NUMERICAL OPTIMIZERS FOR USE IN FLOATING-POINT DATA GENERATION

by

Barbara P. Havens
Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309

CU-CS-197-80 October, 1980

This material is based upon work supported by the National Science Foundation under Grant Nos. MCS78-12288 and MCS8000017 also Department of Energy Grant No. DE-AC02-80ER10718.

Abstract

Automatic floating-point data generation is the problem of finding a set of floating-point input data to a subprogram which forces control down a specified path through that subprogram. Although this problem is not, in general, solvable, several methods do exist which are successful in some instances of the problem. One such method involves the use of multivariate numerical optimizers, and the choice of optimizer often has dramatic effect on the success of the method.

An experiment, which investigates the comparative suitability of several numerical optimizers to this purpose, is described, and recommendations concerning an appropriate optimizer are made.

I. Introduction

The term "floating point data generation," as used in this paper, refers to the following problem: given a subprogram with floating-point input vector $\bar{x}$ (and all integer inputs fixed), and a path through that subprogram (i.e., given the sequence of truth values for logical expressions that correspond to a path), perform the following: a) decide whether floating point data $\bar{x}_0$ exists that will drive control down the specified path, and b) if it does exist, exhibit such an $\bar{x}_0$.

There are several reasons to be interested in floating-point data generation. One is that such a generator could be used as an aid to the powerful validation and verification tool DAVE ([7]). DAVE scans FORTRAN programs for data-flow anomalies such as references to uninitialized variables. If DAVE finds a program path on which such a reference occurs, it reports that path to the user; however, DAVE does not attempt to determine whether actual data exists that would cause that path to be executed. A floating-point data generator could do that, if the integer inputs are fixed (perhaps by a query of the user for typical values).

Another use for such a generator would be to help answer questions about an algorithm. The questions might be of the form, "Could the following sequence of instructions ever be executed?" The sequence would then provide the path information required by the generator.

Finally, someone testing code might wish to exercise a particular subset of the instructions. If he or she supplies a path description that sends control through that subset, the generator could attempt to find proper test data.

It should be noted that this problem, as stated, is unsolvable, since a sub-case can be formulated as the halting problem (see [1]). However, methods do exist that are successful on some instances of the problem. One such method uses numerical optimizers in its approach. There are many different types of numerical optimizers in existence; the purpose of this paper is to compare the effectiveness of several of them in this application.

II. Overview of Numerical Optimization Method of Generating Data

Our method for approaching data generation, first suggested in [11], involves transforming the user's subprogram so that it returns a vector each time it is called with a trial input vector. The numerical components of this vector are derived from the conditional parts of statements whose desired truth values have been specified, and they are used to help determine how close the trial input vector is to a satisfactory data set. A component will be positive if and only if the trial input vector causes the truth value of the expression to be as specified.

For example, suppose that the i^{th} input-dependent conditional statement in a given subprogram is

```
IF(X.GT.Y) Y = Z.
```

If the chosen path requires that the logical expression be .TRUE., this line would be replaced with

```
C(I) = X - Y
Y = Z.
```

If it is to be .FALSE., the replacement would simply be

```
C(I) = Y - X,
```

since the assignment only takes place if the expression is .TRUE. Other relational operators are handled similarly (with the exception of .EQ. and .NE., which can be implemented with the absolute value function). A subprogram with constraints inserted as described will be referred to herein as a "subprogram path."

The vector $\overline{c}$ of constraint values is then summarized in a single floating-point number, computed by one of the following three objective functions.

$$F_1(\bar{c}) = \begin{cases} \sum_{i \ni c_1 < 0} c_i, & \text{if } \exists\ i \ni c_i < 0 \\ \min_i \{c_i\}, & \text{otherwise} \end{cases}$$

$$F_2(\bar{c}) = \begin{cases} -\sqrt{\sum_{i \ni c_i < 0} c_i^2}, & \text{if } \exists\ i \ni c_1 < 0 \\ \min_i \{c_i\}, & \text{otherwise} \end{cases}$$

$$F_\infty(\bar{c}) = \min_i \{c_i\}$$

Note that all three of these above objective functions yield positive values if and only if all constraints are positive, i.e., satisfied. (Any function with this property could be used as an objective function; these are only examples.)

Finally, a multivariate numerical optimizer is used to try to find a positive value for the objective function. After each evaluation of the constraint vector, the optimizer decides either what input vector to try next, or it decides to quit (either because it was successful or because it isn't making adequate progress).

