COMPUTER METHODS IN

LINEAR AND QUADRATIC MODELS

Jørgen Kjær

Haldor Topsøe, Vedbæk, Denmark

1970

Production:

Akademisk Forlag

ISBN 87 500 1072 7

Printing Zano Print
Denmark

PREFACE

The practical use of a digital computer for calculations creates a demand for literature which describes the programs and how to use them. The standard for program manuals varies considerably from one computer installation to another, and it is often found that a very useful program has no manual at all, whereas other programs with magnificent manuals are not popular among the users.

At the Haldor Topsøe computer installation, where the first calculations were made in 1958, the writing of manuals for programs and subroutines soon attained a standard level which at that time was considered a fairly happy compromise, giving the necessary information to the user of the program, whereas details of interest to the programmers were placed in appendices. However, when an IBM 360/44 computer was introduced in 1968 in addition to the GIER computers, a revision of the documentation standard was required.

The possibility of having two sets of programs: ALGOL programs for the GIER computer and FORTRAN programs for the IBM computer led us to divide the documents into three classes:

1. User Manuals. These refer to a particular program in whatever language it may be written, and contain the information strictly necessary for a program user in order to run calculations on the program. The minimum is a discussion of input and output, a set of input specifications, and a typical output report.

2. Method Descriptions. This type of document describes the details of a mathematical, chemical, or other type of computer calculation. In the ideal case the method description explains the matter in normal language without the use of ALGOL, FORTRAN, or other programming language. In most cases, however, it is practical to illustrate the method in question by means of a small procedure in ALGOL or a subroutine in FORTRAN. In the present book, all examples of this type are given in ALGOL, which is generally accepted as a good language for communucation of calculation methods. It should be noted, that these ALGOL procedures are normally different from those actually used in running programs, because

they have been stripped of all special features concerning input/output, data check, matters concerning the computer or operating system used in the implementation, etc. From this it will be understood that the method description does not distinguish between programs and procedures (or programs and subroutines), both aspects are covered by the same document.

3. Program and Subprogram Descriptions. These documents are written for the FORTRAN programs and subprograms. They contain a list of all variables used, source listings, and flow charts. Any difference between the method description and the actual implementation is also explained. All this information is of interest to the programmers, not to the program users. These descriptions are stored and maintained on magnetic tape.

The present book is a slightly shortened version of the internally used Method Descriptions covering the subjects: Solution of linear and non-linear equations, generation of linear and quadratic models, and optimization of such models. Additional books on other methods are planned for later publication. The purpose of the book is to present a handy collection of these methods, mainly for internal work but also for those outside the firm who may be interested in studying such methods that have passed the trial of many years test in the computer. The greater part of the book can be read with only a high school knowledge of mathematics. Illustrating examples have been given whenever possible.

Vedbæk, September 1970

Jørgen Kjær

CONTENTS

1. INTRODUCTION

The present book describes a collection of computer methods within the field of linear and quadratic algebra and a few numerical methods of a related nature that are conveniently described in the same context.

Chapter 2 deals with solution of linear equations. The elementary nature of this problem is easily understood from the following considerations. In the basic arithmetic operation:

(1.1) $X = A/B$

where X is calculated as A divided by B, we can also say that X is the solution to the equation:

(1.2) $B \times X = A$

X must be calculated as the number which multiplied by B yields A. If we have two equations to satisfy, instead of a single equation:

(1.3) $B \times X1 + C \times X2 = A$
(1.4) $D \times X1 + E \times X2 = F$

the problem is now to find the two unknowns, X1 and X2, so that the equations are satisfied. Solution of several equations in several unknowns is thus a generalisation of the division problem, and it is used as a basic building block in some of the other methods described in later chapters.

In Chapter 3 we describe how to generate linear or quadratic models as approximations to more complicated functions. A model with N coefficients is made so that it exactly fits the original function at N points (apart from rounding errors). The advantage of linear or quadratic function models is that they are easy to evaluate and to differentiate. Many types of function analyses can be carried out on models, as described in the following chapters.

Chapter 4 describes the problem of finding where a function becomes zero. It may be a function of a single variable:

(1.5) $F(X) = 0$

or several functions of several variables:

$$\begin{aligned} &F1(X1, X2, X3) = 0 \\ &F2(X1, X2, X3) = 0 \\ &F3(X1, X2, X3) = 0 \end{aligned} \tag{1.6}$$

in which we must find X1, X2, and X3 so that all three functions become zero.

Finally, Chapter 5 deals with optimization: How to find the maximum (or minimum) value of a function. Two different approaches are considered: An analytic method in which the derivatives are calculated from the quadratic model and put equal to zero, and a more direct method that performs a simple function evaluation without complicated mathematical treatment. The latter method is essentially the Direct Search method known from the literature.

2. SOLUTION OF LINEAR EQUATIONS

2.1. General on Linear Equations and Matrix Inversion.

As a basic introduction to those not familiar with linear equations we first give a very elementary example of a problem that can be formulated and solved as a set of linear equations.

We make three purchases in a shop. We first buy 3 apples, 4 bananas, and 5 coconuts and pay kr. 4.19 for this. We then buy 6 apples, 2 bananas, and 3 coconuts for a total price of kr. 4.13. Finally, we buy 1 apple, 7 bananas, and 4 coconuts for kr. 4.17. We have now sufficient information to calculate the price of one apple, of one banana, and of one coconut, provided of course that these prices were the same in all three purchases. If we call the unknown prices A, B, and C, we may formulate the problem as three equations:

(2.1) $3\times A + 4\times B + 5\times C = 4.19$

(2.2) $6\times A + 2\times B + 3\times C = 4.13$

(2.3) $1\times A + 7\times B + 4\times C = 4.17$

We may also write the three unknowns as x[1], x[2], and x[3], i.e. as the array x[1:3]. Similarly, the numerical coefficients and the prices on the right hand side may be written as another array, A, of the dimensions: A[1:3,1:4]. The three equations then become:

(2.4) $A[1,1]\times x[1] + A[1,2]\times x[2] + A[1,3]\times x[3] = A[1,4]$

(2.5) $A[2,1]\times x[1] + A[2,2]\times x[2] + A[2,3]\times x[3] = A[2,4]$

(2.6) $A[3,1]\times x[1] + A[3,2]\times x[2] + A[3,3]\times x[3] = A[3,4]$

Let us now assume, that we make this series of purchases at many different shops, but always with the same three combinations: first 3 apples, 4 bananas, and 5 coconuts, then 6 apples, etc. We can then see that the array A[1:3,1:3] will be the same for all sets of purchases, whereas the right hand sides will probably vary from one shop to the next. It is now more convenient to operate with the two arrays A[1:3, 1:3] and B[1:3]. The three equations may now be written:

(2.7) A[1,1]×x[1] + A[1,2]×x[2] + A[1,3]×x[3] = B[1]
(2.8) A[2,1]×x[1] + A[2,2]×x[2] + A[2,3]×x[3] = B[2]
(2.9) A[3,1]×x[1] + A[3,2]×x[2] + A[3,3]×x[3] = B[3]

The equations must be solved for a fixed array A[1:3,1:3] and for different values of the elements in the array B.

The array A[1:3,1:3] is called a square matrix, and it may be shown, that it is normally possible to transform the given matrix A into another matrix, C[1:3,1:3], which is called the inverse matrix corresponding to A, and so that the solutions may now be calculated from:

(2.10) x[1] := C[1,1]×B[1] + C[1,2]×B[2] + C[1,3]×B[3]
(2.11) x[2] := C[2,1]×B[1] + C[2,2]×B[2] + C[2,3]×B[3]
(2.12) x[3] := C[3,1]×B[1] + C[3,2]×B[2] + C[3,3]×B[3]

We see that the element C[r,c] is a measure of how big a part of the solution x[r] comes from the element B[c]. We may also say that the vector x[1:3] is calculated by a matrix multiplication of the matrix C[1:3,1:3] by the other vector B[1:3].

The problem may thus be formulated in two different ways:

1. Solve the equation system defined by the non-square matrix A[1:3,1:4] directly.

2. Invert the square matrix A[1:3,1:3] to give C[1:3,1:3] and multiply by B.

From a mathematical point of view the two methods should yield the same results and this is normally also the case. However, from a purely numerical point of view the matrix inversion involves much more calculation than the direct solution, and there is therefore a greater risk of accumulation of errors. In certain cases it is actually found that the matrix inversion gives completely wrong results, whereas the direct solution works satisfactorily.

Whenever possible, the direct solution method should be used. For such problems which are formulated with use of matrix inversion it is strongly recommended to reformulate the problem so that the inversion is avoided, if possible. A further discussion of matrix inversion is given in section 2.4.2.

2.2. Solution Method.

Gaussian elimination is the classical solution method for linear equations. It is described in many books on numerical analysis. See for instance Lapidus (1962). The variant considered here is copied directly from the procedure:

Det Gauss

published by Regnecentralen (Zachariassen(1963)). Calculation of the determinant has been deleted.

The method involves three phases which are described below: Equilibration, the elimination proper, and the back substitution.

2.2.1. Equilibration. As an illustration of the calculation method we consider the three equations (2.1) to (2.3) above. The 3×4 matrix is:

3.0000	4.0000	5.0000	4.1900
6.0000	2.0000	3.0000	4.1300
1.0000	7.0000	4.0000	4.1700

It is always possible to multiply an equation by an arbitrary factor (different from zero). The equation is still valid. We can also multiply an equation by a factor ($\neq 0$) and add it to one of the other equations. The principle used in the Gaussian elimination method is to transform the original matrix into another matrix with zeroes below the diagonal:

3	4	5	4.19
0	D	E	F
0	0	G	H

Here, D, E, F, G, and H are new elements obtained during the calculations.

As the elimination calculations are made with a finite accuracy, it is important to arrange the calculation flow in such a way that the rounding errors are reduced as much as possible. Various tricks can be used for this purpose. One of these tricks is the so-called equilibra-

tion which is carried out before the elimination is started. Each equation is multiplied by a power of two so that the sum of the squares of the elements in that row lies in the range from 0.25 to 1.

The effect of this equilibration is to avoid elements of strongly varying order of magnitude. No element will be larger than 1, but there may be very small elements. We have selected a scale factor which is a power of 2, because the scaling can then be carried out without any kind of rounding error. The sum of the squares in a row (the norm of the row) is calculated for the square matrix only, but the scaling is, of course, extended to the right hand side.

In the example above we get the norms:

$$3\uparrow 2 + 4\uparrow 2 + 5\uparrow 2 = 50$$
$$6\uparrow 2 + 2\uparrow 2 + 3\uparrow 2 = 49$$
$$1\uparrow 2 + 7\uparrow 2 + 4\uparrow 2 = 66$$

which give the scale factors:

$$2\uparrow(-3) = 1/8$$
$$2\uparrow(-3) = 1/8$$
$$2\uparrow(-4) = 1/16$$

When these factors are applied, the equations become:

(2.13)	0.3750	0.5000	0.6250	0.5237
(2.14)	0.7500	0.2500	0.3750	0.5162
(2.15)	0.0625	0.4375	0.2500	0.2606

The elements are here given with four decimals only.

2.2.2. Gaussian Elimination. The strategy involved in the elimination is to create zeros below the diagonal:

0.3750	0.5000	0.6250	0.5237
0	xxxxxx	xxxxxx	xxxxxx
0	0	xxxxxx	xxxxxx

This is carried out in steps dealing with one column at a time. The first step is to create a zero in the first column of the second row, i.e. where the element 0.7500 is placed now:

(2.16)	0.3750	0.5000	0.6250	0.5237
(2.17)	0.7500	0.2500	0.3750	0.5162

This is carried out by multiplying equation 1 (2.16) by the factor 0.7500/0.3750 = 2 and subtracting this from the second row. The first row is unchanged:

0.3750	0.5000	0.6250	0.5237	Row 1
0.7500	0.2500	0.3750	0.5162	Row 2
0.7500	1.0000	1.2500	1.0475	2×Row 1
0	-0.7500	-0.8750	-0.5313	Row 2 - 2×Row 1

We can then proceed by creating a zero in the first column of the third row by multiplication of row no. 1 by the factor 0.0625/0.3750 and subtracting this from the third row.

Before we go on with these calculations we must make certain that these operations do not spoil the accuracy of the calculations. When we multiply a row by a factor (here 2 in the first case) and subtract the result from another row, the original information in this other row will be lost completely, if the factor is very large. The factors are calculated as:

0.7500/0.3750	for row 2
0.0625/0.3750	for row 3

The divisor, 0.3750, is the so-called pivot element which is an element on the diagonal of the matrix. If the factor must be small, the pivot element must be large, and this can be acccmplished by performing what is known as complete pivoting, i.e. the rows and columns are exchanged in such a way that the largest possible element is always used as a pivot element. This complete pivoting is included in the present method.

We must now go back to equations (2.13 - 2.15) and find the largest element in the square matrix. This is the number 0.7500 in row no. 2

and column no. 1. In order to move this element to the place of the element A[1,1] we must exchange rows 1 and 2. This gives:

(2.18)	0.7500	0.2500	0.3750	0.5162
(2.19)	0.3750	0.5000	0.6250	0.5237
(2.20)	0.0625	0.4375	0.2500	0.2606

The factors now become:

$0.3750/0.7500 = 0.5$ for row 2
$0.0625/0.7500 = 0.0833$ for row 3

The elimination can now be carried out by multiplying each element in row no. 1 by the factor 0.5 and subtracting it from the corresponding element in row no. 2. Similarly, each element in row no. 1 is multiplied by the factor 0.0833 and subtracted from the corresponding element in row no. 3. The result becomes:

(2.21)	0.7500	0.2500	0.3750	0.5162
(2.22)	0	0.3750	0.4375	0.2656
(2.23)	0	0.4167	0.2188	0.2176

We can now continue the elimination to get a zero instead of the element 0.4167. Before this is done we must search for the maximum element in the 2×2 matrix:

0.3750	0.4375
0.4167	0.2188

The numerically largest element is 0.4375, and this must then be moved to the place A[2,2] by an exchange of columns 2 and 3. The complete matrix then becomes:

(2.24)	0.7500	0.3750	0.2500	0.5162
(2.25)	0	0.4375	0.3750	0.2656
(2.26)	0	0.2188	0.4167	0.2176

The exchange of two rows can be made without further administration, because this is just an exchange of two equations. An exchange of two columns, however, corresponds to an exchange of two variables, here x[2] and x[3], and this exchange must be recorded and taken care of. We do this by means of an integer array, p[1:3], in which we put p[2] equal to 3 to indicate the exchange of columns 2 and 3. During the first elimination, p[1] was set to 1 to indicate that no column exchange was made here.

The last elimination requires the factor:

$$0.2188/0.4375 = 0.5000$$

and we multiply each element in row 2 by this factor and subtract the result from the corresponding element in row 3. The elements which are zero already are not treated. This gives:

(2.27)	0.7500	0.3750	0.2500	0.5162
(2.28)	0	0.4375	0.3750	0.2656
(2.29)	0	0	0.2292	0.0848

2.2.3. Back-Substitution. After the elimination we come to the last phase: Back-substitution. The significance of equations (2.27 - 2.29) is the following:

$$0.7500\times x[1] + 0.3750\times x[3] + 0.2500\times x[2] = 0.5162 \quad (2.30)$$

$$0.4375\times x[3] + 0.3750\times x[2] = 0.2656 \quad (2.31)$$

$$0.2292\times x[2] = 0.0848 \quad (2.32)$$

Note, how x[2] and x[3] are exchanged because of the column exchange carried out above for columns 2 and 3. From equation (2.32) we can find x[2] immediately:

$$x[2] = 0.0848/0.2292 = 0.3700 \quad (2.33)$$

It is convenient to store this value instead of the element 0.0848:

(2.34)	0.7500	0.3750	0.2500	0.5162
(2.35)		0.4375	0.3750	0.2656
(2.36)			0.2292	0.3700

We can then insert x[2] into equation (2.31) and calculate x[3]:

(2.37) $0.4375 \times x[3] = 0.2656 - 0.3750 \times 0.3700 = 0.1269$

(2.38) $x[3] = 0.1269/0.4375 = 0.2900$

This value is stored instead of 0.2656:

(2.39)	0.7500	0.3750	0.2500	0.5162
(2.40)		0.4375	0.3750	0.2900
(2.41)			0.2292	0.3700

Finally, x[1] is found from equation (2.30) by insertion of x[2] and x[3]:

(2.42) $0.7500 \times x[1] = 0.5162 - 0.3750 \times 0.2900 - 0.2500 \times 0.3700 = 0.3150$

(2.43) $x[1] = 0.3150/0.7500 = 0.4200$

This value is also inserted on the right hand side:

(2.44)	0.7500	0.3750	0.2500	0.4200
(2.45)		0.4375	0.3750	0.2900
(2.46)			0.2292	0.3700

The original right hand side column has now been replaced by the solution:

x[1]	0.4200
x[3]	0.2900
x[2]	0.3700

In order to rearrange this array in the correct order we must make use of the p-array introduced above. It contains the values:

p[1]	p[2]	p[3]
1	3	3

The value of p[3] will always be 3 because no elimination is made for the last row. The p-list is scanned backwards and whenever we find an element for which p[i] ≠ i, the columns p[i] and i will be exchanged again. This gives the final solution:

x[1]	0.4200
x[2]	0.3700
x[3]	0.2900

2.3. The Procedure LEQ1.

The procedure LEQ1 performs the solution of N linear equations in N unknowns after the method outlined in the previous section. It has four formal parameters:

integer N: The number of unknowns. In the example above: N = 3.

integer M: The number of right hand sides. For the example above we have M = 1. The extension from M = 1 to higher values is quite easy and is explained below.

array A[1:N,1:N+M]. This is the array defining the set of equations. In the example above we used A[1:3,1:4]. After the call of the procedure the original A-matrix is completely destroyed, and the solutions are found in the last M columns: A[1:N,N+1:N+M].

real eps: It was shown above that it is important to use pivot elements which are as large as possible. If a pivot element becomes zero, the calculations cannot be carried through because we get division by 0. Very small pivot elements will give numerical troubles. The parameter eps is compared to each pivot element selected, and if the procedure finds a pivot element with an absolute value less than eps, the calculation is interrupted.

integer LEQ1: The procedure is of the type integer. After the calculation, LEQ1 will have either of the two values:

0: Solution OK.

1: Pivot trouble. No solution found because the absolute value of one of the pivot elements has become less than eps.

The procedure has the declaration:

```
integer procedure LEQ1(N, M, A, eps);
value N, M, eps;
integer N, M;
array A;
real eps;
begin
   integer i, j, k, i1, j1;
   real max, f2, factor;
   integer array p[1:N];
   M := N + M;
   LEQ1 := 0;
   for i := 1 step 1 until N do
   begin
      max := 0;
      for j := 1 step 1 until N do
      max := max + A[i,j]↑2;
      if max > 1 ∨ max < 0.25 then
      begin
         f2 := 2↑(-entier(ln(max)/1.3863 + 1));
         for j := 1 step 1 until M do
         A[i,j] := A[i,j]×f2;
      end if max;
   end for i: equilibration;
   for k := 1 step 1 until N do
   begin
      max := 0;
      for i := k step 1 until N do
      for j := k step 1 until N do
      begin
         factor := abs(A[i,j]);
         if max ≤ factor then
         begin
            max := factor;
            i1 := i;
            j1 := j;
         end if larger;
```

```
    end for i and j: pivot search;
    if max < eps then
    begin
       LEQ1 := 1;
       go to EX;
    end error exit;
    max := A[i1,j1];
    if i1 ≠ k then
    for j := k step 1 until M do
    begin
       factor := A[k,j];
       A[k,j] := A[i1,j];
       A[i1,j] := factor;
    end for j: row interchange;
    p[k] := k;
    if j1 ≠ k then
    begin
       p[k] := j1;
       for i := 1 step 1 until N do
       begin
          factor := A[i,k];
          A[i,k] := A[i,j1];
          A[i,j1] := factor;
       end for i;
    end interchange of columns;
    for i := k + 1 step 1 until N do
    begin
       factor := A[i,k]/max;
       for j := k + 1 step 1 until M do
       A[i,j] := A[i,j] - A[k,j]×factor;
    end for i: reduction;
end for k;
for k := N + 1 step 1 until M do
for i := N step -1 until 1 do
begin
    factor := A[i,k];
    for j := i + 1 step 1 until N do
    factor := factor - A[i,j]×A[j,k];
    A[i,k] := factor/A[i,i];
end solving;
```

```
   if M ≠ N then
   for i := N - 1 step -1 until 1 do
   begin
      i1 := p[i];
      if i1 ≠ i then
      for k := N + 1 step 1 until M do
      begin
         factor := A[i,k];
         A[i,k] := A[i1,k];
         A[i1,k] := factor;
      end for k;
   end for i and solution interchange;
EX:end LEQ1;
```

A few additional explanations may be required to understand the procedure.

The first for-statement counts i from 1 to N and performs the equilibration by multiplying rows having a norm, max, outside the range from 0.25 to 1 by the factor:

(2.47) $2\uparrow(-\text{entier}(\ln(\text{max})/1.3863 + 1))$

The numerical factor, 1.3863, is ln(4).

The elimination is controlled by a large for-statement counting in k from 1 to N. The pivot search is controlled by the double for-statement counting i and j from k to N. The subscripts of the numerically largest element are assigned to i1 and j1. If the largest element is not placed as A[k,k], exchange of rows and/or columns will be carried out.

After the possible exchange the elimination with the pivot element A[k,k] is controlled by the for-statement counting i from k+1 to N. For each value of i the factor is calculated as the first element in row no. i to be treated (that below the pivot element) divided by the pivot. The pivot row times the factor is then subtracted from the elements in row no. i.

The back substitution is controlled by the for-statement counting in k from N+1 to N+M. Note, that M is internally changed into N+M. One right hand side is treated for each value of k, and it is carried out by another for-statement counting i from N to 1.

Finally, the for-statement counting i from N-1 to 1 performs the re-exchange of the columns.

The physical exchange of rows and columns carried out here can be avoided by use of another integer array, q[1:N], indicating which rows have been exchanged. This requires more space and subscript handling, but should give a faster procedure.

2.4. Matrix Inversion.

2.4.1. Matrix Inversion with LEQ1. If it is required to calculate the inverse A[1:N,1:N] to a given square matrix: P[1:N,1:N], this can be made after the LEQ1-method as follows. Let us assume that N = 4. The original P-matrix is augmented by four right hand sides which make up a unit matrix:

P[1,1]	P[1,2]	P[1,3]	P[1,4]	1	0	0	0
P[2,1]	P[2,2]	P[2,3]	P[2,4]	0	1	0	0
P[3,1]	P[3,2]	P[3,3]	P[3,4]	0	0	1	0
P[4,1]	P[4,2]	P[4,3]	P[4,4]	0	0	0	1

When this matrix is treated by the LEQ1-method using N = 4 and M = 4 the inverse matrix will appear instead of the unit matrix:

-	-	-	-	Q[1,1]	Q[1,2]	Q[1,3]	Q[1,4]
-	-	-	-	Q[2,1]	Q[2,2]	Q[2,3]	Q[2,4]
-	-	-	-	Q[3,1]	Q[3,2]	Q[3,3]	Q[3,4]
-	-	-	-	Q[4,1]	Q[4,2]	Q[4,3]	Q[4,4]

or, more correctly, the element Q[i,j] of the inverse matrix appears as the element P[i,N+j] in the array P[1:4,1:8]. The elements of the left part of P are destroyed.

2.4.2. Direct Solution Versus Inversion. It was stated in section 2.1 that for numerical reasons it is preferable to solve a set of linear equations directly instead of inverting the matrix and then multiplying by the right hand side. In order to illustrate this fact we have used a

program which generates a large number of 3×3 matrices with random elements. For each matrix a corresponding right hand side was calculated by adding the elements in each row. This corresponds to the assumption:

$$x[1] = x[2] = x[3] = 1$$

The system was then solved in two different ways: First by direct solution of the 3×4 matrix and then by inversion of the 3×3 matrix and multiplication by the right hand side. The two sets of solutions found were compared with the theoretical values: 1, 1, 1. The sum of the squares of the deviations was calculated for either methods and whenever a poorer solution was found, the data for that matrix were printed out. The worst example found was:

$6.07500895_{10}5$	$1.03401437_{10}2$	$4.08237112_{10}8$	$4.08844716_{10}8$
$9.99162505_{10}7$	$1.19456064_{10}1$	$2.30505231_{10}4$	$9.99393127_{10}7$
$6.45568033_{10}6$	1.06064020	$1.57040148_{10}3$	$6.45725177_{10}6$

The calculated solutions were:

	Direct Solution	Inversion and Multiplication
x[1]	1.00000001	1.00000003
x[2]	0.95279924	0.68750000
x[3]	1.00000001	1.00000007

Clearly the value of x[2] found by inversion and multiplication is not acceptable. Less horrible examples are available, but they are all in favor of the direct solution.

3. LINEAR AND QUADRATIC MODELS

3.1. Basic Model Principle.

In the discussion of linear and quadratic models we consider one or more functions of one or more variables. We use the notation:

VAR: Number of independent variables.
FUNC: Number of dependent functions.

The basic idea behind the use of linear and quadratic models is the following. The functions we are dealing with are normally neither linear nor quadratic. The values of the functions corresponding to an actual set, xact[1:VAR], of the independent variables are obtained by calculating on a piece of computer program. We assume, that we know nothing about this piece of program, and no mathematical analysis can, therefore, be made a priori on the original functions. If, however, we calculate the original functions in a certain number of points and from these values calculate the coefficients in the model by solution of a set of linear equations, we can then use the model generated in this way for further analysis instead of analysing the original functions. Many types of analyses are very easy to carry out on the linear or quadratic models: differentiation, root determination, optimization, etc. Even use of Monte Carlo methods can be made on the models because they are much faster to evaluate than the original functions.

In the present method it is assumed that the original functions are calculated in exactly as many points as are necessary to solve the linear equations giving the unknown coefficients in the model. The methods involving more points than are strictly necessary will be dealt with in a later book on approximation methods (also including the case of models of higher order than the second degree).

3.2. Nomenclature.

The following notation is used for the variables and arrays:

integer VAR: Number of independent variables.

integer FUNC: Number of dependent functions.

integer OBS: Number of observations, i.e. the number of points at which the original functions must be calculated by the main program (or made available as input, etc.).

array xstart[1:VAR]: A set of start values of the independent variables used as basis point for the model generation.

array delOx[1:VAR]: Increments in the independent variables. The exploration of the function space is carried out in points such as:

```
xstart[i] + delOx[i]
xstart[i] - delOx[i]
```

etc., according to a fixed pattern.

array xact[1:VAR]: This is an actual set of the independent variables as generated during the function exploration.

array yact[1:FUNC]: This is an actual set of the dependent functions. The main program must deliver the values here to be processed by the method.

array MOD[1:OBS, 1:FUNC]: This array will contain the model generated by the method. The model of each function is stored as a column with OBS elements. We could also have stored the models row-wise:

```
MOD[1:FUNC, 1:OBS]
```

but as the most interesting use of the model is for an advanced optimization in which the model of all functions must be evaluated simultaneously, there is not much difference in the two storage methods.

Some simplification could have been obtained by restricting the number of functions to 1. But as the treatment of several functions is important, this is not recommended.

2.2. Model Structure.

A linear model of a function of VAR variables consists of a constant term and VAR coefficients. For VAR = 3 and FUNC = 1 we have:

(3.1) yact[1] = L[0] + L[1]×xact[1] + L[2]×xact[2] + L[3]×xact[3]

The necessary number of observations is then VAR + 1.

The quadratic model of a single function of VAR variables consists of 4 types of terms:

Type	Number of terms
Constant term	1
Linear terms	VAR
Square terms	VAR
Mixed terms	VAR×(VAR-1):2

The total number of terms is:

(VAR+2)×(VAR+1):2

This is also the value of OBS for the quadratic case. A small table of this function is given below:

VAR	(VAR+2)×(VAR+1):2
1	3
2	6
3	10
4	15
5	21
6	28
7	36
8	45
9	55
10	66

The quadratic model for VAR = 3 and FUNC = 1 contains a total of 10 coefficients:

$$
\begin{aligned}
(3.2)\quad yact[1] = {} & Q[0] + \\
& Q[1]\times xact[1] + Q[2]\times xact[2] + Q[3]\times xact[3] + \\
& Q[4]\times xact[1]\uparrow 2 + Q[5]\times xact[2]\uparrow 2 + Q[6]\times xact[3]\uparrow 2 + \\
& Q[7]\times xact[1]\times xact[2] + \\
& Q[8]\times xact[1]\times xact[3] + \\
& Q[9]\times xact[2]\times xact[3]
\end{aligned}
$$

Instead of considering the two different models:

L[0:3] and Q[0:9]

it is more convenient to use the same array:

MOD[1:OBS, 1:FUNC]

for storage of both models. The linear model is then obtained for OBS = 4 and the quadratic model for OBS = 10. In order to make it easier to translate the model handling from ALGOL to FORTRAN, the lower subscript bound has been raised from 0 to 1.

