The Q approach to problem solving

by J. D. McCULLY

TRW Systems
Redondo Beach, California

INTRODUCTION

The problem of determining derivatives on a digital computer has received a great deal of attention for several years. Some exotic systems have been developed and numerous papers have treated the problem. In 1964 it was suggested by Wengert[1] that the chain rule could be applied to values for the determination of derivatives.

This general concept has served as the basis for a series of programs developed at TRW Systems. It has been expanded to permit the essentially simultaneous computation of first and second partial derivatives with respect to several independent variables. Second partials are especially valuable in optimization problems, and excellent results have been obtained with this technique. The first program written at TRW some years ago to apply Wengert's chain rule concept was called ROP (for Restricted Optimization Program) and has been used to optimize sets of algebraic equations. After some experience with this program it was decided that a complete system should be devised to permit wider application of the technique to problems where partial derivatives would be of value. The system was initially named CUE, for Computer Utility for Engineers, but was recently renamed Q in deference to another system named CUE.

The intent was to make Q essentially a computer operating system. On the other hand, it was to be used within an already existing operating system (SCOPE 2.1) on TRW's CDC 6500 machine without modification to the existing system. A good discussion of this type of system is found in Glass.[2] The consequence was necessarily some added overhead operating cost, but it was hoped that two factors would offset this added cost. One of these factors was the planned ma chine-independent characteristic of the Q system which essentially uses only FORTRAN and FORTRAN routines (including I-O). In practice, some of the machine-oriented functions of the SCOPE operating system proved impossible to resist and conversion to another machine may be less easy than was originally planned.

The second factor that would make Q attractive despite the increased machine time was the inclusion of several unique features in the system. The most important of these features is the above mentioned partial derivatives. Another is dynamic storage, and a third feature of interest is a macro processor for the input language. With this feature the system is suitable for use by the engineer who is more or less familiar with FORTRAN and wants his job done quickly even at the expense of some extra machine time.

Sample problems

Before the structure and characteristics of the Q system are described in detail, it may be useful to give some examples of the kind of problem for which it has proved most useful. These examples are taken from INTRODUCTION to SLANG.[3] In general it can be said that Q is suitable for mathematically complex problems. It has been designed to relieve the user of most of the complex calculations involved and to provide him with a short turnaround time that makes practical a series of alternate approaches or formulations.

As an essential part of making Q user-oriented, a high-level language called SLANG has been evolved to allow easy communication with the computer by engineers with little programming knowledge. For purposes of the sample problems it is necessary to keep in mind that the problem statements shown are written in SLANG. The convenience of formulating problems in this way will be apparent.

The first example illustrates the use of SLANG for solving a typical optimization problem with nonlinear implicit equations imbedded in the engineering model. The problem is to minimize the weight of a three-stage liquid rocket vehicle boosting a payload from the surface of Mars. The optimum values of thrust level and burn time for each stage are to be determined for the specified mission. Total burn time, total velocity increment, and payload weight are given. The SLANG statements required to solve this problem are shown in Figure 1.

In this problem, the quantity being minimized is WTØT the statement

ØPTIMIZE WTØT (1)

identifies the payoff function and establishes an optimization loop which ends with the second END LØØP card. The statement

INDEPENDENT THRUST(2), THRUST (3), TBURN(1), TBURN(2) (2)

designates thrust levels of two stages and burn times of two stages as independent variables which are being determined by the optimization. Equations G1 and G2 are being solved to constrain the solution such that total velocity increment and burn time match specified values. The statement

SOLVE G1, G2 (3)

identifies the implicit simultaneous equations being solved and establishes an equation solving loop which ends with the first END LØØP card. The independent variables of the SØLVE loop are identified by the statement.

