# On Developing Mathematical Software

Fred T. Krogh

*Math à la Carte, Inc., P.O. Box 616, Tujunga, CA 91043, USA*

**Abstract**

This paper is primarily a list of things we have found useful in developing mathematical software.

*Key words:* Mathematical Software

## 1 Introduction

I have written mathematical software in a number of areas, frequently managing to improve significantly on the performance of what was available at the time. The appendix to this paper cites some examples of this work for those who are interested.

## 2 The List

Here is the list of things I have to communicate. I hope at least one of them will suggest something to you. Incidentally, I am a fan of making lists. Making a list of possibilities means you may think of something that you would overlook otherwise. Strive for thinking of many more possibilities than you could possibly examine in detail.

(1) Provide a means to see in detail what your algorithm is doing. This should be done early to help with algorithm development and testing, and for later support when users may be encountering difficulties. If done well, the algorithm may improve with time as a result of what can be learned.
(2) Support the debugging of user code. This is not important to the performance of your code, but is extremely important for maintenance if the interface to your code is at all complicated. If your code involves communicating data between your code and the user's code, make it easy to

get well formatted output of what the user inputs to your code and what your code provides back to the user. If you have done your job well, this will greatly speed up the process of helping users to find where they have gone amiss, and perhaps sometimes even show where you have blundered or where you might improve things.

(3) When you provide an error message, dump all information that is potentially of use, and format it so that it is easy to read. It will save a lot of time later. Also, make an effort to catch all the errors you can. The brief examples that follow are meant to encourage you to think of possible problems that might occur in the user input, other than the usual testing for a negative number of equations, etc.

Our variable order Adams code, diva, has caught errors such as asking for accuracy on the order of $10^{-10}$ for problems scaled near 1 when using double precision but having cancellation in the computation that limits the accuracy attainable to only $10^{-8}$. Our quadrature code, dint, will not only flag non-integrable singularities, but tell the user to 15 significant digits just where the singularity is. Our zero finding code, dzero, informs the user if the sign change found for a function occurs at the point of a jump discontinuity. All of these diagnostics have found errors that users might not have been aware of and which other codes might have missed or flagged in a way that made them more difficult for the user to understand.

(4) Details matter. Sometimes it has to do with using analysis to remove the loss of digits from the difference of two nearly equal numbers, sometimes it has to do with code organization An example of the latter from recent work I am doing on optimization is of the second type. There was a loop which I needed to leave and then continue from where I left off at the top of the loop. This was written without using the Fortran 'DO' since one is not allowed to enter a loop in this way, and the loop contained a lot of code where it seemed unlikely to matter. But, using 'do j=j, ...', and coming back to this statement, had a very noticeable effect on performance. There are countless things like this to concern yourself with. It used to be that arithmetic was more expensive than logic, now you should really make an effort to keep 'if' tests out of loops, reduce the number of divides to the bare minimum, etc.

(5) Do initial development using test cases that you understand completely. Imagine you were computing things by hand. Is your algorithm doing the smartest thing you can imagine it doing. If not, fix it and repeat. And when you fix things, be clear that the fix is for some general type problem that the current situation reveals; there is no glory in doing well on a small subset of test cases. Using $y' = -y$ with an absolute error test is one I have found extremely useful in developing ODE software. Using lots of test cases and twiddling the knobs on various parameters will help you get better results on those test cases, but will teach you or your algorithm little if you have nothing in mind other than tuning for these test cases.

(6) Related to the above, verify that the results from your test cases are in fact the results you expect. This was added as I discovered that the gradient values in the solution to a quadratic programming problem which should have been distinctly nonzero for the active constraints were very nearly zero. (In this case indicating a problem in the test problem generation.)

(7) Use your mistakes. For example (just happened to me) suppose you provide bad input to your program through some kind of error and the program doesn't handle it correctly. Don't correct the input until you have corrected the problem it illustrates.