3.4. Model Generation.

The model generation consists of two phases:

1. Systematic variation of the independent variables and collection of the corresponding function values from the main program.
2. Calculation of the coefficients.

The second part can be made either in a general way by setting up a set of linear equations and solving them, or in a simplified way without solution of the equations. In the following we first discuss the general method.

The procedure is first called OBS times. In each call the procedure must generate a new set of xact-values according to a certain pattern. We then return to the main program for calculation of the corresponding function values: yact[1:FUNC]. These will then be stored by the procedure in the next call. It is convenient to have a counter, COUNT, which should be set to zero before the first call and which is increased by 1 in each call. As an example of the variation pattern we consider the case of VAR = 3, FUNC = 1, and OBS = 10 (quadratic model). To save space we write xstart[1] as x1 and del0x[1] as d1, and similarly for the other variables. The variation pattern then becomes:

COUNT before call	Values of the variables after the call			COUNT after call
	xact[1]	xact[2]	xact[3]	
0	x1	x2	x3	1
1	x1 + d1	x2	x3	2
2	x1	x2 + d2	x3	3
3	x1	x2	x3 + d3	4
4	x1 - d1	x2	x3	5
5	x1	x2 - d2	x3	6
6	x1	x2	x3 - d3	7
7	x1 + d1	x2 + d2	x3	8
8	x1 + d1	x2	x3 + d3	9
9	x1	x2 + d2	x3 + d3	10

The generation of this variation pattern requires a fairly simple algorithm, but it is not necessarily the best method from a mathematical point of view. If we consider the case of VAR=2, we get the pattern:

```
      x  x

   x  x  x

      x
```

in which only one out of the 6 points gives information about the interaction between x1 and x2. A more regular pattern in this case is a pentagon:

```
      x
   x     x
      x
    x   x
```

but this is more complicated to generate and impossible to use for VAR > 2.

The storage of the x-values and the y-values requires a matrix of the dimension:

MAT[1:OBS, 1: OBS+FUNC]

In the case VAR = 3, OBS = 10, and FUNC = 1, the content of the matrix will be the following:

1	x1	x2	x3	x1↑2	x2↑2	x3↑2	x1×x2	x1×x3	x2×x3	y1
1	x1	x2	x3	x1↑2	x2↑2	x3↑2	x1×x2	x1×x3	x2×x3	y1
1	x1	x2	x3	x1↑2	x2↑2	x3↑2	x1×x2	x1×x3	x2×x3	y1
1	x1	x2	x3	x1↑2	x2↑2	x3↑2	x1×x2	x1×x3	x2×x3	y1
1	x1	x2	x3	x1↑2	x2↑2	x3↑2	x1×x2	x1×x3	x2×x3	y1
1	x1	x2	x3	x1↑2	x2↑2	x3↑2	x1×x2	x1×x3	x2×x3	y1
1	x1	x2	x3	x1↑2	x2↑2	x3↑2	x1×x2	x1×x3	x2×x3	y1
1	x1	x2	x3	x1↑2	x2↑2	x3↑2	x1×x2	x1×x3	x2×x3	y1
1	x1	x2	x3	x1↑2	x2↑2	x3↑2	x1×x2	x1×x3	x2×x3	y1
1	x1	x2	x3	x1↑2	x2↑2	x3↑2	x1×x2	x1×x3	x2×x3	y1

Here, we have written x1, x2, x3, and y1 instead of xact[1], xact[2], xact[3], and yact[1]. These values differ, of course, from one line to the next.

Each of these OBS lines expresses the linear equation which must be fulfilled:

```
Q[0] + Q[1]×xact[1]    +   Q[2]×xact[2]    +   Q[3]×xact[3]
     + Q[4]×xact[1]↑2  +   Q[5]×xact[2]↑2  +   Q[6]×xact[3]↑2
     + Q[7]×xact[1]×xact[2]+Q[8]×xact[1]×xact[3]+Q[9]×xact[2]×xact[3]
     = yact[1]
```

or, written with the complete MOD-array:

```
MOD[1,1] +  MOD[2,1]×xact[1]    + MOD[3,1]×xact[2]    + MOD[4,1]×xact[3]
         +  MOD[5,1]×xact[1]↑2  + MOD[6,1]×xact[2]↑2  + MOD[7,1]×xact[3]↑2
         +  MOD[8,1]×xact[1]×xact[2] + MOD[9,1]×xact[1]×xact[3]
         +  MOD[10,1]×xact[2]×xact[3] = yact[1]
```

The matrix MAT[1:OBS, 1: OBS+FUNC] contains the OBS linear equations after the usual convention that the right-hand sides are stored as additional columns in the matrix.

The linear equations are now solved. If the solution is made according to the principles in the procedure LEQ1, the results will be found in the last column(s) of the original matrix:

MAT[1:OBS, OBS+1: OBS+FUNC]

and must be transferred to the MOD-array.

3.4.1. General Procedure, GENMOD1. This ALGOL procedure performs the model generation as explained above. It has the declaration:

```
integer procedure GENMOD1(count, VAR, FUNC, OBS, xstart, del0x,
xact, yact, eps, MAT, MOD);
value VAR, FUNC, OBS, eps;
integer count, VAR, FUNC, OBS;
real eps;
```

```
array xstart, del0x, xact, yact, MAT, MOD;
begin
   integer c1, i, j, k;
   real R;
   GENMOD1 := c1 := 0;
   if count > 0 then
   begin
      c1 := 1 + ((count - 1) mod OBS);
      MAT[c1, 1] := 1;
      for i := 1 step 1 until VAR do MAT[c1, 1+i] := xact[i];
      if OBS > VAR + 1 then
      begin
         for i := 1 step 1 until VAR do MAT[c1,1+VAR+i] := xact[i]↑2;
         k := 1 + 2×VAR;
         for j := 1 step 1 until VAR - 1 do
         begin
            R := xact[j];
            for i := j + 1 step 1 until VAR do
            begin
               k := k + 1;
               MAT[c1, k] := R×xact[i];
            end for i;
         end for j;
      end if quadratic;
      for i := 1 step 1 until FUNC do MAT[c1, OBS+i] := yact[i];
   end if count > 0;
   for i := 1 step 1 until VAR do xact[i] := xstart[i];
   if c1 = OBS then
   begin
      GENMOD1 := 1 - 2×LEQ1(OBS, FUNC, MAT, eps);
      for i := 1 step 1 until OBS do
      for j := 1 step 1 until FUNC do
      MOD[i, j] := MAT[i, OBS+j];
   end solution
   else
   if c1 > 0 then
   begin
```

```
            k := 1 + (c1 - 1):VAR;
            if k > 3 then k := 3;
            i := 1 + (c1 - 1) mod VAR;
            case k of
            begin
               xact[i] := xact[i] + del0x[i];
               xact[i] := xact[i] - del0x[i];
               begin
                  k := 2×VAR;
                  for i := 1 step 1 until VAR - 1 do
                  for j := 1 + i step 1 until VAR do
                  begin
                     k := k + 1;
                     if k = c1 then
                     begin
                        xact[i] := xact[i] + del0x[i];
                        xact[j] := xact[j] + del0x[j];
                        go to L1;
                     end if k = c1;
                  end for i, j;
               end case 3;
            end case;
L1:      end if c1 > 0;
         count := count + 1;
      end GENMOD1;
```

The procedure GENMOD1 is for illustration purpose only. It operates on core-stored variables. The system of linear equations is stored in the array MAT[1: OBS, 1:OBS+FUNC]. The solution is made with LEQ1 and the result is transferred to the array MOD[1: OBS, 1: FUNC].

It is assumed that OBS has either the value VAR+1 (linear case) or (VAR+1)×(VAR+2):2 (quadratic case). Intermediate values are not permitted.

Note, that the storage of the x-values in MAT is not made immediately after generation of a new set of x-values, but in the next call of GENMOD1. This means that the y-values are stored simultaneously with the corresponding x-values, and the main program is permitted to change the

x-values, if required. This change may become necessary if the range of the x-values is restricted, and could, of course, also have been included in the procedure. If changes are made, the general pattern of the variation must be retained, otherwise the matrix may become ill-conditioned and the equations cannot be solved.

3.4.2. Simplified Procedure, GENMOD1A. As mentioned above, it is possible to avoid the solution of the OBS linear equations. This gives a faster and smaller program and a greater accuracy. The price for this is that the variation pattern must be strictly adhered to, and it is not even permitted to change the size of the increments in some of the function evaluations.

The ALGOL procedure, GENMOD1A, operates after the simplified method. The parameters eps and MAT are now omitted from the parameter list as they are not required. The variation of the x-values is made in exactly the same way as in GENMOD1. In each call the incoming y-values are treated as much as possible and are stored in the MOD-array. The treatment can be divided into the following five phases:

Phase 1: For count = 1 the y-values for the base point of the model are stored as the constant term in the model:

(3.3)
```
MOD[1,1] = yact[1]
MOD[1,2] = yact[2]
.....
.....
MOD[1,FUNC] = yact[FUNC]
```

In the following we assume that FUNC = 1 and consider only column 1 in the MOD-array.

Phase 2: In the next VAR calls, one of the variables has been given a positive increment. If this is variable no. N, we have:

(3.4) $\quad count = 1 + N$

and the corresponding linear coefficient is preliminarily calculated as:

(3.5) $\quad MOD[1+N,1] = yact[1] - MOD[1,1]$

The linear part of the model including the constant term now have values that assume that the x-values are measured relative to the base point. They are further corrected in phase 3 and phase 5. GENMOD1A is constructed to generate quadratic models only, not linear models, so we proceed with:

Phase 3: In the next VAR calls, one of the variables has been given a negative increment. Let this be variable no. N, so that we have:

(3.6) count = 1 + VAR + N

If we consider only the linear and the quadratic terms in the model, we have for variable no. N:

(3.7) Y = BASE + ...LIN[N]×xact[N]... + ...QUA[N]×xact[N]↑2...

When the variable is measured relative to the base point, we can write it as del0x[N], and we get 3 equations for the determination of BASE, LIN[N], and QUA[N]:

(3.8)
YBAS = BASE
YLIN = BASE + LIN[N]×del0x[N] + QUA[N]×del0x[N]↑2
YQUA = BASE - LIN[N]×del0x[N] + QUA[N]×del0x[N]↑2

From this we find:

(3.9) QUA[N] = ((YLIN-YBAS)+(YQUA-YBAS))/(2×del0x[N]↑2)

For function no. 1, the ALGOL statement corresponding to this takes the form:

(3.10) MOD[count,1] := (MOD[k,1]+yact[1]-MOD[1,1])/(2×del0x[jx]↑2)

in which k = 1 + N and jx = N. Note, that MOD[k,1] contains YLIN - YBAS according to (3.5). The improved value of LIN[N] alias MOD[k,1] becomes:

(3.11) LIN[N] = ((YLIN-YBAS)-(YQUA-YBAS))/(2×del0x[N])

and the correct form of this is:

(3.12) MOD[k,1] := (MOD[k,1]-(yact[1]-MOD[1,1]))/(2×del0x[jx])

Phase 4: In the remaining calls of the procedure, the variables no. ix and no. jx have been given a positive increment and the others are at their base value. When the variables are measured relative to the base point, the equation determining the mixed coefficient takes the form:

(3.13) YMIX = BASE + LIN[ix]×del0x[ix] + LIN[jx]×del0x[jx] +
QUA[ix]×del0x[ix]↑2 + QUA[jx]×del0x[jx]↑2 +
MIX[count]×del0x[ix]×del0x[jx]

When this equation is solved with respect to MIX[count] and the now known values for LIN and QUA are inserted, we finally get the expression:

(3.14) MOD[count,1] := (yact[1]-MOD[1,1]-
(MOD[1+ix,1]+MOD[1+VAR+ix,1]×del0x[ix])×del0x[ix]-
(MOD[1+jx,1]+MOD[1+VAR+jx,1]×del0x[jx])×del0x[jx])
/(del0x[ix]×del0x[jx]);

Phase 5: This is the final transformation so that the x-values are absolute instead of relative to the base point. For the relative x-values, del0x[N], the linear and quadratic part of the model for variable no. N has the form:

(3.15) Y = BASE + ...LIN[N]×del0x[N]...+...QUA[N]×del0x[N]↑2...

Insertion of xact[N]-xstart[N] instead of del0x[N] gives:

(3.16) Y = BASE +...LIN[N]×(xact[N]-xstart[N])...+...
QUA[N]×(xact[N]-xstart[N])↑2...

From this we see, that when xact[N] is to be used in the model instead of del0x[N], the constant term, BASE, must be corrected by subtraction of LIN[N]×xstart[N] and addition of QUA[N]×xstart[N]↑2. No correction is necessary in the quadratic coefficient, but because of the

double product:

(3.17) -2×QUA[N]×xact[N]×xstart[N]

we must subtract 2×QUA[N]×xstart[N] from LIN[N]. The correction of BASE and LIN for function no. j is made as follows:

```
k := 2;
for i := 1 step 1 until VAR do
begin
   MOD[1,j] := MOD[1,j]-(MOD[k,j]-xstart[i]×MOD[k+VAR,j])×xstart[i];
   MOD[k,j] := MOD[k,j]-2×MOD[k+VAR,j]×xstart[i];
   k := k+1;
end for i;
```

For the mixed terms:

(3.18) MIX[count]×del0x[ix]×del0x[jx]

insertion of xact[N] - xstart[N] gives:

(3.19) MIX[count]×(xact[ix]-xstart[ix])×(xact[jx]-xstart[jx])

From this we can see that it is necessary to add:

(3.20) MIX[count]×xstart[ix]×xstart[jx]

to the constant term, and to subtract:

MIX[count]×xstart[jx] from LIN[ix] and
MIX[count]×xstart[ix] from LIN[jx]

These are the final corrections required, and their exact form can be found in the declaration of GENMOD1A which is reproduced here:

```
integer procedure GENMOD1A(count, VAR, FUNC, OBS, xstart, del0x,
xact, yact, MOD);
value VAR, FUNC, OBS;
integer count, VAR, FUNC, OBS;
array xstart, del0x, xact, yact, MOD;
begin
   integer j, ix, jx, k, state;
   GENMOD1A := 0;
   jx := count-1;
   for ix := 1 step 1 until VAR do xact[ix] := xstart[ix];
   if count > 0 then
   begin
      state := if count = 1 then 1 else 2 +(count-2):VAR;
      if state = 3 then jx := jx - VAR else
      if state = 4 then
      begin
         k := 2×VAR + 2;
         for ix := 1 step 1 until VAR -1 do
         for jx := ix + 1 step 1 until VAR do
         begin
            if k = count then go to L1;
            k := k + 1;
         end for ix, jx;
L1:   end if state = 4;
      k := count - VAR;
      for j := 1 step 1 until FUNC do
      case state of
      begin
         MOD[1,j] := yact[j];
         MOD[count,j] := yact[j] - MOD[1,j];
         begin
            MOD[count,j] := (MOD[k,j]+yact[j]-MOD[1,j])/(2×del0x[jx]↑2);
            MOD[k,j] := (MOD[k,j]-yact[j]+MOD[1,j])/(2×del0x[jx]);
         end state = 3;
         MOD[count,j] := (yact[j]-MOD[1,j]-(MOD[ix+1,j]+MOD[ix+VAR+1,j]×
         del0x[ix])×del0x[ix]-(MOD[jx+1,j]+MOD[jx+VAR+1,j]×del0x[jx])×
         del0x[jx])/(del0x[ix]×del0x[jx]);
      end case and for;
```

```
      jx := jx + 1;
      if state > 1 then state := state -1;
      if jx > VAR then
      case state of
      begin
         begin
            state := 2;
            jx := 1;
         end transfer to state 2;
         begin
            if VAR = 1 then go to L2;
            state := 3;
            ix := 1;
            jx := 2;
         end transfer to state 3;
         begin
            ix := ix + 1;
            jx := ix + 1;
            if jx > VAR then go to L2;
         end new ix in state 3;
      end case;
      if state = 2 then
      xact[jx] := xstart[jx] - del0x[jx] else
      xact[jx] := xstart[jx] + del0x[jx];
      if state = 3 then
      xact[ix] := xstart[ix] + del0x[ix];
      go to L3;
L2:   for j := 1 step 1 until FUNC do
      begin
         k := 2;
         for i := 1 step 1 until VAR do
         begin
            MOD[1,j] := MOD[1,j]-(MOD[k,j]-xstart[i]×MOD[k+VAR,j])×xstart[i];
            MOD[k,j] := MOD[k,j]-2×MOD[k+VAR,j]×xstart[i];
            k := k + 1;
         end for i;
         k := k + VAR;
         for ix := 1 step 1 until VAR -1 do
         for jx := ix + 1 step 1 until VAR do
         begin
```

```
            MOD[1,j] := MOD[1,j] + MOD[k,j]×xstart[ix]×xstart[jx];
            MOD[ix+1,j] := MOD[ix+1,j]-MOD[k,j]×xstart[jx];
            MOD[jx+1,j] := MOD[jx+1,j]-MOD[k,j]×xstart[ix];
            k := k + 1;
          end for ix, jx;
        end for j;
        GENMOD1A := 1;
    end if count > 0;
L3:count := count + 1;
end GENMOD1A;
```

3.5. Model Evaluation.

The evaluation of the value of the linear or quadratic model, MOD[1: OBS, 1: FUNC], for a given set of x-values, xact[1: VAR], requires a very simple algorithm. The result is delivered as the array, yact[1: FUNC]. If the model is quadratic, it should be possible to evaluate the model also for the linear case simply by using the smaller value of OBS.

It can be convenient to include differentiation of the model in the evaluation procedure. For VAR = 3, OBS = 10, and FUNC = 1, we can calculate the three derivatives:

dyact[1]/dx[1]
dyact[1]/dx[2]
dyact[1]/dx[3]

The evaluation procedure can be fitted with a formal parameter, n, which for n = 0 gives the normal y-value and for n > 0 gives the derivative with respect to the independent variable no. n. When the normal y-value is found from:

```
yact[1] :=
MOD[1,1] +  MOD[2,1]×xact[1]    + MOD[3,1]×xact[2]    + MOD[4,1]×xact[3]
         +  MOD[5,1]×xact[1]↑2 + MOD[6,1]×xact[2]↑2 + MOD[7,1]×xact[3]↑2
         +  MOD[8,1]×xact[1]×xact[2] + MOD[9,1]×xact[1]×xact[3]
         +  MOD[10,1]×xact[2]×xact[3];
```

the formulas for the three derivatives become:

dyact[1]/dx[1] =
MOD[2,1]+2×MOD[5,1]×xact[1]+MOD[8,1]×xact[2]+MOD[9,1]×xact[3]

dyact[1]/dx[2] =
MOD[3,1]+MOD[8,1]×xact[1]+2×MOD[6,1]×xact[2]+MOD[10,1]×xact[3]

dyact[1]/dx[3] =
MOD[4,1]+MOD[9,1]×xact[1]+MOD[10,1]×xact[2]+2×MOD[7,1]×xact[3]

The ALGOL procedure, MODVAL1, shown below performs the model evaluation or differentiation. It operates directly on the MOD-array stored in the core.

```
real procedure MODVAL1(VAR, FUNC, OBS, n, MOD, xact, yact);
value VAR, FUNC, OBS, n;
integer VAR, FUNC, OBS, n;
array MOD, xact, yact;
begin
   boolean quad;
   integer i, j, k, m;
   real R1, R2;
   quad := OBS > VAR + 1;
   for j := 1 step 1 until FUNC do
   yact[j] := MOD[1+n, j];
   if n = 0 then
   begin
      for i := 1 step 1 until VAR do
      begin
         R1 := xact[i];
         for j := 1 step 1 until FUNC do
         yact[j] := yact[j] + MOD[1+i, j]×R1;
      end for i;
      if quad then
      begin
         for i := 1 step 1 until VAR do
         begin
            R1 := xact[i]↑2;
```

```
            for j := 1 step 1 until FUNC do
            yact[j] := yact[j] + MOD[1+VAR+i, j]×R1;
         end for i;
      end if quad;
   end if n = 0 else
   if quad then
   begin
      R1 := 2×xact[n];
      for j := 1 step 1 until VAR do
      yact[j] := yact[j] + MOD[1+VAR+n, j]×R1;
   end if quad and n > 0;
   if quad then
   begin
      i := 2×(VAR + 1);
      for m := 1 step 1 until VAR - 1 do
      begin
         R1 := xact[m];
         for k := m + 1 step 1 until VAR do
         begin
            R2 := xact[k];
            for j := 1 step 1 until FUNC do
            begin
               if n = 0 then
               yact[j] := yact[j] + MOD[i, j]×R1×R2 else
               if m = n ∨ k = n then
               yact[j] := yact[j]+MOD[i, j]×(if m=n then R2 else R1);
            end for j;
            i := i + 1;
         end for k;
      end for m;
   end if quad;
   MODVAL1 := yact[1];
end MODVAL1;
```

3.6. Use of Models.

Examples of the use of quadratic models are given in Chapter 5 for quadratic optimization, i.e. determination of the maximum or minimum of a function. The models can also be used to find a point where a set of functions becomes zero. This is further discussed in Chapter 4 which describes a general procedure, NOLEQ8, which generates a linear model of a set of functions and finds the zero point. If a linear model is available already, the zero point determination (root determination) can be made immediately with only very little extra work.

For root determination we will normally have as many functions as there are independent variables (VAR = FUNC). The linear model of the VAR functions then has the size:

```
MOD[1:VAR+1, 1:VAR]
```

For VAR = 3 we have MOD[1: 4, 1: 3] and the model is evaluated as:

```
yact[1] := MOD[1,1]+MOD[2,1]×xact[1]+MOD[3,1]×xact[2]+MOD[4,1]×xact[3]
yact[2] := MOD[1,2]+MOD[2,2]×xact[1]+MOD[3,2]×xact[2]+MOD[4,2]×xact[3]
yact[3] := MOD[1,3]+MOD[2,3]×xact[1]+MOD[3,3]×xact[2]+MOD[4,3]×xact[3]
```

The point where all three functions become zero is determined from the three linear equations:

```
MOD[1,1]+MOD[2,1]×xact[1]+MOD[3,1]×xact[2]+MOD[4,1]×xact[3] = 0
MOD[1,2]+MOD[2,2]×xact[1]+MOD[3,2]×xact[2]+MOD[4,2]×xact[3] = 0
MOD[1,3]+MOD[2,3]×xact[1]+MOD[3,3]×xact[2]+MOD[4,3]×xact[3] = 0
```

If we wish to write this after the conventions for LEQ1, the equations become:

```
MOD[2,1]×xact[1]+MOD[3,1]×xact[2]+MOD[4,1]×xact[3] = -MOD[1,1]
MOD[2,2]×xact[1]+MOD[3,2]×xact[2]+MOD[4,2]×xact[3] = -MOD[1,2]
MOD[2,3]×xact[1]+MOD[3,3]×xact[2]+MOD[4,3]×xact[3] = -MOD[1,3]
```

The augmented matrix defining the system of equations has the size:

MAT[1:VAR, 1: VAR+1]

or MAT[1:3, 1:4] for VAR = 3. If we compare MOD and MAT we see, that it is necessary to transpose MOD and to move the columns cyclically so that the column number is reduced by one and the first column becomes the last column with change of sign. The change from MOD to MAT can be made with the algorithm:

```
for i := 1 step 1 until VAR do
begin
   for j := 1 step 1 until VAR do
   MAT[i, j] := MOD[j+1, i];
   MAT[i, i+1] := -MOD[1, i];
end for i;
```

Solution of the set of equations gives the required set of roots, xact[1: VAR].

If a quadratic model is available, we can carry out the root determination in one of the following ways:

1. Approximate the quadratic model with a linear model and proceed as above.

2. Use Newton-Raphson method with calculation of the derivatives from the quadratic model.

3. Direct elimination with removal of one of the equations at a time by solution of the corresponding second order degree equations.

4. SOLUTION OF NON-LINEAR EQUATIONS

4.1. Basic Principles.

A very frequent practical numerical problem consists in finding out when a given function assumes the value zero. If we have a function of a single variable, Y = F(X), we may wish to know for which value or values of X we get:

$$F(X) = 0 \tag{4.1}$$

When such a value of X has been found, we say that we have found a root in the equation F(X) = 0.

We may also have several simultaneous equations with as many unknown X-values:

$$\begin{aligned} &F(X1, X2, X3) = 0 \\ &G(X1, X2, X3) = 0 \\ &H(X1, X2, X3) = 0 \end{aligned} \tag{4.2}$$

Here, we must find a set of numbers: X1, X2, X3, so that the three functions: F, G, and H all become zero. If the three functions are linear expressions in the independent variables:

$$\begin{aligned} F &= A0 + A1\times X1 + A2\times X2 + A3\times X3 \\ G &= B0 + B1\times X1 + B2\times X2 + B3\times X3 \\ H &= C0 + C1\times X1 + C2\times X2 + C3\times X3 \end{aligned} \tag{4.3}$$

we have three linear equations in three unknowns, and we may solve them as described in Chapter 2. However, if the functions are not linear, e.g.:

$$F = A0 + A1\times X1\uparrow 2 + A2\times \sin(X2) + A3\times X1\times X3 \tag{4.4}$$

and similar complicated forms for F and G, we say that we have a set of non-linear equations that must be solved.

In the following sections we first discuss functions of a single variable (section 4.2 and 4.3) and then several functions in several variables (section 4.4).

In very special cases it is possible to give an analytical solution to a non-linear equation. This applies to quadratic equations:

$$A \times X\uparrow 2 + B \times X + C = 0 \qquad (4.5)$$

and to cubic equations:

$$A \times X\uparrow 3 + B \times X\uparrow 2 + C \times X + D = 0 \qquad (4.6)$$

The analytical solutions to these equations are given in section 4.2 below. Quartic equations can also be solved in this way, but the formula system is very complicated.

In most practical cases the solution of non-linear equations can only be carried through by use of iteration. An initial guess on the solution is made, the functions are explored in the vicinity of the first guess, and from this information the solution can be improved. Section 4.3 contains examples of iterative procedures for a single unknown and section 4.4 a procedure for solution of several simultaneous non-linear equations.

As an example of the occurrence of a set of non-linear equations in the field of chemical engineering we can consider the case of an adiabatic reformer used for production of synthesis gas. The three unknowns in equation (4.2) above could then have the significance:

(4.7)
X1: Amount of hydrocarbon
X2: Amount of air
X3: Amount of enriched oxygen

The three function values could be:

(4.8)
F: Error in heat balance
G: Error in hydrogen-nitrogen ratio
H: Error in amount of produced gas

If we assume a set of start values of X1, X2, and X3, and have available a suitable program for calculation of F, G, and H, the iterative calculation then consists in the systematic variation of X1, X2, and X3 until F, G, and H become zero within a given tolerance.

4.2. Quadratic and Cubic Equations.

4.2.1. Quadratic Equations, Procedure QUAREQ2. The quadratic equation has the form:

(4.9) $A \times X\uparrow 2 + B \times X + C = 0$

The solution method can be found in elementary mathematical textbooks, etc. See Hodgman (1956), p. 295. The formula for the roots is:

(4.10) $x1, x2 = (-B \pm \text{sqrt}(B\uparrow 2 - 4 \times A \times C))/(2 \times A)$

The formula can only be used for $A \neq 0$. If we have $A = 0$, the quadratic equation has degenerated into a linear equation:

(4.11) $B \times X + C = 0$

For $B \neq 0$ this has the solution:

(4.12) $X = -C/B$

For $B = 0$, however, no solution of X can be found. In the general case of $A \neq 0$, the formula (4.10) can only be evaluated if the discriminant D:

(4.13) $D = B\uparrow 2 - 4 \times A \times C$

is non-negative. For $D < 0$, the two roots are complex.

The procedure QUAREQ2 has the following parameters:

real A, B, and C. These are the coefficients in equation (4.9).

real x1 and x2. These are the two roots calculated by the procedure. If $A = 0$ and $B \neq 0$, the X-value calculated from equation (4.12) is assigned to both x1 and x2.

integer QUAREQ2. The procedure is of type integer and takes the value 0, if there are two real roots, x1 and x2 (they may be identical). QUAREQ2 = 1 indicates the complex case. No values are then assigned to x1 and x2.

