Some Advances Related to Nonlinear Programming

F. W. Pfeiffer

I Introduction

The purpose of this communication is to inform members of the mathematical programming (MP) community of some advances -- past, present, and future -- which should have a noticeable impact upon applications involving nonlinear programming (NLP). These advances have not been in the theoretical area, as is usually the case, but rather they have come in the area of computer software and hardware designed with nonlinear programming in mind.

II Derivatives

Many of the most elegant, the most general, and the most powerful numerical methods of nonlinear programming require first- and/or second-order partial derivatives -- sometimes in great numbers -- as components of the gradient vector or the Jacobian and Hessian matrices. Obtaining explicit formulas for these derivatives has, for one reason or another, been an obstacle that many have chosen to avoid. As evidence of this, one has only to note the work done so far on quasi-Newton and derivative-free methods. Formulas for the components of the Hessian matrix, in Newton's recurrence relation below,

$$\begin{pmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_m \end{pmatrix}_{n+1} = \begin{pmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_m \end{pmatrix}_{n} - \begin{pmatrix} \frac{\partial^2 F}{\partial x_1^2} & \cdots & \frac{\partial^2 F}{\partial x_m \partial x_1} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \frac{\partial^2 F}{\partial x_1 \partial x_m} & \cdots & \frac{\partial^2 F}{\partial x_m^2} \end{pmatrix}_{n}^{-1} \begin{pmatrix} \frac{\partial F}{\partial x_1} \\ \cdot \\ \cdot \\ \cdot \\ \frac{\partial F}{\partial x_m} \end{pmatrix}_{n}$$

are usually far too much work to derive and code by hand because these formulas are too numerous or become too complicated. If one is tenacious enough to use methods requiring the Hessian at all, one usually resorts to obtaining it by error-ridden, inefficient, and sometimes costly differencing schemes.

The way around this persistent obstacle is to leave the entire job of generating code for the numerical evaluation of derivatives up to some compiler or interpreter. Executing this code and passing the results to an NLP algorithm at each iteration can be handled in this manner also. We call this "automatic derivative evaluation" (ADE). The word "automatic" here means "completely automatic," and the only way to avoid having derivatives evaluated is to choose an algorithm that does not require them. ADE requires that when machine code for an objective function and any constraint functions present is generated, code for evaluating derivatives of the objective and constraint functions with respect to the unknowns (independent variables) will be generated also. The code then loaded into memory will consist of, among other things, code resulting from the objective and constraint functions plus the code generated for evaluating any required first and second derivatives. And thus, when the code executes, all of the derivatives needed by the NLP algorithm in use will be evaluated and available at each iter-

ation. The NLP problem solver is completely relieved of the job of seeing to it that derivatives are evaluated.

There are some important advantages to automatic derivative evaluation, and these of course depend upon how comparisons are made. First, ADE allows the NLP problem solver to make changes to objective and constraint functions at will without rederiving and recoding of derivative formulas. Second, the code generated for derivative evaluation will be free of errors once the compiler or interpreter is free of related errors. Third, the number of man-hours spent on deriving and coding formulas (exact or difference) for derivatives by hand can be completely eliminated from the total amount of time it takes to solve an NLP problem or test a new algorithm. Furthermore, if the code generated is for numerically-exact derivatives, then there are even more advantages to ADE. Fourth, algorithms can be much more reliable -- particularly when the Hessian's eigenstructure is used. Fifth, the execution efficiency of algorithms can be improved, because the number of iterations to convergence can be noticeably less, sometimes, than when approximate derivatives are used, and because of fewer function evaluations required for each derivative. There are enough advantages to ADE to make the devoted NLPer wonder why the major thrust in NLP research has not been in this direction, i.e. how to obtain derivatives instead of how to avoid them. Had this idea been pursued years ago, researchers might have discovered a third way to compute derivatives.