(8) Make sure the units are consistent. Thus if you are choosing a starting stepsize for an ODE solver, a change in units in the independent and dependent variables with an equivalent change in the error tolerance, should result in an equivalent initial stepsize.

(9) Test the extremes. There is no reason your code should not do something reasonable with an error tolerance of 100 or of $10^{-30}$ on a problem scaled near 1. That something reasonable in such extreme cases might be to output an error message, but the algorithm also should be tested at the boundaries of where you feel it should be useful. Much is to be learned in the extreme cases. This should also include testing invalid inputs, and valid ones which "no one will ever use".

(10) The algorithm matters. Actually it matters a lot, and I don't have a lot useful to say on this subject. I'm among those rare individuals who would rather cross the ocean on a plane that has had a lot of flight testing and some time in service, than on one that has been *proved* to be sound, but has never flown before. Theory can be useful, but frequently insisting that one can prove something about a code will slow it down with no noticeable improvement in reliability. Users will be more interested in how the code performs on their problem than in the theorems that can be proved about it.

(11) When the algorithm doesn't work as well as desired, stop and think about what might be done. Most people working on BDF codes accept the poor performance it gives for eigenvalues near the imaginary axis. Clearly such codes would work better if the order were restricted for such problems. Thus it is reasonable to ask if we have data that will help to identify such problems. A very little thought suggests that if $\delta\mathbf{y}^T\delta\mathbf{f}/(||\delta\mathbf{y}||\ ||\delta\mathbf{f}||)$ is close to 0 we are likely to have this problem. Here $\delta\mathbf{y}$ and $\delta\mathbf{f}$ are the corrections to the solution and the derivative. Results obtained by Kris Stewart (while working with me) have demonstrated that significant gains in performance are possible using this simple idea.

(12) Be willing to toss out the approach you are using. I spent a fair amount of time developing a quadrature package using Gregory's rule. (A natural for someone used to working with Adams' methods.) This software had reached a point where it was noticeably better than anything I knew of to compare it with, when I encountered Patterson's paper [1]. Luckily, I

was a referee, and felt an obligation to do some testing. This algorithm performs incredibly well, except it did not do all that well if there are singularities or even very rapid changes inside the interval of integration. Clearly this was an algorithm that 'mattered'. It was time to start over with the goal of designing something that helped this basic algorithm around the problems where the algorithm was weak. No amount of cleverness with Gregory's formula is likely to match what can be done with Patterson's formulas as a base. Coming back to the present, I have been amazed at the large number of incredibly good ideas I have needed to discard while doing my current optimization work.

(13) And for consistency, don't give up too soon. In my optimization work I started out with a bias that automatic scaling should be a good thing if done right. But working with well scaled test cases it seemed to be doing more harm than good. Finally, I saw how to do it right. It was more work that I expected to need, but scaling finally made an improvement.

(14) Your subconscious is a powerful tool; learn to use it. (I wish this had been part of my early education.)

(15) Test assumptions. In my optimization work, I wanted to do a full diagonalization of the active constraints for technical reasons associated with mixed integer optimization. A previous code of mine just triangularized the active constraints. I feared that the full diagonalization would be less accurate. Such does not appear to be the case.[1] Some ODE codes hold the order constant after a stepsize change as there are theoretical results suggesting this is a good idea. Testing this "assumption" would result in removing this constraint.

When writing the Runge Kutta code described in [2], the project paying for the development wanted an easy way to exit the code, use it on some different problems, and then come back to continuing a previous integration. This led to saving all the internal variables in space that was passed into the subroutine. My assumption was that a noticeable price would be paid in additional overhead by doing this. The results in [2] indicate that this is probably not the case. This approach carries with it the advantages of thread-safety and allowing for re-entrant code. This is an example of a case where an assumption of mine would probably not have been tested without outside influence.