The declaration of QUAREQ2 is shown on the next page.

```
integer procedure QUAREQ2(A, B, C, x1, x2);
value A, B, C;
real A, B, C, x1, x2;
begin
   real D;
   QUAREQ2 := 0;
   if A = 0 then
   begin
      if B = 0 then QUAREQ2 := 1 else
      x1 := x2 := -C/B;
   end A = 0 else
   begin
      D := B↑2 - 4×A×C;
      if D < 0 then QUAREQ2 := 1 else
      begin
         x1 := if B < 0 then -1 else 1;
         x1 := (-B - x1×sqrt(D))/2/A;
         x2 := C/A/x1;
      end D ≥ 0;
   end A ≠ 0;
end QUAREQ2;
```

The calculation follows the explanation given above as regards the zero check of A and B and the sign check of D. Note, that only one of the roots is found from equation (4.10), namely the root for which -B has the same sign as ±sqrt... In this way we avoid the loss in accuracy that would occur, when -B and ±sqrt... are of equal magnitude but opposite sign. In the example:

(4.14) $A = 0.001,\ B = 1,\ C = 0.001$

we must calculate:

$$-1 \pm \mathrm{sqrt}(1 - 4\times0.001\times0.001) = -1 \pm \mathrm{sqrt}(0.999996)$$

Use of the plus sign here will give low accuracy. Only the minus sign is used, and we find the second root from the known root product:

(4.15) $x2 := C/A/x1$

4.2.2. Cubic Equations, Procedure CUBEQ1. In a cubic equation:

(4.16) $A \times X\uparrow 3 + B \times X\uparrow 2 + C \times X + D = 0$

there will always be three roots. One of these will always be real and the remaining two roots will be either real or complex. As a special case, two or three of the real roots may be identical.

An analytic solution method for cubic equations can also be found in many mathematical books. The following explanation is taken from Hodgman (1956), p. 295. Division by A gives the form:

(4.17) $X\uparrow 3 + p \times X\uparrow 2 + q \times X + r = 0$

We then transform the equation into:

(4.18) $x\uparrow 3 + a \times x + b = 0$

by substituting for X the value: x - p/3. The two coefficients, a and b, become:

(4.19) $a = q - p\uparrow 2/3$

(4.20) $b = (2 \times p\uparrow 3 - 9 \times p \times q + 27 \times r)/27$

The further actions depend on the value of the discriminant:

(4.21) $DD = b\uparrow 2/4 + a\uparrow 3/27$

Three cases are considered:

DD > 0: One real root and two complex roots.
DD = 0: Three real roots of which two at least are equal.
DD < 0: Three real and unequal roots.

In the first case we can calculate the single real root by performing the calculations:

(4.22) SQ = sqrt(DD)

(4.23) AA = cubrt(-b/2 + SQ)

(4.24) BB = cubrt(-b/2 - SQ)

cubrt(x) is the cube root of x (with sign). The root in equation (4.18) becomes AA+BB and in the original equation (4.17) or (4.16):

(4.25) x1 = AA + BB - p/3

In the second case (DD = 0) we calculate the two remaining real and equal roots as:

(4.26) (x2,x3) = -AA-p3

(AA and BB are equal).

In the third case (DD < 0) the three real roots are found by first calculating an angle, fi, having a cosine given by:

(4.27) cosfi = -b/2/sqrt(-a↑3/27)

Note, that a < 0. When cosfi is known, we can calculate fi itself via the standard arctan function:

(4.28) fi = arctan(sqrt(1-cosfi↑2)/cosfi)

For cosfi = 0 this formula gives division by zero. We must then use the correction:

(4.29) fi = <u>if</u> cosfi = 0 <u>then</u> pi/2 <u>else</u> arctan.....

An additional correction is neccesary, because arctan will deliver an angle in the range from -pi/2 to +pi/2, whereas we require fi to be in the range from 0 to pi. This can be corrected by:

(4.30) <u>if</u> cosfi < 0 <u>then</u> fi := fi + pi

performed after calculation of fi. We also need one third of fi:

(4.31) fi3 = fi/3

The three roots can then be written as:

$$
\begin{aligned}
x1 &= fac\times\cos(fi3) - p3 \\
x2 &= fac\times\cos(fi3 + 2\times pi/3) - p3 \qquad (4.32)\\
x3 &= fac\times\cos(fi3 + 4\times pi/3) - p3
\end{aligned}
$$

in which the factor, fac, is:

$$fac = 2\times sqrt(-a/3) \qquad (4.33)$$

The declaration of the procedure CUBEQ1 performing these calculation is shown below:

```
procedure CUBEQ1(A, B, C, D, x1, x2, x3, comp);
value A, B, C, D;
real A, B, C, D, x1, x2, x3;
boolean comp;
begin
   real a, b, AA, BB, DD, p, q, r, SQ, cosfi, fac, fi3, p3;
   real procedure cubrt(x);
   value x;
   real x;
   cubrt := sign(x)×(abs(x))↑(1/3);
   p := B/A;
   p3 := p/3;
   q := C/A;
   r := D/A;
   a := q - p↑2/3
   b := (2×p↑3 - 9×p×q + 27×r)/27;
   DD := b↑2/4 + a↑3/27;
   comp := DD > 0;
   if DD ≥ 0 then
   begin
      SQ := sqrt(DD);
      AA := cubrt(-b/2 + SQ);
      BB := cubrt(-b/2 - SQ);
      x1 := AA + BB - p3;
      if ¬ comp then x2 := x3 := -AA - p3;
   end
   else
```

```
begin
      cosfi := -b/2/sqrt(-a↑3/27);
      fi3 := if cosfi = 0 then 0.52359878 else
      arctan(sqrt(1-cosfi↑2)/cosfi)/3;
      if cosfi < 0 then fi3 := fi3 + 1.04719755;
      fac := 2×sqrt(-a/3);
      x1 := fac×cos(fi3) - p3;
      x2 := fac×cos(fi3 + 2.0943951) - p3;
      x3 := fac×cos(fi3 + 4.1887902) - p3;
   end
end CUBEQ1;
```

The formal parameters are the four coefficients: A, B, C, and D, the three required roots: x1, x2, and x3, and the boolean: comp. If there are three real roots, they will be calculated and delivered as x1, x2, and x3. If there is only one real root, it will be calculated and stored as x1, and comp is set to true. With three real roots, comp is set to false.

No special precautions have been made to secure good numerical accuracy in CUBEQ1 in a similar way as was done for QUAREQ2.

A simple way of making a rough test on a subroutine is to generate random values of its parameters and perform the calculations so that the calculation results can be compared with the known exact solution. This was used for CUBEQ1. A series of 1000 calls of CUBEQ1 were made. For each call the first coefficient, A, was generated randomly in the range from 1 to 10, and three random roots were generated in the range from -10 to +10. The three other constants, B, C, and D, were calculated from the roots and A.

CUBEQ1 was then called, and the calculated roots were compared to the known roots. A print-out was made as soon as a deviation was larger than the previous maximum deviation. The following results were found:

Call no.	Deviation
1	0.0000003
14	0.0000004
30	0.0000004
38	0.0000018
43	0.0000033
67	0.0000490
284	0.0000919
922	0.0005293

A further inspection of these examples showed that the large deviations occurred when two or three of the roots were very nearly equal. The roots in call no. 922 were:

	Root 1	Root 2	Root 3
Original:	7.0969418	7.1177271	7.4167755
Calculated:	7.0974364	7.1171978	7.4168102
Deviation:	-0.0004946	0.0005293	-0.0000347

The origin of these inaccuracies is to be found in equation (4.28). When fi is close to 0 or pi, cosfi will be close to ± 1, and digits are lost in the evaluation of sqrt(1-cosfi↑2). The values fi = 0 and pi give fi3 = 0 and pi/3 for which two of the angles:

$$fi3,\ fi3 + 2\times pi/3,\ fi3 + 4\times pi/3$$

will have cos-values that are equal. Hence the trouble for nearly equal roots.

The accuracy can be improved, if one of the roots is calculated from the known sum of the roots (-B). This is used in the example given by Kallin (1969), p. 130.

The test of procedures with random numbers can often reveal errors of different kinds, but it is, of cource, not an exhaustive test.

4.3. Iterative Methods for a Single Unknown.

The procedures given in section 4.2 for solution of quadratic equations and cubic equations have the advantage that no iteration is necessary. We always get an answer after using the formulas just once. On the other hand, the formulas are not quite simple, especially for cubic equations.

It is also possible to give a set of formulas for the solution of equations of the fourth degree without iteration, but these are still more complicated. Another drawback in these formulas is the risk of numerical troubles. A thorough programming of the formulas so that this risk is reduced can be a very complicated job. In this situation the use of iterative methods is much to be preferred.

As a further illustration of the two possible approaches to the determination of roots:

1. Complicated analytic methods
2. Simple iterative methods

we consider a typical calculation problem from chemical engineering. The heat content (enthalpy) of a gas in the ideal state can be approximated as a temperature polynomial. Fourth order polynomials in the absolute temperature (deg. K) have proved efficient in most cases. The enthalpy of formation of methane can be written as the polynomial:

$$(4.34) \qquad H(T) = 665.17 + 3.36\times T + 8.50_{10}-3\times T\uparrow 2 - 7.11_{10}-7\times T\uparrow 3 - 2.55_{10}-10\times T\uparrow 4$$

(Not all significant digits are reproduced here). When T is known, the enthalpy, H(T), can be calculated from this formula. The inverse problem of finding T when H is known, is often encountered in practice. When a gas is heated in a reactor, the enthalpy increase is found from a heat balance, and the corresponding temperature must then be found by a root determination in formula (4.34). This has the form:

$$(4.35) \qquad Y = F(X) = A0 + A1\times X + A2\times X\uparrow 2 + A3\times X\uparrow 3 + A4\times X\uparrow 4$$

We wish to calculate the temperature, X, corresponding to a given enthalpy, Y0. We must find X so that:

(4.36) $F(X) = Y0$

or, X must be found as a root in the equation:

(4.37) $F(X) - Y0 = 0$

When F(X) is a polynomial of the fourth degree in X and Y0 has a known value, we must solve an equation of fourth degree. As an example we can take:

$Y0 = 16580$ (kcal/kgmole)

which corresponds to a temperature about 1270 deg. K. Insertion of this value in the formulas gives:

(4.38) $-2.55_{10}-10 \times X\uparrow 4 - 7.11_{10}-7 \times X\uparrow 3 + 8.50_{10}-3 \times X\uparrow 2 + 3.36 \times X - 15914.83 = 0$

A fourth degree equation may have up to four real roots, and this is actually the case here. The four roots are:

-7066.86
-1530.67
1273.35
4533.47

One of these roots (1273.35) is the one we are interested in. The three other roots are very far from the range for which the polynomial was calculated (300 - 1500 deg. K), and they are of no interest to us here. It would have been uneconomical to use a method for a complete solution of a fourth degree equation giving all four roots and then discard the three wrong roots. Here, the iterative method is much to be preferred.

4.3.1. Newton-Raphson Method. The basic principle in most of the iterative methods for root determination is the so-called Newton-Raphson method which is illustrated on figure 1, page 54. See also Frøberg, page 19 (1966).

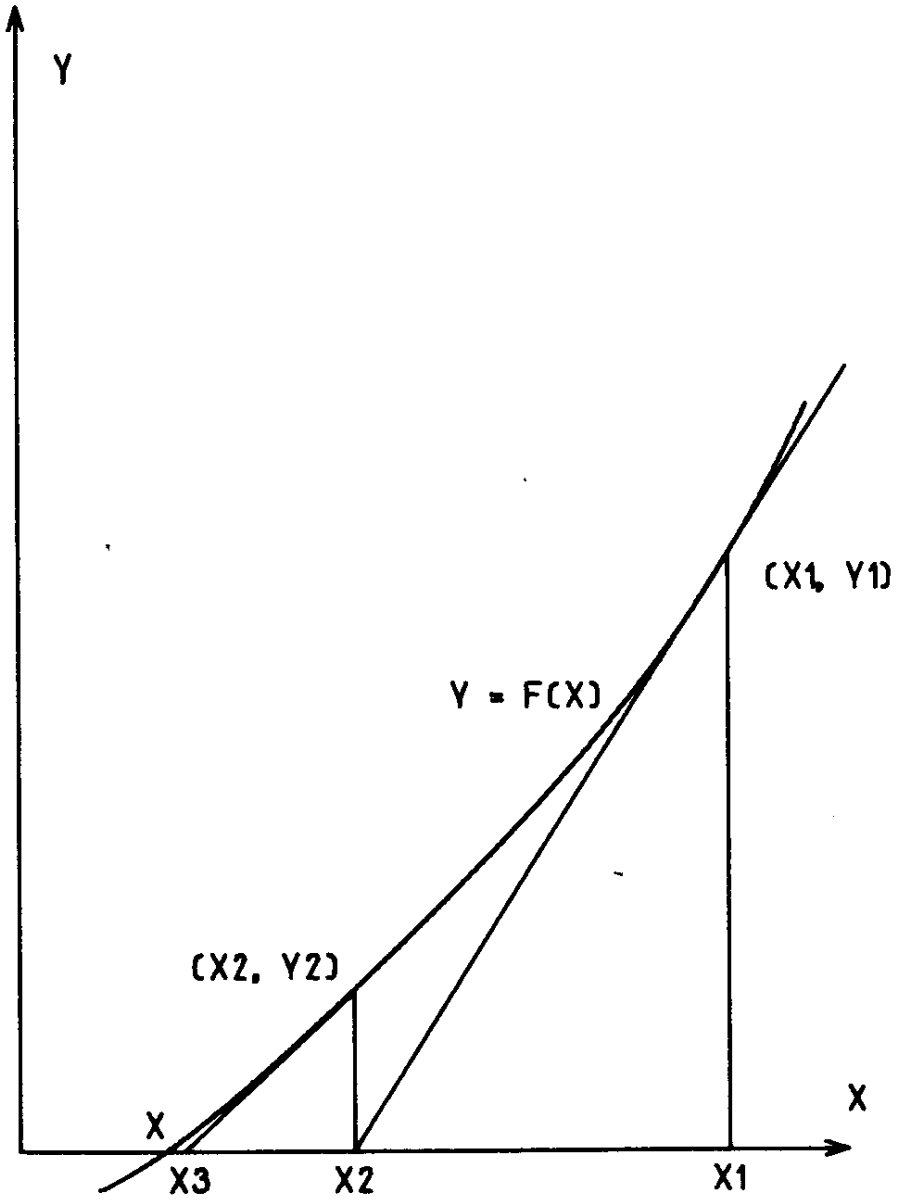


Figure 1

Newton-Raphson Method

Figure 1 shows a function, Y = F(X), and a point (X1,Y1) on the curve. We want to find the root, i.e. the point where the curve intersects the X-axis. In the Newton-Raphson method we use the tangent in the point (X1,Y1) as an approximation to F(X). If we denote the derivative in this point alfa:

(4.39) alfa = dF/dX

the equation for the tangent becomes:

(4.40) Y = Y1 + alfa×(X-X1)

We put Y = 0 and find X from the equation. This gives the next approximation, X2, to the required root:

(4.41) X2 = X1 - Y1/alfa

We may also express the result as the increment, DELX, which must be added to X1 to give X2:

(4.42) DELX = -Y1/alfa

In the point X2 we must then calculate the function value:

(4.43) Y2 = F(X2)

and the new value of alfa in X2. Y2 and the new alfa are inserted in (4.42), giving a new DELX which is added to X2, etc. The iteration is continued until DELX becomes sufficiently small.

In this version of the method we must for each new X-value calculate the function value, Y, and the derivative, dY/dX. It is easy to calculate the derivative if the function is a polynomial, but if F(X) is a very complicated expression, maybe a large procedure or a piece of program, we can only find dY/dX by means of a complicated numerical method.

As it is not necessary to calculate the derivative with a great accuracy, it may be sufficient to calculate the difference quotient:

(4.44) DIFQ = (Y2-Y1)/(X2-X1)

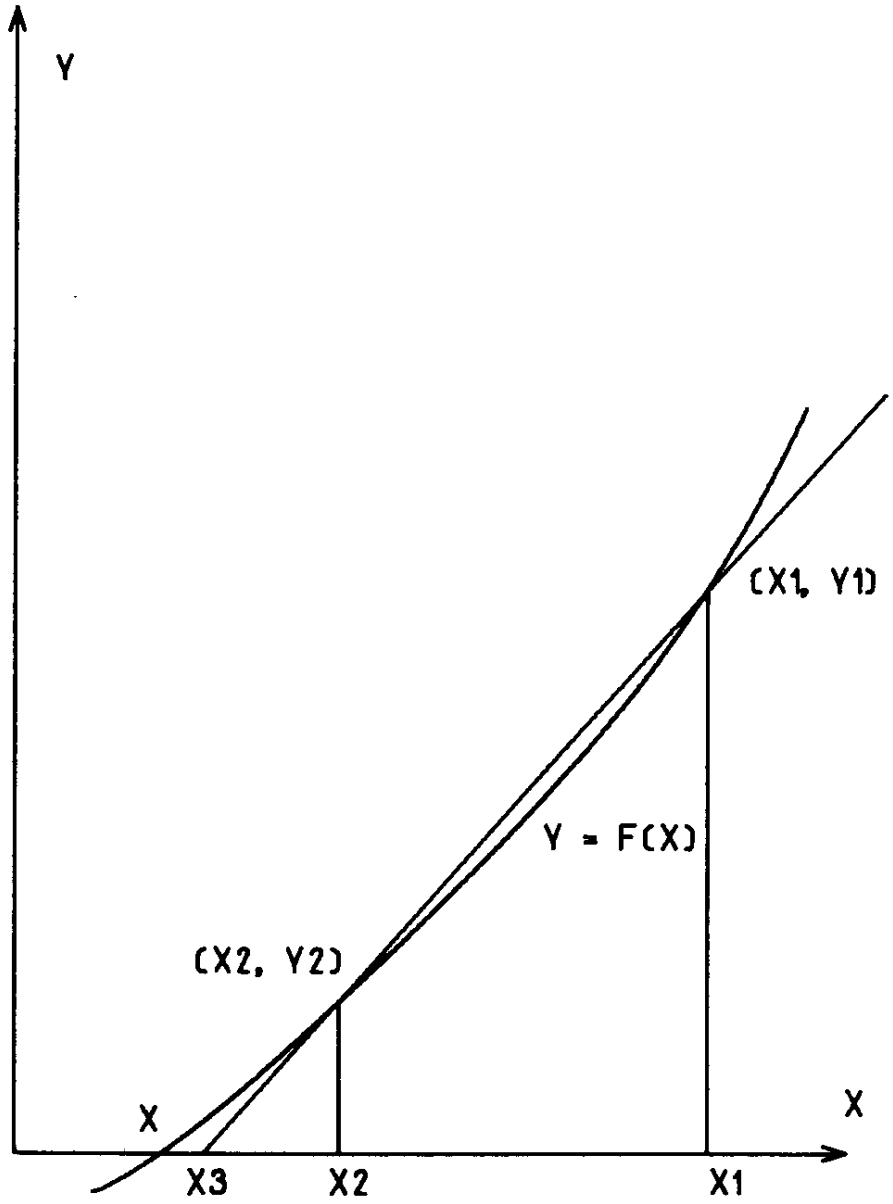


Figure 2

Regula falsi Method

When DIFQ is used as an approximation to alfa, the method is called that of regula falsi and is illustrated in figure 2, page 56. As we do not calculate the difference quotient in the point X1, it becomes necessary to select the point X2 in an arbitrary way in order to get started. The increment, DELX, in X from X2 to X3 is then calculated as:

(4.45) DELX = -Y2/alfa

in which we insert DIFQ from equation (4.44) instead of alfa and get:

(4.46) DELX = -Y2/(Y2-Y1)×(X2-X1)

As X2-X1 is the previous value of DELX, we may also write:

(4.47) DELX = DELX×YNEW/(YOLD-YNEW)

Here, YNEW is the last calculated Y-value (Y2) and YOLD is the previously calculated Y-value (Y1).

In general, the convergence of the regula falsi method will be somewhat slower than that of the Newton-Raphson method. The reason for this is that two function values are required before the iteration can start, and the difference quotient is less accurate than the derivative. On the other hand, the Newton-Raphson method requires the evaluation of two expressions in each step (function and derivative) where the regula falsi method evaluates the function only. In most practical cases the regula falsi method has a satisfactory convergence speed.

4.3.2. The Procedure ROOT5. This procedure uses the regula falsi calculation method. It has four formal parameters:

real y. The function expression for which we require the root. This has normally the form of a real procedure.

real x. The independent variable. The main program must have inserted a start value here before the call of ROOT5. After the call, it contains the root. The two parameters, y and x, are both called by name.

real del0x. First increment in x. The two first function evaluations are made for x=x and x=x+del0x. Must be specified by the user.

real eps. The required accuracy in y. The root search is stopped, when abs(y)≤eps.

The declaration of ROOT5 is:

```
procedure ROOT5(y, x, del0x, eps);
value del0x, eps;
real y, x, del0x, eps;
begin
   real yold, ynew, delx;
   ynew := y;
   yold := 2×ynew;
   delx := del0x;
L: if eps - abs(ynew) ≥ 0 then go to EX;
   delx := delx×ynew/(yold - ynew);
   x := x + delx;
   yold := ynew;
   ynew := y;
   go to L;
EX:
end ROOT5;
```

The procedure ROOT5 works as follows. The function, y, is evaluated and assigned to the local variable, ynew. Another local variable, yold, is put equal to twice this value. The trick behind this is explained below. The local variable, delx, is then put equal to the first increment, del0x.

At the label, L, we then enter the iteration cycle. If abs(ynew) is less than or equal to eps, the iteration is stopped, and the calculation is finished. Otherwise, a new increment is calculated from equation (4.47) written as:

(4.48) $\quad delx = delx \times ynew/(yold-ynew)$

As yold was first put equal to 2×ynew, and delx was set to del0x, the first value of delx becomes del0x after this formula. We then add delx to x, store ynew as yold, and calculate ynew again from the function, y. A jump is then made to label L for exit or a new iteration cycle.

Note, that we stay in the procedure until the root is found (except for the indirect function evaluations).

A test of ROOT5 (and some of the later procedures) was made by means of a small procedure, YPOL, which has the declaration:

```
real procedure YPOL;
begin
   real Z;
   Z:= (((-2.5486772₁₀-10×X - 7.1126416₁₀-7)×X + 8.4959053₁₀-3×X +
   3.359595)×X + 665.17485 - Y0;
   writecr;
   write(‹-ddddd.dddd00›, X, Z);
   YPOL := Z;
end YPOL;
```

The procedure calculates the enthalpy of formation, H, of methane minus a target value, Y0, for a given temperature, X. Equation (4.34) on page 52 is used, but with full accuracy in the coefficients. X and Y0 are global variables, and YPOL assumes the value: H - Y0. The procedure prints a new line with the value of X and YPOL. The program d-391 tests ROOT5 with YPOL:

```
Program d-391. Test of ROOT5.
begin
   real X, Y0;
   copy ROOT5<
   copy YPOL<
   select(8);
   writetext(‹<
Output d-391

     X              Y›);
   X := 1000;
   Y0 := 16580;
   ROOT5(YPOL, X, 10, 0.01);
   writecr;
end;
```

The program gave the output:

Output d-391

X	Y
1000.00000	-5025.45685
1010.00000	-4852.99390
1291.39341	351.671265
1272.38008	-18.845520
1273.34715	-0.057007
1273.35009	0.000000

The convergence of the root determination is quite fast, but this is not surprising because the YPOL-function is nearly linear in the narrow range considered.

4.3.3. The Procedure ROOT8. The procedure ROOT5 is very short and efficient. In a few cases, however, it will not work satisfactorily. If, for some reason, yold and ynew become equal, we will get division by zero, and the calculation is interrupted. If yold and ynew become very nearly equal, the difference quotient will not be very accurate, which may cause trouble. A check for these possibilities will make the procedure larger and slower, but there will be cases in practice where one is willing to pay this price in order to get an extremely reliable method.

When we want to construct a very reliable root finding procedure, there are a number of points concerning strategy, safety measures, etc. that must be settled. These points are discussed in the following, and we finish with the declaration of a procedure, ROOT8, made according to these principles.

1. Iteration principle. The regula falsi method is preferred instead of the Newton-Raphson method, because the latter requires explicite calculation of the derivative, and this can only be done in rare cases. There exists a third method, the bisection method, which is still safer (and slower) than the regula falsi method. It works by narrowing down the interval in which the root is located through bisection of the interval in such a way that the function values at the two end points of the interval always have opposite sign. The procedure ROOT7 (see page 69) is made according to this principle and with additional safeguards. ROOT8 uses regula falsi.

In order to avoid division by zero and poor accuracy we take care that we only use the difference quotient:

(4.49) alfa = (YNEW-YOLD)/DELX

if it is numerically reliable. The value of YNEW-YOLD must not be very small compared with YOLD. In the procedure ROOT8 a new value of alfa is only calculated if the absolute value of YNEW-YOLD is greater than 0.0001 times the absolute value of YOLD.

2. Range of definition. In some cases it is very important that the independent variable, X, stays within a given range (from XMIN to XMAX), but in other cases this is of no importance. In order not to use too many parameters for this purpose, we use only a single parameter:

```
boolean outside
```

If this control is not required, we write the actual parameter as false. If we must check, that XMIN $\leq$ X $\leq$ XMAX, we may write an expression or declare a special procedure:

```
boolean procedure outside;
outside := X < XMIN ∨ X > XMAX;
```

When the iteration procedure has found a new X-value from:

```
X := X + DELX;
```

it makes a call of outside (which is called by name) for control. This method has also been found useful in other iterative procedures, e.g. for optimization. In this way we can easily introduce other conditions, such as that another function of X must lie in a given interval.

3. Increment control. It may be useful to put an upper limit to the increment, DELX, especially when dealing with difficult functions. For this we introduce a real type parameter, maxf, so that the maximum increment is maxf×del0x, where del0x is the start increment.

If we come outside the range of definition or maxf is surpassed, the value of the actual increment, DELX, is halved until accordance.

4. Criterion of convergence. In ROOT5 the criterion that the root was found was that the Y-value became less than a specified tolerance, eps. We could also have used the criterion that DELX must become less than a given tolerance. As a tolerance on X may always be converted into a tolerance on Y by multiplication by alfa, it is not important which of the two methods we select. The experience in use of procedures for root determination and optimization has shown that it is completely satisfactory to use the tolerance on X (which is also more natural from a mathematical point of view), and to assume that the root is found when the absolute value of DELX becomes less than the tolerance. The latter is not quite satisfactory from a mathematical point of view, because we may risk to converge towards an X-value which is not a root, but a local minimum. A rational solution of this problem also depends upon the treatment of the next point:

5. Criterion of error. Some functions can be so difficult to handle that it will be fair to permit that the root procedure gives up and goes to an error label. We must then have a clear definition of the set of functions the procedure must be able to handle, and for which functions it may fail.

Naur (1964) has given an example of an automatic grading of a series of root determination procedures prepared by a class of students. The problem was defined so that F(X) was given in the closed interval $a \leq X \leq b$. It was assumed that F(a) and F(b) have opposite signs, and if this was not the case, the procedure must conclude that there was no root.

In this situation we have a continuous function defined in a closed interval and with opposite signs at the two end points. There will always be at least one root. The question is now, whether this is a realistic way of stating the problem. The experience shows that in practical problems we can very often give a rough estimate of the position of the root, but it may be quite difficult to specify a closed interval inside which the root can be found with certainty, and where there are opposite signs at the two end points. Instead of starting the investigation at the two end points with a calculation of F(a) and F(b) and then narrow down the interval inside which the root must be located, it is more realistic to start the search at a single point:

```
X = XSTART
```

and then try a neighbor point:

X = XSTART + DELOX

and use the regula falsi method from these points. If F(a) and F(b) are to be included in the investigation, we run the risk that a and b are so far from the root, that the function cannot be calculated, because it assumes extreme and unrealistic values.

If we select the strategy to start with XSTART and XSTART+DELOX, it is reasonable to assume that the function is monotonously increasing or decreasing around these points. With this assumption we will always find the root with the regula falsi method. If the assumption is not correct, i.e. we have local minima or maxima, we may risk to converge towards one of these extremal point. In the procedure ROOT8 we store the sign of the first calculated difference quotient and perform the error exit if we at a later time get a difference quotient with opposite sign. The error exit is also used, if the absolute value of the best (smallest) Y-value is many times (100) larger than the absolute value of alfa×DELX, because we must then have converged to an extremal point, not to a root.

6. Other safeguards. In ROOT8 we have the additional security, that the procedure always stores the Y-value having the smallest absolute value, and the corresponding X-value. The latter is always delivered as the requested root. If we are afraid that the procedure does not converge within a reasonable time, we may build in a counting in the Y-procedure and go to the error label, if the number of Y-calls becomes too high.

