UC-405

SAND94-8225
Unlimited Release
Printed March 4, 1994

OPT++: An Object-Oriented Class Library for Nonlinear Optimization

J. C. Meza
Scientific Computing Department
Sandia National Laboratories
P.O. Box 969, MS 9214
Livermore, CA 94551-0969
meza@ca.sandia.gov

ABSTRACT

Object-oriented programming is becoming a popular way of developing new software. The promise of this new programming paradigm is that software developed through these concepts will be more reliable and easier to re-use, thereby decreasing the time and cost of the software development cycle. This report describes the development of a C++ class library for nonlinear optimization. Using object-oriented techniques, this new library was designed so that the interface is easy to use while being general enough so that new optimization algorithms can be added easily to the existing framework.

Contents

1. Introduction

Object-oriented programming (OOP) is becoming a popular way of developing new software. Unlike procedural programming, which emphasizes the development of algorithms to accomplish a specific task, object-oriented programming relies on the implementation of new data types called objects. The promise behind this programming paradigm is that software developed through these concepts will be more reliable and easier to re-use in new applications, thereby decreasing the time and cost of the software development cycle.

The main concept behind object-oriented programming is called data abstraction, which is the separation of the data and the procedures for manipulating that data from an application program. In many ways this is no different than good programming practices that try to keep the unnecessary details of a particular code from an end-user. The major difference in object-oriented programming is the ability to create user-defined data types and add them to an existing language thereby facilitating data abstraction. It is these new objects that give object-oriented programming its name. Through these new objects a computer language can be easily extended to handle new applications. A good example of this feature is the matrix package developed by Davies ([4]). With this package, a user can define vectors and matrices as part of the language as well as use the standard operations defined for these objects, such as matrix addition, matrix multiplication, and inversion.

Another important trend is the renewed interest in nonlinear optimization. Optimization has always occupied a major role in industries, such as the airline industry, where scheduling problems are important. Recently, however, optimization has taken on an increasingly important role in areas such as advanced manufacturing where rapid design and prototyping of new processes and devices is essential. This trend is partly due to increased computer power available to users that allows for the repeated computer simulation of manufacturing processes and devices. While in the past the design process involved a large amount of human interaction, it is now becoming feasible to automate the design process using optimization tools. This trend in increased computer power has also had an effect within the optimization community where there has been an increased interest in large-scale nonlinear optimization problems [1].

Because of the wide variety of applications and the need to take advantage of any special structure in a problem, many software packages have been developed to address various types of optimization problems. For an excellent overview of the available optimization software see for example [7]. Unfortunately, the large number of optimization codes available makes choosing a good algorithm for a particular problem difficult. This is especially true for the novice practitioner of optimization. In addition, even if the methods are inherently similar, the interface to the codes can be quite different making it difficult to experiment with various methods. To resolve some of these issues code designers usually resort to one of two tricks: 1) force the user to use a particular calling sequence or 2) the optimization codes are written using reverse communication. Neither solution is very satisfying for the reasons explained below.

If the optimization algorithm requires a particular calling sequence the user is forced into

writing a subroutine that will interface between the optimizer and the function evaluator. While this is usually a straightforward task it may prove to be unwieldy and costly in certain situations. In particular, we would like to focus on cases where the function evaluator is described by the output of a simulation such as a finite-element analysis. In this case, the prescribed interface may not be general enough to encompass all of the parameters required to do a simulation or it may require the user to package any extra information in a pre-defined packed format.

The second option that is frequently used is called reverse communication. In this case, the optimization algorithm returns to the calling routine whenever it needs information to proceed. This information may be a function value, a derivative, or any other data that is required by the optimization algorithm. From the point of view of the user this is a better solution in that it requires less coding. From the point of view of the software developer however, the job is more difficult. Outside of the fact that this type of coding violates several good programming practices (for example, single entry-single exit codes), the code is also more difficult to debug. Another disadvantage is that software using reverse communication will be slightly more inefficient due to the frequent calling of and returning from subroutines that could involve several layers of subroutines.

