OPT++: An Object-Oriented Toolkit for Nonlinear Optimization

J. C. MEZA and R. A. OLIVA
Ernest Orlando Lawrence Berkeley National Laboratory
and
P. D. HOUGH and P. J. WILLIAMS
Sandia National Laboratories

Object-oriented programming is a relatively new tool in the development of optimization software. The code extensibility and the rapid algorithm prototyping capability enabled by this programming paradigm promise to enhance the reliability, utility, and ease of use of optimization software. While the use of object-oriented programming is growing, there are still few examples of general purpose codes written in this manner, and a common approach is far from obvious. This paper describes OPT++, a C++ class library for nonlinear optimization. The design is predicated on the concept of distinguishing between an algorithm-independent class hierarchy for nonlinear optimization problems and a class hierarchy for nonlinear optimization methods that is based on common algorithmic traits. The interface is designed for ease of use while being general enough so that new optimization algorithms can be added easily to the existing framework. A number of nonlinear optimization algorithms have been implemented in OPT++ and are accessible through this interface. Furthermore, example applications demonstrate the simplicity of the interface as well as the advantages of a common interface in comparing multiple algorithms.

Categories and Subject Descriptors: G.1.6 [Numerical Analysis]: Optimization—Nonlinear programming, quadratic programming methods, gradient methods, constrained optimization, unconstrained optimization

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: Object oriented programming, scientific computing, parallel optimization

ACM Reference Format:

This work at Ernest Orlando Lawrence Berkeley National Laboratory was supported by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.
Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.
Author's address: J. C. Meza; email: JCMeza@lbl.gov.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

1. INTRODUCTION

Many software packages have been developed to solve a range of optimization problems, including integer programming, linear programming, quadratic programming, and nonlinear programming problems. (For an excellent overview of available optimization software, see Moré and Wright [1993].) It is often, though not exclusively, the case that a given software package is focused on a particular class of problems or algorithms. For a novice practitioner of optimization to explore the effectiveness of a variety of algorithms on his problem, he must learn and implement the code-specific interface for each software package. Furthermore, an algorithm developer who wishes to develop and test a new algorithm often must do so from scratch. The goal of this work is to facilitate algorithm development, comparison, and use by developing a common means of describing nonlinear optimization problems with a range of characteristics, defining a common interface to different solution methods, and providing an extensible infrastructure for adding new algorithms or problem descriptions.

Object-oriented programming is a relatively new tool in the development of optimization software. The code extensibility and the rapid algorithm prototyping capability enabled by this programming paradigm enhances the reliability, utility, and ease of use of optimization software. While the use of object-oriented programming is growing, there are still few examples of general purpose codes written in this manner, and a common approach is far from obvious. This article describes OPT++, a C++ class library for nonlinear optimization first proposed in Meza [1994]. The primary goals are to use the ideas of object-oriented programming to: 1) develop better program interfaces for the user of optimization codes, 2) enable rapid evaluation of several optimization codes for a given problem, 3) enable rapid prototyping of new optimization algorithms, and 4) enable more reusability of optimization components and codes. The design is predicated on the concept of distinguishing between an algorithm-independent class hierarchy for nonlinear optimization problems and a class hierarchy for nonlinear optimization methods that is based on common algorithmic traits. The interface is designed for ease of use while being general enough so that new optimization algorithms can be easily added to the existing framework. A number of unconstrained nonlinear optimization algorithms were implemented initially, and over the years, numerous researchers have contributed to the code development and incorporated new capabilities, including novel parallel optimization methods [Hough and Meza 2002], parallel finite-difference gradient calculations [Howle et al. 2000], constrained optimization methods, and additional testing harnesses.

In this work, we will focus on problems for which the cost function is described by the output of a simulation—the so-called simulation-based optimization problem. Thus, we expect the user community to consist primarily of scientists and engineers who are not likely to be experts in optimization. To date, successful applications of OPT++ in this arena include the solution of diverse problems such as molecular conformation, optimal control of chemical vapor deposition furnaces, and parameter identification for an extreme ultraviolet lithography lamp model [Meza and Martinez 1993; Moen et al. 1995; Meza...
et al. 1996]. We provide examples of such problems in Section 3 that demonstrate the simplicity of the interface, as well as the advantages of a common interface, in comparing multiple algorithms.

1.1 Related Software Packages

With the advent of simulation-based optimization a need arose for flexible interfaces between off-the-shelf optimization packages and application software that did not require extensive application code modification. The optimization and application communities responded with the development of many object-oriented optimization packages. In this section, we briefly compare OPT++ to other open-source object-oriented software packages. We assert that OPT++ is the only nonlinear optimization package that has the following three characteristics: 1) it was designed for simulation-based optimization problems, 2) it gives the user multiple different types of algorithms through a common interface, and 3) it provides support for general nonlinear constraints.

The earliest references to object-oriented optimization packages appear to have come out of the geophysical science community. The first such example, which comes out of The Center for Wave Phenomenon (CWP) Object-Oriented Optimization Library (COOOL) [Deng et al. 1996] was motivated by the solution of various problems in geophysics. This library attempted to address some of the unique aspects of simulation-based optimization and had the rather novel idea of using the expect package to communicate between the objective function and the optimization algorithms. The design principles used by the developers of this package included providing a consistent application programming interface, the ability to allow incremental development of the library, and the ability to build other application packages from the library. The optimization solvers that were initially provided by COOOL included a simplex and a Powell’s method for functions that did not require derivatives, and a conjugate gradient and quasi-Newton method for problems with derivative information. However, no methods for the solution of general nonlinear inequality constrained problems were provided. A future release planned on including a genetic algorithm and a method for simulated annealing, although at present the COOOL library is not supported.