INDEPENDENT THRUST(1), TBURN(3) (4)

Even though they are expressed in terms of intermediate variables, the equations G1 and G2 are equivalent to the ultimate form

G1 = G1 (THRUST (1), TBURN (3)) (5)

G2 = G2 (THRUST (1), TBURN (3))

```
     VARIABLE ISP(3),ISPVAC(3),TBURN(3),THRUST(3),XIP(3),
*             WPRØP(3),WSTAGE(3),STRFAC(3),DELV(3),MR(3)
   1 READ DATA
     ØPTIMIZE WTØT
        INDEPENDENT THRUST(2),THRUST(3),TBURN(1),TBURN(2)
        ØLIMITS(FPRIN = 0)
        SØLVE G1,G2
           INDEPENDENT THRUST(1),TBURN(3)
           DLVTØT = 0
           W = WPAYLD
           TBTØT = 0
           DØ FØR L = 1 TØ 3
           I = 4-L
              ISP(I) = ISPVAC(I) * (1 - XIP(I))
              WPRØP(I) = THRUST(I) * TBURN(I) / ISP(I)
              WSTAGE(I) = 0.0234 * THRUST(I) + WPRØP(I)
*             + 1.255 * WPRØP(I) ** 0.704 + 4
              STRFAC(I) = WPRØP(I) / WSTAGE(I)
              W = W + WSTAGE(I)
              MR(I) = W / (W - WPRØP(I))
              DELV(I) = GC * ISP(I) * LØGN(MR(I))
              DLVTØT = DLVTØT + DELV(I)
              TBTØT = TBTØT + TBURN(I)
        REPEAT
           G1 = DLVTØT - DELVIP
           G2 = TBTØT - TBTIP
        END LØØP
        WTØT = W
        PRINT VARIABLES
     END LØØP
     GØ TØ 1
     END
      DATA
      THRUST=5400,1237,317,TBURN=142,127,131,GC=32,174,WPAYLD=
      DELVIP=2.8E4,TBTIP=400,ISPVAC=315,315,315,XIP=0,0,5E-3,
      $END
```

Figure 1—SLANG formulation of sample optimization problem

The purpose of the SØLVE loop is to find the values of THRUST (1) and TBURN (3) that satisfy G1 = 0 and G2 = 0. Engine performance and vehicle weight quantities are computed in a loop beginning with the statement.

DØ FØR L = 1 TØ 3 (6)

and ending with

REPEAT (7)

The equations between these two statements are used three times, one time for each of the three stages. Two characteristics of SLANG should be evident from this example. One is that the SLANG expressions used to describe the engineering model closely resemble those of FØRTRAN. The other is that numerical algorithms for optimization and nonlinear equation solving are invoked using the commands ØPTIMIZE and SØLVE.

The total running time for this problem was eight seconds on the CDC 6500. The printout of the solution is shown in Figure 2.

The second example demonstrates how a SØLVE loop can be used to match an integration boundary

```
                                      Variable Values

DELVIP  2.80000E+04  DELV   1.06014E+04          9.30107E+03          8.09754E+03  DLVTOT  2.80000E+04
GC      3.21740E+01  G1     0.           G2      0.           ISPVAC  3.15000E+02          3.15000E+02
        3.15000E+02  ISP    3.15000E+02          3.15000E+02          3.13425E+02  I       1.00000E+00
L       3.00000E+00  MR     2.84635E+00          2.50361E+00          2.23222E+00  STRFAC  8.51072E-01
        7.95069E-01         7.14542E-01  TBTIP   4.00000E+02  TBTOT   4.00000E+02  TBURN   1.51068E+02
        1.33079E+02         1.15853E+02  THRUST  5.11094E+03          1.27746E+03          3.28288E+02
WPAYLD  5.00000E+01  WPROP  2.45111E+03          5.39694E+02          1.21347E+02  WSTAGE  2.88002E+03
        6.78801E+02         1.69824E+02  WTOT    3.77865E+03  W       3.77865E+03  XIP     0.
        0.                  5.00000E-03
```