The goal of this work is to use the ideas of object-oriented programming to overcome these obstacles. In particular we hope to address the following issues:

- better program interfaces for the user of optimization codes
- rapid evaluation of several optimization codes for a given problem
- rapid development of new optimization algorithms
- more re-usability of optimization codes

The rest of this paper is organized as follows. In Section 2 we introduce some concepts from object-oriented programming that will be useful for our discussion of the optimization classes. The reader who is familiar with object-oriented programming techniques can safely skip this section. Section 3 describes a C++ implementation of an object-oriented class library for unconstrained optimization. In Section 4 we give an example of using a particular class for solving a simple test problem. We conclude in Section 5 with a discussion of future work.

2. Object-Oriented Programming

There are four main ideas that we will use from object-oriented programming:

- abstraction
- classes and objects
- inheritance
- polymorphism

This report does not seek to give a full description of object-oriented programming, but merely to provide enough background material to discuss the new optimization classes. For a fuller description of object-oriented programming see [2, 3, 5, 10].

2.1. Abstraction

The idea of abstraction in software design is an old one. In its most general form, abstraction means the ability to isolate information pertaining to a particular software design. In procedural programming for example, the idea of abstraction has led to the concept of modular programming. In object-oriented programming this idea is taken further through the introduction of abstract data types. For the purposes of this paper we will define an *abstract data type* as a user-defined extension to an existing language type. It will usually consist of a set of data structures and a collection of operations that can manipulate those data structures. Through the use of abstraction, code will hopefully be more robust since details of data structures and the algorithms that manipulate them are isolated from the user.

2.2. Classes

The next concept that is useful is that of a class. A *class* is a user-defined data type that allows for data hiding. A class typically consists of both a data structure and a group of subroutines that can manipulate these data structures. The data inside the structure is hidden from the user in that the only way to access it is through the subroutines defined as part of the class. In this manner, the user does not need to know about the particular implementation of the class but can concentrate on the use of it. An *object* is then just a particular instance of a class. The analogy in a procedural language is that of a variable being a particular instance of a pre-defined type such as an integer.

An overworked but simple example is that of a complex data type. In this example, we could define a class called **complex** that consists of a pair of existing language types, for example, two **floats**. A better example is that of a class called **Vector** that could be defined as an array of **floats** together with an **int** that defines the size of the vector. The difference between the class **Vector** and an array which already exists in most languages is that we can now define operations that can be used with these objects. Thus we could define vector addition using the standard "+" operator between two **Vectors** of the same size.

2.3. Inheritance

Inheritance allows for easy extension of capabilities and is perhaps the most important new concept after that of the class. The idea behind inheritance is that a new class can be defined using a previously defined class as a template. In the terminology of OOP the template is called the *base class* and the new class is *derived* from the base class by adding new features to it.

One of the advantages of inheritance is that all of the algorithms defined as part of the old class are still valid for the new class. This results in more reusable code since it is not necessary to rewrite this portion of the algorithm for the derived classes.

2.4. Polymorphism

The last concept we will discuss is called *polymorphism*. In C++, it is possible to have a pointer to a function that will perform different actions depending on what class it belongs

to. In this way, it is possible to defer an algorithmic design decision until it is required. In the OOP terminology, these functions are called *virtual functions*. If a class contains virtual functions then it is called an *abstract class*. The reason for this distinction is that an abstract class can never be used to create an object, it can only be used as a base class for other derived classes.

3. Optimization Classes

There have been several attempts at designing optimization classes. In [9] Schoenberg developed a set of classes for the unconstrained optimization of arbitrary functions. Schoenberg describes 3 classes that together choose a particular algorithm, set the tolerances, and perform the actual optimization. Nichols et al. [8] have also developed optimization classes for linear operators in the physical sciences and specifically for linear operators arising from geophysical inversion problems.

We will take a slightly different approach by making a distinction between nonlinear problems and the methods used to solve these problems. The rationale for this decision is that users seldom are aware of the intricacies of the various methods nor should they need to become experts in numerical analysis. On the other extreme, the developer of optimization algorithms usually does not care about the details of how a problem is defined other than to know certain mathematical properties and some general problem characteristics. By making a distinction between problems and methods we can develop codes that will hopefully be used by both groups without having to rewrite the class libraries every time a new problem is presented or a new algorithm is developed.