The Hilbert Class Library (HCL) [Gockenbach et al. 1999] is another example of an object-oriented optimization framework for problems arising out of simulation-based optimization. Like the COOOL package, HCL was also motivated by problems arising out of geophysics, and in particular seismic inverse problems. Unlike some of the other packages however, HCL defines mathematical objects used in optimization, such as vectors and linear operators, and leaves the majority of the design of the optimization solvers to the user. The HCL classes can be grouped into four categories: 1) core classes, 2) tool classes, 3) algorithm classes, and 4) concrete classes. The core classes define the mathematical objects such as vectors, vector spaces, linear operators, and functionals. The algorithm classes in turn are where the optimization solvers are defined. As they note in their article, however, the "purpose of HCL is to define an environment… It is hoped that numerical analysts developing algorithms will
consider implementing them using HCL classes.” As such, the HCL algorithm class only includes four classes, a linear solver, a line search, a solver for the minimization of an unconstrained nonlinear functional where the gradient is available, and a similar method for problems where both the gradient and Hessian are available. No methods for the solution of general nonlinear inequality constrained problems are available.

Two extensions to HCL have recently been proposed. The Standard Vector Library (SVL) [Padula et al. 2004] and the Rice Vector Library (RVL) [Padula et al. 2005]. The SVL proposes a set of standard core abstract types for vectors that have enough functionality to build the operators necessary for nonlinear equation and optimization solvers. The RVL is a similar extension but focuses primarily on coordinate-free algorithms, such as Krylov subspace methods for linear systems and eigenvalue problems. The major goal of RVL, as stated in the report, is to create a class hierarchy that “mimics as closely as possible the basic concepts of calculus in Hilbert space.” Both of these frameworks also provide classes for functions and their evaluations. Much like the HCL, the optimization solvers described are for unconstrained optimization problems such as the output least squares formulation for seismic velocity analysis.

The next set of software packages focuses on large-scale nonlinear optimization problems. They generally are limited to active-set methods or interior-point methods and thus provide general nonlinear constraint support. Exploiting known problem structure and specialized linear solvers are the keys to the efficiency of these packages.

Exploitation of problem structure in finance applications has resulted in an interior-point-method-based library that can solve a range of quadratic programming problems without requiring the user to write customized linear algebra routines for each problem type. In OOPS [Gondzio and Sarkissian 2003; Gondzio and Grothey 2003, 2004], the authors exploit the nested block structure of the problem in the implementation of parallel nonlinear primal-dual interior-point methods. Blocks of matrices are the objects of interest as opposed to the nonlinear functions and optimization method classes in OPT++. The object-oriented design uses the concepts of encapsulation, inheritance, and polymorphism. Although OOPS has an object-oriented design, the package is implemented in C for the purposes of speed. Users of the package can solve linear programming, nonlinear portfolio optimization, and nonlinear stochastic programming problems.

Gertz and Wright [2003] implemented structure-independent primal-dual interior-point methods in the Object-Oriented Quadratic Programming (OOQP) software package, which solves convex quadratic programming problems. OOQP features such object-oriented concepts as encapsulation, inheritance, polymorphism, and templates. OOQP also has a layered design starting with a topmost solver layer followed by a problem formulation layer and a linear algebra layer on the bottom. The solver layer corresponds to the optimization class hierarchy in OPT++. The problem formulation layer consisting of data, variables, residuals, and a linear system class loosely corresponds to the function class hierarchy. We say loosely, since in OPT++ the constraint residuals and
linear system, which are solved at each iteration of the interior-point algorithm, belong to the relevant optimization class instead of the function class.

Another example is the Multifunctional Object-Oriented Architecture for Optimization (MOOCHO) [Bartlett http://software.sandia.gov/trilinos/packages/docs/dev/packages/moocho/doc/html/ind] package, which is the successor of rSQP++ [Bartlett 2001]. It consists of active-set and interior-point sequential quadratic programming methods for solving nonlinear optimization problems. MOOCHO is designed to take advantage of structure in problems based on partial differential equations in order to perform large-scale simultaneous analysis and design of PDE-constrained optimization problems. Heavy emphasis is placed on such features as dynamic memory allocation and management, abstract linear algebra interfaces, and flexible algorithm configurations.

The Toolkit of Advanced Optimization (TAO) [Benson et al. 2005] is an optimization library for the solution of large-scale unconstrained and bound constrained optimization problems on high-performance computing architectures. This toolkit was largely motivated by the scattered support for parallel computations and the lack of reuse of linear algebra software. As such much of its development was modeled after the PETSc project and uses many of the PETSc constructs for its linear algebra operations. Also, because of the motivation for solving large-scale problems, the optimization methods included in TAO have been designed to work best in that setting. We note that TAO differs from the other large-scale packages we have mentioned in that it provides implementations of a variety of methods. These methods currently include a limited memory variable metric (LMVM) method, a conjugate gradient method, and a Newton method with both a linesearch and a trust region option. For bound constrained problems, TAO also includes an active set Newton method, a gradient projection CG, an interior point method for quadratic programs, and a method based on a solver for mixed complementarity problems. The mixed complementarity solver can also solve the more general box-constrained variational inequality problems in addition to bound constrained problems. No methods for the solution of general nonlinear inequality constrained problems are available.

The final set of libraries we consider is focused on derivative-free optimization. These libraries offer support for general nonlinear constraints, are ideal for problems with small numbers of variables, and handle black-box functions, but they provide implementations of only one type of algorithm. In particular, they implement variations of pattern search algorithms.

