

Algorithms

J. G. HERRIOT, Editor

ALGORITHM 314

FINDING A SOLUTION OF N FUNCTIONAL EQUATIONS IN N UNKNOWNNS [C5]

D. B. DULLEY AND M. L. V. PITTEWAY (Recd. 7 Apr. 1966, 19 Oct. 1966 and 5 July 1967)

Cripps Computing Centre, University of Nottingham, England

```
procedure ndinv (functions, initstep, error, cycles, x, f, accest, n);  
value n; procedure functions; real initstep, error;  
integer cycles, n; array x, f, accest;
```

comment This procedure performs inverse interpolation in n dimensions, i.e., it will find a set of values for n variables x , such that n functions $f(x)$ are zero. A more sophisticated technique, suitable for large values of n , has been developed by S. M. Robinson (Interpolative Solution of Systems of Nonlinear Equations, *SIAM Journal of Numerical Analysis*, 3 (1966), 650-658). It can also be used to fit a curve with n arbitrary parameters to a set of points, the n functions being formed, in this case, by equating to zero the differential of the sum of the squares of the residues with respect to each parameter in turn.

The functions required are specified by a procedure of the form *functions (f, x)* where f and x are declared as arrays from 1 to n . This procedure should calculate the n functions from a set of values given in x , placing the results in f . The first step is made by forming partial derivatives over an interval *initstep*. 10^{-6} should be suitable for values of x of the order 1 to 10. Exit from the procedure will occur if:

- (i) the root sum square of the x increments is less than *error*. If *error* is negative, this condition must be satisfied for $|error|$, and in addition this process is continued until the root sum square of the increments fails to decrease
- or (ii) the number of iterations is greater than *cycles*, implying that too much accuracy has been requested
- or (iii) the specified equations are singular. In this case exit is by a jump to a label *fails*.

On entry, the array x should contain the starting values. On exit, the array x will contain the accurate root, f the residues and *accest* the last increments made to x as a measure of the accuracy.

This procedure calls on a global procedure *eqnsolve (A, b, n, label)*, which solves n linear simultaneous equations in n unknowns $Ax = b$, placing the result in b . If A is singular, it is assumed that an exit is made by a jump to *label*;

begin

```
real work, sumsqres, prevres;  
integer i, j, count;  
Boolean switch;  
array prevf[1:n], copydelf[1:n, 1:n], delx, delf[1:n, 1:n+1];  
functions (prevf, x);  
for i := 1 step 1 until n do  
begin  
x[i] := x[i] + initstep;  
functions (f, x);  
for j := 1 step 1 until n do  
begin  
delf[i, j] := f[j] - prevf[j];
```

```
delx[i, j] := 0;  
end differencing initial point;  
delx[i, i] := initstep;  
x[i] := x[i] - initstep;  
end setting up the initial matrix of points;  
sumsqres := 1.0E30;  
count := 0;  
iterate:  
switch := true;  
prevres := sumsqres;  
tryagain:  
for i := 1 step 1 until n do  
begin  
f[i] := prevf[i];  
for j := 1 step 1 until n do copydelf[i, j] := delf[i, j]  
end copying delf for destructive use in procedure eqnsolve;  
eqnsolve (copydelf, f, n, inline);  
sumsqres := 0;  
for i := 1 step 1 until n do  
begin  
work := 0;  
for j := 1 step 1 until n do work := work - delx[i, j] × f[j];  
accest[i] := work;  
x[i] := x[i] + work;  
sumsqres := sumsqres + work × work  
end calculation of next point;  
count := count + 1;  
functions (f, x);  
if count > cycles  $\vee$  sumsqres < error  $\times$  error  $\wedge$   
(error > 0  $\vee$  sumsqres > prevres) then go to exit;  
for i := 1 step 1 until n do  
begin  
work := f[i] - prevf[i];  
prevf[i] := f[i];  
for j := n step - 1 until 1 do  
begin  
delx[i, j+1] := delx[i, j] - accest[i];  
delf[i, j+1] := delf[i, j] - work  
end calculation of new differences;  
delx[i, 1] := -accest[i];  
delf[i, 1] := -work  
end moving points up one place in tables;  
go to iterate;  
inline:  
for i := 1 step 1 until n do  
begin  
delx[i, n] := delx[i, n+1];  
delf[i, n] := delf[i, n+1]  
end discarding alternative point;  
switch := ¬ switch;  
if switch then go to fails else go to tryagain;  
exit:  
end ndinv
```

