

# OptSolve++

OptSolve++™

User's Guide

Version 2.0

Tech-X Corporation  
5621 Arapahoe Avenue, Suite A  
Boulder, CO 80303  
<http://www.txcorp.com>  
[info@txcorp.com](mailto:info@txcorp.com)



# 1 Introduction

OptSolve++ is a suite of flexible and extensible software components (object-oriented C++ libraries) for nonlinear optimization of user-defined merit functions. The present release of OptSolve++ features:

- TxOptSolv, a library of nonlinear optimization algorithms
- TxLin, a library of linear algebra algorithms
- TxFunc, a library of unary functors that can wrap or define user-specified merit functions; and
- TxBase, a library of general utilities.

This user manual concentrates on the optimization and function abstraction features of the suite. For more specific information on the API of TxBase and TxLin, please refer to <http://www.txcorp.com>.

This is the second release of OptSolve++ (v2.0), which has an improved class hierarchy. It was derived from the first alpha version (V1.0a1), which is in turn derived from the second alpha release of the predecessor LION++ (Library for Object-oriented Numerics). OptSolve++ can be obtained via the world wide web from URL <http://www.txcorp.com/products/optsolve>.

## 1.1 Optimization with OptSolve++

How OptSolve++ is used, in a nutshell:

1. Download the OptSolve++ library and install it on your platform.
2. Define a merit function you wish to optimize. This can be a single- or multi-dimensional function, piecewise functions, or many other possibilities. You may also want to find an analytic derivative if it possible, as many optimizers can use this added information.
3. Implement a functor using OptSolve++. Functors are C++ objects that describe mathematical functions.
4. Choose an optimizer appropriate for your problem. This involves considering the type of function, the cost of evaluating the function, and your speed and accuracy criteria.
5. Create a method that instantiates an optimizer and finds the minimum of your merit function. This may be done in a stand-alone program or as part of a larger project.

This manual will help you with the installation of OptSolve++, how to define functors, and how to select and use an optimizer appropriate to your problem.

## 1.2 Philosophy of Release

This release of OptSolve++ changes the way functors are created and derived. These modifications result in an easier to use library for the end user, as well as making it easier for developers to extend. Some key changes are outlined below.

1. The functor API is lighter weight, simpler, and easier to use. Merit functions can be created simply by subclassing one of several interface classes and providing `operator()` and (optionally) a derivative function such as `getGrad()`. Function pointers may still be used to implement these functions, as was done in the previous release.
2. Numerical derivative and gradient calculations are now included, and are easily extended by inheritance.

3. Optimizers needing a derivative or gradient can now obtain a numerical calculation through wrapping if an analytic solution is not provided.

API compatibility with v1.0 has been broken. As a result, users of previous versions of OptSolve++ may have to change the inheritance of their functors to match the new class tree. Also, existing functors may require one or two extra methods (or renaming of methods) to match what is required by the interface classes.

New projects will benefit from the ease of use, extendibility, and ease of maintenance provided by v2.0's updated functor class hierarchy. Because the use of optimizers remains the same, projects that are currently using OptSolve++ v1.0 may not experience any significant gains by migrating to v2.0.

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Optimization with OptSolve++ . . . . .	1
1.2	Philosophy of Release . . . . .	1
<b>2</b>	<b>Overview</b>	<b>5</b>
<b>3</b>	<b>Installation</b>	<b>6</b>
3.1	Installation Instructions . . . . .	6
3.2	Linking Against OptSolve++ . . . . .	6
<b>4</b>	<b>How to Choose an Optimizer</b>	<b>7</b>
<b>5</b>	<b>Usage Examples</b>	<b>9</b>
5.1	A First Example . . . . .	9
5.2	The OptSolve++ Functor Hierarchy . . . . .	12
5.3	Using Numerical Derivatives . . . . .	17
5.4	More Example Optimizations . . . . .	18
5.5	Required Header Files for Calling OptSolve++ From <code>main()</code> . . . . .	21
5.6	Using the Levenberg-Marquardt Algorithm . . . . .	21
5.7	Exceptions in OptSolve++ . . . . .	22
5.8	Extracting the Results of the Optimization . . . . .	23
5.9	Using Levenberg-Marquardt with Broyden's Method . . . . .	23
5.10	The Nonlinear Simplex Algorithm and the Use of Boundary Constraints . . . . .	24
5.11	Nonlinear Simplex and Setting the Accuracy and Tolerance Criteria . . . . .	25
5.12	Conjugate Gradient Algorithm and Switching Optimizers in Midstream . . . . .	25
5.13	Powell's Algorithm and Invoking an Optimizer Step by Step . . . . .	26
5.14	Using the Parametric Model . . . . .	26
5.15	Output Examples . . . . .	29
<b>6</b>	<b>Class Reference</b>	<b>36</b>
6.1	The Top-Level Interface . . . . .	39
6.2	Methods Related to Convergence Criteria . . . . .	40
6.3	Methods Related to Setting Simple Bound Constraints . . . . .	41

**Listings**

1	TxFunCIfc.h — parent class of all functors . . . . .	9
2	Header file for functor base class . . . . .	10
3	Implementing test function by deriving from TxFunCIfc . . . . .	10
4	Instantiating and using an optimizer . . . . .	11
5	Header for functor with vector argument and gradient . . . . .	12
6	Implementation for functor with vector argument and gradient . . . . .	14
7	A functor in need of a numerical derivative. . . . .	17
8	Using a wrapper to obtain a numerical gradient. . . . .	17
9	Implementation of two Rosenbrock test functions and their gradients. . . . .	18
10	Defining a functor for Powell and non-linear simplex. . . . .	20
11	Declaration of a pointer to an optimizer. . . . .	21
12	Instantiating concrete functors. . . . .	22
13	Defining a starting point. . . . .	22
14	Using optimizer in a try block. . . . .	22
15	Using Broyden’s method in Levenberg-Marquardt. . . . .	23
16	Reusing optimizer pointer. . . . .	24
17	Setting bounds on optimizer. . . . .	24
18	Accuracy and tolerance. . . . .	25
19	Switching optimizers. . . . .	25
20	Invoking an optimizer step by step. . . . .	26
21	Defining a parametric model class. . . . .	27
22	Defining getPrediction(). . . . .	27
23	Implementation of the parametric model’s constructor. . . . .	27
24	Implementation of the parametric model’s Jacobian. . . . .	28
25	Optimizing using the parametric model. . . . .	28

## 2 Overview

OptSolve++ takes full advantage of templating techniques<sup>1</sup> and object-oriented design in order to provide users maximum flexibility in the choice of argument type and return type for the merit function, and in the configuration of options for the built-in algorithms. It is not necessary for the user to fully understand the structure of OptSolve++ in order to effectively use the software; however, OptSolve++ is readily extensible so that knowledgeable C++ programmers can implement alternative algorithms or create a thin interface to other C and C++ algorithms, while using the same convenient interface.

Four algorithms for the optimization of nonlinear multidimensional user-defined merit functions have been implemented in v2.0 of `TxOptSlv`. A merit function, for example, might be a fuel efficiency calculation in a combustion simulation for an automobile engine. In this example, the multidimensional argument might consist of the relative abundance of components in the fuel mixture, while the returned value might be the amount of unburned fuel and other waste products. In this case, the chosen optimization algorithm would vary the fuel mixture assumed in the simulation in order to minimize the emissions and maximize the fuel economy.

The four built-in multi-dimensional algorithms include:

- Powell<sup>2</sup>
- nonlinear simplex,<sup>3</sup>

which do not need the gradient of the function, as well as

- the conjugate gradient<sup>4</sup> algorithm, which does require access to the gradient, and the
- Levenberg-Marquardt<sup>5</sup> treatment of nonlinear least squares problems.

The various requirements for optimizers and their properties are summarized in table ??.

The Levenberg-Marquardt algorithm uses the gradient of the component functions, if provided; otherwise, it uses the Broyden method to estimate these gradients. Powell uses the Brent algorithm,<sup>6</sup> which does not need the function derivative. The conjugate gradient optimizer uses a modified secant algorithm,<sup>7</sup> which requires the derivative. Additionally, OptSolve++ can provide numerical derivatives for functions if needed. In this way, all optimizers may be used on a given function, even if the analytic derivative is not available. The optimization algorithms also allow users to impose simple bound constraints on the range of each component of the multidimensional argument.

---

<sup>1</sup>B. Stroustrup, *"The C++ Programming Language," Third Edition*, (Addison-Wesley 1997).

<sup>2</sup>M. J. D. Powell, *An Efficient Method for Finding the Minimum of a Function of Several Variables without Calculating Derivatives*, *Comput. J.*, **7**, 155 (1964).

<sup>3</sup>J. A. Nedler and R. Mead, *A Simplex Method for Function Minimization*, *Comput. J.*, **7**, 308 (1965).

<sup>4</sup>E. Polak, *"Computational Methods in Optimization,"* (Academic Press 1971).

<sup>5</sup>R. Fletcher, *"Practical Methods of Optimization,"* (John Wiley & Sons, 1980).

<sup>6</sup>R. P. Brent, *"Algorithms for Minimization without Derivatives,"* (Prentice Hall, 1973).

<sup>7</sup>W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, *"Numerical Recipes in C,"* (Cambridge University Press, 1992).

## 3 Installation

Tech-X has successfully built and run OptSolve++ with the following platform/compiler combinations. OptSolve++ will most likely build for other platforms, but Tech-X does not test or support them.

Table 1: Platforms and compilers supported by OptSolve++

Platform	compiler
Microsoft Windows 2000/XP	Microsoft Visual Studio.NET
Red Hat Linux 9	gcc 3.3
Mandrake 10	gcc 3.3
SuSE Linux 10	gcc 3.3
Mac OS X 10.3 (Panther)	gcc 3.3

### 3.1 Installation Instructions

OptSolve++ includes a graphical installation program that makes copying the libraries and necessary header files to the appropriate locations a simple process.

- To run the installer on Mac OS X or Windows, double-click on the installation icon.
- To run the installer in Linux, type

```
sh install.bin
```

at the command line.

OptSolve++ installs by default in `/usr/local/OptSolve++` on Unix-like platforms and in `My Documents\OptSolve++` in Windows. If you want to install it elsewhere, you can specify a different path during installation.

### 3.2 Linking Against OptSolve++

On Unix-like platforms, OptSolve++ ships with example source code and a simplistic `Makefile` in the `Examples/` subdirectory. The example `Makefile` builds executables that are statically linked with OptSolve++. Dynamic libraries are also provided in the installation and can be used.