The Nonlinear Optimization for Mixed Variables and Derivatives (NOMAD) [Couture et al. 2006] project is a joint venture among researchers from Ecole Polytechnique de Montreal, Rice University, and Air Force Institute of Technology. The NOMAD package implements a class of mesh adaptive direct search methods, which includes generalized pattern search methods, to solve nonlinear mixed variable problems with nonlinear constraints. The support of categorical, continuous, and discrete variables as well as nonlinear constraints, distinguishes NOMAD from other pattern search codes. In addition, NOMAD can make use of surrogates as part of the optimization process. NOMAD's object-oriented design uses the concepts of abstraction, encapsulation, inheritance. APPSPACK [Gray and Kolda 2004; Kolda 2004], which was developed at Sandia
National Laboratories, consists of an asynchronous parallel pattern search to solve unconstrained and bound-constrained continuous optimization problems with continuous variables. The ability to handle linear constraints was recently made available, and support for general nonlinear constraints is forthcoming.

1.2 Organization
The rest of this article is organized as follows. Section 2 contains a short overview of the OPT++ design philosophy as well as descriptions of the major classes within OPT++, the parallelization of OPT++, and the testing infrastructure. Section 3 provides details of algorithmic class designs. In Section 4 we give several examples of using OPT++ for some applications. We conclude in Section 5 with a discussion of future work.

2. SOFTWARE DESCRIPTION

2.1 OPT++ Design Philosophy
The underlying concept behind the OPT++ software design is the distinction between nonlinear problems and the methods used to solve these problems. The rationale for this decision is that while optimization users may know a great deal about the underlying physics of a simulation-based problem, they seldom are aware of the intricacies of the various methods, nor should they need to become experts in numerical analysis. On the other extreme, the optimization algorithm developers usually do not care about the details of how a problem is defined other than to know certain mathematical properties (e.g., continuity, special structure) and some general problem characteristics (e.g., dimension, availability of analytic gradient). By making a distinction between problems and methods, we can enable maximal code reuse by both groups each time a new problem is presented or a new algorithm is developed. Thus, we focus our design efforts on two areas: 1) the development of a class hierarchy for general nonlinear optimization problems, and 2) the development of a common interface for nonlinear optimization algorithms.

We consider nonlinear optimization problems of the form

\[
\min_{x \in \mathbb{R}^n} \quad f(x)
\]

subject to

\[
\begin{align*}
    h_i(x) &= 0, & i &= 1, \ldots, p, \\
    g_i(x) &\geq 0, & i &= 1, \ldots, m.
\end{align*}
\]

where \( x \in \mathbb{R}^n \), and \( f : \mathbb{R}^n \to \mathbb{R} \). We assume that the objective function \( f(x) \) is a nonlinear function, and the constraint functions \( h_i(x) \) and \( g_i(x) \) can be either linear or nonlinear functions. Briefly, the data for unconstrained problems and actions on that data are encapsulated in C++ classes to ensure modularity. A hierarchy of problems based on the availability of analytic derivative information is established through the use of inheritance. Constraints, themselves encapsulated in a C++ class hierarchy based on the type of constraint, are then incorporated by including a pointer to a constraint object. Sophisticated classes of problems, such as those with known mathematical properties or structure,
can be derived from this set of problem classes. The use of virtual functions and templates is prevalent throughout, to facilitate polymorphism and code reuse, respectively. More details on the designs of the nonlinear problem classes and the constraint classes can be found in sections 2.2 and 2.4. (Note: For those unfamiliar with object-oriented programming terminology, please see Budd [1991], Holub [1992], and Stroustrup [1987].)

Many computational techniques exist for solving problem (1). Such methods include conjugate gradient methods, quasi-Newton and Newton methods, direct search methods, genetic algorithms, and simulated annealing for unconstrained problems. Active-set methods and nonlinear interior-point methods are common approaches for solving constrained optimization problems. The algorithm class hierarchy is based on inherent similarity between methods. At a coarse level, such categorizations might include derivative-based versus derivative-free methods or global versus local methods. At the next level of granularity, one can think of a group of Newton-like methods, a group of CG-like, or a group of direct search methods. Some of the common high-level tasks might include generating a search direction, determining step length, and checking for convergence. Polymorphism is particularly important here, as each method may have an implementation that differs slightly from that of its parent. Furthermore, there is a minimal interface that is common to all methods. The user need only provide an algorithm with a pointer to his problem, as constructed from the problem classes, and call the algorithm optimize method. A set of accessor methods is provided for the more knowledgeable users who wish to adjust specific algorithmic parameters. These are the principles upon which the algorithm class design is based. More details are discussed in Section 2.3.

2.2 Nonlinear Problem Classes

One of the first questions that arises in an optimization problem is the degree of continuity in the objective function. This information may not be readily available, but what is clear is the availability of analytic derivatives. Therefore, OPT++ classifies nonlinear programming problems by the availability of functions for computing the derivatives:

- **NLP0**—No derivative function available.
- **NLP1**—First derivative function available.
- **NLP2**—First and second derivative function(s) available.

These classes fall naturally in an object-oriented hierarchy based on the amount of available derivative information. The class **NLP0** represents a nonlinear problem without derivative information; its data members include information that is common to all nonlinear problems, such as the problem dimension, the current point, the current function value, and the method to compute the objective function. The class **NLP1** is derived from **NLP0** with additional data members for the gradient, and the function that computes the gradient. Likewise, **NLP2** is derived from **NLP1** with methods and data members for computing and storing the Hessian.
In our implementation of the nonlinear problem classes, we have defined the methods that evaluate the objective function, gradient, and Hessian as pure virtual functions. This means that the NLP0-2 classes are themselves abstract classes, intended as base classes for user-defined nonlinear function classes. Thus, we defer the definition of how to compute the objective function, the gradient, and the Hessian, so that users can create their own definitions and incorporate application-specific details for a given problem. The optimization method classes (described in the next section) operate with the base NLP0-2 classes, hence the optimization algorithms will work with the concrete user-defined classes without having to be rewritten.