ALGORITHM 315

THE DAMPED TAYLOR'S SERIES METHOD FOR MINIMIZING A SUM OF SQUARES AND FOR SOLVING SYSTEMS OF NONLINEAR EQUATIONS [E4, C5]

H. SPÄTH (Recd. 25 Oct. 1966 and 19 June 1967)

Institut für Neutronenphysik und Reaktortechnik
Kernforschungszentrum Karlsruhe, Germany

```
procedure TAYLOR (n, m, x, h, f, itmax, eps1, eps2, der, S, KENN, EXIT);  
value n, m, eps1, eps2; integer n, m, itmax, KENN;  
real eps1, eps2, S;
```

Boolean *der*; **array** *x, h, f*; **label** *EXIT*;

comment

Let

$$S(x_1, \dots, x_n) = \sum_{i=1}^m f_i^2(x_1, \dots, x_n) \quad (m \geq n) \quad (1)$$

the function to be minimized. Such functions always appear if you apply the method of least squares to estimate nonlinear parameters. The following sequence

$$x^{(k+1)} = x^{(k)} - \beta \Delta x^{(k)} = x^{(k)} - \beta (F'_{x^{(k)}} F'_{x^{(k)}})^{-1} F'^T_{x^{(k)}} F(x^{(k)})$$

$$F = (f_1, \dots, f_m), \quad F'_x = \left(\frac{\partial f_i}{\partial x_j} \right)_{i=1, \dots, m, j=1, \dots, n} \quad (2)$$

where β , which is always possible, is chosen to be such that

$$S(x^{(k)} - \beta \Delta x^{(k)}) \leq (1 - \beta \lambda) S(x^{(k)}) \quad (0 < \lambda < 1) \quad (3)$$

is known to converge [1] for any $x^{(0)}$ to a stationary point of S ($\text{grad } S = 2F'^T_x F(x) = 0$), if on the carrying out of the iteration the matrix $F'^T_x F'_x$ does not become singular.

For $m = n$ you have $\Delta x = F'^{-1}_x F(x)$ and (2) becomes a damped version of Newton's method for solving the system of nonlinear equations

$$F(x) = 0 \quad (4)$$

All zeros of (4) are stationary points of (1). Thus we are able to generate a sequence which converges for any $x^{(0)}$ to a stationary point of (1) and the possible divergence of Newton's method ($\beta=1$) is avoided. It is not assured, however, that the method will always converge to a solution of (4). Numerical experience has shown that though Newton's method ($\beta=1$) diverges for a certain $x^{(0)}$ the damped sequence converges to a solution of (4) for the same $x^{(0)}$.

In the program we have chosen $\lambda = .2$. At each iteration we set first $\beta = 1$ and then, if (3) is not valid, $\beta = 2^{-j}$ ($j=1, 2, \dots, 16$). If j is greater than 16 then $\beta < .00002$ and we assume to have reached a stationary point of S .

Meaning of the formal parameters:

n the number of variables x_i
m the number of functions f_i
x the array $x[1:n]$ which must first contain a starting value $x^{(0)}$ and finally will contain a stationary point of S , if $F'^T_x F'_x$ or for $m = n$ F'_x , respectively, has not become singular
h $h[1:n]$ is a step size vector for the approximation of F'_x (see below)
f the array $f[1:m]$ will contain the function values at the last x calculated in *TAYLOR*
itmax must initially contain the maximum number of iterations to be performed. Leaving *TAYLOR* regularly, *itmax* contains the actual number of performed iterations
eps1 the iteration is stopped when $S < \text{eps1}$
eps2 the iteration is discontinued when $\sum_{i=1}^n |\Delta x_i^{(k)}| < \text{eps2} \times \sum_{i=1}^n |x_i^{(k+1)}|$
der if *der* = **true** the matrix F'_x must be produced by a global procedure named *DERIVE*(x, dfdx) which adjoins to the vector $x[1:n]$ the array $\text{dfdx}[1:m, 1:n]$. In this case the array *h* can be loaded by an arbitrary vector, for instance *x*.

if *der* = **false** the matrix F'_x is approximated by

$$\frac{\partial f_i}{\partial x_j} = \frac{f_i(x_1, \dots, x_j + h_j, \dots, x_n) - f_i(x_1, \dots, x_j - h_j, \dots, x_n)}{2h_j}$$

where *h* is a given step size vector. With a suitable choice of the h_j the convergence behavior of the sequence (2) is not destroyed. *DERIVE*(x, dfdx) must be formally declared outside of *TAYLOR* in this case.