Figure 2—SLANG printout of results from problem shown in Figure 1

```
/SØLID RØCKET ENGINE START-UP TRANSIENT PRØBLEM
/   THE PURPØSE ØF THIS PRØBLEM IS TØ DETERMINE
/   THE PERCENTAGE ØF EQUILIBRIUM CHAMBER PRESSURE
/   ATTAINED BY AN END BURNING SØLID RØCKET ENGINE
/   AT A SPECIFIED TIME (TSPEC) DURING ITS STARTUP
/   TRANSIENT
/   THE PRØBLEM INVØLVES INTEGRATIØN, BØUNDARY CØNDITIØN
/   MATCHING, AND HAS A SØLVE LØØP
READ DATA
PCEQ = (12/32.174 * RHØP * CSTARO * A * K) ** (1/(1 - N - Q))
     SØLVE CØNST
     INDEPENDENT PCSPEC
              FAC = VC / (GAM ** 2 * AT * 12)
              LET TINTEG = INTEGRAL (1 / (CSTARO * PC **
*                 Q * PC * (RHØP * CSTARO * PC ** Q * A *
*                 K * PC ** (N -1) * 12 / 32.174 - 1)),
*                 PC = PCIG TØ PCSPEC IN 10 STEPS)
              TCØMP = FAC * TINTEG
              CØNST = TCØMP - TSPEC
              PRINT VARIABLES
     END LØØP
PERCNT = PCSPEC * 100 / PCEQ
PRINT VARIABLES PERCNT
STØP
END
 DATA
 TSPEC = 0.5,
 PCSPEC = 1500,
 PCIG = 700,
 RHØP = 0.064
 CSTARO = 3320,
 A = 4.4 E-4,
 K = 172.65,
 N = 0.745,
 Q = 0.015,
 VC = 220,
 GAM = 0.66175,
 AT = 0.35,
 $END
```

Figure 3—SLANG formulation of boundary matching problem

condition. The complete set of input is shown in Figure 3.

The expression in the argument of the integration statement is an equation for dt/dP_c (where t = time, P_c = chamber pressure) during the start up transient of a solid rocket engine. The problem is to determine the value of chamber pressure at a specified time. This value is the upper limit of integration, and is being computed such that the integrated time (TCØMP) matches the specified time (TSPEC). That is, when the value of the constraint, CØNST, is zero, the upper integration limit PCSPEC is the value of chamber pressure at TSPEC. The final calculation of PERCNT computes the percentage of equilibrium chamber pressure, PCEQ, achieved at time TSPEC. PCEQ is computed from input data. The lower limit of integration, PCIG, is the ignition pressure, and is an input constant.

Strucutre of the Q system

The Q system is basically a Complier/Interpreter type package with the four major elements of the system shown in Figure 4. The user's input language (SLANG) is converted by a set of system-supplied macros into the MODTRAN language. The MODTRAN compiler then converts this language into an assortment of pseudo instructions and some associated tables. These are processed by the link editor before going to the interpreter for execution.

With this system it is possible to omit the macro processor if the user chooses to write directly in MODTRAN. On the other hand, a user might wish to use only the macro processor to perform some transformations on BCD data.

The ML/I processor was originally designed by P. J. Brown[4] of Cambridge University, who supplied the logic to TRW. The processor was converted to FORTRAN with little difficulty, and this version was included in the CUE system for making an initial pass at the input of non-programmer users. It was found that the average engineer in a hurry (for whom the system was designed) was unwilling to take the trouble of writing his own macros. Ideas for suitable macros were solicited from potential engineer users, and the resulting language was christened SLANG. Additions are continuously being made to SLANG to make it more useful. At one time it was planned to have four

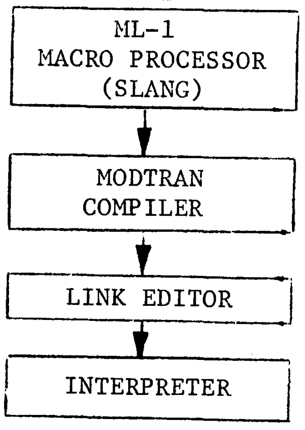


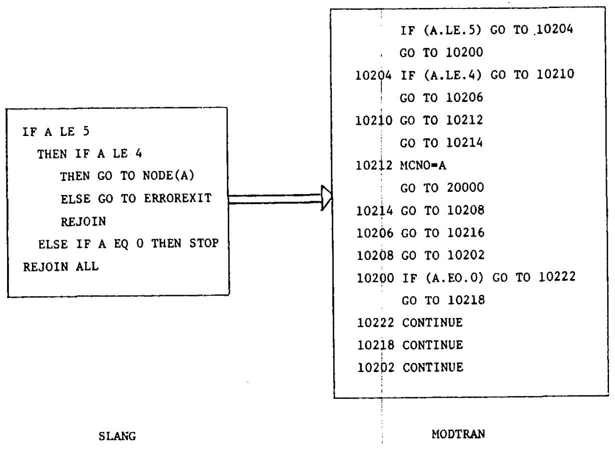
Figure 4—Basic Q system elements

"dialects" of SLANG of increasing degrees of sophistication, but this idea was abandoned in favor of a single version.