Note that in `include/` directory there is a symbolic link called `config.h` that points to `config-gcc3.3.h`. OptSolve++ was build with GNU Autotools, and as such depends on the file `config.h`. If you are including OptSolve++ as part of a larger GNU Autotools project, you may wish to replace this default `config.h` file with your local configuration information.

For users with Microsoft Visual Studio.NET (MSVC.NET), OptSolve++ ships with static libraries for both debug and release versions. These files are located under the `static-lib/` folder. Additionally, there is an example MSVC.NET solution file in `demo/static build sample/` that builds the `optdemo.cxx` example. The other examples in `demo/` and can be built with `nmake` using the following command:

The `demo/` directory contains example files that can be built using the Microsoft `nmake` tool at the CMD prompt. (If the following commands do not work, ensure `nmake` is in your `PATH`.) To build the examples, use the following command in the `demo/` directory:

```
nmake -f Makefile.nmake
```

Generated files can be removed from this directory with the command

```
nmake clean -f Makefile.nmake
```

Given the wide variety of build systems in use, not all systems can be covered in this document. If you are having problems getting your source code to build or link properly with OptSolve++, please contact the user's group or open a support ticket. More information can be found at the following resources:

- <http://fusion.txcorp.com/mailman/listinfo/optsolve-users/>
- <http://www.txcorp.com/global/support.php>
- [support@txcorp.com](mailto:support@txcorp.com)

## 4 How to Choose an Optimizer

The choice of which optimizer to use depends on the type of function to be solved as well as the information present about that function and its derivatives. The four main considerations are:

1. Dimensionality of function
2. Whether an analytic derivative is available
3. Cost of evaluating the function and its derivative
4. Efficiency vs. robustness

The choice of which optimizer to use is important to success. Some guidelines are below, but remember; choosing an optimizer and how best to use it is as much art and experience as a codified decision-making process.

The nonlinear simplex algorithm can be very accurate for low-dimensional optimization problems. It also does not require a gradient calculation, so it can be used when that information is unavailable or is a high-cost function. Its major drawback, however, is that while it is quick to come "in range" of the minimum, it is often very slow to get to it with a high degree of accuracy. This slowness of convergence is a trade-off for not using function or gradient estimations and the imprecision they create.

The Powell optimizer is a good choice for problems where the slowness of nonlinear simplex is an issue but a gradient calculation is not available. It is not as robust as nonlinear simplex, but it converges to a solution more rapidly.

The conjugate gradient algorithm is very robust; however, as the name implies, it requires the gradient of the function. A numerical gradient can be used, of course, but as with all numerical approximations accuracy of the solution will suffer as a result. Powell or nonlinear simplex may be a better choice if this is the case.

The Levenberg-Marquardt is a special case, used mostly for nonlinear parametric optimizations. Other techniques, such as genetic algorithms, have been used, but Levenberg-Marquardt is the de facto standard for these types of problems. However, it also requires a Jacobian calculation, but this can be approximated using Broyden's method or finite differencing.

Figure 1 illustrates the different paths to the solution of the 2-D Rosenbrock functions taken by various optimizers. The x-axis represents the number of function evaluations. The y-axis is the current approximation of the solution. Each marker on a given curve represents one iteration of the optimizer, which may involve one or more function evaluations. The minimum of the function is at  $x = 0$  — therefore the lower the curve on the vertical axis, the closer an optimizer is to the solution.

Figures 2 and 3 show the Rosenbrock function near the minimum, along with the paths the various optimizers take to reach the minimum at (1, 1).

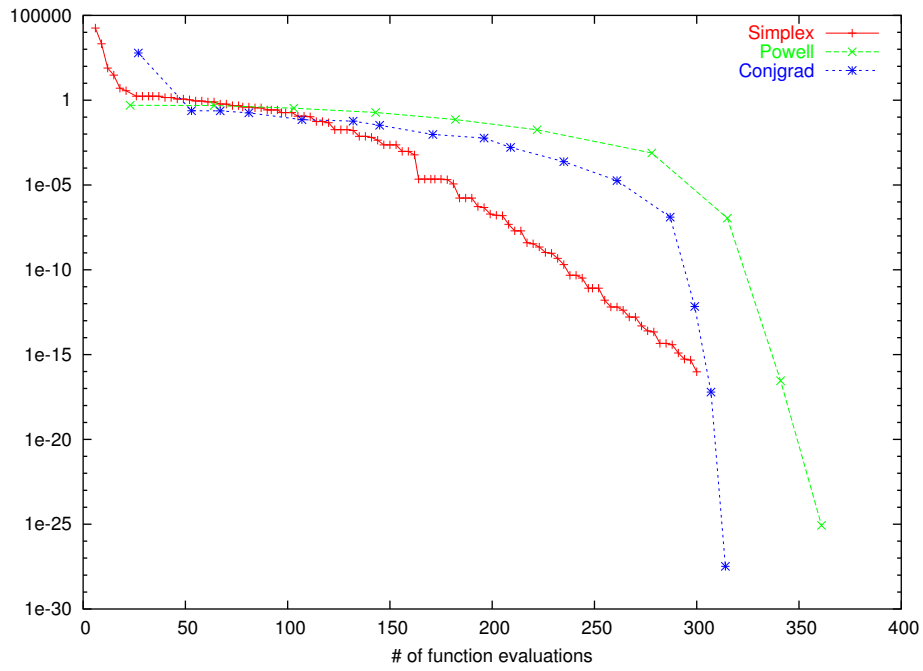


Figure 1: Graphical Comparison of Optimizers

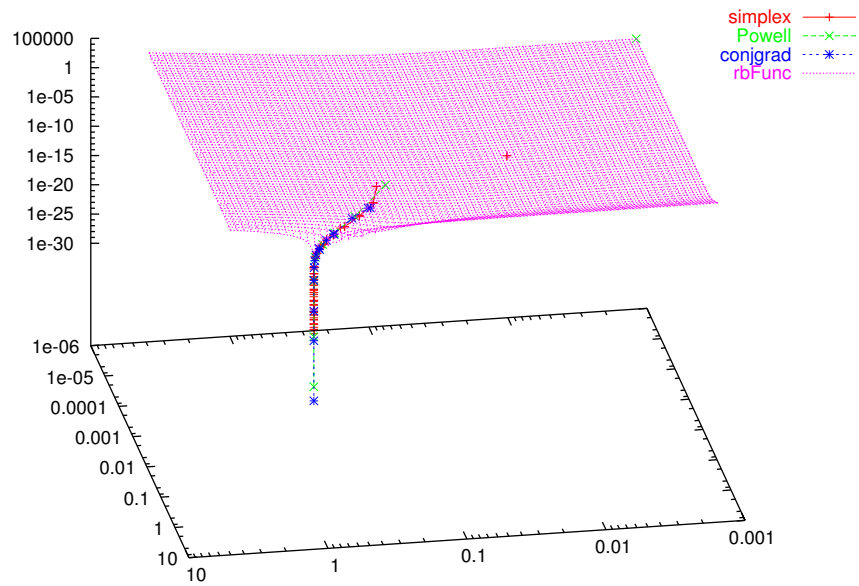


Figure 2: Isometric view of the Rosenbrock function and the optimization paths.

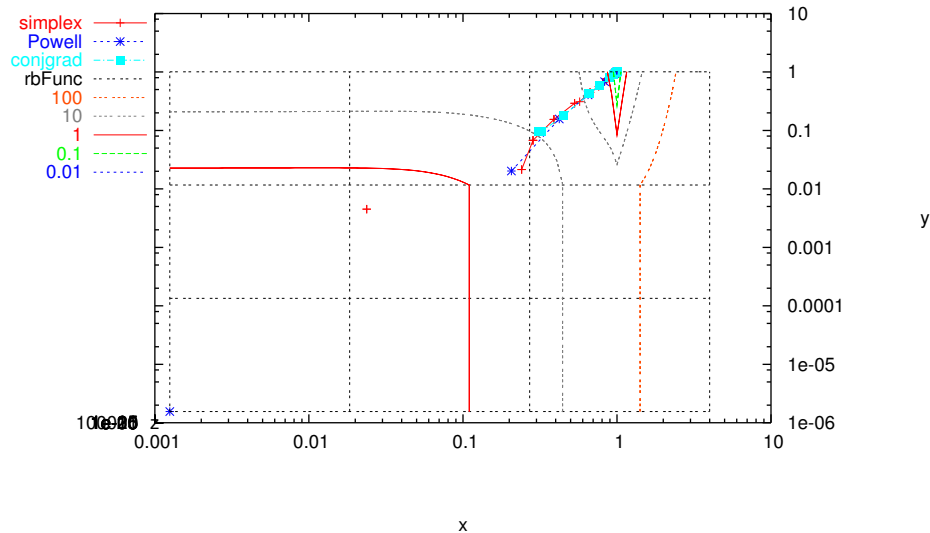


Figure 3: Contour plot of the Rosenbrock function and the optimization paths as seen looking toward  $-z$ .

## 5 Usage Examples

For application developers and system designers, the advantages of an object-oriented C++ optimization library include flexibility and ease of use. Developers of optimization algorithms will find advantages in ease of maintenance and extensibility. Flexibility is achieved through templating, which allows for a variety of numeric argument types and return types, and through polymorphism, which allows for multiple user-defined merit functions to be used interchangeably with multiple optimization algorithms. These goals are also achieved by providing a well-designed interface to the library.

We illustrate some of these features in the following examples. These examples are implemented in the source file `optdemo.cxx` and its output. This file can be found in the `demo` directory.

### 5.1 A First Example

It is tradition in computer science documentation to begin describing any new language with the “Hello, World!” example. Extending this tradition to include optimizer library user’s manuals, the OptSolve++ equivalent of “Hello, World!” will now be presented. This includes the minimum code needed to begin using the library, creation and use of functors and optimizers, and how to obtain results.

The function that will be minimized is  $f(x) = x^2 - 1$ , a simple parabola. Of course, we need to represent this parabola in a way that OptSolve++ can use. This is done by using a C++ object called a functor. This object is simply an object representation of a mathematical function. In OptSolve++, all functors are derived from the interface class `TxFunCIfc`. This class, as with all interfaces, has no implementation — it merely outlines what methods a class needs to be considered a functor.