In addition, we would like the ability to treat these nonlinear problem classes polymorphically, independently of the degree of derivative information that each class contains. To accomplish this goal, we define the abstract base class NLPBase, which defines the abstract interface to the NLP0-2 nonlinear problem classes. Note that under this design the class NLP0 inherits from NLPBase, pure virtual methods for the gradient and the Hessian. Even though analytic derivative information is not available, we have implemented default finite-difference methods to approximate the gradient and Hessian in the corresponding concrete class to preserve static type safety. The problem class hierarchy is shown in Figure 1.

This design decision was driven by the development and implementation of nonlinear constraint classes. In OPT++, a nonlinear constraint is a nonlinear function with an associated righthand side. We elected to construct our nonlinear constraints with a general NLP pointer to 1) prevent an excessive number of constructors in the nonlinear constraint class, and 2) remove the requirement on available derivative information. However, we lost the ability to call
the method to evaluate the gradient because this method was absent in the original implementation of class \texttt{NLP0}. Thus we declared methods that evaluate the gradient and Hessian as pure virtual functions in base class \texttt{NLP}, which is inherited by class \texttt{NLP0}, so that the nonlinear constraint can have access to them. After many design reviews, we concluded that while not optimal, this was a reasonable approach. Arguably, we could have forced the nonlinear constraints to have first derivative information available.

In addition, OPT++ provides three concrete classes derived from \texttt{NLP0-2}, called \texttt{NLF0-2}. These classes provide a particular interface for constructing an NLP from user defined routines for the objective function, the available derivatives, and the constraints. The user can utilize objects of the \texttt{NLF0-2} classes to define the NLP for some simple optimization problems or use them as prototypes to define more sophisticated NLPs. The basic taxonomy is shown in Figure 2. In Section 3, we present examples that show how to use the \texttt{NLF0-2} classes to implement different types of optimization problems.

Finally, there are situations where a user may want to solve a problem using a method that requires first order derivatives, without supplying the analytic gradients explicitly. For instance, the objective function may be the result of a computer simulation for which no analytic expression for the derivative exists, even though the simulation values are smooth enough to approximate first derivatives numerically. The concrete class designed to handle these cases is \texttt{FDNLF1}, which is derived from the abstract \texttt{NLP1} class and provides first derivatives computed via finite-differences.

2.3 Optimization Method Classes

Many possible classifications exist for optimization algorithms, but most well-known methods can be grouped into one of three classes:

1. Direct search methods—methods that do not require or use derivative information.
2. Conjugate gradient like methods—methods based on the conjugate gradient method.
3. Newton like methods—methods that use both derivative and second order information to build a quadratic model.
For example, methods such as the Nelder-Mead simplex method, the box method, the parallel direct search method, and pattern search methods, fall into the direct search class. The nonlinear conjugate gradient method falls into the Conjugate Gradient class. The Newton class could include methods such as finite-difference Newton, quasi-Newton methods, and inexact Newton methods. We give a simple taxonomy for some popular algorithms in Figure 3. Finally, the limited-memory BFGS method falls into a separate class—LBFGS-like—because of the distinct handling of the Hessian approximations.

Based on this classification, we have implemented C++ classes for several methods including: 1) a Newton method, 2) a finite-difference Newton method, 3) a Quasi-Newton method, 4) a nonlinear conjugate gradient method, 5) a parallel direct search method, 6) a generating search set method, and 7) a nonlinear interior point method. In Figure 4, we present the class hierarchies for two of the implemented methods.

The base class, called **OptimizeClass**, consists of information that is required by all optimization classes. Note that once again we use the concept of polymorphism through declaration of the pure virtual function `optimize()`, which is a placeholder for the specific optimization routines. Each method class will redefine `optimize()` to implement its specific algorithm for computing the minimum of a function. However, it is important to provide the interface for `optimize()` in the base class since it is common to all of the derived classes. Similarly, pure virtual functions are provided for high-level tasks common to the optimization process implemented by all algorithms. These include `computeSearch(SymmetricMatrix&)`, `acceptStep(int, int)`, and `checkConvg()`. Again, polymorphism allows for algorithm-dependent implementation of these tasks.

The next two classes **OptNewtonLike** and **OptCGLike** are derived from **OptimizeClass**. The major difference between these two classes is that the Newton-like classes require extra storage for the Hessian matrix. Finally, the classes **OptQNewton** and **OptCG** constitute the actual optimization methods. Inside these two classes, we define the optimization algorithms specific to each method. In the case of the **OptQNewton** class, the algorithm consists of a Quasi-Newton method with a BFGS update formula for the Hessian. The **OptCG** class implements a nonlinear conjugate gradient method.

Now, we will provide a brief overview of the constrained optimization methods in the package. OPT++ contains no less than six methods to solve
bound-constrained optimization problems. They include implementations of Newton’s method, barrier Newton’s method, an interior-point method, and a parallel direct search algorithm. The methods for solving general nonlinear constrained problems belong to the classes derived from `OptNIPSLike`, the base class for nonlinear interior-point methods. `OptFDNIPS` and `OptQNIPS` solve dense convex nonlinear problems with first derivative information available for the objective function and constraints and either finite-difference or BFGS approximations to the Hessian of the Lagrangian, respectively. The `OptDHNIPS` class implements a disaggregated Hessian approximation nonlinear interior-point method with either an NLF2 or least squares function operator and BFGS approximations to the constraint Hessians.

As an example of the reusability of object-oriented codes, all of the linear algebra is handled through the use of NEWMAT, the dense matrix package developed by Davies [2003], with some minor enhancements for the matrices that arise in the optimization algorithms. In addition, all of the Newton-like methods use the same line searches, a simple backtracking scheme, and another one based on the algorithm by Moré and Thuente [1992].

2.4 Constraint Classes

We view a constrained nonlinear problem as an unconstrained problem that has constraints. Implementing constraints in this manner prevents code replication
and preserves the inheritance hierarchy in the nonlinear problem classes. From a code developer's point of view, the advantages to this approach are that we only have to add a pointer to a constraint object, and constraint accessor methods to the existing nonlinear problem classes.