An example of how the processor converts SLANG macros to MODTRAN is shown in Figure 5. It is worth noting that the writing and debugging of macro definitions is considerably easier than would be the modification of the MODTRAN compiler itself. The programmer need in general be concerned only with the particular macro definition he is working on, and both his inputs and his outputs are in BCD.

It was originally planned to incorporate some of the more popular SLANG variations into MODTRAN, thus reducing processing time; unfortunately this project has been continuously postponed because of more pressing work. The more recent versions of the Q system allow for relocatable subroutines, which have served to reduce machine time considerably. Previously an illusion of subroutines was created by suitable macros, but it was necessary to process the user's entire input deck each time the equations were modified.

The MODTRAN language bears a strong resemblance to FORTRAN or BASIC, since it was designed by FORTRAN programmers. Algebraic statements are essentially the same, and DO loops are provided that have the same function except that they provide for backward stepping when desired. Arrays are as in FORTRAN except that they are limited to two indexes. READ and WRITE statements are similar, as are FORMAT statements. All variables are floating point as in BASIC, and corrections are automatically made for round-off errors on comparisons.



```
IF A LE 5
  THEN IF A LE 4
     THEN GO TO NODE(A)
     ELSE GO TO ERROREXIT
     REJOIN
  ELSE IF A EQ 0 THEN STOP
REJOIN ALL
```

```
      IF (A.LE.5) GO TO 10204
      GO TO 10200
10204 IF (A.LE.4) GO TO 10210
      GO TO 10206
10210 GO TO 10212
      GO TO 10214
10212 MCNO=A
      GO TO 20000
10214 GO TO 10208
10206 GO TO 10216
10208 GO TO 10202
10200 IF (A.EO.0) GO TO 10222
      GO TO 10218
10222 CONTINUE
10218 CONTINUE
10202 CONTINUE
```

Figure 5—Example of SLANG/MODTRAN conversion

Some MODTRAN statements are unusual, as for example EXECUTE label, which will cause a transfer to the label. When a JUMPBACK statement is encountered, control is tranferred to the statement following the EXECUTE label.

The FORTRAN subroutine concept is used in MODTRAN, but the COMMON method of communicating between subroutines was eliminated in favor of using the names of the variables themselves to communicate locations, as in BASIC and other languages. Another provision is that a variable can be typed as LOCAL to a particular subroutine, permitting subroutines to be written independently. The FORTRAN concept of calling sequence/argument list is used for communication between such subroutines, so that MODTRAN subroutines may be written and placed in the system library for general use.

The MODTRAN compiler has no provision for user-written functions (arithemetic or other), which makes it possible to determine an indexed variable even though no suitable allocation statement has appeared. When the compiler encounters what appears to be an array (which could be a misspelled system function), it processes the indices and assumes that by the time the statement is executed another statement making the allocation for the array will have been previously executed. The allocation statement can be either GLOBAL or LOCAL. For example, the statement:

$$\text{GLØBAL X (NRØW, NCØL), Y (10), Z} \qquad (8)$$

will cause the release of any arrays previously associated with X and Y and the allocation of ten words to Y as well as the generation of an array NRØW rows by NCØL columns for X. Such statements are executable, and once executed will apply to all other subroutines where the variables X and Y appear as globals. The variable Z in this statement is only given a global assignment by the compiler and that portion of the statement is not executable. If the compiler encounters a variable not defined as GLØBAL or LØCAL it assigns the variable to the nominal category previously defined by the user (normally GLØBAL).