III. The Optimizers Studied Here

There are many types of numerical optimizers, most of which involve the evaluation or approximation of at least one partial derivative of the objective function. These optimizers are typically far more efficient, on smooth functions, than optimizers which use no derivatives. We tried one method, developed by Schnabel ([13]), which used numerical approximations to partial derivatives, but it performed rather poorly on our examples. Our work was not conclusive, but we did not see a way to make our objective functions smooth enough for such techniques. Therefore, we limited our study to direct-search methods, i.e., those that make no use of differentiation.

The four optimization methods we tried were that of Hooke and Jeeves ([9]), that of Rosenbrock ([8]), that of Davies, Swann and Campey ([2]) and the Simplex method ([10]). (Actually, the experiment

started with nine optimizers, all variants of those four, and the best of each type was used.) Each of these optimizers works in R^n, where n is the number of inputs (the number of elements in the input vector $\bar{x}$). All of the optimizers stop if they find a positive value for the objective function, since this indicates success (proper data were found), or if they converge (a maximum of the function was found), or if they simply spend too much time without finding proper data.

It should be noted that each optimizer has several user-set parameters, and changes in their values often has large effect on the performance of the optimizer on a particular problem. For this reason, the parameters of all the optimizers were all 'tuned' on the same problem (a simple geometrical problem, not one of those used in the testing) and then their parameter values remained fixed throughout the experiment.

In the following paragraphs, the term "better point" means "input vector which yields a higher objective function value," and will be used in describing how the individual optimizers work. The descriptions here are intended to give the reader only an intuitive idea of how the optimizers are similar and how they differ (enough to understand what is being compared, but not enough to code such optimizers). The interested reader can obtain a tape of the actual FORTRAN source code for the optimizers, drivers and subprogram paths from Professor Webb Miller, Mathematics Department, University of California, Santa Barbara, California 93107.

The Hooke and Jeeves method is, perhaps, the simplest of the four. After choosing n orthogonal search directions (usually the coordinate axes) and an initial step size, steps are tried along the chosen directions ("Exploratory" moves), keeping the best point found while searching for a better one. When all of the directions have been tried, an additional step is taken in the direction from the first point tried to the best point found (a "Pattern" move), in effect doubling the combined step taken during the Exploratory moves. The Exploratory stage is then repeated, starting at the new point that the pattern move yielded. If no progress is made in the Exploratory stage, the step size is decreased; convergence occurs when the step size falls below a predetermined bound (here, 10^{-4}, with an initial step size of about 30).

The first step in the method of Rosenbrock is similar to that of the method of Hooke and Jeeves, except that there is a step size corresponding to each direction, and the exploratory phase takes longer, with several iterations through the directions and each step size being altered at each iteration. (If a step in a particular direction is successful, the next step in that direction will be a longer one, but if it's unsuccessful, the next step will be shorter and in the opposite direction.) The second phase of Rosenbrock's method is to rotate all of the search directions so that the first direction is that of the total step taken in the first phase (and is hence the most profitable direction, in a sense), and the i-th new direction is the most profitable direction orthogonal to the 1st, ..., i-1st, i = 2, ..., n. The process is repeated through 10 rotations of the search directions.

The Davies, Swann and Campey algorithm also uses a sort of exploratory phase and rotation of search directions, so it is similar to the Rosenbrock method. The line searches used in the exploratory phase are carried out by quadratic interpolation, one interpolation done in each of the n current search directions. Then, one more, similar line search is done (like a Pattern move) in the direction taken in the exploratory phase. Finally, if the total step taken (in the first two stages) is large enough, then the directions which were found profitable are rotated (the others are left alone); otherwise, more searching is done in the same directions as last time. Convergence occurs when a step-size parameter, used for sampling in quadratic interpolations and decreased each time no rotation is done, falls below a certain bound (the parameter starts off at 1000, with a bound of .01).

Finally, the Simplex method is different from the others in that no search directions are required. Instead, a simplex (n+1 points in R^n) is formed and operations are performed on it, the object being to eventually trap the function's maximum somewhere inside the simplex and then to shrink the simplex toward that maximum; convergence occurs when the simplex is sufficiently small (all points are within 10^{-4} of the center; all points started out with random coordinates between -5000 and 5000). The objective function is first evaluated at each of the simplex's vertices; throughout the algorithm these vertices can

be considered to be the best n+1 points (input vectors) found so far. Then, reflections (taking the worst vertex and reflecting it through the hyperplane determined by the other n points), expansion (moving a point farther from the hyperplane determined by the others) and contraction (shrinking the simplex toward its best vertex) are used to replace some or all of the simplex's vertices with better points.

