An Introduction to the COLIN Optimization Interface

William E. Hart
Sandia National Laboratories
P. O. Box 5800, MS 1110
Albuquerque, NM 87185-1110
wehart@sandia.gov

Abstract

We describe COLIN, a Common Optimization Library INterface for C++. COLIN provides C++ template classes that define a generic interface for both optimization problems and optimization solvers. COLIN is specifically designed to facilitate the development of hybrid optimizers, for which one optimizer calls another to solve an optimization subproblem. We illustrate the capabilities of COLIN with an example of a memetic genetic programming solver.

1 Introduction

A wide variety of software packages have been developed to solve various types of optimization applications (e.g. see More and Wright [14]). Unfortunately, the large number of optimization codes available can make it difficult for a novice to apply any particular optimization solver. Setting up and applying an optimizer requires a user to call a particular sequence of functions, and in some cases the user must create a subroutine to provide an interface between the optimizer and function evaluator. Additionally, it can be quite difficult to compare optimizers because interfaces to different optimization software can be quite different even if the underlying solvers are similar.

A variety of general-purpose nonlinear optimization packages have been developed using object-oriented programming to overcome these obstacles [2, 3, 4, 5, 6, 8, 9, 13, 15], as well as many packages focused on a particular class of solvers (e.g. evolutionary algorithm libraries like GAlib [18] and EO [11]). These packages use objects to provide data abstractions for solvers, objective functions, constraints and other optimization components. These objects provide data abstractions that shield a user from unnecessary details of a particular code, and they help ensure that the optimization software is reliable and easier to reuse in new applications.

In this paper we describe COLIN, a collection of C++ classes that provides a Common Optimization Library INterface. The goal of COLIN is to provide an object-oriented mechanism to (1) encapsulate other optimization solvers, and (2) specify data transformations between user data types and solver data types. COLIN enables the use of optimization solvers provided by *different* software packages in a more seamless manner. Additionally, COLIN facilitates the application of hybrid optimizers like multi-start methods and memetic evolutionary algorithms with a wide range of local optimizers.

The next section provides additional background and motivation for the development of COLIN. Subsequently, we illustrate the capabilities with a series of simple examples. In particular, we show how COLIN can be used to apply traditional, nonlinear optimizers to perform local search on S-expressions searched by genetic programming solvers. Finally, we discuss some outstanding issues with the design and application of COLIN.

2 Background and Motivation

Object-oriented programming is a modern approach to support code modularity and reusability. Many programming languages support object-oriented constructs, though C++ is a particularly popular choice for optimization software. C++ offers powerful object-oriented abstractions, and it is readily interfaced with user-application software in languages like C and Fortran. Further, C++ can be run on virtually all hardware platforms, including parallel architectures like Sandia's Cplant machine, which only supports a re-

stricted operating system kernel.

Object-oriented languages like C++ provide support for encapsulation, sharing, and inheritance of code. In particular, encapsulation allows the external aspects of an object, which are accessible to other objects and users, to be separated from its internal implementation details. This helps ensure that objects are used in a robust manner. Thus encapsulation facilitates the development of software components that can be robustly reused in different contexts.

Object-oriented software design offers many advantages for optimization software (e.g, see Meza [13]). First, object-oriented design can provide a simpler and more consistent user interface for optimization libraries. Many of the C++ software packages mentioned above use objects to distinguish between the problem and the method used to solve the problem. This allows users to focus on creating a problem object that represents their application and then selecting a solver object that can be used to perform optimization. Further, the use of a consistent interface for problem and solver objects allows the user to focus on the specific application rather than struggling to understand the requirements of various optimization components. This has the added advantage that a consistent design will encourage users to apply a variety of optimization solvers to their applications.

A second advantage is that object-oriented design provides a better program interface for the user. The encapsulation provided by objects can protect users from inappropriately applying these tools. More fundamentally, software based on object-oriented designs is to be preferred to optimization software designs like reverse-communication, which violates several good programming practices (e.g. single-entry single-exit).