We will write the general nonlinear optimization problem as follows:

$$\begin{aligned} \min_{x \in R^n} \quad & f(x) \\ \text{subject to} \quad & h_i(x) = 0, \qquad i = 1, \ldots, p, \\ & g_i(x) \geq 0, \qquad i = 1, \ldots, m. \end{aligned} \tag{1}$$

In this problem, the objective function $f(x)$ and the constraint functions $h_i(x)$ and $g_i(x)$ are assumed to be general nonlinear functions. In this report, we will limit our scope to consider only the unconstrained optimization problem. The question of whether classes for unconstrained optimization problems should be subclasses of the general optimization problem is rather tricky and we will delay the discussion of this issue until the last section.

The end-users of optimization algorithms are usually quite knowledgeable about the problems they are trying to solve. However, this information usually pertains to the physical problem or to the algorithmic details of the computer model. For instance, the user will know the dimension of the problem, whether analytic first or second derivatives are available, and a general idea about the cost of a function evaluation. The developer of optimization algorithms on the other hand, would usually like to know more about the mathematical properties of the problem as well as any special structure that might be exploited. For example, a developer might ask any or all of the following questions:

- How smooth is the function? Is the function C^0, C^1, C^2, etc.?

- Does the objective function have any special properties, for example, is it a linear function, a quadratic function, etc.?
- Is this a large dimensional problem?
- Is there any other special structure to the problem? For example, is this a partially separable problem?
- How many digits of accuracy does the objective function have? How many digits of accuracy does the derivative function have?
- Is the Hessian matrix sparse or dense?
- Is the objective function expensive to compute?

To consider the first property only, available optimization algorithms could be classified according to the amount of smoothness assumed in the objective function. For example, if the function is C^2 (twice continuously differentiable), then one could use a Newton method. However, if the function is only continuous, then one would probably use a direct-search method. For most users it may be difficult to prove how much continuity the objective function has and therefore they may not be able to pick the most appropriate method. What is more likely is that a user will use the first available optimization software or the easiest one to use among several, usually with mixed results.

It seems appropriate then to define nonlinear problems from the point of view of the user. On the other hand, optimization method classes should be defined from the point of view of the developer, as there is a great deal of similarity between various algorithms. In the rest of this section, we propose such a division and discuss a set of C++ classes for each one of these two cases.

3.1. Nonlinear Problem Classes

One of the first questions that arises is the degree of continuity in the objective function. This information may not be readily available, but what is clear is the availability of analytic derivatives. As such we've chosen to classify nonlinear programming problems by the availability of functions for computing the derivatives:

NLP0 – No derivative information available
NLP1 – Analytic first derivatives available
NLP2 – Analytic first and second derivatives available

In Figure 1, we present one implementation of a nonlinear problem class. The first class we define is called NLP0 for NonLinear Problem C^0. This class contains information common to all problems including: 1) the problem dimension, 2) a current point, 3) a function value, and 4) a function to evaluate the objective function.

The class NLP1 is derived from the base class NLP0 by adding a member for the gradient and a function to evaluate the gradient. Likewise, the class NLP2 is derived from NLP1 by adding the necessary information to compute and store the Hessian. By using inheritance we have been able to take advantage of the code that is already written at the lower levels.

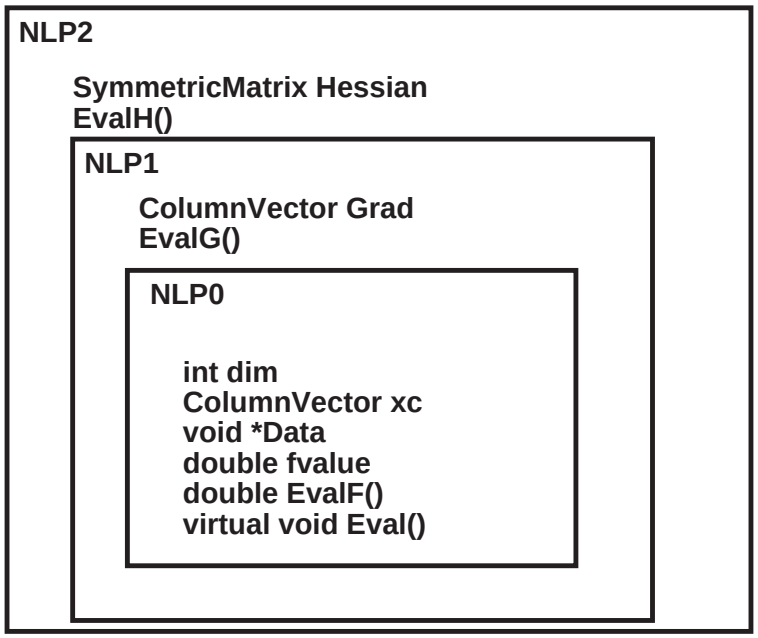

Figure 1: Nonlinear problem classes

It is not intended that these base classes cover every nonlinear problem, but starting with these classes the user can build new classes that contain the specific details of the real problem. Since the optimization method classes described below will use the base classes, the optimization algorithms will still work with the new user classes without having to be rewritten.