(16) Define how every variable is used and document anything that is a bit mysterious. It won't make your code run any faster, but it will help when you come back to it in 10 or 20 years. Incidentally, good code (and some bad ones too) have an extremely long lifetime. Computers come and go; keep in mind that if you do your code right it may well outlive you.

(17) Write your user documentation in rough form prior to writing the code. This will give you a base for understanding where you are headed. Update

[1] Added in proof: This code has reverted back to using an LU factorization. I'm not sure it was necessary, but it does run a bit faster.

this documentation as the code evolves. An example of such documentation can be seen at [3].

(18) Be aware of what should be expected of a code. Years ago I talked with someone who was very proud of code they had developed for ODE's. I asked to see some results, and it happened that there was some overlap with mine. His code was working well over twice as hard as mine, and probably 1.5 times harder than other codes available. He was so convinced of what his theory had to say that he was out of touch with what could be done.

(19) If you pay too much attention to what others are doing, you are not likely to have any kind of breakthrough; if you don't pay enough attention, your work will suffer for it. And perhaps it should be added, if you aren't at least hoping for a significant improvement (capabilities, robustness, efficiency, or . . . ) on what has been done before, perhaps a different project would be appropriate.

(20) "A poem is never finished, only abandoned", Paul Valery (1871 – 1945). The same certainly applies to any large piece of software. Thus one must exercise some judgment as to when the time has come to give up on looking for further improvements.

And perhaps most important is to see setbacks as learning opportunities, and as the stimulus necessary for the generation of new ideas.

## A Some Examples

See [4] for results comparing, DVDQ, an early variable order Adams code. Although this paper certainly helped my reputation, I can still remember Tom Hull being amazed that I felt the paper did not really do justice to all that DVDQ had to offer. People may find this strange now, but I can remember being unable to convince someone that a variable order method starting with order 1 could work. This was prior to writing such a code, and this person was convinced that it was all about order. And I could also mention a paper being refused for publication because it used the difference form of the Adams method, and "such methods clearly would suffer from severe cancellation". (Of course, you all know that no bits are lost when subtracting nearly equal number even though the result has fewer significant digits. And of course the precision loss does not matter when adding the result back into a larger number.) One lesson here is that if you think you are on the right track don't let others discourage you. Another is that one bad referee and editor should not convince you that publishing is really not worth the bother.

Some of the ideas in the quadrature code based on Patterson's formulas mentioned earlier can be found in [5]. This code while being significantly more reliable than the best codes I know of[2], also requires about half the function evaluations on average as the best alternatives. This is an example where the right algorithm makes a big difference.

Reference [6] compares a number of algorithms for finding a zero of a continuous function. Dr. Shi has kindly used the same test program, ENCL0FX, on DZERO (a code of mine). With Dr. Shi's permission, results from Table II of that paper are given below with an additional column 'DZ' added for the results he obtained with DZERO. With the exception of DE and M, all codes solved all of the problems. See [6] for more details. Note that one of the reasons DZERO performs so well is that there was no attempt at proving theorems about guaranteeing a minimal rate of convergence when writing the code.

Total Number of Function Evaluations in
Solving All the Problems Listed in Table I of [6]

| *tol* | BR | DE | M | R | LE | 2.1 | 2.2 | 2.3 | 2.4 | 2.5 | 4.1 | 4.2 | DZ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $10^{-7}$ | 2804 | 2808 | 2839 | 7630 | 2694 | 3154 | 2950 | 2645 | 2791 | 2687 | 2696 | 2650 | 2100 |
| $10^{-10}$ | 2905 | 2963 | 2992 | 7768 | 2821 | 3338 | 3060 | 2789 | 2922 | 2819 | 2835 | 2786 | 2177 |
| $10^{-15}$ | 2975 | 3196 | 3261 | 8014 | 3061 | 3448 | 3151 | 2948 | 3015 | 2914 | 2908 | 2859 | 2236 |
| 0 | 3008 | 2998 | 3146 | 8230 | 3165 | 3509 | 3219 | 3029 | 3060 | 2954 | 2950 | 2884 | 2255 |