A third advantage is that object-oriented software can facilitate the rapid development of new optimization algorithms. The software components used to develop methods like quasi-Newton solvers and evolutionary algorithms can often be modularized. This allows algorithmic designers the ability to consider, implement and compare a wide range of related solvers. The MOOCHO library is a particularly good example of how object-oriented programming can be leveraged in this fashion [1].

A related advantage of object-oriented design is that software components can be reused within different solvers. For example, different line search methods are useful in different contexts. However, by implementing these as objects we can apply any of a number of nonlinear optimization methods that are based on line search in a very flexible manner.

These observations have motivated the development of an increasing number of object-oriented optimization software packages (e.g. see [1, 2, 3, 4, 5, 6, 8, 9, 13, 11, 15, 18]). Unfortunately, each of these packages provides slightly different abstractions for optimization components. Consequently, though we can argue that these advantages exist for any given package, the reality is that users are still faced with the prospect of applying solvers from multiple software packages, each of which potentially has a different programming interface.

A similar issue arises when you consider the challenge faced by algorithm designers who are developing hybrid optimizers (e.g. simple multi-start local search). Although object-oriented optimization software provides valuable encapsulation of optimizers, the interoperability promised by object-oriented design is not achieved in this context because of the different abstractions used by different packages. Thus the application of multi-start local search with different local search methods may require the algorithm designer to support interfaces to different external software packages (e.g. the DAKOTA toolkit [6] includes interfaces to packages like DOT, NPSOL, OPT++, MOOCHO, SGOPT and PICO).

The software components in COLIN were designed to address these two issues by providing C++ optimization components that can be used to easily provide wrappers for other optimization solvers. COLIN provides mechanisms for specifying data transformations between user data types and solver data types, which clearly distinguishes the role of COLIN as providing wrapper mechanisms. Consequently, COLIN facilitates the application of hybrid optimizers like multi-start local search when (a) the global solver applies a COLIN optimizer and (b) COLIN wrappers have been developed for a variety local optimizers.

3 COLIN Overview

There are three main C++ classes provided by COLIN:

- **OptSolver**: defines the procedure for performing optimization on a problem
- **OptProblem**: provides the overall problem definition to a solver object
- **OptApplication**: an internal class visible to a problem object that contains the information

about how the objective and constraint functions are to be computed

Our description of COLIN will focus on the programming interface that these classes provide a user. In particular, we omit examples of how COLIN can be used to provide a wrapper for an optimization solver.

3.1 Basic Functionality

Figure 1 illustrates how COLIN solver and problem classes are used to perform optimization. COLIN supports the use of a user-defined function **func** that defines the objective (and possibly constraints). The COLIN **OptProblem** class is constructed using a template argument to specify the typename of a point; in Figure 1 the type is **vector<double>**. **OptSetup** is an overloaded function that initializes a problem class with information about how the objective (and constraint) functions are to be computed. In particular, **OptSetup** creates an **OptApplication** object for the user.

In addition to initializing a problem with an **OptApplication** object, a user can set options within a problem using **ClassOptions** methods; **OptProblem** is a subclass of **ClassOptions**. The **set_option** method takes a string argument that specifies the option. The second arguement to **set_option** is an arbitrary object that represents the value of the option.

In Figure 1, the **sMC** class is a subclass of the COLIN class **OptSolver**. A COLIN solver class may also be specified with a template argument to specify the typename of a point. The **set_problem** method is used to tell a solver object the problem to be optimized. **OptSolver** is also a subclass of **ClassOptions**, so solver options can be set with the **set_option** method. After the solver has been setup, the **reset** method must be used to initialize internal data structures and to check for errors in the solver configuration.

Finally, the **minimize** method is used to apply a solver to perform minimization on a problem. After optimization is completed, several methods like **min_val** can be used to examine the optimal solution.