[In some cases, particularly when solving nonlinear equations, the extra accuracy achieved by using central differences to estimate the derivatives is not necessary. A considerable saving in execution time can be obtained by using one-sided differences which means only minor changes in the program below. —REF.]

S should initially contain the greatest positive number that the employed computer can store. Finally *S* contains $S = S(x^{(\text{itmax})})$, if *TAYLOR* is regularly left.

KENN if after having called *TAYLOR*

KENN = 0 then one of the above interruptions applies (*eps1, eps2*),

KENN = 1 then *itmax* iterations were carried out and *TAYLOR* is left,

KENN = -1 then $\beta = 2^{-j}$ and *TAYLOR* is left.

EXIT TAYLOR goes to this global label if *i* encounters a singular matrix.

Further two global procedures must be made available to *TAYLOR*:

i) *FUNCTION*(x, f) which is able to calculate for a given vector $x[1:n]$ the function values $f[1:m]$

ii) *GAUSS*(n, A, b, x, EXIT) which solves the linear system of n equations $Ax = b$ for x . If *A* is singular then *GAUSS* returns to the global label *EXIT*. Any linear equation solver may be used for *GAUSS*;

```

begin integer i, j, k, z, l; real hf, hl, hs, hz;
array fp, fm[1:m], b, dx[1:n], dfdx[1:m, 1:n], aa[1:n, 1:n];
hs := S; KENN := z := 0;
ITERATION: z := z + 1;
if z > itmax then begin KENN := 1; go to ENDE end;
l := 0; hl := 1.0;
DAMP: l := l + 1;
if l > 16 then begin KENN := -1; go to ENDE end;
FUNCTION(x, f); hf := 0;
for i := 1 step 1 until m do hf := hf + f[i] × f[i];
if hf > hs × (1.0 - .2 × hl) then
begin hl := hl × .5;
for k := 1 step 1 until n do x[k] := x[k] + hl × dx[k];
go to DAMP
end;
hs := hf; if hs < eps1 then go to ENDE;
if der then DERIVE(x, dfdx) else
begin
for i := 1 step 1 until n do
begin hf := h[i]; hz := 2.0 × hf;
x[i] := x[i] + hf; FUNCTION(x, fp);
x[i] := x[i] - hz; FUNCTION(x, fm);
x[i] := x[i] + hf; hz := 1.0/hz;
for k := 1 step 1 until m do
dfdx[k, i] := hz × (fp[k] - fm[k])
end
end;
if m = n then GAUSS(n, dfdx, f, dx, EXIT) else
begin
for i := 1 step 1 until n do
begin hf := 0;
for k := 1 step 1 until m do
hf := hf + dfdx[k, i] × f[k]; b[i] := hf;
for k := i step 1 until n do
begin hf := 0;
for j := 1 step 1 until m do
hf := hf + dfdx[j, i] × dfdx[j, k];
aa[i, k] := aa[k, i] := hf
end
end;
GAUSS(n, aa, b, dx, EXIT)
end;

```

```

hz := hf := 0;
for i := 1 step 1 until n do
begin
  x[i] := x[i] - dx[i]; hz := hz + abs(x[i]);
  hf := hf + abs(dx[i])
end;
if hf ≥ eps2 × hz then go to ITERATION;
ENDE: FUNCTION(x, f); S := 0; itmax := z;
for i := 1 step 1 until m do S := S + f[i] × f[i]
end TAYLOR

```

REFERENCE:

[1] BRAESS, D. Über Dämpfung bei Minimalisierungsverfahren. *Computing 1* (1966), 264-272.

