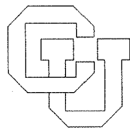
Developing Modular Software for Unconstrained Optimization

Robert. B. Schnabel

CU-CS-148-79 January 1979



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

DEVELOPING MODULAR SOFTWARE FOR UNCONSTRAINED OPTIMIZATION

Robert B. Schnabel
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado U.S.A.

Abstract. In this paper we support the use of modular software in constructing medium to large size numerical algorithms or systems of algorithms. First we discuss the advantages of modular numerical software, which include ease of development, testing, use and modification. Then we suggest a way of progressing from the initial high-level design of the modular system to the computer code through a series of descriptions which also serve as an important aid to understanding the system. As an example we use our recently developed system of quasi-Newton algorithms for unconstrained optimization. Our final pre-code description stage causes us to be interested in programming languages which allow user specification and efficient execution of basic data operations.

1. INTRODUCTION

This paper discusses the modular development of a system of algorithms for unconstrained optimization written recently by the author as part of a forthcoming book by Dennis and Schnabel [2]. The purpose of the paper, however, is not to talk about this specific numerical application, but rather about the issue of modular development of numerical software, its advantages, and some interesting considerations which arise when developing modular numerical software. The issues we discuss have been recognized and discussed by computer scientists and numerical analysts in the last ten years, but the application of these ideas to numerical software is still rather limited, and we hope to add a little impetus.

The activity we discuss, modular development of numerical software, precedes the performance evaluation of numerical software which is the topic of these Proceedings. However, we will point out that two important advantages of modular development are that it greatly facilitates controlled testing, and that it leads naturally to good documentation. Thus modular development enhances performance evaluation although its advantages are not limited to this.

Our paper is divided into two sections. In the first (Section 2) we describe what we mean by a modular system of algorithms, and discuss the important advantages of modular development. As an example we use the system of quasi-Newton algorithms developed by the author for the unconstrained minimization problem

$$\min_{x \in R^n} f: \quad R^n \to R \qquad (1.1)$$

in the case when f is assumed twice continuously differentiable. However, it is important to realize that most of our discussion applies equally well to most other complex numerical problems, especially to other iterative algorithms such as O. D. E. solvers. In Section 3 we discuss the communication (to implementors and

users) and computer implementation of a modular system, again using our system as an example. There are interesting questions concerning how to describe a modular system so that others can understand, code, use or modify it. We propose developing a description at two levels, a system description and an algorithmic description, before generating the actual code. Our algorithmic description leads us to be interested in facilities for defining primitive data operations, which are now available in a number of research computer languages but not yet in any production language.

2. DESCRIPTION AND ADVANTAGES OF THE MODULAR SYSTEM

In this section we describe the structure of our modular system for unconstrained optimization, and use it to illustrate the advantages of modular numerical software in general.

To discuss our modular system, one needs to know just a little bit about algorithms for problem (1.1). A user employing software to solve a particular unconstrained optimization problem will supply the function f, a starting point x_0, and various tolerances. Our system is concerned with the common case when f is also twice continuously differentiable, although the user may or may not be able to supply ∇f or $\nabla^2 f$. (For simplification in this paper only, we assume ∇f is available.) The type of algorithm favored for such a problem is a "quasi-Newton" algorithm, which we view as having the following form. (For background on quasi-Newton methods, see Dennis and Moré [1].)