Although I don't have examples readily available comparing it with other codes, the code described in [7] for nonlinear least squares has been found very useful at the Jet Propulsion Laboratory on problems where function evaluations are very expensive. The reliability of the code in getting answers is the primary reason for selecting this code in place of alternatives. Of most interest, this code does not insist on reducing the objective on every iteration, and in fact has no hard bound on how many iterations there might be without a decrease. Of course, there are things in the algorithm that keep the code from wandering off to infinity, but one of the key tenets frequently cited to make such a code reliable is to insure that the objective decrease at each step. In fact such an insistence will insure the code will be much less efficient than necessary on many important problems. And, it does nothing to improve the reliability of a well crafted code.

In the table below are some recent results from my optimization work. NV, NE, NB, NBA, NI, NIA, and M, give respectively the number of variables, equalities, bounds, active bounds at the solution, inequalities, active inequalities at the solution, and the number or rows in the least squares problem. LP,

[2] Actually, those based on using Taylor's series combined with interval arithmetic are more reliable.

QP and LS are abbreviations for Linear Programming, Quadratic Programming, and constrained Least Squares. The quadratic programming problems are formed using the equivalent normal equations to the least squares problem. The column headed by # gives the number of different (random) problems that were solved with the given characteristics. Times are given in seconds and result from running on an 1.2 GHz Athlon system running Linux and compiled with g77 (GNU Fortran (GCC 3.1.1) 3.1.1 20020725). The errors reported are the $L_2$ norm of the difference between the solution obtained and that expected by the test driver. Some of the error is probably attributable to the fact that the test driver does not generate "exactly" the problem corresponding to the expected solution. Dopt is our current code, and lssol, is version 1.03 of this code by P.E. Gill, S. J. Hammarling, W. Murray, M.A. Saunders and M.H. Wright, with a date of June 19, 1989.

| | | | | | | | | | dopt | | lssol | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NV | NE | NB | NBA | NI | NIA | M | | # | Error | Secs. | Error | Secs. |
| 60 | 20 | 40 | 20 | 40 | 20 | 0 | LP | 200 | 1.1E−12 | 8.2 | 3.9E−13 | 17.4 |
| 100 | 20 | 100 | 80 | 0 | 0 | 0 | LP | 200 | 2.2E−12 | 5.3 | 1.9E−11 | 12.2 |
| 500 | 200 | 500 | 300 | 0 | 0 | 0 | LP | 1 | 3.1E−13 | 11.8 | 3.2E−14 | 54.4 |
| 240 | 80 | 240 | 80 | 160 | 80 | 0 | LP | 2 | 1.0E−13 | 13.6 | 4.3E−14 | 29.6 |
| 60 | 20 | 40 | 20 | 40 | 20 | 0 | LP | 200 | 1.6E−12 | 9.1 | 3.5E−13 | 26.5 |
| 100 | 20 | 100 | 80 | 0 | 0 | 0 | LP | 200 | 1.0E−12 | 6.6 | 1.6E−12 | 14.4 |
| 500 | 200 | 500 | 300 | 0 | 0 | 0 | LP | 1 | 1.6E−14 | 13.8 | 1.3E−14 | 58.6 |
| 240 | 80 | 240 | 80 | 160 | 80 | 0 | LP | 2 | 4.0E−14 | 12.1 | 5.1E−14 | 62.9 |
| 300 | 0 | 300 | 200 | 0 | 0 | 400 | LS | 10 | 2.4E−15 | 42.6 | 2.7E−15 | 57.7 |
| 300 | 0 | 300 | 200 | 0 | 0 | 400 | QP | 10 | 2.2E−15 | 13.5 | 3.4E−15 | 18.5 |
| 300 | 0 | 300 | 50 | 0 | 0 | 400 | LS | 10 | 4.6E−15 | 37.5 | 5.8E−15 | 57.9 |
| 300 | 0 | 300 | 50 | 0 | 0 | 400 | QP | 10 | 5.3E−15 | 8.5 | 1.2E−14 | 19.4 |
| 200 | 0 | 160 | 50 | 160 | 50 | 300 | LS | 10 | 9.9E−15 | 40.2 | 7.6E−15 | 116.9 |
| 200 | 0 | 160 | 50 | 160 | 50 | 400 | QP | 10 | 1.9E−14 | 32.4 | 9.9E−15 | 108.4 |
| 200 | 40 | 60 | 30 | 20 | 10 | 200 | LS | 10 | 1.1E−14 | 10.4 | 5.9E−15 | 23.2 |
| 200 | 40 | 60 | 30 | 20 | 10 | 400 | QP | 10 | 1.3E−14 | 6.9 | 6.4E−15 | 20.0 |

