NONLINEAR OPTIMIZATION IN THE C++ BEAM OPTICS CODE

S. G. Shasharina, D. Bruhwiler, Tech-X Corporation, Boulder, CO, USA
J. R. Cary, Tech-X Corporation, Boulder, CO and University of Colorado, Boulder, CO, USA

Abstract

There are many optimization tasks in accelerator physics: for example, fitting twiss parameters or maximizing dynamic aperture. Optimization algorithms can be generalized to encompass a larger class of problems not restricted to accelerator physics. Modern computational methods provide means to do that, allowing us to treat one-dimensional and many-dimensional non-linear problems, problems for different types of variables (double, float or complex) uniformly. As a part of our project MAPA (Modular Accelerator Particles Analysis) [1], we have developed a C++ library for non-linear optimization and root finding [2]. We applied the results to fit twiss parameters and find fixed points in accelerators. In the paper we give more details on how we achieved the modularity, extensibility and generality of the library.

1 INTRODUCTION

OptSolve++ is a set of object-oriented C++ class libraries for nonlinear optimization and root finding. Each library can be embedded in other software and extended via inheritance and templates to add new capabilities. The components include TxOptSlv (optimizers and solvers), TxFunc (functor classes used to wrap user-defined functions), TxLin (linear algebra classes), and a library of standard test functions. The library compiles on PC, Macintosh, GNU/Linux and all major Unix platforms. It is free for non-commercial use and can be downloaded from www.techxhome.com/products/optsolve/index.html.

2 INTERFACE

By studying sequences of actions taken in optimization and solving, we concluded that the main method of the interface will be 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.

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 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 TXOPTSLV 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.

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 and solving algorithms that have been implemented.

The 1-D optimization algorithms include golden section [3] and Brent [4], which do not need the function derivative. 1-D solvers include secant and bisection methods. From multidimensional solvers, we implemented Newton Raphson method with Broyden update of the jacobian.

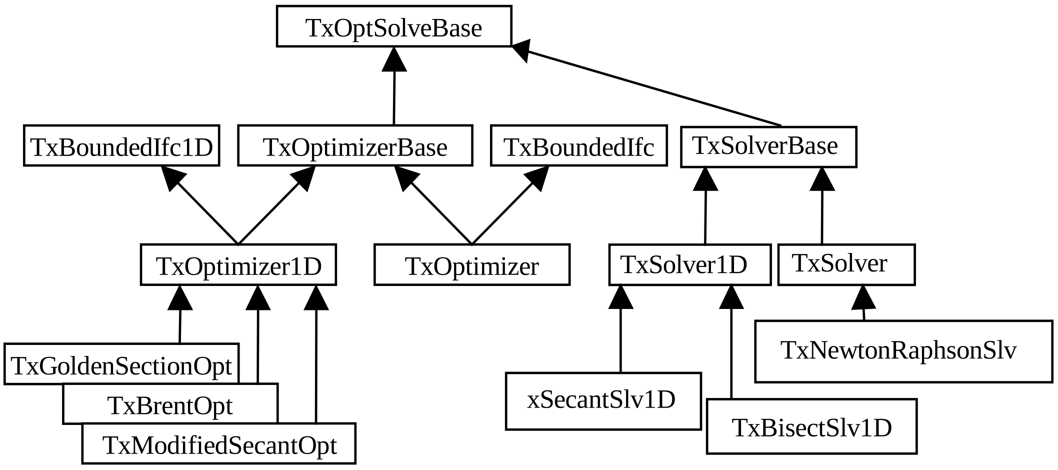

Figure 1: Simple class diagram showing the abstract classes of the TxOptSlv class library, plus the concrete classes that implement 1-D algorithms.

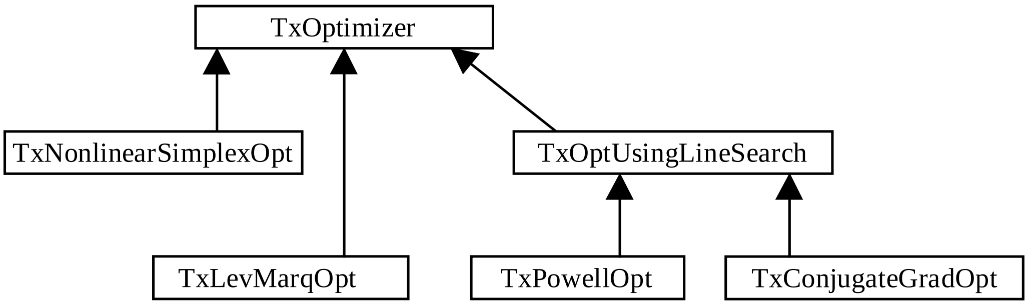

Figure 2: Simple class diagram for the multidimensional nonlinear optimization classes of the TxOptSlv class library.