That derivatives be evaluated automatically, efficiently, and exactly today for the benefit of NLP algorithms and problems is not an unreasonable request, since the algorithms and/or software for doing this were created at IBM, GE, and TRW about 15 or more years ago. Of interest here is the GE-TRW approach. In 1964, Wengert at General Electric described [1] a clever numerical method for obtaining derivatives "automatically" while evaluating elementary functions. At the same time, Wilkins, from the same GE group, discussed [2] an application of the new method to a problem requiring 63 first derivatives. In 1965, Bellman, Kagiwada, and Kalaba reported [3] on their experiences with the method on a multipoint boundary-value problem. Up until this time, however, the use of Wengert's method could still represent a significant coding task if done manually -- which it was. About 2 years after the GE effort, the need for a change in the way derivatives were being obtained was sensed by some in a group at TRW. In 1965, M. W. Alford discovered a more efficient version of Wengert's computational scheme -- independently. Moreover, the TRW group, blessed with compiler writers, carried the idea even further to the point of automating it completely and incorporating the method eventually into a TRW language called SLANG in 1967. So automated was Wengert's method that few users of SLANG appear to have ever been aware of its presence.

Since it is the numerical values of derivatives that are needed in numerical NLP work, and not the formulas for them (the IBM approach), Wengert's method seems to be more appropriate. But just how his method compares with the other two is not known very precisely. The only comparative study [4] is incomplete. In this study, only execution times and numerical accuracy are accurately measured for Wengert's method and central differencing. These measurements resulted in a plot of execution time versus number of independent variables for 2 functions. For both functions, a complete set of first and second derivatives were computed at 20 points inside a unit hypercube. Wengert's

method became faster than central differencing only after the number of unknowns exceeded 5. An educated guess was made as to the space and programming effort required by each of the three methods. As one might expect, differencing still requires the least space and programming. A minimal comparison of the three methods ought to require running many problems three times, once for each method, so that space, time, and accuracy statistics for each could be gathered. This much could be accomplished manually. However, of much more interest would be the experiment of writing three derivative compilers, one for each method, since it is automatic derivative evaluation that is really needed in NLP work.

III Software

The idea of including statements in programming languages for computing derivatives and integrals numerically has not, apparently, occurred to many language designers yet. Statements like

```
U = INTEGRAL(F,A,B), or
V = DERIVATIVE(F)
```

are certainly not too difficult for a compiler to translate. Statements for computing derivatives of scalar and vector functions would be most useful in nonlinear programming, particularly when the derivatives are computed efficiently and exactly. In 1973, J. M. Thames and M. N. Robinson, from the TRW group mentioned earlier, did include statements for the scalar case in a language they were designing. Four examples are shown here.

`U = .PARTIAL(F,X)`	$\partial F/\partial X$
`V = .PARTIAL(F,X,Y)`	$\partial^2 F/\partial X \partial Y$
`G = .GRAD(F)`	∇F
`H = .HESS(F)`	$\nabla^2 F$

These statements can be used to obtain the indicated derivatives by Wengert's method for any real or complex scalar function defined previously. F should of course be continuous and continuously differentiable, once or twice. As for the Jacobian, it can be obtained, row by row, using .GRAD(F) along with other array function calls. Having statements for computing derivatives is a powerful feature, but as the reader will soon see, even the requirement for such statements as these can be eliminated from the programming of NLP problems.

With the problem of automatic derivative evaluation out of the way, Thames and Robinson turned their attention to the problem of automating mathematical software for mathematical programming in a general-purpose programming language. One of the main objectives in the design of PROSE (the name of the new language) was to reduce the programming of mathematical programming problems to a minimum. This required that NLP solvers (algorithms) be referenced and changed as easily as we reference and change trigonometric functions in FORTRAN. All of this was accomplished, in part, by putting all mathematical programming solvers in a common mathematical library, giving each a name, each a common interface, and leaving the job of linking a solver to the user's problem up to the PROSE executive system. Also left up to the executive was the automatic evaluation of derivatives -- behind the scenes -- when first-

or second-order NLP solvers were being used.

A constrained nonlinear programming problem might be described in English as

find $x_1,\ldots,x_n$ such that $l_i \le x_i \le u_i$ i=1,...,n
holding $g_k(x_1,\ldots,x_n) \ge 0$ k=1,...,m
satisfying $h_j(x_1,\ldots,x_n) = 0$ j=1,...,n
to maximize $y = f(x_1,\ldots,x_n)$.