ALGORITHM 316
SOLUTION OF SIMULTANEOUS NON-LINEAR EQUATIONS [C5]

K. M. BROWN (Recd. 27 Oct. 1966, 31 Mar. 1967, 17 July 1967, and 26 July 1967)

Department of Computer Science, Cornell University, Ithaca, New York

procedure *nonlinearsystem* (*n*, *maxit*, *numsig*, *singular*, *x*);
value *n*, *numsig*; **integer** *n*, *maxit*, *numsig*, *singular*; **array** *x*;
comment This procedure solves a system of *n* simultaneous nonlinear equations. The method is roughly quadratically convergent and requires only $((n^2/2) + (3n/2))$ function evaluations per iterative step as compared with $(n^2 + n)$ evaluations for Newton's Method. This results in a savings of computational effort for sufficiently complicated functions. A detailed description of the general method and proof of convergence are included in [1]. Basically the technique consists in expanding the first equation in a Taylor series about the starting guess, retaining only linear terms, equating to zero and solving for one variable, say x_k , as a linear combination of the remaining $n - 1$ variables. In the second equation, x_k is eliminated by replacing it with its linear representation found above, and again the process of expanding through linear terms, equating to zero and solving for one variable in terms of the now remaining $n - 2$ variables is performed. One continues in this fashion, eliminating one variable per equation, until for the *n*th equation, we are left with one equation in one unknown. A single Newton step is now performed, followed by back-substitution in the triangularized linear system generated for the x_i 's. A pivoting effect is achieved by choosing for elimination at any step that variable having a partial derivative of largest absolute value. The pivoting is done without physical interchange of rows or columns.

The vector of initial guesses *x*, the number of significant digits desired *numsig*, the maximum number of iterations to be used, *maxit*, and the number of equations *n*, should be set up prior to the procedure call which activates *nonlinearsystem*. After execution of the procedure, the vector *x* is the solution of the system (or best approximation thereto), *maxit* is now the number of iterations used and *singular* = 0 is an indication that a Jacobian-related matrix was singular—indicative of the process "blowing-up," whereas *singular* = 1 is an indication that no such difficulty occurred. Storage space may be saved by implementing the algorithm in a way which takes advantage of the fact that the strict lower triangle of the array *pointer* and the same number of positions in the array *coe* are not used;

```

begin integer converge, m, j, k, i, jsub, itemp, kmax, kplus, tally;
real f, hold, h, fplus, dermax, test, factor, relconvg;
integer array pointer[1:n, 1:n], isub[1:n-1];
array temp, part[1:n], coe[1:n, 1:n+1];
procedure backsubstitution (k, n, x, isub, coe, pointer);
  value k, n;
  integer k, n; integer array isub, pointer; array x, coe;

```

```

comment This procedure back-solves a triangular linear system for improved  $x[i]$  values in terms of old ones;
begin integer km, kmax, jsub;
  for km := k step -1 until 2 do
  begin kmax := isub[km-1]; x[kmax] := 0;
    for j := km step 1 until n do
    begin jsub := pointer[km, j];
      x[kmax] := x[kmax] + coe[km-1, jsub] × x[jsub]
    end;
    x[kmax] := x[kmax] + coe[km-1, n+1]
  end;
end backsubstitution;
procedure evaluatekthfunction (x, y, k);
  integer k; real y; array x;
begin comment the body of this procedure must be provided by the user. One call of the procedure should cause the value of the kth function at the current value of the vector x to be placed in y;
end evaluatekthfunction;
converge := 1; singular := 1; relconvg := 10 ↑ (-numsig);
for m := 1 step 1 until maxit do
begin
  comment An intermediate output statement may be inserted at this point in the procedure to print the successive approximation vectors x generated by each complete iterative step;
  for j := 1 step 1 until n do pointer [1, j] := j;
  for k := 1 step 1 until n do
  begin if k > 1 then backsubstitution (k, n, x, isub, coe, pointer);
    evaluatekthfunction (x, f, k); factor := .001;
  AAA: tally := 0; for i := k step 1 until n do
    begin itemp := pointer[k, i]; hold := x[itemp];
      h := factor × hold; if h = 0 then h := .001;
      x[itemp] := hold + h;
      if k > 1 then backsubstitution (k, n, x, isub, coe, pointer);
      evaluatekthfunction (x, fplus, k);
      part[itemp] := (fplus - f)/h;
      x[itemp] := hold; if (abs(part[itemp]) = 0) ∨
        (abs(f/part[itemp]) > 1.01020) then tally := tally + 1;
    end;
    if tally ≤ n - k then go to AA; factor := factor × 10.0;
    if factor > .5 then go to SING; go to AAA;
  AA: if k < n then go to A; if abs (part[itemp]) = 0
  then go to SING;
  coe[k, n+1] := 0; kmax := itemp; go to ENDK;
  A: kmax := pointer[k, k]; dermax := abs(part[kmax]);
    kplus := k + 1;
    for i := kplus step 1 until n do
    begin jsub := pointer[k, i]; test := abs(part[jsub]);
      if test < dermax then go to B; dermax := test;
      pointer [kplus, i] := kmax; kmax := jsub;
    go to ENDI;
  B: pointer [kplus, i] := jsub;
  ENDI:
  end;
  if abs(part[kmax]) = 0 then go to SING; isub[k] := kmax;
  coe[k, n+1] := 0;
  for j := kplus step 1 until n do
  begin jsub := pointer[kplus, j];
    coe[k, jsub] := -part[jsub]/part[kmax];
    coe[k, n+1] := coe[k, n+1] + part[jsub] × x[jsub]
  end;
  ENDK:
  coe[k, n+1] := (coe[k, n+1] - f)/ part[kmax] + x[kmax]
end k;
x[kmax] := coe[n, n+1];
if n > 1 then backsubstitution (n, n, x, isub, coe, pointer);
if m = 1 then go to D;
for i := 1 step 1 until n do
  if abs((temp[i] - x[i])/x[i]) > relconvg then go to C;