Listing 1: `TxFunCIfc.h` — parent class of all functors

```
//-----
//
// File:    TxFunCIfc.h
```

```

//
// Purpose: New abstract base class for all functors.
//
// Version: $Id: TxFuncIfc.h,v 1.1 2004/09/07 22:50:14 cary Exp $
//
// Copyright (c) 2002-2003 by Tech-X Corporation. All rights reserved.
//
//-----
#ifndef TX_FUNC_IFC_H
#define TX_FUNC_IFC_H

// txbase includes
#include <TxContainers.h>
#include <TxBadEvalExcept.h>

/** New abstract base class for all functors.
 *
 * @author Johan Carlsson & Nate Sizemore
 */

template <typename RetType, typename ArgType>
class TxFuncIfc {

    public:

    /// Destructor does nothing.
    virtual ~TxFuncIfc() {}

    /// Operator() always returns function value, scalar OR vector.
    virtual RetType operator()(const ArgType& x) const
        throw (TxBadEvalExcept) = 0;

};

#endif // TX_FUNC_IFC_H

```

As you can see, the interface class is extremely light. Only one method has to be defined — `operator()`. So implementing our test function is quite simple.

Listing 2: Header file for functor base class

```

// example functor

// txfunc includes
#include <TxFuncIfc.h>

template <class RetType, class ArgType>
class ExampleFunctor : public TxFuncIfc<RetType, ArgType>
{
    public:
    RetType operator()(const ArgType& x) const
        throw (TxBadEvalExcept);
};

```

Listing 3: Implementing test function by deriving from TxFuncIfc

```

// local includes
#include "exampleFunctor.h"

```

```

template <typename RetType, typename ArgType>
RetType ExampleFunctor<RetType, ArgType>::operator()(const ArgType& x) const
    throw (TxBadEvalExcept) {
    return x * x - 1.0;
}

// ansi instantiation
template class ExampleFunctor<double, double>;

```

With the functor defined, we are halfway to our goal. Now we need to give our functor to an optimizer. For this example, we'll use the Brent optimizer, since it does not use a derivative calculation.

Listing 4: Instantiating and using an optimizer

```

1 // example executable code
2
3 // base includes
4 #include <iostream>
5
6 // txfunc includes
7 #include <TxFuncIfc.h>
8
9 // txoptslv includes
10 #include <TxBrentOpt.h>
11
12 // local includes
13 #include <exampleFunctor.h>
14
15 int main(int argc, char** argv) {
16
17     // create functor object
18     ExampleFunctor<double, double>* myFunctor =
19         new ExampleFunctor<double, double>();
20
21     // create optimizer object, giving it the functor to use
22     TxBrentOpt<double, double> myOptimizer(myFunctor);
23
24     // tell the optimizer to solve with starting point of 10
25     myOptimizer.solve(10);
26
27     // extract results and print them
28     double minValue = myOptimizer.getMinValue();
29     double minPoint = myOptimizer.getMinPoint();
30
31     TXSTRSTD::cout
32         << "\nFor the function y = x^2 - 1, the brent optimizer found the\n"
33         << "minimum to be at x = " << minPoint << " with a value of y = "
34         << minValue << "\n" << TXSTRSTD::endl;
35
36     return 0;
37 }

```

Let's take apart this source code a piece at a time and look at what is happening at each point.

1. Lines 3-13 are the necessary header file includes. We use the C++ stream libraries for output, so `iostream.h` is included. For our functor, we need both the interface definition in `TxFuncIfc.h` and the local header

file we wrote earlier, `exampleFunctor.h`. Finally, we need the Brent optimizer; its interface is defined in `TxBrentOpt.h`.

2. On line 18 we create a pointer to our functor, and then instantiate and assign an object to that pointer. Notice the use of template parameters here.
3. Line 21 creates a Brent optimizer object called `myOptimizer`. It is initialized with the pointer to our functor — so it knows how to access the function it will be optimizing.
4. The real work of the entire program is done on line 24, with a call to the optimizer's `solve` method. The parameter passed in to the `solve` method is the initial guess, the starting point for the calculations.
5. Lines 27 and 28 extract the results from the optimizer with a couple of accessor methods. In this example, we only look at the minimum point the optimizer found, and the value of the function at that point. The optimizer has much more information if we want it, however, including accuracy, tolerance, and number of iterations required. See Section 1.

These are the basic elements needed to make use of `OptSolve++` to minimize merit functions. Of course, the library is capable of much more than simply finding the minimum of a parabola — it can use different optimization algorithms, handle multi-dimensional equations, use analytic or numerical derivatives, and use bounds to constrain a problem to a given domain. All of these features will be described in the following sections.

## 5.2 The `OptSolve++` Functor Hierarchy

The previous version of `OptSolve++` used function pointers heavily. The current version, however, has been redesigned and uses what are called interface classes, as mentioned in Section 5.1. These interface classes are related in a tree structure that makes the properties of functors both easy to grasp and easy to expand.

All functors are derived from the interface class `TxFunCIfc`. This interface defines only a single method that must be implemented — `operator()`. Derivatives, gradients, and Hessian methods are defined at the next layer of the functor tree, specifically the child classes `TxDerivFuncIfc`, `TxGradFuncIfc`, and `TxJacFuncIfc`. Optimizers that require a derivative calculation will specify one of these interfaces in their constructors. The functors use it by implementing one of the `getDeriv()`, `getGrad()`, or `getJac()` methods defined in the interface. However, analytic derivatives are not needed in order to use these optimizers. If an analytical derivative is difficult (or impossible) to find or not worth the effort of solving, you can still use any of the optimizers included in `OptSolve++`.

As an example, let's implement a more complex function, the Helical Valley function that is often used to test optimization algorithms. This is a three-three-dimensional function. We will also show what is needed to create a functor with an analytical gradient, and how to use a numerical gradient instead.

Listing 5: Header for functor with vector argument and gradient

```
//-----
//
// File:      newHelicalValley.h
//
// Purpose:  Implement the 3-D vector-valued Helical Valley function,
//           documented on p. 21 of More', Garbow & Hillstom, "Testing
//           Unconstrained Optimization Software," ACM Trans. Math. Soft.,
//           Vol. 7 (1981), pp. 17-41.
//
// Version:  $Id: newHelicalValley.h,v 1.1 2004/09/07 22:50:14 cary Exp $
//
// Copyright (c) 2003 by Tech-X Corporation. All rights reserved.
//
//-----
// std includes
```

```

#include <vector>

// From configure
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

// txbase includes
#include <txc_std.h>
#include <txc_streams.h>
#include <TxBadEvalExcept.h>

// txfunc includes
#include <TxGradFuncIfc.h>

class NewHelicalValley : public virtual TxGradFuncIfc<double, TXSTD::vector<double> > {
public:
    size_t getArgDim() const;

    double operator()(const TXSTD::vector<double>& x) const
        throw (TxBadEvalExcept);

    TXSTD::vector<double> getGrad(const TXSTD::vector<double>& x) const
        throw (TxBadEvalExcept);

    /// Method returning standard starting point for Helical Valley functions.
    TXSTD::vector<double> getHelicalValleyPoint();

private:
    /// First Helical Valley test function.
    double f0helicalValley(const TXSTD::vector<double>&) const;

    /// Gradient of the first Helical Valley test function.
    TXSTD::vector<double> g0helicalValley(const TXSTD::vector<double>&) const
        throw (TxBadEvalExcept);

    /// Second Helical Valley test function.
    double f1helicalValley(const TXSTD::vector<double>&) const;

    /// Gradient of the second Helical Valley test function.
    TXSTD::vector<double> g1helicalValley(const TXSTD::vector<double>&) const
        throw (TxBadEvalExcept);

    /// Third Helical Valley test function.
    double f2helicalValley(const TXSTD::vector<double>&) const;

    /// Gradient of the third Helical Valley test function.
    TXSTD::vector<double> g2helicalValley(const TXSTD::vector<double>&) const;
};

```

Notice how this functor derives not from `TxFuncIfc`, but from `TxGradFuncIfc`. This is the interface that defines a functor with a gradient. Also, since this functor *has* a gradient, it also implies that the function's argument is a vector. This is, in fact, the case — `TxGradFuncIfc` itself derives from `TxVecFuncIfc`, which defines a functor with scalar return and a vector argument.

The net result of inheriting from TxGradFuncIfc is that we need to define two more methods — getArgDim() and getGrad(). Of course, we still define operator() as before. Below are the implementations of those functions.

Listing 6: Implementation for functor with vector argument and gradient

```
//-----
//
// File:      newHelicalValley.cpp
//
// Purpose: Implement the 3-D vector-valued Helical Valley function,
//          documented on p. 21 of More', Garbow & Hillstrom, "Testing
//          Unconstrained Optimization Software," ACM Trans. Math. Soft.,
//          Vol. 7 (1981), pp. 17-41.
//
// Version: $Id: newHelicalValley.cpp,v 1.1 2004/09/07 22:50:14 cary Exp $
//
// Copyright (c) 2003 by Tech-X Corporation. All rights reserved.
//-----

// txbase includes
#include <txc_math.h>
#include <TxContainers.h>

// demo includes
#include "newHelicalValley.h"

// Method that returns the dimension of the space.
size_t NewHelicalValley::getArgDim() const {
    return 3;
}

// Sum of the squares of the three function values.
double NewHelicalValley::
operator()(const TXSTD::vector<double>& x) const
    throw (TxBadEvalExcept) {

    double f0 = f0helicalValley(x);
    double f1 = f1helicalValley(x);
    double f2 = f2helicalValley(x);

    return 0.5 * (f0 * f0 + f1 * f1 + f2 * f2);
}

// Gradient of the sum-of-squares.
TXSTD::vector<double> NewHelicalValley::
getGrad(const TXSTD::vector<double>& x) const
    throw (TxBadEvalExcept) {

    double f0 = f0helicalValley(x);
    double f1 = f1helicalValley(x);
    double f2 = f2helicalValley(x);

    TXSTD::vector<double> f0grad(3), f1grad(3), f2grad(3);
    f0grad = g0helicalValley(x);
    f1grad = g1helicalValley(x);
    f2grad = g2helicalValley(x);

    TXSTD::vector<double> g(3);
    g[0] = f0 * f0grad[0] + f1 * f1grad[0] + f2 * f2grad[0];
}
```

```

g[1] = f0 * f0grad[1] + f1 * f1grad[1] + f2 * f2grad[1];
g[2] = f0 * f0grad[2] + f1 * f1grad[2] + f2 * f2grad[2];

return g;
}

// ***** The following are functions which are pieces of the whole

// First Helical Valley test function.
double NewHelicalValley::
f0helicalValley(const TXSTD::vector<double>& x) const {

    double theta = 0.0;
    if (x[0] > 0.0) theta = 0.5 * atan2(x[1], x[0]) / M_PI;
    if (x[0] < 0.0) theta = 0.5 * atan2(x[1], x[0]) / M_PI + 0.5;
    if (x[0] == 0.0) theta = 0.25;

    return 10.0 * (x[2] - 10.0 * theta);
}

// Gradient of the first Helical Valley test function.
TXSTD::vector<double> NewHelicalValley::
g0helicalValley(const TXSTD::vector<double>& x) const
    throw(TxBadEvalExcept) {

    double x0sq = x[0] * x[0];
    double x1sq = x[1] * x[1];
    double xSumSq = x0sq + x1sq;

    if (xSumSq < 1.2 * TxUnaryNumberTraits<double>::MIN)
        throw TxBadEvalExcept("xSumSq = 0 in flhelicalValley()!");

    TXSTD::vector<double> g(3);
    g[0] = 50.0 * x[1] / (M_PI * xSumSq);
    g[1] = -50.0 * x[0] / (M_PI * xSumSq);
    g[2] = 10.0;

    return g;
}

// Second Helical Valley test function.
double NewHelicalValley::
flhelicalValley(const TXSTD::vector<double>& x) const {

    return 10.0 * (sqrt(x[0]*x[0] + x[1]*x[1]) - 1.0);
}

// Gradient of the second Helical Valley test function.
TXSTD::vector<double> NewHelicalValley::
glhelicalValley(const TXSTD::vector<double>& x) const
    throw (TxBadEvalExcept) {

    double x0sq = x[0] * x[0];
    double x1sq = x[1] * x[1];
    double xSumSq = x0sq + x1sq;

    if (xSumSq < 1.2 * TxUnaryNumberTraits<double>::MIN)
        throw TxBadEvalExcept("xSumSq=0 in flhelicalValley()!");

    double xRoot = 1.0 / sqrt(xSumSq);