3.2 Mapping Domains

Figure 2 illustrates how COLIN supports the application of solvers with problem classes defined over different problem domains. When a solver object is setup with a problem object defined over a different domain, evaluating trial points generated by the optimizer requires that they be mapped into the domain type of the problem object. COLIN employs

```
// A  test function
double func(vector<double>& point);

// Create a problem class
OptProblem<vector<double> > prob;
// Call 'OptSetup' to initialize 'prob'
// with the function pointer
OptSetup(prob, func);
// Set other class options in 'prob'
prob.set_option("domain","[-1.0,1.0]^3");

// Create a solver class
sMC<vector<double> > opt;
// Tell the solver which problem to solve
opt.set_problem(prob);
// Set other class options in 'opt'
opt.set_option("max_neval",100);
// Reset the solver to prepare it for
// minimization
opt.reset();

// Perform minimization
opt.minimize();
// Print the best value after optimization
cout << opt.min_val() << endl;
```

Figure 1: The basic steps needed to create, initialize and apply OptSolver and OptProblem classes.

the **map_domain** function to perform this mapping, which can be specialized using C++ template specialization mechanisms.

Although domain mapping may simply appear a convenience for users, this capability is essential for the application of optimization solvers in a generic fashion. If COLIN is used to wrap optimizers in different packages, the COLIN wrapper classes may naturally be defined using different domain types. Thus COLIN's facility for domain mapping allows a user to define a single problem object that can be solved using these different COLIN wrapper classes.

3.3 Hybrid Solvers

Hybrid optimization solvers combine two or more different optimization methods so as to exploit their complementary capabilities. A classic application of hybrid optimizers is for global optimization, where it is well-known an effective design for randomized global optimizers is to combine global random search with local refinement [17]. Effective local search methods have been developed for many application domains, so local search can be used to refine the solutions gener-

```
// Mapping function
template<>
void map_domain(vector<double>& x,
                const array<double>& y)
{ for (int i=0; i<y.size(); i++)
    x[i] = y[i]; }

// Problem with domain vector<double>
OptProblem<vector<double> > prob;

// Solver with domain array<double>
sMC<array<double> > opt;
// Set a problem with a domain that is
// mapped by the map_domain() template
// function
opt.set_problem(prob);
```

Figure 2: Interfacing problem and solver objects constructed with different domain types.

ated by global random search methods. This algorithmic idea is employed in simple multi-start local search, as well as more sophisticated methods like topological search and memetic evolutionary algorithms.

COLIN facilitates the development and application of hybrid solvers by (a) providing a simple, general interface for solvers and (b) providing the domain mapping functionality that enables solvers from different packages to communicate with each other. Figure 3 illustrates this capability using the **sMC** solver to perform global search and the **PatternSearch** solver to perform local search. Note that these solvers are defined over different domains, so the **map_domain** template function needs to be implemented. However, mapping needs to be done in both directions: (a) a global trial point is mapped into a local solution before it is locally refined, and (b) the final local solution generated by the local optimizer is mapped back into the global solution type.

3.4 Domain Traits

COLIN clearly supports the use of opaque domain types through the application of the **map_domain** facility. In addition, COLIN offers the ability to define features of a user-defined domain type using C++ traits. Figure 4 illustrates how the **OptDomainTraits** class is specialized to indicate that derivatives can be computed with the **UserDomain** data type because there are real-valued parameters available within **UserDomain** objects.

This facility is particularly valuable when applying hybrid optimizers. In this case, domain traits provide a mechanism for signaling the features of a problem that are supported (e.g. linear constraints). Additionally, domain traits can be used to perform error checks to verify that particular hybrids are valid (e.g. to verify that gradients are available for gradient-based optimizers).

```
// Mapping function
template<>
void map_domain(vector<double>& x,
                const array<double>& y)
{ for (int i=0; i<y.size(); i++)
    x[i] = y[i]; }

// Mapping function
template<>
void map_domain(array<double>& x,
                const vector<double>& y)
{ for (int i=0; i<y.size(); i++)
    x[i] = y[i]; }

// Local solver with domain
// vector<double>
PatternSearch<vector<double> > local;

// Global solver with domain
// array<double>
sMC<array<double> > global;
// Setup the global solver to use the
// local solver
global.set_solver(local);
```

Figure 3: Interfacing two solver objects to form a hybrid solver.