The declaration of ROOT8 is:

```
procedure ROOT8(y, x, del0x, eps, outsi, maxf, ERROR);
value del0x, eps, maxf;
boolean outsi;
real y, x, del0x, eps, maxf;
label ERROR;
begin
   boolean first, up, error;
   real xold, xbest, ybest, yold, ynew, delx, A, diff;
   xbest := xold := x;
   ybest := ynew := y;
   yold := 2×ynew;
   delx := -del0x×sign(ynew);
```

```
    first := true;
    error := false;
    for diff := ynew - yold while
    abs(delx) > eps ∧ yold ≠ 0 ∧ -, error do
    begin
       if first ∨ abs(diff) > 1₁₀-4×abs(yold) then A := diff/delx;
       if -, first ∧ (A < 0 = up) then error := true;
       delx := -ynew/A;
       for x := xold + delx while outsi ∨ abs(delx) > abs(maxf×del0x) do
       delx := 0.5×delx;
       xold := x;
       yold := ynew;
       ynew := y;
       if first then
       begin
          up := (ynew-yold)/delx > 0;
          first := false
       end if first;
       if abs(ynew) < abs(ybest) then
       begin
          ybest := ynew;
          xbest := xold;
       end if better
    end for diff;
    x := xbest;
    if error ∨ abs(ybest) > 100×abs(A×delx) then go to ERROR
end ROOT8;
```

The following parameters are used in ROOT8:

real y, x, del0x. These are the same as in ROOT5 (see page 57).

real eps. The permissible error in x. The iteration is continued as long as the increment is larger than eps.

boolean outsi. This parameter is called by name and is normally a boolean procedure. Whenever ROOT8 has assigned a new value to x, it will call outsi, which must then yield the value true, if x is outside a permissible range, otherwise false.

real maxf. If the increment becomes larger than maxf×del0x, the increment is halved until the condition is satisfied.

label ERROR. An exit to this label is made, if the sign of the

difference quotient changes during the calculation, or if the y-value corresponding to the calculated root is more than 100 times larger than the product of the difference quotient and the last increment in x (all taken as absolute values).

The initial operations in ROOT8 are the same as in ROOT5. The sign of the first increment is adjusted to correspond to an increasing function. If the function is known to be decreasing, del0x may be inserted with a negative value.

The main part of the procedure is a for-statement:

```
for diff := ynew - yold while .....
```

The for-statement continues as long as the increment is larger than eps, and yold is not zero, and no error has been detected. A new value of alfa (A) is calculated, except if abs(ynew-yold) > 0.0001×abs(yold), as explained above. A local boolean, error, is set to true, if alfa has changed sign.

The increment, delx, is then calculated as -ynew/alfa. The double check for x being outside the permitted range or the increment larger than maxf×del0x is then made. A new y-value is then calculated, and in the first cycle the sign of the difference quotient is stored as the local boolean, up. The best set of x and y is checked in each iteration and stored as xbest and ybest.

When the main for-statement is finished, x is assigned from xbest, and the ERROR-exit conditions are checked, as explained above.

The procedure was tested on the program d-392. It finds the root in 11 different functions of which the 9 first are taken from Naur(1964) in the students grading program mentioned above. The two last functions are y = X↑2 - 0.5 and the YPOL function. The program is:

Program d-392. Test of ROOT8.

```
begin
   integer i, j;
   real x;
   copy ROOT8<
   procedure P(xmin, xmax, xstart, eps, root, function);
   value xmin, xmax, xstart, eps, root;
   real xmin, xmax, xstart, eps, root, function;
```

```
   begin
      boolean procedure outside;
      outside := x < xmin ∨ x > xmax;
      real procedure Y;
      begin
         Y := function;
         j := j + 1
      end Y;
      i := i + 1;
      j := 0;
      writecr;
      write(‹-dd›, i);
      x := xstart;
      ROOT8(Y, x, 0.01×(xmax-xmin), eps, outside, 10, ERROR);
      writetext(‹<   ›);
      go to F1;
ERROR:writetext(‹< E ›);
F1:   write(‹-ddddd›, j);
      write(‹    -d.ddddddd10-dd›, root, x, Y);
   end P;
   i := 0;
   select(8);
   writetext(‹<
Output d-392

No   Iteration  Root, true        Root, calc.       y
›);
   P(0, 2, 0.05, 1₁₀-6, 0.1, if x≤0.1 then 10×x - 1 else 1₁₀-20);
   P(-2, 0, -1, 1₁₀-6, 0, -1+x);
   P(5, 6, 5.5, 1₁₀-6, 5, 5-x);
   P(-17, -13, -15, 1₁₀-6, -13, x+13);
   P(0, 20, 10, 1₁₀-4, 0.95, (x+0.05)↑0.1-1);
   P(0.001, 99.9, 0.5, 1₁₀-5, 0.01, x+1/x-100.01);
   P(2, 12, 7, 1₁₀-6, 10, x/10-x↑6/1₁₀8-0.99);
   P(-5, 10, 3, 1₁₀-6, 1.324718, x↑3-1-x);
   P(-3.2, 20, -1, 1₁₀-5, -0.4, sin(x)+x/4+0.489418);
```

```
    P(0, 1, 0.5, 1₁₀-5, sqrt(0.5), x↑2-0.5);
    P(0, 2000, 1000, 1₁₀-2, 1273.35, 665.17485+x×(3.359595+x×(8.4959053₁₀-3
    +x×(-7.1126416₁₀-7-x×2.5486772₁₀-10)))-16580);
    writecr;
end program;
```

The program gave the following output:

Output d-392

No	Iteration	Root, true	Root, calc.	y
1	4	1.0000000 $_{10}$-1	1.0000000 $_{10}$-1	-3.7252903 $_{10}$-9
2 E	16	0.0000000	-6.3137652 $_{10}$-7	-1.0000006
3	10	5.0000000	5.0000000	0.0000000
4	10	-1.3000000 $_{10}$ 1	-1.3000000 $_{10}$ 1	0.0000000
5	12	9.5000000 $_{10}$-1	9.4999994 $_{10}$-1	-7.4505806 $_{10}$-9
6	13	1.0000000 $_{10}$-2	9.9999986 $_{10}$-3	1.3828278 $_{10}$-5
7	9	1.0000000 $_{10}$ 1	9.9999999	-3.7252903 $_{10}$-9
8	11	1.3247180	1.3247180	-3.7252903 $_{10}$-9
9	7	-4.0000000 $_{10}$-1	-3.9999971 $_{10}$-1	0.0000000
10	8	7.0710678 $_{10}$-1	7.0710678 $_{10}$-1	0.0000000
11	7	1.2733500 $_{10}$ 3	1.2733501 $_{10}$ 3	0.0000000

The program writes the function number, the number of iterations, the known root (which is not used), the calculated root, and the corresponding y-value. Function no. 2 has given the error exit, because x-1 has no root in the range from -2 to 0. The procedure yields x=0 as the value giving the lowest absolute value of y. The average number of iterations in the other functions is 9.

4.3.4. Bisection Method for Difficult Functions. The procedure ROOT7 explained below has been designed to take care of a special kind of difficult root determinations occurring in connection with the solution of two-point boundary conditions for diffusion in catalyst particles. The situation is illustrated on figure 3.

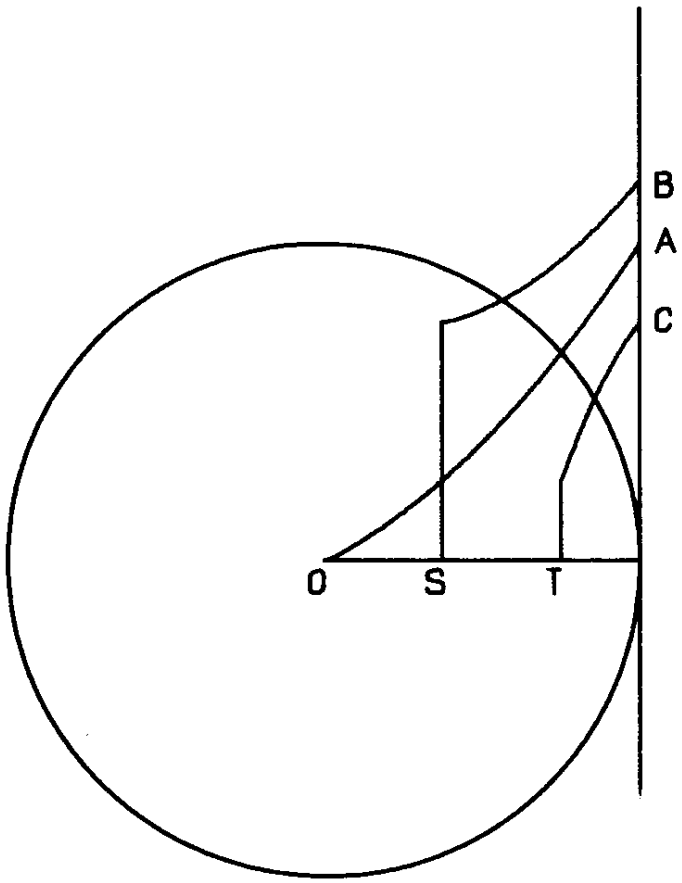


Figure 3

Integration and Root Determination

in a Catalyst Pellet

The problem is to integrate a function starting with the value A at the pellet surface and ending with the value zero at the pellet center. If the start-A is not selected properly, the center value will not be zero as it should. Thus, we have a root determination problem: The independent X-value is the start-A, and the Y-function is the function value at the center that must be reduced to zero.

If A is selected too far from the correct value (point B or C), it may happen that the integration cannot be carried through to the particle center, but must be stopped before (point S or T). The stop in the integration is normally caused by some of the mole fractions becoming negative, so that the reaction rates, etc. cannot be calculated.

When the integration is stopped before the end point is reached, we must use the function value at the pseudo-end-point as an approximation to the true end-point function value. We must also collect information on how far the integration could be carried out, i.e. the abscissa of the points S and T. When two function evaluations are compared, the one having the lower penetration is simply disregarded.

The procedure ROOT7 described below uses bisection, not the regula falsi method. The reason for this is that the approximation of the difference quotient calculated via regula falsi is not likely to be very reliable because of the varying penetration. In the bisection method, the interval inside which X must be located is gradually narrowed down by halving. The interval halving is not started until the procedure has located two X-values giving function values of opposite sign at the end point of the integration. This is further explained in the next section.

4.3.5. The Procedure ROOT7. The declaration is:

```
procedure ROOT7(y, x, del0x, xmin, xmax, delmax, eps, cfirst, clast,
cact, trouble);
value del0x, xmin, xmax, delmax, eps, cfirst, clast;
boolean trouble;
integer cfirst, clast, cact;
real y, x, del0x, xmin, xmax, delmax, eps;
begin
   integer type, clow, chigh, cnew;
   real delx, ynew, yoldl, yoldh;
```

```
      yoldl := -1₁₀100;
      yoldh := -yoldl;
      type := 0;
      trouble := false;
      clow := chigh := cfirst;
LL:   ynew := y;
      cnew := cact;
      if ynew < 0 then
      begin
         if abs(cnew-clast) < abs(clow - clast)
         ∨ cnew = clow ∧ ynew > yoldl then
         begin
            clow := cnew;
            yoldl := ynew;
            xmin := x;
         end if improvement;
         type := if type ≤ 1 then 1 else 3;
      end if negative
      else
      begin
         if abs(cnew-clast) < abs(chigh-clast)
         ∨ cnew = chigh ∧ ynew < yoldh then
         begin
            chigh := cnew;
            yoldh := ynew;
            xmax := x;
         end if improvement;
         type := if type = 0 ∨ type = 2 then 2 else 3;
      end if positive;
      delx := if type = 3 then 0.5×(xmax+xmin) - x else
      if type = 1 then del0x else - del0x;
      if type = 3 ∧ xmax - xmin < eps then go to EX;
      if abs(delx) > delmax then delx := sign(delx)×delmax;
      if x + delx ≥ xmax ∨ x + delx ≤ xmin then
      begin
         if type < 3 then trouble := true;
         delx := 0.5×(xmax + xmin) - x;
      end if out of range;
```

```
    x := x + delx;
    go to LL;
    EX:
end ROOT7;
```

The formal parameters in ROOT7 are:

real y, x, del0x. The same as in ROOT8 and ROOT5 (see page 57).

real xmin, xmax. Initial lower and upper bounds on x. Are used internally to narrow down the range, but as they are called by value, the new limits are not available outside the procedure.

real delmax. The maximum value of the change in x.

real eps. The permissible error in x (as in ROOT8).

integer cfirst, clast, cact. These numbers define the degree of penetration of the integration. Example:

cfirst = 0 (no penetration)
clast = 10 (full penetration)

Whenever the procedure y is called for an actual value of x, the procedure must also deliver a value of cact, defining the actual degree of penetration: cfirst $<$ cact $\leq$ clast.

boolean trouble. The procedure sets this to false at the start and changes it to true, if y has the same sign for all trials in the initial search.

The procedure operates as follows:

The best negative y-value is stored as the local variable, yoldl, and the best positive y-value as yoldh. These are first set to very large negative and positive numbers ($\pm 1_{10}100$).

The best values of cact are stored in clow (negative y) and chigh (positive y) and these are both set to cfirst in the start.

A local counter, type, is used to indicate the progress of the calculation:

type = 0 Start phase, no y-values found yet
type = 1 Only negative y-values met so far
type = 2 Only positive y-values met so far
type = 3 Both negative and positive y-values met.

The iteration cycle starts at the label: LL. The procedure, y, is called, and its value is assigned to ynew. The corresponding value of cact is assigned to cnew.

If ynew is negative, we have improved the solution, if either of the two possibilities is fulfilled:

1. cnew is better than the previously best value (clow), i.e. cnew is closer to clast.

2. cnew is equal to clow, but the y-value is improved (ynew>yoldl).

We then assign cnew to clow and ynew to yoldl. The lower limit of x, xmin, is put equal to x.

If ynew is positive, a similar updating is made for chigh, yoldh, and xmax.

The value of type is updated, if possible.

As long as the procedure has seen y-values of only one sign (type = 1 or 2), x is changed in fixed steps, del0x, for y negative and -del0x for y positive. We assume that y is an increasing function (use negative del0x for a decreasing function). As soon as it has found an interval with opposite sign of the y-values at the two end points, the next value of x is found by bisection of this interval, and the iteration is continued, until xmax-xmin < eps.

The program d-393 shown below makes a simple test of ROOT7 on the function:

(4.50) $\quad Y = X\uparrow 2 - 0.5$

The root is X = 0.707107, and we simulate the integration penetration troubles by assigning a value of cact which is 10 close to the root and decreases quickly when we move away from the root:

(4.51) $\quad cact = 10 - 20\times abs(X - 0.7)$

The declaration is:

```
Program d-393. Test of ROOT7.
begin
   boolean trouble;
   integer cact, i;
```

```
   real X, Z;
   copy ROOT7 <
   real procedure Y;
   begin
      Z := X↑2 - 0.5;
      cact := 10 - 20×abs(X - 0.7);
      writecr;
      write(‹-dddd›, i, cact);
      write(‹-ddd.ddddddd›, X, Z);
      Y := Z;
      i := i+1;
   end Y;
   select(8);
   writetext(‹<
Output d-393:

    i  cact      X           Y
›);
   i := 1;
   X := 0.2;
   ROOT7(Y, X, 0.1, 0, 1, 0.2, 1₁₀-5,  -10, 10, cact, trouble);
   if trouble then writetext(‹<Error›);
   writecr;
end;
```

The program gives the output:

```
Output d-393:

    i  cact      X           Y

    1    0   0.2000000  -0.4600000
    2    2   0.3000000  -0.4100000
    3    4   0.4000000  -0.3400000
    4    6   0.5000000  -0.2500000
    5    8   0.6000000  -0.1400000
    6   10   0.7000000  -0.0100000
    7    8   0.8000000   0.1400000
```

8	9	0.7500000	0.0625000
9	10	0.7250000	0.0256250
10	10	0.7125000	0.0076562
11	10	0.7062500	-0.0012109
12	10	0.7093750	0.0032129
13	10	0.7078125	0.0009985
14	10	0.7070312	-0.0001068
15	10	0.7074219	0.0004457
16	10	0.7072266	0.0001694
17	10	0.7071289	0.0000313
18	10	0.7070801	-0.0000378
19	10	0.7071045	-0.0000032
20	10	0.7071167	0.0000140
21	10	0.7071106	0.0000054

The rather slow operation of ROOT7 is due to the bisection method, which actually finds only a single bit of X in each iteration.

An interesting root determination procedure has been published by Schreiner Andersen (1970). It uses a weighted combination of regula falsi and bisection with adaptive improvement of the weight factor.

For the sake of good order it should be noted here, that the numerical difficulties in the problem illustrated in figure 3 (page 68) are connected with the two-point boundary condition inhererent in the catalyst effectiveness calculation. When the differential equations are solved by means of the Runge-Kutta method, the correct solution of the boundary conditions may become quite difficult. If, however, global solution methods are used instead of the Runge-Kutta method, the solution may become much easier. Villadsen (1970) treats this problem in considerable detail by means of the orthogonal collocation method. The adaption of this method or similar methods may remove the necessity of having a procedure like ROOT7, but the general approach in ROOT7 could be useful in other connections.

4.4. Several Non-Linear Equations.

4.4.1. The Procedure NOLEQ8. This standard procedure performs the solution of several non-linear equations after the Newton-Raphson method or its approximation by difference quotients (regula falsi in the one-dimensional case).

The calculation of the difference quotients is equivalent to the generation of a linear model of the functions (see Chapter 3), and the model generation may, therefore, be carried out by GENMOD1 or a similar procedure, after which the non-linear equations are solved as explained on page 41. In other cases it is convenient to have the generation and the solution combined into a single procedure, and this is done in NOLEQ8. The declaration is:

```
integer procedure NOLEQ8 (var, count, cyc, cmax, cymax, outsi,
first, epsy, new, d, xstart, del0x, delx, xact, epsx, yact, y0, yold,
eps, maxf);
value var, cmax, cymax, epsy, new, eps, maxf;
integer var, count, cyc, cmax, cymax;
boolean outsi, first, epsy, new, d;
real eps, maxf;
array xstart, del0x, delx, xact, epsx, yact, y0, yold;
begin
   boolean found;
   integer c, i, j, error;
   real R, S;
   NOLEQ8 := 0;
   c := count - 1;
   found := false;
   for i := 1 step 1 until var do xact[i] := xstart[i];
   if count = 0 then
   begin
      count := count + 1;
      if new then d := false;
      go to L3
   end if count = 0;
```

```
if count = 1 ∨ -, first then
begin
   found := epsy;
   for i := 1 step 1 until var do
   begin
      R := -yact[i];
      if epsy then
      begin
         if abs(R) > epsx[i] then found := false
      end if epsy;
      y0[i] := R
   end for i;
   if found then
   begin
      NOLEQ8 := 1;
      for i := 1 step 1 until var do xstart[i] := xact[i];
      go to L3
   end if found;
end if count = 1 or not first;
if first then
begin
   if count > 1 then
   for i := 1 step 1 until var do
   begin
      yold[i,c] := if new then yact[i]
      else (yact[i] + y0[i])/delx[c];
   end for i and if count > 1;
   if count ≤ var then
   begin
      if new then
      begin
         d := true;
         go to L4
      end if new;
      j := 1;
L1:   delx[count] := del0x[count]/j;
      xact[count] := xstart[count] + delx[count];
```

```
            if outsi then
            begin
               j := 2×j;
               go to L1
            end if outside;
L4:      end if count ≤ var
         else
         begin
            for i := 1 step 1 until var do
            for j := 1 step 1 until var do
            yold[i, var+j] := if i = j then 1 else 0;
            error := NOLEQ8 := -LEQ1(var, var, yold, eps);
            if error < 0 then go to L3;
            for i := 1 step 1 until var do
            for j := 1 step 1 until var do
            yold[i, j] := yold[i, var+j];
            first := false;
            count := 0;
            cyc := cyc + 1;
            if new then d := false
         end inversion
      end if first;
      if -, first then
      begin
         found := -, epsy;
         S := 0;
         for i := 1 step 1 until var do
         begin
            R := 0;
            for j := 1 step 1 until var do
            R := R + yold[i, j]×y0[j];
            delx[i] := R;
            R := abs(R/del0x[i]);
            if R > S then S := R;
            if -, epsy then
            begin
               if abs(delx[i]) > epsx[i] then found := false
            end if not epsy
         end for i;
```

```
      S := if S > maxf then maxf/S else 1;
L2:   for i := 1 step 1 until var do
      xact[i] := xstart[i] + delx[i]×S;
      if outsi then
      begin
         S := 0.5×S;
         go to L2
      end if outsi;
      for i := 1 step 1 until var do xstart[i] := xact[i]
   end if not first;
   if found then NOLEQ8 := count := 1
   else
   if cyc > cymax then NOLEQ8 := -1
   else
   begin
      count := count + 1;
      if count ≥ cmax ∧ -, first then
      begin
         count := 1;
         first := true
      end
   end normal exit;
   L3:
end NOLEQ8;
```

4.4.1.1. Parameters. The procedure has 20 formal parameters which are explained below.

integer var. The number of non-linear equations (= the number of unknowns).

integer count. An iteration counter which must be set to zero before the first call. The procedure increases count by one at each exit during build-up of the linear model and resets it to zero when the model is finished.

integer cyc. The number of cycles carried out. Must be set to zero before the first call. Is increased by the procedure.

integer cmax. The maximum number of iterations before a new linear model is generated.

integer cymax. The maximum number of cycles.

boolean outsi. This will normally be a boolean procedure. It is called (by name) by NOLEQ8 every time a new set of actual x-values, xact[1:var], is calculated. It must yield the value true, if the actual set of x is outside the permitted range, otherwise false.

boolean first. Must be put to false, if the old linear model should be used, otherwise true.

boolean epsy. True: epsx refers to y, false: epsx refers to x.

boolean new. True gives Newton-Raphson calculation method using the derivatives. False gives the approximate calculation method using difference quotients only.

boolean d. When new is true, the procedure will vary the value of d. For d true the main program must deliver the derivatives in yact and for d false the main program must deliver the function values in yact. For new false, d is not used.

array xstart[1:var]. Start values of x.

array del0x[1:var]. Start increments in x.

array delx[1:var]. The actual increments in x.

array xact[1:var]. Actual values of x.

array epsx[1:var]. Permissible errors in x (or in y, if epsy is true).

array yact[1:var]. Actual y-values (or derivative values, if new and d are true).

array y0[1:var]. Used for storage of the y-values (with opposite sign) corresponding to xstart.

array yold[1:var,1:2×var]. Used by the procedure for storage of derivatives or difference quotients during generation of the linear model, and for the complete linear model.

real eps: Minimum pivot accepted by LEQ1. If any of the pivots becomes less than eps, the calculation is stopped.

real maxf: The procedure checks that the change in variable no. i does not exceed maxf×del0x[i] for all values of i. If the change is too large, all increments will be reduced proportionally.

The procedure is of the open type: We leave the procedure when a new set of x-values has been assigned and the main program must then find corresponding y-values, and NOLEQ8 is called again. The control of these calls is made by the counter, count, and the overall state of the calculation is stored in NOLEQ8 itself, which is of type integer. The following values are possible:

NOLEQ8 = 0: Normal exit. The main program must calculate new values of yact.

NOLEQ8 = 1: The solution is found and contained in xact.

NOLEQ8 = -1: Calculation impossible. Either inversion trouble or cyc > cymax.

The non-local procedure LEQ1 must be available to the procedure.

4.4.1.2. Model Generation. When NOLEQ8 is used with difference quotients, not derivatives, the first part of the calculation is taken up by the generation of the linear models of the functions that are to be reduced to zero. A detailed description is given for three equations in three unknowns. We can write the equations as:

$$\begin{aligned} &F1(x1, x2, x3) = 0 \\ &F2(x1, x2, x3) = 0 \\ &F3(x1, x2, x3) = 0 \end{aligned} \tag{4.52}$$

and we must find the set of numbers, x1, x2, and x3, so that the three expressions: F1, F2, and F3 become zero within a given tolerance.

The calculation is started at a specified point:

```
xstart[1], xstart[2], xstart[3]
```

and the three functions are calculated in this point and in certain neighbor points. We then express the three functions as linear functions of three variables, we put these equal to zero, and solve the set of linear equations obtained in this way. The solution found will be correct, if the original functions were linear. Otherwise, the calculation is repeated around the new point.

At the first call (count = 0) of NOLEQ8, it simply inserts the start values of xact:

```
for i := 1 step 1 until var do xact[i] := xstart[i];
```

The main program must then calculate the corresponding y-values.

In the next call (count = 1), the yact-values are stored in y0 (with opposite sign):

```
for i := 1 step 1 until var do y0[i] := -yact[i];
```

The procedure then inserts the new values of xact:

```
xact[1] := xstart[1] + del0x[1];
xact[2] := xstart[2];
xact[3] := xstart[3];
```

The main program must then calculate the corresponding yact-values.

In the next call (count = 2), the procedure stores the yact-values in the first column of the yold-matrix:

```
for i := 1 step 1 until var do
yold[i,1] := (yact[i] + y0[i])/delx[1];
```

They are the difference quotients of the three functions with respect to the first variable.

Now the new x-values are set:

```
xact[1] := xstart[1];
xact[2] := xstart[2] + del0x[2];
xact[3] := xstart[3];
```

and after exit and a new call (count = 3), the new difference quotients are stored as the second column of yold.

Finally, for count = 4 we store the difference quotients for the x-values:

```
xact[1] := xstart[1];
xact[2] := xstart[2];
xact[3] := xstart[3] + del0x[3];
```

as the third column of yold. The element yold[i,j] is then the difference quotient of function no. i with respect to the variable no. j.

4.4.1.3. Solution. We have now collected the necessary material to find the solution. To save space we write the difference quotients as:

```
a[i,j] := yold[i,j];
```

The required solution, x[1:3], is expressed by means of the start value, xstart[1:3], and a correction, delx[1:3]:

```
x[1] := xstart[1] + delx[1];
x[2] := xstart[2] + delx[2];
x[3] := xstart[3] + delx[3];
```

In order to find the values of delx, we express the three functions as linear approximations:

```
F[1] := a[1,1]×x[1]+a[1,2]×x[2]+a[1,3]×x[3]+k[1];
F[2] := a[2,1]×x[1]+a[2,2]×x[2]+a[2,3]×x[3]+k[2];
F[3] := a[3,1]×x[1]+a[3,2]×x[2]+a[3,3]×x[3]+k[3];
```

In matrix form this may be written:

(4.53) $F = ax + k$

We put the vector x equal to xstart + delx and get:

(4.54) $F = a(xstart + delx) + k$

For xstart we have F = -y0:

(4.55) $-y0 = xstart + k$

and for x = xstart + delx we have F = 0:

(4.56) $0 = a(xstart + delx) + k$

We can now eliminate xstart and k and find:

(4.57) $a\ delx = y0$

or, written out in detail:

```
a[1,1]×delx[1]+a[1,2]×delx[2]+a[1,3]×delx[3] = y0[1]
a[2,1]×delx[1]+a[2,2]×delx[2]+a[2,3]×delx[3] = y0[2]
a[3,1]×delx[1]+a[3,2]×delx[2]+a[3,3]×delx[3] = y0[3]
```

(4.58)

This is a simple system of linear equations. We could solve it directly on LEQ1, but as we sometimes wish to use the linear model several times without recalculation, it is more convenient to perform a matrix inversion. This is also done be means of LEQ1, but before this we must insert the unit matrix in the second half of the yold-matrix:

yold[i,j]:

	j = 4	j = 5	j = 6
i = 1	1	0	0
i = 2	0	1	0
i = 3	0	0	1

When LEQ1 is called with var unknowns and var right-hand sides, this unit matrix is replaced by the inverse of the a-matrix. It is then moved back to the first half of yold and multiplied by y0. This gives the required delx-vector.

Before the delx-vector is added to the xstart-vector, various small tests are made. If epsy is false, the procedure tests if all delx-elements are less than or equal to the corresponding epsx-element. If this in the case, the solution has been found, and the procedure is finished. If not, we find the maximum value of:

abs(delx[i]/del0x[i])

for all variables. If the maximum value, S, is greater than maxf, all increments will be multiplied by maxf/S before the addition.

The procedure then calls the parameter (procedure) outsi, and if this is true, new x-values are calculated with half of the previous increments, outsi is called again, etc. The halving is repeated as long as outsi is true. When outsi becomes false, xstart is also put equal to xact.

We then leave the procedure, and the main program must calculate the new yact-values, which are then stored as y0 (with opposite sign), and we start again with a new cycle.