Algorithm 2.1 -- Quasi-Newton framework for unconstrained optimization

I. Initialization

II. Iteration

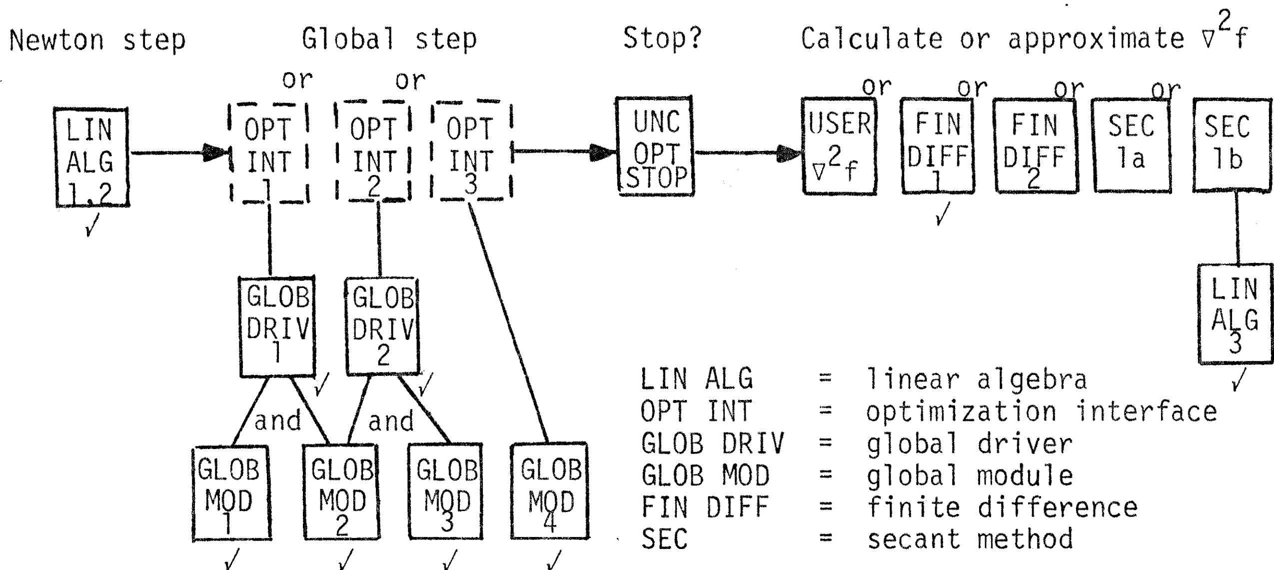
given: $x_i \in R^n$, $\nabla f(x_i)$, $\nabla^2 f(x_i)$ or an approximation to it

find: $x_{i+1} \in R^n$ a better approximation to the minimum

1. Calculate "Newton-ish step" p_n, using an appropriate local model of f(x)
2. Using p_n, calculate x_{i+1} ("Global strategy")
3. Decide whether to Stop; if not
4. Calculate or approximate $\nabla^2 f(x_{i+1})$

The structure of Algorithm 2.1 is all we wished to say about unconstrained optimization algorithms themselves -- just the fact that the algorithms have an initialization step followed by an iterative step with four distinct parts. But we should mention immediately that some researchers in unconstrained optimization take issue with our type of description, because they feel the pieces within the iteration cannot be completely separated. Their beliefs preclude a modular system. However, if one accepts the above (or any similar structure) a modular system follows easily. We give a diagram of our entire system of algorithms for the iteration step below.

Fig. 2.2 -- Modular iteration for unconstrained optimization



The four sections in Figure 2.2 correspond to the four steps in Algorithm 2.1. The solid boxes are modules. The Newton step and stopping criteria correspond to two and one rather simple modules, respectively. The other two parts are more interesting, and serve to illustrate some capabilities of modular numerical software. For the global step our system provides three alternative strategies, as there is a real difference of opinion in the field as to which is the best way to proceed. The first two are "model trust region" strategies which are seen to be related because they share a module. They each use a driver which does little more than call the two modules (1 and 2, or 3 and 2, respectively) alternately until a satisfactory next point is found. (The dashed "interface" box is explained at the end of this section.) The third strategy is a line search. What is important to note is that in a particular run of our system, a certain one of these three strategies would be selected and used exclusively. However, a software system developed from our description could contain all three with the user choosing one at run time, or it could contain just one of the three strategies if the implementor has a strong personal preference.

The five alternative modules for the "$\nabla^2 f$" step are used a bit differently. The choice of which to use depends on whether $\nabla^2 f$ is provided analytically, and if not, on how expensive calculation of f and ∇f is. Therefore a *system* of algorithms should contain all five; again, a particular run will choose just one.