4 An Application Example

We illustrate the utility of COLIN by describing its application for the design of a memetic evolutionary algorithm. The evolutionary algorithm we consider is a genetic program [12], for which the search domain is the set of s-expressions defined over some alphabet. This alphabet includes constants (typically floating point values), n-ary operators and variables. Thus an s-expression defines a computational graph that can be evaluated given the input values of the variables. Figure 5 illustrates a simple s-expression that expresses a simple algebraic expression.

We consider the application of a genetic program to solve a simple nonlinear least squares problem. Let $\{(x_1, y_1), \ldots, (x_k, y_k)\}$ be a set of input-output pairs for which we wish to build a model $f(x, c_1, \ldots, c_n)$,

```
// A user-defined domain type
class UserDomain {
  vector<double> x;
  vector<int> y;
  };

// Define domain traits to show that
// UserDomain contains real parameters
template <>
bool OptDomainTraits<UserDomain>::
         reals=true;

// Create a problem class
double func(UserDomain& point);
OptProblem<UserDomain > prob;
OptSetup(prob, func);

// Evaluate the partial derivative of
// a point with respect to real
// parameters.
UserDomain point;
vector<double> gradient;
prob.EvalG(point,gradient);
```

Figure 4: Evaluate the partial derivative with a user-defined domain type.

where the values c_i are selected to minimize a least squares criterion:

$$\min_{c_i \in \mathbf{R}} \sum_{i=1}^{k} (y_i - f(x_i, \overline{c}))^2 .$$

Genetic programming methods can evolve s-expressions to identify a function f along with the values for the parameters $\overline{c}$.

We consider $\omega = \{+, -, \cdot, /\}$, a set of binary operators

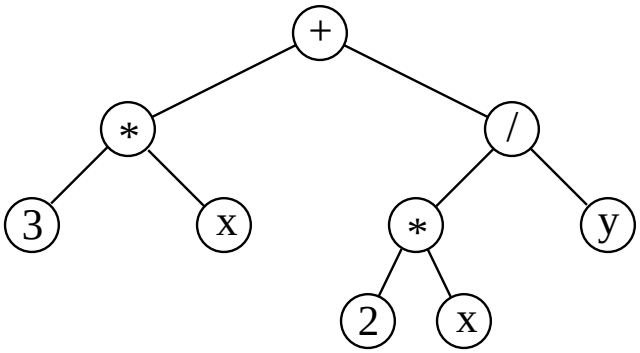

Figure 5: An s-expression that describes the algebraic expression $3x + 2x/y$.

used to defined s-expressions, and $\tau = \{x, c_1, \ldots, c_n\}$, a set of terminals used in these s-expressions. Let $\mathcal{S}_{\omega,\tau}$ be the set of possible s-expressions that can be represented. Thus our goal is to solve the problem

$$\min_{g \in \mathcal{S}_{\omega,\tau}} \sum_{i=1}^{k} (y_i - g(x_i))^2 .$$

Topchy and Punch [16] note that this problem can be decomposed into a search for the functional form of the optimal s-expression and a search for the values c_i, which are constants in the s-expression. Further, they note that in applications like nonlinear least squares, the objective function is differentiable with respect to the values c_i if suitable operators are chosen in ω. Topchy and Punch demonstrate that a memetic genetic programming algorithm using a simple gradient-based local search method is more effective than a standard genetic programming solver.

An obvious extension of this work is to consider memetic genetic programming methods using alternative local search techniques. There are a wide variety of gradient-based nonlinear optimizers that could be applied to this application, and in particular the Gauss-Newton and Levenberg-Marquardt methods are well suited for nonlinear least squares applications (particularly when the least squares objective of the optimal solution is not large [7]). Unfortunately, it would be an onerous task to couple standard genetic programming software with numerical optimization packages that support these types of solvers. COLIN simplifies the development of this hybrid through the use of domain mapping and domain traits; the programming interface for a user would be a combination of the examples in Figures 3 and 4.

5 Discussion

The design goal for COLIN has been to provide optimization middleware that can facilitate the use of (a) hybrid optimizers and (b) conventional optimizers on subdomains of more generic search spaced (e.g. the application of nonlinear optimizers to s-expressions). The class structure of COLIN makes rather sophisticated use of C++ template capabilities to provide this capability. Thus while the user interface for COLIN is quite simple, the underlying mechanisms are rather complex.