```

```

    TXSTD::vector<double> g(3);
    g[0] = 10.0 * x[0] * xRoot;
    g[1] = 10.0 * x[1] * xRoot;
    g[2] = 0.0;

    return g;
}

// Third Helical Valley test function.
double NewHelicalValley::
f2helicalValley(const TXSTD::vector<double>& x) const {
    return x[2];
}

// Gradient of the third Helical Valley test function.
TXSTD::vector<double> NewHelicalValley::
g2helicalValley(const TXSTD::vector<double>& x) const {

    TXSTD::vector<double> g(3);
    g[0] = 0.0;
    g[1] = 0.0;
    g[2] = 1.0;

    return g;
}

// Method returning standard starting point for the Helical Valley functions.
TXSTD::vector<double> NewHelicalValley::
getHelicalValleyPoint() {

    TXSTD::vector<double> point(3);
    point[0] = 1.0;
    point[1] = 0.0;
    point[2] = 0.0;

    return point;
}

```

As you can see, in this implementation we implement three functions — `f0helicalValley`, `f1helicalValley`, and `f2helicalValley`, along with their gradients. These are used to calculate the return values for `operator()` and `getGrad()`. The method `getArgDim()` returns the value 3, since the dimensionality of the argument of the function will never change.

A variety of functions can be represented easily by inheriting from the proper parent class in `OptSolve++`. Choosing among the available functor classes is a matter of matching the functor with

- The dimensionality of the return value of your function
- The dimension of the argument to your function
- Whether you have an analytical derivative, gradient, or Jacobian for the function
- Whether you desire a numerical derivative, gradient, or Jacobian for the function
- Whether the function will be implemented through inheritance or function pointers
- Any unique properties about the function

The ability to define functors through the use of function pointers has been retained in the new version of OptSolve++; however, this may make the code less extensible than if derived classes were used.

Table 5.2 lists some of the common interface classes and their properties.

Table 2: Common interface classes

Interface	Properties
TxFuncIfc	Base class of all functor interfaces
TxVecArgFuncIfc	Defines a functor with vector argument and vector return value
TxVecFuncIfc	Defines a functor with a vector argument and scalar return
TxDerivFuncIfc	Scalar functor which includes a derivative calculation
TxGradFuncIfc	Functor with vector argument that includes a gradient
TxJacFuncIfc	Functor with vector argument and vector return that can calculate a Jacobian
TxHessFuncIfc	Functor with vector argument and vector return that can calculate a Hessian
TxBoundsIfc	Interface that defines bounds for optimizer
TxLevMarFuncIfc	Interface with methods specific to the Levenberg-Marquardt optimizer

### 5.3 Using Numerical Derivatives

Wrapper classes are used to provide a functor with a numerical derivative. Wrapping is simply a way to extend the functionality of a class or change its interface by creating one class from another.

For example, suppose we have the following functor:

Listing 7: A functor in need of a numerical derivative.

```
// txfunc includes
#include <TxGradFuncIfc.h>

template <class RetType, class ArgVecType>
class TestFunc : public TxVecFuncIfc<RetType, ArgVecType>
{
public:
    RetType operator()(const ArgVecType& x) const
        throw (TxBadEvalExcept) {
        return 0.5 * (x[0] * x[0] + x[1] * x[1]);

    size_t getArgDim() const {
        return 2;
    }
};
```

The conjugate gradient optimizer requires a gradient, and so we cannot use this functor with it unless we satisfy that requirement. We can do that by inheriting from `TxGradFuncIfc` and implementing an analytic gradient, or we can wrap the function and use the numerical gradient provided by the wrapper, as shown:

Listing 8: Using a wrapper to obtain a numerical gradient.

```
1 TestFunc<double, TXSTD::vector<double> > func;
2 TXSTD::vector<double> vecArg(2, 2.0);
3
4 tts << "Test: Scalar functor w/ vector parameters - function" << TXSTRSTD::endl;
5 tts << "Function value is\n"
6   << "Result: " << func(vecArg) << TXSTRSTD::endl;
7 tts << TXSTRSTD::endl;
8
```

```

9   TxNumGradFunc<double, TXSTD::vector<double> >* vecGradient;
10
11   vecGradient = new TxNumGradFunc<double, TXSTD::vector<double> >(func);
12
13   gradientValue = vecGradient->getGrad(vecArg);
14
15   tts << "Test: Scalar functor w/ vector parameters - gradient" << TXSTRSTD::endl;
16   tts << "Numerical gradient value is\n"
17       << "Result: " << gradientValue[0] << "\t" << gradientValue[1] << TXSTRSTD::endl;
18   tts << TXSTRSTD::endl;

```

On line 9 we create a pointer to a `TxNumGradFunc` object — the wrapper that provides a numerical gradient for functions with scalar return values and a vector parameter. We then instantiate the wrapper by passing it our functor on line 11. The wrapper can now be used in the same manner as the functor. All calls to methods in the wrapper are passed to the original functor, with the exception of `getGrad()` which is defined by the wrapper itself. With this method defined by the wrapper, we can now use this object in optimizers requiring a gradient calculation.

See the file `demo/dmfunc/deriv.cxx` for a longer demo using wrappers.

The following table summarizes the wrapper classes provided in `OptSolve++` and the functionality they add. Note the special case of `TxNumLevMarFunc.h` — functors that are to be used with Levenberg-Marquardt should use this wrapper, if needed.

Table 3: Wrapper files providing numerical derivatives in `OptSolve++`

Name	Functionality
<code>TxNumDerivFunc.h</code>	Provides the method <code>getDeriv()</code> that calculates a derivative using finite differences
<code>TxNumGradFunc.h</code>	Provides the method <code>getGrad()</code> that calculates a gradient using finite differences
<code>TxNumJacFunc.h</code>	Provides the method <code>getJac()</code> that calculates a Jacobian using finite differences
<code>TxBroydGradFunc.h</code>	Provides the method <code>getDeriv()</code> that calculates a gradient using Broyden's method
<code>TxBroydJacFunc.h</code>	Provides the method <code>getGrad()</code> that calculates a Jacobian using Broyden's method
<code>TxNumLevMarFunc.h</code>	Finite differencing <code>getJac()</code> for use in the Levenberg-Marquardt optimizer
<code>TxBroydLevMarFunc.h</code>	Broyden's method <code>getJac()</code> for use in the Levenberg-Marquardt optimizer

## 5.4 More Example Optimizations

In the previous section we briefly described functors. Here, we go into more detail by implementing the 2-D Rosenbrock optimizer test functions.<sup>8</sup> We will also introduce the other optimizers that are available in `OptSolve++`.

To demonstrate a solution of the nonlinear least squares problem using the Levenberg-Marquardt algorithm, we define two Rosenbrock functions and their gradients in a class derived from `TxJacFuncIfc`. The implementation is shown here.

Listing 9: Implementation of two Rosenbrock test functions and their gradients.

```

template <typename RetVecType, typename ArgVecType>
class RosenbrockVector : public TxJacFuncIfc<RetVecType, ArgVecType> {

    // operator() returns a vector made up of the Rosenbrock function
    // return values
    TXSTD::vector<double> operator()(const TXSTD::vector<double>& x) const
        throw (TxBadEvalExcept) {

        TXSTD::vector<double> f(2);
        f[0] = f0rb(x);

```

<sup>8</sup>J. J. Moré, B. S. Garbow and Hillstom, "Testing Unconstrained Optimization Software," ACM Trans. Math. Soft., Vol. 7 (1981), pp. 17-41.

```

    f[1] = flrb(x);

    return f;
}

// getJac returns the Jacobian matrix of the functor, created from
// the gradients of the Rosenbrock functions
TxMatrix<double> getJac(const TXSTD::vector<double>& x) const
    throw (TxBadEvalExcept) {

    TxSquareMatrix<double> g(2);

    TXSTD::vector<double> temp = g0rb(x);
    for (size_t j=0; j<2; j++) g(0, j) = temp[j];

    temp = glrb(x);
    for (size_t j=0; j<2; j++) g(1, j) = temp[j];

    return g;
}

// specify dimension of the argument vector -- if this is wrong,
// program behavior will be unpredictable.
size_t getArgDim() const {
    return 2;
}

// specify dimension of the returned square matrix -- if this is
// wrong, program behavior will be unpredictable.
size_t getRetDim() const {
    return 2;
}

// First of the Rosenbrock test functions.
double f0rb(const TXSTD::vector<double>& x) const {
    return 10.0 * ( x[1] - x[0] * x[0] );
}

// Second of the Rosenbrock test functions.
double flrb(const TXSTD::vector<double>& x) const {
    return 1.0 - x[0];
}

// Gradient of the first Rosenbrock test function.
TXSTD::vector<double> g0rb(const TXSTD::vector<double>& x) const {
    TXSTD::vector<double> g(2);
    g[0] = -20.0 * x[0];
    g[1] = 10.0;

    return g;
}

// Gradient of the second Rosenbrock test function.
TXSTD::vector<double> glrb(const TXSTD::vector<double>& x) const {
    TXSTD::vector<double> g(2);
    g[0] = -1.0;
    g[1] = 0.0;

    return g;
}

```

```

    }
};

```

The container type for the multidimensional argument is also a template parameter in OptSolve++, but at present only `std::vector` is supported. This limitation will vanish if full source code is provided to users or if compiler vendors begin to support the ANSI standard export keyword.

