# The OptSolve++ Software Components for Nonlinear Optimization and Root-Finding

David L. Bruhwiler[1], Svetlana G. Shasharina[1], John R. Cary[1,2]

[1] Tech-X Corporation, 1280 28th Street, Suite 2, Boulder, Colorado 80303, U.S.A.
bruhwile@txcorp.com, sveta@txcorp.com, cary@txcorp.com
http://www.techxhome.com
[2] University of Colorado, Physics Department, Boulder, Colorado 80309-0390, U.S.A.
cary@colorado.edu
http://www-beams.colorado.edu/~cary/cary.html

**Abstract.** OptSolve++ is a set of C++ class libraries for nonlinear optimization and root-finding. The primary components include `TxOptSlv` (optimizer and solver classes), `TxFunc` (functor classes used to wrap user-defined functions), `TxLin` (linear algebra), and a library of test functions. These cross-platform components use exception handling and encapsulate diagnostic output for the calling application. Use of the "template" design pattern in `TxOptSlv` provides a convenient interface that allows for interchange of existing algorithms by users and straightforward addition of new algorithms by developers. All classes are templated, so that optimization over the various floating-point types uses the same source code, and so that optimization over integers or more esoteric types can be readily accommodated with an identical interface. `TxOptSlv` and `TxFunc` use the template-based "traits" mechanism, allowing them to work with any user-specified container class, and future versions of OptSolve++ will allow users to interchange `TxLin` with other linear algebra libraries.

## 1 Introduction

OptSolve++ is a set of object-oriented C++ class libraries for nonlinear optimization and root-finding. Each library is designed as a software component that can be embedded in other software and extended via inheritance to add new capabilities. The primary components include TxOptSlv (the optimizer and solver classes), TxFunc (functor classes used to wrap user-defined functions), TxLin (linear algebra classes), and a library of standard test functions. These components compile on PC, Macintosh, GNU/Linux and all major Unix platforms. Use of exception handling and the encapsulation of all diagnostic output makes them robust and reliable.

## 2 The TxOptSlv Library for Nonlinear Optimizers and Solvers

The TxOptSlv library provides a convenient and flexible interface for nonlinear optimization and root-finding with user-specified functions, providing an extensible class hierarchy that allows developers to readily implement new algorithms or interface to existing C and C++ routines.

### 2.1 The Interface

The primary method of the top level interface is `solve()`, which in an optimizer class will attempt to find the minimum of the specified merit function and, in a solver class, will attempt to find a multidimensional zero of the specified vector-valued function. The `solve()` method calls `reset()`, which returns the optimizer or solver object to its initial state, and `continue()`, which can be called in place of `solve()` if the user does not wish the object to be reinitialized:

```
void solve() throw(TxOptSlvExcept)
{ reset();
  continue(); }
```

In the event of a bad function evaluation, failure to converge, or other problems, this method will throw an exception of type `TxOptSlvExcept`.

The next code segment shows how `continue()` loops over three basic methods: `isSolved()` checks whether the algorithm has converged, `step()` carries out one optimization or solver step, and `prepareStep()` handles any tasks required before the next step is taken. The results are stored by `setResult()`, whether or not the algorithm converges. If the maximum number of iterations is exceeded, then an exception is thrown.

This interface works well for a wide variety of 1-D and multidimensional optimization and root-finding algorithms. While the `solve()` method is fixed, the other methods are virtual, so new algorithms can be added by overloading the virtual building-block methods. Existing implementations of algorithms can be included within this interface by overloading the `continue()` method to call the desired implementation, then overloading the building-block methods, to either throw an exception or else call some appropriate method of the implemented algorithm.

```
void continue() throw (TxOptSlvExcept)
{ // loop over a specified number of iterations
  for (numIter=0; numIter<maxNumIter; numIter++) {
    // check to see if algorithm has converged
    if ( isSolved() ) {
      setResult();         // store successful results
      return;              // return
    }
    // take one optimizing step
    step();
    // prepare for the next step
    prepareStep();
  }   // maximum number of iterations was exceeded
  setResult();          // store unsuccessful results
  throw TxOptSlvExcept("Failure to converge");  }
```

### 2.2 The Class Hierarchy

Figure 1 shows a simplified class diagram for the `TxOptSlv` class library. The classes are templated over both the argument type (`ArgType`) and the return type (`RetType`) of the function to be optimized. This allows for situations where, for example, double precision arguments are used to avoid round-off error, but a single precision value is returned, or perhaps `ArgType` could be binary, integer or complex, while `RetType` could be of floating point type. When `ArgType` and `RetType` are both of floating point type, it is expected that `ArgType` will be of equal or higher precision than `RetType`.