Consequently, a limitation of COLIN is that it may be difficult to extend it to support new solver interfaces that are not part of the existing class interfaces. For example, solvers for specific domains may need information about the problem structure that might be

difficult to support within COLIN in a generic fashion. Additionally, support for new data about the solution may be difficult to integrate without adapting several core COLIN classes; the structure of the classes in COLIN does not facilitate the use of C++ inheritance to achieve this goal.

COLIN current is composed of a collection of about 20 C++ files, which employ STL objects for standard data types. Additionally, COLIN depends on the UTILIB utility library [10] for miscellaneous support of error management, mathematical routines and parallel communication tools. COLIN is currently being developed within the Coliny optimization library, which provides COLINized versions of the optimizers in SGOPT [9], and we are working on making COLIN and Coliny available for public distribution and development through the Lesser Gnu Public License.

Acknowledgements

We are grateful to Mike Eldred for his many helpful discussions. Sandia is a multipurpose laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

References

[1] R. Bartlet. MOOCHO Multifunctional Object-Oriented arCHitecture for Optimization: Users guide. Technical report, Sandia National Laboratories, 2003. in preparation.

[2] S. J. Benson, L. C. McInnes, and J. J. More. GPCG: A case study in the performance and scalability of optimization algorithms. *ACM Transactions on Mathematical Software (TOMS)*, 27(3):361 – 376, 2001.

[3] D. L. Bruhwiler, S. G. Shasharina, J. Cary, and D. Alexander. Design and implementation of an object oriented C++ library for nonlinear optimization. In M. E. Henderson, C. R. Anderson, and S. L. Lyons, editors, *Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing*, pages 165–173, 1998.

[4] H. L. Deng, W. Gouveia, and J. Scales. The CWP object-oriented optimization library. Technical report, Colorado School of Mines, 1994.

[5] H. L. Deng, W. Gouveia, and J. Scales. An object-oriented toolbox for studying optimization problems. In *Inverse Methods*. Springer-Verlag, 1996.

[6] M. S. Eldred, A. A. Giunta, B. V. B. Waanders, J. Wojtkiewicz, Steven F., W. E. Hart, and M. P. Alleva. DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 3.0 users manual. Technical Report SAND2001-3796, Sandia National Laboratories, 2001.

[7] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, 1981.

[8] M. S. Gockenbach, M. J. Petro, and W. W. Symes. C++ classes for linking optimization with complex simulations. *ACM Transactions on Mathematical Software*, 25:191–212, 1999.

[9] W. E. Hart. SGOPT user manual version 2.0. Technical Report SAND2001-3789, Sandia National Laboratories, 2001.

[10] W. E. Hart. UTILIB user manual version 1.0. Technical Report SAND2001-3788, Sandia National Laboratories, 2001.

[11] M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving objects: A general purpose evolutionary computation library. In *Proc Evolution Artificielle*, 2001.

[12] J. Koza. *Genetic Programming*. MIT Press, 1993.

[13] J. C. Meza. OPT++: An object-oriented class library for nonlinear optimization. Technical Report SAND94-8225, Sandia National Laboratories, 1994.

[14] J. J. More and S. J. Wright. *Optimization Sofiware Guide*. SIAM Press, Philadelphia, PA, 1993.

[15] A. Raggl and W. Slany. A reusable iterative optimization library to solve combinatorial problems with approximate reasoning. *International Journal of Approximate Reasoning*, 19(1-2):161–191, July/August 1998.

[16] A. Topchy and W. Punch. Faster genetic programming based on local gradient search of numeric leaf values. In *Proc Genetic and Evolutionary Computation Conf*, pages 155–162, San Francisco, 2001. Morgan Kaufmann.

[17] A. Törn and A. Žilinskas. *Global Optimization*, volume 350 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[18] M. Wall. GAlib: A C++ library of genetic algorithm components, 1998. http://lancet.mit.edu/ga/.