Powell's algorithm and the nonlinear simplex algorithm only need a single C++ function that defines the sum of the squares of the two Rosenbrock test function values. The conjugate gradient algorithm also requires implementation of the gradient of the sum of squares. The implementation of this functor is shown here.

Listing 10: Defining a functor for Powell and non-linear simplex.

```

template <typename RetType, typename VecType>
class RosenbrockSumOfSq : public TxGradFuncIfc<RetType, VecType> {

    // Powell's algorithm and the nonlinear simplex algorithm only need
    // a single C++ function that defines the sum of the squares of the
    // two Rosenbrock test function values. This is the result defined
    // by this functor's operator().

    // Sum of the squares of the two function values.
    double operator()(const TXSTD::vector<double>& x) const
        throw (TxBadEvalExcept) {
        double f0 = f0rb(x);
        double f1 = f1rb(x);

        return 0.5 * ( f0 * f0 + f1 * f1 );
    }

    // The conjugate gradient algorithm also needs the gradient of the
    // sum of squares to be defined.
    VecType getGrad(const TXSTD::vector<double>& x) const
        throw (TxBadEvalExcept) {

        TXSTD::vector<double> g(2);

        double f0 = f0rb(x);
        double f1 = f1rb(x);

        TXSTD::vector<double> f0grad(2), f1grad(2);
        f0grad = g0rb(x);
        f1grad = g1rb(x);

        g[0] = f0*f0grad[0] + f1*f1grad[0];
        g[1] = f0*f0grad[1] + f1*f1grad[1];

        return g;
    }

    // specify dimension of the argument vector -- if this is wrong,
    // program behavior will be unpredictable.
    size_t getArgDim() const {
        return 2;
    }

    // First of the Rosenbrock test functions.
    double f0rb(const TXSTD::vector<double>& x) const {
        return 10.0 * ( x[1] - x[0] * x[0] );
    }

```

```

}

// Second of the Rosenbrock test functions.
double flrb(const TXSTD::vector<double>& x) const {
    return 1.0 - x[0];
}

// Gradient of the first Rosenbrock test function.
TXSTD::vector<double> g0rb(const TXSTD::vector<double>& x) const {
    TXSTD::vector<double> g(2);
    g[0] = -20.0 * x[0];
    g[1] = 10.0;

    return g;
}

// Gradient of the second Rosenbrock test function.
TXSTD::vector<double> glrb(const TXSTD::vector<double>& x) const {
    TXSTD::vector<double> g(2);
    g[0] = -1.0;
    g[1] = 0.0;

    return g;
}
};

```

## 5.5 Required Header Files for Calling OptSolve++ From main()

The header files required for programs which use the OptSolve++ library are shown below.

Table 4: Required header files for OptSolve++

Location of include	Name	Purpose
TxBASE includes	txc_streams.h	cross-platform header for I/O streams
TxOptSlv includes	TxLevMarqOpt.h	Levenberg-Marquardt algorithm
	TxConjugateGradOpt.h	conjugate gradient algorithm
	TxNonLinSimplexOpt.h	nonlinear simplex algorithm
	TxPowellOpt.h	Powell's algorithm

You will also need the header file for each functor interface you wish to use. For example, if you have a functor created from TxGradFuncIfc, you will need to include TxGradFuncIfc.h.

## 5.6 Using the Levenberg-Marquardt Algorithm

This section describes how to apply the Levenberg-Marquardt optimization algorithm to a nonlinear least squares problem. This and subsequent examples will also show how this optimizer pointer can be used polymorphically to represent any of the available optimization algorithms. This is useful in prototyping code to determine which optimizer may work best for a particular merit function. Using an optimizer polymorphically also allows them to be combined, as described in section 5.12.

To use optimizers polymorphically, we first define a TxOptimizer pointer that will be used for all of the optimizers:

Listing 11: Declaration of a pointer to an optimizer.

```
// Define the TxOptimizer pointer that will be used. This optimizer
// pointer can be used polymorphically, because it has a common interface,
// no matter what algorithmic implementation it is instructed to use.
TxOptimizer<double, TXSTD::vector<double> >* optimizer;
```

Optimizers specify their requirements for dimensionality and derivative by the functors that can be used in their construction. The signature of the optimizer must correspond to the signature of the implemented merit function — in other words, the template parameters should be the same. The first template parameter in the above code listing, **double**, is the return type of the merit function, while the second template parameter, `std::vector<double>`, is the container type and (templated) numeric type of the multidimensional argument of the functor to be used.

In order to use the optimizer, however, a concrete functor must also be instantiated, as seen below.

Listing 12: Instantiating concrete functors.

```
RosenbrockSumOfSq<double, TXSTD::vector<double> > rbFunc;
RosenbrockVector<TXSTD::vector<double>, TXSTD::vector<double> > rbVecFunc;
```

The Tech-X functor classes are used to ensure a common interface for functions and optimizers. These can be easily extended to match the exact requirements of the merit function calculation, while keeping the use of optimizers consistent and easy.

The order and meaning of the template parameters is similar for functors and optimizers. The first template parameter defines the numeric return type of the multiple merit functions and the type of container that will be used to hold the array of return values from these functions. The second template parameter specifies the container type and numeric type of the multidimensional argument.

We also need to define a vector and load it with an initial guess as to the location of the minimum. The sum of the squares of the Rosenbrock functions has a minimum value of 0, located at (1, 1).

Listing 13: Defining a starting point.

```
// Define the starting point for the optimizers (i.e. the initial guess)
startPoint[0] = 4.;
startPoint[1] = -4.;
```

## 5.7 Exceptions in OptSolve++

The OptSolve++ libraries use exception handling, so that they will not kill the calling application with unwanted core dumps. Sometimes a thrown exception indicates a fairly benign problem, such as failure to converge in the specified number of iterations. Thus, OptSolve++ methods should be invoked within a C++ **try** block, as shown.

Listing 14: Using optimizer in a try block.

```
try {
    optimizer -> solve(startPoint);
}
catch (TxOptSlvExcept& e) {
    // In the event that an exception is thrown, we catch it and write out some
    // error diagnostics to errFile (using the through stream object tts).
    tts.setFilter( TxThroughStream::TX_ERRORS );
    tts << "-----" << TXSTRSTD::endl;
    tts << "-----" << TXSTRSTD::endl;
    tts << "Exception caught in " << __FILE__ << ", at line " << __LINE__ << TXSTRSTD::endl;
    tts << "Here's what the exception has to say for itself:" << TXSTRSTD::endl;
    tts << e.getMessage() << TXSTRSTD::endl << TXSTRSTD::endl;
}
```

The exception class `TxOptSolveExcept` is thrown by the `OptSolve++` optimization algorithms if something unexpected occurs, and must be caught by the calling program.

## 5.8 Extracting the Results of the Optimization

Extracting data from the optimizer is quite simple — all values are accessible with standard accessor methods. Below is a table of the values that can be obtained from an optimizer and the name of the accessor that returns the data. Note that data vectors (such as the starting point or minimum point in multi-dimensional functions) have accessors that return values element-by-element, not the entire vector. These accessors take an argument stating which component  $n$  should be returned.

Table 5: Accessing data from optimizers

Value	assessor
number of iterations used by optimizer	<code>getNumIter()</code>
minimum function value found by optimizer	<code>getMinValue()</code>
achieved accuracy	<code>getAchAccuracy()</code>
achieved gradient tolerance	<code>getAchGradTolerance()</code>
achieved tolerance	<code>getAchTolerance(<math>n</math>)</code>
minimum point found by optimizer	<code>getMinPoint(<math>n</math>)</code>
specified accuracy	<code>getTotAccuracy()</code>
specified gradient tolerance (normalized)	<code>getAbsGradTolerance()</code>
specified tolerance	<code>getTotTolerance(<math>n</math>)</code>
starting point given to optimizer	<code>getStartPoint(<math>n</math>)</code>

The resulting output for the Levenberg-Marquardt optimization example appears in Table 5.15. The correct minimum value was found at the correct location. During 24 iterations of the optimization algorithm, the functions were evaluated 51 times and the gradients were evaluated 52 times. The maximum normalized gradient criterion was satisfied. The specified accuracy criterion (change in function value from one iteration to the next) was not satisfied, but the specified tolerance criteria (change in  $x[0]$  and  $x[1]$  from one iteration to the next) was satisfied.

## 5.9 Using Levenberg-Marquardt with Broyden's Method

Often, merit functions that one would wish to optimize have gradients that are very costly to evaluate; or they may be noisy or not available at all. For situations such as these, the Levenberg-Marquardt algorithm can still be used. By using Broyden's method to estimate the Jacobian (i.e. the array of gradients), we can overcome these limitations.

Because Broyden's method is implemented in only the Levenberg-Marquardt optimizer and not the more general `TxOptimizer` class, we must downcast the pointer from `TxOptimizer` to `TxLevMarqOpt`, so that the specialized `setBroydGrad()` method can be invoked.