In PROSE, the same problem is described by a single statement of the form:

```
FIND x1,...,xn IN model BY solver
WITH LOWER l1,...,ln
AND UPPER u1,...,un
HOLDING g1,...,gm
MATCHING h1,...,hn
TO MAXIMIZE y
```

All mathematical programming solvers are activated by the FIND statement. This statement contains other optional clauses for changing solver parameters from preset values and for increasing or decreasing output from the solver, but are not shown here. Refer [5]. In the example FIND above, the unknowns list is shown as a list of scalars, but it, as well as the others, may be comprised of scalars, vectors, and matrices. The LOWER, UPPER, HOLDING, and MATCHING clauses are optional, and MAXIMIZE can be replaced with MINIMIZE and sometimes with EXTREMIZE.

As an example, consider the following problem from Bracken and McCormick [6].

minimize $f(x_1,x_2) = (x_1-2)^{**}2 + (x_2-1)^{**}2$
subject to $h = x_1 - 2*x_2 + 1 = 0$
and $g = -x_1^{**}2/4 - x_2^{**}2 + 1 \ge 0$

In PROSE, this problem might look like the following.

```
PROBLEM .BRACKEN
  X = .DATA(2,2)
  FIND X IN .EQUATIONS BY JOVE
    MATCHING H
    HOLDING G
    TO MINIMIZE F
END
MODEL .EQUATIONS
  H = X(1) - 2*X(2) + 1
  G = -X(1)**2/4 - X(2)**2 + 1
  F = ( X(1)-2 )**2 + ( X(2)-1 )**2
END
```

This is all the programming that is required for this problem. The user should supply initial estimates for the unknowns, must supply a FIND statement describing the problem, and must supply at least an objective function in a MODEL block. From this small amount of information, the PROSE executive knows what type of a mathematical programming problem is given and generates, loads, and executes the code necessary to solve the problem. All NLP and LP solvers have preformatted execution summaries, so that no output statements are required.

In the example above, execution of the FIND statement turns control over, so to speak, to the solver named JOVE, a SUMT algorithm from Fiacco and McCormick [7]. It is at this point that execution of MODEL .EQUATIONS begins. As H, G, and F are computed, the required derivatives will be computed also -- automatically, efficiently, and exactly. The MODEL .EQUATIONS block is executed iteratively until convergence or until the maximum number of iterations (preset) is reached. If the solver JOVE should have trouble or if the user is curious about how other solvers perform, then JOVE can be changed to ZEUS, or to THOR, or to whatever solver is applicable. Actually, one can use N different FIND statements, each referencing a different solver, all in the same program. This feature allows one to quickly validate answers by comparing the results from several solvers, all in the same run. If one wishes, however, to use his or her own algorithm and it requires derivatives, then the statements mentioned earlier can be used. The derivatives obtained in this manner will be obtained efficiently and exactly also.

Thames and Robinson, having automated NLP solvers and derivative evaluation, did not stop here however; they also designed the internal workings of this new language to allow for the nesting of NLP solvers inside NLP solvers to any depth -- a powerful feature requiring the application and management of the chain rule and implicit function theorem by the PROSE executive system. Actually, this nesting capability was extended even further to include solvers for implicit algebraic and ordinary differential equations.

IV Hardware

Even though efficient when compared to differencing, Wengert's method is still done in PROSE in software and thus could be speeded up considerably if done in hardware. The idea of computing the numerical value of a derivative, or integral for that matter, of a given scalar or vector function in a computer's hardware has not occurred to most computer designers yet. Large scientific computers of the future will probably all have this capability; in fact, this will very likely become part of the definition of a scientific computer in the years to come. It may be some time before integrals are computed this way, perhaps, simply because we do not see (very often) large arrays of integrals like the large arrays of derivatives seen in nonlinear programming. As for derivatives however, the time is not far away.