Generation of partial derivatives

Perhaps the most interesting feature of the Q system is the way in which partial derivatives are treated. The MODTRAN language provides for specification of three levels of partials:

NO PARTIALS

FIRST PARTIALS List (9)

SECOND PARTIALS List

In these statements, List specifies which variables are to be the independent variables. An INDEPENDENT List statement might also be used for this purpose. A typical set of statements might be:

SECOND PARTIALS X, Y, Z

F = Y * X/Z (10)

D = F * F

These statements will cause the dependent variables D and F to be evaluated and all of the first and second partial derivatives of these two variables with respect to X, Y, and Z will be computed. The resulting storage requirements can become quite large; in the case of three independent variables one word is required for the value, three for first partials, and six for second partials, making a total of ten words (see equation 11). In the case of 15 independent variables 136 words of storage are required for each dependent variable. The system tries to hold down the total storage required by returning the partial storage to the free area wherever possible. We are considering a scheme to reduce the number of words required in the case of a dependent variable that is not a function of all the independent variables.

The actual operation of computing partial derivatives is carried out by the interpreter in the course of evaluating the given expressions of the problem. This evaluation consists essentially of a sequence of operations, which may be unary (performed on a single variable), for example SIN (X) or binary (performed on two variables), for example X*Y. The result of an operation either becomes one of the variables going into the next operation or, if the sequence is complete, the result is stored as the answer in the appropriate location. An operation is performed by the interpreter causing a transfer to one of the appropriate subroutines. Each subroutine has either one primary input (unary), or two primary inputs (binary), and a single output. The inputs (operands) may or may not have partials, and if they do it may be necessary to compute only first partials or both first and second partials. Consider the division operator, for example; either or both the divisor and dividend may or may not have partials, leading to four different possible cases. Each case is different with respect to how the partials of the resultant variables are computed, and four separate subroutines have been written for the division operator; the appropriate subroutine is selected by the interpreter during the execution of the user's program. If an equation is evaluated several times, it is entirely possible that a variable may have partials during one evaluation and none during another, in which case the appropriate subroutine would be executed during each evaluation. At the time that the link edit is performed every variable is given a core location assignment. If the variable has no partials then the value associated with the variable is stored in this location. If, however, during the execution of the model the variable develops partial derivatives by being a function of variables which have partials, then a vector is opened for the variable and the initial location replaced by a pointer to this vector. As an illustration, consider the following sample vector for a variable F when there are three independent variables X, Y, and Z:

$$F, \frac{\partial F}{\partial X}, \frac{\partial F}{\partial Y}, \frac{\partial F}{\partial Z}, \frac{\partial^2 F}{\partial X \partial X}, \frac{\partial^2 F}{\partial X \partial Y}, \frac{\partial^2 F}{\partial X \partial Z}, \frac{\partial^2 F}{\partial Y \partial Y}, \frac{\partial^2 F}{\partial Y \partial Z}, \frac{\partial^2 F}{\partial Z \partial Z} \qquad (11)$$

All of the variables which have partials will have similar associated vectors. The independent variables will each have such a vector where all of the partials are zero except for the one corresponding to the derivative of the independent with respect to itself where a value of one will be stored. When an INDEPENDENT statement is encountered all of the vectors which happen to be active at that point are deleted and a new set of independent vectors set up. As the run progresses new dependent vectors will be allocated.

In MODTRAN statements for unary operations, the subroutines tend to be similar except for the three lines for the evaluation of F, S1, and S2 (see below for definition of S1 and S2). In the example of Figure 6, SINX is used as the name of the interpreter subroutine for evaluating the sine of a variable. NUMIND indicates the number of independent variables, E is the operand vector, and F is the resultant vector.