Listing 15: Using Broyden's method in Levenberg-Marquardt.

```
dynamic_cast< TxLevMarqOpt<double, TXSTD::vector<double> >*>
    (optimizer)->setBroydGrad();

// Now we reset the internal data of the optimizer
optimizer -> reset();

// Send some pretty diagnostics to outFile.
tts.setFilter( TxThroughStream::TX_OUTPUT );
tts << TXSTRSTD::endl;
tts << "*****" << TXSTRSTD::endl;
tts << "*****" << TXSTRSTD::endl;
tts << "Levenburg-Marquardt with Broyden's estimate of Jacobian --" << TXSTRSTD::endl;
```

```
// Send status message to the screen.
tts.setFilter( TxThroughStream::TX_MESG );
tts << TXSTRSTD::endl;
tts << "Optimizing again with Levenburg-Marquardt, but this          " << TXSTRSTD::endl;
tts << "  time using Broyden's method to estimate the Jacobian..." << TXSTRSTD::endl;

try {
    optimizer -> solve(); // previously specified startPoint is used again
}
catch (TxOptSlvExcept& e) {
    // In the event that an exception is thrown, we catch it and write out some
    // error diagnostics to errFile (using the through stream object tts).
    tts.setFilter( TxThroughStream::TX_ERRORS );
    tts << "-----" << TXSTRSTD::endl;
    tts << "-----" << TXSTRSTD::endl;
    tts << "Exception caught in " << __FILE__ << ", at line " << __LINE__ << TXSTRSTD::endl;
    tts << "Here's what the exception has to say for itself:" << TXSTRSTD::endl;
    tts << e.getMessage() << TXSTRSTD::endl << TXSTRSTD::endl;
}
}
```

The resulting output is shown in Table 5.15. The correct minimum value was found at the correct location. During 84 iterations of the optimization algorithm, the functions were evaluated 175 times — but the gradients were evaluated (via finite difference) just once. This occurs after either the accuracy or tolerance convergence criteria is satisfied, in order to check the small gradient criteria in order to make sure the algorithm has not stopped at a false minimum. As before, the accuracy criterion was not satisfied, but the tolerance criteria were met.

Another example of using Levenberg-Marquardt with Broyden's method can be found in the `demo/dmoptslv/levMarBroyd.cxx` demo.

## 5.10 The Nonlinear Simplex Algorithm and the Use of Boundary Constraints

The nonlinear simplex algorithm does not require any information about the gradient. Its only requires is a functor implementing `operator()` with a vector argument and a scalar return. The `RosenbrockSumOfSq` class (and its instantiation `rbFunc`) shown earlier by using the parenthesis operator of the top-level `TxFunCIfc` interface to return the sum of squares of the function values. It also defines a gradient with `getGrad()`, but this information is not used by the optimizer. (You can see this in the output — the gradient is never evaluated.) It is a simple matter to change our optimizer and use this new functor:

Listing 16: Reusing optimizer pointer.

```
delete optimizer;
optimizer = new TxNonLinSimplexOpt<double, TXSTD::vector<double> >(rbFunc);
```

We also set simple boundary constraints on the argument of the function. Boundary capability is implemented in the high-level `TxOptimizer` class (using a variable transformation approach), so all optimizers can take advantage of this functionality.

Setting bounds on an optimizer is very , as seen below.

Listing 17: Setting bounds on optimizer.

```
// Now we set simple bound constraints on the argument.
optimizer -> setLowerBound(-50.0, 0); // lower bound on x[0]
optimizer -> setUpperBound( 0.5, 0); // upper bound on x[0]
optimizer -> setLowerBound( 0.0, 1); // lower bound on x[1]
optimizer -> setUpperBound(100.0, 1); // upper bound on x[1]
```

Calling the `optimizer -> solve()` method produces the output shown in Table 5.15. Again, the correct minimum value was found at the correct location (given the imposed constraints). During 143 iterations of the optimization algorithm, the single sum-of-squares function was evaluated 414 times. As before, the accuracy criterion was not satisfied, but the tolerance criteria were met.

More examples of using the nonlinear simplex optimizer with and without bounds constraints can be found in `demo/dmoptslv/simplex.cxx` and `demo/dmoptslv/simplexb.cxx` demos, respectively.

## 5.11 Nonlinear Simplex and Setting the Accuracy and Tolerance Criteria

The nonlinear simplex algorithm can be very robust for low-dimensional optimization problems, but it can also be very slow to zero in on a local minimum after it has approximately located it. This example uses the nonlinear simplex algorithm once more, but first overwrites the default accuracy and tolerance specifications with much less stringent values. Additionally, we choose to instantiate a new optimizer object, using one C++ function that returns the sum of squares of the Rosenbrock test functions.

Listing 18: Accuracy and tolerance.

```

delete optimizer;
optimizer = new TxNonLinSimplexOpt<double, TXSTD::vector<double> >(rbFunc);

// Send some pretty diagnostics to outFile.
tts.setFilter( TxThroughStream::TX_OUTPUT );
tts << TXSTRSTD::endl;
tts << "*****" << TXSTRSTD::endl;
tts << "*****" << TXSTRSTD::endl;
tts << "Nonlinear simplex algorithm, relaxed accuracy & tol.'s -- " << TXSTRSTD::endl;

// Write out the default values for accuracy and tolerance.
tts.precision(8);

```

The resulting output after calling `optimizer -> solve()` is shown in Table 5.15. Again, the correct minimum value was found at the correct location, within the greatly relaxed convergence criteria. During 46 iterations of the algorithm the single function was evaluated 141 times.

## 5.12 Conjugate Gradient Algorithm and Switching Optimizers in Midstream

Now that the nonlinear simplex has narrowed down where the minimum for the function is, we can pick up with the conjugate gradient algorithm. We instantiate the new optimizer object, again using our Rosenbrock sum-of-squares functor `rbFunc`. The starting point of the new optimizer is set using the minimum point found by the previous one.

Listing 19: Switching optimizers.

```

// Here we pick up with the conjugate gradient algorithm, where the
// nonlinear simplex algorithm left off.

delete optimizer;
optimizer = new TxConjugateGradOpt<double, TXSTD::vector<double> >(rbFunc);

// Set the starting point to the minPoint previously achieved by the
// nonlinear simplex algorithm.

optimizer -> setData(minPoint); // minPoint is a global variable, set from

```

The resulting output when the optimizer is run is shown in Table 5.15. The correct minimum value was found (now very accurately) at the correct location. During 8 iterations, the single sum-of-squares function was evaluated

136 times, while the corresponding gradient was evaluated only 9 times.

For more examples of using the conjugate gradient optimizer, see the demos `demo/dmoptslv/conjgrad.cxx` and `demo/dmoptslv/conjgradb.cxx`.

### 5.13 Powell's Algorithm and Invoking an Optimizer Step by Step

Finally, we demonstrate how the user can control the optimization process, step by step. For this example, we'll use Powell's optimization algorithm. We choose to instantiate the new optimizer object, using the old `rBFunc` object we have been using.

In the following code snippet, we demonstrate how to assume control of the optimization process and proceed step by step, rather than just invoking the `solve()` method, which leaves everything to the optimizer object. The diagnostic code that writes out the results is not shown here.

Listing 20: Invoking an optimizer step by step.

```
try {
  for (int i=0; i<30; i++) {
    optimizer->step();
    tts << TXSTRSTD::setw(6) << i+1
        << TXSTRSTD::setw(17) << optimizer->getMinPoint(0)
        << TXSTRSTD::setw(17) << optimizer->getMinPoint(1)
        << TXSTRSTD::setw(17) << optimizer->getMinValue() << TXSTRSTD::endl;
  }
  tts << TXSTRSTD::endl;

  // Once it appears the optimizer is on track, the user can ask it to
  // complete the optimization process.
  tts << "Now we stop calling the optimization algorithm step by step," << TXSTRSTD::endl;
  tts << "    and instead allow it to complete() on its own..." << TXSTRSTD::endl << TXSTRSTD::endl;
  optimizer -> complete();
}
```

The resulting output is shown in table 5.15. One can see the function value decrease monotonically, as the point varies from the initial guess of (4,-4) toward the optimal value of (1,1). Once the control is handed over to the optimizer, it finds the correct minimum function value at the correct location. During 7 (plus 30) iterations, the single sum-of-squares function was evaluated 1,197 times, while the corresponding gradient was evaluated just once (via finite differences).

Now we stop calling the optimization algorithm step by step, and instead allow it to complete on its own in Table 5.15.

Note that an optimizer's `solve()` method essentially calls the `step()` member function repeatedly until the solution criteria is met or the optimizer exceeds the set number of iterations. However, executing the optimizer step-by-step can be useful; for example, in creating graphs of the results of the optimization as it moves toward convergence. The user is free to decide how to run the optimizer.

More examples of using Powell's algorithm can be seen in the `demo/dmoptslv/powell.cxx` and `demo/dmoptslv/powellb.cxx` files.

### 5.14 Using the Parametric Model

OptSolve++ can also be used to fit a parametric model to a set of data points. This is done using a derived functor class. Once this specialized functor is created, it can then be passed to an optimizer and solved. The following example shows how to use this functionality. Source code for this example can be found in the file `demo/dmoptslv/fitting.cxx`.

The header file `TxLinParamModel.h` is needed to define a functor using a linear parametric model. Models that provide a Jacobian should use `TxJacParamModel.h`. In our example, we will implement the latter. The following code listing shows the class that will contain our model is defined.

Listing 21: Defining a parametric model class.

```
// typedefs
typedef TXSTD::vector<double> vec_t;

// The parametric model
class DemoModel : public TxJacParamModel<vec_t, vec_t> {

    typedef TxJacFuncIfc<vec_t, vec_t>::JacType JacType;

public:

    DemoModel(vec_t points, vec_t values);

    vec_t getPrediction(const vec_t& params) const throw (TxBadEvalExcept);

    JacType getJac(const vec_t& params) const throw (TxBadEvalExcept);

};
```

The class is similar to other functors, with one major exception — the `operator()` method is replaced by `getPrediction()`. This will be the method that implements the model. The `operator()` method is inherited from the parent class of `TxJacParamModel` and uses `getPrediction()` to calculate how much the model differs from the data set provided. Implementation of this method is shown below.

Listing 22: Defining `getPrediction()`.

```
/* getPrediction() is where you implement your parametric model.
   operator(), which is inherited from TxParametricModel, calls
   getPrediction() and calculates how much the model deviates from
   the observed data:  $f(t_j; a, \alpha) - y_j$ . */
vec_t DemoModel::
getPrediction(const vec_t& params) const throw (TxBadEvalExcept) {

    for (size_t j = 0; j < numPoints; j++)
        pred[j] = fitParams[0] * exp(-fitParams[1] * fitPoints[j]);

    return pred;
}
```

The constructor for our class takes two parameters (points and values) and uses these to initialize the inherited fields `fitPoints` and `fitVals`. The constructor also calls the constructor of the parent class, passing the number of parameters and the number of observed data points. The source code for the constructor is shown below.