In our implementation of the optimization classes, we have defined the functions that evaluate the objective function, gradient, and Hessian as virtual functions. As we mentioned in the previous section, this means that the NLPX classes (where X can stand for 0, 1, or 2) are abstract classes and can only be used as base classes for other classes. This allows us to defer the definition of how the function, gradient, and Hessian are actually computed so that users can create their own definitions. In essence, the base classes contain placeholders for the codes that will be called to compute the objective function.

As part of our implementation we also provide 3 classes derived from NLPX called NLFX that have a particular calling sequence to the required functions. These classes can be used to solve some simple optimization problems or can be used as templates for more sophisticated objective functions. In Section 4, we will give some examples using the NLFX classes to demonstrate some of the features of our class libraries.

3.2. Optimization Method Classes

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

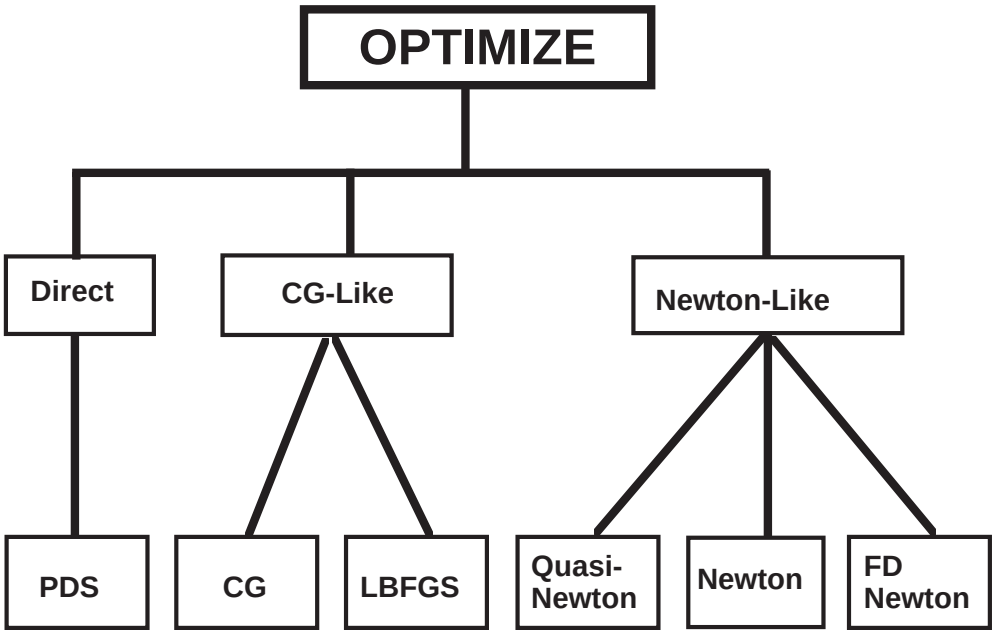

Figure 2: Optimization method hierarchy

- Direct Search methods
- Conjugate gradient like methods
- Newton like methods

For example, methods such as the Nelder-Mead simplex method, the box method, and the parallel direct search method fall into the direct search class. The nonlinear conjugate gradient method and limited memory BFGS methods fall into the Conjugate Gradient class. Finally the Newton class, could include methods such as finite-difference Newton, quasi-Newton methods, and inexact Newton methods. A simple taxonomy for some popular algorithms is given in Figure 2 as an example.

Based on this classification, we have implemented C++ classes for 4 different methods: 1) a Newton method, 2) a finite-difference Newton method, 3) a Quasi-Newton method, and 4) a nonlinear conjugate gradient method. In Figure 3, we present the class hierarchy for two of the implemented methods. The base class, called **Optimize** consists of information that is required by all optimization classes. We note that once again we have used the concept of polymorphism through the use of the virtual function **optimize()**. This function is intended to be a placeholder for the actual function that will do the optimization. Since each method class will have its own algorithm for computing the minimum of a function, it is not necessary to define it in the base class. However, it is important to define the interface at this point since it is common to all of the derived classes.