All test cases were generated using uniform random numbers, except for the second set of 4 for the LP case which were generated using Gaussian random numbers.

The code dopt is under active development. There is much left to be done, see [3]. Even though the results above would indicate the code solves QP problems there is still some work required in this area for semidefinite and indefinite (local minima only) problems.

The gains in efficiency in the linear programming case come primarily from allowing moves in a subspace, rather than moving from one vertex to another as is done in the simplex method. (One might wonder why this was not done in the first place.) This gain is more pronounced as the problem size increases, as one might expect. In the quadratic programming and constrained least squares cases, the gains are due to better algorithms for getting feasible (from the LP code) and, probably of less significance, care in how the loops involving Givens rotations are coded and perhaps in the way in which constraints are selected for removal from the active set.

As an example of the kind of thinking that leads to algorithmic improvements I'd like to describe one more thought from the optimization work. If no provision is made to avoid it, active set methods will on some problems cycle indefinitely. Of course, we have an algorithm that avoids this problem. But even though cycling is prevented, the code may still take a number of iterations prior to getting an improvement in the objective. Although what we have done is good enough, is there some possibility for improvement in this area? I know too well the tendency to say that good enough is, well, good enough. At this point I think there is a possibility for improvement in this area, but am pointing this out here to suggest that fighting the tendency to accept "good enough" will provide rewards in the long run. Whether it will in this case or just be a waste of time is open to question.

The codes mentioned in this appendix are available from `http://mathalacarte.com`, except for nonlinear least squares code and the optimization code which requires a bit more work prior to being made available. (This code will be released in pieces as its full functionality will take a significant amount of time to complete.)

One lesson that I hope will be taken from these results is that even in well worked areas of computational mathematics there are still opportunities for significant advancements.

## References

[1] T.N.L. Patterson, The Optimum Addition of Points to Quadrature Formulae, *Math. of Comp.* **22**, 847–856(1968).

[2] F. T. Krogh, An Adams Guy Does the Runge-Kutta, available from `http:`

//mathalacarte.com/fkrogh.

[3] F.T. Krogh, Preliminary documentation for "9.4 Mixed Integer Linear Programming, Constrained $L_1$, $L_2$, $L_\infty$ Fits, ..., Semidefinite Programming", available from a link at http://mathalacarte.com/fkrogh.

[4] T.E. Hull, W.H. Enright, B.M. Fellen, and A.E. Sedgwick, Comparing Numerical Methods for Ordinary Differential Equations, *SIAM J. Numerical Analysis* **9** 603–637(1972).

[5] W.V. Snyder and F.T.Krogh, "In Preparation". This will be available at http://mathalacarte.com/fkrogh soon.

[6] G.E.Alefeld, F.A. Potra, and Y. Shi, Algorithm 748: Enclosing Zeros of Continuous Functions, *ACM Trans. on Math. Software* **21** 327–344(1995).

[7] R.J. Hanson and F.T. Krogh, A Quadratic-Tensor Model Algorithm for Nonlinear Least-Squares Problems with Linear Constraint, *ACM Trans. on Math. Software* **18** 115–133(1992)