$ compared to the exact value of 10.)

4.4 Summary of model problems

We have shown how to set up and solve a variety of linear and nonlinear PDE-constrained optimization problems using both full-space and reduced-space methods. In each case, the same `LagrangianFunctional` object was used to produce code for the full-space and reduced-space formulations. Notice also that the code for the PDE constraints is just that needed to write a forward simulator, so it is a simple matter to take a forward simulator programmed in Sundance and extend it for use in PDE-constrained optimization.

5 Conclusions

PDECO is required to solve large scale inverse problems in engineering and science. However the implementation of these methods is time consuming as well as plagued with complications. A significant obstacle preventing PDECO

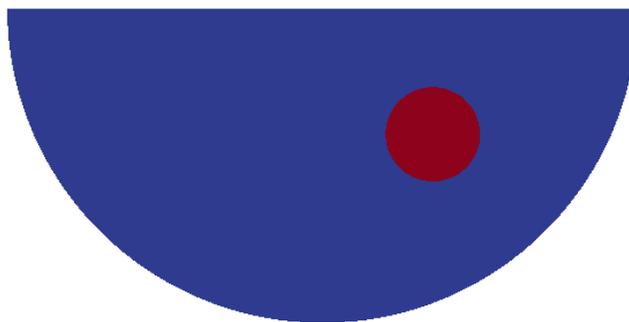


Figure 12: Exact location of the enhanced conductivity region in the frequency-domain inversion example. The bulk conductivity is 1, that in the enhanced region is 10.

from becoming a mainstream analysis tool is rooted in the practise of developing simulation codes without planning for future use in optimization. To overcome this obstacle, we introduce a high-level simulation toolkit called Sundance with which both very efficient forward and inverse problems can be programmed conveniently. Using components provided by Sundance the finite element weak form can be represented and a fully functional simulator can be built. More importantly, the same infrastructure can be used to differentiate a Lagrangian function, thereby automatically providing a solution mechanism for PDECO. We demonstrate the use of Sundance on several numerical examples using different PDECO solution strategies. These examples are non-trivial but it is apparent that this capability can be applied to more complicated applications without much additional effort.

Sundance is available as part of the Trilinos suite. It leverages multiple Trilinos packages for distributed linear algebra, low-level finite element libraries, linear and nonlinear solvers, and utilities. Sundance complements the low-level capabilities in Trilinos with a unique high-level optimization-enabled simulation development capability for Trilinos.

6 Acknowledgements

KRL acknowledges support from NSF awards 0830655 and 0904834, from a subcontract from Sandia National Laboratories, and from startup funds from Texas Tech University. PTB acknowledges support from the Department of Energy Office of Advanced Scientific Computing Research under contract 10-014804.

References

- [1] Volkan Akcelik, George Biros, and Omar Ghattas. Parallel multiscale gauss-newton-krylov methods for inverse wave propagation. In *Proceedings of the IEEE/ACM SC2002 Conference*, 2002.
- [2] L. Biegler, O. Ghattas, and B. van Bloemen Waanders. *Large-Scale PDE-constrained Optimization*. Springer, 2003.
- [3] Lorenz T. Biegler, Omar Ghattas, Matthias Heinkenschloss, and Bart van Bloemen Waanders, editors. *Large-Scale PDE-Constrained Optimization*, volume 30 of *Lecture Notes in Computational Science and Engineering*. Springer-Verlag, Heidelberg, 2003.
- [4] Paul T. Boggs and Jon W. Tolle. Sequential quadratic programming. *Acta Numerica*, 1995:1–52, 1995.
- [5] J. P. Boyd. An analytical and numerical study of the two-dimensional bratu equation. *Journal of Scientific Computing*, 1(2):183–206, 1986.
- [6] J. E. Dennis, Jr. and Robert. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

- [7] R. Fletcher. *Practical Methods of Optimization*. Wiley, 1987.
- [8] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*. Academic Press, New York, 1981.
- [9] M. Gunzburger. *Flow Control*. Springer, 1995.
- [10] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.
- [11] Victoria E. Howle, Robert C. Kirby, Kevin Long, Brian Brennan, and Kimberly Kennedy. Playa: High-performance programmable linear algebra. *Scientific Programming*, X:X, 2011.
- [12] John David Jackson. *Classical Electrodynamics Third Edition*, volume 67. Wiley, 1998.
- [13] C. T. Kelley. *Iterative Methods for Optimization*. SIAM, 1999.
- [14] Anders Logg, Kent-Andre Mardal, and Garth Wells. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2011.
- [15] Kevin Long, Robert Kirby, and Bart van Bloemen Waanders. Unified embedded parallel finite element computations via software-based fréchet differentiation. *SIAM Journal on Scientific Computing*, 32(6):3323–3351, 2010.
- [16] Jorge Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of Computation*, 35:772–782, 1980.
- [17] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer-Verlag, New York, 1999.
- [18] P. J. Roache. Verification of codes and calculations. *AIAA Journal*, 36 (5):696–702, 1998.
- [19] P. J. Roache. Code verification by the method of manufactured solutions. *J. Fluids Eng.*, 124 (1):4–10, 2002.
- [20] Alar Toomre. Some flattened isothermal models of galaxies. *Astrophys. Jour.*, 259:535–543, 1982.
- [21] B. van Bloemen Waanders, R. Bartlett, K. Long, P. Boggs, and A. Salinger. Large scale non-linear programming for PDE constrained optimization. Technical Report SAND2002-3198, Sandia National Laboratories, 2002.
- [22] Curtis R. Vogel. *Computational Methods for Inverse Problems*. SIAM, 2002.