The next next two classes **OptQNewtonLike** and **OptCGLike** are derived from the **Optimize** class. The major difference between these two classes is that the Newton-like

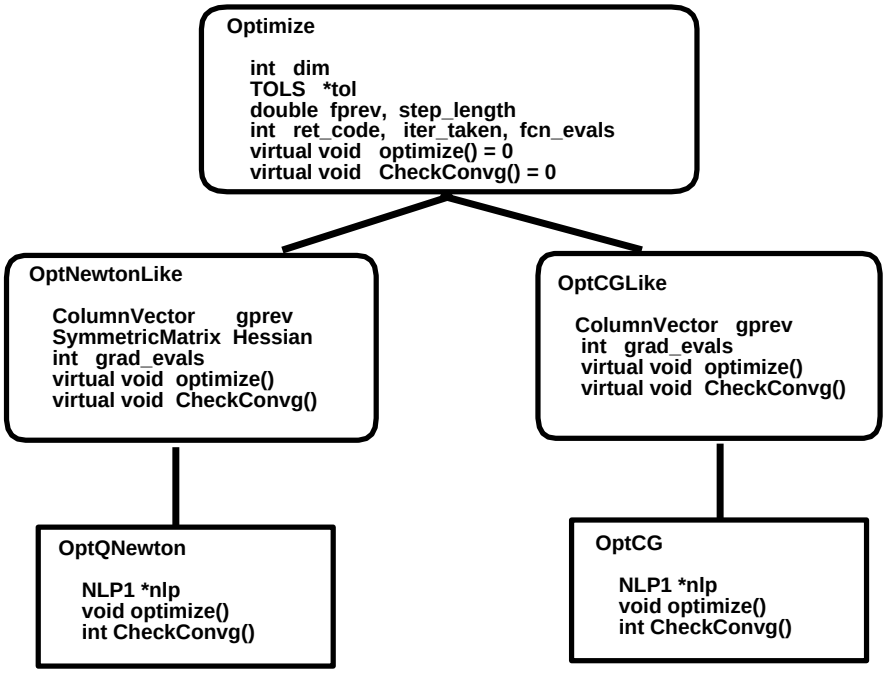

Figure 3: Optimization method classes

classes require extra storage for the Hessian matrix. Finally, the last two classes **OptQNewton** and **OptCG** constitute the actual optimization methods. It is these two classes that define the optimization algorithms specific to each method. In the case of the **OptQNewton** class, the algorithm consists of a Quasi-Newton method with a BFGS update formula for the Hessian. The **OptCG** class implements a nonlinear conjugate gradient method.

As an example of the re-usability of object-oriented codes, all of the linear algebra is handled through the use of the matrix package developed by Davies [4], with some minor enhancements for the matrices that arise in the optimization algorithms. In addition, all of the optimization methods use the same line search, which is based on the algorithm by More and Thuente [6].

4. Example Code

To illustrate some of the concepts, we now present an example that solves a small nonlinear optimization problem using the optimization classes. The test problem consists of Rosenbrock's function,

$$\min_x \; 100(x_2 - x_1^2)^2 \; + \; (1 - x_1)^2,$$

with an initial guess of $(-1.2, 1.0)$. In this example, we will assume that first derivatives are available but that second derivatives are not available. We will use a quasi-Newton method that employs a BFGS update formula for the Hessian.

```
1 #include "opt.h"
2 void rosen(int mode, int n, ColumnVector x, double& fx, ColumnVector& g);
3
4 main ()
5 {
6   int n = 2;
7   ColumnVector x(n), g(n);
8
9   USERFCN1 tstf = &rosen;                          // Define the test function
10
11  NLF1 nlp(n,tstf);                              // Define the Nonlinear problem
12
13  x(1) = -1.2;
14  x(2) = 1.0;
15  nlp.SetX(x);
16  nlp.Eval();                                      // Evaluate the function at x
17
18  TOLS tol;                                    // Create a "Tolerances" object and
19  tol.SetDefaultTol();                                          // set the tolerances
20  tol.SetFtol(1.e-9);
21  tol.SetMaxIter(100);
22
23  OptQNewton objfcn(&nlp,&tol); // Build a Quasi-Newton object and optimize
24
25  objfcn.optimize();
26
27  nlp.PrintState("Solution from quasi-newton");
28 }
```

Figure 4: Example code for solving Rosenbrock's function

Figure 4 displays the source listing for the sample problem. There are three major sections in the example code: 1) the problem definition, 2) the tolerance definition, and 3) the method definition. Since only first derivatives are available, we first create an object of type **NLF1** on line 11. The two components needed to specify this object are the dimension of the problem and a pointer to a function. The next step is to set the initial guess for this problem. Here we are using two of the member functions for **NLF1** to access the data in the class and to evaluate the function at the current point.