In Figure 5, we present the Constraint class hierarchy. From the ConstraintBase class, we derive BoundConstraint, LinearConstraint, NonLinearConstraint, and CompoundConstraint classes.

- **BoundConstraint**—defines upper and lower bounds on real-valued variables;
- **LinearConstraint**—an abstract class, which provides common data and functionality to classes LinearEquation and LinearInequality;
- **NonlinearConstraint**—an abstract class, which provides common data and functionality to classes NonLinearEquation and NonLinearInequality.

The following pure virtual functions are declared in ConstraintBase: evalResidual, evalGradient, evalHessian, getUpper, getLower, getConstraintValue, getNumOfCons, and amIFeasible. A pure virtual function is a function that has been declared but not defined in the base class; see Holub [1992]. To prevent compile-time errors, each derived class must provide a definition for a pure virtual function.

Let us consider the class LinearConstraint. We derive the class LinearConstraint from ConstraintBase by adding members for the coefficient matrix $A$, the matrix-vector product $Ax$, lower and upper bounds, and a flag to determine whether the constraint is written in standard form. The derived classes of LinearConstraint, LinearEquality and LinearInequality differ in their implementation of evalResidual, evalGradient, evalHessian, and amIFeasible functions.

Similarly, the NonLinearConstraint class is derived from ConstraintBase by adding a pointer to an NLP object. Like the objective function, a nonlinear constraint function is classified according to the availability of derivative information. However, the constructor argument lists for the two objects differ in the storage types for the function's value, gradient, and Hessian.
In 

\[
\begin{align*}
\text{min} & \quad f(x) \\
\text{s.t.} & \quad A_e x = b_e, \\
& \quad A_i x \geq b_i, \\
& \quad h(x) = 0, \\
& \quad g(x) \geq 0, \\
& \quad l \leq x \leq u,
\end{align*}
\]  

(2)

where \( f(x), h_i(x), g_i(x) \) are nonlinear scalar-valued functions and \( A_e \) and \( A_i \) are real-valued linear coefficient matrices.

To simplify the use of mixed constraint sets in OPT++, we create a \texttt{CompoundConstraint} class. A \texttt{CompoundConstraint} is an array of heterogeneous constraints. Construction of the \texttt{CompoundConstraint} class eliminates the need to implement different versions of optimization algorithms based on constraint type. The design of \texttt{CompoundConstraint} places the burden of properly managing the constraints on the algorithm developer. In this design, separate treatment of constraint classes is hidden from the user.

We derive the class \texttt{CompoundConstraint} from \texttt{ConstraintBase} by adding a member for an array of constraints, a counter of the number of constraint sets in the problem formulation, as well as comparison and insertion sort functions. Inside the \texttt{CompoundConstraint} constructor, OPT++ sorts the constraints so that equality constraints precede inequality constraints. This is done for efficiency reasons, since some optimization algorithms treat categories of constraints differently. If the constraints are presorted, the optimization algorithm does not have to constantly check for constraint type.

An important point is that if a user inadvertently selects an unconstrained optimization method to solve a constrained optimization problem, the method will issue an error message that the chosen method does not support constraints, recommends that the user try one of the constrained methods, and exits.
2.5 Parallel Optimization Methods

One approach to mitigating the high computational cost of the functions targeted by OPT++ algorithms is to take advantage of the availability of multiple processors. In order to enable this capability, parallel versions of some of the OPT++ algorithms have been implemented using MPI [Gropp et al. 1994]. The parallelism in OPT++ currently takes the form of coarse-grained parallelism, and in particular, it consists of the ability to perform multiple function evaluations simultaneously. This is particularly advantageous when finite-difference gradient approximations are needed or when derivative-free methods are used, both of which are common in simulation-based optimization. We now describe in more detail how the algorithms in OPT++ make use of parallelism.

When an analytic gradient is not available, the gradient-based algorithms in OPT++ must approximate it, using finite-difference computations. When each function evaluation requires the execution of an expensive simulation, the cost of solving the optimization problem can very quickly become prohibitive. One approach to addressing this problem was introduced by Byrd, Schnabel, and Shultz in 1988 [Byrd et al. 1988]. They suggest a straightforward way to take advantage of multiple processors when using a line search method with finite-difference gradients. In particular, extra processors are used to compute components of the finite difference gradient at the trial point while the function is being evaluated at that point. This is referred to as a speculative gradient computation, and the idea applies equally well to any gradient-based algorithm. One particularly nice feature of note is that while this approach clearly leads to substantial computational savings, there is no penalty when the trial point is not accepted.

Figure 7 shows the flow of a generic gradient-based optimization algorithm with the speculative gradient modification. The speculative gradient evaluation is embedded in the function evaluation and the finite-difference computation of the FDNLF1 class. It is hidden from the user and the algorithm developer and invoked automatically when the parallel version of OPT++ has been built. An accessor method is available to disable the speculative computation. It is unlikely that a user would want to do this; however, an algorithm developer may need to perform a function or gradient evaluation for which a speculative gradient does not make sense. An example of this occurs when computing some of the merit functions used by the nonlinear interior-point method [El-Bakry et al. 1996] implemented in OPT++.

As an additional note, it is observed in Byrd et al. [1988] that it is possible to extend this idea to using any additional processors to compute as many components of a finite-difference Hessian as possible. In OPT++, we currently rely on other Hessian approximations (e.g., BFGS) but plan to incorporate speculative Hessian evaluations in the future.