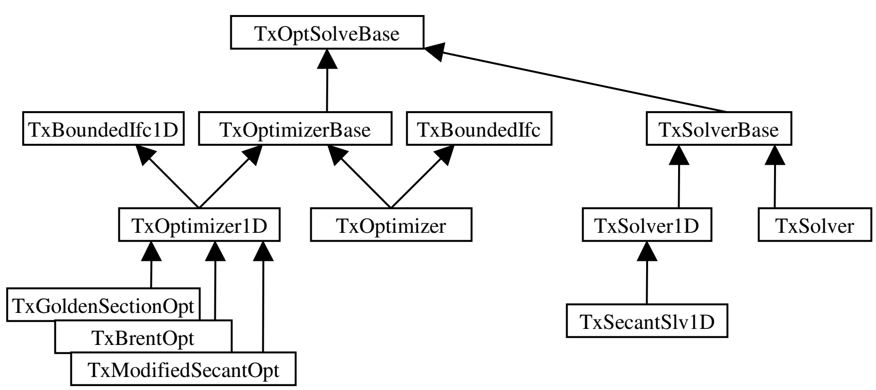


**Fig. 1** Simple class diagram showing the abstract classes of the TxOptSlv class library, plus the concrete classes that implement 1-D algorithms.

From the `TxOptSolveBase` class, which defines the top-level interface described above, the hierarchy splits into optimizers and nonlinear solvers. From the `TxOptimizerBase` and `TxSolverBase` class, the two sub-hierarchies branch into 1-D and multidimensional classes. Figure 1 shows the 1-D algorithms that have been implemented at present. Figure 2 shows the multidimensional optimization algorithms that have been implemented. The multidimensional solvers are not discussed here, due to space limitations.

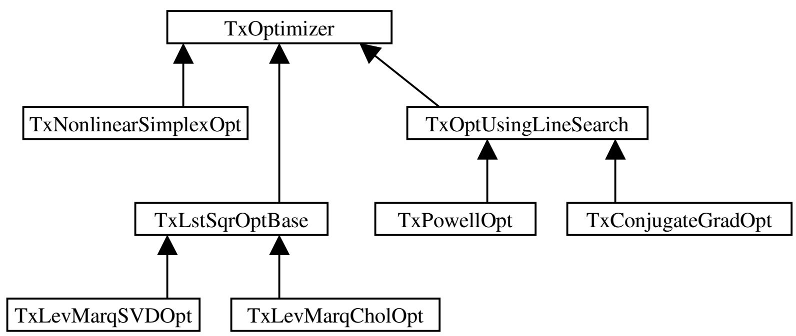


**Fig. 2** Simple class diagram for the multidimensional nonlinear optimization classes of the TxOptSlv class library.

The TxOptSlv classes provide constructors that internally wrap the function to be optimized inside an appropriate TxFunc functor object. This approach allows users to instantiate an optimization object with a variety of function pointer types, while preserving the common interface provided by the functors. Users do not require any knowledge of the TxFunc library, if they can provide an appropriate C++ function pointer or pointers.

### 2.3 The Algorithms

A number of nonlinear optimization algorithms have already been implemented in TxOptSlv. The 1-D algorithms include golden section [1] and Brent [2], which do not need the function derivative, and a modified secant algorithm [3], which does require access to the derivative. The multidimensional algorithms include nonlinear simplex [4] and Powell [5], which do not need the gradient of the function, and the conjugate gradient method [1], [6], which does require the gradient. The `TxOptimizer` class provides an implementation of bound constraints (using a variable transformation approach), which is available to any unconstrained optimization class that inherits from it. General nonlinear constraints, using a penalty function approach [1], [6], [7], will be made available to all algorithms in the near future.

We have also implemented algorithms for linear and nonlinear least squares. For the linear least square problem,

$$\min\left\|A * x - b\right\|_2 \ , \tag{1}$$

we have implemented methods using Cholesky, QR and singular value (SVD) decompositions [8], [9], which are encapsulated in the `TxMatrixOpt` class. The advantage of SVD is that is allows users to solve singular or close to singular systems of equations, as well as underdetermined problems. For nonlinear least squares,

$$\min\left\|f(x)\right\|_2, \qquad f(x) = 0.5 * \sum r_i^2(x) \ , \tag{2}$$

we implemented the Levenberg-Marquardt algorithm, including quadratic constraints, using either Cholesky decomposition or SVD [10], [11].