The next step is to create a **TOLS** object on lines 18-21 that contains the tolerances that will be used in the optimization method. In fact, the optimization method object can be created without a specific reference to a **TOLS** object but if the user wishes to change any of the default tolerances it is necessary to create the **TOLS** object.

The last step consists of creating an optimization method object from the **OptQNewton** class using the **NLF1** and **TOLS** objects. We then call the member function **optimize** on line 25 to do the actual optimization. Finally the solution is printed using the **PrintState** member function.

We note that if the user would now like to try a different optimization method, the procedure would consist of replacing line 23 with the creation of a different type of object, for example an **OptCG** object to try the nonlinear conjugate gradient method.

5. Summary

In this report, we have presented a C++ class library for nonlinear unconstrained optimization. We have proposed that a clear distinction be made between nonlinear problems and optimization methods. Based on this distinction, we have implemented a set of object-oriented classes specifically suited to each case. In this way, we have been able to develop a set of classes that address the important issues for both the users and the developers of optimization algorithms. From the point of view of a user requiring an optimization algorithm to solve a particular problem, these libraries have been written so that they are easily used. From the point of view of someone developing optimization algorithms, these classes have been designed so that new algorithms can be easily incorporated into the existing framework.

We currently have four methods implemented: 1) a Newton method, 2) a finite-difference Newton method, 3) a Quasi-Newton method, and 4) a nonlinear conjugate gradient method. Future work will concentrate on incorporating new algorithms. In particular, we are currently working on developing new algorithms based on pattern search methods for the case of noisy optimization. We are also working on implementing new classes for large-scale optimization. Since most of the popular methods for large-scale optimization use variations of one of the methods already implemented, the extension to large-scale problems should be straightforward.

Another area we will address concerns the case of constrained optimization problems. The question we wish to address is whether the constrained optimization case is a sub-class of the unconstrained optimization case or is a constrained optimization problem an unconstrained problem that happens to have constraints. In the OOP terminology, this is the "is-a" versus a "has-a" question, which has implications in the implementation of new classes.

Finally, we note that the libraries presented in this article should not be considered as a finished product. The true test will be the usefulness of these class libraries for solving real-world applications. Towards this end, we are also developing a suite of test problems from various manufacturing design problems using the nonlinear problem classes developed here.

REFERENCES

[1] Brett M. Averick and Jorge J. More. User guide for the MINPACK-2 test problem collection. Technical Report ANL/MCS-TM-157, Argonne National Laboratory, 1991.

[2] Timothy Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley, Reading, MA, 1991.

[3] David M. Butler. Fundamentals of object-oriented programming. Limit Point Systems, Fremont CA, 1992.

[4] R. B. Davies. NEWMAT07, an experimental matrix package in C++. robertd@kauri.vuw.ac.nz, 1993.

[5] Allen I. Holub. *C+ C++ Programming With Objects in C and C++.* McGraw-Hill, New York, NY, 1992.

[6] Jorge J. More and David J. Thuente. Line search algorithms with guaranteed sufficient decrease. Technical Report MCS-P330-1092, Argonne National Laboratory, 1992.

[7] Jorge J. More and Stephen J. Wright. *Optimization Software Guide.* SIAM Press, Philadelphia, PA, 1993.

[8] Dave Nichols, Geoff Dunbar, and Jon Claerbout. The C++ language in physical science. In *OON-SKI '93*, pages 339–353, April 1993. Proceedings of the First Annual Object-Oriented Numerics Conference.

[9] Ronald Schoenberg. An object-oriented design of an optimization module. In *OON-SKI '93*, pages 132–139, April 1993. Proceedings of the First Annual Object-Oriented Numerics Conference.

[10] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1987.