Listing 23: Implementation of the parametric model's constructor.

```
/* Constructor calls the constructor of parent class and then
   initializes the data members holding the observed data. */
DemoModel::
DemoModel(vec_t points, vec_t values) :
    TxJacParamModel<vec_t, vec_t>(2, points.size()) {
    fitPoints = points;
    fitVals = values;
}
```

Finally, we must define the Jacobian calculation for the model, as seen below.

Listing 24: Implementation of the parametric model's Jacobian.

```

/* getJac() returns the Jacobian matrix of the parametric model,
   i. e. the derivative of operator() wrt the parameters. */
DemoModel::JacType DemoModel::
getJac(const vec_t& params) const throw (TxBadEvalExcept) {

    for (size_t j = 0; j < numPoints; j++) {
        jac(j, 0) = exp(-fitParams[1] * fitPoints[j]);
        jac(j, 1) =
            -fitPoints[j] * fitParams[0] * exp(-fitParams[1] * fitPoints[j]);
    }
    return jac;
}

```

Now that the model is defined, it can be instantiated, given data, and passed to an optimizer to be solved. This example generates its “observed data” artificially — in reality, the vectors  $t$  and  $y$  would come from experimental results or a similar source.

Listing 25: Optimizing using the parametric model.

```

// Number of observations.
const size_t N = 1000;

// Independent and dependent variables.
vec_t t(N), y(N);

// Generate the 'data': a time vector and its values.
for (size_t j = 0; j < N; j++) {
    t[j] = 1.0e-3 * j;
    y[j] = 1.0 * exp(-0.5 * t[j]);
}

// Initial point (a pretty bad one).
vec_t init_guess(2);
init_guess[0] = 0.1; // a
init_guess[1] = 20.0; // alpha

// Create the model and a LevMar optimizer.
DemoModel model(t, y);
TxLevMarOpt<double, vec_t> opt(model);

try {
    TXSTRSTD::cout << "Running the optimizer, please wait...\n"
        << TXSTRSTD::endl;

// Force the rate constant to be positive or the model can diverge.
    opt.setLowerBound(1.0, 0);
// Now optimize away!
    opt.solve(init_guess);
} catch (TxOptSlvExcept& osx) {
    TXSTRSTD::cerr << "Exception: Caught TxOptSlvExcept exception:\n"
        << osx.getMessage() << "\nFitting failed!!!"
        << TXSTRSTD::endl;

    return -1;
}

```

## 5.15 Output Examples

Table 6: Output from Levenberg-Marquardt optimizer

---

```

*****
Solving 2-D Rosenbrock problem with Levenberg-Marquardt --
The optimization algorithm obtained the following:

f(x) =      8.1006154e-28
x[0] =              1
x[1] =              1

-----
The optimization algorithm required the following:

Iterations:                6
Function evaluations:      7
Gradient evaluations:     18

-----
The specified and achieved minimum gradient tolerance and
the components of the gradient at the estimated minimum:

Normalized gradient tolerance:      0
Achieved gradient tolerance:        0

-----
The specified and achieved accuracies and tolerances:

Specified Accuracy:            0
Achieved Accuracy:            0

Specified Tol[0]:      4.7592825e-08
Achieved Tol[0]:      4.4310423e-09

Specified Tol[1]:      4.7592825e-08
Achieved Tol[1]:      1.1100038e-08

*****

```

---

Table 7: Using Levenberg-Marquardt with Broyden's Method

---

```
*****
Levenberg-Marquardt with Broyden's estimate of Jacobian --
The optimization algorithm obtained the following:
```

```
f(x) =      120.71552
x[0] =      2.6182558
x[1] =      5.3099081
```

```
-----
The optimization algorithm required the following:
```

```
Iterations:           100
Function evaluations: 101
Gradient evaluations:   6
```

```
-----
The specified and achieved minimum gradient tolerance and
the components of the gradient at the estimated minimum:
```

```
Normalized gradient tolerance:      0
Achieved gradient tolerance:        0
```

```
-----
The specified and achieved accuracies and tolerances:
```

```
Specified Accuracy:      0
Achieved Accuracy:       0

Specified Tol[0]:      4.7592825e-08
Achieved Tol[0]:       1

Specified Tol[1]:      4.7592825e-08
Achieved Tol[1]:       1
```

```
*****
```

---

Table 8: Output of nonlinear simplex using boundary constraints

---

```

*****
Using nonlinear simplex algorithm with simple bounds--

Simple bound constraints: -50.0 < x[0] <  0.5
                        0.0 < x[1] < 100.0

The optimization algorithm obtained the following:

f(x) =          3.25
x[0] =          0.5
x[1] =    2.265302e-13

-----
The optimization algorithm required the following:

Iterations:          1
Function evaluations: 6
Gradient evaluations: 0

-----
The specified and achieved minimum gradient tolerance and
the components of the gradient at the estimated minimum:

Normalized gradient tolerance:          0
Achieved gradient tolerance:           0

-----
The specified and achieved accuracies and tolerances:

Specified Accuracy: 1.2771119e-152
Achieved Accuracy: 4.6396461e+199

Specified Tol[0]:    2.4032021e-08
Achieved Tol[0]:    0

Specified Tol[1]:    2.4032021e-08
Achieved Tol[1]:    4.4855986e-15

*****

```

---

Table 9: Output of nonlinear simplex changing accuracy and tolerance criteria

---

```

*****
Nonlinear simplex algorithm, relaxed accuracy & tol.'s --

Default absolute accuracy = 2.2204462e-19
Default relative accuracy = 2.220446e-16

Default absolute tolerance = 4.7121608e-10
Default relative tolerance = 4.7121609e-08

Default gradient tolerance = 4.7121608e-10

Accuracies and tolerances have been greatly relaxed...
The optimization algorithm obtained the following:

f(x) =          18054.5
x[0] =             4
x[1] =            -3

-----
The optimization algorithm required the following:

Iterations:                1
Function evaluations:      6
Gradient evaluations:      0

-----
The specified and achieved minimum gradient tolerance and
the components of the gradient at the estimated minimum:

Normalized gradient tolerance:          0
Achieved gradient tolerance:           0

-----
The specified and achieved accuracies and tolerances:

Specified Accuracy: 1.2771119e-152
Achieved Accuracy: 4.6396461e+199

Specified Tol[0]:          471.2632
Achieved Tol[0]:           0

Specified Tol[1]:          471.2632
Achieved Tol[1]:           1

*****

```

---

Table 10: Finishing optimization with conjugate gradient.

---

```

*****
Conjugate gradient algorithm, using previous minPoint  --
The optimization algorithm obtained the following:

f(x) =      1.8444194e-24
x[0] =              1
x[1] =              1

-----
The optimization algorithm required the following:

Iterations:              16
Function evaluations:    333
Gradient evaluations:    17

-----
The specified and achieved minimum gradient tolerance and
the components of the gradient at the estimated minimum:

Normalized gradient tolerance:          0
Achieved gradient tolerance:            0

-----
The specified and achieved accuracies and tolerances:

Specified Accuracy:  2.2204462e-19
Achieved Accuracy:   8.1957578e-12

Specified Tol[0]:    4.7592825e-08
Achieved Tol[0]:     7.0085129e-09

Specified Tol[1]:    4.7592825e-08
Achieved Tol[1]:     1.3049889e-08

*****

```

---

Table 11: Using the Powell optimizer step by step

---

```
*****
Now using Powell's algorithm  --
```

Iteration	x[0]	x[1]	f(x)
0	4	-4	20004.5
1	0.001248439	1.5586e-06	0.49875234
2	0.16126505	0.026006417	0.35173816
3	0.21134071	0.044664897	0.31099173
4	0.24508382	0.06006608	0.28494922
5	0.27112335	0.073507871	0.26563059
6	0.29257092	0.085597744	0.25022795
7	0.31093159	0.096678451	0.23740764
8	0.32705704	0.1069663	0.22642612
9	0.34147981	0.11660846	0.21682442
10	0.35455686	0.12571057	0.20829842
11	0.3665401	0.13435164	0.20063573
12	0.37761443	0.14259266	0.1936819
13	0.38791999	0.15048192	0.18732097
14	0.39756566	0.15805846	0.18146357
15	0.40663793	0.16535441	0.17603927
16	0.4152067	0.1723966	0.1709916
17	0.42332935	0.17920774	0.16627452
18	0.43105362	0.18580722	0.16184999
19	0.43841968	0.19221182	0.15768623
20	0.4454617	0.19843613	0.15375636
21	0.45220898	0.20449296	0.1500375
22	0.45868687	0.21039364	0.14650995
23	0.46491746	0.21614824	0.14315666
24	0.47092011	0.22176575	0.13996276
25	0.47671193	0.22725427	0.1369152
26	0.48230808	0.23262108	0.13400246
27	0.48772208	0.23787282	0.13121434
28	0.49296603	0.24301551	0.12854172
29	0.49805085	0.24805465	0.12597648
30	0.50298636	0.25299527	0.1235112

---

Table 12: Finishing the Powell optimization.

---

The optimization algorithm obtained the following:

```
f(x) = 6.2746798e-25
x[0] = 1
x[1] = 1
```

-----

The optimization algorithm required the following:

```
Iterations: 7
Function evaluations: 1268
Gradient evaluations: 0
```

-----

The specified and achieved minimum gradient tolerance and the components of the gradient at the estimated minimum:

```
Normalized gradient tolerance: 0
Achieved gradient tolerance: 0
```

-----

The specified and achieved accuracies and tolerances:

```
Specified Accuracy: 2.2204462e-19
Achieved Accuracy: 1.325618e-08

Specified Tol[0]: 4.7592825e-08
Achieved Tol[0]: 1.7592359e-09

Specified Tol[1]: 4.7592825e-08
Achieved Tol[1]: 3.6292604e-09
```

---

## 6 Class Reference

This is a complete reference for all functor classes contained in OptSolve++.

Table 13: The Broyden classes

TxBroydGradFunc.h	Concrete class that calculates the gradient of any functor (with scalar value and vector argument) using the Broyden approximation.
TxBroydJacFunc.h	Concrete class that calculates the Jacobian of any functor (with vector value and vector argument) using the Broyden approximation.
TxBroydLevMarFunc.h	Levenberg-Marquardt functor that uses Broyden Jacobian.