There would of course be similar routines for COS, EXP, TAN, etc., which might appear in the user's input. In the general case all of these subroutines would be identical except for F, S1 and S2. Suppose oper corresponds to the unary operator that is being used, then F, S1 and S2 can be expressed in general as follows:

```
        SUBROUTINE SINX(E,F)
        DIMENSION
        CØMMØN/NUMIND/NUMIND
1       F(1) = SIN(E(1))
2       S1= COS(E(1))
3       S2= -SIN(E(1))
        M=NUMIND
        DO 20 K=1,NUMIND
        IF (FIRST) GØ TO 20
        S3=S2*E(K)
        DO 10 L=1,K
        M=M+1
10      F(M)=E(M)*S1+S3*E(L)
20      F(K)=F(K)*S1
        RETURN
        END
```

Figure 6—Sample interpreter subroutine for unary operation

$$F = \text{oper}(E)$$

$$S1 = \frac{\partial \text{oper}(E)}{\partial E} \qquad (12)$$

$$S2 = \frac{\partial^2 \text{oper}(E)}{\partial E^2}$$

Should it be necessary to evaluate only first partials then at the time each of the subroutines is executed the logical variable FIRST will be set to true and the computing of the second partials will be bypassed.

Binary functions vary considerably, but an example of this type of function is given in Figure 7 for the multiplication operation. D and E are the operands and F is the resultant vector.

Perhaps it would be useful to demonstrate the manner in which the equations of the MUL routine were derived. Assuming for purposes of explanation that X & Y are the only independent variables then we know that

$$F = D \cdot E \qquad (13)$$

```
        SUBROUTINE MUL(D,E,F)
        DIMENSION D(1),E(1),F(1)
        CØMMØN/NUMIND/NUMIND
        M=NUMIND
        DO 20 K=1,NUMIND
        DO 10 L=1,K
        IF(FIRST) GØ TO 20
        M=M+1
10      F(M)=D(M)*E(1)+E(M)*D(1)+D(K)*E(L)+D(L)*E(K)
20      F(K)=D(1)*E(K)+D(K)*E(1)
        F(1)=D(1)*E(1)
        RETURN
        END
```

Figure 7—Sample interpreter subroutine for binary operation

Then from any table of derivatives

$$\frac{\partial F}{\partial X} = D.\frac{\partial E}{\partial X} + \frac{\partial D}{\partial X} \cdot E \qquad (14)$$

while

$$\frac{\partial F^2}{\partial X \partial Y} = D.\frac{\partial^2 E}{\partial X \partial Y} + \frac{\partial D.}{\partial Y}\frac{\partial E}{\partial X} + \frac{\partial D.}{\partial X}\frac{\partial E}{\partial Y} + \frac{\partial^2 D.}{\partial X \partial Y}E \qquad (15)$$

The reader should be able to convince himself that the statement at label 20 on Figure 7 corresponds to (14) while the statement at label 10 corresponds to (15). It should also be possible to place these statements in the context of a generalized number of independent variables by referencing equation No. 11.

Tabular function defined by arrays of input data are handled by a system routine which fits a polynomial to the data and then assumes that the derivatives of the polynomial correspond to those of the function. This is of course rather cumbersome and the results may not be accurate for many functions.

System supplied routines

In addition to the usual system-supplied routines such as those illustrated above, the Q system attempts to provide rather elaborate sets of routines which are called algorithms. These routines should remove some of the burden off the user to provide a method of solution. They are kept in the Q FORTRAN library and are called as needed. Since one of the main features of the system is the ability to take partial derivatives, it is not surprising that most of these routines are built around this capability. The most important and most frequently used of these algorithms are called SOLVE, OPTIM, and INTEG.

The SOLVE algorithm makes use of the Newton-Raphson technique in order to drive specified functions to zero. In order to do this it is necessary to evaluate the first partial derivative of the functions and apply correction factors to the independent variables based on this information until the convergence criteria is reached. Since it is possible to obtain first partial derivatives by numerical techniques this method of solving functions is rather common. The partials of the Q system should be more accurate, however, especially in the neighborhood of singularities.

The optimization function is initiated by writing MAXIMIZE, MINIMIZE, or CRITICALIZE followed by the variable to be optimized and an INDEPENDENT statement for the variables the system will vary in an attempt to find a solution. The partial derivatives play a major role in this algorithm. Originally the system made use of Lagrangian multipliers in conjunction with the Newton-Raphson technique for optimization, but this method has been superseded by a modified version of rotational discrimination, as described by Law and Fariss.[5]

The INTEG algorithm is used to integrate a set of simultaneous differential equations by a fourth-order Runge-Kutta method. It can be combined with the SOLVE algorithm to solve two-point boundary value problems, as in the second SLANG example given earlier. In this case the INTEG routine is imbedded within a SOLVE loop, where the solution to the SOLVE operation is the end points to match certain expressions. Other routines are available to save and restore partial derivatives, to add and delete independent variables, to input or printout all global variables, etc.