4.4.1.4. Examples. Two calculation examples are given to illustrate the use of NOLEQ8. The first example uses an ammonia converter test polynomial, POL, which calculates two functions of two variables. It is a model of the performance of an ammonia converter of the quench type. It is only used as an illustration, not for design purpose.

The two independent variables in the POL-procedure are:

tinlet: Inlet temperature to the first catalyst bed (deg. C).

ginlet: The relative gas flow (in per cent of the total flow) entering the first catalyst bed.

The two functions calculated by the POL-procedure are:

PROD: The ammonia production (metr. t/24h).

LNEC: The necessary height of the lower exchanger (meter).

We wish to calculate a set of tinlet and ginlet for which:

PROD = 71.5
LNEC = 2

This is done by calculating the yact-values as:

```
yact[1] := POL(1, xact[1], xact[2]) -71.5;
yact[2] := POL(2, xact[1], xact[2]) - 2;
```

xact[1] is tinlet and xact[2] is ginlet. The program is:

```
Program d-327. Test of NOLEQ8
begin
   boolean fin, first, deriv;
   integer i, cyc, count, j;
   real PROD, LNEC;
   array xstart, del0x, xact, delx, epsx, yact, y0[1:2], yold[1:2,1:4];
   copy LEQ1 <
   copy NOLEQ8 <
   copy POL <
```

```
    for j := 1, 2, 3 do
    begin
       xstart[1] := 440;
       xstart[2] := 68;
       del0x[1] := 5;
       del0x[2] := 2;
       epsx[1] := 1;
       epsx[2] := 0.2;
       count := cyc := 0;
       select(8);
       writetext(‹<
  t-inlet   g-inlet      PROD       LNEC
›);
       fin := false;
       first := true;
H1:    i := NOLEQ8(2, count, cyc, j, 50, false, first, false, false,
       deriv, xstart, del0x, delx, xact, epsx, yact, y0, yold, 1₁₀-8, 4);
       fin := i ≠ 0;
       if fin then
       begin
          writecr;
          writetext(if i > 0 then ‹<FOUND› else ‹<ERROR›);
       end if fin;
       writecr;
       if (count - 1) mod 3 = 0 then writecr;
       write(‹-dddd.dddd›, xact[1], xact[2]);
       PROD := POL(1, xact[1], xact[2]);
       yact[1] := PROD - 71.5;
       LNEC := POL(2, xact[1], xact[2]);
       yact[2] := LNEC - 2;
       write(‹-dddd.ddddOO›, PROD, LNEC);
       if -, fin then go to H1;
       writecr;
    end for j;
end;
```

The program gave the following output:

t-inlet	g-inlet	PROD	LNEC
440.0000	68.0000	72.538228	2.599180
445.0000	68.0000	72.669139	2.669936
440.0000	70.0000	72.853762	2.411905
420.0000	68.1038	71.298674	2.348836
425.0000	68.1038	71.759268	2.428249
420.0000	70.1038	72.145485	2.178332
411.0457	70.5275	71.587530	2.092726
416.0457	70.5275	72.014447	2.123248
411.0457	72.5275	72.266506	1.958033
406.2680	71.4713	71.500424	1.998335
411.2680	71.4713	71.969902	2.030972
406.2680	73.4713	72.175319	1.877078

FOUND

406.3359	71.4511	71.499290	2.000093

t-inlet	g-inlet	PROD	LNEC
440.0000	68.0000	72.538228	2.599180
445.0000	68.0000	72.669139	2.669936
440.0000	70.0000	72.853762	2.411905
420.0000	68.1038	71.298674	2.348836
412.2756	70.6618	71.757206	2.092342
417.2756	70.6618	72.149467	2.120371
412.2756	72.6618	72.378906	1.957585
405.7259	71.4874	71.449183	1.993049

FOUND

406.5205	71.4504	71.518405	2.001573

t-inlet	g-inlet	PROD	LNEC
440.0000	68.0000	72.538228	2.599180
445.0000	68.0000	72.669139	2.669936
440.0000	70.0000	72.853762	2.411905
420.0000	68.1038	71.298674	2.348836
412.2756	70.6618	71.757206	2.092342
404.0239	70.4010	70.722371	2.021311
409.0239	70.4010	71.324718	2.083419
404.0239	72.4010	71.636650	1.928161
406.3692	71.4840	71.516311	1.998341

FOUND

406.3690	71.4484	71.501646	2.000520

The solution is found to tinlet = 406 and ginlet = 71.5. The program illustrates the simple use of the difference quotient method. In this example the solution is found three times, for three different values of cmax. The total number of iterations are:

cmax	Iterations
1	13
2	9
3	10

This clearly shows the advantage of using the linear model a few times (but not too many), before a new model is generated.

The second example gives a comparison between the difference quotient method and the derivative (Newton-Raphson) method. We wish to solve the two non-linear equations:

$$(4.59) \quad X1^{1/2} + X2^{1/2} - 2 = 0$$

$$(4.60) \quad 1/X1^{1/2} + X2^{1/2} - 2 = 0$$

They have the solution X1 = X2 = 1 for positive values of X1 and X2. The program is:

```
Program d-402. Second test of NOLEQ8
begin
   boolean fin, first, deriv, new;
   integer i, cyc, count, cmax, type, eval;
   array xstart, del0x, xact, delx, epsx, yact, y0[1:2],
   yold[1:2,1:4];
   copy LEQ1 <
   copy NOLEQ8 <
   select(8);
   writetext(‡<<
Output d-402

  type    eval
     cmax        x1              x2              y1              y2
‡);
   for type := 1, 2 do
   for cmax := 1 step 1 until 4 do
   begin
      writecr;
      xstart[1] := 2;
      xstart[2] := 3;
      del0x[1] := del0x[2] := 0.5;
      epsx[1] := epsx[2] := 1₁₀-3;
      count := cyc := eval := 0;
      fin := false;
      first := true;
      new := type = 2;
      deriv := false;
H1:   i := NOLEQ8(2, count, cyc, cmax, 50,
      xact[1] ≤ 0 ∨ xact[2] ≤ 0, first, false, new, deriv,
      xstart, del0x, delx, xact, epsx, yact, y0, yold, 1₁₀-8, 4);
      fin := i ≠ 0;
      eval := eval + 1;
```

```
      yact[1] := if deriv then 2×xact[count-1] else
      xact[1]↑2 + xact[2]↑2 -2:
      yact[2] := if deriv then                    )
     (case count -1 of(-2/xact[1]↑3, 2×xact[2])else
      1/xact[1]↑2 + xact[2]↑2 -2;
      if -, fin then go to H1;
      write(‹dddd›, type, cmax, eval);
      write(‹  -d.dddddd10-dd›, xact[1], xact[2], yact[1], yact[2]);
   end for type and cmax;
   writecr;
end;
```

Output from the program is:

Output d-402

type	cmax	eval	x1	x2	y1	y2
1	1	22	1.000000	1.000145	2.910048_{10}-4	2.905726_{10}-4
1	2	17	1.000000	1.000126	2.514198_{10}-4	2.511814_{10}-4
1	3	16	1.000000	1.000093	1.856461_{10}-4	1.856163_{10}-4
1	4	17	1.000000	1.000062	1.245812_{10}-4	1.245812_{10}-4
2	1	16	1.000000	1.000000	0.000000	0.000000
2	2	13	1.000000	1.000003	6.273389_{10}-6	6.280839_{10}-6
2	3	15	1.000000	1.000000	0.000000	0.000000
2	4	13	1.000000	1.000031	6.211549_{10}-5	6.212294_{10}-5

The program contains the two for-statements:

```
for type := 1, 2 do
for cmax := 1 step 1 until 4 do
```

Calculations for type = 1 use the difference quotients, in which only the two functions:

```
yact[1] := xact[1]↑2 + xact[2]↑2 - 2;
yact[2] := 1/xact[1]↑2 + xact[2]↑2 - 2;
```

are evaluated. For type = 2 the Newton-Raphson method is used. The function evaluation then depends upon the value of the parameter: deriv. If deriv is false, the y-functions are calculated as above, but if it is true, we must calculate the partial derivatives:

	count = 2	count = 3
yact[1]	2×xact[1]	2×xact[2]
yact[2]	-2/xact[1]↑3	2×xact[2]

Variation of the parameter, cmax, illustrates the effect of using the linear model more than once. The number of iterations required are:

Iterations

cmax	type = 1	type = 2
1	22	16
2	17	13
3	16	15
4	17	13

This clearly shows the advantage of the Newton-Raphson method. Note, however, that this method requires programming of expressions for the derivatives.

4.4.2. Separation of Model Generation and Solution. As mentioned in sections 3.6 and 4.4.1, the action of the NOLEQ8 procedure can be separated into two parts: Model generation and solution of the linear equations. It should also be mentioned here, that the optimization method described in Chapter 5 in which side conditions are taken care of by means of penalties, can also be used on problems which involve only non-linear equations and no optimization as such.

4.5. Series of Roots.

In some cases we wish to calculate the root of a function for several different values of a parameter occurring in the function expression. As an example we take the function:

(4.61) $Y = X\uparrow 6 + P \times X\uparrow 5 - 20 \times X - 1$

For P = 20 we have a root at X = 1. We now wish to calculate this root for P = 20, 21, 22, etc. up to 30. If calculations of this type are to be made by hand, we will gradually build up a table of the roots and the differences:

P	Root	Delta1	Delta2	Delta3
20	1.00000			
		-0.01130		
21	0.98870		+0.00061	
		-0.01069		-0.00005
22	0.97801		+0.00056	
		-0.01013		
23	0.96788			

If we at this point try to predict the root for P = 24 by extrapolation from the differences, we find:

0.96788 + (-0.01013 + (+0.00056 + (-0.00005))) = 0.95826

Insertion shows that this is very close to the correct root.

4.5.1. The Procedure ROOT4. This extrapolation to the next root in a series of roots by means of the differences may be carried out with the procedure ROOT4, for which the declaration is shown on the next page.

```
procedure ROOT4 (q, n, x0, del0x, xmin, xmax, eps, epsy, dx, x, y);
value n, x0, del0x, xmin, xmax, eps, epsy;
integer q, n;
boolean epsy;
real x0, del0x, xmin, xmax, eps, x, y;
array dx;
begin
   integer j;
   boolean first;
   real delx, yold, ynew, A;
   procedure incr(z);
   real z;
   begin
      A := z;
L1:   if -, epsy then
      begin
         if eps - abs(A)≥ 0 ∧ -, first then go to L4
      end;
      if x + A < xmin ∨ x + A > xmax then
      begin
         A := 0.5×A;
         go to L1
      end;
      delx := A;
      x := x + delx
   end incr;
   first := true;
   x := if q=0 then x0 else dx[0];
   if q≤1 then go to L3;
   delx := 0;
   for j:=2 step 1 until q do
   delx := delx + dx[j-1];
   incr(delx);
L3:yold := ynew;
   ynew := y;
   if epsy∧eps - abs(ynew)≥0 then go to L4;
```

```
   if first then
   begin
      incr(del0x);
      first := false;
      go to L3
   end first;
   delx := if yold ≠ ynew then delx×ynew/(yold-ynew) else -0.5×delx;
   incr(delx);
   go to L3;
L4:if q=0 then go to L5;
   A := x;
   for j:=1 step 1 until q do
   A := dx[j-1] := A - dx[j-1];
   if q<n then dx[q] := dx[q-1];
   if q=1 then go to L5;
   for j:=q-1 step -1 until 1 do
   dx[j] := dx[j-1];
L5:dx[0] := x;
   if q<n then q := q + 1
end ROOT4;
```

The parameters in ROOT4 are:

integer q. A counter which must be set to zero before the first call of ROOT4. The procedure adds 1 here after each call, until it reaches n.

integer n. The required order of the difference table.

real x0. Start value of x.

real del0x. First increment in x.

real xmin, xmax. Lower and upper limits of x.

real eps. The permissible error in x or y.

boolean epsy. Is specified as true, if eps refers to y, and as false if it refers to x.

array dx[0:n]. Used by the procedure for storage of the differences.

real x. The independent variable. Contains the root at the exit.

real y. The given function (expression) for which we must find the root.

The program d-394 shows the application of ROOT4 for the calculation of the root for the 11 values of P in the example above. The program is shown on the next page.

```
Program d-394. Test of ROOT4.
begin
   integer P, Q, J;
   real X;
   array diffx[0:4];
   copy ROOT4 <
   real procedure Y;
   begin
      Y := X↑6 + P×X↑5 - 20×X - 1;
      J := J + 1;
   end Y;
   Q := 0;
   select(8);
   writetext({<
Output d-394:

 P  Iteration     X                    Y
});
   for P := 20 step 1 until 30 do
   begin
      writecr;
      write({-dd}, P);
      J := 0;
      ROOT4(Q, 4, 0.9, 0.025, -100, 100, 1₁₀-5, false, diffx, X, Y);
      write({-ddddd}, J);
      write({    -d.ddddddd₁₀-dd}, X, Y);
   end for P;
   writecr;
end;
```

The output is shown on the next page.

Output d-394:

P	Iteration	X	Y
20	6	1.0000040	3.4344196 $_{10}$-4
21	5	9.8869904 $_{10}$-1	1.7404556 $_{10}$-5
22	4	9.7800693 $_{10}$-1	-1.9073486 $_{10}$-4
23	3	9.6787753 $_{10}$-1	2.8550625 $_{10}$-4
24	3	9.5824559 $_{10}$-1	8.6784363 $_{10}$-5
25	3	9.4907493 $_{10}$-1	-1.6885996 $_{10}$-4
26	3	9.4033272 $_{10}$-1	-1.9788742 $_{10}$-5
27	3	9.3197938 $_{10}$-1	3.9458275 $_{10}$-5
28	3	9.2398415 $_{10}$-1	-5.2630901 $_{10}$-5
29	3	9.1632312 $_{10}$-1	-4.1663647 $_{10}$-5
30	3	9.0897147 $_{10}$-1	5.5432320 $_{10}$-6

The essential part of the program is a for-statement controlled by P. For each value of P ROOT4 is called once, and we write a line with the number of iterations, the value of X, and the value of Y. We have put n = 4, and we see how the number of iterations decreases from 6 to 3 when the difference table is built up.

ROOT4 operates as follows. There is a local procedure, incr(z), which increases x by z. If we then come outside the range from xmin to xmax, z is halved until the condition is satisfied.

As the start value in the root determination we first use x0. In the next trials, the start value is extrapolated from the difference table. The iteration within a single call of ROOT4 uses the normal regula falsi expression:

(4.62) delx = delx×ynew/(yold-ynew)

In order to avoid division by zero, we use halving of the previous interval, if yold=ynew.

When the root has been found, the new differences are calculated before the exit.

The extrapolation principle in ROOT4 can be extended to functions of several variables.

5. OPTIMIZATION

5.1. Basic Principles.

A very important practical computer problem for which efficient methods are required, is to find a maximum value of a function, i.e. to find a point where the function value is greater than or equal to the function values in all other points in the interval of definition.

If we have a function of, say 3 variables:

$$Y = F(X1, X2, X3) \tag{5.1}$$

we have seen in Chapter 3 how to generate a quadratic model which is an approximation to the original function near the basic point around which the model was generated. If we assume, that the maximum is actually situated in this region and not on some boundary curve of the definition region, the necessary condition for the maximum point is, that the three derivatives are all zero:

$$\begin{aligned} dY/dX1 &= 0 \\ dY/dX2 &= 0 \\ dY/dX3 &= 0 \end{aligned} \tag{5.2}$$

For a quadratic model the derivatives will be linear expressions, and we can solve the three linear equations in the three unknowns: the coordinates of the maximum point. This approach is used by the procedure OPTQUA1 described in section 5.2.

The solution of equations (5.2) can yield a maximum, a minimum, or a saddle point. Section 5.2 explains how to dislinguish between these possibilities. Examples are also given of the use of OPTQUA1.

If the original function is approximated by a linear model, the optimization calculation can be made by the method of steepest descents, described in section 5.3. This method is not so efficient as the quadratic method.

It is possible to use methods which do not require a detailed algebraic treatment of the functions or their models. Two methods of this type are described here:

The procedure OPT1B explores a function of a single variable in discrete points with a fixed distance. A secondary function is tested at the same time, and the procedure finds the maximum of the first function with the side condition that the secondary function is non-negative. The procedure can also be used for functions of several variables, if one variable is treated at a time.

The second procedure, DIRSEARCH, is the direct search or pattern search method described in the literature. It finds the optimum of a function of several variables by simple upward and downward moves of the variables. Successful patterns of moves in all the variables are applied at appropriate places of the search.

In the present context DIRSEARCH works on a single function only. An extended use of the procedure can be obtained by combining it with generation of quadratic models from the true functions available, and the inclusion of side conditions (equalities or inequalities). The latter are handled by means of penalties, i.e. the object function is reduced by an amount which increases as the non-fulfilment of the side conditions become larger. The direct search optimization is carried out in cycles using increasing penalty coefficients. The adaption of the method to take care of side conditions was made by Mr. E. Balslev. The method is described in section 5.5.

The classical method of linear programming is not discussed here. It is described in many different books. The basic principle is the optimization of a linear function with a large number of simultaneous, linear side conditions.

5.2. Quadratic Optimization.

When a quadratic model is available for a function of VAR variables, it is very easy to find a true optimum of the function, i.e. a point where all the derivatives are zero. A further check will then reveal whether this point is a minimum, a maximum, or a saddle point.

If we take the example VAR = 3, the quadratic model is shown on page 41 together with the corresponding three first-order derivates. If we put these derivatives equal to zero, we get the three linear equations:

$$2\times MOD[5,1]\times xact[1] + MOD[8,1]\times xact[2] + MOD[9,1]\times xact[3] = -MOD[2,1]$$
$$MOD[8,1]\times xact[1]+2\times MOD[6,1]\times xact[2] + MOD[10,1]\times xact[3] = -MOD[3,1]$$
$$MOD[9,1]\times xact[1] + MOD[10,1]\times xact[2]+2\times MOD[7,1]\times xact[3] = -MOD[4,1]$$

A simple optimization procedure can now be made as follows. The procedure generates the VAR×(VAR+1) matrix defining the linear equations and solves these by means of LEQ1 or a similar procedure. The solution defines the optimum point.

5.2.1. The Procedure OPTQUA1. This procedure works after this method and has the declaration:

```
integer procedure OPTQUA1(cycount, cymax, VAR, FUNC, MOD, MAT, out,
maxf, eps, xstart, delx, delOx, xact, epsx, yact, yweigh);
value cymax, VAR, FUNC, maxf, eps;
boolean out;
integer cycount, cymax, VAR, FUNC;
real maxf, eps;
array MOD, MAT, xstart, delx, delOx, xact, epsx, yact, yweigh;
begin
   boolean good;
   integer i, j, k, m;
   real R, S;
   procedure SCAN(base);
   value base;
   integer base;
   begin
      R := 0;
      for m := 1 step 1 until FUNC do
      R := R + MOD[base, m]×yweigh[m];
   end SCAN;
   k := 1 + 2×VAR;
   for i := 1 step 1 until VAR do
   begin
      SCAN(1 + i);
      MAT[i, VAR+1] := -R;
      SCAN(1 + VAR + i);
      MAT[i, i] := 2×R;
```

```
   if i < VAR then
   for j := i + 1 step 1 until VAR do
   begin
      k := k + 1;
      SCAN(k);
      MAT[i, j] := MAT[j, i] := R;
   end for j and if i
end for i;
OPTQUA1 := i := - LEQ1(VAR, 1, MAT, eps);
if i = 0 then
begin
   good := true;
   S := 0;
   for i := 1 step 1 until VAR do
   begin
      xact[i] := MAT[i, VAR+1];
      delx[i] := xact[i] - xstart[i];
      if abs(delx[i]) > abs(epsx[i]) then good := false;
      R := abs(delx[i]/del0x[i]);
      if R > S then S := R;
   end for i;
   if S > maxf then
   begin
      for i := 1 step 1 until VAR do
      begin
         delx[i] := delx[i]/S×maxf;
         xact[i] := xstart[i] + delx[i];
      end for i;
   end if large;
   S := 1;
   for S := S×0.5 while out do
   begin
      for i := 1 step 1 until VAR do
      begin
         delx[i] := 0.5×delx[i];
         xact[i] := xstart[i] + delx[i];
      end for i;
   end for S;
```

```
        if good then OPTQUA1 := 1 else
        begin
           cycount := cycount + 1;
           if cycount > cymax then OPTQUA1 := -2;
        end not good;
        for i := 1 step 1 until VAR do xstart[i] := xact[i];
        MODVAL1(VAR, FUNC, (VAR+1)×(VAR+2):2, 0, MOD, xact, yact);
     end if not pivot trouble;
  end OPTQUA1;
```

The procedure uses only variables and arrays stored in the core. The parameters are:

integer cycount. A counter increased by 1 in each call of the procedure. No initial resetting is required.

integer cymax. A maximum value of cycount.

integer VAR. The number of independent variables.

integer FUNC. The number of functions included in the model. As we assume that the model is quadratic, OBS is not a formal parameter (OBS = (VAR+1)×(VAR+2):2).

array MOD[1: OBS, 1: FUNC]. The quadratic model. Must be available before the call of the procedure.

array MAT[1: VAR, 1: VAR+1]. This auxiliary array is used by the procedure for storage of the matrix and solution of the equations.

boolean out. This global procedure must yield the value true, if the calculated optimum point, xact[1: VAR], is outside a permitted range, otherwise false.

real maxf. The procedure checks that the range in variable no. i does not exceed maxf×del0x[i] for all i. If the change is too large, all changes are reduced accordingly.

real eps. Minimum permissible pivot in LEQ1.

array xstart [1: VAR]. Basis point of the independent variables. Must be available before the call. The optimum x-values are also assigned to xstart at the end of the call.

array delx[1: VAR]. Contains the actual change in the x-values after the call.

array del0x[1: VAR]. Increments in the independent variables.

array epsx[1: VAR]. The permissible error in calculation of the maximum point. When the calculated change in xact[i] is less than epsx[i] for all variables, the value of OPTQUA1 is set to 1. The two arrays, del0x and epsx, are not changed by the procedure.

array yact[1: FUNC]. The procedure calculates the function values in the optimum point by a call of MODVAL1 and stores them in yact.

array yweigh[1: FUNC]. This is a set of weights to be applied to the y-values. The procedure operates on the model of FUNC functions, but it can only optimize a single of these, or a linear combination of them. If we have FUNC = 3 and want to optimize function no. 2, we must assign:

```
yweigh[1] := 0;
yweigh[2] := 1;
yweigh[3] := 0;
```

The procedure multiplies all coefficients extracted from the quadratic model by the corresponding value of yweigh.

integer OPTQUA1. After the call, this will have one of the values:

```
OPTQUA1 = -2:    cycount > cymax.
OPTQUA1 = -1:    pivot trouble in LEQ1.
OPTQUA1 =  0:    OK, go on.
OPTQUA1 =  1:    Solution found.
```

The global procedure, LEQ1, must be available to the procedure.

The procedure contains a local procedure, SCAN, which forms the product sum of one set of coefficients and yweigh.

The procedure first generates the matrix, MAT, and then calls LEQ1. If there is no pivot trouble, the values of xact are extracted from the last column in MAT. Then delx is calculated as:

```
(5.3)        delx[i] := xact[i] - xstart[i];
```

and the maximum value, S, of abs(delx[i]/del0x[i]) is found. If S > maxf, all increments are reduced accordingly. The procedure, out, is called, and if it is true, all increments are halved. This is repeated until out becomes false.

Finally, xstart is put equal to xact, and yact at the optimum point is found by a call of MODVAL1. The weights are not used in this evaluation.

5.2.2. Check for Minimum-Maximum. The OPTQUA1 procedure described above finds the point where the first-order derivatives of the quadratic model are zero. It is not checked whether this point corresponds to a maximum, a minimum, or a saddle point. We shall now see, how this can be done.

Consider first the case of a function of a single varaible. The quadratic model is:

(5.4) $$MOD[1,1]+MOD[2,1]\times xact[1]+MOD[3,1]\times xact[1]\uparrow 2$$

and the derivative:

(5.5) $$MOD[2,1]+2\times MOD[3,1]\times xact[1]$$

The derivative is zero in the point:

(5.6) $$x[1] := -MOD[2,1]/(2\times MOD[3,1])$$

If we introduce a new coordinate:

(5.7) $$xnew := xact[1] - x[1]$$

with origin in the point where the derivative is zero, the quadratic model may be written:

(5.8) $$k := MOD[3,1]\times xnew\uparrow 2$$

This is a parabola which clearly has a maximum, if $MOD[3,1] < 0$ and a minimum, if $MOD[3,1] > 0$. The special case of $MOD[3,1] = 0$ corresponds to a straight line.

The situation is more complicated when VAR > 1. It is normally possible to perform a transformation of the coordinate system, so that the quadratic model can be written with the quadratic terms only. For VAR = 3 we can get:

(5.9) $\quad k + MNEW[1]\times xnew[1]\uparrow 2 + MNEW[2]\times xnew[2]\uparrow 2 + MNEW[3]\times xnew[3]\uparrow 2$

If the three coefficients MNEW[1: 3] are all positive, we have a minimum, and if they are all negative, we have a maximum. If there are positive and negative coefficients, we have a saddle point.

It may be shown (see Korn and Korn (1961), p. 316 and 372) that the test for maximum or minimum is equivalent to a test for negative or positive definiteness of the quadratic form made up from the square terms in the model. For VAR = 3 we must consider the matrix (see page 98):

(5.10)
$$\begin{matrix} 2\times MOD[5,1] & MOD[8,1] & MOD[9,1] \\ MOD[8,1] & 2\times MOD[6,1] & MOD[10,1] \\ MOD[9,1] & MOD[10,1] & 2\times MOD[7,1] \end{matrix}$$

The test for negative or positive definiteness of this symmetric matrix is equivalent to a test for negativeness or positiveness of the eigenvalues of the matrix. Special procedures are available for determination of eigenvalues of matrices. If may be shown, that the eigenvalue determination is equivalent to a transformation of the coordinate system which transforms the matrix above into a diagonal matrix:

(5.11)
$$\begin{matrix} MNEW[1] & 0 & 0 \\ 0 & MNEW[2] & 0 \\ 0 & 0 & MNEW[3] \end{matrix}$$

The eigenvalues are then the three diagonal terms, MNEW[1:3], for which we must investigate the sign. The eigenvalue problem can also be formulated as a polynomial of degree VAR, having the eigenvalues as the roots. In any case we are up to a fairly complicated calculation, except for small values of VAR.

These difficulties are avoided in the direct optimization methods.