Derivative-free optimization methods are also commonly used to solve simulation-based optimization problems. They have the obvious advantage of not requiring a gradient, but they are also robust when the number of digits of accuracy in the function is low. Furthermore, they are embarrassingly parallel and can be implemented using coarse-grained parallelism. A set of
Fig. 7. This figure illustrates a gradient-based algorithm with a speculative gradient computation. For simplicity of illustration, we show the case where \( n = 2p - 1 \). While the function is being evaluated at the trial point, the remaining \( p - 1 \) processors are used to calculate up to \( p - 1 \) components of the finite difference gradient. If the trial point is accepted, we already have \( p - 1 \) components of the gradient available, and only need to calculate the remaining \( n - (p - 1) \) components, where \( n \) is the dimension of the problem. If the trial point is not accepted, we simply try again at the next trial point. Note that no time is lost because the function evaluation is required regardless.

search directions is generated according to the rules of the search method. The approach currently implemented in OPT++, the parallel direct search (PDS) algorithm of Dennis and Torczon [1991], generates directions based on a simplex. The forthcoming release of OPT++ will include a generating set search (GSS) approach described by Kolda et al. [2003] in which the directions are generated through the construction of a particular basis. In both cases, each
function evaluation is assigned to a processor, evaluated simultaneously, and the function values are compared to determine the next iterate and/or set of directions.

The final OPT++ algorithm, which leverages the simultaneous function evaluation capability, is Trust Region-Parallel Direct Search (TRPDS) algorithm developed by Hough and Meza [2002]. TRPDS employs the standard trust-region framework, but uses PDS to solve a nonstandard subproblem to compute the step at each iteration. An example iteration of the TRPDS algorithm is shown in Figure 8. Since TRPDS has both gradient-based and derivative-free stages in the algorithm, it makes use of simultaneous function evaluations in both of the manners described above. The minor exception is that TRPDS does not currently support a speculative gradient evaluation, though it does perform the finite-difference calculation in parallel. We plan to incorporate a speculative capability in the future.

As mentioned earlier, the parallelism in OPT++ is currently coarse-grained parallelism in the form of simultaneous function evaluations. However, it is often the case in simulation-based optimization problems that the simulation itself can also be executed in parallel. While this can be done now through the clever use of system calls and function wrappers, it can be a tricky process with the onus on the user. We intend to integrate this capability into OPT++ through the manipulation of MPI communicators in order to provide more flexibility with less effort for the user.

3. APPLICATIONS

3.1 Example Code

To illustrate some of the concepts, we now present several examples that solve a small nonlinear optimization problem using the optimization classes. The first

Fig. 8. Overview of the TRPDS algorithm. The point $x_{CP}$ is the Cauchy point, $x_N$ is the Newton point, and $x_c$ is the current point. These points are used to initialize the simplex over which PDS approximately minimizes the function. The solid circle represents the trust region. The step length is allowed to be twice the size of the trust region (dotted circle) to allow for the possibility of taking a step longer than the Newton step.
two test problems consist of Rosenbrock’s function,
\[
\min_x 100(x_2 - x_1^2)^2 + (1 - x_1)^2.
\]

In the first example, we assume that first derivatives are not available. In the second example, we assume that first derivatives are available but that second derivatives are not available. For the solution method, we will use a quasi-Newton method that employs a BFGS update formula for the Hessian.

Figures 9 and 10 display the source listings for the two sample problems. There are three major sections in the example code: 1) the function definition, 2) the nonlinear problem definition, and 3) the optimization method definition. In the first case, no analytic derivatives are supplied by the function, so
we allow OPT++ to compute them via a finite-difference approximation. This is encapsulated through the definition of the problem as an FDNLF1. In the second example (see Figure 10), analytic first derivatives are available, so we create an object of type NLF1. The corresponding definition of the objective function is also modified to reflect the availability of first derivatives. The components needed to specify the object are the dimension of the problem as well as pointers to the method that performs the objective function evaluation and to the method that initializes the function. In both cases the last step consists of creating an optimization method object from the OptQNewton class. We then call the member function optimize to do the actual optimization. Finally the solution is printed using the printStatus member function.

If the user would now like to try a different optimization method, the procedure would consist of replacing the creation of the OptQNewton object with a different method object, for example an OptCG object, to try the nonlinear conjugate gradient method.

The next example highlights the construction and use of a CompoundConstraint object. Figure 11 displays the source code for the following constrained optimization problem taken from the Hock and Schittkowski test set:

\[
\min_x (x_1 - x_2^2)^2 + \frac{1}{9}(x_1 + x_2 - 10)^2 + (x_3 - 5)^2, \tag{3}
\]

subject to

\[
-4.5 \leq x_1 \leq 4.5, \\
-4.5 \leq x_2 \leq 4.5, \\
-5.0 \leq x_2 \leq 5.0, \\
x_1^2 + x_2^2 + x_3^2 \leq 48.
\]

For the solution method, we use a Newton interior-point method. To represent the constraints, we create a BoundConstraint object, and a NonLinearInequality object. Specification of the BoundConstraint requires the dimension of the problem and a vector containing the bounds of the optimization variables. Nonlinear constraints are created in a similar manner as the objective function. The calling sequence includes the dimension of the problem, the total number of nonlinear constraints, the pointer to the method that performs the constraint evaluation, and the pointer to the method that initializes the constraint. In this example, we have incorporated the righthand side of the inequality constraint into the definition of the associated nonlinear function. The default bound for nonlinear inequality constraints is zero. Once each particular type of constraint has been created, we combine them into one object by creating a CompoundConstraint object. To define the constrained nonlinear problem, we need the dimension of the problem, the pointer to the method that performs the objective function evaluation, the pointer to the method that initializes the objective function, and the pointer to the constraints.