Implementation of the system

As implemented on the CDC SCOPE system Q requires two back-to-back executions under SCOPE with a compilation by the SCOPE FORTRAN compiler separating the executions. The user need not be aware of these efforts in his behalf, however, as he submits one job and gets one output. It is even possible to place the SCOPE control cards necessary to run the Q system onto a file, along with the various other files required by the system, so that the user need only see a few of the SCOPE control cards.

In the first execution under the Q system a basic monitor surveys the user control cards to determine the objective of the decks which the user supplies. Thus in one run the user might have some SLANG decks to be sent via the ML/I processor to the MODTRAN compiler, some MODTRAN decks which would go directly to that compiler, some FORTRAN decks for compilation by the SCOPE FORTRAN compiler when it is called in between the executions, and perhaps even some FORTRAN and/or MODTRAN relocatable decks. Control cards are intermixed with other input and some action is normally taken immediately with the cards following a control card until the next control card in encountered. Sometimes a set of cards is sent directly to a processor, such as a MODTRAN deck going to the MODTRAN compiler, while in other cases it is necessary to place the deck on a file for later processing, such as a FORTRAN deck. The flow of operations is shown in Figure 8.

Once all the user's input except for data cards has been read and either processed or assigned, the link editor is called in to tie together the various MODTRAN routines. The link editor assigns all of the variables to their final locations in the data portion of the bucket and performs the required relocation of the pseudo instructions. An attempt is made to satisfy all of the externals referenced from MODTRAN routines with MODTRAN entry points, including a search of the Q MODTRAN library file. The references which are still unsatisfied are assumed to be for FORTRAN routines and a search of the Q FORTRAN library file is performed. Any routines found there are pulled off for loading by the next execution. At this point the link editor writes a FORTRAN routine which will be compiled by the CDC FORTRAN compiler. This routine consists of a computed GO TO followed by a call to each of the routines which it has determined are FORTRAN. For example it might write:

```
  SUBROUTINE CALLI
  COMMON/N/N
  GO TO (1,2),N
1 CALL INTEG                    (15)
  RETURN
2 CALL BROP
  RETURN
  END
```

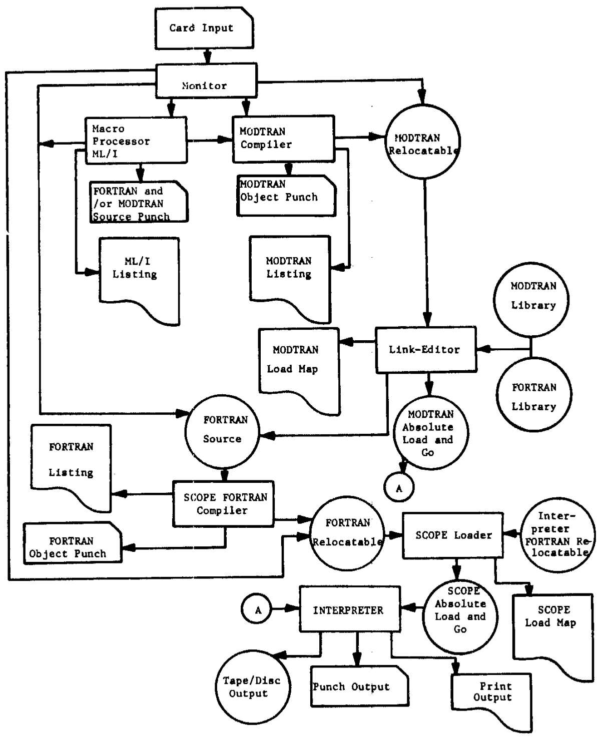


Figure 8—Flow diagram of the Q system

Actually the routine CALLI will be more complicated than this example, since the user is allowed to have arguments to these FORTRAN routines. The basic concept is, however, that this is the manner in which it is made posible to call a FORTRAN routine from a MODTRAN routine. Should the user, for example write

CALL INTEG (16)

in MODTRAN he will in actual fact be calling subroutine CALLI with N set equal to 1. Since the routine CALLI is placed in the input stream to the FORTRAN compiler the user receives a listing of this routine in the middle of his output. It was not deemed worthwhile to try to suppress this listing, since the user might very well be compiling some of his own routines on the same call to the FORTRAN compiler.