5.2.3. Example 1. Optimization of Quadratic Function. This example illustrates the simple case of a purely quadratic function. We first choose a function of the form:

(5.12) $y = 10 - x[1]\uparrow 2 - x[2]\uparrow 2$

This function has a maximum in (0,0) and the contour lines are circles around this point. If we change the function into:

(5.13) $y = 10 - x[1]\uparrow 2 - 4\times x[2]\uparrow 2$

the contour lines become ellipses, but the maximum is still at (0,0). If we want the maximum in another point, say in (5,5), we must write:

(5.14) $y = 10 - (x[1] - 5)\uparrow 2 - 4\times(x[2] - 5)\uparrow 2$

The axes of the contour ellipses are parallel to the coordinate axes. Rotation of the coordinate system can be made by applying the normal formulas for this transformation. If the system is rotated the angle alfa around the origin (0,0) the relations between the new and the old coordinates become:

(5.15) $xold1 = xnew1\times\cos(alfa) - xnew2\times\sin(alfa)$
$xold2 = xnew1\times\sin(alfa) + xnew2\times\cos(alfa)$

Therefore, if we replace the coordinates:

(5.16) $u = x[1] - 5$
$v = x[2] - 5$

by the new coordinates:

(5.17) $xnew1 = 0.8\times u - 0.6\times v$
$xnew2 = 0.6\times u + 0.8\times v$

we have rotated the coordinate system an angle -alfa around the point (5,5), in which alfa is determined from:

(5.18) $\cos(alfa) = 0.8$

The final expression for the function we want to investigate then becomes:

(5.19) $y = 10-(0.8\times(xact[1]-5)-0.6\times(xact[2]-5))\uparrow 2$

$- 4\times(0.6\times(xact[1]-5)+0.8\times(xact[2]-5))\uparrow 2$

Program d-354 finds the maximum of this function. The program is:

```
Program d-354.  Test of OPTQUA1 with simple maximum.
begin
   integer count, cycount, state, i;
   array xstart, del0x, delx, xact, epsx[1:2],
   yact, yweigh[1:1], MAT[1:6,1:7], MOD[1:6,1:1];
   copy GENMOD1 <
   copy LEQ1 <
   copy MODWL1 <
   copy OPTQUA1 <
   xstart[1] := 1;
   xstart[2] := 2;
   del0x[1] := del0x[2] := 0.5;
   epsx[1] := epsx[2] := 0.0001;
   yweigh[1] := 1;
   count := cycount := 0;
   select(8);
   writetext(<<
Output d-354
    x[1]        x[2]        y
>>);
H: state := GENMOD1(count, 2, 1, 6, xstart, del0x, xact,
   yact, 10-12, MAT, MOD);
   yact[1] := 10-(0.8x(xact[1]-5)-0.6x(xact[2]-5))↑2
   -4x(0.6x(xact[1]-5)+0.8x(xact[2]-5))↑2;
   if state = 0 then
   begin
      writecr;
      write(<-dddd.dddddd>, xact[1], xact[2], yact[1]);
      go to H;
   end if state = 0;
   writecr;
```

```
    state := OPTQUA1(cycount, 6, 2, 1, MOD, MAT,
    false, 7, 1₁₀-12, xstart, delx, del0x, xact, epsx, yact, yweigh);
    writecr;
    if state ≥ 0 then
    begin
       write({-dddd.dddddd}, xact[1], xact[2], yact[1]);
       writecr;
       count := 0;
       if state = 0 then go to H;
    end if state;
    writecr;
    for i := 1 step 1 until 6 do
    begin
       writecr;
       write({-d.dddddd₁₀-dd}, MOD[i,1]);
    end for i;
    writecr;
end;
```

Output from the program is:

```
Output d-354
    x[1]        x[2]        y

    1.000000    2.000000   -84.120000
    1.500000    2.000000   -72.000000
    1.000000    2.500000   -70.330000
    0.500000    2.000000   -97.280000
    1.000000    1.500000   -99.370000
    1.500000    2.500000   -58.930000

    4.500000    4.625001    8.529375
```

```
4.500000    4.625001    8.529378
5.000000    4.625001    9.589377
4.500000    5.125001    9.614376
4.000000    4.625001    6.429379
4.500000    4.125001    5.984380
5.000000    5.125001    9.954374

5.000000    5.000000    9.999999

5.000000    5.000000   10.000000
5.500000    5.000000    9.480000
5.000000    5.500000    9.270000
4.500000    5.000000    9.480000
5.000000    4.500000    9.270000
5.500000    5.500000    8.030000

5.000000    5.000000    9.999999


-1.869999 10 2
 3.519997 10 1
 4.360000 10 1
-2.079997
-2.920000
-2.880000
```

The program starts by assigning start values to xstart and setting the increments. We then enter a cycle in which GENMOD1 is called six times for generation of a quadratic model. The values of x1, x2, and the function are written out. After the six calls of GENMOD1, the model has been generated and stored in MOD[1:6,1:1]. We then call the procedure OPTQUA1 for determination of the optimum point. The calculated point is printed. We first get:

$$x1 = 4.5, \quad x2 = 4.625$$

The reason why we do not get the true maximum (5,5) at once is that we have used maxf = 7. As the move from x1 = 1 to x1 = 5 is 8 times the value of del0x[1] the procedure reduces this to 7 times del0x or 3.5. The calls of GENMOD1 are then repeated with generation of a new model around the point (4.5, 4.625). A new call of OPTQUA1 then takes us to the point (5,5). Finally, the calls of GENMOD1 and OPTQUA1 are repeated to bring the increments in x below the specified tolerance, 0.0001.

At the end of the calculation the program prints the six coefficients in the model:

$$y = -187 + 35.2 \times x1 + 43.6 \times x2 - 2.08 \times x1\uparrow 2 - 2.92 \times x2\uparrow 2 - 2.88 \times x1 \times x2$$

This expression is, of course, identical to what is obtained if we rearrange the expression for yact[1] used in the program. As an example of the eigenvalue concept, let us verify that we have a true maximum. We must then consider the quadratic form (see page 103):

$$\begin{matrix} 2 \times MOD[4,1] & MOD[6,1] \\ MOD[6,1] & 2 \times MOD[5,1] \end{matrix}$$

Insertion of the numerical values gives:

$$\begin{matrix} -4.16 & -2.88 \\ -2.88 & -5.84 \end{matrix}$$

If this symmetric matrix is run on an eigenvalue program, we find the two eigenvalues to:

$$-2 \quad \text{and} \quad -8$$

and the corresponding eigenvectors:

$$\begin{matrix} (1) & 0.8 & -0.6 \\ (2) & 0.6 & 0.8 \end{matrix}$$

As the eigenvalues are both negative we have a true maximum according to this theory. Furthermore, we recognize the two eigenvectors as those defining the rotation we made on the coordinate system.

5.2.4. Example 2. Optimization with Lagrange Multiplier. This calculation simulates the optimization of an ammonia converter by means of the test procedure, POL, described on page 84. We wish to calculate the ammonia production:

(5.20) PROD := POL(1, tinlet, ginlet);

as a function of the inlet temperature, tinlet, to the first bed and the relative gas flow, ginlet, in the same bed. PROD must be a maximum with the simultaneous condition, that the necessary height of the lower exchanger is exactly 2 meters. The second function:

(5.21) EXCESS := POL(2, tinlet, ginlet) - 2;

must then be zero. Optimization of a function with the simultaneous constraint that one or more other functions of the same variables must be zero, may be solved by the use of the so-called Lagrange multipliers. Instead of the two functions, PROD and EXCESS, we make a single new function:

(5.22) yact[1] := PROD + lambda×EXCESS;

The new variable, lambda, is the undetermined Lagrange multiplier, which must satisfy the condition:

(5.23) dyact[1]/dlambda = 0

We also have:

(5.24) dyact[1]/dtinlet = 0
(5.25) dyact[1]/dginlet = 0

and we can then solve the problem by straightforward use of OPTQUA1 with a model based upon the three variables: tinlet, ginlet, and lambda. We utilize that OPTQUA1 does not search for a true optimum, but simply a point where the derivatives are zero.

If there are further functions which must also be zero, they must be included in yact after multiplication by lambda2, lambda3, etc.

The program is:

```
Program d-355.  Optimization with Lagrange Multiplier.
begin
   integer i, count, cycount, state;
   array xstart, del0x, delx, xact, epsx, yact, yweigh[1:3],
   MAT[1:10, 1:11], MOD[1:10, 1:1];
   copy LEQ1 <
   copy POL <
   copy GENMOD1<
   copy MODVAL1<
   copy OPTQUA1<
   for i := 1 step 1 until 3 do
   begin
      xstart[i] := case i of (430, 70, -1);
      del0x[i] := case i of (10, 2, 0.1);
      epsx[i] := case i of (1, 0.1, 0.01);
   end for i;
   yweigh[1] := 1;
   count := cycount := 0;
   select(8);
   writetext(<<
Output d-355
    x[1]      x[2]      x[3]      yact[1]     PROD        EXCESS
   tinlet    ginlet    lambda
>);
H: state := GENMOD1(count, 3, 1, 10, xstart, del0x, xact, yact, 10-12,
   MAT, MOD);
   yact[2] := POL(1, xact[1], xact[2]);
   yact[3] := POL(2, xact[1], xact[2]) - 2;
   yact[1] := yact[2] + xact[3]xyact[3];
   if state = 0 then
   begin
      writecr;
      for i := 1 step 1 until 3 do write(<-dddd.dddd>, xact[i]);
      for i := 1 step 1 until 3 do write(<-dddd.dddd00>, yact[i]);
      go to H;
   end if state = 0;
   writecr;
```

```
   state := OPTQUA1(cycount, 6, 3, 1, MOD, MAT, false, 4, 1₁₀-12,
   xstart, delx, del0x, xact, epsx, yact, yweigh);
   if state ≥ 0 then
   begin
      for i := 1 step 1 until 3 do write(⟨-dddd.dddd⟩, xact[i]);
      write(⟨-dddd.dddd00⟩, yact[1]);
      writecr;
      count := 0;
      if state = 0 then go to H;
   end if state;
   writecr;
   for i := 1 step 1 until 10 do
   begin
      writecr;
      write(⟨-d.ddddddd₁₀-dd⟩, MOD[i, 1]);
   end for i;
   writecr;
end;
```

The output is:

Output d-355

x[1] tinlet	x[2] ginlet	x[3] lambda	yact[1]	PROD	EXCESS
430.0000	70.0000	-1.0000	72.363989	72.636263	0.272274
440.0000	70.0000	-1.0000	72.441857	72.853762	0.411905
430.0000	72.0000	-1.0000	72.694273	72.844715	0.150442
430.0000	70.0000	-0.9000	72.391216	72.636263	0.272274
420.0000	70.0000	-1.0000	71.922982	72.109606	0.186624
430.0000	68.0000	-1.0000	71.579330	72.082275	0.502946
430.0000	70.0000	-1.1000	72.336761	72.636263	0.272274
440.0000	72.0000	-1.0000	72.426437	72.719479	0.293042
440.0000	70.0000	-0.9000	72.483047	72.853762	0.411905
430.0000	72.0000	-0.9000	72.709317	72.844715	0.150442
424.7773	72.7437	-0.6000	72.776750		

424.7773	72.7437	-0.6000	72.793084	72.830400	0.062192
434.7773	72.7437	-0.6000	72.639308	72.749701	0.183989
424.7773	74.7437	-0.6000	72.714877	72.713295	-0.002637
424.7773	72.7437	-0.5000	72.799304	72.830400	0.062192
414.7773	72.7437	-0.6000	72.552436	72.534840	-0.029326
424.7773	70.7437	-0.6000	72.483194	72.581715	0.164202
424.7773	72.7437	-0.7000	72.786865	72.830400	0.062192
434.7773	74.7437	-0.6000	72.274788	72.348102	0.122190
434.7773	72.7437	-0.5000	72.657706	72.749701	0.183989
424.7773	74.7437	-0.5000	72.714613	72.713295	-0.002637
422.2666	73.7190	-0.6234	72.815140		
422.2666	73.7190	-0.6234	72.818766	72.818893	0.000203
432.2666	73.7190	-0.6234	72.602960	72.677007	0.118777
422.2666	75.7190	-0.6234	72.655633	72.620283	-0.056705
422.2666	73.7190	-0.5234	72.818786	72.818893	0.000203
412.2666	73.7190	-0.6234	72.638873	72.578391	-0.097019
422.2666	71.7190	-0.6234	72.619359	72.672112	0.084620
422.2666	73.7190	-0.7234	72.818746	72.818893	0.000203
432.2666	75.7190	-0.6234	72.161292	72.201301	0.064176
432.2666	73.7190	-0.5234	72.614837	72.677007	0.118777
422.2666	75.7190	-0.5234	72.649963	72.620283	-0.056705
422.3182	73.7476	-0.4211	72.818863		
422.3182	73.7476	-0.4211	72.818596	72.818519	-0.000184
432.3182	73.7476	-0.4211	72.620635	72.670565	0.118562
422.3182	75.7476	-0.4211	72.637805	72.613873	-0.056827
422.3182	73.7476	-0.3211	72.818578	72.818519	-0.000184
412.3182	73.7476	-0.4211	72.626126	72.584933	-0.097814
422.3182	71.7476	-0.4211	72.643439	72.678607	0.083510
422.3182	73.7476	-0.5211	72.818614	72.818519	-0.000184
432.3182	75.7476	-0.4211	72.162719	72.189762	0.064213
432.3182	73.7476	-0.3211	72.632491	72.670565	0.118562
422.3182	75.7476	-0.3211	72.632122	72.613873	-0.056827
422.3012	73.7339	-0.4196	72.818600		

-9.4990113 $_{10}$ 2
2.6754309
1.2401000 $_{10}$ 1
-2.9263114
-1.9521338 $_{10}$-3
-4.4493727 $_{10}$-2
7.9626490 $_{10}$-7
-1.3856203 $_{10}$-2
1.1874467 $_{10}$-2
-2.8321968 $_{10}$-2

The program contains the normal sections: Setting of start values, generation of model with GENMOD1, and the call of OPTQUA1. The calculated maximum production occurs for tinlet = 422 deg. C and ginlet = 73.7 per cent. Four cycles each with 10 function evaluations are required here.

It is a drawback in this use of a Lagrange multiplier that a start value of lambda must be available, having the correct order of magnitude and sign. This may be difficult to obtain in practice.

5.2.5. Example 3. Optimization with Elimination. This is exactly the same problem as example 2, but now we use elimination instead of the Lagrange multiplier. A model of PROD and EXCESS is generated with the two variables, tinlet and ginlet:

MOD1[1:6,1:2]

From this model we generate a simpler model having only one variable written as xact2[1]. This is actually tinlet. The model is smaller:

MOD2[1:3,1:1]

because there is only one variable and one function. For each value of xact2[1] a local procedure, ELIM, is called which eliminates ginlet by a call of NOLEQ8 (solution of non-linear equations). This elimination uses values calculated from MOD1 by means of MODVAL1.

When MOD2 has been generated we call OPTQUA1 to determine the optimum of the latter function.

The program is:

Program d-356. Optimization with elimination.

```
begin
   boolean first, deriv;
   integer count1, count2, count3, cyc3, cyc4,
   state1, state2, state3, state4, i;
   array xstart1, xstart2, xstart3, del0x, epsx, yweigh, xact1, neps,
   xact2, xact3, yact1, yact2, yact3, delx, y0[1:2], yold[1:1, 1:2],
   MAT1[1:6, 1:8], MAT2[1:3, 1:4], MOD1[1:6, 1:2], MOD2[1:3, 1:1];
   copy GENMOD1<
   copy LEQ1<
   copy MODVAL1<
   copy NOLEQ8<
   copy OPTQUA1<
   copy POL<
   procedure ELIM;
   begin
      count3 := cyc3 := 0;
      first := true;
      xstart3[1] := xstart1[2];
A:    state3 := NOLEQ8(1, count3, cyc3, 1, 10, false, first, false,
      false, deriv, xstart3, del0x, delx, xact3, neps, yact3, y0, yold,
      1_10-20, 4);
      xact1[2] := xact3[1];
      MODVAL1(2, 2, 6, 0, MOD1, xact1, yact1);
      yact3[1] := yact1[2];
      if state3 = 0 then go to A;
   end ELIM;
   procedure PRINT;
   begin
      writecr;
      write(<-dddd.dddd>, xact1[1], xact1[2]);
      write(<-dddd.dddd00>, yact1[1], yact1[2]);
   end PRINT;
```

```
   select(8);
   writetext(‡<
Output d-356
    x[1]      x[2]      y[1]      y[2]
   tinlet    ginlet     PROD      EXCESS
‡);
   xstart1[1] := 420;
   xstart1[2] := 72;
   del0x[1] := del0x[2] := 2;
   epsx[1] := 0.1;
   epsx[2] := 0.01;
   neps[1] := 0.001;
   yweigh[1] := yweigh[2] := 1;
   count1 := cyc4 := 0;
B: state1 := GENMOD1(count1, 2, 2, 6,
   xstart1, del0x, xact1, yact1, 1₁₀-20, MAT1, MOD1);
   yact1[1] := POL(1, xact1[1], xact1[2]);
   yact1[2] := POL(2, xact1[1], xact1[2]) - 2;
   if state1 = 0 then
   begin
      PRINT;
      go to B;
   end if state1 = 0;
   writecr;
   count2 := 0;
   xstart2[1] := xstart1[1];
C: state2 := GENMOD1(count2, 1, 1, 3,
   xstart2, del0x, xact2, yact2, 1₁₀-20, MAT2, MOD2);
   if state2 = 0 then
   begin
      xact1[1] := xact2[1];
      ELIM;
      yact2[1] := yact1[1];
      go to C;
   end if state2 = 0;
   state4 := OPTQUA1(cyc4, 6, 1, 1,  MOD2, MAT2,
   false, 4, 1₁₀-12, xstart2, delx, del0x, xact1, epsx, yact1, yweigh);
```

```
    writecr;
    ELIM;
    PRINT;
    xstart1[1] := xstart2[1];
    xstart1[2] := xact1[2];
    count1 := 0;
    if state4 = 0 then go to B;
    writecr;
end;
```

The output is:

Output d-356

x[1] tinlet	x[2] ginlet	y[1] PROD	y[2] EXCESS
420.0000	72.0000	72.635492	0.051393
422.0000	72.0000	72.706796	0.067741
420.0000	74.0000	72.801409	-0.033774
418.0000	72.0000	72.548644	0.036520
420.0000	70.0000	72.109606	0.186624
422.0000	74.0000	72.810282	-0.011943
422.9529	73.9651	72.810839	-0.000108
422.9529	73.9651	72.810956	-0.000174
424.9529	73.9651	72.798600	0.022667
422.9529	75.9651	72.553741	-0.055020
420.9529	73.9651	72.808248	-0.022335
422.9529	71.9651	72.730827	0.077809
424.9529	75.9651	72.480859	-0.032747
422.1802	73.7045	72.817314	0.000003
422.1802	73.7045	72.818756	-0.000263
424.1802	73.7045	72.820461	0.022071
422.1802	75.7045	72.625135	-0.057292
420.1802	73.7045	72.801550	-0.021818
422.1802	71.7045	72.666651	0.084694
424.1802	75.7045	72.564250	-0.035254
422.1723	73.6945	72.818811	0.000005

The result shows that we now need only 3×6 = 18 function evaluations instead of 40 with the Lagrange multiplier. The internal elimination of one of the variables takes some time, but as the elimination is made on the model, not the original functions, this is not significant. Troubles may occur, of course, if we are far from the desired point and the elimination becomes impossible.

5.3. Method of Steepest Descents.

This optimization method does not use a quadratic model, but only a linear model. It does not immediately yield the maximum or minimum of the function, but tells us in what direction to move in order to get the highest increase (or decrease) in the function. The necessary calculations are very simple. For VAR = 3 the linear model has the form:

(5.26) MOD[1,1]+MOD[2,1]×xact[1]+MOD[3,1]×xact[2]+MOD[4,1]×xact[3]

We calculate the sum of the squares of the derivatives:

(5.27) SSQ := MOD[2,1]↑2+MOD[3,1]↑2+MOD[4,1]↑2

The directional cosines of the desired line are:

(5.28) MOD[2,1]/sqrt(SSQ), MOD[3,1]/sqrt(SSQ), MOD[4,1]/sqrt(SSQ)

If we move from the basis point:

xstart[1], xstart[2], xstart[3]

with increments proportional to the directional cosines, we will move along the fastest increase. The question is then how far to move in that direction. This can be done in different ways. We can either make a number of equidistant steps and stop as soon as the original function starts to decrease, or we can make two steps, calculate the original function in these points, fit a parabola to the three known points, and finally go to the maximum of the parabola. When the new basis point has been found from either of the two methods, a new model must be generated around this point.

The method of steepest descent is excellent for investigation of a function far from the maximum. As soon as we approach the maximum, a quadratic optimization is better, because it gives a better definition of the end point of the search. Another possibility is to use the direct methods described in the following sections.

5.4. Direct Method for Single Variable.

A direct optimization method is a method which explores the unknown function after a certain strategy and only performs simple comparisons of the function values thus obtained. No use is made of derivatives or other analytic features. A procedure of this type has been in use for many years at the Haldor Topsøe computer installation and represents a good compromise between simplicity and efficiency. The original version of the procedure (in GIER machine language) was called OPT1 and a later ALGOL version: OPT1A. The latest version, OPT1B, in GIER ALGOL 4 is explained in the following. As the major part of the procedure is taken up by administration, not arithmetic, a detailed description is first given of the strategy used in OPT1B.

5.4.1. Strategy in the Procedure OPT1B. The procedure first of all finds the maximum of a function of a single variable:

$$Y1 = F1(X) \tag{5.29}$$

by variation of X in steps of a fixed size, DELX:

$$\begin{aligned} &X \\ &X + DELX \\ &X + 2\times DELX \\ &X + 3\times DELX \\ &\text{etc.} \end{aligned} \tag{5.30}$$

The increase in X is continued as long as the Y1-values also increase. When they start to decrease, the last but one X-value is selected as the optimal one. This may appear too simple to warrant a special procedure, and OPT1B, therefore, contains some further features.

If the optimum value is lower than the start value of X, we get the following order of the X-values:

(5.31)
X
X + DELX
X - DELX
X - 2×DELX
X - 3×DELX
etc.

A permissible range from XMIN to XMAX must be specified. The procedure will not permit X to grow outside this interval, but selects the optimum value.

The procedure will also take into account a second function of the same variable:

(5.32) $Y2 = F2(X)$

It is required that the value of Y2 must be non-negative ($Y2 \geq 0$). The procedure varies X upwards or downwards until it finds the X-value giving the highest value of Y1 and for which at the same time $Y2 \geq 0$. If the search for maximum Y1 yields negative Y2-values, the corresponding X-values are not considered. We assume that the two functions, F1(X) and F2(X), have not more than one maximum and no minimum inside the given interval. With this assumption the procedure can make its way out of a forbidden interval ($Y2 < 0$) by selecting the direction which makes Y2 increase. The essential part of the procedure is a table permitting the procedure to make the proper decision with the available knowledge of the values and signs of the last three sets of X, Y1, and Y2. Older sets are not considered.

5.4.2. Declaration of OPT1B. This is shown on the following pages.

```
integer procedure OPT1B(COUNT, X, Y1, Y2, XMIN, XMAX,
DELX, XOLD, Y1OLD, Y2OLD, XOPT, Y1OPT, Y2OPT);
value Y1, Y2, XMIN, XMAX;
integer COUNT;
real X, Y1, Y2, XMIN, XMAX, DELX, XOPT, Y1OPT, Y2OPT;
array XOLD, Y1OLD, Y2OLD;
begin
   integer ROW, COL, I, R;
   integer array T2[1:4], T3[1:16];
   switch TABLE2 := UP, DN, S1, S2, ER;
   switch TABLE3 := S1, S2, S3, D1, U3, DE, UE, ER;
   integer procedure NEG(z);
   value z;
   real z;
   NEG := if z < 0 then 1 else 0;
   procedure MOVE(N, M);
   value N, M;
   integer N, M;
   begin
      XOLD[M] := XOLD[N];
      Y1OLD[M] := Y1OLD[N];
      Y2OLD[M] := Y2OLD[N];
   end MOVE;
   OPT1B := 0;
   T2[1] := 2252;
   T2[2] := 2541;
   T2[3] := 1352;
   T2[4] := 1511;
   T3[1] := 44848886;
   T3[2] := 88888888;
   T3[3] := 44882288;
   T3[4] := 28882837;
   T3[5] := T3[6] := T3[7] := T3[8] := 88888888;
   T3[9] := 22818886;
   T3[10] := 88888888;
   T3[11] := 22882288;
   T3[12] := 28882837;
   T3[13] := 52818886;
```

```
      T3[14] := 88888888;
      T3[15] := 52dd52dd;
      T3[16] := 58885857;
      XOLD[1] := X;
      Y1OLD[1] := Y1;
      Y2OLD[1] := Y2;
      if COUNT = 0 then
      begin
         MOVE(1, 2);
         DELX := abs(DELX);
UP:      X := X + DELX;
      end COUNT = 0 else
      if COUNT = 1 then
      begin
         MOVE(1, 3);
         ROW := 1 + NEG(Y2OLD[2]-Y2OLD[3]) + 2×NEG(Y1OLD[2]-Y1OLD[3]);
         COL := 1 + NEG(Y2OLD[3]) + 2×NEG(Y2OLD[2]);
         I := T2[ROW];
         R := 10↑(4-COL);
         go to TABLE2[(I:R) mod 10];
ER:      OPT1B := -1;
         go to EX;
DN:      MOVE(3,4);
         MOVE(2,3);
         R := if COUNT = 1 then 2 else 3×sign(DELX);
         DELX := -abs(DELX);
         if R < 0 then go to UP;
         X := X - R×abs(DELX);
      end if COUNT = 1 else
      begin
         MOVE(1, 3 + sign(DELX));
         ROW := 1 + NEG(Y2OLD[3]-Y2OLD[4]) + 2×NEG(Y2OLD[2]-Y2OLD[3])
         +4×NEG(Y1OLD[3]-Y1OLD[4]) + 8×NEG(Y1OLD[2]-Y1OLD[3]);
         COL := 1 + NEG(Y2OLD[4]) + 2×NEG(Y2OLD[3]) + 4×NEG(Y2OLD[2]);
         I := T3[ROW];
         R := 10↑(8-COL);
         go to TABLE3[(I:R) mod 10];
         go to ER;
S1:      I := 2;
         go to S4;
```

```
S2:   I := 3;
      go to S4;
S3:   I := 4;
S4:   OPT1B := 1;
      XOPT := XOLD[I];
      Y1OPT := Y1OLD[I];
      Y2OPT := Y2OLD[I];
      go to EX;
D1:   go to if XOLD[2] - abs (DELX) ≥ XMIN then DN else S1;
DE:   go to if XOLD[2] - abs(DELX) ≥ XMIN then DN else ER;
U3:   go to if DELX < 0 then ER else if XOLD[4] + DELX≤XMAX
      then L1 else S3;
UE:   if DELX < 0 then go to ER;
      if XOLD[4] + DELX > XMAX then go to ER;
L1:   MOVE(3,2);
      MOVE(4,3);
      go to UP;
   end COUNT > 1;
EX:COUNT := COUNT + 1;
end OPT1B;
```

The formal parameters are:

integer COUNT: A counter which the user must set to zero before the first call. The procedure adds 1 here at the end of each call.

real X: The actual value of the independent variable. The start value must have been inserted here before the first call. The procedure inserts a new value in each call.

real Y1, Y2: The two function values corresponding to X. Before the first call the main program must have inserted the two values corresponding to the start value of X. After each call the main program must also calculate the new Y-values for the new value of X.

real XMIN, XMAX: Lower and upper limits of X.

real DELX: The fixed increment (step length) in X. As the procedure inserts -abs(DELX), if X must be decreased, the actual parameter should be a variable, not a number.

array XOLD, Y1OLD, Y2OLD[1:4]: Used by the procedure for storage of old sets of X, Y1, and Y2. A new set is first stored as element no. 1 and later moved to one of the next elements. Only three old sets are stored.

real XOPT, Y1OPT, Y2OPT: The procedure delivers the optimum set of X, Y1, and Y2 here when it is finished.

integer OPT1B: The procedure is of type integer and can assume the three values:

OTP1B = -1: Optimum cannot be found. This happens, if Y2 is negative for all X, or if the condition of only one maximum and no minimum is not satisfied.
OTP1B = 0: Calculation OK, find next set of Y1 and Y2.
OPT1B = +1: Optimum found.

Two local procedures are used: NEG(z) = 1 for z < 0, otherwise 0, and MOVE(N,M) which moves the old data set no. N to set no. M.

The procedure operates as follows.

The two decision tables, T2[1:4] and T3[1:16], are first given the proper (fixed) content. In FORTRAN this would have been written as a data initialization statement, but that is not available in ALGOL. The tables are further explained below.

The new set of X, Y1, and Y2 is stored in XOLD[1], Y1OLD[1], and Y2OLD[1].

In the first call (COUNT = 0), the new set of X, Y1, and Y2 is moved to XOLD[2], etc. The sign of DELX is set to plus, and X is increased:

(5.33) X = X + DELX

The main program must then calculate new values of Y1 and Y2.

In the next call (COUNT = 1), the new data set is stored in XOLD[3], etc. The program must then study the six numbers in the two sets:

(5.34) XOLD[2], Y1OLD[2], Y2OLD[2]
XOLD[3], Y1OLD[3], Y2OLD[3]

and make a choice between the five possibilities:

UP: X is further increased.
DN: X is decreased.
(5.35) S1: Select first set as the optimum.
S2: Select second set as the optimum.
ER: Error exit, inconsistent data.

The decision is made according to the following table:

Y1	Y2		Sign of Y2			
			++	+-	-+	--
			1	2	3	4
down	down	1	DN	DN	ER	DN
down	up	2	DN	ER	S2	UP
up	down	3	UP	S1	ER	DN
up	up	4	UP	ER	UP	UP

The four rows in the table correspond to the four combinations of increasing or decreasing values of Y1 and Y2 as indicated by the words to the left. The four columns in the table correspond to the four sign combinations of the two Y2-values: ++, +-, -+, and --. The procedure calculates the row number as:

(5.36) ROW = 1+NEG(Y2OLD[2]-Y2OLD[3])+2×NEG(Y1OLD[2]-Y1OLD[3])

and the column number:

(5.37) COL = 1+NEG(Y2OLD[3])+2×NEG(Y2OLD[2])

When the row number and the column number have been found, we must pick out the proper label in the table and make a jump. The program contains a switch:

(5.38) switch TABLE2 := UP, DN, S1, S2, ER;

with the five different labels in the table. We could have extended the

switch to contain 16 labels, but this is not economical. This point of view becomes especially important when we use the big decision table, based upon three data sets, and containing 128 labels of which only 8 are different. The economical programming of this can be made in the following way:

We give each label a number:

(5.39)
```
UP: 1
DN: 2
S1: 3
S2: 4
ER: 5
```

The table of labels:

(5.40)
```
DN  DN  ER  DN
DN  ER  S2  UP
UP  S1  ER  DN
UP  ER  UP  UP
```

is then written as:

(5.41)
```
2  2  5  2
2  5  4  1
1  3  5  2
1  5  1  1
```

and we condense this into the four numbers:

(5.42)
```
T2[1] = 2252
T2[2] = 2541
T2[3] = 1352
T2[4] = 1511
```

We then pick out the correct label be means of the statements:

```
I := T2[ROW];
R := 10↑(4-COL);
go to TABLE2[(I:R) mod 10];
```

If we arrive to the third or later calls (COUNT $\geq$ 2), the new set of data will be stored in XOLD[2], etc., if X is decreasing or in XOLD[4], etc. if X is increasing. The decision table based upon the three old data sets is shown on the opposite page.