Table 14: Bounds classes

TxBoundsIfc.h	Abstract base class describing imposition of bounds.
TxBounds.h	Base class describing all manipulations of bounds on functor arguments.
TxBoundedFunc.h	Wrapper for functors of type <code>TxVecArgFuncIfc;RetType, ArgVecType<math>\zeta</math></code> that adds boundedness, which is sometimes needed by the optimizers.
TxBoundedGradFunc.h	Wrapper for functors of type <code>TxGradFuncIfc;RetType, ArgVecType<math>\zeta</math></code> that adds boundedness, which is sometimes needed by the optimizers. Adds the function <code>getGrad()</code> to <code>TxBoundedFunc</code> .
TxBoundedHessFunc.h	Wrapper for functors of type <code>TxHessFuncIfc;RetType, ArgVecType<math>\zeta</math></code> that adds boundedness, which is sometimes needed by the optimizers. Adds the function <code>getHess()</code> to <code>TxBoundedGradFunc</code> .
TxBoundedLevMarFunc.h	Wrapper for functors of type <code>TxLevMarFuncIfc;RetType, ArgVecType<math>\zeta</math></code> that adds boundedness, which is sometimes needed by the Levenberg-Marquardt optimizer. Adds the function <code>calcGradAndHess()</code> to <code>TxBoundedHessFunc</code> and overloads <code>getHess()</code> .

Table 15: Function pointer classes

TxPtrFunctor.h	Concrete class for a functor with scalar value and scalar argument and no derivative defined. The user must pass in a pointer to a function defining the function value.
TxVecArgPtrFunctor.h	Concrete class for a functor with scalar value and vector argument and no gradient defined. The user must pass in a pointer to a function defining the function value and the dimension of the argument.
TxVecPtrFunctor.h	Concrete class for a functor with vector value and vector argument and no Jacobian defined. The user must pass in an array of either function or functor pointers defining the value of each component of the return vector, and the dimensions of the argument and the return value.
TxDerivPtrFunctor.h	Concrete class for a functor with scalar value and scalar argument and the derivative defined. The user must pass in two function pointers, one for the function value and one for the gradient.
TxGradPtrFunctor.h	Concrete class for a functor with scalar value and vector argument and the gradient defined. The user must provide function pointers that define the function value and gradient. The dimension of the argument must additionally be passed in as the last argument.
TxJacPtrFunctor.h	Concrete class for a functor with vector value and vector argument and the Jacobian defined. The user must pass in either two arrays of pointers to functions that define a function return value and a gradient, respectively. Alternatively, the user can pass in a single array of pointers to either a function or a functor that defines both a function return value and a gradient. Additionally, the dimensions of the argument and the return value must be passed in to the constructor.

Table 16: Base classes

TxFuncIfc.h	New abstract base class for all functors.
TxVecArgFuncIfc.h	New abstract intermediate base class for all functors with vector arguments.
TxVecFuncIfc.h	New abstract intermediate base class for all functors with vector arguments and vector return value.
TxDerivFuncIfc.h	New abstract base class for all differentiable functors with scalar value and scalar argument.
TxGradFuncIfc.h	New abstract intermediate base class for all differentiable functors with scalar value and vector argument.
TxJacFuncIfc.h	New abstract intermediate base class for all differentiable functors with vector value and vector argument.
TxHessFuncIfc.h	New abstract intermediate base class for all twice differentiable functors with scalar value and vector argument.
TxLevMarFuncIfc.h	New abstract intermediate base class for TxLevMarFunc and TxBoundedLevMarFunc.
TxLevMarFunc.h	Functor to be used by the Levenberg-Marquardt optimizer.

Table 17: Numerical classes

TxNumDerivFunc.h	Concrete class that calculates derivative of any functor (with scalar value and scalar argument) using finite differencing.
TxNumGradFunc.h	Concrete class that calculates the gradient of any functor (with scalar value and vector argument) using finite differencing.
TxNumJacFunc.h	Concrete class that calculates the Jacobian of any functor (with vector value and vector argument) using finite differencing.
TxNumLevMarFunc.h	Levenberg-Marquardt functor that uses numerical (finite differencing) Jacobian.
TxSecantDerivFunc.h	Concrete class that calculates derivative of any functor (with scalar value and scalar argument) using the secant approximation.
TxNumDiffMixin.h	Mixin class to provide accessor/mutator methods and hold data members needed for numerical differentiation.

Table 18: Fitting functor classes

TxFittingFunc.h	Functor for tuning parametric models to give the best least-square fit to observed data.
TxParametricModel.h	Abstract base class for all parametric models.
TxLinParamModel.h	Concrete class for linear parametric models.
TxJacParamModel.h	Abstract intermediate base class for all parametric models that provide a Jacobian.
TxSumSqFunc.h	Scalar-valued holder functor for a vector of functors as used, e.g. by the Levenberg-Marquardt optimizer.

Details regarding the structure of the OptSolve++ class libraries can be found in the technical report, "The OptSolve++ Software Components for Nonlinear Optimization and Root-Finding" at <http://www.txcorp.com/products/optsolve>, where documentation of the complete interface can also be found in HTML form. This manual describes the most commonly used methods and data members of the interface, but it is not a complete or exhaustive list.

When full source code is distributed, or when the ANSI C++ keyword `export` is widely supported by compiler vendors, the OptSolve++ template parameters can be chosen by users. For release, these template parameters may have only the following values:

ArgType	<b>double</b>
ArgAbsType	<b>double</b> (absolute value)
RetType	<b>double, float</b>
RetAbsType	<b>double, float</b> (absolute value)
PromoteType	<b>double</b> (result of mixing ArgType and RetType)
VecType	<code>std::vector&lt;double&gt;</code>
ArgVecType	<code>std::vector&lt;double&gt;</code> , <code>std::vector&lt;float&gt;</code> (vector of ArgTypes)
RetVecType	<code>std::vector&lt;double&gt;</code> , <code>std::vector&lt;float&gt;</code> (vector of RetType's)
PromoteVectorType	<code>std::vector&lt;double&gt;</code> (a vector of PromoteType's)

## 6.1 The Top-Level Interface

These are some of the most common methods in the OptSolve++ API. Information on the lower-level methods and data members can be found in the Doxygen-generated class reference.

- **void solve(const VecType& point) throw(TxOptSlvExcept)**  
Set initial point for solver (optimization or root-finding), then solve
  
- **void solve(const VecType& point, const VecType& scale) throw(TxOptSlvExcept)**  
Set initial point and scale/direction for solver (optimization or root-finding), then solve
  


---

- **void setData(const VecType&, const VecType&) throw(TxOptSlvExcept)**  
Set initial point and scale (or direction) in the initial guess.
  
- **void setData(const VecType&) throw(TxOptSlvExcept)**  
Set the initial point, but leave default value of the initial scale (or direction).
  


---

- **RetType getMinValue() const**  
Access the minimum function value (or value where algorithm failed.)
  
- **ArgType getMinPoint(size\_t i) const**  
Access the i-th coordinate of the point where the function is minimized.
  


---

- **void complete() throw(TxOptSlvExcept)**  
Complete the optimizing or root-finding after some pause, without resetting to initial guess.
- **void step() throw(TxOptSlvExcept)**  
Carry out one optimizer step.
- **void prepareStep() throw(TxOptSlvExcept)**  
Prepare for the next optimizer step.
- **void reset() throw(TxOptSlvExcept)**  
Reset the point and scale/direction to their initial values.
- **void setMaxIter(size\_t i)**  
Set the maximum number of iterations used.
- **unsigned getNumIter() const**  
Access the number of iterations used.

## 6.2 Methods Related to Convergence Criteria

- **bool isSolved() throw(TxOptSlvExcept)**  
Determine whether the desired accuracy or tolerance was achieved.
  - **bool isAccuracyAchieved()**  
Determine whether the desired accuracy was achieved – occurs when the change in the function value from one iteration to the next is sufficiently small.
  - **RetAbsType getRelAccuracy() const**  
Get the specified relative accuracy.
  - **void setAbsAccuracy(RetAbsType)**  
Set the desired absolute accuracy.
  - **RetAbsType getAbsAccuract() const**  
Get the specified absolute accuracy.
  - **RetAbsType getTotAccuracy() const**  
Get the total specified accuracy at convergence (depends on estimated minimum function value).
  - **RetAbsType getAchAccuracy() const**  
Get the accuracy actually achieved at convergence (or after maximum number of iterations).
-

- **bool isToleranceAchieved()**  
Determine whether the desired accuracy was achieved – occurs when the change in the location of the minimum point from one iteration to the next is sufficiently small.
- **void setAbsTolerance(const AbsArgType& tol)**  
Set the desired absolute tolerance in all dimensions to the same value.
- **void setAbsTolerance(const AbsArgType& tol, unsigned int i)**  
Set the desired absolute tolerance of the i-th variable.
- **AbsArgType getAbsTolerance(unsigned int i) const**  
Get the specified absolute tolerance of the i-th variable.
- **void setRelTolerance(const AbsArgType& tol)**  
Set the desired relative tolerance in all dimensions to the same value.
- **AbsArgType getAchTolerance(unsigned int i) const**  
Get the tolerance actually achieved at convergence (of the i-th variable.)
- **bool isGradientZero()**  
Determine whether the local gradient (normalized) is sufficiently close to zero.
- **void setAbsGradTolerance(const PromoteType&)**  
Set the desired absolute gradient tolerance.
- **PromoteType getAbsGradTolerance() const**  
Get the specified absolute gradient tolerance.
- **PromoteType getAchGradTolerance() const**  
Get the gradient tolerance actually achieved.

### 6.3 Methods Related to Setting Simple Bound Constraints

- **void setAllUpperBounds(const ArgType& bound)**  
Set all of the upper bounds to the same specified value.
- **void setAllUpperBounds(const VecType& boundVec)**  
Set all of the upper bounds to the specified array of values.
- **void setUpperBound(const VecType& bound, unsigned int i)**  
Set the value of the i-th upper bound.
- **ArgType getUpperBound(unsigned int i) const**  
Get the value of the i-th upper bound.

- 
- **void removeAllUpperBounds()**  
Remove all upper bounds.
  - **void removeUpperBound(unsigned int i)**  
Remove the i-th upper bound.
- 
- **void setAllLowerBounds(const ArgType& bound)**  
Set all of the lower bounds to the same specified value.
  - **void setAllLowerBounds(const VecType& boundVec)**  
Set all of the lower bounds to the specified array of values.
  - **void setLowerBound(const VecType& bound, unsigned int i)**  
Set the value of the i-th lower bound.
  - **ArgType getLowerBound(unsigned int i) const**  
Get the value of the i-th lower bound.
- 
- **void removeAllLowerBounds()**  
Remove all lower bounds.
  - **void removeLowerBound(unsigned int i)**  
Remove the i-th lower bound.