```

```

converge := converge + 1;
if converge ≥ 3 then go to TERMINATE else go to D;
C: converge := 1;
D: for i := 1 step 1 until n do temp[i] := x[i]
end m;
go to THROUGH;
TERMINATE:
maxil := m; go to THROUGH;
SING:
singular := 0;
THROUGH:
end nonlinearssystem

```

APPENDIX

We include a sample procedure *evaluatekthfunction* for the 2×2 system:

$$\left(1 - \frac{1}{4\pi}\right) (e^{2x_1} - e) + \frac{e}{\pi} x_2 - 2ex_1 = 0$$

$$\frac{1}{2} \sin(x_1 x_2) - \frac{x_2}{4\pi} - \frac{x_1}{2} = 0,$$

one solution of which is $(.5, \pi)$ see [2]

```

procedure evaluatekthfunction (x, y, k);
integer k; real y; array x;
begin switch functionnumber := F1, F2;
go to functionnumber [k];
F1: y := 2.71828183 × (.920422528 × (exp(2×x[1]-1)-1)+
x[2]/3.14159265-2×x[1]);
go to RETURN;
F2: y := .5 × sin(x[1]×x[2]) - x[2]/12.5663706 - x[1]/2;
RETURN:
end evaluatekthfunction;

```

REFERENCES:

- BROWN, K. M. A quadratically convergent method for solving simultaneous non-linear equations. Doctoral Thesis, Dept. Computer Sciences, Purdue U., Lafayette, Ind., Aug., 1966.
- BROWN, K. M., AND CONTE, S. D. The solution of simultaneous nonlinear equations. Proc. ACM 22nd Nat. Conf., pp 111-114.

ALGORITHM 317*

PERMUTATION [G6]

CHARLES L. ROBINSON (Recd. 12 Apr. 1967, 2 May 1967 and 10 July 1967)

Institute for Computer Research, U. of Chicago, Chicago, Ill.

* This work was supported by AEC Contract no. AT (11-1)-614.

```

procedure permute(n, k, v); value n, k; integer array v;
integer n, k;
comment This procedure produces in the vector v the kth
permutation on n variables. When k = 0, v takes on the value
1, 2, 3, 4, ..., n. This algorithm is not as efficient as pre-
viously published algorithms [1], [2], [3] for generating a
complete set of permutations but it is significantly better
for generating a random permutation, a property useful in
certain simulation applications. Any non-negative value of
k will produce a valid permutation. To generate a random
permutation, k should be chosen from the uniform distribu-
tion over the integers from 0 to n! - 1 inclusive;
begin integer i, q, r, x, j;
for i := 1 step 1 until n do v[i] := 0;
for i := n step -1 until 1 do
begin
q := k ÷ i; r := k - q × i; x := 0; j := n;