3.2 Multi-Material Heat Equation

The next example involves an inverse problem formulation of a problem in thermal analysis. The objective is to find the parameters of a model that would
Fig. 11. Constraints example.

produce a known temperature field. This problem is representative of several problems that we have worked on, including optimal control of a chemical vapor deposition furnace [Meza and Plantenga 1995; Moen et al. 1995], and a problem in parameter identification for an extreme ultraviolet lithography lamp model. For this problem, the objective function is defined as follows:

$$\min_x \| T(x) - T^* \|_2, \quad (4)$$

where $T^*$ is a prescribed temperature field, and $T(x)$ is the solution to the heat equation,

$$\frac{\partial T}{\partial t} - \kappa \nabla^2 T = 0, \quad (5)$$
with appropriate boundary conditions over a specified domain. For this example the domain is the region $[0, 3] \times [0, 2]$, and each unit square within the domain represents a different material with the boundary condition (fluxes) as indicated in Figure 12.

We use the Sundance framework [Long 2003] to develop a simulation of the heat equation on this multimaterial domain. Given three input variables corresponding to the three fluxes, our simulation computes the temperature on a regular mesh of fixed size. We denote the temperature field as $T(x)$. In a real-world scenario the prescribed temperature field $T^*$ would originate from a physical experiment, but for the purposes of this example $T^*$ was generated numerically using our simulation with a specific set of input parameters $x^*$, thus $T^* = T(x^*)$. Next, we attempted to recover $x^*$ by solving (4). Since analytic derivatives were not available, we used two direct search methods and
Table I. Comparison of Direct Search Methods for the Multimaterial Problem (Note: dpu = Divisions Per Unit Length)

|       | dpu | fevals | \(f(x^*)\)     | \(|s|\)     |
|-------|-----|--------|----------------|-----------|
| PDS   | 16  | 1092   | 1.7264e-3      | 1.5259e-5 |
| GSS2  | 16  | 1168   | 1.5125e-3      | 8.1739e-6 |
| GSS1  | 16  | 3125   | 6.2097e-3      | 7.9511e-6 |
| GSSa  | 16  | 1077   | 5.1055e-4      | 6.5979e-6 |
| PDS   | 8   | 1156   | 6.3770e-4      | 3.0518e-5 |
| GSS2  | 8   | 1483   | 5.1255e-3      | 8.5457e-6 |
| GSS1  | 8   | 3125   | 6.2097e-3      | 7.8511e-6 |
| GSSa  | 8   | 1077   | 1.2042e-4      | 6.5979e-6 |

Table II. Comparison of Gradient-Based Methods for the Multimaterial Problem (Note: dpu = Divisions Per Unit Length)

<table>
<thead>
<tr>
<th></th>
<th>dpu</th>
<th>fevals</th>
<th>(f(x^*))</th>
<th>(|g(x^*)|)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QNewton</td>
<td>16</td>
<td>287</td>
<td>1.7695e-8</td>
<td>7.3354e-5</td>
</tr>
<tr>
<td>LBFGS</td>
<td>16</td>
<td>623</td>
<td>2.2434e-7</td>
<td>2.8814e-3</td>
</tr>
<tr>
<td>QNewton</td>
<td>8</td>
<td>309</td>
<td>7.0871e-9</td>
<td>3.9285e-5</td>
</tr>
<tr>
<td>LBFGS</td>
<td>8</td>
<td>315</td>
<td>3.5252e-7</td>
<td>5.3536e-3</td>
</tr>
</tbody>
</table>

compared the solution to two gradient-based methods that used finite-difference gradients supplied by OPT++. The results are presented in Tables I and II. For the direct search methods, we used a function tolerance of \(10^{-4}\) and a step tolerance of \(10^{-4}\). For the gradient based methods, we used a function tolerance of \(10^{-8}\), and a finite-difference step size of \(10^{-6}\).

The first direct search method used was PDS, the Parallel-Direct Search method discussed in Section 2.5. The second was GSS, or Generating Set Search method, using three types of generating basis. In GSS2 we use the standard basis of size \(2N\), namely the columns of \([I, -I]\), where \(I\) is the identity. In GSS1 we use the standard basis of size \(N + 1\), namely \([I, -1]\), where \(1\) is the vector with all entries equal to 1. In GSSa we augment the standard \(2N\) basis with search directions that combine the standard directions in pairs.

To check the effects of the mesh size on the results we used meshes of size \(24 \times 16\) and \(32 \times 48\). Both meshes produced consistent results. The main difference was in the objective function value at the solution; this difference is expected, as the objective function depends on the size of the mesh.

3.3 Protein Folding

In the final example we use OPT++ to minimize the AMBER potential energy of a protein. AMBER is a prominent empirical energy model used in optimization approaches to the problem of predicting the spatial configuration of a protein given its sequence of amino acids, also known as the protein folding problem. All optimization approaches to this problem assume that the natural configuration of the protein minimizes an energy potential, but the exact form of this energy is a current area of research.
The AMBER model considers the distances $r_i$ between pairs of bonded atoms, the angles $\theta_i$ between consecutive bonds, and the dihedral angles $\phi_i$ defined by sequences of four bonded atoms, and assigns positive energies to the deviation from empirical values for these parameters. In addition to these bonded interactions, AMBER includes terms for the electric potential energy and Lennard-Jones interactions between pairs of non-bonded atoms. The form of the AMBER function can then be expressed as:

$$
E_{\text{AMBER}} = \sum_{i=1}^{N_b} \tilde{a}_i (r_i - \bar{r}_i)^2 + \sum_{i=1}^{N_a} \tilde{b}_j (\theta_i - \bar{\theta}_i)^2 + \sum_{i=1}^{N_d} \tilde{c}_i \cos(\bar{m}_i \phi_i - \bar{\gamma}_i) + \sum_{i=1}^{N} \sum_{j=i}^{N} \left[ \frac{q_i q_j}{r_{ij}^2} + \left( \frac{\bar{q}_i \bar{q}_j}{r_{ij}} \right)^6 + \left( \frac{\bar{q}_i \bar{q}_j}{r_{ij}} \right)^{12} \right]
$$

where the first three terms are the bonded interactions, with $N_b$ the number of bonds, $N_a$ the number of angles, and $N_d$ the number of dihedral angles in the protein, and where the last summation incorporates the interactions among pairs $(i, j)$ of nonbonded atoms, with $r_{ij}$ denoting the distance between them. Quantities in (6) marked with an overbar are predefined constants dependent on the specific atoms involved in the interaction.