IV. Subprogram Paths Used in Testing the Optimizers

Nine subprogram paths were used for tests for this study; they were given the names "bad," "decomp," "fmin," "good," "house," "linerbad," "linpack," "seval" and "zero." They will be described below.

"Bad" is a subprogram path on which a variable is referenced before it is set, but there is no data which would drive control down this path, as the requirement on the 2-element input vector is that $x_2 < x_1 < x_2 - 1$.

"Decomp" is Moler's matrix decomposition subroutine ([12]), with a path that requires, for input, a non-singular matrix in which, for each column, the pivot element is the last element in the column. Clearly, such matrices exist.

"Fmin" is Forsythe, Malcolm and Moler's one-dimensional function optimizer ([4]), and the chosen path through it is a sequence of four golden-section search steps followed by convergence. The function being optimized is $f(x) = x^2$, and since golden-section search is only performed on poorly-behaved functions (it is the slower of the methods used in fmin), no data exists that will drive control down this path.

"Good" is similar to "bad," except that in this case, satisfactory input data does exist; the only requirement on the 2-element input vector $\bar{x}$ is that $x_2 < x_1 < x_2 + 1$.

"House" is Householder's method on a 4 x 3 input matrix. The chosen path requires that the matrix be constructed in such a way that several intermediate values computed during the reflection process have positive signs; this property has algorithmic significance in that it reduces the error introduced by the computations. Many suitable input sets exist.

"Linerbad" is similar to "good" and "bad," but in this case, though no satisfactory data exists, it is possible to get arbitrarily close to satisfactory data, as the requirement on the 2-element input vector is that $x_2 < x_1 < x_2$.

"Linpack" is part of routine SGEDI from LINPACK ([3]). The chosen path requires the input vector to have a small element ($\approx 10^{-2}$), followed by a large element ($\approx 10^{2}$), followed by another small one and another large one.

"Seval" is from Forsythe, Malcolm and Moler ([5]). The chosen path requires a certain sequence of comparison outcomes in a binary search. Data sets do exist that will drive control down this path

Finally, "zero" is ZEROIN from Forsythe, Malcolm and Moler ([6]) and it is analogous to the "fmin" data set, but this time, bisection is the slow step and the function is $f(x) = x$. No satisfactory data exists for this path.

These subprogram paths were chosen because they were thought to be good approximations to ones which might actually be used in the applications listed in the introduction.

V. Testing Procedure and Evaluation Criteria

There are four numerical optimizers, three sample objective functions and nine subprogram paths described in this paper; every combination of numerical optimizer, objective function and subprogram path was tested. Each test consisted of starting the optimizer 100 times (at random starting points) and recording, each time, whether the search was successful and how many evaluations of the objective function were required. Success rate and speed (average number of function evaluations) will be the criteria for evaluation.

VI. Results

Table 1 shows the success rates and Table 2 shows the average speeds for each of the optimizer-objective function-subprogram path combinations. The numbers in Table 1 show how many times a satisfactory data set was found, out of 100 tries (100 random starting points). Note that, overall, objective function F_2 gave the best results. Note also

that no one optimizer stands out as the best in all cases; for example, using objective function F_2, the Hooke and Jeeves optimizer was by far the most successful in finding suitable input data for the "linpack" subprogram path (the Simplex method was the only other one that had any success), but the other three optimizers worked better on the "seval" subprogram path. This suggests that, perhaps, some sort of hybrid optimizer might prove most successful.

Table 2 shows average speed, in terms of the number of objective function evaluations, for each optimizer-objective function-subprogram path combination; averages were tabulated over 100 restarts of the optimizer. Objective functions F_1 and F_2 were usually slightly faster than F_∞, with F_1 slightly faster than F_2. Again, no one optimizer stands out as the best in all cases, but the Hooke and Jeeves optimizer does appear to be the slowest. Although the Simplex method is faster on several subprogram paths for which the data searches are strictly successful ("decomp," "house" and "seval," with F_1 or F_2), the Rosenbrock optimizer is the fastest when both successful and unsuccessful searches are considered. Again, a hybrid optimizer is suggested.

Perhaps a reasonable compromise would be the following technique: use the Simplex method for a certain number of objective function evaluations (perhaps computed from the size of the input vector), and then let the best point found be the starting point for the Rosenbrock method. (Of course, either method would halt if the objective function yields a positive value for some trial input vector.) This would exploit the speeds of both optimizers while assuring a good starting point for the Rosenbrock method.