## 3 The TxFunc Library of Functors

A functor object represents a generalized function, returning a unique value associated with a set of input arguments. Functors can also contain auxiliary data or methods. This concept is used in the C++ standard library (see e.g. Sec.'s 11.9 and 18.4 of Ref. [12]). A functor can be used to wrap one or more function pointers, and it can be used to scale or to place constraints on a wrapped function. Most generally, a functor object can wrap an arbitrary sequence of calls to user-specified libraries, in order to calculate a value associated with the input arguments.

Figure 3 shows a simplified class diagram for the `TxFunc` class library. The abstract base class `TxFunctorBase` defines the fundamental interface for a functor, which consists primarily of the parentheses operator, (), declared pure virtual and templated over both the argument type (`ArgType`) and the return type (`RetType`). The hierarchy then branches into 1-D and multidimensional functors, very much like the optimizer and solver subhierarchies shown in Figure 1. Some of the concrete classes, which wrap specific types of function or functor pointers, are also shown in Figure 3.

For multidimensional functors with no specified derivatives, the `TxNoDerivPtrFunc` class defines an internal function pointer through a new template parameter, `FunPtrType`, which must be a pointer to a C++ function, a `TxFunctor` or some other object for which the parentheses operator () is appropriately defined. The `TxDualPtrFunctor` and `TxManyPtrFunctor` classes wrap different combinations of C++ function pointers in order to represent both the function value and its gradient.

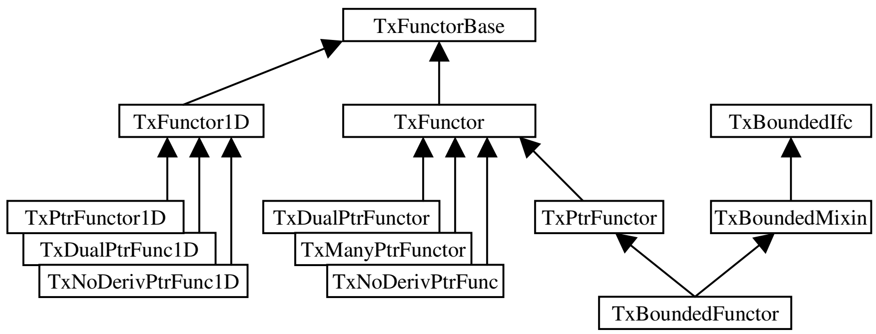


**Fig. 3.** Simple class diagram for the TxFunc class library, showing the abstract 1-D and multidimensional functor classes, as well as some of the concrete classes.

The `TxPtrFunctor` class, also shown in Fig. 3, wraps a function or functor pointer that can simultaneously calculate the full gradient and return the function value. This class is also used as an intermediate base class for all functors that apply a variable transformation to the argument. For example, the `TxBoundedFunctor` uses a variable transformation to impose bound constraints on the argument of the interior functor. This functor implements the `TxBoundedIfc` interface class, which is also implemented by the `TxOptimizer` class, as shown above in Figure 1.

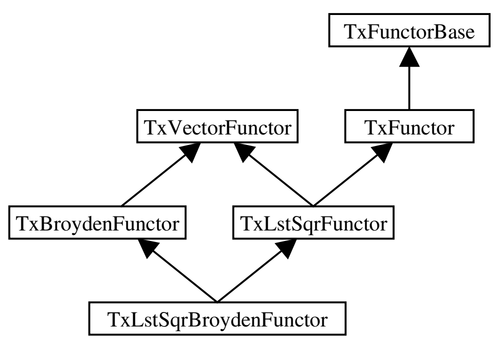


**Fig. 4.** Class diagram for the vector and least square functor classes of the `TxFunc` library.

Figure 4 shows the vector and least squares functors of the `TxFunc` library. The `TxVectorFunctor` class holds an array of `TxFunctor` pointers and is used to define a vector-valued function for use in root-finding by the multidimensional nonlinear solver classes. The `TxLstSqrFunctor` class inherits multiply from `TxFunctor` and `TxVectorFunctor`, so that it can be used polymorphically by the `TxLevMarqCholOpt` and `TxLevMarqSVDOpt` classes for Levenberg-Marquardt optimization -- the parentheses operator () is overloaded to return the sum of the square of the function values -- and also provide direct access to each of the individual functions. The Broyden functors use Broyden's method to estimate derivatives for functions that do not provide derivatives.