From this description, the advantages of modular development become evident. We identify them in three categories:

a) Development: A modular structure allows naturally for top-down development, that is, starting at the highest level and successively adding levels of detail. This has become acknowledged by most computer scientists as the most effective manner to develop code. In Section 3 we will see that another advantage of our manner of top-down development is that it leads naturally to good documentation *at any early stage*.

b) Testing and research: A modular system is generally preferred for testing or verifying a large system, as it allows this to be done in manageable parts. It is also excellent for research, because it provides a controlled environment for testing alternative strategies. For example, in unconstrained optimization research one often wants to compare a new global strategy (Step 2, Alg. 2.1) or a new derivative approximation strategy (Step 4) to an existing one. If one changes only this portion of a system like Fig. 2.2, the differences in performance are clearly attributable to the new strategy. This is in sharp contrast to many test results reported

today, where the two strategies being compared are embedded in two different codes, severely confusing the interpretation of the test results. (A common difference between codes which is acknowledged to often hinder the drawing of conclusions from test results is different stopping criteria.)

c) Software library: A modular system provides two additional advantages when used as part of a software library. One is the convenient provision of alternative capabilities or algorithms, as was discussed above. The second is that routines for related problems may share many of the same modules. Indeed, this was a prime motivation in our development of a modular system for unconstrained optimization. We have concurrently developed a modular system for solving systems of nonlinear equations,

$$\text{given } F: \ R^n \rightarrow R^n, \text{ find } x \in R^n \text{ such that } F(x) = \underline{0} \qquad (2.3)$$

which are also solved by a quasi-Newton algorithm with the same four-part iteration as Alg. 2.1. It makes use of the ten modules indicated by checks (✓) in Fig. 2.2, including all the global modules which are the bulk of the system. Thus the development of the second system is *far* less work than the first. This also explains the purpose of the three interfaces in Fig. 2.2: while the systems for unconstrained optimization and nonlinear equations use the same global modules, they call these modules with parameters particular to their application. The interfaces simply give the correspondences of the calling sequences of the main programs to the parameter sequences of the modules.

3. COMMUNICATION AND COMPUTER IMPLEMENTATION OF THE MODULAR SYSTEM

At this point one may be convinced of the advantages of a modular system, but wary of the difficulty involved in transforming it into code, and in supplying documentation which will make it reasonably easy for an outsider to use, and for a numerical analyst to understand or modify. In this section we discuss the resolution of these problems in a medium to large size numerical system such as ours. We show that we are led rather naturally to two intermediate descriptive steps between the modular diagram of Fig. 2.2 and the ultimate code: a "system description" and then an "algorithmic description." The latter leads to an interesting consideration in programming language design.

Perhaps the best way to start is to ask what our final optimization system corresponding to Fig. 2.2 will consist of. The modules will probably each be subroutines or procedures, each with certain specified input, output and global variables. Note that the main driver (which along with the initialization module was omitted from Fig. 2.2) is itself such a module, with its input and output going from and to the user respectively. The interfaces will essentially disappear, having specified calling sequences in the code. The documentation which will be required is: the modular diagram Fig. 2.2 and guidelines on how to choose between its options, the input requirements and guidelines on specifying the input, and perhaps guidelines on interpreting the output. It is important to note that this is *all* the documentation required.

The question remaining is how the modules are gotten to fit together. To accomplish this easily is one reason why we suggest a system description as the next stage after the modular diagram of Fig. 2.2. Two related reasons are that this is sensible top-down development, and that it leads to a very understandable high-level description of the system.

By a system description we mean a level of description specifying the purpose and arguments of each piece of Fig. 2.2, and also describing how the entire system is to be used. Ours consists of three parts: high level module descriptions, interfaces and guidelines. Each module description at this level is simply a list of input, output and global variables used by the module, and a very brief statement of its function. An example is the top third (down to "Output") of Fig. 3.1. Each interface is just a list matching the parameter sequence of a module with the

calling sequence it will be invoked with. The guidelines are precisely those for module selection, input and output mentioned above. Note that this amount of information completely specifies the linkage and use of our system. In addition, the documentation has naturally been completed at an early stage of system development.

What remains is to specify the algorithm of each module. One could of course just go and code each one. However, we believe a second intermediate stage is highly desirable, both in the initial development, and so that afterwards people can learn about the algorithms without having to go immediately to the code. Therefore we have produced a description of the algorithm of each module before the actual code. This level added to the system description forms our algorithmic description.