On the drawing board for several years now has been just such a computer. It is being designed to execute PROSE, FORTRAN 77, Pascal, etc. directly. Thus for each language, there is a parser and a hardware/microcode interpreter. The new mainframe is a vector, or more generally, a list processor with variable-length data structures, i.e. variable-length words. Pipelining has been used in the control and arithmetic-logic units, and parallelism at

a function level is inherent to the overall design. The largest model will have a central memory of 4 million bytes, i.e. 500,000 64-bit words, and an I/O buffer memory of 4 million bytes also. Of more interest to NLPers, however, is the incorporation of Wengert's method, as much as is possible, into the machine's hardware. Computing gradients, Jacobians, and Hessians in this manner should reduce current execution times on NLP problems significantly. The designers have estimated that execution times on medium to large NLP problems will be reduced by factors exceeding 100. The greater the number of derivatives the greater the reduction.

V Conclusions

The programming environment for the study and development of mathematical software for nonlinear programming has been greatly improved. Statements for computing gradients and Hessians allows the NLP researcher to code and test algorithms much more quickly. Multiple FIND statements in the same program, each referencing a different NLP solver in the library, allows one to benchmark new algorithms against standard solvers and very quickly obtain comparative statistics. A programming language, designed with nonlinear programming in mind, will make the computational activities of researchers and problem solvers a far more productive endeavor.

Automatic derivative evaluation has led to the automation of mathematical software for nonlinear programming. No longer is it necessary for the NLP analyst to collect from a few sources or design, code, and test the several subroutines typically required for each NLP code. Calculus-level programming languages will supply several NLP solvers in their libraries. Resulting from this automation will be the widespread usage of nonlinear programming by many who are only slightly acquainted with the subject.

Fast derivative evaluation in a computer's hardware (mostly) will allow for the economical use of first- and second-order techniques on medium and large NLP problems. As a result, many such problems, heretofore regarded as too costly or too ambitious, will be attempted with far more confidence and success. ADE in hardware will have as much impact upon applications involving nonlinear programming as the fast Fourier transform has been having in areas where it's been applied.

Just what will be the impact of the capability to nest NLP algorithms inside one another is hard to foresee; this is an applications area yet to be explored. This type of problem requires that an NLP problem be solved completely at each iteration of another NLP problem. This nesting can be continued to any level. These types of problems are not, at present, encountered very often, partly because many, outside the MP field, have no experience at identifying this more difficult type of problem when it arises. Even if identified, such problems would be very difficult to solve using FORTRAN-like languages.

Nonlinear programming, however, will not be the only branch of numerical mathematics to benefit from the automation of NLP solvers and efficient derivative evaluation in hardware. Solvers from nonlinear programming have been used to solve systems of nonlinear implicit algebraic equations for many years, but because of the need for derivatives, the usage of these

solvers has not been as great as it could have been. Thus where nonlinear implicit algebraic equations arise in ordinary and partial differential equations, the automatic, efficient, and exact evaluation of derivatives of known functions will someday play an important role.

One last thought. Evaluating derivatives numerically has been an obstacle for several centuries. Undoubtedly, many thousands of manhours have been devoted to the design of algorithms which have attempted to improve matters somewhat. Even today, NLP researchers continue to develop new or modified NLP algorithms -- new or modified because of the need for derivatives. With automatic derivative evaluation now in software and soon in hardware, with this obstacle about to disappear forever, some researchers might want to re-evaluate the worthwhileness of developing algorithms which avoid the need to compute derivatives exactly. NLPers might like to give some thought to the future of quasi-Newton and derivative-free methods in view of advances now taking place.

TRW - DSSG
One Space Park
Redondo Beach, CA 90278

References.

1. Wengert, R. E. A Simple Automatic Derivative Evaluation Program. Comm. ACM (August 1964).

2. Wilkins, R. D. Investigation of a New Analytical Method for Numerical Derivative Evaluation. Comm. ACM (August 1964).

3. Bellman, R. E., Kagiwada, H., and Kalaba, R. E. Wengert's Numerical Method for Partial Derivatives, Orbit Determination and Quasilinearization. Comm. ACM (April 1965).

4. McDonough, J. M. A Comparison of Three Methods for Evaluating Derivatives by Digital Computer. (Unpublished) (July 1974).

5. PROSE Calculus Operations Manual.

6. Bracken, J. and McCormick, G. P. Selected Applications of Nonlinear Programming J. Wiley & Sons (1968).

7. Fiacco, A. V. and McCormick, G. P. Nonlinear Programming: Sequential Unconstrained Minimization Techniques J. Wiley & Sons (1968).