## 4 Template-Based "Traits" Mechanisms

The template-based "traits" mechanism [13], [14] allows C++ developers to write code for a templated class that refers to traits of the various numeric types in a very general way, making it possible to code numerical algorithms that with one piece of source code that works equally well for types `float`, `complex<double>`, `double` and so on. In OptSolve++ traits are implemented via templated structs. The upper portion of Fig. 5 shows how various "unary" traits have been defined for `double` and `complex<double>` types. The lower portion of the figure shows how the "binary" trait `PromoteType` (result of combining two different numeric types in a binary arithmetic operation) has been specified for the combination of `float` and `double` types. The traits definitions are contained in a header file distributed with the libraries, so users are free to modify them.

```
struct TxUnaryNumberTraits<double>
---
typedef double RealType;
typedef double AbsType;
typedef complex<double> ComplexType;
static double Norm(double x) {return x*x;}
static double Abs(double x) return abs(x);}
static double Real(double x){return x;}
static double Imag(double x){return 0;}
static inline AbsType Epsilon() {return AbsType(DBL_EPSILON);}
```

```
struct TxUnaryNumberTraits<complex<double> >
---
typedef double RealType;
typedef double AbsType;
typedef complex<double> ComplexType;
static double Norm(complex<double> x){ return norm(x);}
static double Abs(complex<double> x){ return sqrt(norm(x));}
static double Real(complex<double> x){ return real(x);}
static double Imag(complex<double> x){ return imag(x);}
static inline AbsType Epsilon() {return AbsType(DBL_EPSILON);}
```

```
struct TxBinaryNumberTraits<float, double>
---
typedef double PromoteType
```

**Fig. 5.** Examples of the OptSolve++ implementation of unary (above) and binary (below) traits, allowing for flexible templating over numeric types.

The traits idea and mechanism can be extended to create "container-free" C++ libraries [15], which allow the user to specify what sort of C++ "container" class the library must use. As for the traits of numeric types, container traits must be specified in a header file. The use of 1-D array classes in OptSolve++ is container-free in this sense, so the user is free to choose the type of the array argument taken by the multidimensional functor, optimizer and solver classes.

```
struct TxUnaryContainerTraits<std::vector<ArgType> >
---
typedef ArgType ValueType;
static void setSize(std::vector<ArgType>& vec, unsigned int dim)
   {  vec.resize(dim);   }
static unsigned int getSize(std::vector<ArgType>& vec) const
   {  return vec.size();  }
```

```
struct TxBinaryContainerTraits<std::vector<ArgType>,RetType>
---
typedef typename TxBinaryNumberTraits<ArgType, RetType> ::
                                        PromoteType   PromoteType;
typedef std::vector<PromoteType>  PromoteVectorType;
```

**Fig. 6.** Examples of the OptSolve++ implementation of unary (above) and binary (below) container traits, allowing the `TxOptSlv` and `TxFunc` components to be "container free".

As shown in Figure 6, container traits in OptSolve++ are implemented via templated structs. The upper box uses the std::vector<ArgType> as an example, showing that any container is expected to know the numeric type of the objects it contains and is also expected to be able to report its size and to resize itself. Also, containers are expected to allow clients to get and set individual elements through the square brackets operator. The use of "partial specialization" allows these traits to be defined while leaving the ArgType template parameter completely general. The lower box in Figure 6 shows how the binary numeric traits shown in Figure 5 can be used to define binary container traits.

## 5 The TxLin Linear Algebra Library

Linear algebra is a cornerstone for mathematical libraries. OptSolve++ comes with a linear algebra component, `TxLin`, which is required by some of the optimization and solver algorithms. One distinguishing feature of `TxLin` is that it uses reference counting [16], which allows for efficient handling of large matrices by allowing multiple instances of matrices with the same data to share the same region of memory for as long as possible. The public interface is presented by the "envelope" classes, named with a `Tx` prefix, while the implementation is hidden in the parallel hierarchy of "letter" classes, named with a `Pr` (private) prefix.