We have actually decided to describe the algorithm of each module in two stages. The first is a rather brief (typically 5-10 line) description in prose or outline form which relates the main points but not all the details of the algorithm. An example is the "Description" in the middle of Fig. 3.1 But the most interesting part is that then, rather than going directly to the code as would be expected, we have specified each algorithm in complete detail in "PASCALish" code, the first few lines of which for one module are shown below (under "Implementation"). We now discuss the reasons for this.

Fig. 3.1 -- Description of global module 3

Global variables: D_S - diagonal scaling matrix for x, (described in Guideline 2)

Input: $n \in R$, $x \in R^n$, $g \in R^n$, lower triangular $L \in R^{n \times n}$, $p_N \in R^n$ (where $p_N \triangleq -M^{-1} g$, $M \triangleq LL^T$), $\delta \in R$

Find: $x_+ \in R^n$ which approximates the solution to

$$\min f_Q(x_+) = f(x) + (x_+ - x)^T g + \frac{1}{2}(x_+ - x)^T M (x_+ - x)$$

$$\text{subject to } \| D_S(x_+ - x) \| \le \delta.$$

(by finding the x_+ which solves this problem among all points on the double dogleg curve)

Output: $x_+ \in R^n$, $\delta \in R$.

Description: If $\| D_S p_N \|_2 \le \delta$, $x_+ = x + p_N$ (p_N is the Newton step to the minimum of the quadratic model). Otherwise x_+ is chosen to be the (unique) point on the double dogleg curve such that $\| D_S(x_+ - x) \| = \delta$. The double dogleg curve consists of the three line segments connecting x, $x + p_C$, $x + p_N$, $x + \eta p_N$, where $x + p_C$ is the Cauchy point, the minimum of the quadratic model in the Cauchy (steepest decent) direction $-D_S^{-2} g$, and $\eta \le 1$ (see Ch. 6).

Implementation:

```
Newtlen := ||D_S p_N||_2;
if Newtlen ≤ δ
    then begin (* x_+ is Newton point *)
          x_+ := x + p_N;
          δ := Newtlen
    end
else begin (* Newton step too long -- x_+ on dogleg curve *)
```

$$\rho := \frac{\| D_S^{-1} g \|_2^4}{\| L^T D_S^{-2} g \|_2^2 (g^T p_N)}; \qquad (* \; \| L^T D_S^{-2} g \|_2^2 = g^T D_S^{-2} M D_S^{-2} g \; *)$$

Our PASCALish code is algorithmically *complete*, and syntactically correct with three exceptions:

a) mathematical variables (Greek, sub or superscripted) are allowed.

b) input and output statements are informally specified.

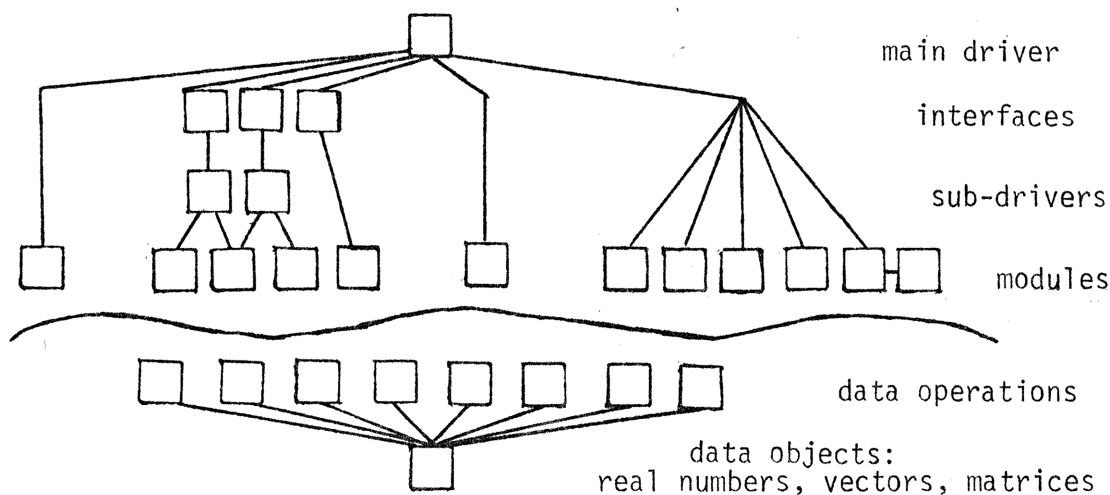
c) basic mathematical and linear algebra operations are allowed.