In this problem, a protein with $N$ atoms is represented as a vector of length $3N$, corresponding to the atom’s Cartesian coordinates. Hence, typical problem sizes are in the thousands, as real proteins are made up of hundreds of amino acids, each with a dozen or so atoms. For some optimization methods, the size of this problem can create difficulties, especially if they either store or solve a size $N$ linear system at each iteration. One method that is more suitable for large-scale unconstrained optimization problems is the Limited Memory BFGS algorithm, or LBFGS, described in Liu and Nocedal [1989]. In this method, information from only a subset of $M$ Hessian columns is kept at any iteration, with $M \ll N$. We have found that a value of $M \approx 30$ is adequate for proteins with a few thousand atoms.

Figure 14 contains the code to minimize AMBER for a 593 atom protein known as 1e0m. The protein is encoded in pdb format, handled by our PDB.h library [Oliva 2003]. The user function $\text{amberInit}()$ returns the starting point for the optimization, and initializes global variables used by $\text{amberEval}()$ to perform the energy computations. In the main routine, we first load the pdb file for the protein, and initialize the problem dimension to three times the number of atoms. The energy function provides first order derivatives, hence we construct a nonlinear problem of type NLF1, which we pass to the constructor of the OptLBFGS optimization object. Before running the optimization algorithm, we adjust the default values of a few parameters with values more suitable for this large-scale problem. After the optimization call, the final status of the algorithm is appended to the iteration output file. The second argument to $\text{printStatus}()$ suppresses printing of the final iterate (a very long vector). Instead, we obtain the final iterate from the nlp object and save it in PDB format.

In Figure 15, we show a plot of the value of the energy after each of the 7224 iterations of the optimization algorithm.
Fig. 14. Protein energy minimization code.

```c
#include "OptLBFGS.h"
#include "PDB.h"
USERINITFCN amberInit();
USEREVALFCN amberEval();
PDB protein;

int main() {
    protein.load("1e0m.pdb");
    int natoms = protein.nofAtoms();
    int ndim = 3 * natoms;
    NLF1 nlp(ndim, amberEval, amberInit);
    OptLBFGS optobj(ndim, nlp);
    // adjust parameters
    optobj.setMem(30);
    optobj.setMaxIter(10000);
    optobj.setFcnTol(1e-6);
    optobj.optimize(); // optimization call
    optobj.printStatus("status", false);
    ColumnVector X = nlp.getXc();
    protein.writepdb("final.pdb", X.Store());
}
```

Fig. 15. AMBER energy per LBFGS iteration.
Fig. 16. Protein initial and final configurations.

Table III. OPT++ Algorithms Available for Different Problem Classes

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>NLF0</th>
<th>PDLNF1</th>
<th>NLF1</th>
<th>NLF2</th>
</tr>
</thead>
<tbody>
<tr>
<td>OptPDS</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>OptGSS</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>OptCG</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>OptLBFGS</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>OptQN</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>OptBCQNNewton</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>OptFDNewton</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>OptFDNIPS</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>OptNewton</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>OptBCNewton</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>OptNIPS</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

The initial and final configurations for the protein are shown in Figure 16. Minimizing the AMBER energy has effectively put the protein in a more compact configuration.

4. SUMMARY

In this article, we have presented a C++ class library for nonlinear optimization. The design is predicated on the concept of distinguishing between an algorithm-independent class hierarchy for nonlinear optimization problems and a class hierarchy for nonlinear optimization methods that is based on common algorithmic traits. We have defined a hierarchy of problems based on the availability of analytic derivative information, as well as a hierarchy of constraints based on the type of constraint. Not only does this design provide users with the ability to describe a nonlinear optimization problem in a general sense, but it also provides the foundation from which one can derive more sophisticated classes of problems, such as those with known mathematical properties or structure. The algorithm class hierarchy is based on inherent similarity between methods, making it easy to develop new algorithms within the existing infrastructure. Furthermore, a simple common interface based on the problem classes makes it easy to switch among algorithms.

In addition to providing a class structure for nonlinear optimization algorithms, we have also implemented a range of algorithms that are accessible through the interface described in this article. Those methods are roughly grouped into the following four categories: 1) direct search methods, 2) conjugate gradient-like methods, 3) Newton-like methods, and 4) nonlinear interior-point methods. Table III summarizes the optimization methods currently implemented, and maps them to the nonlinear problem classes included in OPT++. The application examples discussed in Section 3 demonstrate the utility of OPT++, the simplicity of the interface, and the ease with which these algorithms can be compared. We note that OPT++ is not yet a finished product; however, the success we report in this article, combined with the positive responses received since making it available under the GNU LGPL, serve as evidence that object-oriented programming is a viable tool for developing optimization software.
Future work will concentrate on incorporating new algorithms. We are currently working on implementing new classes for large-scale optimization. The challenges will lie in incorporating an appropriate set of linear solvers in a robust, general, and efficient way. As a secondary focus, we are exploring the use of Java-based GUIs and XML as means of further simplifying the interface for users who are not skilled programmers.

ACKNOWLEDGMENTS

The authors wish to thank Vicki Howle and Suzanne Shontz for their contributions to OPT++. The authors also wish to thank two anonymous reviewers for their many helpful comments and suggestions.

REFERENCES


Received September 2003; revised December 2004, March 2006; accepted May 2006.