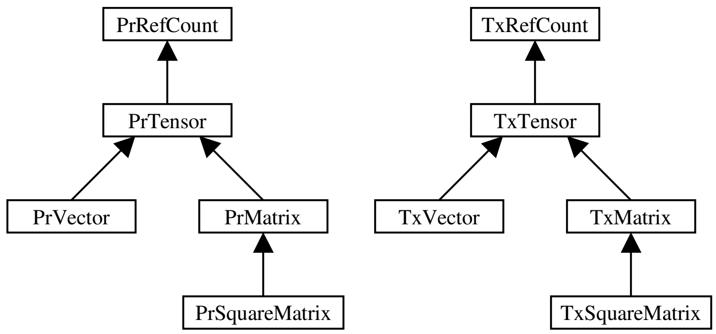


**Fig. 7.** Simple class hierarchy for the reference-counted TxLin linear algebra library.

Figure 7 shows the parallel letter and envelope hierarchies for `TxLin`. The underlying reference counting mechanism is set up in the `Pr/TxRefCount` classes, while the `Pr/TxTensor` classes define a general tensor class that is templated over both the dimension and the numeric type of the elements. The `Pr/TxVector` classes specialize to the case of a 1-D tensor, while the `Pr/TxMatrix` classes specialize to the case of a 2-D tensor, and the `Pr/TxSquareMatrix` classes further specialize to the case of a 2-D tensor with an equal number of rows and columns. The `TxLin` vector and matrix classes overload the usual mathematical operators for matrix and vector arithmetic, and they implement many of the standard linear algebra operations.

At present, TxOptSlv explicitly uses TxLin. Future versions of OptSolve++ will abstract this connection, so that users can replace TxLin with other C++ linear algebra class libraries. This abstraction may be accomplished through the container-free techniques described above, although this becomes syntactically very awkward and is also dangerous in that one must then assume a great deal regarding the capabilities of the other libraries that might be used. Thus, we see a strong need for some degree of standardization in both the interface and the minimum capability of C++ linear algebra components. There has been some informal discussion of possible standards [17], but no consensus has yet emerged.

## Acknowledgements

The authors wish to thank D. Alexander and K. Luetkemeyer of Tech-X Corporation for discussions regarding design and interface, for contributions to early versions of the library, for assistance in software testing and in porting to new platforms, and for comments on drafts of this paper. This work is supported by the U.S. Department of Energy, under grant number DE-FG03-96ER82292, and by Tech-X Corporation.

## References


1. E. Polak, Computational Methods in Optimization, (Academic Press, 1971).
2. R. Brent, Algorithms for Minimization without Derivatives, (Prentice-Hall, 1973).
3. W. Press, B. Flannery, S. Teukolsky and W. Vetterling, Numerical Recipes, (Cambridge University Press, 1989).
4. J. Nelder and R. Mead, Computer Journal 7 (1965), p. 308.
5. D. Himmelblau, Applied Nonlinear Programming, (McGraw-Hill, 1972), p. 167.
6. R. Fletcher, Practical Methods of Optimization, 2nd edition (John Wiley & Sons, 1987).
7. J. Moré and S. Wright, Optimization Software Guide, (SIAM, 1993).
8. L. Trefethen and D. Bau, III, Numerical Linear Algebra, (SIAM 1997).
9. J. Stoer and R. Bulirsch, Introduction to Numerical Analysis, (Springer-Verlag, 1980).
10. A. Bjorck, Numerical Methods for Least Square Problems, (SIAM 1996).
11. J. Moré, The Levenberg-Marquardt Algorithm: Implementation and Theory, in Numerical Analysis, Lecture Notes in Mathematics 630 (Springer-Verlag, 1978), p. 105.
12. B. Stroustrup, The C++ Programming Language, 3rd edition (Addison-Wesley, 1997).
13. N. C. Meyers, Traits: a New and Useful Template Technique, C++ Report 7 (June, 1995).
14. T. Veldhuizen, Using C++ Trait Classes for Scientific Computing, (March, 1996), available on-line at URL http://oonumerics.org/blitz/traits.html.
15. G. Furnish, Container-Free Numerical Algorithms in C++, Computers in Physics 12 (May, 1998).
16. S. Meyers, More Effective C++, 2rd edition (Addison-Wesley, 1998).
17. The html archive for the OONSTD (Object-Oriented Numerics Standards) mailing list, on-line at URL http://oonumerics.org/oon/oonstd/archive.