Here, we must decide between 8 different possibilities:

(5.43)

S1: Select first set as the optimum.
S2: - second - - - - .
S3: - third - - - - .
D1: Decrease X or select first set.
U3: Increase X - - third - .
DE: Decrease X or go to error exit.
UE: Increase X - - - - - .
ER: Error exit, inconsistent data.

In the two cases D1 and U3 the selection of set 1 or 3 is used, if a further change in X brings us outside the interval from XMIN to XMAX. The same applies to the error possibility in the two cases DE and UE.

The row number in the decision table is selected from the 16 sign combinations of:

Y1OLD[2]-Y1OLD[3], Y1OLD[3]-Y1OLD[4]
Y2OLD[2]-Y2OLD[3], Y2OLD[3]-Y2OLD[4]

and the column number is determined from the 8 possible sign combinations of:

Y2OLD[2], Y2OLD[3], Y2OLD[4]

The sign combinations are written over the 8 columns as +++, ++-, etc. For the rows in the table we use the words:

down, min, max, up

to indicate the relative values of Y1OLD and Y2OLD: decreasing, having a minimum, having a maximum, or increasing. The minimum situation is not in accordance with the assumptions and gives selection of the error label in the table. The same applies to column 3.

Y1	Y2		Sign of Y2							
			+++	++-	+-+	+--	-++	-+-	--+	---
			1	2	3	4	5	6	7	8
down	down	1	D1	D1	ER	D1	ER	ER	ER	DE
down	min	2	ER	ER	ER	ER	ER	ER	ER	ER
down	max	3	D1	D1	ER	ER	S2	S2	ER	ER
down	up	4	S2	ER	ER	ER	S2	ER	S3	UE
min	down	5	ER	ER	ER	ER	ER	ER	ER	ER
min	min	6	ER	ER	ER	ER	ER	ER	ER	ER
min	max	7	ER	ER	ER	ER	ER	ER	ER	ER
min	up	8	ER	ER	ER	ER	ER	ER	ER	ER
max	down	9	S2	S2	ER	S1	ER	ER	ER	DE
max	min	10	ER	ER	ER	ER	ER	ER	ER	ER
max	max	11	S2	S2	ER	ER	S2	S2	ER	ER
max	up	12	S2	ER	ER	ER	S2	ER	S3	UE
up	down	13	U3	S2	ER	S1	ER	ER	ER	DE
up	min	14	ER	ER	ER	ER	ER	ER	ER	ER
up	max	15	U3	S2	ER	ER	U3	S2	ER	ER
up	up	16	U3	ER	ER	ER	U3	ER	U3	UE

The table look-up is made in the same way as for the small decision table.

5.4.3. Examples. Two calculation examples of the use of OPT1B are given here. The ammonia converter test polynomial, POL, is used in both examples (see page 64). In the first case we vary the inlet temperature, tinlet, in steps of 10 deg. C in order to find the maximum ammonia production. At the same time, the excess height of the lower exchanger must be positive. For an actual (physical) exchanger height of 2.1 meter, the excess height is calculated as:

(5.44) 2.1 - POL(2, inlet, ginlet)

The calculation is made for three different values of ginlet: 70, 74, and 78 per cent, but without any connection between the three calculation parts. The program is:

Program d-403. Test of OPT1B with a single variable.

```
begin
   integer COUNT, STATE;
   real X, Y1, Y2, ginlet, DELX;
   array XOLD, Y1OLD, Y2OLD[1:4];
   copy POL <
   copy OPT1B <
   select(8);
   writetext(<<
```

Output d-403

```
tinlet  ginlet  PROD   EXCESS
>);
   for ginlet := 70, 74, 78 do
   begin
      COUNT := 0;
      X := 400;
      DELX := 10;
AA:   writecr;
      Y1 := POL(1, X, ginlet);
      Y2 := 2.1 - POL(2, X, ginlet);
```

```
        write(‹dddddd›, X, ginlet);
        write(‹-ddd.dd›, Y1, Y2);
        STATE := OPT1B(COUNT, X, Y1, Y2, 300, 500, DELX, XOLD,
        Y1OLD, Y2OLD, X, Y1, Y2);
        if STATE = 0 then go to AA;
        writecr;
        write(‹dddddd›, X, ginlet);
        write(‹-ddd.dd›, Y1, Y2);
        writecr;
    end for ginlet;
end;
```

The following output was obtained:

Output d-403

tinlet	ginlet	PROD	EXCESS
400	70	69.89	0.16
410	70	71.24	-0.01
400	70	69.89	0.16
400	74	71.83	0.26
410	74	72.52	0.23
420	74	72.80	0.13
430	74	72.70	0.02
420	74	72.80	0.13
400	78	72.54	0.40
410	78	72.54	0.32
390	78	72.21	0.45
400	78	72.54	0.40

In the second example ginlet is varied in an inner loop and tinlet in an outer loop, both using OPT1B. The program is:

```
Program d-404. Test of OPT1B with two variables.
begin
   integer COUNT1, COUNT2, EVAL, STATE1, STATE2;
   real tinlet, ginlet, Y1, Y2, delt, delg;
   array XOLD1, XOLD2, Y1OLD1, Y1OLD2, Y2OLD1, Y2OLD2[1:4];
   copy POL <
   copy OPT1B <
   procedure PRINT;
   begin
      writecr;
      write(‹dddddd›, EVAL, tinlet, ginlet);
      writetext(‹<  ›);
      write(‹-ddd.dd›, Y1, Y2);
   end PRINT;
   select(8);
   writetext(‹<
Output d-404

  eval  tinlet  ginlet  PROD   EXCESS
›);
   EVAL := 0;
   COUNT1 := 0;
   tinlet := 400;
   delt := 10;
   ginlet := 70;
A1:COUNT2 := 0;
   delg := 2;
A2:EVAL := EVAL + 1;
   Y1 := POL(1, tinlet, ginlet);
   Y2 := 2.1 - POL(2, tinlet, ginlet);
   PRINT;
   STATE2 := OPT1B(COUNT2, ginlet, Y1, Y2, 66, 80, delg,
   XOLD2, Y1OLD2, Y2OLD2, ginlet, Y1, Y2);
   if STATE2 = 0 then go to A2;
   STATE1 := OPT1B(COUNT1, tinlet, Y1, Y2, 380, 460, delt,
   XOLD1, Y1OLD1, Y2OLD1, tinlet, Y1, Y2);
```

```
   if STATE1 = 0 then
   begin
      ginlet := ginlet - 2;
      go to A1;
   end STATE1;
   writecr;
   PRINT;
end;
```

The output was:

Output d-404

eval	tinlet	ginlet	PROD	EXCESS
1	400	70	69.89	0.16
2	400	72	71.01	0.18
3	400	74	71.83	0.26
4	400	76	72.31	0.36
5	400	78	72.54	0.40
6	400	80	72.42	0.43
7	410	76	72.71	0.28
8	410	78	72.54	0.32
9	410	74	72.52	0.23
10	420	74	72.80	0.13
11	420	76	72.63	0.19
12	420	72	72.64	0.05
13	430	72	72.84	-0.05
14	430	74	72.70	0.02
14	420	74	72.80	0.13

This use of OPT1B on a problem in two variables is not much different from the direct search method described in the next section, but it is not so elegant, because OPT1B is written for a single variable only, and has no automatic step size reduction.

5.5. Direct Search - Pattern Search.

The method described in this section is normally designated as: DIRECT SEARCH or PATTERN SEARCH. It was first published by Hooke and Jeeves (1961). See also the book by Wilde (1964), pag. 145-150. The present version is essentially based on algorithms published in Communications of the Association for Computing Machinery.

The direct search method has the basic principle that only very elementary operations are carried out on the functions and variables. This excludes the use of time- and space consuming matrix inversions, etc. Each variable is varied one at a time by a move upwards or downwards and the response in the objective function is observed. It is only checked whether the function becomes better or worse, no further numerical information is collected.

The term pattern search is a slight refinement of the direct search technique. When all variables have been moved up or down, the procedure stores a list of these moves (a pattern). It will then make a single move in which all variables are moved according to the pattern, but with scaled-up step lengths. The direct search is then repeated from the new point.

The pattern search is initiated whenever a direct search move of at least one of the variables has improved the objective function. If no improvement is obtained, all step lengths are reduced by a certain factor, and the direct search is started again with due generation of a new pattern. The calculation is finished when a lower limit of the step size has been reached. A maximum number of function evalulations is also included.

The direct search is normally applied to quadratic models generated from the original functions as described in section 3. The use of penalties for handling of side restrictions is described in section 5.5.6.

Kaupe (1963) has published an algorithm in CACM for direct search. A number of remarks to this algorithm has been published later in CACM by M. Bell and M. P. Pike, R. de Vogelaere, F. K. Tomlin and L.B. Smith. The procedure given in the present book corresponds approximately to the latest version from CACM with a few corrections made by E. Balslev, who has also written a FORTRAN version of it in the Haldor Topsøe GIPS System.

Unfortunately, the names of the variables selected by Hooke and Jeeves and used by the algorithm authors are not very descriptive. We have, therefore, changed the names. A list of the original names and the present names is given here:

Original name	Present name	Significance
K	VAR	Number of independent variables.
psi	XACT	Actual set of independent variables.
DELTA	DELSTART	Start value of increment in all variables.
rho	REDFAC	Reduction factor applied on DELSTART.
S	FUNC	Real procedure calculating the objective function.
Spsi	FACT	Function value for XACT.
phi	XNEW	New set of X-values.
Sphi	FNEW	Function value for XNEW.
SS	FMIN	Best function value.
theta	work	Intermediate variable.
s	DELACT	Pattern, i.e. actual set of increments.

A further description of the parameters is given in the following sections.

5.5.1. Direct Search. The basic principle of direct search can be explained in a few words.

Let us assume that we have reached a point where the best set of X-values is available as the array:

```
XNEW[1:VAR]
```

We also assume that we have a set of increments:

```
DELACT[1:VAR]
```

to be applied to the variables. At the start of the procedure all elements in DELACT have been set equal to DELSTART. At a later stage they may have been reduced by the factor, REDFAC, possibly several times. It is also possible that the sign may have been changed of one or more of the elements. Of course, none of the elements must be zero.

A for-statement is then carried out:

```
for i := 1 step 1 until VAR do
```

For each value of i we first add the increment to the variable:

```
XNEW[i] := XNEW[i] + DELACT[i];
```

The object function is then evaluated, and if its value has been improved, we let XNEW[i] keep its changed value. If the object function is worse, we do the following:

```
DELACT[i] := -DELACT[i];
XNEW[i] := XNEW[i] + 2×DELACT[i];
```

i.e. the sign of DELACT[i] is changed, and we add twice the new value to XNEW[i]. This is the same as moving XNEW[i] in the opposite direction of the first move. The object function is then evaluated again. If it is improved, we leave XNEW[i] and DELACT[i] at their new values, if not, we subtract DELACT[i] from XNEW[i] again, i.e. XNEW[i] has now a value corresponding to the original value.

The result of the test can be summarized as follows:

1. Original DELACT[i] > 0:

	XNEW[i]	DELACT[i]	Function
A:	Increased	Unchanged	Improved
B:	Decreased	Negative	Improved
C:	Unchanged	Negative	Worse

2. Original DELACT[i] < 0:

	XNEW[i]	DELACT[i]	Function
A:	Decreased	Unchanged	Improved
B:	Increased	Positive	Improved
C:	Unchanged	Positive	Worse

The original version of the algorithm is designed to find the minimum value of a function. In the present version we use an additional formal parameter, signfactor, for multiplication of the function value whenever the function value has been calculated. By setting signfactor equal to -1, the procedure can be used to find the maximum value of a function. Internally the search is for a minimum.

The test for function improvement shown in the scheme above is made by comparing the new function value, FNEW, with the previous best value, FMIN. Whenever FNEW becomes less than FMIN, we put FMIN equal to the inproved value:

```
if FNEW < FMIN then FMIN := FNEW else .....
```

When all variables have been treated according to the scheme above, we end up with two possibilities:

1. FMIN has decreased and one or more elements of XNEW are changed.
2. FMIN and all elements of XNEW are unchanged.

Case 1 is continued with a pattern search, whereas case 2 indicates a new direct search with reduced step lengths.

5.5.2. Pattern Search. The pattern search is always made immediately after a direct search. Before the direct search we have stored the best set of the independent variables in the array (the base point):

```
XACT[1:VAR]
```

and the corresponding function value in FACT. Neither XACT nor FACT are changed during the direct search, which operates on XNEW[1:VAR] and FMIN, respectively, as explained above. If the direct search yields:

```
FMIN < FACT
```

the pattern search is started. We have again a for-statement:

```
for i := 1 step 1 until VAR do
```

For each value of i the following operations are made:

1. Check sign of DELACT[i]. From the scheme on page 134 we can see, that the sign of DELACT[i] may not always correspond to the actual difference between XNEW[i] and XACT[i]. This is corrected by the statement:

```
if XNEW[i] > XACT[i] = DELACT[i] < 0 then DELACT[i] := -DELACT[i];
```

which will make the sign of DELACT[i] equal to the sign of XNEW[i] - XACT[i].

2. Store the old X-value, XACT[i], in a work cell:

```
work := XACT[i];
```

3. Move the new X-value from the direct search, XNEW[i], to XACT[i]:

```
XACT[i] := XNEW[i];
```

We remember, that the result of the direct search was that the set XNEW was definitely better than the set XACT, so that we do not really need XACT any more.

4. Finally, the pattern move is made after the formula:

```
XNEW[i] := 2×XNEW[i] - work;
```

As the work cell contains the original value of XACT[i], before this was replaced by XNEW[i], we can also write the pattern move as:

```
XNEW[i] := XNEW[i] + (XNEW[i] - XACT[i]);
```

i.e. XNEW[i] is increased by an amount which is equal to the difference between XNEW[i] and XACT[i]. This is the kernel of the pattern search, a single move is made with the same increments as just found in the direct search.

We must then find out whether the pattern move was a success or not. The previously best function value is stored in FACT, and the function is evaluated and stored in FMIN and FNEW. We could now compare FACT and FMIN immediately, but it is more realistic to add a direct search around the new point found by the pattern search. If this additional direct

search is successful, it will make FMIN still smaller. If FMIN $\geq$ FACT, the combined effect of pattern move and direct search was a complete failure, and we can give up this pattern and start again with the simple direct search.

If FMIN < FACT, the pattern move and direct search was a success. We must then find out if it might pay to make a new pattern move. The move is made as long as we have for at least one of the variables:

abs(XNEW[i] - XACT[i]) > 0.5×abs(DELACT[i])

The condition is given this form to avoid rounding errors. It says, that at least one of the variables has been changed with success.

If the condition is not fulfilled for any of the variables, we continue with the step reduction.

5.5.3. Step Reduction. We arrive at this point in the method, when no improvement can be obtained by direct search or pattern search. All increments are reduced by multiplication by a factor, REDFAC, less than unity. A new direct search is then started. The step reduction is stopped, when the original step size, DELSTART, has been reduced below a given tolerance, eps. When this point is reached, the optimization is considered finished. A further check on the calculation is made by having a maximum number of function evaluations. If this number is exceeded the calculation is stopped, and an alarm boolean is set.

A simplified flow sheet of the optimization method is shown in figure 4 on the next page.

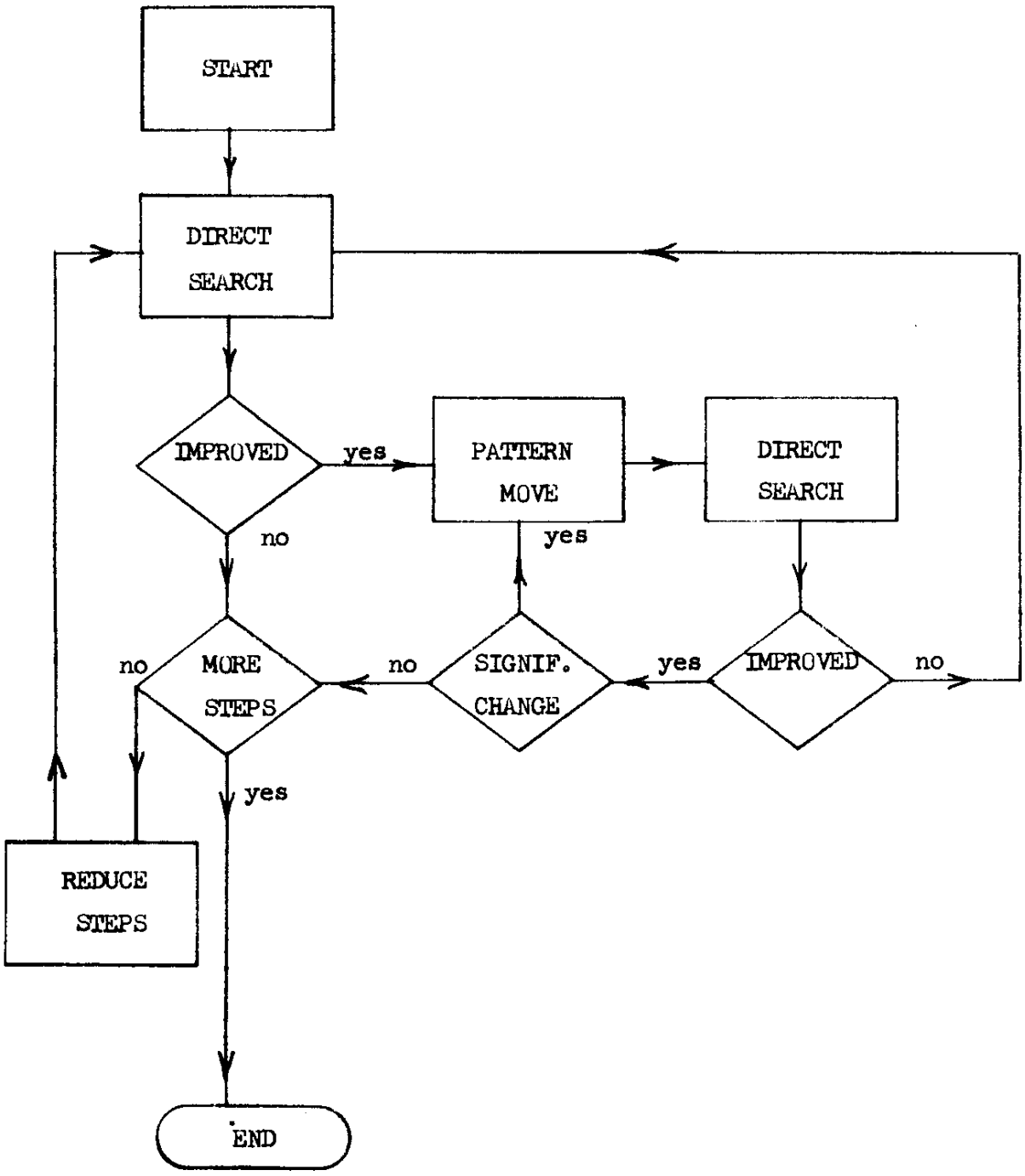


Figure 4

Direct Search and Pattern Search

5.5.4. The Procedure DIRSEARCH. This procedure operates after the principles described above. The declaration is shown on page 140 - 141. The following formal parameters are used:

integer VAR: The number of independent variables.

array XACT[1:VAR]: The set of independent variables. Must contain the start values at the entry to the procedure and will contain the optimal values at the exit.

real DELSTART: Must contain the initial value of the increment for the independent variables. The same value is used for all variables. Contains the last used value at the exit.

real eps: Minimum permissible step length. DELSTART is scaled down as long as it is not less than eps.

real REDFAC: DELSTART is multiplied by this reduction factor in each step length reduction.

real procedure FUNC: An external procedure which must calculate the value of the objective function for a set of the independent variables given as a parameter: FUNC(XACT) or FUNC(XNEW).

integer signfactor: Must be specified as -1 for determination of a maximum and +1 for a minimum.

real FACT: Contains the optimum function value (multiplied by signfactor) at the exit.

integer maxev: The maximum permissible number of function evaluations with the FUNC-procedure.

boolean conv: The procedure normally sets this to true at the exit, except when maxev has been exceeded, when it is set to false.

It may be practical to replace the formal parameter, eps, by an integer: MAXRED indicating how many times the increment DELSTART may be reduced by multiplication by REDFAC. It may also be convenient to have the actual number of functions as a parameter.

The declaration of DIRSEARCH is:

```
procedure DIRSEARCH(VAR, XACT, DELSTART, eps, REDFAC, FUNC,
signfactor, FACT, maxev, conv);
value VAR, signfactor;
boolean conv;
integer VAR, signfactor, maxev;
real DELSTART, eps, REDFAC, FACT;
real procedure FUNC;
array XACT;
begin
   integer i, eval;
   real FNEW, FMIN, work;
   array XNEW, DELACT[1:VAR];
   procedure E;
   for i := 1 step 1 until VAR do
   begin
      XNEW[i] := XNEW[i] + DELACT[i];
      test eval;
      FNEW := FUNC(XNEW)×signfactor;
      if FNEW < FMIN then FMIN := FNEW else
      begin
         DELACT[i] := -DELACT[i];
         XNEW[i] := XNEW[i] + 2×DELACT[i];
         test eval;
         FNEW := FUNC(XNEW)×signfactor;
         if FNEW < FMIN then FMIN := FNEW else
         XNEW[i] := XNEW[i] - DELACT[i];
      end;
   end E;
   procedure test eval;
   if eval < maxev then eval := eval + 1 else
   begin
      conv := false;
      go to EXIT;
   end test eval;
   for i := 1 step 1 until VAR do DELACT[i] := DELSTART;
   FACT := FUNC(XACT)×signfactor;
   eval := 1;
   conv := true;
```

```
L1:writetext(‡<
DIRECT:‡);
   FMIN := FACT;
   for i := 1 step 1 until VAR do XNEW[i] := XACT[i];
   E;
   if FMIN < FACT then
   begin
L2:   writetext(‡<
PATTERN:‡);
      for i := 1 step 1 until VAR do
      begin
         if XNEW[i] > XACT[i] = DELACT[i] < 0 then
         DELACT[i] := -DELACT[i];
         work := XACT[i];
         XACT[i] := XNEW[i];
         XNEW[i] := 2×XNEW[i] - work;
      end for i;
      FACT := FMIN;
      test eval;
      FMIN := FNEW := FUNC(XNEW)×signfactor;
      E;
      if FMIN ≥ FACT then go to L1;
      for i := 1 step 1 until VAR do
      if abs(XNEW[i]-XACT[i]) > 0.5×abs(DELACT[i]) then go to L2;
   end;
L3: writetext(‡<
REDUCE:‡);
   if DELSTART ≥ eps then
   begin
      DELSTART := REDFAC×DELSTART;
      for i := 1 step 1 until VAR do DELACT[i] := REDFAC×DELACT[i];
      go to L1;
   end if;
EXIT: maxev := eval;
end DIRSEARCH;
```

The following local variables and procedures are used:

integer i: Counter in for-statements.

integer eval: Counter of function evaluations.

real FNEW: Used for actual function value generated during the direct search.

real FMIN: Used for best function value during direct search.

real work: Work cell.

array XNEW[1:VAR]: New set of X-values generated during direct and pattern search.

array DELACT[1:VAR]: The actual set of increments in the variables. Initially, the procedure puts all these equal to DELSTART. They are scaled down together with DELSTART during the calculation by multiplication by REDFAC. The signs of DELACT are also changed.

procedure E: This procedure performs the direct search as explained on page 134. It may change the items in XNEW and the sign of the elements in DELACT. It updates FMIN to contain the best function value.

procedure test eval: This procedure is called immediately before each call of FUNC. If eval < maxev it adds 1 to eval, otherwise it sets conv to false and terminates the calculation.

The calculations performed by DIRSEARCH correspond closely to the flow sheet on page 138 and the explanations on the previous pages. In order to illustrate the calculations the procedure has been fitted with printing of the three text strings:

DIRECT:
PATTERN:
REDUCE:

at the three labels where these operations are started. The direct search always following immediately after the pattern search is not indicated in this way.

5.5.4.1. Example with Quadratic Function. The following example illustrates the use of DIRSEARCH on the quadratic function given in equation (5.19), page 105:

$$Y = 10 - (0.8\times(X1-5) - 0.6\times(X2-5))\uparrow 2 - 4\times(0.6\times(X1-5) + 0.8\times(X2-5))\uparrow 2$$

The function has its maximum at (5,5) and the contours are ellipses around this point. The test program for this example has the form:

```
Program d-388. Test of DIRSEARCH.
begin
   boolean conv;
   real DELSTART, REDFAC, FACT, Y;
   array X[1:2];
   copy DIRSEARCH<
   real procedure FUNC(X);
   array X;
   begin
      FUNC := Y := 10 - (0.8×(X[1]-5) - 0.6×(X[2]-5))↑2
      -4×(0.6×(X[1]-5) + 0.8×(X[2]-5))↑2;
      writecr;
      write(‹-dddd.dddddd›, X[1], X[2], Y);
   end FUNC;
   select(8);
   writechar(72);
   writetext(‹<
Output d-388.
    X[1]        X[2]        Y
›);
   X[1] := 1;
   X[2] := 2;
   REDFAC := 0.5;
   DELSTART := 0.5;
   DIRSEARCH(2, X, DELSTART, 1₁₀-2, REDFAC, FUNC,
   -1, FACT, 400, conv);
   writecr;
   writetext(if conv then ‹<conv› else ‹<not conv›);
   writecr;
end;
```

The output from the program is shown on the following pages:

Output d-388.

X[1]	X[2]	Y
1.000000	2.000000	-84.120000
DIRECT:		
1.500000	2.000000	-72.000000
1.500000	2.500000	-58.930000
PATTERN:		
2.000000	3.000000	-37.680000
2.500000	3.000000	-29.080000
2.500000	3.500000	-20.370000
PATTERN:		
3.500000	4.500000	2.430000
4.000000	4.500000	5.750000
4.000000	5.000000	7.920000
PATTERN:		
5.500000	6.500000	0.750000
6.000000	6.500000	-2.970000
5.000000	6.500000	3.430000
5.000000	7.000000	-1.680000
5.000000	6.000000	7.080000
DIRECT:		
3.500000	5.000000	5.320000
4.500000	5.000000	9.480000
4.500000	4.500000	8.030000
4.500000	5.500000	9.470000
PATTERN:		
5.000000	5.000000	10.000000
5.500000	5.000000	9.480000
4.500000	5.000000	9.480000
5.000000	4.500000	9.270000
5.000000	5.500000	9.270000
PATTERN:		
5.500000	5.000000	9.480000
6.000000	5.000000	7.920000
5.000000	5.000000	10.000000
5.000000	4.500000	9.270000
5.000000	5.500000	9.270000

```
DIRECT:
    4.500000    5.000000    9.480000
    5.500000    5.000000    9.480000
    5.000000    5.500000    9.270000
    5.000000    4.500000    9.270000
REDUCE:
DIRECT:
    5.250000    5.000000    9.870000
    4.750000    5.000000    9.870000
    5.000000    4.750000    9.817500
    5.000000    5.250000    9.817500
REDUCE:
DIRECT:
    4.875000    5.000000    9.967500
    5.125000    5.000000    9.967500
    5.000000    5.125000    9.954375
    5.000000    4.875000    9.954375
REDUCE:
DIRECT:
    5.062500    5.000000    9.991875
    4.937500    5.000000    9.991875
    5.000000    4.937500    9.988594
    5.000000    5.062500    9.988594
REDUCE:
DIRECT:
    4.968750    5.000000    9.997969
    5.031250    5.000000    9.997969
    5.000000    5.031250    9.997148
    5.000000    4.968750    9.997148
REDUCE:
DIRECT:
    5.015625    5.000000    9.999492
    4.984375    5.000000    9.999492
    5.000000    4.984375    9.999287
    5.000000    5.015625    9.999287
```

```
REDUCE:
DIRECT:
    4.992187    5.000000    9.999873
    5.007812    5.000000    9.999873
    5.000000    5.007812    9.999822
    5.000000    4.992187    9.999822
REDUCE:
conv
```

The search is started in the base point: (1,2) with increments of the size 0.5. The first direct search takes us to the point (1.5,2.5) with an increase in both variables. The pattern is now (0.5,0.5) and the first pattern move takes us to (2,3). Additional direct search leads to the point (2.5,3.5).