After the link editor has relinquished control to the FORTRAN compiler and that processor has completed its task, the second execution of the user's job begins. This consists of a loading of the Q interpreter and all of the FORTRAN routines which have been collected by the link editor on the previous execution and placed on the FORTRAN relocatable file. The nature of this core load varies radically depending on what the user requires. Control is initially passed to the main MODTRAN routine but after that the user is on his own.

During execution of a MODTRAN routine, the pseudo instructions put out by the MODTRAN compiler are being interpreted. As is usual with interpretive schemes, quite a bit of control can be exercised in making sure that the user is not getting into trouble and in taking some appropriate action when he attempts to do something which would be improper.

There are three user data areas in the Q system: variables, arrays, and partials. The three areas are rather heavily intertwined with pointers, a pointer being distinguishable from a value by the fact that it is a positive integer while a value is a normalized floating point number. Initially only the variable area is assigned (by the link editor) and the interpretation of the user's program causes the buildup of the other two areas. Thus suppose the user says

GLOBAL X(10) (17)

where X was previously only a value. An array will be opened in the array area and the location at which X was assigned will be replaced by a pointer to the array. A double tag system as described by Knuth[6] is used for the allocation of arrays, a system which allows a good method of returning variable length arrays to the free area. Two more words are used to specify the dimensions on the array, causing the use of four words in addition to the actual size of the user's array. When an array is released, the two words which were used for indexing are replaced by linking pointers to facilitate the search for free areas of adequate size. The user of course need not be aware of this process when he opens or closes an array.

It is also frequently the case that a variable will not only have a value associated with it but will have some partial derivatives. In this case the location of the variable, or the indexed location within its array, is replaced by a pointer into the partial area. At the location in the partial area the value and the associated partials are stored. Some rather complicated chaining-down pointers may result before the desired location is finally achieved; but normally if the user is taking partials he will be spending most of his time during execution doing just that, computing partials, and the time spent on pointers will be relatively small. It was also necessary to make some provision for returning these partial vectors to the free area, but this is a rather simple matter since all of these vectors are of the same length.

Additional complications are entered into the system when the user performs such operations as saving partials and beginning a new set. This is basically performed by closing off the current partial area and opening up a new one. A swapping of pointers with values occurs so that the partials can be restored later.

SUMMARY

No claims are made that the Q system is a direct challenge to other computer systems. It does, however, offer anapproach to some rather difficult problems. As was pointed out earlier, it is easy to introduce modifications into the SLANG language, a characteristic which is not common to programming languages. It is also rather easy to introduce new algorithms into the system, thereby expanding its problem solving capability. It is hoped that the Q system constitutes a basis for further development along these lines since the user is frequently denied this flexibility in a computer system.

ACKNOWLEDGMENTS

Bob Kennedy helped in the preparation of this article and Dave Adamson provided the sample SLANG problems along with the discussion of them.

REFERENCES

1 R E WENGERT
A simple automatic derivative evaluation program
C A C M Vol 7 1964 463-464

2 R L GLASS
An elementary discussion of compiler/interpreter writing
Computing Surveys 1 1969 55-77

3 D ADAMSON
Introduction to SLANG
TRW Doc 99900-6672-R0-00 1968

4 P J BROWN
Macro processors and their use in implementing software
Thesis Univ Math Lab Cambridge England 1968

5 V J LAW R H FARISS
Rotation discrimination for optimization with limits on the variables
Preprint 19B Second Joint AICHE-IIQPR Meeting
May 19-22 1968 Tampa Fla

6 D E KNUTH
The art of computer programming
Addison Wesley Publishing Co Vol 1 Chap 2 1968 p.442