SUCCESS CHART

Objective function F_1

	BAD*	DECOMP	FMIN*	GOOD	HOUSE	LINERBAD*	LINPACK	SEVAL	ZERO*
Hooke & Jeeves	0	99	0	98	97	0	66	3	0
Rosenbrock	0	99	0	62	100	0	0	100	0
Davies, Swann & Campey	0	88	0	100	76	0	0	84	0
Simplex	0	100	0	100	100	0	2	100	0

Objective function F_2

	BAD*	DECOMP	FMIN*	GOOD	HOUSE	LINERBAD*	LINPACK	SEVAL	ZERO*
Hooke & Jeeves	0	99	0	98	99	0	89	3	0
Rosenbrock	0	100	0	62	100	0	0	100	0
Davies, Swann & Campey	0	93	0	100	84	0	0	100	0
Simplex	0	100	0	100	100	0	11	100	0

Objective function F_∞

	BAD*	DECOMP	FMIN*	GOOD	HOUSE	LINERBAD*	LINPACK	SEVAL	ZERO*
Hooke & Jeeves	0	78	0	98	94	0	38	0	0
Rosenbrock	0	96	0	62	99	0	0	0	0
Davies, Swann & Campey	0	78	0	100	75	0	0	0	0
Simplex	0	100	0	100	100	0	6	0	0

TABLE 1:

Number of successes out of 100 tries (100 random starting points).

*No satisfactory input data exists for this subprogram path.

SPEED CHART

Objective function F_1

	BAD*	DECOMP	FMIN*	GOOD	HOUSE	LINERBAD*	LINPACK	SEVAL	ZERO*
Hooke & Jeeves	745	372	1176	434	527	957	1565	1692	2398
Rosenbrock	49	56	98	39	70	51	104	177	118
Davies, Swann & Campey	84	123	154	39	264	71	200	314	197
Simplex	2000	18	662	45	18	2000	1052	43	447

Objective function F_2

	BAD*	DECOMP	FMIN*	GOOD	HOUSE	LINERBAD*	LINPACK	SEVAL	ZERO*
Hooke & Jeeves	1158	284	1868	434	556	957	1228	1692	1342
Rosenbrock	50	50	102	39	67	51	104	183	141
Davies, Swann & Campey	68	135	256	39	186	71	350	85	233
Simplex	2000	18	513	45	18	2000	1341	47	368

Objective function F_∞

	BAD*	DECOMP	FMIN*	GOOD	HOUSE	LINERBAD*	LINPACK	SEVAL	ZERO*
Hooke & Jeeves	1162	1210	1585	434	823	957	1561	1394	2552
Rosenbrock	50	78	102	39	108	51	106	620	139
Davies, Swann & Campey	71	210	377	39	295	71	518	999	223
Simplex	2000	19	927	45	18	2000	1432	2000	424

TABLE 2:

Average number of function evaluations required, tabulated over 100 tries, regardless of success or failure.

*No satisfactory input data exists for this subprogram path.

References

[1] Clarke, L. A., "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, Sept., 1976.

[2] Dixon, Nonlinear Optimization, The English Universities Press, Ltd., 1972.

[3] Dongarra, J. J., et. al., Linpack Users' Guide, Society for Industrial and Applied Mathematics, 1979, pp. C.9-C.10.

[4] Forsythe, et. al., Computer Methods for Mathematical Computations, Prentice-Hall, 1977, pp. 182-187.

[5] Forsythe, et. al., Computer Methods for Mathematical Computations, Prentice-Hall, 1977, pp. 76-79.

[6] Forsythe, et. al., Computer Methods for Mathematical Computations, Prentice-Hall, 1977, pp. 161-166.

[7] Fosdick, L. D., and Drey, C. M., "The Dave System User Manual," University of Colorado, Department of Computer Science Technical Report #CU-CS-106-77, March, 1977.

[8] Gill, P. E., and Murray, W., Numerical Methods for Constrained Optimization, Academic Press, 1974, pp. 21-22.

[9] Kowalik, J., and Osborne, M. R., Methods for Unconstrained Optimization Problems, Elsevier, 1968, pp. 21-22.

[10] Kowalik, J., and Osborne, M. R., Methods for Unconstrained Optimization Problems, Elsevier, 1968, pp. 24-26.

[11] Miller, W., and Spooner, D., "Automatic Generation of Floating-Point Test Data," IEEE Transaction on Software Engineering, Sept., 1976.

[12] Moler, C., "Algorithm 423, Linear Equation Solver," Comm. Ass. Comput. Mach., vol. 15, p. 274, Apr. 1972.

[13] Schnabel, Robert B., "Determining Feasibility of a Set of Nonlinear Inequality Constraints," University of Colorado Department of Computer Science Technical Report #CU-CS-172-80, February, 1980.