The second pattern move gives (3.5,4.5), which is improved by the direct search to (4,5). Note, how a series of successful pattern moves immediately one after another gives a doubling of the steps in each move (although DELACT is unchanged).

The third pattern move to (5.5,6.5) is a failure, and the additional direct search which takes us to the point (5,6) with the function value 7.08 is not better than the old base (4,5) with the value 7.92.

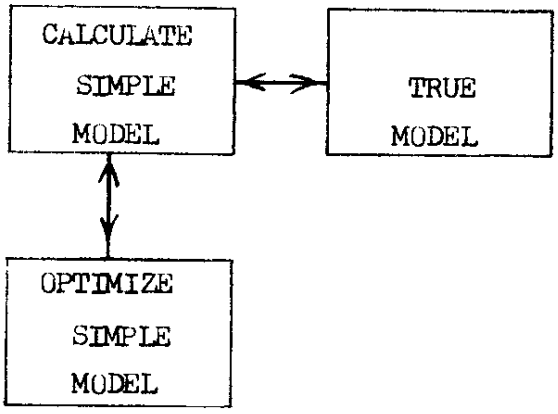
We then start a fresh direct search taking us from (4,5) to the point (4.5,5.5) with the function value 9.47. From there, a pattern move takes us to the optimum point (5,5) with the value 10. A second pattern move is attempted but without success.

The rest of the calculation consists of direct search and step size reduction. Pattern moves are not tried, because no improvement is obtained from the direct search. The optimum is obtained with the required accuracy.

The practical experience with the use of the direct search - pattern search has indicated that the actual benefit in use of the pattern feature is not very pronounced. It could be an advantage to eliminate this feature from the procedure.

5.5.5. Optimization with Model Generation. When the direct search optimization is to be included in a routine program for optimization of any kind of data generated by a computer program, various features must be included in addition to the direct search itself.

As the basic strategy in the direct search method is the use of a large number of function evaluations and a minimum amount of algebraic treatment of the data, it will nearly always be necessary to operate on a simplified model which can be evaluated very quickly. The program must then have a flow sheet of the form:



The generation of the simple model (e.g. a quadratic model) from the true model requires certain criteria for how often and under which circumstances the simple model must be recalculated from the true model. The control parameters used in this strategy are explained in the following pages.

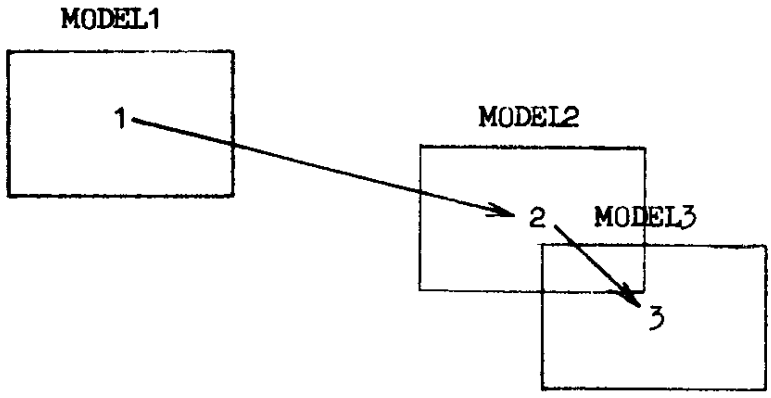
In view of the fast operation of the direct search method when working on a simple model, it is possible to solve the problem of simultaneous side conditions (equalities or inequalities) in a fairly efficient way by means of penalties. If we are looking for the maximum value of the object function with side conditions, we perform the optimization not on the object function itself but on a modified function in which a certain penalty is subtracted from the object function. The penalty is zero when the side condition is satisfied and positive when it is not. The penalty is made to increase quadratically with the distance from the target value. More details are given in section 5.5.6. below.

The generation of a simple (quadratic) model from a true, complex model involves two problems:

1. Generation of a single model.
2. Strategy for repeated model generation.

In order to calculate a single model, we must have a set of start values of the independent variables (a base point) and a set of increments. The mathematics involved in the model generation is fully explained in section 3. The actual way in which the control of the calculation is transferred from the part of the program performing the optimization to the part of the program evaluating the true model depends on the program administration system used. In the Haldor Topsøe GIPS System, the generation of a model requires access to a job list performing the calculation of the true model.

The necessity of repeating the model calculation comes from the fact that a simple quadratic model will not be identical to the true model, except in the case when this is also quadratic. This is illustrated by the figure:



We start by generating the quadratic MODEL1 from the true model in the range around the point 1. The optimum point of MODEL1 is calculated to be, say point 2. As this point is far away from point 1, it is necessary to generate a new quadratic model, MODEL2, around point 2. When this model is optimized, we may arrive at point 3, etc.

It is evident that it is necessary to generate a new model when the predicted optimum is far outside the range covered by the previous model

generation. But it is not sufficient that the points 1, 2, 3, etc. converge to a single point, it is also necessary to reduce the increments used for the variables in the successive model generations, because of the error introduced when the true model is approximated by a quadratic model. This error is reduced when the increments are reduced. On the other hand, the increments must not be reduced too much as this may give rise to poor numerical accuracy.

The following parameters are adequate for the control of the model generation:

real MXSTP: The maximum step factor for a new model. This is a general safety against having a new model generated at a point very far from the center of the old model. Example:

Base point of old model: X1 = 7, X2 = 17.
Original increments: DX1 = 2, DX2 = 3.
MXSTP: 4.

MXSTP is measured relative to the original increments. If the center of the new model is found to: X1NEW = 13, X2NEW = 29, the relative distance between the two models becomes:

$$DIST = sqrt(((X1NEW-X1)/DX1)\uparrow 2 + ((X2NEW-X2)/DX2)\uparrow 2)$$
$$= sqrt(((13-7)/2)\uparrow 2 + ((29-17)/3)\uparrow 2) = 5$$

As DIST = 5 is higher than the permitted value MXSTP = 4, the new center must be moved closer to the old, so that DIST = 4.

integer MAXGEN: The maximum permissible number of model generations. If the number of variables is high and the calculation time on the true model is long, MAXGEN should not be selected too high.

real RANGE: Model range. This is the radius of the region around the model center inside which the model is assumed to be reliable. The RANGE is in units of the actual model size, i.e. measured relatively to the last used increments for the model generation. The RANGE criterion is only used to determine whether the increments should be reduced in a new model generation. If the new model center is inside RANGE, the model increments will be reduced. RANGE = 1.5 is normally a reasonable value.

real ACC: Required accuracy of the optimization. ACC is measured in units of the original increments used in the model generation. The optimization is considered successfully finished when the latest model

differs less than ACC from the previous center. ACC = 0.1 is a typical value. The conditions:

$$ACC < RANGE < MXSTP$$

must, of course, be adhered to.

real REDGEN: Factor for reduction of increments in model generation. When the increments are to be reduced, all increments are multiplied by this factor. It must lie in the range: $0 < REDGEN < 1$.

5.5.6. Use of Penalties for Side Restrictions. The methods and conventions for handling side conditions in the GIPS program, OPTI, developed by E. Balslev are described in the following. When side conditions are to be included in an optimization, we must have more than one function to operate upon. The total number of functions considered in the calculation is denoted: FUNTOT. For each of these functions we must indicate its type by means of a code:

YCODE[1:FUNTOT]

YCODE is an integer type indicator:

YCODE = 0: Objective function to be optimized.
YCODE = 1: Equality constraint.
YCODE = 2: Inequality constraint.
YCODE = 3: Other function not included in optimization or constraint.

Only one of the functions can be the objective function. In some cases it can be convenient to operate with an objective function which is a linear function of all the available functions:

$$YOBJ = CMIX[1]\times Y[1] + CMIX[2]\times Y[2] + \ldots + CMIX[FUNTOT]\times Y[FUNTOT]$$

Some of the mixing coefficients, CMIX[1:FUNTOT], may be zero. If this feature is used, YCODE should not be zero for any of the functions, otherwise YCODE must be zero for just one of the functions. In the latter case, the program will set CMIX to 1 for this function and to 0 for the other functions.

When side conditions are used, we must define the required target values, YTARG[1:FUNTOT], of the conditions. For equalities, YTARG is the required value of the function. For inequalities, we assume that they are of the form:

$$Y[J] \geq YTARG[J]$$

i.e. YTARG defines the lower limit of the function. For YCODE = 0 and 3 YTARG should be given as zero.

In order to calculate the penalties which are to be subtracted from the objective function when we are searching a maximum (and added for a minimum), we must introduce a set of penalty coefficients:

CPEN[1:FUNTOT]

The total penalty is then calculated as:

```
PENALTY = CPEN[1]×(Y[1]-YTARG[1])↑2×signfactor
        +CPEN[2]×(Y[2]-YTARG[2])↑2×signfactor
        +.....
        +CPEN[FUNTOT]×(Y[FUNTOT]-YTARG[FUNTOT])↑2×signfactor
```

The value of signfactor is -1 for maximum and 1 for minimum. For the variable having YCODE = 0, the corresponding term is omitted (or CPEN set to zero from the beginning). For YCODE = 2 (inequalities) the term is only included, if Y < YTARG. The term is also omitted for YCODE = 3.

It now remains to calculate good values for the penalty coefficients CPEN, for YCODE = 1 and 2. The basic strategy is to find an acceptable start value of CPEN and to repeat the optimization several times with increasing values of CPEN, until the target values are satisfied within a certain error. This iteration is done for each model. The total flow sheet for the model generation, optimization, and side conditions then takes the form as shown in figure 5 on the next page.

A reasonable start value of the penalty coefficients, CPEN, can be obtained as follows. From the model generation we have accumulated information on the variation of the functions calculated. This information is stored together with the model as the array:

YRANGE[1:FUNTOT,1:2]

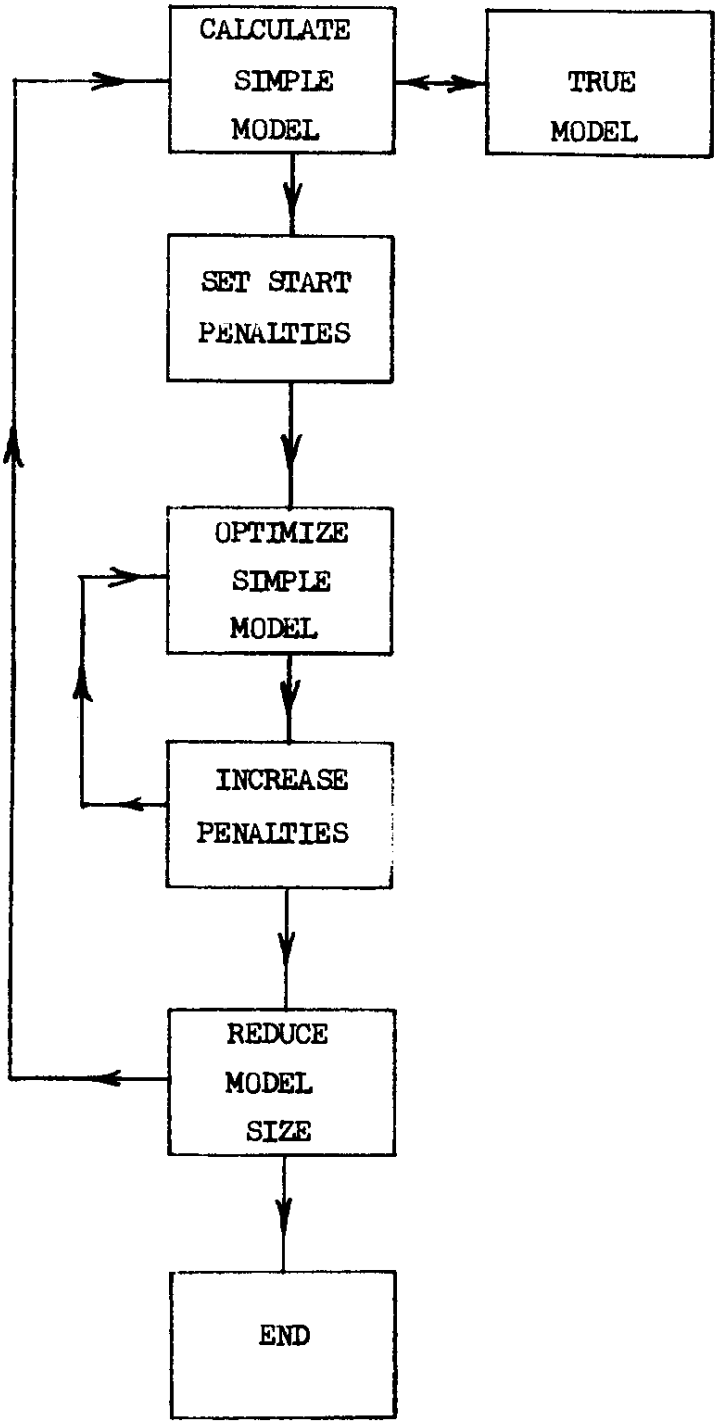


Figure 5

Optimization Flow Sheet

Here, column 1 is the lower limit, YLOW[1:FUNTOT], and column 2 the upper limit, YHIGH[1:FUNTOT]. In other words, the values of function no. J were in the range:

$$YLOW[J] \leq Y[J] \leq YHIGH[J]$$

during the model generation. As the penalty is something to be subtracted from the object function, it may be a reasonable start guess to let the start value of the penalty be a certain fraction, say 10 per cent, of the variation of Y in the model range:

$$PENALTY = 0.1 \times (YHIGH[OBJ] - YLOW[OBJ])$$

OBJ is the subscript of the object function. If a mixed object function is used, we must insert the weighted differences using CMIX:

$$PENALTY = 0.1 \times (CMIX[1] \times (YHIGH[1] - YLOW[1]) + CMIX[2] \times (YHIGH[2] - YLOW[2]) + \ldots \text{ etc.})$$

After the equation on page 151, the penalty arising from function no. I has the form:

$$PENALTY[I] = CPEN[I] \times (Y[I] - YTARG[I])\uparrow 2$$

This gives the approximate formula:

$$CPEN[I] \times (Y[I]-YTARG[I])\uparrow 2 = 0.1 \times (YHIGH[OBJ] - YLOW[OBJ])$$

or:

$$CPEN[I] = 0.1 \times (YHIGH[OBJ]-YLOW[OBJ])/(Y[I]-YTARG[I])\uparrow 2$$

It then remains to find a good value of Y[I] to insert in this formula. At the time when the penalty coefficients are calculated, we have only access to YLOW[I] and YHIGH[I] from the model generation. It is important to avoid that the denominator becomes zero or nearly zero. One way of avoiding this is to insert YHIGH[I]-YLOW[I] instead of Y[I] - YTARG[I], but this may not be satisfactory, if YTARG[I] is far outside the range from YLOW[I] to YHIGH[I]. The figure 6 on the next page shows how the denominator is calculated at present in the program OPTI.

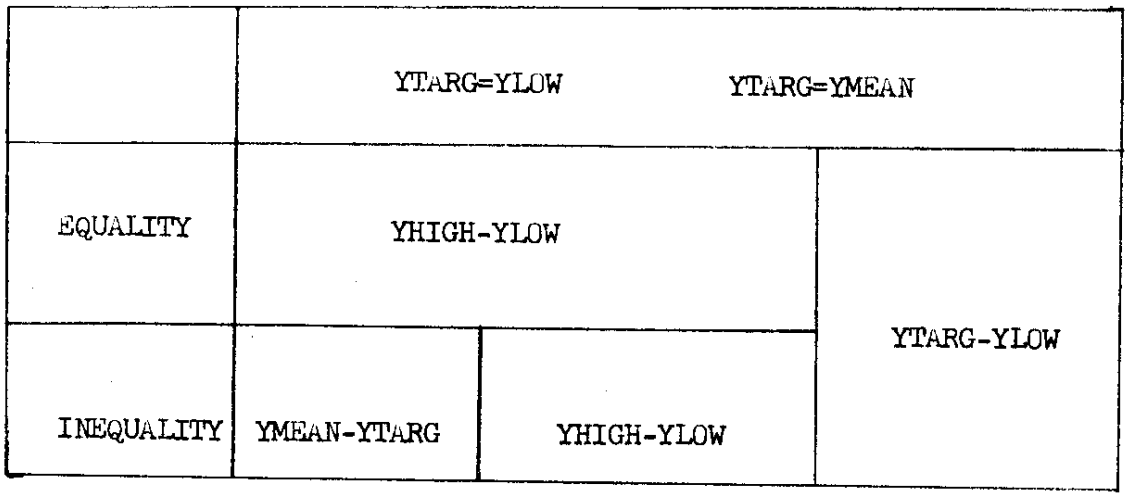


Figure 6

Calculation of DEN in $0.1\times(YHIGH[OBJ]-YLOW[OBJ])/DEN\uparrow 2$
YMEAN is the mean value of YLOW[I] and YHIGH[I].

When the start values of the penalty coefficients have been calculated in this way, we need only a few more parameters in order to control the iterations required in the flow sheet on page 152:

integer MAXOPT: A maximum number of optimization cycles in which the penalty coefficients are increased by a certain factor and the optimization repeated.

array ACCYT[1:FUNTOT]: The required accuracy in satisfying the side conditions. The optimization cycles with penalty coefficient increase are stopped as soon as Y[I] differs less than ACCYT[I] from YTARG[I] for all the side condition functions. ACCYT is only used for YCODE = 1 or 2.

Finally, we need the factor by which to increase the penalty coefficients between each new optimization. In the program OPTI this factor is built into the program as the fixed values: 3.0 for equalities and 10.0 for inequalities. It may be more practical to let these factors and the coefficient 0.1 used in the calculation of the start values of CPEN become input data to the program.

5.5.6.1. Calculation Example. The program d-387 shown on the following pages performs an optimization of a function of two variables with a simultaneous equality condition for a second function. The problem is the same as treated on page 109 with a Lagrange multiplier and on page 113 with elimination of one of the variables.

The program has the declaration:

Program d-387. Optimization with DIRSEARCH.

```
begin
   boolean conv, first;
   integer count, si, cyc, state, LINE, i, j, count2;
   real PENAL, DELSTART, DEN, x1, x2, REDFAC, FACT, R;
   array xstart, del0x, xact, yact, XACT, delx[1:2],
   MAT[1:6,1:8], MOD[1:6,1:2], YR[1:2, 1:2];
   copy DIRSEARCH<
   copy GENMOD1<
   copy LEQ1<
   copy MODVAL1<
   copy POL<
   procedure PRINT;
   begin
      writecr;
      write(<ddd>, LINE);
      write(<-dddd.dddd>, xact[1], xact[2]);
      write(<-dddd.dddd00>, yact[1], yact[2]);
   end PRINT;
   real procedure FUNC(x);
   array x;
   begin
      for si := 1, 2 do
      xact[si] := xstart[si] + x[si]xdelx[si];
      MODVAL1(2, 2, 6, 0, MOD, xact, yact);
      FUNC := R := yact[1] - PENALxyact[2]↑2;
      PRINT;
      write(<-dddd.dddd00>, R);
      if LINE mod 40 = 0 then writechar(72);
      LINE := LINE+1;
   end FUNC;
   procedure Q;
   begin
      writecr;
      for i := 1 step 1 until 6 do
      write(<dddddddddd>, 390+10xi);
      writecr;
      for j := 1 step 1 until 8 do
      begin
         writecr;
```

```
            x2 := xact[2] := 64 + 2×j;
            write(‡dd‡, x2);
            for i := 1 step 1 until 6 do
            begin
               x1 := xact[1] := 390 + 10×i;
               if first then
               begin
                  yact[1] := POL(1, x1, x2);
                  yact[2] := POL(2, x1, x2);
               end else
               MODVAL1(2, 2, 6, 0, MOD, xact, yact);
               write(‡-dddd.dddd‡, yact[1]-PENAL×yact[2]↑2);
            end for i;
         end for j;
      end Q;
      select(8);
      writechar(72);
      writetext(‡‡
Output d-387.
       x[1]      x[2]      y[1]      y[2]    y[1]-PENAL×y[2]↑2
      tinlet    ginlet     PROD      EXCESS
‡);
      xstart[1] := 430;
      xstart[2] := 70;
      del0x[1] := 10;
      del0x[2] := 2;
      REDFAC := 0.5;
      cyc := 0;
      PENAL := 0;
      first := true;
      Q;
      first := false;
A:    count := LINE := 0;
B:    state := GENMOD1(count, 2, 2, 6, xstart, del0x,
      xact, yact, 1₁₀-20, MAT, MOD);
```

```
   yact[1] := POL(1, xact[1], xact[2]);
   yact[2] := POL(2, xact[1], xact[2]) - 2;
   if count = 1 then
   for si := 1, 2 do
   YR[si, 1] := YR[si, 2] := yact[si];
   for si := 1, 2 do
   begin
      if yact[si] < YR[si, 1] then YR[si, 1] := yact[si];
      if yact[si] > YR[si, 2] then YR[si, 2] := yact[si];
   end for si;
   if state = 0 then
   begin
      PRINT;
      go to B;
   end if state = 0;
   cyc := cyc + 1;
   DEN := if abs(YR[2,2]) < abs (YR[2,1]) then
   YR[2,2]-YR[2,1] else YR[2,2];
   PENAL := 0.1×(YR[1,2]-YR[1,1])/DEN↑2;
   count2 := 1;
   if cyc < 3 then
   begin
D:    writecr;
      write(‹-dddd.dddd›, PENAL);
      Q;
      writecr;
      XACT[1] := XACT[2] := 0;
      delx[1] := del0x[1];
      delx[2] := del0x[2];
      DELSTART := 1;
      LINE := 0;
      DIRSEARCH(2, XACT, DELSTART, 0.032, REDFAC, FUNC, -1,
      FACT, 400, conv);
      writecr;
      writetext(if conv then ‹<conv› else ‹<not conv›);
      for si := 1, 2 do
      xstart[si] := xstart[si] + XACT[si]×delx[si];
      writechar(72);
      count2 := count2 + 1;
```

```
      PENAL := 3×PENAL;
      go to if count2 < 8 ∧ abs(yact[2]) > 0.02 then D else A;
   end if cyc;
   writecr;
end;
```

The program uses the procedure DIRSEARCH already explained, and the two procedures GENMOD1 and MODVAL1 for generation and evaluation of the models. The program has three local procedures:

PRINT: Prints a line with actual values of X and Y.

FUNC: The function evaluation procedure used by DIRSEARCH. Calculates the Y-values corresponding to the actual values of the two values of X. The value of the object function is calculated as:

yact[1] - PENAL×yact[2]

where PENAL is the penalty coefficient, CPEN. The target value of the second function is zero.

Q: This function prints a small table of the object function in order to show how the function varies when CPEN is increased.

The program contains the following parts:

Setting of start values.

Generation of the quadratic models. Starts at the label A. The two independent variables are the catalyst inlet temperature, x[1], and the relative gas flow, x[2], in the first catalyst bed. By means of the procedure POL we calculate the two functions:

y[1]: Ammonia production.

y[2]: Necessary height of the lower exchanger.

The problem is to vary x[1] and x[2] so that y[1] attains its maximum and y[2] becomes equal to 2. In the program we have subtracted 2 from y[2] before the model is generated, so that the target value of y[2] becomes zero.

During the model generation we also calculate the start value of the penalty coefficient.

Optimization cycle. This is a number of calls of DIRSEARCH with increasing values of the penalty coefficient. The cycle is stopped when

y[2] becomes less than 0.02 or the cycle counter becomes higher than 7.

After the first optimization cycle, a new model is generated and optimized again.

The complete output from the program is too big to be reproduced here. A summary is given in the following table:

Optimization cycle	Penalty coefficient	Number of calls	x[1]	x[2]	y[1]	y[2]	Object function
1	0.305	99	391.88	79.19	73.08	0.52	73.00
	0.915	72	402.50	77.19	72.97	0.32	72.93
	2.74	64	409.06	75.94	72.92	0.09	72.90
	8.23	59	412.50	75.19	72.89	0.05	72.87
	24.7	71	414.69	74.56	72.87	0.03	72.84
	74.1	55	414.06	74.31	72.85	0.02	72.80
	222	63	413.13	74.06	72.80	0.02	72.75
2	2.65	76	424.69	73.13	72.84	0.06	72.83
	7.95	63	423.13	73.50	72.83	0.02	72.82
	23.8	41	422.50	72.63	72.82	0.01	72.82

Result found by use of OPTQUA1 with elimination (page 116):

422.17 73.69 72.81 0.00

This example illustrates the basic principle of direct search optimization with use of penalties for the side conditions. The required number of function evaluations is much higher than in the OPTQUA1 method but this should normally be acceptable when the quadratic models are evaluated directly in the core.

The effect of the penalty coefficient on the model optimization can be illustrated by copying some of the tables printed by the Q-procedure in the program d-387. Two tables of the object function are shown here, one for CPEN = 0.305 and the other for CPEN = 222, i.e. for the lowest and the highest value used:

CPEN = 0.305

x1:	400	410	420	430	440	450
x2:						
66	66.44	68.28	69.79	70.97	71.80	72.27
68	68.49	69.99	71.16	72.01	72.52	72.68
70	70.12	71.27	72.10	72.61	72.80	72.65
72	71.36	72.17	72.66	72.84	72.69	72.21
74	72.26	72.72	72.87	72.70	72.22	71.39
76	72.80	72.92	72.72	72.20	71.37	70.19
78	72.97	72.75	72.21	71.34	70.14	68.59
80	72.74	72.18	71.28	70.05	68.48	66.54

CPEN = 222

x1:	400	410	420	430	440	450
x2:						
66	-63.59	-52.32	-59.38	-86.62	-139.77	-228.47
68	29.91	35.94	31.94	15.84	-18.32	-80.37
70	63.14	65.94	64.37	56.15	35.13	-8.76
72	70.88	72.00	71.81	67.81	53.63	18.98
74	72.12	72.70	72.40	68.51	54.43	19.69
76	70.03	70.78	68.45	60.12	38.99	-5.63
78	52.00	53.18	46.47	28.75	-7.02	-71.73
80	-10.41	-8.96	-22.82	-55.35	-113.77	-209.22

The last table clearly shows how the original function is completely distorted so that it has normal values only in a narrow range where the second function is close to zero.

5.5.7. Survey of Control Parameters. A summary is given here of the control parameters used in the three parts of the complete optimization:

Model Generation:

MXSTEP	Maximum step factor for new model.
MAXGEN	Maximum number of models.
RANGE	Model range. If the new model is inside the range, the increments are reduced.
ACC	Required accuracy of the optimization.
REDGEN	Factor for reduction of increments.

Penalty Handling:

MAXOPT	Maximum number of penalty coefficient increases.
ACCYT	Required accuracy in satisfying the side conditions.

Direct Search Optimization:

eps	Minimum permissible step length.
REDFAC	Factor for reduction of increments in the optimization.
maxev	Maximum number of function evaluations.

6. REFERENCES

Frøberg, C.-E.: Introduction to Numerical Analysis, Addison-Wesley Publ. Co. (1966).

Hodgman, C. D. et al.: Handbook of Chemistry and Physics, Edition 38, Chemical Rubber Publ. Co. (1956).

Hooke, R. and Jeeves, T.A.: J. Ass. Comp. Mach., 8, 212-229 (1961).

Kallin, S.: Lærobok i FORTRAN, Studentlitteratur (1969).

Kaupe, A. F., Jr.: Algorithm no. 178, CACM, 6, 313 (1963).

Korn, G. A. and Korn, T. M.: Mathematical Handbook for Scientists and Engineers, McGraw-Hill Book Co., pag. 316 and 372 (1961).

Lapidus, L.: Digital Computation for Chemical Engineers, McGraw-Hill Book Co. (1962).

Naur, P.: Automatic grading of students ALGOL programming, BIT, 4, 177 - 188 (1964).

Schreiner Andersen, N.: RCSL No. 53-M1, ALGOL 5 procedure zero1, Regnecentralen (1970).

Villadsen, J.: Selected Approximation Methods for Chemical Engineering Problems (1970).

Wilde, D. J.: Optimum Seeking Methods, Prentice Hall, pag. 145 (1964).

Zachariassen, J.: GIER System Library Order No. 162, Regnecentralen (1963).

7. ALPHABETIC INDEX