```

```

no: if v[j] = 0 then
begin
if x = r then go to it else x := x + 1
end;
j := j - 1; go to no;
it: v[j] := i; k := q;
end
end

```

REFERENCES:

- COVEYOU, R. R., AND SULLIVAN, J. G. Algorithm 71, Permutation. *Comm. ACM* 4 (Nov. 1961), 497.
- PECK, J. E. L., AND SCHRACK, G. F. Algorithm 86, Permute. *Comm. ACM* 5 (Apr. 1962), 208.
- TROTTER, H. F. Algorithm 115, Perm. *Comm. ACM* 5 (Aug. 1962), 434.

Algorithms Policy • Revised August, 1966

A contribution to the Algorithms Department should be in the form of an algorithm, a certification, or a remark. Contributions should be sent in duplicate to the editor, typewritten double spaced. Authors should carefully follow the style of this department with especial attention to indentation and completeness of references.

An algorithm must normally be written in the ALGOL 60 Reference Language [*Comm. ACM* 6 (Jan. 1963), 1-17] or in ASA Standard FORTRAN or Basic FORTRAN [*Comm. ACM* 7 (Oct. 1964), 590-625]. Consideration will be given to algorithms written in other languages provided the language has been fully documented in the open literature and provided the author presents convincing arguments that his algorithm is best described in the chosen language and cannot be adequately described in either ALGOL 60 or FORTRAN.

An algorithm written in ALGOL 60 normally consists of a commented procedure declaration. It should be typewritten double spaced in capital and lower-case letters. Material to appear in **boldface** type should be underlined in black. Blue underlining may be used to indicate italic type, but this is usually best left to the Editor. An algorithm written in FORTRAN normally consists of a commented subprogram. It should be typewritten double spaced in the form normally used for FORTRAN or it should be in the form of a listing of a FORTRAN card deck together with a copy of the card deck. Each algorithm must be accompanied by a complete driver program in its language which generates test data, calls the procedure, and produces test answers. Moreover, selected previously obtained test answers should be given in comments in either the driver program or the algorithm. The driver program may be published with the algorithm if it would be of major assistance to a user.

For ALGOL 60 programs, input and output should be achieved by procedure statements, using any of the following eleven procedures (whose body is not specified in ALGOL) [See "Report on Input-Output Procedures for ALGOL 60," *Comm. ACM* 7 (Oct. 1964), 628-629]:

<i>in</i> symbol	<i>in</i> real	<i>ou</i> tarray	<i>in</i> teger
<i>ou</i> tsymbol	<i>ou</i> treal	<i>ou</i> tboolean	<i>ou</i> tinteger
<i>le</i> ngth	<i>in</i> array	<i>ou</i> tstring	

If only one channel is used by the program for output, it should be designated by 1 and similarly a single input channel should be designated by 2. Examples:

```

outstring (1, 'x='); outreal (1,x);
for i := 1 step 1 until n do outreal (1,A[i]);
integer (2, digit [17]);

```

For FORTRAN programs, input and output should be achieved as described in the ASA preliminary report on FORTRAN and Basic FORTRAN.

It is intended that each published algorithm be well organized, clearly commented, syntactically correct, and a substantial contribution to the literature of Algorithms. It is necessary but not sufficient that a published algorithm operate on some machine and give correct answers. It must also communicate a method to the reader in a clear and unambiguous manner. All contributions will be refereed both by human beings and by an appropriate compiler. Authors should pay considerable attention to the correctness of their programs, since referees cannot be expected to debug them.

Certifications and remarks should add new information to that already published. Readers are especially encouraged to test and certify previously uncertified algorithms. Rewritten versions of previously published algorithms will be refereed as new contributions and should not be imbedded in certifications or remarks.

Galley proofs will be sent to authors; obviously rapid and careful proof-reading is of paramount importance.

Although each algorithm has been tested by its author, no liability is assumed by the contributor, the editor, or the Association for Computing Machinery in connection therewith.

The reproduction of algorithms appearing in this department is explicitly permitted without any charge. When reproduction is for publication purposes, reference must be made to the algorithm author and to the *Communications* issue bearing the algorithm.—J.G.Herriot