The multidimensional optimizers include nonlinear simplex [5] and Powell [6], which do not need the gradient of the function, and the conjugate gradient method [7], which does require the gradient. We have also implemented Levenberg-Marquardt algorithm [7], [8] for nonlinear least squares. 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.

4 FUNCTORS AND THEIR USE

All optimizers are implemented in such a way that they take as an input either a pointer to a function or a pointer to a functor. Functor is an abstraction of a function and can wrap an arbitrary sequence of calls to user-specified functions and libraries, in order to calculate a value associated with the input arguments.

The abstract base class TxFunctorBase (for more detail see [2]) consists primarily of the parentheses operator, (), declared pure virtual and templated over both the argument type (ArgType) and the return type (RetType). From this class all specialized functors are defined. Thus, ArgType and RetType can be scalars of any type, or be vector type entities, so that all possibilities (one dimensional and multidimensional functors returning a number, or multidimensional functors returning a vector) can be generated.

In building this library we found one particular concept especially valuable. This is an idea of delegation of the action performed by class functions to an outside object. This is achieved by adding a new template parameter, a pointer to an object whose obligation is to define the required function or operator. For example, consider TxVectorFunctor class. It is derived from TxFunctor<RetVecType, ArgVecType>, and in addition to parenthesis operator has a pure virtual function RetVecType getFunctionValues (const ArgVecType&) declared. This function is needed to encapsulate subjects of multidimensional root finding and nonlinear least square problems. The parenthesis operator here returns the sum of squares of the functions values divided by 2. From TxVectorFunctor we can derive another class, TxVectorPtrFunctor, which will make the hierarchy particular useful for complex systems:

```
template <class RetVecType, class ArgVecType, class Ptr
= RetVecType (*) (const ArgVecType&) >
class TxVectorPtrFunctor : public
TxVectorFunctor<RetVecType, ArgVecType> {// etc.};
```

At the construction time a pointer to an object of Ptr type is passed to the instance. Then getFunctionValues method just calls for the function of the same name of the Ptr object. For example, if we want to fit twiss parameters or find fixed points of an accelerator, we should make sure the Accelerator class provides the function we want to be solved or minimized. Then we will be able to instantiate the following functor:

```
TxVectorPtrFunctor<std:vector<double>,
 std:vector<double>, Accelerator*>* func =
  new TxVectorPtrFunctor
       <std:vector<double>,
        std:vector<double>,
        acc*> (acc, acc.getDimension(),
                acc.getDimension());
```

where acc is an Accelerator object. Although this code looks a little bit heavy, it allows to instantiate a needed functor in the main program and use the same functor for various problems. We used this approach for finding fixed points in MAPA.

Another way to deal with complex objects is to create a specialized functor class for each problem, which will hold an instance of the pointer to the object and perform its functor duties through this pointer. This requires a new class for any new problem, but can be more attractive to people who get scared by heavily templated code like shown above. We used such approach to fit twiss parameters in MAPA.

5 CONCLUSIONS AND FUTURE DIRECTIONS

We have created a C++ library for nonlinear root finding and optimization. Due to generous use of templates and inheritance, the modules of the library are general and can be used for a vast class of problems and various types of arguments. Object oriented approach allows users to query optimization and solver objects about their internal state (achieved accuracy and tolerance, number of function and gradient evaluations, current number of steps etc.). All these internals can be changed if the progress towards solutions is not satisfactory. In addition, the optimizers can be interchanged while the optimized object stays the same. All this provides a great flexibility and makes the process of solution “more aware.” We used the library for finding fixed points in accelerator and fitting twiss parameters for desired values. The library has been also tested on a suite of infamously tricky functions traditionally used for testing of optimization routines [9].

In the nearest future we are planning to implement nonlinear constraints using penalty function approach and add integer optimization algorithms to the suite.

REFERENCES

[1] D. L. Bruhwiler, J. R. Cary and S. G. Shasharina, "MAPA: an Interactive Accelerator Design Code with GUI," Application of Accelerators in Research and Industry, AIP Conference Proceedings 475, (1999), p. 940.

[2] D. L. Bruhwiler, S. G. Shasharina and J. R. Cary, “Design and Implementation of Object Oriented C++ Library for Nonlinear Optimization,” Object Oriented Methods for Interoperable Scientific and Engineering Computing, Proceedings of the 1998 SIAM Workshop, 165 (1999).

[3] E. Polak, Computational Methods in Optimization, (Academic Press, 1971).

[4] R. Brent, Algorithms for Minimization without Derivatives, (Prentice-Hall, 1973).

[5] J. Nelder and R. Mead, Computer Journal 7 (1965), p. 308.

[6] D. Himmelblau, Applied Nonlinear Programming, (McGraw-Hill, 1972), p. 167.

[7] R. Fletcher, Practical Methods of Optimization, 2nd edition (John Wiley & Sons, 1987).

[8] A. Bjorck, Numerical Methods for Least Square Problems, (SIAM 1996).

[9] J. Moré and S. Wright, Optimization Software Guide, (SIAM, 1993).