The big advantage of this description over actual code is that it is very readable and understandable, whereas removing a or c (and to a lesser extent b) renders it *much* harder to understand. In addition, it is almost trivially translatable into any programming language of interest. Our students have programmed much of the system in FORTRAN and its dialects WATFOR and FLEX, as well as PASCAL, with great ease; PL/1 and ALGOL would be equally easy. The reason PASCAL was chosen for the descriptive language is that it is an algorithmic language designed for readability and easy understanding. (PL/1 or ALGOL would also have been appropriate.) Our descriptions do not make use of any features of PASCAL which are not readily convertible into all languages of interest, including FORTRAN.

The PASCALish code completes the algorithmic description of our modular system. However, there turns out to be an interesting consideration in programming language design which arises in converting this description to code. Note that all that has to be done is to translate the PASCAL statements to the corresponding statements in another language if necessary, and eliminate exceptions a, b and c above. Exceptions a and b are trivial to rectify (e.g., δ probably becomes DELTA). Exception c is also quite easy, but what we would really like is for c to be allowed by the programming language. This turns out to be an issue that is well recognized by researchers in programming language design.

Notice that what we have done up to the algorithmic description level is form top-down a modular system, underlying which are certain data types (integer, real, vector, matrix) and a small number of basic data operations on these types. This is shown in Figure 3.2.

Fig. 3.2 -- Overall structure of the software system



In our whole system, the data operations turn out to be

$$\sum_{i=1}^{n} \alpha_i, \ \max\{\alpha,\beta\}, \ \min\{\alpha,\beta\}, \ \gamma \in [\alpha,\beta], \ v^T w, \ \|v\|_2, \ A \cdot B, \ A \cdot v$$

where $\alpha, \beta, \gamma \in R$, $v, w \in R^n$, $A, B \in R^{n \times n}$. The two-part hierarchy of Fig. 3.2 (algorithms and data) has become recognized as the form which arises from developing almost any medium or large system (in non-numeric systems the underlying data types and operations are often far more complex than in numeric systems), and there are

starting to be ways to actually write the system so that the underlying data operations can be primitives in the code which are also efficiently executed.

Of course the data operations can simply be made procedures, which is what has usually been done in coding our system. While this is a convenient solution, it is less efficient than we would like due to the cost of procedure linkage. A more efficient solution is the use of a macro-generator, which allows one to specify simple procedures which are then compiled and executed as in-line code with the proper parameter substitution. This permits the basic data operations to be executed efficiently. (For an example, see Myers [4].) Perhaps better still, there are several research programming languages which not only enable the expression and efficient use of basic data operations, but also are oriented to supporting a system with the form of Fig. 3.2. These include SIMULA, MODULA and EUCLID; for a good reference to the work in this field, including references to all of these languages, see Goos and Kastens [3]. Since it is clear that most numerical algorithms or systems of algorithms, if they are developed modularly, will take a form similar to Fig. 3.2, we believe that the numerical software community should be interested in programming languages which allow the specification and use of basic data operations, and perhaps which are also oriented to supporting such modular systems.

Acknowledgement

My thanks to P. Zeiger (Dept. of Computer Science, University of Colorado at Boulder) for first pointing out the basic data operations issue in my algorithmic descriptions, and to L. Osterweil (same department) and W. Waite (Dept. of Electrical Engineering, University of Colorado at Boulder) for informing me about some of the work in this area.

Boulder, Colorado January, 1979

References

[1] J. E. Dennis and J. J. Moré, "Quasi-Newton methods, motivation and theory", SIAM Review 19 (1977) 46-89.

[2] J. E. Dennis and R. B. Schnabel, "Quasi-Newton methods for nonlinear problems", manuscript in preparation.

[3] G. Goos and U. Kastens, "Programming languages and the design of modular programs", in: P. G. Hubbard, S. A. Schuman eds., Constructing Quality Software, Proceedings of the IFIP TC2 working conference on constructing quality software (North-Holland, Amsterdam, 1978).

[4] E. Myers, "The BIGMAC user's manual", report no. CU-CS-145-78, Department of Computer Science, University of Colorado, Boulder, Colorado (1978).